Value Recursion in Monadic Computations

Levent Erkök
M.S. Computer Science, 1998, The University of Texas at Austin
B.S. Computer Engineering, 1994, Middle East Technical University

A dissertation presented to the faculty of the
OGI School of Science and Engineering
at Oregon Health and Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

October 2002

The dissertation "Value Recursion in Monadic Computations" by Levent Erkök has been examined and approved by the following Examination Committee:

Dr. John Launchbury
Professor
Thesis Research Advisor

Dr. Mark P. Jones
Associate Professor

Dr. Richard Kieburtz
Professor

Dr. David Maier
Professor

Dr. Ross Paterson
Lecturer, City University, London

Acknowledgements

I would like to thank my advisor, John Launchbury, for his guidance, help, and friendship over the last four years. I would have never completed this thesis without him leading the way. John knew exactly when to intervene and when to set me free, providing just the right balance. I always felt privileged to be working with him. Thank you John.

Many people contributed to this work. Ross Paterson, Mark Jones, Dick Kieburtz, and Dave Maier have all done an excellent job of providing me with invaluable feedback and guidance. In particular, Ross took an early interest in our work, helping out with the technical development over the years. Sava Krstić was a constant source of help, answering my endless questions with great enthusiasm. Nick Benton, Andrzej Filinski, and Amr Sabry answered many e-mail questions. I am grateful to Amr for constructively challenging our work, making us understand it better. Andy Moran collaborated with us in developing the semantics of value recursion for the IO monad. Jeff Lewis helped me understand the internals of Hugs, and helped with the implementation of the mdo-notation. Simon Peyton-Jones answered many questions over e-mail, and implemented the mdo-notation in GHC. The PacSoft research group at OGI provided a fun and stimulating environment at all times. I had many joyful conversations with Zine Benaissa, Magnus Carlsson, Iavor Diatchki, John Matthews, Thomas Nordin, Emir Pašalić, Walid Taha, and many other friends at OGI. My heartfelt thanks go to all of them.

I am indebted to numerous people for shaping up my thinking and expanding my horizons over the years, including Sadi Yalgın, Halit Oğuztüzün, Cem Bozşahin, Hamilton Richards, and Nicholas Asher. I thank them all.

My research was supported by grants from the Air Force Materiel Command (F19628-96-C-0161) and the National Science Foundation (CCR-9970980). I am thankful to our administrative staff for taking care of all the necessary details with meticulous care.

I would like to thank my parents, Fatma and Atila Erkök, for always being supportive, and trusting me with my decisions. I sincerely hope that I am worthy of their trust.

Finally, I would like to thank my wife Şengül Vurgun, for being on my side through thick and thin, taking over many of my responsibilities, and especially for believing in me when I did not. Thank you Şengül, I could not have done it without you.

Contents

Abstract

Value Recursion in Monadic Computations
Levent Erkök

Ph.D., OGI School of Science and Engineering,
Oregon Health and Science University
October 2002

Thesis Advisor: Dr. John Launchbury

This thesis addresses the interaction between recursive declarations and computational effects modeled by monads. More specifically, we present a framework for modeling cyclic definitions resulting from the values of monadic actions. We introduce the term *value recursion* to capture this kind of recursion.

Our model of value recursion relies on the existence of particular fixed-point operators for individual monads, whose behavior is axiomatized via a number of equational properties. These properties regulate the interaction between monadic effects and recursive computations, giving rise to a characterization of the required recursion operation. We present a collection of such operators for monads that are frequently used in functional programming, including those that model exceptions, non-determinism, input-output, and stateful computations.

In the context of the programming language Haskell, practical applications of value recursion give rise to the need for a new language construct, providing support for recursive monadic bindings. We discuss the design and implementation of an extension to Haskell's do-notation which allows variables to be bound recursively, eliminating the need for programming with explicit fixed-point operators.

Chapter 1

Introduction

This thesis addresses the interaction between two fundamental notions in programming languages: Recursion and effects. Recursion is the essence of cyclic definitions, both for recursive functions and circular data structures. Effects are the essence of computational features, including I/O, exceptions, and stateful computations. Although both notions have been studied extensively on their own, their interaction has received relatively little attention.

1.1 Recursion and effects

In the traditional domain theoretic setting, the denotational semantics of recursive definitions are understood in terms of fixed-points of continuous functions. That is, the semantics of a definition of the form $x = f\ x$ is taken to be the least fixed-point of the map corresponding to f [82, 83]. The same principle works for both recursive functions and circular data structures, a rather pleasing situation.

Handling of effects in the denotational framework, however, proved to be much more problematic, often summed up by the phrase "denotational semantics is not modular" [53, 64]. Briefly, addition of new effects require substantial changes to the existing semantic description. For instance, exceptions can be modeled by adding a special *failure* element to each domain, representing the result of a failed computation. But then, even such a simple thing as the meaning of an arithmetic operation requires a messy denotational description; one needs to check for failure at each argument, and propagate accordingly. The story is similar for other cases, including I/O and assignments, two of the most "popular" effects found in many programming languages [76, 77].

It was Moggi's influential work on monads that revolutionized the semantic treatment of effects, which he referred to as *notions of computation*. Moggi showed how monads can be used to model programming language features in a uniform way, providing an abstract view of programming languages [62, 63]. In the monadic framework, values of a given type are distinguished from computations that *yield* values of that type. Since the monadic structure hides the details of how computations are internally represented and composed, programmers and language designers work in a much more flexible environment. This flexibility is a huge win over the traditional approach, where everything has to be explicit.

Perhaps what Moggi did not quite envision was the response from the functional programming community, who took the idea to heart. Wadler wrote a series of articles showing how monads can be used in structuring functional programs themselves, not just the underlying semantics [89, 91]. Very quickly, the Haskell committee adopted monadic I/O as the standard means of performing input and output in Haskell, making monads an integral part of a modern programming language [68, 69]. The use of monads in Haskell is further encouraged by special syntactic support, known as the *do-notation* [47].

As the monadic programming style became more and more popular in Haskell, programmers started realizing certain shortcomings. For instance, function application becomes tedious in the presence of effects. Or, the *if-then-else* construct becomes unsightly when the test expression is monadic. However, these are mainly syntactic issues that can easily be worked around. More seriously, the monadic sublanguage lacks support for recursion over the *values* of monadic actions. The issue is not merely syntactic; it is simply not clear what a recursive definition means when the defining expression can perform monadic effects.

This problem brings us to the subject matter of the present work: Semantics of recursive declarations in monadic computations. More specifically, our aim is to study recursion resulting from the cyclic use of values in monadic actions. We use the term *value recursion* to describe this notion.

1.2 A motivating example: Modeling circuits using monads

To illustrate value recursion, we will consider the example that motivated our work in the first place: modeling circuits using monads. Microarchitectural design languages have been the target of programming language research in recent years, aiming at providing better language support for managing the complexity of such designs [12, 58]. Lava [8] and Hawk [49, 59] are two recent systems designed to address this need. In this section, we will consider a stripped down version of such a language, embedded in Haskell.

To familiarize ourselves with the types of circuits we can define, let us first consider a simple non-monadic implementation. We represent signals by lists, successive elements representing the values at each clock tick. Haskell is already expressive enough to define the basic building blocks without much difficulty:

```
type Sig α = [α]

and, xor   :: Sig Bool → Sig Bool → Sig Bool
and xs ys = zipWith (&&) xs ys
xor xs ys = zipWith (≠) xs ys

inv      :: Sig Bool → Sig Bool
inv xs = map not xs

delay           :: String → α → Sig α → Sig α
delay _ v xs = v : xs
```

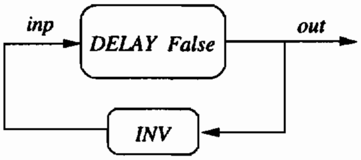
The *delay* element forms a signal that behaves as its second argument during the first clock cycle, behaving as its third argument afterwards. (The first argument to *delay* is intended to be a name for *v*. We will use it later.) Of course, a more realistic example would come equipped with multiplexers, registers, etc., but the elements above will be sufficient for our purposes. For instance, we can model a half-adder simply by:

```
halfAdd          :: Sig Bool → Sig Bool → (Sig Bool, Sig Bool)
halfAdd xs ys = (sum, carry)
      where sum   = xor xs ys
            carry = and xs ys
```

Here is a sample run:

```
Main> halfAdd [True, True] [False, True]
([True,False],[False,True])
```

As another example, we can create a circuit that toggles its output at each clock tick, starting from the value *False*:



```
toggle :: Sig Bool
toggle = out
   where inp = inv out
            out = delay "False" False inp
```

Variables *inp* and *out* are defined mutually recursively, corresponding to the feedback loop in the circuit diagram. The recursive definition capability of Haskell's **where** clause plays a crucial role in expressing the required cyclic dependency. We have:

```
Main> toggle
[False,True,False,True,False,True,False,True,...
```

Note that the result is an infinite signal.

What can we do with circuit descriptions? Since we model circuits by functions, we can pass them around and combine them to build bigger circuits. But, eventually, all we can do with a circuit is to simulate it, that is, run it on a particular input. As pointed out by Launchbury et al. [49], and Claessen [12], this model does not allow for multiple interpretations. Ideally, we would like to be able to analyze our circuits, translating them to other hardware description languages such as VHDL. Alternatively, we may want to render the circuit graphically, obtaining a schematic diagram, or recast the circuit description in the language of a theorem prover to let us reason about it. We would like our language to be flexible enough to support all of these views.

The standard way of attacking this problem is to abstract away from any particular signal or circuit model, hiding the control flow behind a monad, and basic circuit elements behind a type class. Each alternative semantics will be represented as an instance of this class, providing new views of circuits. Then, by simply switching to a different monad, we will be able to obtain an alternative interpretation *without* changing existing circuit descriptions. Here is one way of capturing the required structure:[1]

```
class Monad m ⇒ Circuit m where
  and, xor :: Sig Bool → Sig Bool → m (Sig Bool)
  inv      :: Sig Bool → m (Sig Bool)
  delay    :: String → α → Sig α → m (Sig α)
```

For instance, the description of the half-adder becomes:

```
halfAdd       :: Circuit m ⇒ Sig Bool → Sig Bool → m (Sig Bool, Sig Bool)
halfAdd i1 i2 = do sum   ← xor i1 i2
                   carry ← and i1 i2
                   return (sum, carry)
```

Note that the new model of *halfAdd* is not committed to any particular circuit model, or signal data type. It is a generic description of half-adders. To simulate, all we need is the identity monad for expressing the control structure, and the list model for signals:

```
type Sig α        = [α]
data Simulate α = Sim α  deriving Show

instance Monad Simulate where
  return x      = Sim x
  Sim x ≫= f = f x
```

Unsurprisingly, the *Circuit* instance for the *Simulate* monad will simply mimic our non-monadic implementation:

[1] A better alternative would be parameterizing the *Circuit* class over the *Sig* type as well, using a multiparameter type class. We refrain from doing so, however, for the sake of simplicity.

```
instance Circuit Simulate where
  and xs ys     = return (zipWith (&&) xs ys)
  xor xs ys     = return (zipWith (≠) xs ys)
  inv xs        = return (map not xs)
  delay _ v xs  = return (v:xs)
```

Using this model, we have:

```
Main> halfAdd [True, True] [False, True] :: Simulate ([Bool], [Bool])
Sim ([True,False],[False,True])
```

More interestingly, we can consider an alternative semantics which will create a wire-by-wire description of a given circuit. In this model, signals will be identified by symbolic names. Our monad will have to generate new names for intermediate wires, accumulating a textual "drawing" of the circuit as it is built. Hence, we employ a combination of state and output monads:

```
type Sig    α = String
data Draw α = D (Int → (α, [String], Int))

instance Show α ⇒ Show (Draw α) where
  show (D f) = let (l, s, _) = f 0
               in concatMap (++"\n") s ++ "Result: " ++ show l

instance Monad Draw where
  return x    = D (λi. (x, [], i))
  D f ≫= g   = D (λi. let (a, o, i')   = f i
                            D h          = g a
                            (b, o', i'') = h i'
                        in (b, o ++ o', i''))
```

We will need the following auxiliary functions:

```
newWire :: Draw String
newWire = D (λi. ('w':show i, [], i+1))

output   :: String → Draw ()
output s = D (λi. ((), [s], i))

item       :: String → String → String → Draw String
item a b c = do n ← newWire
                output (n ++ " = " ++ a ++ " " ++ b ++ " " ++ c)
                return n
```

The function *newWire* simply returns a new name. (The variable *i* keeps track of the number of wires.) The function *output* lets us emit intermediate descriptions. Finally, *item* is a generic function for creating a new wire together with a description of how it is obtained. Using these auxiliaries, the *Circuit* instance for the *Draw* monad becomes:[2]

```
instance Circuit Draw where
  and a b     = item “and” a b
  xor a b     = item “xor” a b
  delay s v a = item “delay” s a
  inv a       = do n ← newWire
                   output (n ++ “ = inv ” ++ a)
                   return n
```

We have:

```
Main> print (halfAdd "a" "b" :: Draw (Sig Bool, Sig Bool))
w0 = xor a b
w1 = and a b
Result: ("w0","w1")
```

It is worth emphasizing that the description of *halfAdd* did not change, we simply used a different monad. This is the strength of the monadic approach.

Unfortunately, a similar translation for the toggle circuit does not work. Consider:

```
toggle :: Circuit m ⇒ m (Sig Bool)
toggle = do inp ← inv out
            out ← delay “False” False inp
            return out
```

Although the description perfectly fits the circuit diagram we had before, we have lost the feedback loop. The variables *inp* and *out* are no longer recursively defined! (In fact, the definition above is not even valid Haskell; the variable *out* is not in scope in the first line.) Our non-monadic implementation did not have this problem, as it relied on the recursive definition capabilities of Haskell. But now, we are on our own: Haskell does not let us write recursive specifications in the presence of monadic effects.

Unfortunately, the problem is not merely syntactic. It is not clear how to perform this kind of recursion at all: we want the values (i.e., the signals) to be defined recursively, but we certainly do not want the effects to be repeated or lost (i.e., we do not want to create circuit elements repeatedly, or not to create them at all). We refer to this kind of recursion as *value recursion*. In short, to be able to express the required recursive structure, we need the underlying monad to support *recursive monadic bindings* [18]. Just as the usual fixed-point operator handles “normal” recursion, we expect to find *value recursion operators*,

[2] The *delay* element did not use its first argument in the simulation model, and here it does not use the second. The name is irrelevant for simulation, while it is all we need in a textual representation.

generically called *mfix*, mediating the interaction between the underlying effect and the recursion operation.

Getting back to circuit modeling, we will require circuits to be modeled by monads for which such fixed-point operators are available, captured by the *MonadFix* class:

```
class Monad m ⇒ MonadFix m where
  mfix :: (α → m α) → m α

class MonadFix m ⇒ Circuit m where
  -- and, xor, inv, delay as before
```

Now, we can tie the recursive knot over *inp* and *out*, expressing *toggle* as follows:

```
toggle :: Circuit m ⇒ m (Sig Bool)
toggle = mfix (λ ~(inp, out). do inp ← inv out
                                 out ← delay "False" False inp
                                 return (inp, out))
         ≫= λ(inp, out). return out
```

The final missing piece is the *MonadFix* instances for *Simulate* and *Draw* monads. At this point, we ask the reader to simply accept the following definitions:

```
instance MonadFix Simulate where
  mfix f = Sim (let Sim a = f a in a)

instance MonadFix Draw where
  mfix f = D (λi. let D g        = f a
                      (a, s, i') = g i
                  in (a, s, i'))
```

Note that the *Simulate* instance is essentially the same as the usual fixed-point operator. The *Draw* instance is a bit more complicated, but the reader can see that we perform the fixed-point computation over the variable *a*, (i.e., the value), passing around *i* and *s* untouched. Now, to simulate *toggle*, we just use our *Simulate* monad:

```
Main> toggle :: Simulate (Sig Bool)
Sim [False,True,False,True,False,True,False,...
```

and, to get a simple textual drawing, we simply switch to the *Draw* monad:

```
Main> toggle :: Draw (Sig Bool)
w0 = inv w1
w1 = delay False w0
Result: "w1"
```

The handling of recursion via *mfix* is somewhat mysterious at this point. The whole point of this thesis is to expose the mystery, and to explore the interaction between recursion and effects, heading toward an equational theory of value recursion.

1.3 Recursive monadic bindings

The use of *mfix* to tie the recursive knot in a monadic computation is similar to the handling of recursive bindings in usual let-expressions. For clarity, we will use the keyword **letrec** here when a binding can be recursive, and **let** otherwise. In the pure world, we have:

$$\begin{aligned} & \textbf{letrec } x = e \textbf{ in } e' \\ \equiv\ & \textbf{let } x = \mathit{fix}\ (\lambda x.\ e) \textbf{ in } e' \\ \equiv\ & (\lambda x.\ e')\ (\mathit{fix}\ (\lambda x.\ e)) \end{aligned}$$

What happens in a monadic computation? Similar to **letrec**, let us use the keyword **mdo**[3] for monadic bindings that can be recursive, and **do** otherwise. We have:

$$\begin{aligned} & \textbf{mdo } \{\ x \leftarrow e;\ e'\ \} \\ \equiv\ & \textbf{do } \{\ x \leftarrow \mathit{mfix}\ (\lambda x.\ e);\ e'\ \} \\ \equiv\ & \mathit{mfix}\ (\lambda x.\ e) \gg\!= \ \lambda x.\ e' \end{aligned}$$

In Chapter 7, we will describe an extension to the do-notation of Haskell allowing bindings to be recursive, using an enhanced version of this translation. Then, we will be able to write the *toggle* example of the previous section as follows, the compiler taking care of the insertion of appropriate calls to *mfix*:

```
toggle = mdo inp ← inv out
             out ← delay "False" False inp
             return out
```

There is an opportunity here to clarify a potentially confusing issue about value recursion. Consider a recursive definition of the form:

```
countDown n = if n == 0
              then print "Done!"
              else do print n
                      countDown (n−1)
```

The intention is clear: Each time *countDown* is called, we want the effect of printing to take place. In this thesis, we will not be dealing with such definitions, as they are already explained in terms of the usual fixed-point construction:

```
countDown = fix (λf. λn. if n == 0
                         then print "Done!"
                         else do print n
                                 f (n−1))
```

[3] The closest we can get to μ**do** using ASCII. (We would have used **dorec**, but that is just too long.) Note that the use of Haskell-like syntax is just for convenience. We could have used Moggi's $\textbf{let}_T\ x \Leftarrow e \textbf{ in } e'$ notation and the keyword $\textbf{letrec}_T$ as well [63].

Note that effects are part of *countDown*'s execution, rather than its definition. That is, the effect of printing is *not* performed to determine the meaning of *countDown* itself. In the *toggle* example, however, we see that effects are part of the definition: They are performed in order to determine the values of *inp* and *out*, and the cyclic dependence gives rise to the need for value recursion. In a sense, the use of recursion and effects in *countDown* are orthogonal, with no interference in between. As shown above, this kind of recursion is already explained in terms of *fix*, the usual fixed-point operator.

1.4 A generic *mfix*?

In Section 1.2, we saw two particular examples of *mfix*, one for the *Simulation* monad, and another one for the *Draw* monad. Are these functions actually instances of a generic schema? That is, can we find a definition of *mfix* that will work for all monads, regardless of which kind of effect we deal with? Let us pause briefly and consider how one might go about defining such a generic operator.

Recall that the least fixed-point operator on domains satisfies the property:

$$\begin{aligned} &\mathit{fix} && :: (\alpha \to \alpha) \to \alpha \\ &\mathit{fix}\ f && = f\ (\mathit{fix}\ f) \end{aligned}$$

which also serves as a definition for *fix* in a lazy language such as Haskell. One might think that a similar defining equation can be found for *mfix* as well. Indeed, it is not hard to generalize to the monadic case:

$$\begin{aligned} &\mathit{mfix} && :: \mathit{Monad}\ m \Rightarrow (\alpha \to m\ \alpha) \to m\ \alpha \\ &\mathit{mfix}\ f && = \mathit{mfix}\ f \gg\!= f \end{aligned}$$

Note that this definition makes sense for all monads (i.e., it is polymorphic in m). But is it a "good" definition? That is, can we use it sensibly to implement value recursion?

The short answer to this question is, unfortunately, no. To see why not, simply note that this definition is equivalent to:

$$\mathit{mfix}\ f = \mathit{fix}\ (\lambda m.\ m \gg\!= f) = \bigsqcup \{\bot,\ \bot \gg\!= f,\ \bot \gg\!= f \gg\!= f,\ \ldots\}$$

which will diverge whenever the $\gg\!=$ operator is strict in its first argument.[4] Furthermore, even when $\gg\!=$ is not strict, this definition will attempt to compute the fixed-point over

[4] Note that the call to *mfix f* will diverge regardless of what f is. In general, monads based on sum types will suffer from this problem, as the $\gg\!=$ operator needs to inspect its first argument to see how to proceed. Haskell's *maybe* and *list* monads are two popular examples that are based on sum types. Other important examples where the $\gg\!=$ operator is strict in its first argument include the frequently used IO and strict state monads.

both values and effects, which is simply not what we are trying to achieve. In value recursion, we want the fixed-point to be computed only over the values, without repeating or losing the effects. We will codify what we mean by value recursion in Chapter 2 using a number of equational properties, exploring the interaction between recursion and effects in depth. Then, we will be able to see more clearly why this default definition is not appropriate for implementing value recursion.

1.5 The basic framework and notation

For most of this thesis we investigate value recursion in the usual domain theoretic semantics of programming languages, where types are modeled by domains [77, 82]. We write $\bot_\tau$ for the least element of the domain representing the type τ, dropping the subscript whenever unambiguous. Functions are modeled by continuous (and hence monotonic) maps between domains, not necessarily strict. Recursion is modeled via least-fixed points. We use monads to model effects, following Moggi [63]. Although by no means comprehensive, the reader may find it useful to skim over Appendix A, which contains a brief review of fixed-point operators.

We expect readers to be familiar with functional programming [35, 87], particularly Haskell [7, 68]. For the most part, we use Haskell simply as a syntactically beefed up version of λ-calculus [30], so familiarity with any functional language should be sufficient. A basic understanding of domain theoretic semantics of programming languages is necessary to follow the technical development [76, 83]. Except for Chapter 6, we will be mainly interested in the "functional programming view" of monads [4, 91], rather than the categorical one [2, 55]. Finally, we will have occasion to use the parametricity principle, allowing us to derive theorems from the types of polymorphic functions [50, 75, 88].

Naturally, the theory of value recursion is independent of any particular programming language. However, our work is closely tied to Haskell, and we will be careful in pointing out the cases when the domain theoretic semantics and the semantics of Haskell do not quite match up. The main differences show up in the treatment of products. Since tuples are lifted in Haskell, it is *not* the case that $(\bot_\alpha, \bot_\beta) = \bot_{\alpha\times\beta}$. Therefore, the equality $x = (\pi_1\ x,\ \pi_2\ x)$ fails. Similarly, $\lambda(x :: \alpha).\bot_\beta \neq \bot_{\alpha\to\beta}$, i.e., the function type is lifted too. Similar comments apply to sum types as well. Finally, the unit data type has two members, $\bot_{()}$ and $()$ itself, that is, it is not really a terminal object. Luckily, these differences do not cause much trouble in practice, as long as one is aware of them. We point out the cases where the difference becomes significant.

In our exposition, we will stick to Haskell notation as much as possible, deviating from it only for typographical purposes. The difference mainly shows up in compositions and λ-bindings. For instance, we will write Haskell's: `\f -> \g -> \x -> (f . g) x` as $\lambda f.\ \lambda g.\ \lambda x.\ (f\ \cdot\ g)\ x$.

1.6 Outline of the thesis

Our aim is to get through the basics of value recursion rather quickly, before we actually investigate individual instances. To this end, we use the next two chapters to introduce a number of equational properties that govern the behavior of value recursion operators. Among these, we will identify three fundamental properties (namely strictness, purity, and left shrinking), and in the remainder of the thesis we will consider only those operators that satisfy this minimal core.

Chapters 4 and 5 are dedicated to the study of individual instances. In Chapter 4, we investigate a wide range of monads that are frequently used in functional programming, presenting value recursion operators for them. In Chapter 5, we argue that it is highly unlikely that the continuation monad has an associated value recursion operator that will satisfy our requirements.

Chapter 6 takes a step back and looks at a possible categorical theory of value recursion, based on the notion of premonoidal categories and traces. Even though the theory of traces does not provide a perfect fit, it is illuminating to see how recent work in this area can be generalized to capture value recursion for a certain class of monads.

Chapters 7 and 8 deal with the Haskell language in particular. In Chapter 7, we will turn our attention to syntactic support for value recursion, presenting a recursive version of Haskell's do-notation. In Chapter 8, we will study Haskell's IO monad. Since the IO monad is hardwired into Haskell, it is not possible to investigate value recursion for it directly. Hence, we present a model language (complete with I/O operations and mutable variables), and show how one can model value recursion in this world.

Chapter 9 presents a number of examples, which, in addition to the circuit modeling example of this chapter, provides a tour of potential applications of value recursion.

Chapter 10 concludes the thesis with a discussion of related work and future research directions. A brief review of fixed-points, along with several proofs that are omitted from the main body of the thesis are given in the appendices.

Each chapter in the remainder of this thesis starts with a brief description of its contents. Although we intend the chapters to be read in order, readers may find it useful to quickly skim over these segments to determine a particular reading plan according to their own interests.

Chapter 2

Properties of value recursion operators

What kinds of properties do we expect value recursion operators to satisfy? So far, we have been using phrases like *"recursion without repeating or losing effects,"* or *"recursion only over the values"* to characterize value recursion. The aim of this chapter is to formalize our intuitions by means of equational properties.

Synopsis. We discuss a number of equivalences that we expect value recursion operators to satisfy. These properties range from those that imitate properties of the usual fixed-point operator over domains, to those that govern the interaction between recursion and effects. We also provide a number of derived properties, including those that are granted by virtue of parametricity. Several properties that might be naively expected, yet unsatisfiable for a wide range of monads, are discussed as well.

2.1 Strictness (Nothing from nothing)

The domain theoretic treatment of recursion in programming languages relies on least fixed-points [76, 83]. That is, given a specification of the form $x = f\ x$, where $f :: \alpha \to \alpha$, we expect x to be the least α satisfying this equation. In this setting, one can show that a function is strict if and only if its least fixed-point is $\bot$. Since $\bot$ represents *no information* in the domain theoretic ordering, our slogan in this case is simply *nothing from nothing.* Generalizing to value recursion, we expect the following property to hold:

Property 2.1.1 (*Strictness.*) Let $f :: \alpha \to m\ \alpha$,

$$f\ \bot_\alpha = \bot_{m\ \alpha} \quad \Leftrightarrow \quad \mathit{mfix}\ f\ = \bot_{m\ \alpha} \tag{2.1}$$

Remark 2.1.2 In Section 2.6.2, we will be able to derive the right to left implication from other properties, i.e., we will show that if *mfix* f is $\bot$, then f must be strict. We prefer expressing the strictness law as it is, however, as it uniquely characterizes strict functions of type $\alpha \to m\ \alpha$.

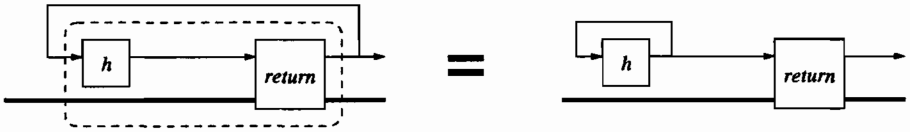
2.2 Purity (Just like *fix*)

Purity formalizes the intuition that *mfix* should behave exactly like *fix*, in case there are no effects:

Property 2.2.1 (*Purity.*) Let $h :: \alpha \to \alpha$,

$$mfix\ (return \cdot h) = return\ (fix\ h) \tag{2.2}$$

Diagrammatically, we capture purity as follows:



Remark 2.2.2 We use wiring diagrams to capture properties pictorially. Note that we do not formalize these diagrams, nor use them for any purpose other than illustration. Dashed boxes represent where value recursion is performed. Thin lines show data flow. The thick line, called the *effect line*, refers to the details of the monadic computation. Although it is not correct to consider the effect line as carrying data, it usually helps to think of it as such. (The effect line analogy holds very well for the state monad, but it is not very intuitive for, say, the exception monad.) We indicate pure computations by not letting them use the effect line, as illustrated by the *h* box in the above diagram. The solid loop on the right hand side indicates the use of *fix*. (Note that there are no dashed boxes on the right hand side as there are no applications of *mfix*.)

2.3 Left shrinking (No recursion – No *fix*)

Recall our naive translation schema for the recursive do-notation from Section 1.3. Naturally, we would like **mdo** to behave exactly like **do**, provided there are no recursive bindings. That is, the following two code fragments should have the same meaning:

$$\begin{array}{lll} \textbf{mdo}\ x \leftarrow A & \qquad\qquad & \textbf{do}\ x \leftarrow A \\ \qquad B & & \qquad \textbf{mdo}\ B \end{array}$$

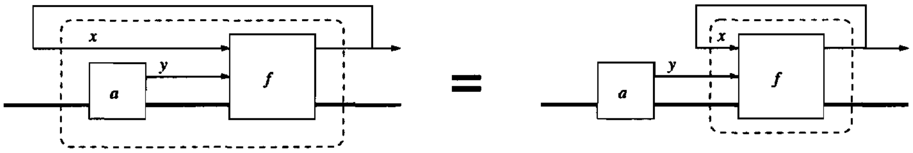
provided A does not make use of x, or any variable defined in the block B. If B does not have any recursive bindings either, we can push **mdo** further down, eventually eliminating it altogether. We capture this correspondence by the left shrinking property:

Property 2.3.1 (*Left shrinking.*) Let $f :: \alpha \to \beta \to m\ \alpha$, $a :: m\ \beta$,

$$mfix\ (\lambda x.\ a \gg\!= \lambda y.\ f\ x\ y) = a \gg\!= \lambda y.\ mfix\ (\lambda x.\ f\ x\ y) \tag{2.3}$$

where x does not occur free in a.

The name "left shrinking" is suggested by the corresponding diagram:



Remark 2.3.2 The reader might expect an analogous right shrinking property as well. But, as we will see in Chapter 3, arbitrary lifting of computations from the right hand side of a $\gg\!=$ is not possible in general. We can, however, lift pure computations out. We will provide a derived law to deal with this case in Section 2.6.3.

2.4 Sliding (Pure mobility)

Let $f :: \alpha \to \beta$ and $h :: \beta \to \alpha$. As reviewed in Appendix A, the equation

$$fix\ (h \cdot f) = h\ (fix\ (f \cdot h)) \tag{2.4}$$

expresses the dinaturality condition for *fix*, an extremely important law for manipulating fixed-points. We expect value recursion operators to satisfy a similar law as well.

Two problems arise in translating Equation 2.4 to the world of value recursion. The order of f and h is swapped, and h is duplicated on the right hand side. Obviously, if f and h can both perform effects, swapping and duplication are both out of question. When h is pure, however, we expect to be able to slide it over f:

Property 2.4.1 (*Sliding.*) Let $f :: \alpha \to m\ \beta$, $h :: \beta \to \alpha$,

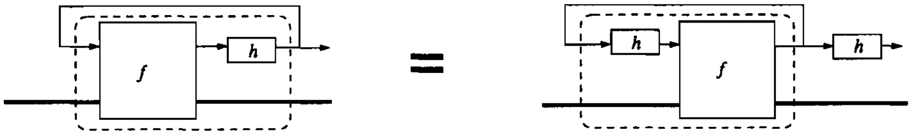
$$f\ (h\ \bot) = f\ \bot \ \Rightarrow\ mfix\ (map\ h \cdot f) = map\ h\ (mfix\ (f \cdot h)) \tag{2.5}$$

where $map :: (a \to b) \to m\ a \to m\ b$ is the usual lifting function.[1] The consequent can be equivalently expressed as:

$$mfix\ (\lambda x.\ f\ x \gg\!= return \cdot h) = mfix\ (f \cdot h) \gg\!= return \cdot h \tag{2.6}$$

[1] The function *map* is defined by the equation $map\ f\ m = m \gg\!= return \cdot f$. Note that, in Haskell notation *map* is called *fmap*, and the name *map* is reserved to be used with the *list* monad only [68]. Deviating from Haskell, we use the name *map* consistently for all monads.

Diagrammatically:



The side condition, i.e., $f \cdot h$ and f should agree on $\bot$, is essential. When we think of recursion as an iterative process that starts with $\bot$, we see that f first receives $\bot$ on the left hand side in the recursive loop, but receives $h\ \bot$ on the right. If $h\ \bot \neq \bot$, f will have more information to start with on the right hand side. The side condition guarantees that this extra knowledge is irrelevant: f must not distinguish between $\bot$ and $h\ \bot$. It is worth noting that dinaturality of *fix* (Equation 2.4) does not require any such conditions. As we will see in Chapter 3, however, without the side condition, sliding is unsatisfiable for many practical monads of interest.

Observation 2.4.2 The side condition is trivially satisfied if h is strict. It turns out that this particular case is derivable from parametricity (see Corollary 2.6.12).

Note The alert reader will note that the order of effects does not matter for commutative monads, and hence one might expect a swapping property where both computations are effectful. This is indeed the case, see Section 3.3 for details.

2.5 Nesting (Two for the price of one)

Bekič's property for usual fixed-points states that simultaneous recursion over multiple variables is equivalent to recursion over one variable at a time (see Appendix A.) In the value recursion world, one way to express this relation is to assert the equivalence of the following two expressions:

```
mdo x ← mdo y ← f (x, y)
            return y                    mdo x ← f (x, x)
    return x                                return x
```

The nesting property[2] stipulates this equivalence:

Property 2.5.1 (*Nesting.*) Let $f :: (\alpha, \alpha) \to m\ \alpha$,

$$mfix\ (\lambda x.\ mfix\ (\lambda y.\ f\ (x,\ y))) = mfix\ (\lambda x.\ f\ (x,\ x)) \tag{2.7}$$

[2] This property was first suggested to us by Ross Paterson (personal communication).

The following proposition states an equivalent form of nesting, which is quite useful in symbolic manipulations:

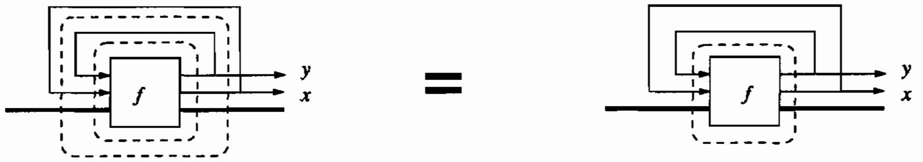
Proposition 2.5.2 Let $f :: (\tau, \sigma) \rightarrow m\ (\tau, \sigma)$. Assuming true products, the equation

$$\mathit{mfix}\ (\lambda(x,\ \_).\ \mathit{mfix}\ (\lambda(\_,\ y).\ f\ (x,\ y))) = \mathit{mfix}\ f \tag{2.8}$$

is satisfied exactly when nesting holds, provided *mfix* satisfies the sliding property.

Proof See Appendix B.1. □

Using Equation 2.8, it is easy to describe nesting diagrammatically:



Just like the Bekič property for *fix*, nesting generalizes to any number of variables. For instance, one can derive:

$$\begin{aligned}&\mathit{mfix}\ (\lambda(x,\ \_,\ \_).\ \mathit{mfix}\ (\lambda(\_,\ y,\ \_).\ \mathit{mfix}\ (\lambda(\_,\ \_,\ z).\ f\ (x,\ y,\ z))))\\ &\quad= \mathit{mfix}\ (\lambda(x,\ y,\ z).\ f\ (x,\ y,\ z))\end{aligned} \tag{2.9}$$

Note that the order of nesting is also immaterial, we could have recursed over any permutation of the variables; for instance, first over z, then x and finally y, etc.

Remark 2.5.3 We will take a closer look at Equation 2.8 in the case of lifted products (as in Haskell). Assuming *mfix* satisfies strictness, the left hand side will always be $\perp$, due to strict matching against pairs. Using irrefutable patterns, one might attempt:

$$\mathit{mfix}\ (\lambda\tilde{}(x,\ \_).\ \mathit{mfix}\ (\lambda\tilde{}(\_,\ y),\ f\ (x,\ y))) = \mathit{mfix}\ f$$

However, a problem still remains. If f is strict, then the right hand side will be $\perp$, but the left hand side might produce an answer, because $\perp \neq (\perp, \perp)$.[3] The proper way of expressing Equation 2.8 with lifted products is:

$$\mathit{mfix}\ (\lambda\tilde{}(x,\ \_).\ \mathit{mfix}\ (\lambda\tilde{}(\_,\ y),\ f\ (x,\ y))) = \mathit{mfix}\ (\lambda\tilde{}(x,\ y).\ f\ (x,\ y)) \tag{2.10}$$

Similar to Proposition 2.5.2, one can establish:

Proposition 2.5.4 In the case of lifted products, Equation 2.7 is equivalent to Equation 2.10, provided *mfix* satisfies sliding. □

[3] Peter Thiemann was first to notice this problem (personal communication).

2.6 Derived properties

One can derive new equalities using the properties we have described so far, and properties of the underlying domain-theoretic framework. This section presents a collection of such laws—those that we have found to be the most useful when reasoning about value recursion.

2.6.1 Constant functions

Left shrinking and purity properties imply an expected property of fixed-point operators: If the fixed-point variable is not used, recursion is irrelevant:

Proposition 2.6.1 Let $a :: m\ \alpha$ be a constant (i.e., x does not occur free in a). Then,

$$mfix\ (\lambda x.a)\ =\ a \tag{2.11}$$

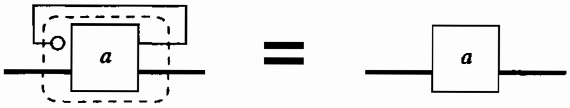
provided *mfix* satisfies purity and left shrinking laws.

Proof

$$\begin{aligned} mfix\ (\lambda x.\ a) &= mfix\ (\lambda x.\ a \gg\!= \ \lambda y.\ return\ y) & \\ &= a \gg\!= \lambda y.\ mfix\ (\lambda x.\ return\ y) & \{left\ shrinking\} \\ &= a \gg\!= \lambda y.\ return\ (fix\ (\lambda x.\ y)) & \{purity\} \\ &= a \gg\!= \lambda y.\ return\ y & \\ &= a & \end{aligned}$$

Note that $fix\ (\lambda x.\ y) = (\lambda x.\ y)\ (fix\ (\lambda x.\ y)) = y$. □

The diagram in this case is trivial:



Similarly, we can lift a conditional expression from inside an *mfix*, if the test expression is not involved in the recursion computation:

Proposition 2.6.2 Let a be a boolean expression where x does not occur free in a. Let f, $g :: \alpha \to m\ \alpha$. We have:

$$mfix\ (\lambda x.\ \textbf{if}\ a\ \textbf{then}\ f\ x\ \textbf{else}\ g\ x) = \textbf{if}\ a\ \textbf{then}\ mfix\ f\ \textbf{else}\ mfix\ g \tag{2.12}$$

Proof Case analysis on the value of a. The *True* and *False* cases are obvious. When $a = \bot$, the left hand side yields $\bot$ by Proposition 2.6.1, guaranteeing the equivalence. □

2.6.2 Approximation property

Monotonicity implies that $f\ \bot$ always provides an approximation to $\mathit{mfix}\, f$:

Proposition 2.6.3 Let $f :: \alpha \to m\ \alpha$. Then,

$$f\ \bot \sqsubseteq \mathit{mfix}\ f$$

provided *mfix* satisfies purity and left shrinking.

Proof Since $(\lambda x.\ f\ \bot) \sqsubseteq (\lambda x.\ f\ x)$, we have $\mathit{mfix}\ (\lambda x.\ f\ \bot) \sqsubseteq \mathit{mfix}\ f$ by the monotonicity of *mfix*. But the left hand side is $f\ \bot$ by Proposition 2.6.1, completing the proof. □

Remark 2.6.4 Proposition 2.6.3 states more than a rudimentary fact: $f\ \bot$ yields valuable information on the structure of the fixed-point. Consider the list monad, for instance. If $f :: a \to [a]$, and if $f\ \bot$ is a cons-cell, then so is $\mathit{mfix}\ f$. In particular, if $f\ \bot$ is a finite list of length k, then the length of the fixed-point is k as well. In general, for any monad based on a sum type, $f\ \bot$ determines the top level structure of *mfix f*.

We can now establish the strictness property in one direction (see Remark 2.1.2):

Corollary 2.6.5 Let $f :: \alpha \to m\ \alpha$, and $\mathit{mfix}\ f = \bot$. Then f is strict, provided *mfix* satisfies purity and left shrinking laws.

Proof By Proposition 2.6.3, $f\ \bot \sqsubseteq \bot$, implying that $f\ \bot = \bot$. □

2.6.3 Pure right shrinking

The sliding property allows lifting of pure computations from the right hand side of a $\gg\!=$:

Corollary 2.6.6 Let $f :: \alpha \to m\ \alpha$, and $h :: \alpha \to \beta$,

$$\begin{aligned} &\mathit{mfix}\ (\lambda(x,\ y).\ f\ x \gg\!= \ \lambda z.\ \mathit{return}\ (z,\ h\ z)) \\ &\quad = \ \mathit{mfix}\ f \gg\!= \ \lambda z.\ \mathit{return}\ (z,\ h\ z) \end{aligned} \tag{2.13}$$

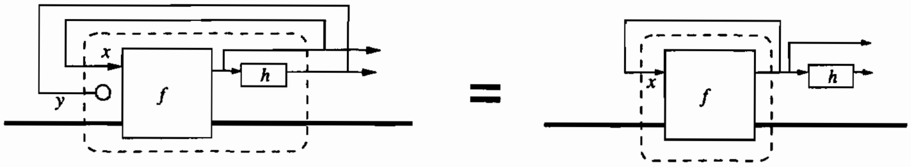
provided *mfix* satisfies sliding. (On the left hand side, the value-recursion loop is over (α, β), while the one on the right hand side it is over α only.)

Proof We have

$$\begin{aligned} &\mathit{mfix}\ (\lambda(x,\ y).\ f\ x \gg\!= \ \lambda z.\ \mathit{return}\ (z,\ h\ z)) \\ = \ &\mathit{mfix}\ (\mathit{map}\ (\lambda z.\ (z,\ h\ z)) \cdot f \cdot \pi_1) && \{\mathit{slide}\} \\ = \ &\mathit{map}\ (\lambda z.\ (z,\ h\ z))\ (\mathit{mfix}\ (f \cdot \pi_1 \cdot (\lambda z.\ (z,\ h\ z)))) \\ = \ &\mathit{mfix}\ f \gg\!= \ \lambda z.\ \mathit{return}\ (z,\ h\ z) \end{aligned}$$

Sliding applies, since $(f \cdot \pi_1)\ \bot = (f \cdot \pi_1 \cdot \lambda z.(z,\ h\ z))\ \bot = f\ \bot$. □

The diagram in this case looks like:



which suggests the name *pure right shrinking.*

Warning 2.6.7 In case we have lifted products, as in Haskell, the pattern matches against pairs should be done lazily. That is, every formula of the form: $\lambda(x,\ y).\ f\ x\ y$ should be replaced with $\lambda t.\ f\ (\pi_1\ t)\ (\pi_2\ t)$, or the Haskell equivalent $\lambda\tilde{}(x,\ y).\ f\ x\ y$. (And similarly for triples, quadruples, etc.) For instance, Equation 2.13 should be expressed as:

$$\mathit{mfix}\ (\lambda t.\ f\ (\pi_1\ t) \gg= \lambda z.\ \mathit{return}\ (z,\ h\ z)) = \mathit{mfix}\ f \gg= \lambda z.\ \mathit{return}\ (z,\ h\ z)$$

or,

$$\mathit{mfix}\ (\lambda\tilde{}(x,\ y).\ f\ x \gg= \lambda z.\ \mathit{return}\ (z,\ h\ z)) = \mathit{mfix}\ f \gg= \lambda z.\ \mathit{return}\ (z,\ h\ z)$$

avoiding the strict match against the tuple.

It is possible to generalize Equation 2.13, so that h can use x and y as well. We call this variant the *scope change* law:

Proposition 2.6.8 Let $f :: \alpha \to m\ \alpha$, $h :: \alpha \to (\alpha,\beta) \to \beta$,

$$\begin{aligned}&\mathit{mfix}\ (\lambda(x,\ y).\ f\ x \gg= \lambda z.\ \mathit{return}\ (z,\ h\ z\ (x,\ y)))\\ &= \mathit{mfix}\ f \gg= \lambda z.\ \mathit{return}\ (\mathit{fix}\ (\lambda(x,\ y).\ (z,\ h\ z\ (x,\ y))))\end{aligned} \tag{2.14}$$

provided *mfix* satisfies purity, left shrinking, nesting, and sliding laws.

Proof See Appendix B.2. □

Remark 2.6.9 Simple manipulation of the right hand side of Equation 2.14 yields the following equation:

$$\begin{aligned}&\mathit{mfix}\ (\lambda(x,\ y).\ f\ x \gg= \lambda z.\ \mathit{return}\ (z,\ h\ z\ (x,\ y)))\\ &= \mathit{mfix}\ f \gg= \lambda z.\ \mathit{return}\ (z,\ \mathit{fix}\ (\lambda y.\ h\ z\ (z,\ y)))\end{aligned} \tag{2.15}$$

This form of the scope changing property is quite useful in derivations, although somewhat less symmetric than Equation 2.14.

2.6.4 Parametricity: The "free" theorem

The least fixed-point operator on domains satisfies the following *uniformity* law [60, 82]: Let $f :: \alpha \rightarrow \alpha$, $g :: \beta \rightarrow \beta$, and $s :: \alpha \rightarrow \beta$, where s is strict. Then,

$$s \cdot f = g \cdot s \Longrightarrow s\ (\mathit{fix}\ f) = \mathit{fix}\ g \tag{2.16}$$

This extremely useful law is exactly the free theorem for the type $(\alpha \rightarrow \alpha) \rightarrow \alpha$, and hence granted by virtue of parametricity in our setting [75]. For *mfix*, parametricity gives us the following theorem for free:

Theorem 2.6.10 Let $f :: \alpha \rightarrow m\ \alpha$, $g :: \beta \rightarrow m\ \beta$, $s :: \alpha \rightarrow \beta$,

$$\mathit{map}\ s \cdot f = g \cdot s \Longrightarrow \mathit{map}\ s\ (\mathit{mfix}\ f) = \mathit{mfix}\ g \tag{2.17}$$

provided s is strict. □

Remark 2.6.11 It is worth emphasizing that we use Theorem 2.6.10 freely in our treatment of value recursion.[4] If one takes a more abstract view, of course, we expect Equation 2.17 to be postulated as a property to be checked, rather than taken for granted. Of course, this begs the question exactly what *strict* would mean in this new setting. See Simpson and Plotkin's recent work for a modern account of such questions [79]. (We will return to the treatment of value recursion in more abstract settings in Chapter 6.)

As we pointed out before, sliding strict computations is a direct consequence of parametricity:

Corollary 2.6.12 Let $f :: \alpha \rightarrow m\ \beta$, $h :: \beta \rightarrow \alpha$. Then,

$$\mathit{mfix}\ (\mathit{map}\ h \cdot f) = \mathit{map}\ h\ (\mathit{mfix}\ (f \cdot h)) \tag{2.18}$$

provided h is strict.

Proof Direct consequence of the free theorem with $F \mapsto f \cdot h, G \mapsto \mathit{map}\ h \cdot f$ and $S \mapsto h$, where we use capital letters to identify variables in Equation 2.17. □

[4] A word of caution is in order regarding Haskell and parametricity. It is well known that the **seq** primitive weakens the parametricity properties of Haskell [50, 68, 88]. We do not make use of this primitive in our work.

Parametricity allows us to take mirror images of our properties. For instance, the following equation is essentially the same as Equation 2.13:

$$\mathit{mfix}\ (\lambda(x,\ y).\ f\ y \gg\!= \ \lambda z.\ \mathit{return}\ (h\ z,\ z))\ =\ \mathit{mfix}\ f \gg\!= \ \lambda z.\ \mathit{return}\ (h\ z,\ z)$$

Obviously, we can consider the same equation over arbitrary length tuples and arbitrary permutations as well. We capture the essence of this process in the following corollary:

Corollary 2.6.13 Let $f, g :: (\alpha, \beta) \to m\ (\alpha, \beta)$. The equation $\mathit{mfix}\ f = \mathit{mfix}\ g$ holds exactly when its mirror image, that is:

$$\mathit{mfix}\ (\mathit{map}\ \mathit{swap}\ \cdot\ f\ \cdot\ \mathit{swap})\ =\ \mathit{mfix}\ (\mathit{map}\ \mathit{swap}\ \cdot\ g\ \cdot\ \mathit{swap})$$

holds, where $\mathit{swap}\ (x,\ y)\ =\ (y,\ x)$.

Proof Simple application of Corollary 2.6.12 on both sides. Note that *swap* is strict. □

As a final corollary to the free theorem, we consider the following *injection* law:

Corollary 2.6.14 Let $f :: \alpha \to m\ \alpha,\quad i :: \alpha \to \beta,\quad p :: \beta \to \alpha$, where p is strict and $p \cdot i = id_\alpha$. We have:

$$\mathit{mfix}\ f\ =\ \mathit{map}\ p\ (\mathit{mfix}\ (\mathit{map}\ i\ \cdot\ f\ \cdot\ p)) \tag{2.19}$$

Proof Let $F \mapsto \mathit{map}\ i\ \cdot\ f\ \cdot\ p,\ \ G \mapsto f$, and $S \mapsto p$ in the free theorem. Again, capital letters denote the variables in Equation 2.17. □

Note that Corollary 2.6.14 also follows from the sliding property. The intended reading of Equation 2.19 is as follows. The function i injects α's to β's, while p projects back. Hence, we can introduce spurious variables into the recursive loop, as long as they are not used anywhere.

2.7 Stronger properties

In this section we present two laws, strong sliding and right shrinking, which might be naively expected to be satisfied by value recursion operators. As we will prove in Chapter 3, however, they are both unsatisfiable for a wide variety of monads of practical interest. The most important monad satisfying both these properties is the lazy state monad (Section 4.4).

2.7.1 Strong sliding

If Equation 2.5 holds unconditionally, (i.e., without requiring $f\ (h\ \bot) = f\ \bot$), we say that the given value recursion operator satisfies the *strong sliding* property. As we will see in Chapter 3, strong sliding is not satisfiable for a variety of practical monads. However, when available, it allows us to deduce several interesting equalities:

Proposition 2.7.1 Let $f :: \alpha \to m\ \alpha$, and $q :: \alpha$. Then,

$$mfix\ (\lambda(x,\ \_).\ f\ x \gg\!= \lambda y.\ return\ (q,\ y)) = f\ q \gg\!= \lambda y.\ return\ (q,\ y) \tag{2.20}$$

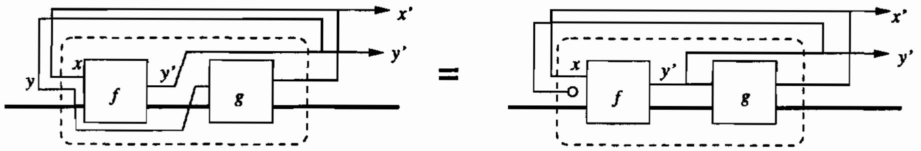
provided *mfix* satisfies the purity, left shrinking and strong sliding properties.

Proof See Appendix B.3. □

Proposition 2.7.2 Let $f :: \alpha \to m\ \beta$, $g :: \beta \to m\ \alpha$. Then,

$$\begin{aligned} &mfix\ (\lambda(x,\ y).\ f\ x \gg\!= \lambda y'.\ g\ y \gg\!= \lambda x'.\ return\ (x',\ y')) \\ &= mfix\ (\lambda(x,\ \_).\ f\ x \gg\!= \lambda y'.\ g\ y' \gg\!= \lambda x'.\ return\ (x',\ y')) \end{aligned} \tag{2.21}$$

provided *mfix* satisfies the purity, left shrinking, nesting and strong sliding properties. Diagrammatically:



Proof Straightforward applications of nesting, left shrinking, and the mirror image of the previous proposition on the left hand side. □

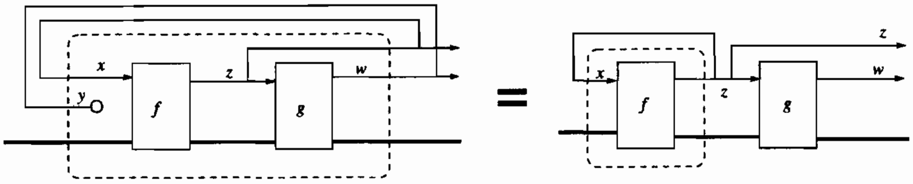
2.7.2 Right shrinking

Pure right shrinking (Corollary 2.6.6) tells us how to pull pure computations from the right hand side of a $\gg\!=$. Although it is not possible to pull out effectful computations in general, there are certain monads for which it is possible to do so, the most important examples being the output monad (or, in general, monads based on monoids—see Section 4.5), and the lazy state monad (Section 4.4). The following property captures the situation:

Property 2.7.3 (*Right shrinking.*) Let $f :: \alpha \to m\ \alpha$, $g :: \alpha \to m\ \beta$,

$$\begin{aligned} &mfix\ (\lambda(x,\ y).\ f\ x \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ return\ (z,\ w)) \\ &= mfix\ f \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ return\ (z,\ w) \end{aligned} \tag{2.22}$$

Diagrammatically:



Fact 2.7.4 Obviously, Equation 2.22 generalizes 2.13. That is, if a given value recursion operator satisfies right shrinking, it will automatically satisfy the pure version as well.

The combination of right shrinking and strong sliding allow us to generalize the scope change law (Proposition 2.6.8) as well:

Proposition 2.7.5 Let $f :: \alpha \to m\ \alpha$, $g :: \alpha \to (\alpha, \beta) \to m\ \beta$,

$$\begin{aligned} & \mathit{mfix}\ (\lambda(x,\ y).\ f\ x \gg\!= \ \lambda z.\ g\ z\ (x,\ y) \gg\!= \ \lambda w.\ \mathit{return}\ (z,\ w)) \\ & \quad = \mathit{mfix}\ f \gg\!= \ \lambda z.\ \mathit{mfix}\ (\lambda b.\ g\ z\ (z,\ b)) \gg\!= \ \lambda w.\ \mathit{return}\ (z,\ w) \end{aligned} \tag{2.23}$$

provided *mfix* satisfies purity, left shrinking, nesting, strong sliding and right shrinking.

Proof Analogous to the proof of Proposition 2.6.8. □

2.8 Classification and summary

Our properties try to capture the expected behavior of value recursion operators, formalizing our intuitions. It is worth reiterating the most important goals:

- Recursion should be performed only over the values, and the fixed-point computation should be similar to that of *fix*,
- Effects should be neither repeated nor lost,
- In the case when there are no recursively bound variables, **mdo** should behave exactly like a **do**.

How do our properties match these goals? Strictness states that the fixed-point is $\perp$ exactly when the given function is strict, analogous to *fix*. Purity states that, in case there are no effects, *mfix* should behave exactly like *fix*. These two properties are as close as we get to the behavior of the usual fixed-point operator on domains. Left shrinking states that **mdo** is exactly the same as **do**, in case there are no recursive bindings. We consider these three properties to be the most essential, leading to the following definition:

Definition 2.8.1 (*Value recursion operators.*) A value recursion operator for a monad $(m, \gg\!=, \mathit{return})$ is a function $\mathit{mfix} :: (\alpha \to m\ \alpha) \to m\ \alpha$, satisfying:

- Strictness: $f\ \bot_\alpha = \bot_{m\ \alpha} \quad \Leftrightarrow \quad \mathit{mfix}\ f\ = \bot_{m\ \alpha}$,
- Purity: $\mathit{mfix}\ (\mathit{return} \cdot h) = \mathit{return}\ (\mathit{fix}\ h)$,
- Left shrinking: $\mathit{mfix}\ (\lambda x.\ a \gg\!= \lambda y.\ f\ x\ y) = a \gg\!= \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y)$, provided x is not free in a.

At this point, two questions arise. First, why are sliding and nesting properties left out from Definition 2.8.1, even though we have found that they are both satisfied by many instances of *mfix* in practice (see Chapter 4)? And second, are there other properties of interest that we have completely missed?

The answer to the first question is a matter of choice. We would like to keep the requirements as simple as possible, but no simpler. As we will see several examples in Chapter 4, operators that do not satisfy the basic properties mandated by Definition 2.8.1 yield results that are not very sensible for value recursion. Other properties are just as important theoretically, but it is our belief that they are in a secondary status from a practical point of view.

It is much harder to answer the second question. Whether we have the "right" definition should become apparent as value recursion finds its place in practical functional programming. Our work, both in the context of this thesis and in using recursive monadic bindings in practical Haskell programs, led us to conclude that Definition 2.8.1 satisfactorily captures the minimal common core.

Finally, a comment on uniqueness is in order. Given a particular monad, we do *not* require a unique value recursion operator for it. There may be none, exactly one, or many operators satisfying the requirements of Definition 2.8.1. (For instance, in Chapter 4, we will be able to show that identity, maybe and list monads of Haskell have unique value recursion operators, while the state monad has an infinite chain of them. On the other extreme, the continuation monad probably has none—see Chapter 5 for details.) Furthermore, different operators for the same monad might satisfy different sets of properties in addition to the basic set mandated by Definition 2.8.1. In such a case, the user has the responsibility to pick the most appropriate operator for the problem at hand, possibly using our properties as a guide. We will see a concrete example of this situation in Section 4.4.

Chapter 3

Structure of monads and value recursion

So far, our study of value recursion was set in the context of arbitrary monads. We will now take a closer look at various properties that monads may satisfy, such as idempotency, commutativity, or additivity. The aim of this chapter is to investigate the implications of structural properties of monads for value recursion.

Synopsis. We first consider monads whose $\gg\!=$ operator is strict in its first argument, covering many practical monads of interest. We show that strong sliding and right shrinking properties are not satisfiable for such monads. We then consider idempotent, commutative and additive monads, trying to identify how value recursion operators should behave in each case. Finally, we briefly discuss embeddings and monad transformers.

3.1 Monads with a strict bind operator

Consider a monad m whose $\gg\!=$ operator is strict in its first argument. That is:

$$\bot_{m\ \tau} \gg\!= f = \bot_{m\ \sigma} \tag{3.1}$$

for all $f :: \tau \to m\ \sigma$. Haskell's *maybe*, *list*, IO, and strict state monads are examples of such monads. In this section, we will prove that neither strong sliding, nor right shrinking properties can be satisfied for such a monad, unless it is trivial in the following sense:

Definition 3.1.1 (*Trivial monad.*) A monad $(m, \gg\!=, \mathit{return})$ is trivial if, for all types τ, the domain corresponding to the type $m\ \tau$ consists only of $\bot_{m\ \tau}$.

Remark 3.1.2 The canonical example of a trivial monad is:

```
data Void α      -- no constructors, all we have is ⊥

return x  = ⊥
m >>= f   = ⊥
```

Note that all of our properties hold for a trivial monad, with the only possible definition $\mathit{mfix}\ f\ =\ \bot$.

Lemma 3.1.3 Let $(m,\ \gg\!=,\ \mathit{return})$ be a monad where $\gg\!=$ is strict in its first argument. If *return* is strict as well, then m is trivial.[1]

Proof Pick an arbitrary type τ, and let a be an arbitrary element of $m\ \tau$. We have:

$$\begin{array}{lll} a & = & \mathit{const}\ a\ \bot_\tau & \{\mathit{const}\ x\ y\ =\ x\} \\ & = & \mathit{return}_\tau\ \bot_\tau\ \gg\!=\ \mathit{const}\ a & \{\mathit{left\ unit}\} \\ & = & \bot_{m\ \tau}\ \gg\!=\ \mathit{const}\ a & \{\mathit{return\ is\ strict}\} \\ & = & \bot_{m\ \tau} & \{\gg\!=\ \mathit{is\ strict}\} \end{array}$$

The result now follows by Definition 3.1.1. □

Note that Lemma 3.1.3 requires *return* to be strict at all types. The following lemma simplifies this requirement, reducing the proof obligation to *return* being strict at only one particular type:[2]

Lemma 3.1.4 Let $(m,\ \gg\!=,\ \mathit{return})$ be a monad where $\gg\!=$ is strict in its first argument. If *return* is strict at one type, (i.e., there exists a type τ s.t. $\mathit{return}_\tau\ \bot_\tau\ =\ \bot_{m\ \tau}$), then it is strict at all types.

Proof See Appendix B.4. □

After these preliminary results, we can now proceed with our original goal:

Proposition 3.1.5 Let $(m,\ \gg\!=,\ \mathit{return})$ be a monad where $\gg\!=$ is strict in its first argument. If there is a value recursion operator for m that satisfies the strong sliding property of Section 2.7.1, then m is trivial.

Proof We will first establish that if such an operator exists, then *return* must be strict. Define:[3]

$$\begin{array}{ll} f\ \ ::\ ()\to m\ () & h\ \ ::\ ()\to () \\ f\ ()\ =\ \mathit{return}\ () & h\ \_\ =\ () \end{array}$$

Note that $f\ \cdot\ h\ =\ \lambda x.\ \mathit{return}\ ()$. Let *mfix* be a value recursion operator for m satisfying the strong sliding property. Then, Equation 2.6 must hold with no side conditions. The right hand side of Equation 2.6 reads:

[1] For brevity, we simply refer to a monad $(m,\ \gg\!=,\ \mathit{return})$ by the name of its type constructor, i.e., m.

[2] This lemma and its proof has been suggested to us by Ross Paterson (personal communication).

[3] The domain corresponding to the unit type, written () following the Haskell notation, consists of exactly two elements: $\bot$ and $()$, with the obvious ordering $\bot \sqsubset ()$.

$$mfix\ (f \cdot h) \gg\!= \ return \cdot h$$

and, by Proposition 2.6.1 and the left unit law, it must be equal to *return* (). Similarly, the left hand side of Equation 2.6 reads:

$$mfix\ (\lambda x.\ f\ x \gg\!= \ return \cdot h)$$

and, by the strictness property, it must compute to $\perp$. (Note that f is strict because it matches its argument against (), and $\gg\!=$ is strict in its first argument by hypothesis.) Hence, strong sliding implies *return* $() = \perp$. By monotonicity, then, *return* must be strict at the type (). Hence, by Lemmas 3.1.4 and 3.1.3, m must be trivial. □

A similar argument shows that right shrinking property shares the same fate:

Proposition 3.1.6 Let (m, $\gg\!=$, *return*) be a monad where $\gg\!=$ is strict in its first argument. If there is a value recursion operator for m that satisfies the right shrinking property of Section 2.7.2, then m is trivial.

Proof Define:

$$\begin{array}{lcl} f & :: & [Int] \to m\ [Int] \\ f\ xs & = & return\ (1 : xs) \end{array} \qquad\qquad \begin{array}{lcl} g & :: & [Int] \to m\ Int \\ g\ [x] & = & return\ x \\ g\ \_ & = & return\ 1 \end{array}$$

It is easy to see that the left hand side of Equation 2.22 must yield $\perp$ by the strictness property (note that g will diverge on $1 : \perp$). By purity, we have

$$mfix\ f = return\ (fix\ (\lambda xs.\ 1 : xs))$$

Hence, the right hand side of Equation 2.22 evaluates to

$$return\ (1,\ fix\ (\lambda xs.\ 1 : xs))$$

implying that $\perp = return\ (1,\ fix\ (\lambda xs.\ 1 : xs))$. By monotonicity, then, *return* must be strict at the type (Int, [Int]) . Hence, by Lemmas 3.1.4 and 3.1.3, m must be trivial. □

In other words, unless a given monad m is trivial, no value recursion operator for m can satisfy strong sliding and right shrinking properties, provided m's $\gg\!=$ operator is strict in its first argument. This is an important result, as it identifies inherent limitations on properties that can be expected to hold for many practical monads of interest.

Corollary 3.1.7 Neither strong sliding nor right shrinking properties are satisfiable for Haskell's *maybe*, *list*, strict state and IO monads, as none of these monads are trivial (no pun intended—see Definition 3.1.1). □

3.2 Idempotent monads

A monad m is said to be idempotent if the equation[4]

$$a \ggg= \lambda x.\ a \ggg= \lambda y.\ \mathit{return}\ (x,\ y) \ = a \ggg= \lambda x.\ \mathit{return}\ (x,\ x) \qquad (3.2)$$

holds for all $a :: m\ \tau$ [46]. Identity, *maybe* and environment monads are examples of idempotent monads. Intuitively, a monad is idempotent if computations can be duplicated whenever their results are needed.

Note that Equation 3.2 does not specify any data flow between repeated computations. That is, the equation

$$\lambda x.\ f\ x \ggg= f \ = \ f \qquad (3.3)$$

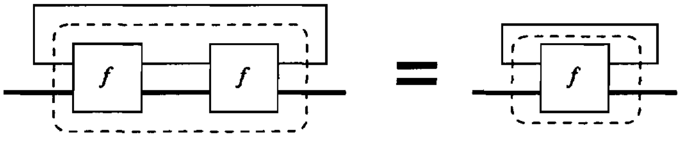
is *not* required to hold.[5] However, if a monad is idempotent, we expect both sides of Equation 3.3 to be indistinguishable by *mfix*. Furthermore, once *mfix f* is computed for a function f, further applications of f should not change the result. We capture these intuitions in the following property:

Property 3.2.1 (*Idempotency.*) Let $f :: \alpha \to m\ \alpha$, where m is an idempotent monad with a value recursion operator *mfix*. Then,

$$\mathit{mfix}\ (\lambda x.\ f\ x \ggg= f) \ = \ \mathit{mfix}\ f \qquad (3.4)$$

$$\mathit{mfix}\ f \ggg= f \ = \ \mathit{mfix}\ f \qquad (3.5)$$

The first equality can be captured diagrammatically as follows:



We leave it to the reader to picture Equation 3.5.

Remark 3.2.2 It is important to note that Property 3.2.1 does *not* state that Equation 3.4 or 3.5 can be used as definitions of value recursion operators whenever the underlying monad is idempotent.[6] For instance, Equation 3.5 will always produce $\perp$ for

[4] In category theory, a monad m is called idempotent if its $\mathit{join} :: m\ (m\ \alpha) \to m\ \alpha$ operator is an isomorphism [55]. The definition we use is more useful from a practical point of view, however. For instance, the *maybe* monad is idempotent with our definition, although its *join* operator is *not* an isomorphism.

[5] As a counterexample, consider the identity monad where Equation 3.3 is satisfied only for idempotent functions (i.e., $f^2 = f$), but not in general.

[6] Individual definitions might coincide, of course. For instance, in Chapter 4, we will see that Equation 3.5 does indeed define value recursion operators for identity and environments, but not for exceptions.

mfix in a monad with a $\gg=$ operator that is strict in its first argument, which is clearly undesirable.

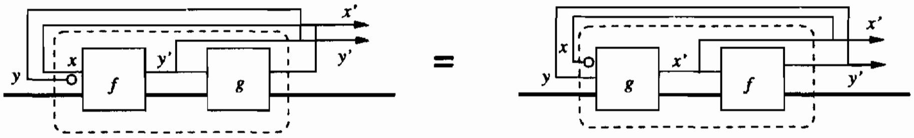
We will discuss idempotency property with respect to identity, exception, monads based on idempotent monoids, and environments in Chapter 4.

3.3 Commutative monads

A monad m is said to be commutative if the order of effects does not matter. That is, if the equation

$$\begin{aligned} &\lambda(x,\ y).\ f\ x \gg= \ \lambda x'.\ g\ y \gg= \ \lambda y'.\ \mathit{return}\ (x',\ y') \\ = \ &\lambda(x,\ y).\ g\ y \gg= \ \lambda y'.\ f\ x \gg= \ \lambda x'.\ \mathit{return}\ (x',\ y') \end{aligned} \tag{3.6}$$

holds for all $f :: \alpha \to m\ \beta$ and $g :: \tau \to m\ \sigma$. For a commutative monad, we expect *mfix* to satisfy swapping of computations similarly, as depicted in the following diagram:



Property 3.3.1 (*Commutativity.*) Let $f :: \alpha \to m\ \beta$, $g :: \beta \to m\ \alpha$, where m is a commutative monad with a value recursion operator *mfix*. Then,

$$\begin{aligned} &\mathit{mfix}\ (\lambda(x,\ \_).\ f\ x \gg= \ \lambda y'.\ g\ y' \gg= \ \lambda x'.\ \mathit{return}\ (x',\ y')) \\ = \ &\mathit{mfix}\ (\lambda(\_,\ y).\ g\ y \gg= \ \lambda x'.\ f\ x' \gg= \ \lambda y'.\ \mathit{return}\ (x',\ y')) \end{aligned} \tag{3.7}$$

In case a value recursion operator satisfies nesting and strong sliding laws, Equation 3.7 can be derived automatically:

Proposition 3.3.2 Equation 3.7 follows from nesting and strong sliding laws.

Proof Straightforward applications of Equation 2.21, Equation 3.6, nesting, left shrinking, and Equation 2.20 on the left hand side. □

Examples of commutative monads include identity, environments, and monads based on commutative monoids. We will investigate the commutativity property with respect to these monads in Chapter 4.

3.4 Monads with addition

A monad m is said to be additive if there exists an element $\mathit{zero} :: m\ \tau$, and an operation $\oplus :: m\ \tau \to m\ \tau \to m\ \tau$, such that:

$$\begin{array}{rcl} \mathit{zero} \oplus p & = & p \\ p \oplus \mathit{zero} & = & p \\ (p \oplus q) \oplus r & = & p \oplus (q \oplus r) \end{array} \qquad \begin{array}{rcl} \mathit{zero} \gg\!= f & = & \mathit{zero} \\ p \gg \mathit{zero} & = & \mathit{zero} \end{array}$$

The relation between $\oplus$ and $\gg\!=$ is not specified, although one generally checks for the following distributive laws:

$$(p \oplus q) \gg\!= f \;=\; p \gg\!= f \oplus q \gg\!= f \tag{3.8}$$

$$p \gg\!= (\lambda x.\, q\ x \oplus r\ x) \;=\; p \gg\!= q \oplus p \gg\!= r \tag{3.9}$$

In Haskell, additive monads are captured as instances of the *MonadPlus* class, where *zero* is called *mzero* and $\oplus$ is called *mplus* [68]. The *maybe* and *list* monads are instances of this class.[7] It is interesting to note that the *list* monad satisfies Equation 3.8, but not 3.9; while the *maybe* monad satisfies Equation 3.9, but not 3.8.

For an additive monad, we expect the following property to hold:

Property 3.4.1 (*Distributivity.*) Let m be an additive monad with $\oplus$ as the binary operator. Let *mfix* be a value recursion operator for m. Distributivity states:

$$\mathit{mfix}\ (\lambda x.\, f\ x \oplus g\ x) = \mathit{mfix}\ f \oplus \mathit{mfix}\ g \tag{3.10}$$

If Equation 3.8 holds, left shrinking is sufficient to establish the distributivity property:

Proposition 3.4.2 Let m be an additive monad with $\oplus$ as the binary operator, and let *mfix* be a value recursion operator for m. If $\oplus$ satisfies Equation 3.8, then *mfix* will satisfy distributivity.

Proof See Appendix B.5. □

Remark 3.4.3 It is worth noting that Equation 3.8 is a sufficient, but *not* a necessary condition for satisfying distributivity. As we will see in Chapter 4, the *maybe* monad does not satisfy Equation 3.8, yet it has a value recursion operator satisfying distributivity.

[7] In fact, the law $p \gg \mathit{zero} = \mathit{zero}$ fails for both the *maybe* and *list* monads when $p = \bot$. This discrepancy does not cause any trouble for our purposes. (Recall: $m \gg k = m \gg\!= \lambda\_.k$.)

3.5 Embeddings

Consider Haskell's *maybe* and *list* monads. Intuitively, every value of type *Maybe* τ can be considered as a value of type $[\tau]$, mapping *Nothing* to [ ] and *Just* x to $[x]$. In a certain sense, the *list* monad is rich enough to capture the features of the *maybe* monad. Formally, this relation is captured by monad homomorphisms and embeddings [53, 89]:

Definition 3.5.1 (*Monad homomorphisms and embeddings.*) Let m and n be two monads. A monad homomorphism, $\epsilon :: m \rightarrow n$, is a family of functions, one for each type τ, $\epsilon_\tau :: m\ \tau \rightarrow n\ \tau$, such that:

$$\begin{aligned} \epsilon \cdot \mathit{return}_m &= \mathit{return}_n && (3.11) \\ \epsilon_\sigma\ (k \gg\!=_m f) &= \epsilon_\tau\ k \gg\!=_n \epsilon_\sigma \cdot f && (3.12) \end{aligned}$$

where $k :: m\ \tau$ and $f :: \tau \rightarrow m\ \sigma$. An embedding is a monad homomorphism where each ϵ_τ is monic (i.e., injective).

Equations 3.11 and 3.12 precisely describe how ϵ interacts with the proper morphisms of the involved monads. For value recursion, we also need to specify how ϵ and *mfix* interacts:

Definition 3.5.2 (*Monad homomorphisms and embeddings for value recursion.*) Let m and n be two monads with respective value recursion operators mfix_m and mfix_n. Let $\epsilon :: m \rightarrow n$ be a monad homomorphism or embedding. We say that ϵ respects value recursion if, for all $f :: \tau \rightarrow m\ \tau$,

$$\epsilon\ (\mathit{mfix}_m\ f) = \mathit{mfix}_n\ (\epsilon \cdot f) \qquad (3.13)$$

In Chapter 4, we will see several concrete examples, including the embeddings of maybe into list, environment and output into state, and identity into any other monad.

Proposition 3.5.3 Let $\epsilon : m \rightarrow n$ be an embedding of a monad m into a monad n. Let mfix_n be a value recursion operator for n. Let $g :: (\tau \rightarrow m\ \tau) \rightarrow m\ \tau$ be a function, satisfying the strictness property. If ϵ satisfies Equation 3.13 where g plays the role of mfix_m, then g is a value recursion operator for m, i.e., it will satisfy purity and left shrinking properties as well.

Proof Simple equational reasoning. We present the left shrinking case to illustrate the idea:

$$\begin{array}{ll}
\quad \epsilon\ (g\ (\lambda x.\ a \gg= \ \lambda y.\ f\ x\ y)) & \\
= \mathit{mfix}\ (\lambda x.\ \epsilon\ (a \gg= \ \lambda y.\ f\ x\ y)) & \{\mathit{Eqn.}\ 3.13\} \\
= \mathit{mfix}\ (\lambda x.\ \epsilon\ a \gg= \ \lambda y.\ \epsilon\ (f\ x\ y)) & \{\mathit{Eqn.}\ 3.12\} \\
= \epsilon\ a \gg= \lambda y.\ \mathit{mfix}\ (\lambda x.\ \epsilon\ (f\ x\ y)) & \{\mathit{left\ shrink}\} \\
= \epsilon\ a \gg= \lambda y.\ \epsilon\ (g\ (\lambda x.\ f\ x\ y)) & \{\mathit{Eqn.}\ 3.13\} \\
= \epsilon\ (a \gg= \ \lambda y.\ g\ (\lambda x.\ f\ x\ y)) & \{\mathit{Eqn.}\ 3.12\}
\end{array}$$

Since ϵ is injective, we obtain:

$$g\ (\lambda x.\ a \gg= \ \lambda y.\ f\ x\ y) = a \gg= \ \lambda y.\ g\ (\lambda x.\ f\ x\ y)$$

showing that g satisfies left shrinking. □

Remark 3.5.4 It is unfortunate that strictness is not necessarily reflected. Using the proof technique above, one gets: $\epsilon\ (g\ f) = \mathit{mfix}_n\ (\epsilon \cdot f)$, but we cannot conclude that g satisfies strictness unless ϵ is strict. It turns out that requiring ϵ to be strict is an overkill; many embedding examples we will see in Chapter 4 are not strict.

Proposition 3.5.5 The sliding, nesting, strong sliding and right shrinking properties are reflected through embeddings as well. That is, if $\epsilon : m \rightarrow n$ is an embedding respecting value recursion, and if mfix_n satisfies any of these properties, then so will mfix_m.

Proof Similar to the previous proposition. □

Observation 3.5.6 Composition of two embeddings is still an embedding, hence properties are reflected through multiple embeddings as well.

Is it possible to derive value recursion operators using embeddings? Intuitively, if a monad m embeds into another monad n, and if n has a value recursion operator, one might expect to be able to derive a value recursion operator for m. In this case, we will need the embedding to be a split monic, i.e., to possess a left inverse, in order to be able to map results back to m. For instance, the embedding of the *maybe* monad into the *list* monad, and its left inverse, are given by:

$$\begin{array}{llll}
\epsilon\ \mathit{Nothing} & = [\,] \qquad\qquad & \epsilon^{\ell}\ [\,] & = \mathit{Nothing} \\
\epsilon\ (\mathit{Just}\ x) & = [x] & \epsilon^{\ell}\ (x{:}xs) & = \mathit{Just}\ x
\end{array}$$

More formally, let $\epsilon :: m \rightarrow n$ be an embedding with the left inverse $\epsilon^{\ell} :: n \rightarrow m$, i.e., $\epsilon^{\ell}_{\tau} \cdot \epsilon_{\tau} = \mathit{id}_{m\ \tau}$. Note that, in general, ϵ^{ℓ} is not a monad homomorphism.[8] Let mfix_n be a

[8] Furthermore, ϵ and e^{ℓ} are not required to form a retraction pair, i.e., $\epsilon \cdot \epsilon^{\ell} \not\sqsubseteq \mathit{id}$ [77]. In fact, $\epsilon \cdot \epsilon^{\ell}$ is generally incomparable to *id*, as demonstrated by the embedding of *maybe* into *list*.

value recursion operator for n. When is the function:

$$\begin{array}{lcl} g & :: & (\alpha \to m\ \alpha) \to m\ \alpha \\ g\ f & = & \epsilon^{\ell}\ (\mathit{mfix}_n\ (\epsilon \cdot f)) \end{array} \tag{3.14}$$

a value recursion operator for m? Since e^{ℓ} is not a monad homomorphism, not all required properties will follow automatically. Still, this construction gives a way of obtaining a candidate value recursion operator, and we can test whether ϵ respects value recursion with respect to it. In this case, we need to verify:

$$(\epsilon \cdot \epsilon^{\ell})\ \ (\mathit{mfix}_n\ (\epsilon \cdot f)) = \mathit{mfix}_n\ (\epsilon \cdot f) \tag{3.15}$$

for all $f :: \alpha \to m\ \alpha$. If Equation 3.15 holds, Propositions 3.5.3 and 3.5.5 will be sufficient to establish properties for g automatically.

Remark 3.5.7 It is easy to see that e^{ℓ} will always satisfy Equation 3.11. In general, Equation 3.12 will only be satisfied on the subset of values that are in the image of ϵ. The *maybe* into *list* embedding given above illustrates this point. However, we suspect that the subset of values on which Equation 3.12 is satisfied might be sufficient to establish further properties of the derived operator. We leave the exploration of this idea for future work.

3.6 Monad transformers

Closely related to monad homomorphisms is the idea of monad transformers. It is often the case that one wants to add new features to an already existing monad. For instance, one can add exceptions, state or non-determinism to a monad, obtaining a monad with new computational features. Monad transformers have been designed to solve this problem in a modular manner. Intuitively, given a monad m, a monad transformer t yields a new monad $t\ m$, transforming return_m to $\mathit{return}_{t\ m}$ and $\gg\!=_m$ to $\gg\!=_{t\ m}$. Furthermore, one requires a monad homomorphism $\mathit{lift}_\tau :: m\ \tau \to t\ m\ \tau$, lifting computations in m to the new monad. We refrain from going into details here, the reader is referred to the rich literature on monad transformers for details [22, 42, 53, 54].

For value recursion, we ask a similar question. Given a monad transformer t, is there a natural way of obtaining $\mathit{mfix}_{t\ m}$ from mfix_m? A generic approach would be to convert a given function $f :: \alpha \to t\ m\ \alpha$ to a function of type $\alpha \to m\ \alpha$, apply mfix_m to get the fixed-point $m\ \alpha$, and transfer it back to $t\ m\ \alpha$ using *lift*. Unfortunately, to do the conversion from $\alpha \to t\ m\ \alpha$ to $\alpha \to m\ \alpha$, one would need a morphism with type $t\ m\ \alpha \to m\ \alpha$, the inverse of *lift*, which is clearly not available in general.

On the other hand, it is generally possible to lift arbitrary value recursion operators, provided we know the exact structure of the monad transformer. We will consider three examples of monad transformers in Section 4.9, namely errors, environments, and state, and show how we can lift the value recursion operators through these transformers. (This technique does not always work, however, as illustrated by the continuation monad transformer. See Section 5.2 for details.)

3.7 Summary

In this chapter, we have concentrated on properties of value recursion operators that follow from the structural properties of underlying monads. As we have seen, if the $\gg=$ operator is strict in its first argument, then the strong sliding and right shrinking properties cannot be satisfied. This is an important point: there are inherent limitations on what we can expect from recursion in the presence of effects. (We will return to this issue in Chapter 6.)

The latter part of this chapter dealt with how value recursion operators reflect properties such as idempotency, commutativity, and additivity, and how individual properties are reflected through monad embeddings. In Chapter 4, we will get a chance to review these properties with respect to concrete examples of value recursion operators.

Chapter 4

A catalog of value recursion operators

In this chapter, we present value recursion operators for monads that are frequently used in functional programming, providing a catalog of *mfix*'s for the working programmer. Although there is no magic recipe, we believe that these examples present enough patterns to guide the construction of value recursion operators for new monads.

Synopsis. We establish a framework with the identity monad and then cover exceptions, lists, state, output, environments, trees, and fudgets. The continuation monad proves to be problematic; we consider it separately in Chapter 5. We also discuss monad transformers, enabling us to create new *mfix*'s from old.

4.1 Identity

The identity monad is the monad of pure values, modeling computations with no effects:

$$\textbf{type}\ \mathit{Identity}\ \alpha = \alpha$$

$$\begin{aligned} \mathit{return} &= \mathit{id} \\ x \gg\!= f &= f\ x \end{aligned}$$

with *fix* as the corresponding value recursion operator, i.e:

$$\begin{aligned} \mathit{mfix} &:: (\alpha \to \mathit{Identity}\ \alpha) \to \mathit{Identity}\ \alpha \\ \mathit{mfix}\ f &= \mathit{fix}\ f \end{aligned} \tag{4.1}$$

Proposition 4.1.1 Equation 4.1 defines the unique value recursion operator for the identity monad.

Proof It is easy to show that *fix* satisfies strictness, purity, and left shrinking properties. For uniqueness, we will show that any value recursion operator for the identity monad must equal *fix*. Let *mfix'* be such an operator. We have:

$$\mathit{mfix}'\ f = \mathit{mfix}'\ (\mathit{return} \cdot f) = \mathit{return}\ (\mathit{fix}\ f) = \mathit{fix}\ f$$

by using purity and the fact that *return* = *id*. □

Remark 4.1.2 Although we will stick to Haskell notation, we will generally avoid using explicit tags to reduce clutter as long as we can. For instance, for overloading purposes, the proper way to define the identity monad and *mfix* in Haskell is:[1]

$$\begin{aligned}
&\textbf{newtype}\ \mathit{Identity}\ \alpha = \mathit{Id}\ \{\ \mathit{unId} :: \alpha\ \}\\[1ex]
&\textbf{instance}\ \mathit{Monad}\ \mathit{Identity}\ \textbf{where}\\
&\quad \mathit{return}\ x \qquad = \mathit{Id}\ x\\
&\quad \mathit{Id}\ x \gg\!= \ f = f\ x\\[1ex]
&\mathit{mfix}\ f = \mathit{fix}\ (f \cdot \mathit{unId})
\end{aligned}$$

Properties It is easy to see that all of our properties hold for Equation 4.1, including nesting, strong sliding and right shrinking. Furthermore, the identity monad is both idempotent and commutative, and it is an easy exercise to show that Properties 3.2.1 and 3.3.1 both hold.

The identity monad embeds into any other monad n, as long as return_n is monic. The homomorphism $\epsilon = \mathit{return}_n$ easily satisfies Equations 3.11-3.13, assuming n has a value recursion operator. (In other words, the identity monad is initial in the category of monads and monad homomorphisms.)

4.2 Exceptions: The *maybe* monad

The *maybe* monad of Haskell can be used to model exceptions:

$$\begin{aligned}
&\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing}\ |\ \mathit{Just}\ \alpha\\[1ex]
&\mathit{return} \qquad\qquad\ = \mathit{Just}\\
&\mathit{Nothing} \gg\!= \ f = \mathit{Nothing}\\
&\mathit{Just}\ x\ \ \gg\!= \ f = f\ x
\end{aligned}$$

with the following unique value recursion operator:

$$\begin{aligned}
&\mathit{mfix}\ \ :: (\alpha \to \mathit{Maybe}\ \alpha) \to \mathit{Maybe}\ \alpha\\
&\mathit{mfix}\ f = \mathit{fix}\ (f \cdot \mathit{unJust})\\
&\qquad \textbf{where}\ \mathit{unJust}\ (\mathit{Just}\ x) = x
\end{aligned} \tag{4.2}$$

Proposition 4.2.1 Equation 4.2 defines the unique value recursion operator for the *maybe* monad.

[1] The **newtype** declaration avoids adding a separate $\perp$ element. If a **data** declaration is used, $\gg\!=$ should match lazily (i.e., $\tilde{}(\mathit{Id}\ x) \gg\!= \ f = f\ x$) to avoid strictness problems. (See Section 3.1 for details.)

Proof Strictness and purity are straightforward. For left shrinking, we need to show:

$$\mathit{mfix}\ (\lambda x.\ a \gg\!= \ \lambda y.\ f\ x\ y) = a \gg\!= \ \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y)$$

where a is a free variable. Case analysis on a suffices to show the equivalence. When $a = \bot$, both sides yield $\bot$. When $a = \mathit{Nothing}$, we get *Nothing*. Finally, when $a = \mathit{Just}\ z$ for some z, both sides yield $\mathit{mfix}\ (\lambda x.\ f\ x\ z)$.

To show uniqueness, we do a similar case analysis. If $f\ \bot = \bot$, $\mathit{mfix}\ f$ must be $\bot$ by strictness. If $f\ \bot = \mathit{Nothing}$, monotonicity implies that $f = \mathit{const}\ \mathit{Nothing}$, and Proposition 2.6.1 guarantees that $\mathit{mfix}\ f = \mathit{Nothing}$. Finally, if $f\ \bot = \mathit{Just}\ z$ for some z, then f must factor through *Just* by monotonicity, i.e., there must be a function h such that $f = \mathit{Just} \cdot h$, or equivalently, $h = \mathit{unJust} \cdot f$. Therefore,

$$\begin{aligned} \mathit{mfix}\ f &= \mathit{mfix}\ (\mathit{Just} \cdot h) \\ &= \mathit{mfix}\ (\mathit{return} \cdot h) \\ &= \mathit{return}\ (\mathit{fix}\ h) && \{\mathit{purity}\} \\ &= \mathit{return}\ (\mathit{fix}\ (\mathit{unJust} \cdot f)) \end{aligned}$$

To summarize, we have:

$$\begin{aligned} \mathit{mfix}\ f = \ &\mathbf{case}\ f\ \bot\ \mathbf{of} \\ &\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\ &\quad \mathit{Just}\ \_ \quad\ \rightarrow \mathit{return}\ (\mathit{fix}\ (\mathit{unJust} \cdot f)) \end{aligned} \tag{4.3}$$

Note that we did not make any choices in constructing Equation 4.3; the behavior of *mfix* is completely dictated by the properties that must be satisfied by all value recursion operators. We leave it to the reader to show that Equations 4.2 and 4.3 are equivalent, establishing uniqueness. □

Remark 4.2.2 By Proposition 2.6.3, $f\ \bot$ is always an approximation to $\mathit{mfix}\ f$, justifying the case expression in Equation 4.3. Note that the case when $f\ \bot = \bot$ is implicitly handled by pattern match failure.

Properties It is easy to show that Equation 4.2 also satisfies sliding and nesting properties. As stated in Corollary 3.1.7, strong sliding and right shrinking properties fail.

How about idempotency (Proposition 3.2.1) and commutativity (Proposition 3.3.1)? It turns out that the exception monad is indeed idempotent (i.e., satisfies Equation 3.2). Equations 3.4 and 3.5 are both satisfied. On the other hand, exceptions are not commutative, due to the possibility of non-termination: $\mathit{Nothing} \gg\!= \ \lambda x.\ \bot = \mathit{Nothing}$, whereas $\bot \gg\!= \ \lambda x.\ \mathit{Nothing} = \bot$. Consequently the commutativity property is not applicable.

Finally, we consider the distributivity (Property 3.4.1). As mentioned in Section 3.4, the *maybe* monad is additive:

$$\begin{aligned} &\mathit{zero} &&= \mathit{Nothing} \\ &\mathit{Nothing} \oplus y &&= y \\ &\mathit{Just}\ x \ \ \oplus y &&= \mathit{Just}\ x \end{aligned}$$

To establish

$$\mathit{mfix}\ (\lambda x.\ f\ x \oplus g\ x) = \mathit{mfix}\ f \oplus \mathit{mfix}\ g$$

it suffices to do a case analysis on $f\ \bot$. In case $f\ \bot = \bot$, both sides will yield $\bot$. In case $f\ \bot = \mathit{Nothing}$, we will get *mfix g* on both sides. Finally, if $f\ \bot$ takes the form of a *Just*, both sides will reduce to *mfix f*. We leave the details to the reader.

Remark 4.2.3 It is instructive to study failing definitions of *mfix* as well. Consider:

$$\begin{aligned} \mathit{mfix}'\ f\ = &\ \mathbf{let}\ \mathit{Just}\ x = f\ x \\ &\ \mathbf{in}\ \mathit{return}\ x \end{aligned}$$

which is somewhat intuitive, considering how the recursive knot is tied over x. Obviously, strictness fails. More seriously, left shrinking fails as well:

$$\begin{aligned} &\mathit{mfix}'\ (\lambda x.\ \mathit{Nothing} \gg\!= \ \lambda y.\ \mathit{return}\ 1) = \mathit{Just}\ \bot \\ &\mathit{Nothing} \gg\!= \ \lambda y.\ \mathit{mfix}'\ (\lambda x.\ \mathit{return}\ 1) = \mathit{Nothing} \end{aligned}$$

compromising the equivalence of **do** and **mdo** expressions in the absence of recursion. We also have $\mathit{mfix}'\ (\lambda x.\ \mathit{Nothing}) = \mathit{Just}\ \bot$, which is truly bizarre.

4.3 Lists

The *list* monad of Haskell can be used to model computations with multiple results:

$$\begin{aligned} &\mathit{return}\ x &&= [x] \\ \\ &[\,]\ \ \ \ \ \ \gg\!=\ f &&= [\,] \\ &(x{:}xs) \gg\!=\ f &&= f\ x \mathbin{+\!\!+} (xs \gg\!=\ f) \end{aligned}$$

Given a function $f :: \alpha \rightarrow [\alpha]$, how do we compute $\mathit{mfix}\ f :: [\alpha]$? Intuitively, we need to select a pivot value to tie the recursive knot. Consider the following two candidates:

$$\begin{aligned} &\mathbf{let}\ (a : \_) = f\ a \qquad\qquad &&\mathbf{let}\ (\_ : a : \_) = f\ a \\ &\mathbf{in}\ f\ a &&\mathbf{in}\ f\ a \end{aligned}$$

where we pivot over the first and the second element of the result, respectively. Of course, there is an infinite family of such functions, one for each particular position. As we will see later in this section, none of these alternatives give rise to a value recursion operator. Instead, we consider a moving pivot: Rather than fixing a single pivot element for the entire computation, we compute each element in the result using its own position as the pivot element. That is, the ith element of the fixed-point of f can be selected as the fixed-point of the function $head^i \cdot f$, suggesting:

$$\mathit{mfix}\ f = \mathit{fix}\ (\mathit{head} \cdot f) : \mathit{mfix}\ (\mathit{tail} \cdot f)$$

There is a slight problem with this approach, however: It always generates an infinite list, repeating $\bot$ after reaching the actual end of the list. Luckily, there is an easy solution. Rather than computing $\mathit{fix}\ (\mathit{head} \cdot f)$, we can compute $\mathit{fix}\ (f \cdot \mathit{head})$, and stop when the result is [ ]. Putting these ideas altogether, we obtain the following operator:

$$\begin{array}{lll} \mathit{mfix} & :: & (\alpha \to [\alpha]) \to [\alpha] \\ \mathit{mfix}\ f & = & \mathbf{case}\ \mathit{fix}\ (f \cdot \mathit{head})\ \mathbf{of} \\ & & \quad [\,] \quad \to [\,] \\ & & \quad (x{:}\_) \to x : \mathit{mfix}\ (\mathit{tail} \cdot f) \end{array} \tag{4.4}$$

As the following proposition shows, this definition of *mfix* is extremely well-behaved:

Proposition 4.3.1 The function *mfix* given by Equation 4.4 satisfies:

$$\mathit{mfix}\ f = \bot \;\Leftrightarrow\; f\ \bot = \bot \tag{4.5}$$

$$\mathit{mfix}\ f = [\,] \;\Leftrightarrow\; f\ \bot = [\,] \tag{4.6}$$

$$\mathit{head}\ (\mathit{mfix}\ f) = \mathit{fix}\ (\mathit{head} \cdot f) \tag{4.7}$$

$$\mathit{tail}\ (\mathit{mfix}\ f) = \mathit{mfix}\ (\mathit{tail} \cdot f) \tag{4.8}$$

$$\mathit{mfix}\ (\lambda x.\ f\ x : g\ x) = \mathit{fix}\ f : \mathit{mfix}\ g \tag{4.9}$$

$$\mathit{mfix}\ (\lambda x.\ f\ x +\!\!+\ g\ x) = \mathit{mfix}\ f +\!\!+\ \mathit{mfix}\ g \tag{4.10}$$

Proof See Appendix B.6. □

Remark 4.3.2 From the first two equivalences in Proposition 4.3.1, we see that *mfix f* structurally follows $f\ \bot$. That is, if *mfix f* is $\bot$ or [ ], then so is $f\ \bot$, and vice-versa. Similarly, *mfix f* is a cons-cell exactly when $f\ \bot$ is. We see this correspondence over and over in monads that are based on sum-like data structures. (See also Remark 2.6.4.)

Proposition 4.3.3 Equation 4.4 defines the unique value recursion operator for the *list* monad.

Proof Strictness is exactly the first equivalence in Proposition 4.3.1. Purity is easy to establish; we leave it to the reader. Left shrinking is more interesting. We show:

$$\mathit{mfix}\ (\lambda x.\ a \gg\!= \ \lambda y.\ f\ x\ y) = a \gg\!= \ \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y)$$

by structural induction on a. The base cases, $a = \bot$ and $a = []$, are immediate. For the inductive step, we assume $a = q : qs$, and reason as follows:

$$\begin{array}{lll}
 & \mathit{mfix}\ (\lambda x.\ (q : qs) \gg\!= \ \lambda y.\ f\ x\ y) & \\
= & \mathit{mfix}\ (\lambda x.\ f\ x\ q \mathbin{+\!\!+} qs \gg\!= \ \lambda y.\ f\ x\ y) & \\
= & \mathit{mfix}\ (\lambda x.\ f\ x\ q) \mathbin{+\!\!+} \mathit{mfix}\ (\lambda x.\ qs \gg\!= \ \lambda y.\ f\ x\ y) & \{\mathit{Eqn.}\ 4.10\} \\
= & \mathit{mfix}\ (\lambda x.\ f\ x\ q) \mathbin{+\!\!+} qs \gg\!= \ \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y) & \{\mathit{I.H.}\} \\
= & (\lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y))\ q \mathbin{+\!\!+} qs \gg\!= \ \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y) & \\
= & (q : qs) \gg\!= \ \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y) &
\end{array}$$

establishing the left shrinking property, and completing the proof that we have a legitimate value recursion operator.

For uniqueness, we will appeal to the approximation lemma.[2] Let *mfix* refer to the function defined by Equation 4.4, and let *mfix′* be another value recursion operator for the list monad. We will show that:

$$\forall n.\forall f.\ \mathit{approx}\ n\ (\mathit{mfix}\ f) = \mathit{approx}\ n\ (\mathit{mfix}'\ f)$$

establishing uniqueness. The proof is by induction on n. The base case $(n = 0)$ is immediate. The induction hypothesis is:

$$\forall f.\ \mathit{approx}\ k\ (\mathit{mfix}\ f) = \mathit{approx}\ k\ (\mathit{mfix}'\ f)$$

for a fixed natural number k. We need to show that:

$$\forall f.\ \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ f) = \mathit{approx}\ (k{+}1)\ (\mathit{mfix}'\ f)$$

Pick an arbitrary function f. The proof proceeds by case analysis on the value of $f\ \bot$. If $f\ \bot = \bot$, then both sides yield $\bot$ by the strictness property. If $f\ \bot = []$, then $f = \mathit{const}\ []$ by monotonicity, and both sides yield $[]$ by Proposition 2.6.1. The case when $f\ \bot$ is a cons-cell is a bit more involved. By monotonicity, we have

$$f\ x = (\mathit{head} \cdot f)\ x : (\mathit{tail} \cdot f)\ x = [(\mathit{head} \cdot f)\ x] \mathbin{+\!\!+} (\mathit{tail} \cdot f)\ x \tag{4.11}$$

for all x, since f will always produce a cons-cell given any argument. Furthermore, the *list* monad satisfies Equation 3.8, and hence *mfix′* must satisfy Equation 3.10 by Proposition 3.4.2, where $\oplus = \mathbin{+\!\!+}$. Now, it is easy to see that:

[2] See Appendix B.6 for a more detailed example use of this lemma.

$$\begin{aligned} \mathit{mfix}'\ f &= \mathit{mfix}'\ (\lambda x.\ [(\mathit{head} \cdot f)\ x] +\!\!+ (\mathit{tail} \cdot f)\ x) && \{\mathit{Eqn.}\ 4.11\} \\ &= \mathit{mfix}'\ (\mathit{return} \cdot \mathit{head} \cdot f) +\!\!+ \mathit{mfix}'\ (\mathit{tail} \cdot f) && \{\mathit{Eqn.}\ 3.10\} \\ &= \mathit{return}\ (\mathit{fix}\ (\mathit{head} \cdot f)) +\!\!+ \mathit{mfix}'\ (\mathit{tail} \cdot f) && \{\mathit{purity}\} \\ &= [\mathit{fix}\ (\mathit{head} \cdot f)] +\!\!+ \mathit{mfix}'\ (\mathit{tail} \cdot f) \\ &= \mathit{fix}\ (\mathit{head} \cdot f) : \mathit{mfix}'\ (\mathit{tail} \cdot f) \end{aligned}$$

Also note that Equation 4.4 will take its second branch when $f\ \bot$ is a cons-cell. Therefore, the proof obligation reduces to:

$$\begin{aligned} & \mathit{head}\ (\mathit{fix}\ (f \cdot \mathit{head})) : \mathit{approx}\ k\ (\mathit{mfix}\ (\mathit{tail} \cdot f)) \\ =\ & \mathit{fix}\ (\mathit{head} \cdot f) : \mathit{approx}\ k\ (\mathit{mfix}'\ (\mathit{tail} \cdot f)) \end{aligned}$$

by the definition of *approx*, and the above derivation. But this equation is immediate: First elements are equivalent by the dinaturality of *fix*, and the tails are equivalent by the induction hypothesis. □

Properties It is not very hard to show that the sliding and nesting properties hold. By the last equation in Proposition 4.3.1, distributivity holds as well (Property 3.4.1). On the negative side, both strong sliding and right shrinking properties fail, as pointed out in Corollary 3.1.7.

Remark 4.3.4 The *maybe* monad embeds into the *list* monad, as described in Section 3.5. Furthermore, the value recursion operator for the *maybe* monad is exactly the one predicted by Equation 3.14.

Remark 4.3.5 We close this section by discussing failing definitions of *mfix* for the *list* monad. Consider the function:

$$f\ xs = [\mathit{take}\ 3\ (1 : xs),\ \mathit{take}\ 3\ (2 : xs)] \tag{4.12}$$

What should *mfix f* be? Our definition yields: [[1, 1, 1], [2, 2, 2]], but the reader might wonder about [[1, 1, 1], [2, 1, 1]], or [[1, 2, 2], [2, 2, 2]], which are produced by the two alternatives we have seen at the beginning of this section, i.e., by pivoting over the first and second elements of the result. As we have mentioned, there is an infinite family of such operators:[3]

$$\mathit{mfix}_i\ f = \mathit{fix}\ (f \cdot \mathit{head} \cdot \mathit{tail}^i), \qquad i \geq 0 \tag{4.13}$$

[3] Note that these alternatives do *not* form a chain; they are all incomparable. Furthermore, they are all incomparable to our definition of *mfix* (i.e., Equation 4.4) as well.

How about properties? It is easy to see that strictness holds for all mfix_i, but that's where the good news ends. Except for mfix_0, all members violate purity. We have:

$$\mathit{mfix}_i\ (\mathit{return} \cdot f) = \mathit{return}\ (f\ \bot), \qquad i > 0$$

Furthermore, the left shrinking property fails for all of them. For instance,

$$\begin{aligned} \mathit{mfix}_0\ (\lambda x.\ [1,\ 2] \gg\!\!= \lambda y.\ [y,\ x]) &= [1,\ 1,\ 2,\ 1] \\ [1,\ 2] \gg\!\!= \lambda y.\ \mathit{mfix}_0\ (\lambda x.\ [y,\ x]) &= [1,\ 1,\ 2,\ 2] \end{aligned}$$

compromising the equivalence of **do** and **mdo** expressions in the absence of recursion. Intuitively, these definitions cause interference between elements. Note that:

$$\lambda x.\ [1,\ 2] \gg\!\!= \lambda y.\ [y,\ x] = \lambda x.\ [1,\ x,\ 2,\ x]$$

and there is no reason to expect anything but $\bot$ to play the role of x in the fixed-point, as there is no information on what it can be. Indeed, our definition of *mfix* yields:

$$\begin{aligned} \mathit{mfix}\ (\lambda x.\ [1,\ 2] \gg\!\!= \lambda y.\ [y,\ x]) &= [1,\ \bot,\ 2,\ \bot] \\ [1,\ 2] \gg\!\!= \lambda y.\ \mathit{mfix}\ (\lambda x.\ [y,\ x]) &= [1,\ \bot,\ 2,\ \bot] \end{aligned}$$

In the light of this discussion, we see that neither the list [[1, 1, 1], [2, 1, 1]], nor the list [[1, 2, 2], [2, 2, 2]] constitute a viable fixed-point for the function defined by Equation 4.12. Each indicate interference between the elements of the fixed-point, violating the left shrinking property.

In Section 9.1, we will see an example use of value recursion on the *list* monad, providing practical evidence for the definition given by Equation 4.4 being preferable over those given by Equation 4.13.

4.4 State

State monads capture the notion of computations that depend on modifiable stores, providing safe access to imperative features [51, 52]. A typical state monad, manipulating an internal state with type τ, has the following structure [7, 91]:

$$\textbf{type}\ \mathit{ST}\ \tau\ \alpha = \tau \to (\alpha,\ \tau)$$

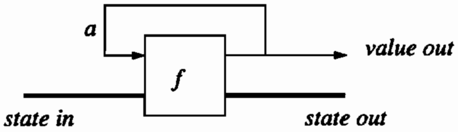
$$\begin{aligned} \mathit{return}\ x &= \lambda s.\ (x,\ s) \\ f \gg\!\!= g &= \lambda s.\ \textbf{let}\ (a,\ s') = f\ s \\ &\ \textbf{in}\ \ g\ a\ s' \end{aligned}$$

The corresponding value recursion operator is given by:

$$\begin{aligned} \mathit{mfix}_\omega \quad &:: \ (\alpha \to ST\ \tau\ \alpha) \to ST\ \tau\ \alpha \\ \mathit{mfix}_\omega\ f \ &= \ \lambda s.\ \mathbf{let}\ (a,\ s') = f\ a\ s \\ &\qquad\quad \mathbf{in}\ (a,\ s') \end{aligned} \tag{4.14}$$

(The reason for the name will be clear in a moment.)

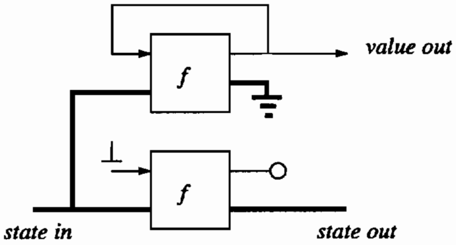
Remark 4.4.1 The following picture depicts the operation of the value recursion operator for the state monad, providing the intuition for the diagrams we have been using so far (see also Remark 2.2.2):



The monads we have considered up to now (i.e., identity, exceptions, and lists) enjoy the property that they all have unique value recursion operators. Is this the case for the state monad as well? Referring to the picture above, we see that the resulting state transformer is required to return the fixed-point value in the *value out* line in order to satisfy purity, but it is not clear how we should determine the final state, i.e., the value of the *state out* line. Equation 4.14 captures the case when *state out* is obtained by running f on the fixed-point value and the current state. It is possible to consider an alternative semantics, where the resulting state is determined without any regard to the value part, i.e., without any use of the fixed-point value. That is, a definition of the form:

$$\mathit{mfix}\ f \ = \ \lambda s.\ \mathbf{let}\ (a,\ \_) = f\ a\ s\ \mathbf{in}\ (a,\ \pi_2\ (f\ \perp\ s)) \tag{4.15}$$

with the following picture:



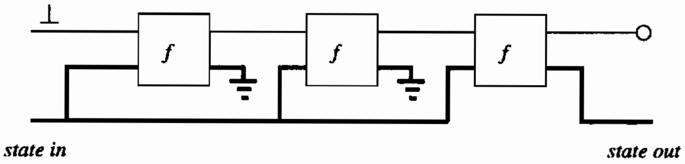
We might think of this operator as being strictly sequential in the state, i.e., it does not make use of any "future" knowledge in determining what the final state should be. There is a whole family of such operators, using approximations to the fixed-point value:

$$\mathit{mfix}_i\ f \ = \ \lambda s.\ \mathbf{let}\ (a,\ \_) = f\ a\ s\ \mathbf{in}\ (a,\ \mathit{pick}_i\ f\ s), \qquad i \geq 0 \tag{4.16}$$

where

$$pick_i\ f\ s = \pi_2\ (f\ ((\lambda a.\ \pi_1\ (f\ a\ s))^i\ \bot)\ s) \tag{4.17}$$

For instance, the picture for $pick_2$ is:



Note that $mfix_0$ is precisely the operator defined by Equation 4.15. By construction, each $pick_i$ is an approximation to the next, i.e., $pick_i \sqsubseteq pick_{i+1}$, implying $mfix_i \sqsubseteq mfix_{i+1}$. Furthermore, it is easy to see that:

$$mfix_\omega = \bigsqcup_{i=0}^{\infty} mfix_i \tag{4.18}$$

where the $mfix_\omega$ on the left hand side is the operator defined by Equation 4.14.

Example 4.4.2 The functions $mfix_i$, for all i, and $mfix_\omega$ will always agree on the value part of the fixed-point. It is the final state that will be approximated by each $mfix_i$, the limit being delivered by $mfix_\omega$. To demonstrate, consider the following function:

$$\begin{aligned} f &\ :: \ [Int] \to ST\ [Int]\ [Int] \\ f\ xs\ s &= (1 : xs,\ xs) \end{aligned}$$

We have:

$$\pi_2\ (mfix_i\ f\ [\,]) = \overbrace{1 : 1 : \ldots : 1}^{i\ times} : \bot$$

As expected, $\pi_2\ (mfix_\omega\ f\ [\,])$ yields the infinite list of 1's. Notice how approximations are reflected in the final state. (In all cases $\pi_1\ (mfix\ f\ [\,])$, i.e., the value part, will always be the infinite list of 1's.)

Proposition 4.4.3 The functions $mfix_i$, for all i (Equation 4.16), and $mfix_\omega$ (Equation 4.14) are value recursion operators for the state monad.

Proof For brevity, we will only consider $mfix_\omega$ here. Proofs for $mfix_i$ are a bit more tedious, but equally easy. For strictness, we note that a function f of type $\alpha \to ST\ \tau\ \alpha$ is strict exactly when $f\ \bot_\alpha\ s = (\bot_\alpha, \bot_\tau)$ for all s. We have:

$$\begin{aligned} mfix_\omega\ f\ s &= \textbf{let}\ (a,\ s') = f\ a\ s\ \textbf{in}\ (a,\ s') \\ &= \textbf{let}\ a = fix\ (\lambda a.\ \pi_1\ (f\ a\ s))\ \textbf{in}\ (a,\ \pi_2\ (f\ a\ s)) \end{aligned}$$

Because the function $\lambda a.\ \pi_1\ (f\ a\ s)$ is strict, its fixed-point is $\perp$. Therefore, $\mathit{mfix}_\omega\ f\ s = (\perp, \perp)$, establishing that $\mathit{mfix}_\omega\ f$ is $\perp$.[4]

For purity, we have:

$$\begin{aligned}
\mathit{mfix}_\omega\ (\mathit{return} \cdot f) &= \lambda s.\ \textbf{let}\ (a,\ s') = (\mathit{return} \cdot f)\ a\ s\ \textbf{in}\ (a,\ s') \\
&= \lambda s.\ \textbf{let}\ (a,\ s') = (f\ a,\ s)\ \textbf{in}\ (a,\ s') \\
&= \lambda s.\ \textbf{let}\ a = \mathit{fix}\ (\lambda a.\ f\ a)\ \textbf{in}\ (a,\ s) \\
&= \lambda s.\ (\mathit{fix}\ f,\ s) \\
&= \mathit{return}\ (\mathit{fix}\ f)
\end{aligned}$$

For left shrinking, we need to show that:

$$\mathit{mfix}_\omega\ (\lambda x.\ g \gg\!= \lambda y.\ f\ x\ y) = g \gg\!= \lambda y.\ \mathit{mfix}_\omega\ (\lambda x.\ f\ x\ y)$$

Simple symbolic manipulation reduces both sides to:

$$\begin{aligned}
\lambda s.\ \textbf{let}\ (a,\ s') &= g\ s \\
(a',\ s'') &= f\ a'\ a\ s' \\
\textbf{in}\ (a',\ s'')&
\end{aligned}$$

completing the proof. □

Remark 4.4.4 Abusing the terminology a bit, one might consider mfix_ω as a *lazy-in-the-state* value recursion operator, while mfix_0 is *strict*. As we will see in Section 4.8 and in Chapter 8 in detail, the operation of mfix_0 is quite similar to the operation of value recursion operators for stream processing and IO monads. It is hard to develop a corresponding intuition for mfix_i when $i \neq 0$. We do not know any applications that might benefit from them. Furthermore, they behave strangely with respect to the nesting property, as we will see shortly.

Properties Having established that mfix_i for all i, and mfix_ω are value recursion operators, we now take a look at other properties. It turns out that sliding (Property 2.4.1) is satisfied by all of them, but nesting (Property 2.5.1) only holds for mfix_0 and mfix_ω. Strong sliding and right shrinking properties only hold for mfix_ω.

Counterexample 4.4.5 Let us first consider nesting. Let

[4] We caution the reader about the use of true products. In case of lifted products, we would get $\mathit{mfix}\ f = \lambda s.\ (\perp,\ \perp) \neq \lambda s.\ \perp$, violating strictness. But this is hardly surprising—even monad laws fail in this case. It is easy to see that $(\lambda s.\ \perp) \gg\!= \mathit{return} = \lambda s.\ (\perp,\ \perp)$, failing the right unit law.

$$\begin{array}{lcl} f & :: & ([\mathit{Int}],\ [\mathit{Int}]) \to \mathit{ST}\ [\mathit{Int}]\ [\mathit{Int}] \\ f\ x\ s & = & (1 : \pi_1\ x,\ \pi_2\ x) \end{array}$$

Considering left and right hand sides of Equation 2.7, for each $i > 0$, we have:

$$\begin{array}{rcl} \pi_2\ (\mathit{mfix}_i\ (\lambda x.\ \mathit{mfix}_i\ (\lambda y.\ f\ (x,\ y)))\ [\,]) & = & 1^{i+1} : \bot \\ \pi_2\ (\mathit{mfix}_i\ (\lambda x.\ f\ (x,\ x))\ [\,]) & = & 1^i : \bot \end{array}$$

where 1^k denotes a list of k 1's. Since the final states differ, nesting fails. (The value part will be the infinite list of 1's in both cases.) For the single call to mfix_i in the second line, we simply get a snapshot of the value after i iterations, that is, exactly i 1's. The nested calls to mfix_i, and hence to pick_i, result in the extra 1 in the first line. This behavior is truly bizarre from the viewpoint of value recursion. In case of mfix_0, the final states will both be $\bot$, since the inner call to *pick* will be ignored by the outer one. In case of mfix_ω, the final state will be the infinite list of 1's, as expected.

For strong sliding (Section 2.7.1), consider:

$$\begin{array}{lcl} f & :: & [\mathit{Int}] \to \mathit{ST}\ [\mathit{Int}]\ [\mathit{Int}] \\ f\ \mathit{xs}\ s & = & (\mathit{xs},\ \mathit{xs}) \end{array} \qquad \begin{array}{lcl} h & :: & [\mathit{Int}] \to [\mathit{Int}] \\ h\ \mathit{xs} & = & 1 : \mathit{xs} \end{array}$$

Note that $f\ (h\ \bot) = \lambda s.(1 : \bot,\ 1 : \bot) \neq \lambda s.(\bot, \bot) = f\ \bot$, hence sliding (Property 2.4.1) does not apply. Considering Equation 2.5, we have:

$$\begin{array}{rcl} \pi_2\ (\mathit{mfix}_0\ (\mathit{map}\ h\ \cdot\ f)\ [\,]) & = & \bot \\ \pi_2\ (\mathit{map}\ h\ \ (\mathit{mfix}_0\ (f\ \cdot\ h))\ \ [\,]) & = & 1 : \bot \end{array}$$

showing that strong sliding fails. For right shrinking (Property 2.7.3), let

$$\begin{array}{lcl} f & :: & [\mathit{Int}] \to \mathit{ST}\ [\mathit{Int}]\ [\mathit{Int}] \\ f\ \mathit{xs}\ s & = & (1{:}\mathit{xs},\ \mathit{xs}) \end{array} \qquad \begin{array}{lcl} g & :: & [\mathit{Int}] \to \mathit{ST}\ [\mathit{Int}]\ [\mathit{Int}] \\ g\ \mathit{xs}\ (\_ : k : \_) & = & (\mathit{xs},\ [k]) \end{array}$$

We leave it to the reader to show that right shrinking fails for mfix_0 with this instantiation. It is possible to generalize these examples for all other mfix_i, whenever i is finite.

Remark 4.4.6 We do not know whether there are other value recursion operators for the state monad.

4.5 Output monad and monads based on monoids

Every monoid gives rise to a monad, referred to as its *representation monad* [2]. In programming, the best known example is the output monad, as we will see shortly. Let

$(M, \oplus, \mathit{unit})$ be a monoid, where M is the underlying type. The corresponding representation monad is given by:

$$
\begin{aligned}
&\textbf{type } \mathit{RepM}\ \alpha = (\alpha,\ M)\\[1em]
&\mathit{return}\ x \ = (x,\ \mathit{unit})\\
&\mathit{ma} \gg\!= \ f = \textbf{let } (a,\ m) = \mathit{ma}\\
&\qquad\qquad\qquad\quad (b,\ n) = f\ a\\
&\qquad\qquad\quad \textbf{in } (b,\ m \oplus n)
\end{aligned}
$$

For instance, substituting *String* for M, “” for *unit*, and ++ for $\oplus$, one obtains the usual output monad [7, 91]. The obvious value recursion operator is given by:

$$
\begin{aligned}
\mathit{mfix}_\omega &:: (\alpha \to \mathit{RepM}\ \alpha) \to \mathit{RepM}\ \alpha\\
\mathit{mfix}_\omega\ f &= \textbf{let } (a,\ m) = f\ a \textbf{ in } (a,\ m) \qquad (4.19)
\end{aligned}
$$

As with the state monad, the choice of the name mfix_ω is not arbitrary. We have a family of recursion operators:

$$\mathit{mfix}_i\ f = \textbf{let } (a,\ \_) = f\ a \textbf{ in } (a,\ \mathit{pick}_i\ f), \qquad i \geq 0 \qquad (4.20)$$

where

$$\mathit{pick}_i\ f = \pi_2\ (f\ ((\pi_1 \cdot f)^i\ \bot)) \qquad (4.21)$$

A straightforward calculation (analogous to Equation 4.18) shows that:

$$\mathit{mfix}_\omega = \bigsqcup_{i=0}^{\infty} \mathit{mfix}_i \qquad (4.22)$$

The correspondence with the state monad is not accidental. Any such representation monad embeds into the state monad via the embedding:

$$\epsilon\ (a,\ m) = \lambda n.\ (a,\ n \oplus m) \qquad (4.23)$$

with the left inverse: $\epsilon^\ell\ f\ =\ f\ \mathit{unit}$. Furthermore, ϵ works uniformly over all value recursion operators, including mfix_ω. That is, for any monoid M:

$$\epsilon\ (\mathit{mfix}_i^{\mathit{RepM}}\ f) = \mathit{mfix}_i^{ST}\ (\epsilon \cdot f) \qquad (4.24)$$

where i is either a natural number or ω. It is an easy exercise to show that the embedding requirements (i.e., Equations 3.11-3.13) hold for ϵ.

Properties By Proposition 3.5.3, whenever an *mfix* for the state monad satisfies purity or left shrinking, the corresponding operator for the representation monad of a given monoid will satisfy it too. Note that ϵ is not strict, hence strictness is not automatically guaranteed (see Remark 3.5.4). However, it is easy to see that all mfix_i and mfix_ω satisfy strictness. Therefore, we have an infinite family of value recursion operators for representation monads, similar to the case for the state monad.

By Proposition 3.5.5, sliding, nesting, strong sliding, and right shrinking properties hold whenever the corresponding operator for the state monad satisfies them. On the negative side, all of the counterexamples we gave for the state monad can be converted to counterexamples for representation monads with no difficulty, invalidating nesting for mfix_i when $i > 0$, and strong sliding and right shrinking for all but mfix_ω.

If the underlying monoid is idempotent, the representation monad will be idempotent as well. Similarly, commutativity of the monoid implies the commutativity of the monad. In both cases, mfix_ω will preserve idempotency and commutativity (Properties 3.2.1 and 3.3.1). Unfortunately, this result does not extend to mfix_i automatically.[5]

Remark 4.5.1 Similar to the case for the state monad, it is an open question whether there are other value recursion operators for monads based on monoids.

4.6 Environments

The environment monad, also known as the reader monad, captures computations that use a store to read values without modifying them. Using an environment of type ρ, the environment monad has the following structure:

$$\textbf{type } \mathit{Env}\ \rho\ \alpha = \ \rho \to \alpha$$

$$\begin{aligned} \mathit{return}\ x &= \lambda e.\ x \\ f \gg\!= \ g &= \lambda e.\ g\ (f\ e)\ e \end{aligned}$$

The corresponding value recursion operator is given by:

$$\begin{aligned} \mathit{mfix} \ &:: (\alpha \to \mathit{Env}\ \rho\ \alpha) \to \mathit{Env}\ \rho\ \alpha \\ \mathit{mfix}\ f &= \lambda e.\ \textbf{let}\ a = f\ a\ e \\ &\qquad\quad \textbf{in}\ a \end{aligned} \tag{4.25}$$

[5] For instance, Equation 3.4 will hold for mfix_0 only when $\pi_2\ (f\ \bot) \oplus \pi_2\ (f\ (\pi_1\ (f\ \bot))) = \pi_2\ (f\ \bot)$, which is *not* guaranteed just by the fact that $\oplus$ is idempotent. Similar arguments apply to Equations 3.5 and 3.7 as well.

Remark 4.6.1 It is an easy exercise to show that Equation 4.25 is equivalent to the generic *mfix* given in Section 1.4. To the best of our knowledge, identity and environment monads are the only examples where the generic version acts as the value recursion operator.

Unsurprisingly, the environment monad embeds into the state monad. The embedding[6] is given by $\epsilon\ f = \lambda s.\ (f\ s,\ s)$, with the left inverse $\epsilon^{\ell}\ f = \pi_1 \cdot f$. It is easy to see that strictness holds for Equation 4.25. Therefore, Proposition 3.5.3 guarantees that Equation 4.25 defines a value recursion operator for the environment monad.

Properties By Proposition 3.5.5 and the observations made above, Equation 4.25 satisfies all the properties satisfied by mfix_{ω} of the state monad. That is, sliding, nesting, strong sliding, and right shrinking properties, along with the basic requirements of strictness, purity and left shrinking are all satisfied.

Finally, the environment monad is both idempotent and commutative, and Properties 3.2.1 and 3.3.1 are both satisfied.

Remark 4.6.2 We do not know whether Equation 4.25 defines the unique value recursion operator for the environment monad.

4.7 Tree monad

In this section, we will briefly cover the *tree* monad [42]:

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf}\ \alpha\ \mid\ \mathit{Fork}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)$$

$$\begin{array}{lll}
\mathit{return}\ x & & = \mathit{Leaf}\ x \\
\mathit{Leaf}\ x & \gg\!= & f = f\ x \\
\mathit{Fork}\ l\ r & \gg\!= & f = \mathit{Fork}\ (l \gg\!= f)\ (r \gg\!= f)
\end{array}$$

The effect of $\gg\!=$ is to splice new subtrees on every *Leaf* of the first argument. The corresponding value recursion operator is given by:

$$\begin{array}{lll}
\mathit{mfix} & :: & (\alpha \to \mathit{Tree}\ \alpha) \to \mathit{Tree}\ \alpha \\
\mathit{mfix}\ f & = & \textbf{case}\ \mathit{fix}\ (f \cdot \mathit{unL})\ \textbf{of} \\
& & \quad \mathit{Leaf}\ x \ \to \mathit{Leaf}\ x \\
& & \quad \mathit{Fork}\ \_\ \_ \to \mathit{Fork}\ (\mathit{mfix}\ (\mathit{lc} \cdot f))\ (\mathit{mfix}\ (\mathit{rc} \cdot f))
\end{array} \tag{4.26}$$

The functions *unL*, *lc*, and *rc* are defined as follows:

[6] It does not matter which *mfix* is chosen for the state monad (i.e., mfix_{ω} of Equation 4.14, or any mfix_i given by Equation 4.16).

$$\begin{aligned} &\mathit{unL} && :: \mathit{Tree}\ \alpha \to \alpha \\ &\mathit{unL}\ (\mathit{Leaf}\ x) && = x \end{aligned} \qquad \begin{aligned} &\mathit{lc}, \mathit{rc} && :: \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha \\ &\mathit{lc}\ \ (\mathit{Fork}\ l\ r) && = l \\ &\mathit{rc}\ \ (\mathit{Fork}\ l\ r) && = r \end{aligned}$$

Compared to the value recursion operator for the *list* monad (Equation 4.4), we see that *unL* plays the role of *head*, while *tail* is replaced by *lc* and *rc*, projecting out the children at each node. Otherwise, the definitions are structurally the same.

Remark 4.7.1 Despite all the similarities, the *list* monad does not embed into the *tree* monad. There is no suitable element to map [] to, since our trees are always non-empty. (An alternative formulation of trees, where data is stored in the nodes and leaves are empty, does not give rise to a monad structure.)

Proposition 4.7.2 The function *mfix* given by Equation 4.26 satisfies:

$$\begin{aligned} \mathit{mfix}\ f = \bot \ &\Leftrightarrow\ f\ \bot = \bot && (4.27) \\ \mathit{unL}\ (\mathit{mfix}\ f) \ &=\ \mathit{fix}\ (\mathit{unL} \cdot f) && (4.28) \\ \mathit{lc}\ (\mathit{mfix}\ f) \ &=\ \mathit{mfix}\ (\mathit{lc} \cdot f) && (4.29) \\ \mathit{rc}\ (\mathit{mfix}\ f) \ &=\ \mathit{mfix}\ (\mathit{rc} \cdot f) && (4.30) \\ \mathit{mfix}\ (\lambda x.\ \mathit{Fork}\ (f\ x)\ (g\ x)) \ &=\ \mathit{Fork}\ (\mathit{mfix}\ f)\ (\mathit{mfix}\ g) && (4.31) \end{aligned}$$

Proof Similar to the proof of Proposition 4.3.1. □

Proposition 4.7.3 Equation 4.26 defines the unique value recursion operator for the *tree* monad.

Proof Analogous to the proof for the *list* monad (Proposition 4.3.3). Note that we need to use a different version of *approx* that works on trees [38] (see Appendix B.6). For uniqueness, we cannot refer to distributivity, as the *tree* monad is not additive. (There is no appropriate *unit* element.) However, we still have the operator: $x \oplus y = \mathit{Fork}\ x\ y$, which satisfies:

$$x \oplus y \gg\!= f = x \gg\!= f \oplus y \gg\!= f$$

hence a similar argument applies as in the case for the *list* monad. We leave the details to the reader. □

Properties Sliding and nesting properties can be shown to hold for the *tree* monad, while strong sliding and right shrinking fails by Propositions 3.1.5 and 3.1.6.

4.8 Fudgets

In this section, we will take a look at *fudgets*,[7] a monad that has been designed to model stream based computations. In its simplest form, the *fudgets* monad looks like:

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Fudget}\ \alpha & = & \mathit{Val}\ \alpha \\
 & \mid & \mathit{Put}\ \mathit{Char}\ (\mathit{Fudget}\ \alpha) \\
 & \mid & \mathit{Get}\ (\mathit{Char} \to \mathit{Fudget}\ \alpha)
\end{array}$$

$$\mathit{return} = \mathit{Val}$$

$$\begin{array}{lll}
\mathit{Val}\ a & \gg\!= f & = f\ a \\
\mathit{Put}\ c\ m & \gg\!= f & = \mathit{Put}\ c\ (m \gg\!= f) \\
\mathit{Get}\ h & \gg\!= f & = \mathit{Get}\ (\lambda c.\ h\ c \gg\!= f)
\end{array}$$

We will model functional I/O using a simple interpreter over this data type:[8]

$$\begin{array}{lll}
\mathit{run} & & :: \mathit{Fudget}\ \alpha \to \mathit{String} \to (\mathit{String},\ \alpha,\ \mathit{String}) \\
\mathit{run}\ (\mathit{Val}\ a) & \mathit{inp} & = (\text{``''},\ a,\ \mathit{inp}) \\
\mathit{run}\ (\mathit{Put}\ c\ m) & \mathit{inp} & = \textbf{let}\ (o,\ a,\ r) = \mathit{run}\ m\ \mathit{inp}\ \textbf{in}\ (\text{'!'} : c : o,\ a,\ r) \\
\mathit{run}\ (\mathit{Get}\ f) & [\,] & = \mathit{error}\ \text{``}\texttt{trying to Get from an empty stream!}\text{''} \\
\mathit{run}\ (\mathit{Get}\ f) & (c{:}cs) & = \textbf{let}\ (o,\ a,\ r) = \mathit{run}\ (f\ c)\ cs\ \textbf{in}\ (\text{'?'} : c : o,\ a,\ r)
\end{array}$$

The function *run* accepts a fudget and an input stream, runs the computation and delivers the list of I/O operations that took place, together with the final value and the remainder of the input. The list of operations consists of all characters that are printed via *Put* (prefixed by !), and all characters that are read from the input via *Get* (prefixed by ?). Note that the order is important, as it indicates the temporal relationship between I/O actions. For instance, we have:

$$\mathit{run}\ (\mathit{Put}\ \text{'a'}\ (\mathit{Get}\ (\lambda c.\ \mathit{Put}\ c\ (\mathit{Val}\ c))))\ \text{``123''} \;=\; (\text{``!a?1!1''},\text{'1'},\text{``23''})$$

For value recursion, we are interested in the meanings of fudgets of the form:

$$\mathit{mfix}\ (\lambda xs.\ \mathit{Put}\ \text{'a'}\ (\mathit{Val}\ (1 : xs))) \tag{4.32}$$

[7] It would me more appropriate to call these "fudget-style stream processor monads," as the presentation here is only loosely based on the original work on fudgets by Carlsson and Hallgren [28, 29]. For brevity, however, we will continue using the word *fudget*.

[8] We will investigate Haskell's internal IO monad in detail in Chapter 8.

which intuitively models a computation that will print the character a and then deliver an infinite list of 1's. Or, more interestingly:

$$mfix\ (\lambda cs.\ Get\ (\lambda c.\ Val\ (c : cs))) \tag{4.33}$$

which will first read a character from the input stream (if available), and then return an infinite list containing copies of that character.

One possible value recursion operator for the *fudgets* monad is given by:

$$\begin{aligned} mfix\ f\ =\ &\textbf{case}\ f\ \bot\ \textbf{of} \\ &\quad Val\ \_ \quad\ \ \rightarrow fix\ (f \cdot unVal) \\ &\quad Put\ c\ \_ \rightarrow Put\ c\ (mfix\ (unPut \cdot f)) \\ &\quad Get\ \_ \quad\ \rightarrow Get\ (\lambda c.\ mfix\ (unGet\ c \cdot f)) \end{aligned} \tag{4.34}$$

where

$$\begin{aligned} unVal\ (Val\ a) &= a \\ unPut\ (Put\ \_\ m) &= m \\ unGet\ c\ (Get\ h) &= h\ c \end{aligned}$$

With this definition, Expression 4.32 yields:

$$run\ (mfix\ (\lambda xs.\ Put\ \text{'a'}\ (Val\ (1 : xs))))\ \text{“z”}\ = (\text{“!a”},\ \bar{1},\ \text{“z”})$$

where $\bar{1}$ denotes the infinite list of 1's. The result indicates that there was one I/O action, which was printing the character 'a', and no input was consumed. Expression 4.33 yields:

$$run\ (mfix\ (\lambda cs.\ Get\ (\lambda c.\ Val\ (c : cs))))\ \text{“z”}\ = (\text{“?z”},\ \overline{\text{'z'}},\ \text{“ ”})$$

indicating that the character 'z' is read from the input, the infinite list of z's are returned, and all of the input was consumed. If the input stream was empty to start with, we would end up with the *error* case, i.e., the result would be undefined.

So far, the behavior of *mfix* seems to be consistent with the way we perceive I/O. Here is a slightly more challenging expression:

$$mfix\ (\lambda c.\ Put\ c\ (Val\ \text{'a'})) \tag{4.35}$$

What should the result be? Two possibilities arise. If we consider *Put* as an action causing I/O, we see that it will not have its character ready for printing until after the computation proceeds. That is, we should have:

$$run\ (mfix\ (\lambda c.\ Put\ c\ (Val\ \text{'a'})))\ \text{“ ”}\ = (\text{“!}\bot\text{”},\ \text{'a'},\ \text{“ ”})$$

leaving the printed character undefined.[9] Another option is to make the fixed-point value available throughout the whole computation, yielding:[10]

$$\mathit{run}\ (\mathit{mfix}\ (\lambda c.\ \mathit{Put}\ c\ (\mathit{Val}\ \text{'a'})))\ \text{“”}\ = (\text{“!a”},\ \text{'a'},\ \text{“”})$$

However, this alternative behavior is quite questionable. Consider the expression:

$$\mathit{mfix}\ (\lambda c.\ \mathit{Put}\ c\ (\mathit{Get}\ (\lambda d.\ \mathit{Val}\ d)))$$

In this case, we have to look past *Get* to determine what *Put* should print. However, this character is simply not available until we *run* this fudget with a particular input stream. Such an operator would violate the temporal relationship between *Put* and *Get*. (Furthermore, to achieve this effect, one would need to combine the operation of *run* and *mfix*, making the input stream available when the fixed-point is computed.)

Proposition 4.8.1 Equation 4.34 defines a value recursion operator for fudgets.

Proof Strictness and purity are immediate. Left shrinking can be established by induction. (As discussed briefly above, uniqueness is not guaranteed as we can speculate on the character to be printed whenever we have a *Put* constructor.) □

Properties Strong sliding and right shrinking both fail by Propositions 3.1.5 and 3.1.6. Although we have not constructed the proofs, we believe that sliding and nesting properties should hold.

4.9 Monad transformers

As pointed out in Section 3.6, monad transformers allow construction of new monads from old ones. Although there is no magic recipe that will automatically lift a given *mfix* through a transformer, it is possible to do so in many practical cases. In this section, we will study three of the most common instances, namely error, environment, and state monad transformers. (For a discussion of the continuation monad transformer, see Section 5.2.)

Liang defines the error monad transformer as follows [53]:

[9] The *mfix* we have given in Equation 4.34 produces this answer. As we will see in Chapter 8, the function *fixIO*, the value recursion operator for Haskell's IO monad, behaves similarly. (See Example 8.2.2.)

[10] Ignoring the *Get* constructor, the fudgets monad is very similar to the *output* monad of Section 4.5. The second alternative corresponds to the function mfix_ω (Eqn. 4.19), while the first one corresponds to mfix_0 (Eqn. 4.20). It is possible to think of operators that correspond to mfix_i when $i \neq 0$ too.

$$\begin{array}{l}
\mathbf{data}\ \mathit{Err}\ \alpha \qquad = \mathit{Ok}\ \alpha \mid \mathit{Err}\ \mathit{String} \\
\mathbf{type}\ \mathit{ErrT}\ m\ \alpha = m\ (\mathit{Err}\ \alpha) \\
\\
\mathit{return}\ a \ \ = \mathit{return}\ (\mathit{Ok}\ a) \\
m \gg\!= \ k = m \gg\!= \ \lambda a.\ \mathbf{case}\ a\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Ok}\ x \ \rightarrow k\ x \\
\qquad\qquad\qquad\qquad\qquad \mathit{Err}\ s \rightarrow \mathit{return}\ (\mathit{Err}\ s) \\
\\
\mathit{lift}\ m = m \gg\!= \ \lambda a.\ \mathit{return}\ (\mathit{Ok}\ a)
\end{array}$$

Note that the *return* and $\gg\!=$ on the left hand side are the definitions for the new monad *Err m*, while those on the right belong to the monad m. If m has a value recursion operator *mfixM*, we can lift it up to *Err m* as follows:

$$\begin{array}{l}
\mathit{mfixErrM} \quad :: (\alpha \rightarrow \mathit{ErrT}\ m\ \alpha) \rightarrow \mathit{ErrT}\ m\ \alpha \\
\mathit{mfixErrM}\ f \ = \mathit{mfixM}\ (f\ \cdot\ \mathit{unErr}) \\
\qquad\qquad\qquad \mathbf{where}\ \mathit{unErr}\ (\mathit{Ok}\ a) = a
\end{array} \tag{4.36}$$

The similarities between Equations 4.36 and 4.2 are not accidental. The function *unErr* plays the same role as *unJust*, it providing access to the value part of the computation. While the value recursion operator for the *maybe* monad uses *fix* (i.e., the value recursion operator for the identity monad) to tie the recursive knot, *mfixErrM* uses the value recursion operator for the underlying monad to do so.

Proposition 4.9.1 Let *mfixM* be a value recursion operator for a given monad m. The function *mfixErrM*, defined by Equation 4.36, is a value recursion operator for the monad *ErrT m*.

Proof See Appendix B.7. □

Let us now consider the environment monad transformer, which adds an immutable store to arbitrary monads. The definitions for the environment monad transformer are [53]:

$$\begin{array}{l}
\mathbf{type}\ \mathit{EnvT}\ \rho\ m\ \alpha = \rho \rightarrow m\ \alpha \\
\\
\mathit{return}\ a \ \ = \lambda e.\ \mathit{return}\ a \\
m \gg\!= \ k = \lambda e.\ m\ e \gg\!= \ \lambda a.\ k\ a\ e \\
\\
\mathit{lift}\ m = \lambda e.\ m
\end{array}$$

If the underlying monad has a value recursion operator *mfixM*, we can lift it to the transformed monad as follows:

$$\begin{array}{l}
\mathit{mfixEnvM} \quad :: (\alpha \rightarrow \mathit{EnvT}\ \rho\ m\ \alpha) \rightarrow \mathit{EnvT}\ \rho\ m\ \alpha \\
\mathit{mfixEnvM}\ f \ = \lambda e.\ \mathit{mfixM}\ (\lambda a.\ f\ a\ e)
\end{array} \tag{4.37}$$

The definition of *mfixEnvM* exactly mimics the value recursion operator for the environment monad (Equation 4.25), just like the case for the error monad transformer and the *maybe* monad. Analogous to Propisition 4.9.1, we have:

Proposition 4.9.2 Let *mfixM* be a value recursion operator for a given monad m. The function *mfixEnvM*, defined by Equation 4.37, is a value recursion operator for the monad *EnvT* ρ m. □

Finally, we consider the state monad transformer [53]:

$$\textbf{type}\ \mathit{StateT}\ \sigma\ m\ \alpha = \sigma \to m\ (\alpha,\ \sigma)$$

$$\begin{aligned} \mathit{return}\ a\ &= \lambda s.\ \mathit{return}\ (a,\ s) \\ m \gg\!= \ k &= \lambda s.\ m\ s \gg\!= \ \lambda(a,\ s').\ k\ a\ s' \end{aligned}$$

$$\mathit{lift}\ m = \lambda s.\ m \gg\!= \ \lambda x.\ \mathit{return}\ (x,\ s)$$

Applying the pattern we have seen with the previous two examples, a given *mfixM* can be lifted through the state monad transformer as follows:

$$\begin{aligned} \mathit{mfixStateM}\ &:: (\alpha \to \mathit{StateT}\ \sigma\ m\ \alpha) \to \mathit{StateT}\ \sigma\ m\ \alpha \\ \mathit{mfixStateM}\ f\ &= \lambda s.\ \mathit{mfixM}\ (\lambda r.\ f\ (\pi_1\ r)\ s) \end{aligned} \tag{4.38}$$

Proposition 4.9.3 Let *mfixM* be a value recursion operator for a given monad m. The function *mfixStateM*, defined by Equation 4.38, is a value recursion operator for the monad *StateT* σ m. □

Remark 4.9.4 The lifting given by Equation 4.38 behaves analogously to mfix_ω as given by Equation 4.14. It does not seem possible to lift arbitrary value recursion operators so that they will behave similarly to any of the mfix_i where i is finite (Equation 4.16).

4.10 Summary

In this chapter we have considered a wide range of monads and value recursion operators for them. Although there is no magic recipe to automate the process, the examples provide sufficient detail to guide the construction of value recursion operators for new monads.

There is one notable exception, however. The continuation monad does not seem to possess a value recursion operator. Chapter 5 contains the details.

We summarize the properties of value recursion operators we have studied in this chapter in the following table, along with the IO monad (studied in Chapter 8). The last column indicates whether the corresponding value recursion operator is unique. A cell marked with * indicates a conjecture.

		Str.	Pure	Left	Slide	Nest	S. Slide	Right	Unique
Identity		✓	✓	✓	✓	✓	✓	✓	✓
Exceptions		✓	✓	✓	✓	✓	✗	✗	✓
Lists		✓	✓	✓	✓	✓	✗	✗	✓
State	mfix_0	✓	✓	✓	✓	✓	✗	✗	✗
	mfix_i	✓	✓	✓	✓	✗	✗	✗	
	mfix_ω	✓	✓	✓	✓	✓	✓	✓	
Monoids	mfix_0	✓	✓	✓	✓	✓	✗	✗	✗
	mfix_i	✓	✓	✓	✓	✗	✗	✗	
	mfix_ω	✓	✓	✓	✓	✓	✓	✓	
Environment		✓	✓	✓	✓	✓	✓	✓	✓*
Tree		✓	✓	✓	✓	✓	✗	✗	✓
Fudgets		✓	✓	✓	✓*	✓*	✗	✗	✗
IO		✓	✓	✓	✓*	✓*	✗	✗	✓*

Let us conclude this chapter by making several observations about value recursion operators:

- We might hope that *mfix* constructs a fixed point value in the process of computation. Unfortunately, in general, we cannot expect to find a value z_f such that $\mathit{mfix}\ f\ =\ f\ z_f$. Consider the function $f\ xs\ =\ [1\ :\ xs,\ 2\ :\ xs]$. There is no integer value z_f such that $f\ z_f\ =\ [1\ :\ 1\ :\ ...,\ 2\ :\ 2\ :\ ...]$, which is the required result in this case. Similarly, in the case of the state monad, the closest we can get is: $\lambda s.\ f\ (\mathit{fix}\ (\lambda a.\ \pi_1\ (f\ a\ s)))\ s$, which shows that the state in which the recursive computation gets performed is essential in determining the final result. Similar comments apply to the expression $\mathit{mfix}\ f\ =\ m_f \gg\!= \ f$ as well.

- Similarly, one might hope for a morphism *suppress* $::\ m\ \alpha \rightarrow m\ \alpha$, such that

  $$\mathit{mfix}\ f\ =\ \mathit{suppress}\ (\mathit{mfix}\ f) \gg\!=\ f$$

  The aim of *suppress* is to strip out effects. There are some monads for which such a morphism is available, but not in general. For instance, for the state monad:

  $$\mathit{suppress}\ f\ =\ \lambda s.\ \mathbf{let}\ (a,\ \_)\ =\ f\ s\ \mathbf{in}\ (a,\ s)$$

Intuitively, *suppress* can only exist when there is a clear structural separation between values and effects. For instance, such a separation seems impossible for the *maybe* or *list* monads

- The equality $\mathit{mfix}\ f \gg g = f\ \bot \gg g$ does *not* hold in general. Since the value produced by *mfix f* is discarded, one might think that the recursive computation may be skipped as well. However, g might depend on the effects performed by the first computation, which might very well be different for *mfix f* and $f\ \bot$.

- It is worth reemphasizing some differences between *fix* and *mfix*. Recall that *fix* satisfies the equality $\mathit{fix}\ (f \cdot f) = \mathit{fix}\ f$, for all f. However, it is *not* the case that: $\mathit{mfix}\ f = \mathit{mfix}\ (\lambda x.\ f\ x \gg= f)$, unless f is pure. In general, this equation only holds when the underlying monad is idempotent (see Section 3.2). Similarly, the equation $\mathit{fix}\ (f \cdot h) = f\ (\mathit{fix}\ (h \cdot f))$ translates into

$$\mathit{mfix}\ (\mathit{map}\ h \cdot f) = \mathit{map}\ h\ (\mathit{mfix}\ (f \cdot h))$$

and requires $f\ \bot = f\ (h\ \bot)$ (see Section 2.4). Most importantly, the defining equation for *fix*, $\mathit{fix}\ f = f\ (\mathit{fix}\ f)$, simply does not have any counterpart in the value recursion world. The unfolding view of recursion is not suitable for explaining value recursion except for very *mild* effects (such as identity and environments), as it does not distinguish between values and effects at all.

Chapter 5

Continuations and value recursion

Is there a value recursion operator for the continuation monad? Originally designed to model jumps, continuations come close to being the "universal" monad [24], and their interaction with recursion proves to be quite intricate. In this chapter, we will take a closer look at the structure of continuations from the viewpoint of value recursion.

Synopsis. We start with a review of the continuation monad, and continue by showing that a value recursion operator for continuations is highly unlikely to exist. After a brief discussion of the continuation monad transformer, we turn to first-class continuations, as found in Standard-ML and Scheme languages. We explore the interaction between recursive binding constructs and first-class continuations, showing that the left shrinking property is unattainable in such a setting.

5.1 A monad for continuations

Traditionally, continuation-passing style (CPS) has been used to model jumps in programming languages [90]. Continuations provide an extremely powerful effect, especially first-class continuations as supported by SML of New Jersey and Scheme [31, 44], hence effective use of continuations require great care: As demonstrated by Thielecke, many seemingly obvious equivalences fail to hold in the presence of a call-by-current-continuation construct [84]. We will see a particular example related to recursion in Section 5.3.

Computations based on CPS can be described using monads. Wadler discusses monads for continuations in a typed setting [90], while Espinosa's thesis contains a discussion in the untyped world [22]. A typical continuation monad has the following structure:

$$\mathbf{type}\ \mathit{Cont}\ \sigma\ \tau = (\tau \to \sigma) \to \sigma$$

$$\begin{aligned} \mathit{return}\ x &= \lambda k.\ k\ x \\ m \gg\!= h &= \lambda k.\ m\ (\lambda v.\ h\ v\ k) \end{aligned}$$

The type variable σ encodes the result type. For any type σ, continuation-based computations with a result value of type of σ are modeled by the monad *Cont* σ. Other operations on continuations include *run*, which provides an initial continuation; *abort*, which ignores its continuation and immediately returns a result; and *callcc*, which enables saving the current continuation for later use:

$$\begin{array}{llll} \mathit{run} & :: \mathit{Cont}\ \sigma\ \sigma \to \sigma \qquad\qquad & \mathit{abort} & :: \sigma \to \mathit{Cont}\ \sigma\ \sigma \\ \mathit{run}\ m & =\ m\ \mathit{id} & \mathit{abort}\ e & =\ \lambda k.\ e \end{array}$$

$$\begin{array}{ll} \mathit{callcc} & :: ((\tau \to \mathit{Cont}\ \sigma\ \tau) \to \mathit{Cont}\ \sigma\ \tau) \to \mathit{Cont}\ \sigma\ \tau \\ \mathit{callcc}\ h & =\ \lambda k.\ h\ (\lambda v.\ (\lambda c.\ k\ v))\ k \end{array}$$

It is worth noting that *run* takes continuations of type *Cont* σ σ, i.e., the argument and the result types are the same. (Similarly, the result of *abort* is also restricted.) In *callcc*, the function h is given a handle to the current continuation k. If h uses its first argument, the control will be transferred to the point where *callcc* h was originally invoked. Note that the inner argument, $\lambda c.\ k\ v$, ignores its own continuation c, transferring the control back to k. Otherwise h might ignore its first argument, proceeding normally.

Let us now turn to the question of value recursion for the continuation monad. Recall that a value recursion operator has type $(\alpha \to m\ \alpha) \to m\ \alpha$, where m is the underlying monad. Expanding this type for continuations, we get

$$\mathit{mfix}\ ::\ (\alpha \to (\alpha \to \sigma) \to \sigma) \to (\alpha \to \sigma) \to \sigma \tag{5.1}$$

where σ is the type of answers. Following the general pattern for value recursion, we need to perform the fixed-point computation over α. However, it is simply not possible to obtain a plausible value of type α by only using the arguments to *mfix*. Indeed, we were not able to produce a plausible definition of *mfix* of even the correct type for the continuation monad, let alone a definition that would satisfy the required properties.

Let us explore the situation a bit more closely. Being explicit about the quantification, we can rewrite Type 5.1 as:

$$\forall\sigma.\forall\alpha.(\alpha \to (\alpha \to \sigma) \to \sigma) \to (\alpha \to \sigma) \to \sigma \tag{5.2}$$

What are the inhabitants of this type? Fixing an answer type σ, we see that the Type 5.2 is isomorphic to:

$$\forall\alpha.((\alpha \to \sigma) \to \alpha \to \sigma) \to (\alpha \to \sigma) \to \sigma \tag{5.3}$$

and it is not hard to see that this type is (infinitely) inhabited if we have a fixed-point operator. Each one of the following cases forms a class of inhabitants:

$$mfix'\ f\ k = \begin{cases} f^i\ (const\ v)\ \bot_\alpha, & i \geq 0,\ v \in \sigma \\ f^i\ k\ \bot_\alpha, & i \geq 0 \\ (fix\ f)\ \bot_\alpha \end{cases} \tag{5.4}$$

By $v \in \sigma$, we mean that v is an element of the domain that models the type σ. Each $mfix'$ gives rise to an $mfix$ via the equation $mfix = mfix' \cdot flip$, and vice versa.[1]

We conjecture that the Equation set 5.4 completely covers all the inhabitants of Type 5.3. The proof attempt for such a claim would require an in-depth analysis of the type, and is beyond the scope of the current work.

Conjecture 5.1.1 Let σ be an arbitrary type. Every inhabitant of Type 5.3 falls into one of the categories given by Equation set 5.4. □

Proposition 5.1.2 None of the candidate definitions for $mfix'$ gives rise to an $mfix$ that would satisfy the purity law.

Proof We will only prove the case

$$mfix'\ f\ k = f^i\ k\ \bot_\alpha, \quad i \geq 0$$

Other cases are similar, if not simpler. Let α be a type and h be a function of type $\alpha \to \alpha$. By purity, we must have:

$$mfix\ (return\ \cdot\ h)\ k\ =\ return\ (fix\ h)\ k\ =\ k\ (fix\ h)$$

Fix a natural number i. By the chosen definition of $mfix'$, we need:

$$(flip\ (return\ \cdot\ h))^i\ k\ \bot_\alpha = k\ (fix\ h) \tag{5.5}$$

It is easy to see that:

$$(flip\ (return\ \cdot\ h))^i\ k = k\ \cdot\ h^i \tag{5.6}$$

Substituting 5.6 in 5.5, we get:

$$k\ (h^i\ \bot_\alpha) = k\ (fix\ h) \tag{5.7}$$

Obviously, Equation 5.7 does not hold for all k and h, given that i is a fixed natural number. □

[1] The function $flip$ is defined by the equation $flip\ f\ x\ y\ =\ f\ y\ x$.

Remark 5.1.3 By the previous proposition, we conclude that the continuation monad (as defined in Section 5.1) does not possess a value recursion operator, provided Conjecture 5.1.1 holds.

The reader might wonder what happens if we restrict α to be the same as σ in the Type 5.1, providing positive occurrences of α to work on. It is possible to show that there is an infinite family of candidate *mfix*'s in this case as well, but none of them satisfy our requirements. We leave the details to the interested reader.

5.2 The continuation monad transformer

The continuation monad transformer [53] is defined by:

$$\textbf{type}\ \mathit{ContT}\ \sigma\ m\ \alpha = (\alpha \to m\ \sigma) \to m\ \sigma$$

$$\begin{aligned} \mathit{return}\ a &= \lambda k.\ k\ a \\ m \gg\!= f &= \lambda k.\ m\ (\lambda a.\ f\ a\ k) \end{aligned}$$

$$\mathit{lift}\ m = \lambda k.\ m \gg\!= k$$

Let *mfixM* be a value recursion operator for a monad m. Can we lift it through the continuation monad transformer, obtaining a value recursion operator for the monad $\mathit{ContT}\ \sigma\ m$? Following the recipe set forth by the examples of Section 4.9, we are led to the following ill-typed definition:

$$\begin{aligned} &\mathit{mfixContM} && :: (\alpha \to \mathit{ContT}\ \sigma\ m\ \alpha) \to \mathit{ContT}\ \sigma\ m\ \alpha \\ &\mathit{mfixContM}\ f && = \lambda k.\ \mathit{mfixM}\ (\lambda a.\ f\ a\ k) \qquad \text{-- } \textit{ill-typed}! \end{aligned} \tag{5.8}$$

Since the argument to *mfixM* has type $\alpha \to m\ \sigma$, the application is ill-typed. This failure is hardly surprising, as setting m to be the identity monad would have resulted in a value recursion operator for the continuation monad.

Remark 5.2.1 Magnus Carlsson has suggested that such a lifting might be possible when restricted to monads that support the notion of mutable variables (personal communication). In collaboration with Carlsson, we investigated a number of possible liftings, but none of our attempts were satisfactory. In each case, it was fairly easy to show that the required properties were violated. We conjecture that a viable lifting is not possible even in this restricted setting, leaving the exploration of this idea for future work.

5.3 First-class continuations and value recursion

What sort of properties can we expect from value recursion operators in a setting with first-class continuations? First-class continuations allow programs to seize the control state of their own evaluators [31]. This facility is definitely more powerful than what the continuation monad of Section 5.1 provides, where programs can only manipulate continuations that are explicitly created and passed around by the programmer.

Many seemingly obvious equivalences fail to hold in the presence of first-class continuations. For instance, as shown by Thielecke, the equivalence $(\lambda x.\ \mathit{False})\ (k\ \mathit{True}) = \mathit{False}$ fails in the context $\mathit{callcc}\ (\lambda k.\ [\,])$. (We refer the interested reader to Thielecke's work for many other interesting examples [84].) When we consider the equivalences dictated by our properties, we see that they are simply too strong to hold in a language with first-class continuations as well. For instance, consider the left shrinking property (Section 2.3), which states the following equivalence:

$$\mathit{mfix}\ (\lambda x.\ a \gg\!= \ \lambda y.\ f\ x\ y) = a \gg\!= \ \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y)$$

Recall that the computation represented by a does *not* use the recursion variable x (i.e., x is not free in a). However, in the presence of first class continuations, a can capture its continuation via a call to *callcc*, thereby getting a handle on f which uses x. That is, a can indirectly access x through f, breaking the left shrinking property.

The following example in Scheme provides further insight into the problem. The example demonstrates that a simple equality between recursive and non-recursive bindings (even simpler than our left shrinking law) fails to hold in the Scheme case. (This example was brought to our attention by Amr Sabry, who traces it back to a message sent to the `comp.lang.scheme` newsgroup in 1988 by A. Bawden, titled *"letrec and callcc implement references."*) Consider the following two Scheme expressions:

```
(define (test1)
  (letrec ((x (call-with-current-continuation
                 (lambda (c) (list #T c)))))
    (if (car x) ((cadr x) (list #F (lambda () x)))
                (eq? x ((cadr x))))))

(define (test2)
  (let    ((x (call-with-current-continuation
                 (lambda (c) (list #T c)))))
    (if (car x) ((cadr x) (list #F (lambda () x)))
                (eq? x ((cadr x))))))
```

Note that these two expressions are the same character for character, except the first one uses the recursive binding construct (`letrec`) of Scheme, while the second one uses the non-recursive version (`let`). Intuitively, these expressions should evaluate to the same result, since the bound variable, x, is not even mentioned in the right hand sides of the bindings. Alas, these two expressions are not equivalent! When run, `test1` evaluates to `#T`, i.e., *True*, while `test2` yields `#F`, i.e., *False*. Regarding this example, Bawden wondered if there were any "*... real compilers that make this mistaken optimization*," regarding that we might view `test2` as an optimized version of `test1`. Of course, our concern is quite the opposite. We rather ask if there are any language constructs that might render the implied equivalence invalid.

Understanding why these expressions yield different values requires an understanding of how Scheme is interpreted. We will try to convey the idea here as it is essential in understanding why the left shrinking property is likely to be too strong a requirement in the presence of first-class continuations. To keep the notation simple, let us rewrite these expressions in a more Haskell-like syntax:[2]

$$
\begin{array}{l}
\textit{test1} \equiv \\
\quad \textbf{letrec}\ x = \textit{callcc}\ (\lambda c.\ (\textit{True},\ c)) \\
\quad \textbf{in if}\ \textit{fst}\ x \\
\qquad \textbf{then}\ \textit{snd}\ x\ (\textit{False},\ \textit{const}\ x) \\
\qquad \textbf{else}\ \textit{eq?}\ x\ (\textit{snd}\ x\ ())
\end{array}
$$

$$
\begin{array}{l}
\textit{test2} \equiv \\
\quad \textbf{let}\quad x = \textit{callcc}\ (\lambda c.\ (\textit{True},\ c)) \\
\quad \textbf{in if}\ \textit{fst}\ x \\
\qquad \textbf{then}\ \textit{snd}\ x\ (\textit{False},\ \textit{const}\ x) \\
\qquad \textbf{else}\ \textit{eq?}\ x\ (\textit{snd}\ x\ ())
\end{array}
$$

Intuitively, **letrec** $x = A$ **in** B in Scheme is implemented by allocating a cell called x with a bogus error value, computing the value of the expression A (with x in scope), and then overwriting the cell x with the result [44]. This *allocate-compute-overwrite* paradigm practically achieves the knot-tying implementation of recursion. The evaluation then goes on with the expression B, again with x in scope. A simple **let** binding, on the other hand, does not create a cell to start with: **let** $x = A$ **in** B is interpreted by evaluating A, storing the result into a newly created cell x, and evaluating B with x in scope. With this model in mind, consider the **letrec** expression in the definition of *test1*:

$$\textbf{letrec}\ x \ = \ \textit{callcc}\ (\lambda c.\ (\textit{True},\ c))\ \textbf{in} \ldots$$

To interpret this expression, one allocates a cell named x, and initializes it with $\perp$. Then, the right hand side is interpreted. The crucial point is realizing what continuation is captured by the call to *callcc*. Recalling our description above, the following continuation will be captured:

1. Let a be the argument passed to the continuation. Overwrite the cell x with a,

[2] The function *eq?* checks for pointer equality in Scheme, rather than structural equality.

2. Evaluate the expression part of **letrec**, i.e., evaluate:

$$\begin{array}{l}\textbf{if } \mathit{fst}\ x \textbf{ then } \mathit{snd}\ x\ (\mathit{False},\ \mathit{const}\ x) \\ \qquad\qquad\ \textbf{else } \mathit{eq?}\ x\ (\mathit{snd}\ x\ ())\end{array}$$

Let us call the continuation described above κ. Now, the right hand side of the **letrec** binding is computed, which returns the tuple (*True*, κ). Since the definition is not actually recursive, the initial (undefined) value of x is not used. Then, the cell pointed to by x is overwritten by this tuple and the interpreter continues on with the evaluation of the body. Since *fst* x is *True*, the conditional takes its first branch. And it is exactly at this point that we invoke the continuation through the expression *snd* x, which is passed the argument (*False, const* x). Recalling the description of κ above, this tuple overwrites the cell x. It is crucial to note the cyclic structure thus created: When called with an argument, the function stored in the second element of x will return a pointer back to the tuple itself. As dictated by step 2 of κ, we now evaluate the body with this new value stored in the cell pointed to by x. But this time *fst* x is *False*, hence we end up evaluating the expression *eq?* x (*snd* x ()). Since, *snd* x () returns a pointer back to x, the call to *eq?* checks for the pointer equality of x and x, which simply results in the value *True.*

What happens with *test2*? Since we have a non-recursive **let** expression, the cell for x is *not* created before the right hand side is computed. Let us call this continuation ϕ. Here is our description of it:

1. Let a be the argument passed to the continuation. Store a in a new cell called x,

2. Evaluate the expression part of **let**, which is exactly the same as before.

To evaluate *test2*, we proceed by computing the right hand side of the **let** binding. As before, we immediately get back the tuple (*True*, ϕ). Now a new cell named x is created, which stores this tuple. The conditional again takes its first branch, and the continuation is called with (*False, const* x). Unlike the previous case, however, the call to the continuation creates a new cell named x, shadowing the earlier value of x: The cyclic structure is no longer available! It is not hard to see what happens now. The body is evaluated as before and the conditional takes its second branch. But this time we compare two different tuples in the call to *eq?*. Hence the result is simply *False.*

The relevance of this example to the left shrinking property is obvious. Basically, the right hand side of the **letrec** binding, which is not recursive, corresponds to the constant computation in the left shrinking property. If left shrinking were to hold, we would be allowed to pull it out of the *mfix* loop, i.e., replace the recursive binding with a non-recursive one. As we have seen, at least in the Scheme case, such a transformation is not valid in the presence of first-class continuations.

5.4 Summary

As we have seen, the continuation monad in Haskell (as defined in Section 5.1) does not seem to have a suitable value recursion operator. A similar comment applies to the continuation monad transformer. Furthermore, in the case of first-class continuations, the properties we expect to hold for value recursion operators are simply too strong.

Regarding the handling of recursive definitions arbitrarily mixed with computational effects in Scheme, Andrzej Filinski states (personal communication):

> ...as far as I know, the only popular functional language that allows such definitions is Scheme; and I believe that allowing them was a mistake. The extra generality is virtually never used, but it disallows some useful optimizations by unnecessarily constraining the implementation. It is well known that in the presence of call/cc, one can expose the imperative nature of letrec and use it to define a general mutable cell; any RnRS-conforming system must support this behavior no matter how it implements recursion...

We share the same point of view.

Chapter 6

Traces and value recursion

Trace operators were introduced into category theory by Joyal et al., as a means for modeling feedback operations arising in physics and mathematics [43]. Later work by Hasegawa bridged the gap between recursion and traces, establishing a one-to-one correspondence between fixed-point operators and traces over cartesian categories [9, 10, 32, 33, 79]. Can we explain value recursion in this framework as well? The aim of this chapter is to review the recent research in this area, trying to gain a better understanding of value recursion.

Synopsis. First, we will introduce parameterized value recursion operators, making the dependence on the environment explicit. After reviewing traced monoidal categories, we will show that value recursion operators give rise to traces for a restricted class of monads. Although the set of monads for which this is possible is quite small, the correspondence is strong enough for us to explore. The restriction arises as a consequence of trace axioms, which are simply too strong for value recursion in general. Motivated by this discussion, we will briefly review recent work by Paterson [66], and Benton and Hyland [5], which aims to generalize traces to premonoidal categories.

6.1 Parameterized value recursion

Recall that a value recursion operator for a monad m has type $(\alpha \to m\ \alpha) \to m\ \alpha$. In a categorical setting, one needs explicitly to account for terms that contain free variables, i.e., variables that are defined in the enclosing environment. To do so, we parameterize our type to:

$$((e, \alpha) \to m\ \alpha) \to (e \to m\ \alpha)$$

where e represents the environment. In the concrete case, e is generally a product, using the cartesian structure of the underlying language. Parameterized and non-parameterized value recursion operators are interdefinable:

$$\begin{aligned} \mathit{pmfix}_{e,\alpha} &:: ((e,\alpha) \to m\ \alpha) \to (e \to m\ \alpha) \\ \mathit{pmfix}_{e,\alpha}\ f &= \lambda e.\ \mathit{mfix}_{\alpha}\ (\lambda a.\ f\ (e,a)) \qquad (6.1) \\ \mathit{mfix}_{\alpha} &:: (\alpha \to m\ \alpha) \to m\ \alpha \\ \mathit{mfix}_{\alpha}\quad f &= \mathit{pmfix}_{\mathbb{1},\alpha}\ (f \cdot \pi_2)\ () \qquad (6.2) \end{aligned}$$

where $\mathbb{1}$ is the terminal object whose only element is written $()$. The choice for the terminal object is the natural one for e in Equation 6.2, as it represents the empty environment. In fact, any type would do, since the environment is simply ignored.

Remark 6.1.1 Before proceeding further, a word on notation is in order. In this chapter, we will be using a more categorical notation where appropriate. For instance, types will be written with capital letters (as objects in a certain category), products will be written with ×, etc. This shift is unfortunate, but necessary. We do not want to impose a Haskell-like notation when talking about categorical constructs: Such a coercion seems to complicate matters even more. As an example, the type for *pmfix* in 6.1 will be written:

$$\mathit{pmfix}_{A,X} : \mathcal{D}(A \times X,\ T\ X) \to \mathcal{D}(A,\ T\ X)$$

where $\mathcal{D}$ is the category of domains and T is the underlying functor for the monad we are considering. (The notation $\mathcal{D}(A,\ B)$ denotes the set of arrows from A to B in $\mathcal{D}$.) We will stick to Latin letters for objects, following the general practice. The use of particular letters (i.e., X for the recursion variable, and A for the parameter) is inherited from Hasegawa's work [33]. Also, we will use categorical products and function spaces, rather than Haskell's lifted versions.

The second generalization we want to make is more technical than the first. Rather than considering the morphisms in the base category, we move to the Kleisli category of the given monad. There is one difficulty, however. The Kleisli category is not necessarily cartesian. More specifically, the binary operator inherited from the cartesian product of the base category is not necessarily bifunctorial. We will see the details and implications of this problem in Section 6.4.2. For the time being, let us just assume that we have a product-like operation in the Kleisli category, named ×. Let $\mathcal{D}_T$ denote the Kleisli category of a given strong[1] monad T over $\mathcal{D}$. It is easy to see that *pmfix* can be considered

[1] A monad over a category with a monoidal operation $\otimes$ is called strong if there exists a natural transformation $t_{A,B} : A \otimes T\ B \to T\ (A \otimes B)$, called the *strength*, subject to certain conditions [63]. It turns out that all Haskell monads are strong, with the strength defined as $t\ (a,\ tb) = tb \gg\!\!= \ \lambda b.\ \mathit{return}\ (a,\ b)$.

as a family of functions with the type:

$$pmfix_{A,X} : \mathcal{D}_T(A \times X, X) \to \mathcal{D}_T(A, X) \tag{6.3}$$

If $\mathcal{D}_T$ is cartesian, Type 6.3 is precisely the same as that of a Conway operator (see Appendix A). This view of value recursion will prove essential in the following discussion.

6.2 Preliminaries

In this section, we review the central notions in Joyal et al., and Hasegawa's work [33, 43], covering symmetric monoidal categories, traces, and the correspondence between traces over cartesian categories and Conway operators.

6.2.1 Symmetric monoidal categories

In computer science, we often deal with binary operators that are associative only up to isomorphism. Monoidal operators and monoidal categories provide a setting where such operators can be modeled formally [2, 55]:

Definition 6.2.1 (*Symmetric Monoidal Category.*) A symmetric monoidal category, SMC for short, $\mathcal{M} = (\mathcal{M}, \otimes, I, a, l, r, s)$ is a category $\mathcal{M}$ with a bifunctor $\otimes : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, an object $I \in \mathcal{M}$, and natural isomorphisms:

$$\begin{array}{lcl@{\qquad}lcl} a_{A,B,C} & : & A \otimes (B \otimes C) \to (A \otimes B) \otimes C & l_A & : & I \otimes A \to A \\ s_{A,B} & : & A \otimes B \to B \otimes A & r_A & : & A \otimes I \to A \end{array}$$

such that the following diagrams commute:
Associativity Pentagon:

$$\begin{tikzcd}
A \otimes (B \otimes (C \otimes D)) \arrow[r, "a"] \arrow[d, "A\otimes a"'] & (A \otimes B) \otimes (C \otimes D) \arrow[r, "a"] & ((A \otimes B) \otimes C) \otimes D \\
A \otimes ((B \otimes C) \otimes D) \arrow[rr, "a"'] & & (A \otimes (B \otimes C)) \otimes D \arrow[u, "a\otimes D"']
\end{tikzcd}$$

Unit triangles and symmetry:

$$\begin{tikzcd}
A \otimes (I \otimes B) \arrow[d, "a"'] \arrow[dr, "A\otimes l"] & \\
(A \otimes I) \otimes B \arrow[r, "r\otimes B"'] & A \otimes B
\end{tikzcd}
\qquad
\begin{tikzcd}
A \otimes B \arrow[d, "s"'] \arrow[dr, "A\otimes B"] & \\
B \otimes A \arrow[r, "s"'] & A \otimes B
\end{tikzcd}
\qquad
\begin{tikzcd}
A \otimes I \arrow[d, "s"'] \arrow[dr, "r"] & \\
I \otimes A \arrow[r, "l"'] & A
\end{tikzcd}$$

Bilinearity:

$$\begin{tikzcd}
A \otimes (B \otimes C) \arrow[r, "a"] \arrow[d, "A\otimes s"'] & (A \otimes B) \otimes C \arrow[r, "s"] & C \otimes (A \otimes B) \arrow[d, "a"] \\
A \otimes (C \otimes B) \arrow[r, "a"'] & (A \otimes C) \otimes B \arrow[r, "s\otimes B"'] & (C \otimes A) \otimes B
\end{tikzcd}$$

Example 6.2.2 All cartesian categories are symmetric monoidal. Let $\mathcal{C} = (\mathcal{C}, \times, \mathbb{1})$ be a cartesian category where $\times$ is the direct product with projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. In this case, the natural isomorphisms of Definition 6.2.1 are given by:

$$a = \langle\langle \pi_1, \pi_1 \cdot \pi_2\rangle, \pi_2 \cdot \pi_2\rangle \qquad l = \pi_2 \qquad r = \pi_1 \qquad s = \langle \pi_2, \pi_1\rangle$$
$$a^{-1} = \langle \pi_1 \cdot \pi_1, \langle \pi_2 \cdot \pi_1, \pi_2\rangle\rangle \qquad l_A^{-1} = \langle !_A, A\rangle \qquad r_A^{-1} = \langle A, !_A\rangle \qquad s^{-1} = \langle \pi_2, \pi_1\rangle$$

where $!_A : A \to \mathbb{1}$ denotes the unique map to the terminal object. In Haskell notation these morphisms correspond to the following functions (with more suggestive names):

$$\begin{array}{llcl@{\qquad}llcl}
\mathit{assoc} & (x,\ (y,\ z)) & = & ((x,\ y),\ z) & \mathit{assoc}^{-1} & ((x,\ y),\ z) & = & (x,\ (y,\ z)) \\
\mathit{left} & ((),\ y) & = & y & \mathit{left}^{-1} & y & = & ((),\ y) \\
\mathit{right} & (x,\ ()) & = & x & \mathit{right}^{-1} & x & = & (x,\ ()) \\
\mathit{swap} & (x,\ y) & = & (y,\ x) & \mathit{swap}^{-1} & (y,\ x) & = & (x,\ y)
\end{array}$$

6.2.2 Traced symmetric monoidal categories

Trace operators provide a categorical framework for studying cyclic structures:[2]

Definition 6.2.3 (*Traced SMC.*) A traced symmetric monoidal category is a symmetric monoidal category $\mathcal{M} = (\mathcal{M}, \otimes, I, a, l, r, s)$ with a family of functions:

$$\mathit{Tr}^X_{A,B} : \mathcal{M}(A \otimes X, B \otimes X) \to \mathcal{M}(A, B)$$

subject to the following conditions:

- Naturality in A (left tightening):

$$\begin{tikzcd}
A \arrow[d, "g"'] & \mathcal{M}(A \otimes X, B \otimes X) \arrow[r, "\mathit{Tr}^X_{A,B}"] & \mathcal{M}(A, B) \\
A' & \mathcal{M}(A' \otimes X, B \otimes X) \arrow[r, "\mathit{Tr}^X_{A',B}"'] \arrow[u, "\mathcal{M}(g\otimes X, B\otimes X)"] & \mathcal{M}(A', B) \arrow[u, "\mathcal{M}(g,B)"']
\end{tikzcd}$$

For all $f : A' \otimes X \to B \otimes X,\ g : A \to A',\quad \mathit{Tr}\,(f \cdot (g \otimes X)) = \mathit{Tr}\, f \cdot g.$

[2] The original work on traces was presented in the slightly more general setting of *braided monoidal categories* [43]. Following Hasegawa [33], we only consider *symmetric monoidal categories* here.

- Naturality in B (right tightening):

$$\begin{tikzcd}
B \arrow[d, "g"'] & \mathcal{M}(A \otimes X, B \otimes X) \arrow[rr, "\mathit{Tr}^X_{A,B}"] \arrow[d, "\mathcal{M}(A\otimes X, g\otimes X)"'] & & \mathcal{M}(A,B) \arrow[d, "\mathcal{M}(A,g)"] \\
B' & \mathcal{M}(A \otimes X, B' \otimes X) \arrow[rr, "\mathit{Tr}^X_{A,B'}"'] & & \mathcal{M}(A,B')
\end{tikzcd}$$

For all $f : A \otimes X \to B \otimes X,\ g : B \to B',\quad \mathit{Tr}\,((g \otimes X) \cdot f) = g \cdot \mathit{Tr}\, f.$

- Dinaturality in X (sliding):

$$\begin{tikzcd}
X \arrow[d, "g"'] & \mathcal{M}(A \otimes X', B \otimes X) \arrow[rr, "\mathcal{M}(A\otimes g, B\otimes X)"] \arrow[d, "\mathcal{M}(A\otimes X', B\otimes g)"'] & & \mathcal{M}(A \otimes X, B \otimes X) \arrow[d, "\mathit{Tr}^X_{A,B}"] \\
X' & \mathcal{M}(A \otimes X', B \otimes X') \arrow[rr, "\mathit{Tr}^{X'}_{A,B}"'] & & \mathcal{M}(A,B)
\end{tikzcd}$$

For all $f : A \otimes X' \to B \otimes X,\ g : X \to X',\quad \mathit{Tr}\,(f \cdot (A \otimes g)) = \mathit{Tr}\,((B \otimes g) \cdot f).$

- Vanishing:
  - For all $f : A \to B,\quad \mathit{Tr}^I_{A,B}(r^{-1} \cdot f \cdot r) = f.$
  - For all $f : A \otimes (X \otimes Y) \to B \otimes (X \otimes Y),$

$$\mathit{Tr}^X_{A,B}\,(\mathit{Tr}^Y_{A\otimes X, B\otimes X}\,(a \cdot f \cdot a^{-1})) = \mathit{Tr}^{X\otimes Y}_{A,B}\ f\,.$$

- Superposing: For all $f : A \otimes X \to B \otimes X,$

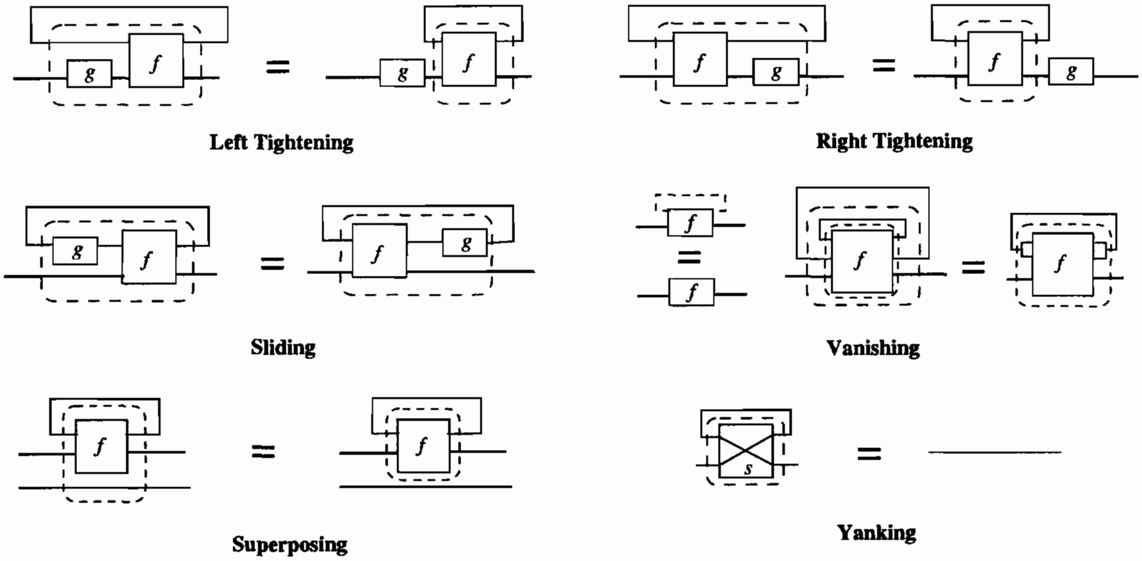
$$\mathit{Tr}^X_{C\otimes A, C\otimes B}\,(a \cdot (C \otimes f) \cdot a^{-1}) = C \otimes \mathit{Tr}^X_{A,B}\ f\,.$$

- Yanking: For all $X,\quad \mathit{Tr}^X_{X,X}\,(s_{X,X}) = X.$

The graphical versions of these axioms are given in Figure 6.1 [33, 43]. It is worth comparing these diagrams to those that we have given in Chapter 2 for *mfix*. The thick lines in the figures for *mfix* represent monadic actions, i.e., side-effects, changes in the state, etc., while the corresponding lines in Figure 6.1 represent data flow. The fixed-point argument (i.e., X) is not directly available to the outside world in the formulation of trace (although this limitation can be easily circumvented). In *mfix*, however, the result is the fixed-point value together with monadic actions.

6.2.3 Traces and Conway operators

The following theorem of Hasegawa (also independently established by Hyland) states the connection between traces and Conway operators (See Appendix A for a brief review of Conway operators):



Left Tightening

Right Tightening

Sliding

Vanishing

Superposing

Yanking

Figure 6.1: Trace Axioms

Theorem 6.2.4 (Hasegawa, Hyland) A cartesian category is traced exactly when it possesses a Conway operator.

Proof See Theorem 7.1.1 of Hasegawa's thesis [32]. □

The correspondence can be summarized as follows. Assuming we have a trace operator Tr, we can define a Conway operator $(\cdot)^\dagger : \mathcal{C}(A \times X, X) \to \mathcal{C}(A, X)$ as follows:

$$f^\dagger = \mathit{Tr}^X_{A,X}\ (\Delta_X \cdot f) \ \ : \ \ A \to X \tag{6.4}$$

Similarly, given a Conway operator $(\cdot)^\dagger$, we can define the following trace operator:

$$\mathit{Tr}^X_{A,B}\ f = \pi_1 \cdot f \cdot \langle A, (\pi_2 \cdot f)^\dagger \rangle \ \ : \ \ A \to B \tag{6.5}$$

Since Conway operators provide a generalization of fixed-point operators on domains, traces on symmetric monoidal categories provide a firm categorical framework for studying fixed-point operators.

Example 6.2.5 In the setting of domains and continuous functions, the unique least fixed-point operator for a function $f : A \to A$ is given by:

$$\mathit{fix}\, f = \bigsqcup_i f^i \perp_A$$

which gives rise to the following Conway operator: Given $f : A \times X \to X$,

$$f^\dagger\ a = \mathit{fix}\ (\lambda x. f\ (a, x)) : A \to X$$

And, by Equation 6.5, we obtain the following (unique) trace operator: Given $f : A \times X \to B \times X$,

$$Tr^X_{A,B}\, f = \pi_1 \cdot f \cdot \langle A, \lambda a.\, fix\, (\lambda x.\, \pi_2\, (f\, (a, x)))\rangle \quad : \quad A \to B$$

In Haskell-like notation, this definition simply reads:

$$\begin{array}{lcl} trace & :: & ((\alpha,\ \gamma) \to (\beta,\ \gamma)) \to \alpha \to \beta \\ trace\ f\ \ a & = & \mathbf{let}\ (b,\ x) = f\ \ (a,\ x) \\ & & \mathbf{in}\ b \end{array} \tag{6.6}$$

which clearly shows the intent: The recursive knot is tied over x, leaving a function of type $A \to B$ as the residue.

6.3 Traces and value recursion

As we have seen in the preceding section, traces provide a natural framework for studying fixed-point operators, and by virtue of Theorem 6.2.4, the usual notion of recursion can be explained by traces over cartesian categories. Does the correspondence hold up when we consider value recursion? It turns out that a close relationship can be established for commutative monads whose Kleisli categories are traced, but the trace axioms are simply too strong for the general case. Still, we will explore this limited correspondence closely, as it will help us identify the problems that arise in the general case. We start by examining two particular classes of monads: commutative monads and monads based on commutative monoids.

6.3.1 Commutative monads and traces

Let T be a strong commutative monad over an SMC $\mathcal{M} = (\mathcal{M}, \otimes, I, a, l, r, s)$ with the given strength t. We write η for the unit, and μ for the multiplier of T. The monoidal structure over $\mathcal{M}$ carries over to the Kleisli category of T, denoted $\mathcal{M}_T$, as follows:

$$\mathcal{M}_T = (\mathcal{M}_T,\ \otimes',\ I,\ \eta \cdot a,\ \eta \cdot l,\ \eta \cdot r,\ \eta \cdot s)$$

The monoidal operator $\otimes$ is lifted to $\mathcal{M}_T$ as follows. On objects, $\otimes'$ is defined to be the same as $\otimes$. On arrows, $f \otimes' g$ is defined to be the arrow $\Theta \cdot (f \otimes g)$ in $\mathcal{M}$, where

$$\begin{array}{lcl} \Theta & : & T\,A \otimes T\,B \to T\,(A \otimes B) \\ \Theta & = & \mu \cdot T\,t \cdot t' \end{array}$$

Recall that t' is the dual of t, given by $T\ s \cdot t \cdot s$. Since T is commutative, the other candidate for Θ, i.e., $\mu \cdot T\,t' \cdot t$, yields exactly the same arrow.

In the case of CCC's, a trace operator on the Kleisli category of a commutative monad gives rise to a parameterized value recursion operator on the underlying category. To see this, let $\mathcal{C}$ be a CCC, and T be a commutative monad over $\mathcal{C}$. If $\mathcal{C}_T$ is traced, we have a family of functions:

$$Tr^X_{A,B} : \mathcal{C}_T(A \times' X, B \times' X) \to \mathcal{C}_T(A, B)$$

which implies the existence of the following family of functions in $\mathcal{C}$:

$$Tr^X_{A,B} : \mathcal{C}(A \times X, T\ (B \times X)) \to \mathcal{C}(A, T\ B)$$

Hence, a candidate parameterized value recursion operator can be defined by setting:

$$pmfix_{E,A}\ f = Tr^A_{E,A}\ (T\ \triangle_A \cdot f) \tag{6.7}$$

where $\Delta\ a = (a,\ a)$.

Example 6.3.1 The environment monad provides a nice example of obtaining a value recursion operator from a trace. For a fixed object E in a CCC, the functor $T\ A = E \Rightarrow A$, i.e., the exponentiation functor with the first argument fixed, gives rise to the environment monad. For convenience, we will stick to the Haskell notation. The monad structure and the strength are given by:

$$\begin{aligned} &return\ a &&= \lambda e.\ a \\ &join\ f &&= \lambda e.\ f\ e\ e \\ &t\ (a,\ f) &&= \lambda e.\ (a,\ f\ e) \end{aligned}$$

It is easy to see that T is commutative. The Kleisli category is traced, and the corresponding family of functions in the base category is given by:

$$\begin{aligned} &trace &&::\ ((\alpha,\ \tau) \to (E \to (\beta,\ \tau))) \to \alpha \to (E \to \beta) \\ &trace\ f\ a &&= \lambda e.\ \mathbf{let}\ (b,\ x) = f\ (a,\ x)\ e \\ & && \qquad\quad \mathbf{in}\ b \end{aligned} \tag{6.8}$$

Using Equations 6.7 and 6.2 we get:

$$\begin{aligned} &mfix &&::\ (\alpha \to (E \to \alpha)) \to E \to \alpha \\ &mfix\ f &&= \lambda e.\ \mathbf{let}\ (b,\ x) = (map\ (\lambda z.\ (z,\ z)) \cdot f \cdot \pi_2)\ ((),\ x)\ e \\ & && \qquad\quad \mathbf{in}\ b \end{aligned}$$

Recalling $map\ f\ g = f \cdot g$ for the environment monad, we can simplify this definition to:

$$\begin{aligned} mfix\ f\ = \lambda e.\ &\mathbf{let}\ x = f\ x\ e \\ &\mathbf{in}\ x \end{aligned}$$

which is precisely the value recursion operator that we have given in Section 4.6 for the environment monad.

Example 6.3.2 This example demonstrates that having a commutative monad is not sufficient to guarantee the construction of a value recursion operator: The corresponding Kleisli category should be traced as well. As an example, consider modeling exceptions in *Set* by disjoint sums, using the endofunctor $T\ A = \mathbb{1} + A$, where $\mathbb{1}$ is the terminal object. In Haskell-like notation, the monad structure and the strength are given by:

$$\begin{aligned} \eta\ a &= \mathit{inr}\ a \\ \mu\ (\mathit{inl}\ ()) &= \mathit{inl}\ () \\ \mu\ (\mathit{inr}\ a) &= a \end{aligned} \qquad \begin{aligned} t\ (a,\ \mathit{inl}\ ()) &= \mathit{inl}\ () \\ t\ (a,\ \mathit{inr}\ b) &= \mathit{inr}\ (a,\ b) \end{aligned}$$

It is easy see that T gives rise to a commutative monad, and hence its Kleisli category is symmetric monoidal. If Set_T is traced, then we should have a family of arrows $Tr^X_{A,B} : Set_T(A \otimes X, B \otimes X) \to Set_T(A, B)$, where $\otimes$ is the lifting of the cartesian product. Hence, we must have a family of arrows $Set(A \times X, \mathbb{1} + (B \times X)) \to Set(A, \mathbb{1} + B)$. However, since the computation might fail, we do not have a way of getting an X to tie the recursive knot. In this case, the Kleisli category does not seem to possess a trace.

The reader might appreciate the situation in Haskell. The exception monad is the usual *Maybe* monad, except the Haskell version is not commutative (due to the possibility of non-termination). Ignoring the non-termination issue for a moment, we would need to find a trace operator with the type:

$$((\alpha,\ \tau) \to \mathit{Maybe}\ (\beta,\ \tau)) \to \alpha \to \mathit{Maybe}\ \beta$$

Here, τ is the recursion argument, on which we need to tie the recursive knot. However, the required trace operator just does not exist, since we are not guaranteed to get a τ to form the required recursive loop. (Recall that there is no such problem for value recursion in our setting—see Section 4.2 for details.)

6.3.2 Monads arising from commutative monoids

In Section 4.5, we explored monads that arise from monoids. In this section, we will concentrate on those monads that are obtained from commutative monoids, and see how a trace operator in the underlying category can be used to obtain a value recursion operator for the corresponding representation monad.

The usual definition of monoids on sets can be generalized to arbitrary monoidal categories [55]. Let $\mathcal{M} = (\mathcal{M}, \otimes, I, a, l, r, s)$ be a symmetric monoidal category. A monoid in $\mathcal{M}$ is a triple $M = (M, +, e)$ where $M \in \mathcal{M}$, $+ : M \otimes M \to M$, $e : I \to M$, with the usual associativity and unit laws. (The monoid is commutative if $+ \cdot s = +$, i.e., if the order of arguments to $+$ does not matter.) For such a monoid, the endofunctor $T\ A = M \otimes A$ gives rise to the following strong monad, known as M's representation monad [2]:

$$\eta_A = (e \otimes A) \cdot l_A^{-1} \tag{6.9}$$

$$\mu_A = (+ \otimes A) \cdot a_{M,M,A} \tag{6.10}$$

$$t_{A,B} = a^{-1}_{M,A,B} \cdot (s_{A,M} \otimes B) \cdot a_{A,M,B} \tag{6.11}$$

If M is commutative, then T will be commutative as well.

After all this machinery, we can finally state our goal. Let $\mathcal{M}$ be a traced SMC, M be a commutative monoid in $\mathcal{M}$, whose representation monad is T. As we have seen, T is commutative and hence its Kleisli category is symmetric monoidal. Furthermore, the trace on $\mathcal{M}$ lifts into $\mathcal{M}_T$, i.e., $\mathcal{M}_T$ is also traced. If $Tr^X_{A,B} : \mathcal{M}(A \otimes X, B \otimes X) \to \mathcal{M}(A, B)$ is the trace operator on $\mathcal{M}$, the trace operator on $\mathcal{M}_T$ is given by:

$$\begin{aligned} Tr'^X_{A,B} &: \mathcal{M}_T(A \otimes' X, B \otimes' X) \to \mathcal{M}_T(A, B) \\ Tr'^X_{A,B}\, f &= Tr^X_{A,M\otimes B}\, (a \cdot f) \end{aligned} \tag{6.12}$$

Example 6.3.3 Consider the monoid: $(\mathbb{N}, +, 0)$, where $\mathbb{N}$ is the flat domain of natural numbers, and $+$ is addition. The corresponding functor is: $T\ A = \mathbb{N} \times A$. As outlined above, the monad structure is given by (in Haskell):

```
return x            = (0, x)
join   (m, (n, x))  = (m+n, x)
t      (x, (m, y))  = (m, (x, y))
t'     ((m, x), y)  = (m, (x, y))
```

Since *map f (m, x) = (m, f x)*, we have:

```
(join · map t' · t) ((m, x), (n, y)) = (n+m, (x, y))
(join · map t · t') ((m, x), (n, y)) = (m+n, (x, y))
```

Hence, the commutativity follows from the commutativity of $+$, as promised. Recall from Example 6.2.5 that the trace on the underlying category is given by:

```
trace f a = let (b, x) = f (a, x) in b
```

which, by Equation 6.12, can be treated as a trace operator on the Kleisli category of T with the type: $(A \times X \to \mathbb{N} \times (B \times X)) \to (A \to \mathbb{N} \times B)$. More explicitly, we have (where we use *Integer* to represent $\mathbb{N}$):

$$\begin{aligned} &tr' &&:: ((\alpha,\ \sigma) \to (\mathit{Integer},\ (\beta,\ \sigma))) \to (\alpha \to (\mathit{Integer},\ \beta)) \\ &tr'\ f\ a &&= \mathbf{let}\ (b,\ x) = (\mathit{assoc} \cdot f)\ (a,\ x)\ \mathbf{in}\ b \end{aligned}$$

By Equation 6.7, we obtain the following parameterized value recursion operator:

$$\begin{aligned} \mathit{pmfix}\ f\ = \lambda a.\ &\mathbf{let}\ (b,\ x) = (\mathit{assoc} \cdot \mathit{map}\ (\lambda x.\ (x,\ x)) \cdot f)\ (a,\ x) \\ &\mathbf{in}\ b \end{aligned}$$

which gives rise to the following value recursion operator by Equation 6.2:

$$\begin{aligned} &\mathit{mfix} &&:: (\alpha \to (\mathit{Integer},\ \alpha)) \to (\mathit{Integer},\ \alpha) \\ &\mathit{mfix}\ f &&= \mathbf{let}\ (b,\ x) = (\mathit{assoc} \cdot \mathit{map}\ (\lambda x.\ (x,\ x)) \cdot f)\ x\ \mathbf{in}\ b \end{aligned}$$

By expanding the definitions and simplifying, one obtains:

$$\begin{aligned} &\mathit{mfix} &&:: (\alpha \to (\mathit{Integer},\ \alpha)) \to (\mathit{Integer},\ \alpha) \\ &\mathit{mfix}\ f &&= \mathbf{let}\ (n,\ x) = f\ x \\ & &&\quad\ \mathbf{in}\ (n,\ x) \end{aligned} \tag{6.13}$$

which is precisely the value recursion operator we have given for monads based on monoids (Equation 4.19) in Section 4.5.

Remark 6.3.4 It is important to note that the commutativity of the monoid does not play any role in establishing the requirements of value recursion, although it is essential for constructing a trace. If the monoid is not commutative, the representation monad will not be commutative either, failing to yield a monoidal structure on the Kleisli category. In that case, one cannot even talk about the notion of trace, as Definition 6.2.3 only applies to symmetric monoidal categories. We will return to this issue in Section 6.4.2.

6.3.3 The correspondence

We now turn to the correspondence between value recursion operators for commutative monads and trace operators over Kleisli categories. Before doing so, we will need to consider what trace axioms mean in the Kleisli category of a given monad. Let T be the monad under consideration. In this setting, the trace axioms read:[3]

[3] In these equations, we use the Haskell notation and try to name variables according to their types, i.e., a variable named a is of type A. Note the use of shadowing in λ-bindings, where we reuse variable names to stick to our convention. Compared to the original trace axioms, these versions are indeed very ugly to look at, but they are much more intuitive from a programming perspective. Also, to save space, we use η to abbreviate *return*.

- Left tightening: For all $f : A' \times X \to T\ (B \times X), g : A \to T\ A'$,

$$\mathit{Tr}\ (\lambda(a,\ x).\ g\ a \gg\!= \ \lambda a'.\ f\ (a',\ x)) = \lambda a.\ g\ a \gg\!= \ \mathit{Tr}\ f \tag{6.14}$$

- Right tightening: For all $f : A \times X \to T\ (B \times X), g : B \to T\ B'$,

$$\begin{aligned} &\mathit{Tr}\ (\lambda(a,\ x).\ f\ (a,\ x) \gg\!= \ \lambda(b,\ x).\ g\ b \gg\!= \ \lambda b'.\ \eta\ (b',\ x)) \\ &= \lambda a.\ \mathit{Tr}\ f\ a \gg\!= \ g \end{aligned} \tag{6.15}$$

- Sliding: For all $f : A \times X' \to T\ (B \times X), g : X \to T\ X'$,

$$\begin{aligned} &\mathit{Tr}\ (\lambda(a,\ x).\ g\ x \gg\!= \ \lambda x'.\ f\ (a,\ x')) \\ &= \mathit{Tr}\ (\lambda(a,\ x').\ f\ (a,\ x') \gg\!= \ \lambda(b,\ x).\ g\ x \gg\!= \ \lambda x'.\ \eta\ (b,\ x')) \end{aligned} \tag{6.16}$$

- Vanishing: For all $f : A \to T\ B$,

$$\mathit{Tr}\ (\lambda(a,\ ()).\ f\ a \gg\!= \ \lambda b.\ \eta\ (b,\ ())) = f \tag{6.17}$$

  and, for all $f : A \times (X \times Y) \to T\ (B \times (X \times Y))$,

$$\begin{aligned} &\mathit{Tr}\ (\mathit{Tr}\ (\lambda((a,\ x),\ y).\ f\ (a,\ (x,\ y)) \gg\!= \ \lambda(b,\ (x,\ y)).\ \eta\ ((b,\ x),\ y))) \\ &= \mathit{Tr}\ f \end{aligned} \tag{6.18}$$

- Superposing: For all $f : A \times X \to T\ (B \times X)$,

$$\begin{aligned} &\mathit{Tr}\ (\lambda((c,\ a),\ x).\ f\ (a,\ x) \gg\!= \ \lambda(b,\ x).\ \eta\ ((c,\ b),\ x)) \\ &= \lambda(c,\ a).\ \mathit{Tr}\ f\ a \gg\!= \ \lambda b.\ \eta\ (c,\ b) \end{aligned} \tag{6.19}$$

- Yanking:

$$\mathit{Tr}\ (\lambda(x_1,\ x_2).\ \eta\ (x_2,\ x_1)) = \eta \tag{6.20}$$

After these preliminaries, we can finally state the main result of this chapter:

Proposition 6.3.5 Let $\mathcal{D}$ be the category of domains, and T be a commutative monad over $\mathcal{D}$. Let *mfix* be a value recursion operator for T, further satisfying strong sliding, nesting, and right shrinking laws. Then, the family of functions

$$\begin{aligned} \mathit{trace}^X_{A,B} &: \mathcal{D}(A \times X,\ T\ (B \times X)) \to \mathcal{D}(A,\ T\ B) \\ \mathit{trace}\ f &= \lambda a.\ \mathit{mfix}_{B \times X}\ (\lambda(b,\ x).\ f\ (a,\ x)) \gg\!= \ \eta \cdot \pi_1 \end{aligned} \tag{6.21}$$

will satisfy Equations 6.14-6.20, i.e., it will provide a trace operator for $\mathcal{D}_T$.

Proof See Appendix B.8 for the full derivation. We try to summarize the correspondence at a higher level here. Unsurprisingly, left and right tightenings depend on the left and right shrinking properties of *mfix* respectively. Sliding requires the use of Proposition 3.3.2, which depends on the commutativity of the monad and strong sliding (of *mfix*). The first vanishing rule depends on left shrinking and purity, the second one also uses nesting. The superposing rule only needs pure right shrinking (which is guaranteed by right shrinking). Finally, yanking is a direct consequence of purity. □

Remark 6.3.6 Ideally, we should also establish that a trace operator on the Kleisli category of a commutative monad yields a value recursion operator, using a translation of the form:

$$\begin{array}{lcl} \mathit{mfix}_A & : & (A \to T\ A) \to T\ A \\ \mathit{mfix}\ f & = & \mathit{Tr}^{A}_{\mathbf{1},A}\ (\lambda(\_,\ a).\ f\ a \gg\!= \lambda a.\ \eta\ (a,a))\ () \end{array}$$

But we will refrain from pursuing the correspondence in this direction for the following reasons:

- Our treatment of value recursion operators takes place in the setting of continuous functions over domains. On the other hand, trace operators are presented in the abstract setting of monoidal categories, hence the assumptions for the underlying structure are significantly weaker. For instance, it is not clear what our strictness axiom (i.e., $f\ \bot_\alpha = \bot_{T\ \alpha}$ iff $\mathit{mfix}_\alpha\ f = \bot_{T\ \alpha}$) would correspond to in this setting.[4]

- As we explored above, the correspondence of traces and value recursion is rather limited. Very few monads are commutative, and even fewer have their Kleisli categories traced. What we should seek, then, is a notion of trace in the non-monoidal case. In short, trace axioms are just too strong for value recursion.

6.4 Dropping the monoidal requirement

As we have seen in the preceding section, the trace-based categorical account of fixed-point operators falls short of explaining value recursion for all but a very restricted set of monads. Is it possible to generalize the theory of traces so that we can accommodate value recursion more satisfactorily? In this section, we will briefly review two recent

[4] Hasegawa suggests that it might be possible to study strictness via the notion of *uniform trace operators*. See Proposition 7.1.4 in Hasegawa's thesis [33].

attempts in this direction. First, we will look at Paterson's work, which lifts *mfix* to the world of arrows [66]. Second, we will review Benton and Hyland's work on traced premonoidal categories [5]. It turns out that both attempts describe essentially the same axiomatization, although presented in slightly different contexts. The general idea is to move to premonoidal categories, effectively dropping the monoidal requirement.[5]

6.4.1 Arrows and *loop*

Hughes introduced arrows as a generalization of monads, making the input-output flow more explicit [36]. An arrow $\Rightarrow$ is a binary type constructor equipped with:

$$\begin{aligned} \mathit{arr} &: (\alpha \to \beta) \to (\alpha \Rightarrow \beta) \\ \ggg &: (\alpha \Rightarrow \beta) \to (\beta \Rightarrow \gamma) \to (\alpha \Rightarrow \gamma) \\ \mathit{first} &: (\alpha \Rightarrow \beta) \to (\alpha \times \gamma \Rightarrow \beta \times \gamma) \end{aligned}$$

Intuitively, $\alpha \Rightarrow \beta$ represents a computation that receives an input of type α, performs a computation with possible side effects, and delivers a result of type β, corresponding to what an imperative programmer might call a *procedure*. The morphism *arr* makes a procedure out of a pure function, while $\ggg$ runs two procedures in sequence, threading the result of the first to the second. The function *first* lets information to be passed around for later use, mainly used for storing results of intermediate computations. The morphisms *arr*, $\ggg$, and *first* are required to satisfy a number of laws, similar to monad laws.

Example 6.4.1 Arrows generalize monads in the following sense. For every monad m, the type *Kleisli* m gives rise to an arrow, where:

$$\begin{aligned} &\textbf{type}\ \mathit{Kleisli}\ m\ \tau\ \sigma = \tau \to m\ \sigma \\ &\mathit{arr}\ f \quad = \mathit{return} \cdot f \\ &f \ggg g = \lambda a.\ f\ a \gg\!= \ g \\ &\mathit{first}\ f \ \ = \lambda(a,\ c).\ f\ a \gg\!= \ \lambda b.\ \eta\ (b,\ c) \end{aligned}$$

Paterson argues that Power and Thielecke's Freyd categories are equivalent to Hughes's arrows [73]. (We will briefly review Freyd categories in Section 6.4.2.)

Is there a corresponding notion of value recursion for arrows? Paterson generalizes *mfix* to arrows, introducing the following *loop* operator [66]:

$$\mathit{loop} : (\alpha \times \gamma \Rightarrow \beta \times \gamma) \to \alpha \Rightarrow \beta \tag{6.22}$$

[5] We mention in passing that Jeffrey also used the so-called *partial* traces (ordinary traces that are restricted to be applied only to certain maps) to model flow graphs and recursion in programming languages [39]. We will not review his work here, however, as it is not directly related to value recursion.

Note the similarity between this type and the type of trace operators (Definition 6.2.3). As expected, value recursion operators give rise to loop operators for the corresponding Kleisli arrows. Given a value recursion operator *mfix*, Paterson defines:

$$\begin{array}{l} \mathit{loop}\ f\ =\ \mathit{map}\ \pi_1\ \cdot\ \mathit{mfix}\ \cdot\ f' \\ \quad\quad\mathbf{where}\ f'\ x\ y\ =\ f\ (x,\ \pi_2\ y) \end{array} \tag{6.23}$$

which can be shown to be equivalent to the function we have given for obtaining a *trace* operator from *mfix* (Equation 6.21).

Paterson generalizes the trace axioms of Section 6.2.2 for *loop*, and adds a law called *extension*, similar to our purity property. As expected, he weakens the sliding axiom so that the function moved over is of the form *arr* k for some function k, syntactically guaranteeing purity. Unlike our sliding property for *mfix*, however, Paterson does not require a further guarding equation to regulate the behavior on $\perp$ (i.e., the antecedent in Equation 2.5). Similarly, right tightening is postulated as an axiom as well. Therefore, the failure of strong sliding or right shrinking properties for the underlying *mfix* will cause the trace axioms to fail. Similar comments apply to arrows that are not derived from monads as well. Paterson makes similar observations, although he does not weaken his axiomatization to accommodate accordingly [66].

6.4.2 Traced premonoidal categories

Closely related to Paterson's work is Benton and Hyland's recent generalization of traces to premonoidal categories [5]. As we have seen throughout this chapter, the crux of the problem lies in the monoidal requirement that comes with traces. Motivated by this observation, Benton and Hyland generalize traces to premonoidal categories. If the category is indeed monoidal, their definition simply reduces to the usual definition of traces over monoidal categories.

Let us review the problem with the monoidal requirement more formally. What happens when the monad is not commutative? Let $\mathcal{C}$ be symmetric monoidal, with $\otimes$ as the monoidal operation. Let T be a strong monad over $\mathcal{C}$, with strength t. We do *not* assume that T is commutative. Consider the Kleisli category of T, $\mathcal{C}_T$. For clarity, we will use the symbol $\rightharpoonup$ to denote arrows in $\mathcal{C}_T$. The symmetry in $\mathcal{C}$ lifts into $\mathcal{C}_T$ with no problems, i.e., $\mathcal{C}_T$ is symmetric as well. For any fixed object A, we have the functor $A \otimes -$ in $\mathcal{C}_T$, mapping a given object B to $A \otimes B$ and an arrow $f : B \rightharpoonup B'$ to the arrow $t \cdot (A \otimes f) : A \otimes B \to T\ (A \otimes B')$ in $\mathcal{C}$, which corresponds to the required arrow $A \otimes B \rightharpoonup A \otimes B'$ in $\mathcal{C}_T$. It is easy to see that $- \otimes A$ also yields a functor in $\mathcal{C}_T$. However, $\otimes$ is not a bifunctor, unless T is commutative. To see this, let $f : A \rightharpoonup A'$ and $g : B \rightharpoonup B'$

be two arrows in $\mathcal{C}_T$. There are two ways of obtaining the arrow $f \otimes g : A \otimes B \rightharpoonup A' \otimes B'$, as captured by the following Haskell expressions:

$$\begin{aligned} &\lambda(a,\ b).\ f\ a \gg= \lambda a'.\ g\ b \gg= \ \lambda b'.\ \mathit{return}\ (a',\ b') \\ &\lambda(a,\ b).\ g\ b \gg= \lambda b'.\ f\ a \gg= \ \lambda a'.\ \mathit{return}\ (a',\ b') \end{aligned}$$

The first composition is denoted by $\ltimes$, i.e., $f \ltimes g = A' \otimes g \cdot f \otimes B$. Similarly, we define $f \rtimes g = f \otimes B' \cdot A \otimes g$. Unless the monad is commutative, these two compositions are generally different, as they reflect the order in which f and g are performed. This discrepancy is the main reason why the monoidal structure in the base category does *not* lift to a monoidal structure in the Kleisli category.

Of course, even when we only have a non-monoidal operation, there might exist a subset of arrows for which the order does not matter—think of f (or g) having the form $\mathit{return} \cdot h$ in the expressions above. Such arrows are called *central*. More formally, an arrow f is central if, for all g, $f \ltimes g = f \rtimes g$, and $g \ltimes f = g \rtimes f$.

Generalizing from this example, Power and Robinson introduced *premonoidal categories* [72]. In short, a premonoidal category is just like a monoidal category, except the binary operation is only required to be functorial in each of the variables separately. (Note that every monoidal category is trivially premonoidal.) As we have sketched above, Kleisli categories of strong monads are examples of premonoidal categories.

Given a symmetric premonoidal category, can we come up with a notion of trace? Recall that traces are only meaningful in symmetric monoidal categories. Naively, one might hope that the definition of trace (Definition 6.2.3) might very well apply in this case as well. Unfortunately this is not the case:

Proposition 6.4.2 (Benton, Hyland [5]) A symmetric premonoidal category with a trace (Definition 6.2.3) is actually monoidal. □

As expected, the sliding axiom causes the trouble. Benton and Hyland show that sliding implies $f \ltimes g = f \rtimes g$ for all arrows f and g, establishing that the category is indeed monoidal. To remedy the situation, Benton and Hyland generalize traces to *centered symmetric premonoidal categories.* A centered symmetric premonoidal category is a premonoidal category $\mathcal{K}$, with a distinguished monoidal center $\mathcal{M}$, and an identity-on-objects strict symmetric premonoidal functor $J : \mathcal{M} \to \mathcal{K}$ [72]. For our purposes, it suffices to consider $\mathcal{M}$ as a subcategory of $\mathcal{K}$, where all arrows in $\mathcal{M}$ are central.

Kleisli categories of strong monads over symmetric monoidal categories are classical examples of premonoidal categories. Let $\mathcal{M}$ be symmetric monoidal, and let T be a strong monad over $\mathcal{M}$. As we have seen above, $\mathcal{M}_T$ is symmetric premonoidal. Recall that a Kleisli category has the same objects as the base category. Let the functor $J : \mathcal{M} \to \mathcal{M}_T$

be defined as follows. On objects, J is the identity. Given an arrow $f : A \to B$, let $J\ f = \eta_b \cdot f : A \rightharpoonup B$, where η is the unit of T. Then $J : \mathcal{M} \to \mathcal{M}_T$ is a centered symmetric premonoidal category, with the distinguished monoidal center $\mathcal{M}$. (Of course, J is nothing but the usual inclusion functor.) In this case, a central arrow in $\mathcal{M}_T$ is simply any arrow that is lifted from the monoidal center, i.e., any arrow that factors through η in the base category.

The intuitive understanding of a centered symmetric premonoidal category $J : \mathcal{M} \to \mathcal{K}$ is as follows: $\mathcal{K}$ is considered to be the category where arrows denote computations, possibly with observable effects. As expected, $\mathcal{K}$ does not possess a monoidal structure. $\mathcal{M}$, on the other hand, is a subcategory of $\mathcal{K}$ denoting values, i.e., where we can swap the order of computations, duplicate values only to discard later, etc. A crude analogy in programming terms is given by any "almost" functional language: For instance, think of $\mathcal{K}$ as corresponding to the Standard-ML language, containing references etc., and $\mathcal{M}$ as the purely functional subset of Standard-ML.

Getting back to traces, Benton and Hyland define [5]:

Definition 6.4.3 (*Traced centered symmetric premonoidal categories.*) A trace on a centered symmetric premonoidal category $J : \mathcal{M} \to \mathcal{K}$ is a family of functions:

$$Tr^X_{A,B} : \mathcal{K}(A \otimes U, B \otimes U) \to \mathcal{K}(A, B)$$

satisfying the same conditions as given in Definition 6.2.3, except (i) the sliding condition is weakened such that g is assumed central, and (ii) given a central arrow $f : A \otimes X \to B \otimes X$, $Tr^X_{A,B}\ f : A \to B$ is required to be central.

It is easy to see that this definition generalizes the notion of trace, since all arrows are central in a symmetric monoidal category.

In order to generalize Theorem 6.2.4, Benton and Hyland also develop the notion of Conway operators on Freyd categories. Briefly, a *Freyd category* is a symmetric premonoidal category $J : \mathcal{C} \to \mathcal{K}$, where $\mathcal{C}$ is cartesian [73]. A *parameterized fixed point operator* on a Freyd category $J : \mathcal{C} \to \mathcal{K}$ is defined to be a family of functions

$$(\cdot)^*_{A,X} : \mathcal{K}(A \otimes X, X) \to \mathcal{K}(A, X) \tag{6.24}$$

Benton and Hyland require $(\cdot)^*$ to satisfy the so-called center preservation, naturality, and central fixed-point properties, corresponding to our left shrinking and purity laws.

To be able to establish a correspondence between traces over Freyd categories and parameterized fixed point operators, Benton and Hyland define Conway operators, which further satisfy laws that correspond to our right shrinking and nesting properties. Hence, similar to Paterson's axiomatization of *loop*, the correspondence with premonoidal traces only holds for the set of value recursion operators that further satisfy strong sliding and right shrinking properties. As we have seen in Section 3.1, these two properties are unsatisfiable for value recursion operators in general (Corollary 3.1.7).

6.5 Summary

In this chapter, we summarized the notion of traces from category theory, and investigated how value recursion might fit into the picture. As we have seen, for a very small class of monads, value recursion operators correspond to trace operators over Kleisli categories. The environment monad is the most important example exhibiting this correspondence (other than the obvious identity monad). In the general case, however, the correspondence fails because of the monoidal requirement in the formalization of trace operators.

It turns out that Paterson's *loop* axioms and Benton and Hyland's generalization of traces to premonoidal categories are essentially the same, although developed independently and presented in slightly different contexts [5, 66]. Both these axiomatizations take the correspondence one step further, but not to the point where a satisfactory theory for value recursion can emerge. To summarize, both require right shrinking and strong sliding properties, which are known to be unsatisfiable for many monads (see Chapter 3). In terms of concrete monads, their work can handle the lazy state and the output monads, but not exceptions, lists, strict state, and the IO monad of Haskell. In this respect, we consider both attempts to be significant steps in understanding and generalizing value recursion, but not the final categorical account of the whole problem.

Chapter 7

A recursive do-notation

Haskell's do-notation simplifies monadic programming significantly, but it lacks support for recursive bindings, a key syntactic feature for value recursion. In this chapter, we describe an enhanced translation schema for the do-notation and its integration into Haskell.[1] The new translation will allow variables to be bound recursively, provided the underlying monad comes equipped with a value recursion operator.

Synopsis. We start with a motivating example, showing the need for recursive bindings in the do-notation. The issues related to let-generators and the need for segmentation are discussed next, followed by a detailed description of the translation algorithm. We also provide several comments on the integration of the new do-notation into Haskell.

7.1 Introduction

Recursive declarations are ubiquitous in the functional paradigm. While fixed-point operators provide a solid framework for reasoning about and understanding recursion, they are hardly suitable for practical programming tasks. For instance, compare:

$$\mathbf{let}\ \mathit{sum}\ n = \mathbf{if}\ n == 0\ \mathbf{then}\ 0\ \mathbf{else}\ n + \mathit{sum}\ (n - 1)\ \mathbf{in}\ \mathit{sum}\ 10$$

to its non-recursive equivalent:

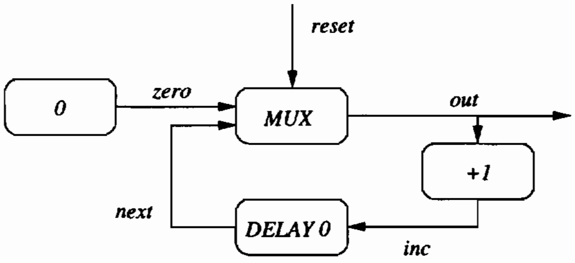
$$\mathbf{let}\ \mathit{sum} = \mathit{fix}\ (\lambda f.\lambda n.\ \mathbf{if}\ n == 0\ \mathbf{then}\ 0\ \mathbf{else}\ n + f\ (n - 1))\ \mathbf{in}\ \mathit{sum}\ 10$$

Clearly, the use of *fix* makes the definition much harder to read, especially for beginning programmers. The situation gets worse with mutually recursive bindings.

As we have briefly mentioned in Section 1.3, a similar problem arises in the framework of value recursion. Rather than using explicit calls to *mfix*, we would like to have a complementary binding construct, providing syntactic support for value recursion. In the

[1] The material in this chapter is based on a paper that appears in the *Haskell Workshop'02* [19].

context of Haskell, an extension to the do-notation allowing recursive bindings seems to fit the bill. To illustrate, we will revisit the circuit modeling example from Section 1.2. This time, we will model a simple counter, one that increments its output by 1 at each clock tick. The count goes back to 0 whenever the *reset* line goes high:



By extending the *Circuit* class (see Section 1.2) with multiplexers and monadic lift functions, we can model this circuit monadically as follows:

```
counter       :: Circuit m ⇒ Sig Bool → m (Sig Int)
counter reset = mfix (λ~(next, inc, out, zero).
                          do next ← delay "zero" 0 inc
                             inc  ← lift1 "add1" (+1) out
                             out  ← mux reset zero next
                             zero ← lift0 "zero" 0
                             return (next, inc, out, zero))
                     ≫= λ(next, inc, out, zero). return out
```

As we have argued in Section 1.2, the monadic implementation has numerous advantages. Syntactically, however, it carries a lot of baggage, making it hard to understand and maintain. (Note that binders can be arbitrary patterns in general, as in "*Just x ← f x*", making the situation even worse.) As pointed out by Launchbury et al. [49], and as we have outlined in Section 1.3, what we need is a recursive counterpart of the do-notation, allowing us to write simply [49]:

```
counter reset = do next ← delay "zero" 0 inc
                   inc  ← lift1 "add1" (+1) out
                   out  ← mux reset zero next
                   zero ← lift0 "zero" 0
                   return out
```

eliminating the explicit call to *mfix*. Note that this description of the circuit follows the diagram given above almost literally. The translation we will introduce in this chapter will handle such recursive definitions automatically, without bothering programmers with the details of the necessary plumbing.

7.2 The basic translation and design guidelines

For clarity, we refer to the recursive version of the do-notation as the *mdo-notation*, and write mdo-expressions using the keyword **mdo**.[2] Whenever we refer to the do-notation, we mean the currently available notation in Haskell that does not allow variables to be bound recursively.

Inspired by the counter circuit example of the previous section, one might naively translate mdo-expressions as follows:

$$\begin{array}{lcl} \textbf{mdo}\ p_1 \leftarrow e_1 & & \mathit{mfix}\ (\lambda\,\tilde{}BV.\ \textbf{do}\ p_1 \leftarrow e_1 \\ \qquad \ldots & \Longrightarrow & \qquad\qquad .. \\ \qquad p_n \leftarrow e_n & & \qquad\qquad p_n \leftarrow e_n \\ \qquad e & & \qquad\qquad \mathit{return}\ BV) \\ & & \quad \gg\!= \lambda BV.\ e \end{array}$$

where BV stands for the tuple consisting of all variables occurring in patterns $p_1 \ldots p_n$. The lazy match, obtained by ~, is essential in avoiding strictness problems.

However, there are a number of problems raised by the schema above. First of all, do-expressions in Haskell can use let-generators to introduce polymorphic bindings for pure expressions [68]. It is not clear how such bindings can be integrated into this translation. Similarly, ordinary do-expressions can bind identifiers repeatedly, later bindings shadowing earlier ones. When bindings can be recursive, shadowing becomes problematic. Furthermore, the use of a single *mfix* to handle recursion over the entire body of an mdo-expression may induce poor termination properties whenever the right-shrinking laws fails (see Section 7.2.2)—intuitively, recursion should only be performed over generators that depend on each other cyclically, leaving the rest untouched. Finally, we would like to address these issues within the boundaries of the "syntactic-sugar" approach. That is, the translation should produce only valid (well-formed and well-typed) Haskell code. This approach keeps the extension simple, providing a smooth transition.

To summarize, the basic design guidelines for the mdo-notation are:

- Syntactic agreement with the do-notation: Programmers familiar with the do-notation should have no trouble using the recursive version.

- Semantic agreement with the do-notation: To the extent possible, valid do-expressions should also be valid mdo-expressions, with their meanings preserved.

- Segmentation: Calls to *mfix* should be isolated to recursive segments only, leaving the non-recursive parts out of the fixed-point computation. As we will see, segmentation

[2] The closest we can get to μ**do** using ASCII.

is essential because extending the scope of recursion can give poorer results for those monads that fail to satisfy the right shrinking property.

- Pure syntactic sugar: The translation should only produce well-formed and well-typed Haskell code.

In the remainder of this section, we address these issues, refining the basic translation scheme as we go along.

7.2.1 Let generators

The do-notation of Haskell allows let-generators, with the following translation [68]:

```
do let p1 = e1                 let p1 = e1
       ...                         ...
                      ==>
       pn = en                     pn = en
   stmts                       in do stmts
```

The variables bound in $p_1 \ldots p_n$ can be polymorphically typed. In mdo-expressions, these variables should be visible throughout the entire body as well, suggesting the translation:

```
                              mfix (λ~BV. do stmts1
mdo stmts1                                   let p1 = e1
    let p1 = e1                                  ...
        ...                ==>                   pn = en
        pn = en                              stmts2
    stmts2                                   return BV)
    e                           >>= λBV. e
```

where the variables bound in $p_1 \ldots p_n$ will appear in BV as well. Unfortunately, the resulting code is not guaranteed to be well-typed. To illustrate, consider:

```
                                  mfix (λ~(z, y, f).
mdo z <- f 2 y                          do z <- f 2 y
    y <- f 'a' z                           y <- f 'a' z
                            ==>
    let f x _ = return x                   let f x _ = return x
    return (f y z, f z y)                  return (z, y, f))
                                    >>= λ(z, y, f). return (f y z, f z y)
```

Since f is λ-bound, it becomes monomorphically typed, making its use at two different types illegal. In fact, the situation is even worse: Referring to the schematic translation above, let-bound variables in patterns $p_1 \ldots p_n$ will have monomorphic types over $stmts_1$

and e, while they will retain their polymorphic typings over $stmts_2$ and $e_1 \dots e_n$. This situation is quite bizarre. Unfortunately, there is no easy solution to this problem. Since the tuple BV is λ-bound, the variables that appear in it will be monomorphically typed when we attempt to type check the body of the do-expression and the final expression e.

How should we deal with this problem? Clearly, it is unacceptable to ban let-generators completely because they are quite useful in practice. (Requiring let-bound variables to be visible only in the textually following generators would also be wrong.) An alternative is to go slightly beyond Haskell 98, using records with polymorphically typed fields [40]. Rather than using tuples, we can package the arguments into a record with polymorphic fields, retaining the polymorphic typings of let-bound variables. However, the resulting translation is overly complicated (as we need to perform type inference during the translation), making it hard to formalize and automate [17]. One might also argue that we can go beyond the "syntactic-sugar" approach, i.e., let the translation produce ill-typed code, provided we can come up with special typing rules for mdo-expressions. We will not pursue this option here, however, in order to be able to keep the translation as simple as possible. (We will return to this point in Section 7.3.3.)

The solution we adopt is to require let bindings to be monomorphic in mdo-expressions. That is, **let** becomes just a syntactic sugar within **mdo**, translated as:

$$\begin{array}{l} \textbf{let}\ p_1 = e_1 \\ \quad\ \dots \\ \quad\ p_n = e_n \end{array} \quad \Longrightarrow \quad \begin{array}{l} BV \leftarrow \mathit{return}\ (\textbf{let}\ p_1 = e_1 \\ \qquad\qquad\qquad\quad\ \dots \\ \qquad\qquad\qquad\quad\ p_n = e_n \\ \qquad\qquad\qquad\ \textbf{in}\ BV) \end{array}$$

where BV is the tuple corresponding to the variables bound in $p_1 \dots p_n$. This idea easily extends to more complicated forms of function definitions as well. For instance:

$$\begin{array}{l} \textbf{mdo let}\ f\ [\,] \qquad\ = 0 \\ \qquad\quad\ f\ (x{:}xs) = 1 + f\ xs \\ \quad\ \mathit{return}\ (f\ [1,2,3],\ f\ [\,]) \end{array} \quad \Longrightarrow \quad \begin{array}{l} \textbf{mdo}\ f \leftarrow \mathit{return}\ (\textbf{let}\ f\ [\,] \qquad\ = 0 \\ \qquad\qquad\qquad\qquad\quad f\ (x{:}xs) = 1 + f\ xs \\ \qquad\qquad\qquad\quad\ \textbf{in}\ f) \\ \qquad\ \mathit{return}\ (f\ [1,2,3],\ f\ [\,]) \end{array}$$

Note that we do not commit to a specific monomorphic type for f. As long as f is used consistently at a single monomorphic type, the translation will be well-typed.

We expect this restriction to be negligible in practice. Such polymorphic let-generators are hardly ever used in practice, and experience suggests that there is almost always an obvious way to rewrite the required polymorphic bindings using an explicit let-expression, avoiding the whole problem. Therefore, we believe that the simplicity of this design far outweighs any generality that might be obtained by more complicated translation schemas.

Remark 7.2.1 It might help programmers if monomorphic bindings were visually distinguishable from polymorphic ones. In a recent paper, Hughes argues that the syntax of let-expressions should be extended to allow monomorphic bindings, suggesting the use of the symbol := to differentiate them from polymorphic ones [34]. If this idea ever gets adopted in Haskell, let-generators in mdo-expressions can be restricted to use := as well, emphasizing the fact that they will be monomorphically typed.

7.2.2 Segmentation

Consider the following mdo-expression, which creates two infinite lists consisting of 1's and 2's respectively, and its translation:

```
                                       mfix (λ~(ones, twos).
mdo putStr "all 1s"                             do putStr "all 1s"
    ones ← return (1 : ones)                       ones ← return (1 : ones)
    putStr "all 2s"              ⟹                 putStr "all 2s"
    twos ← return (2 : twos)                       twos ← return (2 : twos)
    putStr "done"                                  return (ones, twos))
                                       >>= λ(ones, twos). putStr "done"
```

The resulting code is quite unsatisfactory. The only recursion we need is in independently computing the lists *ones* and *twos*, suggesting a segmented translation of the form:

```
do putStr "all 1s"
   ones ← mdo ones ← return (1 : ones)
              return ones
   putStr "all 2s"
   twos ← mdo twos ← return (2 : twos)
              return twos
   putStr "done"
```

where the inner mdo-expressions will further be translated accordingly. This process is analogous to the handling of ordinary let-expressions in Haskell, where mutually dependent bindings are grouped together to enhance types of bound variables [68]. In our case, all variables are λ-bound, i.e., monomorphic, so typing is not an issue. However, we still need segmentation to avoid the unwanted interference from trailing computations. As an example, let

```
checkSingle     :: [Int] → IO ()
checkSingle [x] = putStr "singleton"
checkSingle _   = putStr "not-singleton"
```

and consider the following translation:[3]

$$\begin{array}{l} \textbf{mdo}\ xs \leftarrow \mathit{return}\ (1 : xs) \\ \qquad \mathit{checkSingle}\ xs \\ \qquad \mathit{return}\ () \end{array} \quad \Longrightarrow \quad \begin{array}{l} \mathit{fixIO}\ (\lambda xs.\ \textbf{do}\ xs \leftarrow \mathit{return}\ (1 : xs) \\ \qquad\qquad\qquad\quad \mathit{checkSingle}\ xs \\ \qquad\qquad\qquad\quad \mathit{return}\ xs) \\ \qquad \gg\!= \ \lambda xs.\ \mathit{return}\ () \end{array}$$

Intuitively, we expect this mdo-expression to print "`not-singleton`", as the value of xs should simply be the infinite list of 1's. Alas, the translation will diverge! The reason is simply that the pattern matching in *checkSingle* is too strict for the computation to proceed, failing the match immediately. However, with segmentation, we will get the code:

$$\begin{array}{l} \textbf{do}\ xs \leftarrow \mathit{fixIO}\ (\lambda xs.\ \mathit{return}\ (1 : xs)) \\ \qquad \mathit{checkSingle}\ xs \\ \qquad \mathit{return}\ () \end{array}$$

which will happily print "`not-singleton`", avoiding the unintended interference. Interestingly, if the final "*return* ()" is omitted, the original translation will work as well, since the call to *checkSingle* will be the final expression, automatically pushed outside of the *mfix* loop. Just adding "*return* ()" should not change the result, pointing out the need for segmentation. Note that this problem will arise whenever right shrinking fails (Section 2.7.2), which is the case for many practical monads of interest. (See Corollary 3.1.7.)

7.2.3 Shadowing

The current syntax of do-expressions allows variable names to be bound repeatedly, later bindings shadowing earlier ones. One can accommodate such bindings in the mdo-notation as well, by appropriately renaming them. As a design choice, however, we reject this possibility. Although shadowing might be convenient at times, it is also a constant source of bugs. Since bound variables are visible throughout the entire body in an mdo-expression, allowing repetitions is much more likely to cause confusion.[4] Therefore, we disallow shadowing in mdo-expressions. (This design choice also implies that the scoping rules for mdo-expressions are the same as those for let and where expressions, providing a consistent view of scoping in Haskell's binding constructs, both pure and monadic.)

[3] As we will see in Chapter 8, the library function $\mathit{fixIO} :: (\alpha \to IO\ \alpha) \to IO\ \alpha$ is the value recursion operator for Haskell's IO monad [20].

[4] In a similar vein, it can be argued that repetitions should not have been allowed in the do-notation either. List comprehensions become especially horrible: $f\ x = [x \mid x \leftarrow [x\ ..\ x{+}5],\ x \leftarrow [x\ ..\ x{+}10]]$ is a confusing (yet legal) Haskell function.

7.3 Translation of mdo-expressions

We now present an algorithm to translate mdo-expressions to core Haskell.

7.3.1 Preliminaries

In the following discussion, we assume that let-generators are already de-sugared into their *return* equivalents, as described in Section 7.2.1. We use the meta-variable p to range over patterns, v over variables, and e over expressions.

Definition 7.3.1 (*Defined variables.*) A generator $p \leftarrow e$ defines the variables that appear in the pattern p. If the generator is of the form e, i.e., without any binding patterns, then it defines no variables. An mdo-expression m defines a variable v, if v is defined in a generator of m.

Definition 7.3.2 (*Used variables.*) A defined variable v is used in a generator $p \leftarrow e$ if v occurs free in e. (And similarly when there is no binding pattern p.)

Definition 7.3.3 (*Recursive variables.*) Let m be an mdo-expression, and v be a used variable of m. Let g be the generator that defines v. The variable v is recursive if it is either used by g itself, or by a generator of m that appears textually before g.

Remark 7.3.4 Every defined variable comes from a distinct generator, due to the no-repetition requirement. Furthermore, only defined variables can be used, and only used variables can be recursive. That is, for an arbitrary mdo-expression, we have:

$$\text{Recursive Variables} \subseteq \text{Used Variables} \subseteq \text{Defined Variables}$$

Definition 7.3.5 (*Dependent generators.*) A generator g is dependent on a textually following generator g', if

- g' defines a variable that is used by g,
- or, g' textually appears in between g and g'', where g is dependent on g''.

Remark 7.3.6 The second condition in the above definition can be considered as interval closure. Note that, unlike a usual let-expression, we cannot reorder the generators: Order does matter in performing side effects. Hence, if a generator is dependent on another, we are forced to package them together with all the generators in between.

Definition 7.3.7 (*Segments.*) A segment of an mdo-expression is a minimal sequence of generators such that no generator of the sequence depends on an outside generator. As a special case, although it is not a generator, the final expression in an mdo-expression is considered to form a segment by itself.

Remark 7.3.8 To compute the segments, it suffices to start with the first generator of an mdo-expression, and search for the last generator that it depends on. If such a generator exists, we add all the generators up to and including it to the segment. This process is repeated for each and every one of the generators in the segment, until we cannot add any new generators. Once a segment is found, the very next generator starts a new segment. Note that the number of segments is bounded above by the number of generators in the mdo-expression, plus one for the segment corresponding to the final expression.

Definition 7.3.9 (*Free variables of a segment.*) Let m be an mdo-expression, v be a defined variable, and s be a segment of m. We say that v is free in s if (i) v appears free in the right hand side of a generator of s, and (ii) v is defined in a segment textually preceding s.

Definition 7.3.10 (*Exported variables of a segment.*) A variable that is defined in a segment is exported if it is free in any of the textually following segments.

7.3.2 The translation algorithm

We describe the algorithm step by step using the following schematic running example:

$$\begin{array}{llllll}
\textbf{mdo} & \{a\ b\} & \leftarrow & \{c\ d\} & \qquad\qquad & s_0 \\
 & \{e\} & \leftarrow & \{f\} & & s_1 \\
 & \{g\} & \leftarrow & \{h\} & & s_2 \\
 & \{f\} & \leftarrow & \{a\} & & s_3 \\
 & \{i\ j\} & \leftarrow & \{i\ e\} & & s_4 \\
 & \{j\ g\ k\} & & & & s_5
\end{array}$$

where $\{v_1 \ldots v_n\}$ stands for a pattern that binds the variables $v_1 \ldots v_n$ on the left hand side of a generator, and for an expression whose free variables are $v_1 \ldots v_n$ on the right hand side. Note that the actual patterns or expressions are not important for our purposes. For instance, the generator s_3 uses the variable a, and defines f. Generator s_2 defines g, but does *not* use h, since h is not defined in this expression. For our purposes, it is nothing but a constant. Similar remarks apply to the variables c, d, and k as well.

Segmentation step: Starting with the first generator, form the segments as described in Remark 7.3.8.

To perform this step, we will need the defined (D_i) and used variables (U_i) of each generator s_i. Luckily, for our running example, these sets are obvious:

$$\begin{array}{llllll} D_0 = \{a, b\} & D_1 = \{e\} & D_2 = \{g\} & D_3 = \{f\} & D_4 = \{i, j\} & D_5 = \emptyset \\ U_0 = \emptyset & U_1 = \{f\} & U_2 = \emptyset & U_3 = \{a\} & U_4 = \{i, e\} & U_5 = \{j, g\} \end{array}$$

To compute the segments, we start with s_0. Since s_0 does not use any variables, it cannot depend on other generators, i.e., it forms a segment by itself. The next generator to consider is s_1, which uses the variable f. Since f is defined by s_3, we have to package everything in between, i.e., s_1, s_2, and s_3 together. Since none of them depends on s_4 or s_5, we stop the iteration, forming our second segment. It is easy to see that s_4 and s_5 form the next two segments by themselves. Hence, we obtain:

$$S_0 = \{s_0\},\ \ S_1 = \{s_1, s_2, s_3\},\ \ S_2 = \{s_4\},\ \ S_3 = \{s_5\}$$

Analysis step: For each segment S_i do the following: For each variable v defined in the segment, determine whether it is recursive (Definition 7.3.3). Collect all recursive variables of the segment S_i in the set R_i. If R_i is empty, this segment does not need fixed-point computation, leave it untouched. If R_i is not empty, compute the exported variables of the segment, E_i, and mark this segment as *recursive* for future processing. Returning to our example, we have:

$$R_0 = \emptyset \qquad R_1 = \{f\},\ E_1 = \{e, g\} \qquad R_2 = \{i\},\ E_2 = \{j\} \qquad R_3 = \emptyset$$

Since only R_1 and R_2 are non-empty, we mark S_1 and S_2 as recursive; other segments are left untouched. (Note that the last segment can never be recursive.)

Translation step: At this point, we are left with a number of segments, some of which are marked recursive by the previous step. For each marked segment, create the tuples ET and RT corresponding to the sets E and R. (If E is empty, ET will be the empty tuple.) Create and add a brand new variable v to the tuple RT. Then, form the generator:

```
ET ← mfix (λ˜RT. do .....
                 .....
                 v ← return ET
                 return RT)
      ≫= λRT. return v
```

where the dotted lines are filled with the generators of the segment.

Note that segments that are marked recursive by the previous step are turned into a single generator, while non-recursive segments are left untouched.[5] Returning to our example, we create the following generator for S_1:

```
(e, g) ← mfix (λ~(f, v). do {e} ← {f}
                            {g} ← {h}
                            {f} ← {a}
                            v   ← return (e, g)
                            return (f, v))
                   ≫= λ(f, v). return v
```

and the following for S_2:

```
j ← mfix (λ~(i, v). do {i j} ← {i e}
                       v     ← return j
                       return (i, v))
              ≫= λ(i, v). return v
```

Finalization step: Now, concatenate all segments and form a single do-expression out of them. For our example, we obtain:

```
do {a b} ← {c d}
   (e, g) ← mfix (λ~(f, v). do {e} ← {f}
                               {g} ← {h}
                               {f} ← {a}
                               v   ← return (e, g)
                               return (f, v))
                      ≫= λ(f, v). return v
   j      ← mfix (λ~(i, v). do {i j} ← {i e}
                               v     ← return j
                               return (i, v))
                      ≫= λ(i, v). return v
   {j g k}
```

Remark 7.3.11 If there are no recursive bindings present to start with, the algorithm we have described will just leave the input untouched (except for replacing the keyword **mdo** by **do**). That is, the left shrinking property is automatically applied by the algorithm to get rid of unnecessary calls to *mfix*. (See Section 2.3.)

[5] Depending on the sets E and R, several other improvements are possible in forming the required generator. For instance, if E is a subset of R, then we do not need a new variable. We skip a detailed discussion of these improvements here, as they are not essential for the translation.

Desugaring step: Now we are left with a non-recursive do-expression, and we can apply the standard translation to replace the **do** with explicit ≫='s, completing the translation [68].

7.3.3 Type checking mdo-expressions

To accommodate for the overloading of the name *mfix*, we simply add the following type class to Haskell:

```
class Monad m ⇒ MonadFix m where
      mfix :: (α → m α) → m α
```

Intuitively, an mdo-expression is well-typed if its translation produces a well-typed Haskell expression. In order to perform type-inference, a type judgement of the form:

$$\frac{\Gamma' \vdash e_i : m\ \tau_i \qquad \Gamma' \vdash p_i : \tau_i \qquad \Gamma' \vdash e : m\ \tau}{\Gamma \vdash \mathbf{mdo}\ \{p_i \leftarrow e_i\}\ e : m\ \tau}$$

suffices, with the side condition that m must belong to the *MonadFix* class. In this rule, Γ' is obtained by extending Γ with the variables defined in the given mdo-expression. Each such variable is assigned a monomorphic type variable to begin with. (For simplicity, we assume all generators have the form $p \leftarrow e$.) The only special care is needed in handling let-generators, which can be typed similarly to normal let-expressions. To ensure that let-bound variables are monomorphic, it suffices to leave out the generalization step in the type inference algorithm for let-bound variables [17, 41].

As we have promised in Section 7.2.1, let us reconsider the typing of let-generators, aiming to find a solution that would allow polymorphic bindings. In fact, it is arguable that we should have a more liberal scheme, where normal bindings can be polymorphic as well. For instance, there is no reason why the following expression should be ill-typed:

```
poly :: Maybe ([Bool], [Int])        -- ill−typed
poly = do nil ← return []
          return (True : nil, 1 : nil)
```

However, *poly* is not a well-typed Haskell expression, since the binding to *nil* is required to be monomorphic. Of course, we cannot allow polymorphic typings arbitrarily, as illustrated by the infamous ML-typing problem [93], coded here in Haskell:

```
do rf ← newSTRef (λx. x)
   writeSTRef rf (λx. x + 1)
   f  ← readSTRef rf
   return (f True)
```

Following the previous example, we might think that *rf* might be assigned the type $\forall\alpha.\ STRef\ s\ (\alpha \to \alpha)$, which leads to disaster. So, it seems that the *maybe* monad is mild enough that generalization is acceptable, but the state monad is not. It is beyond the scope of our current work to investigate exactly when one might allow generalization, but we conjecture that it is safe to do so in the following two cases:

- For any variable, provided the underlying monad is completely definable in Haskell, and not built on top of one of the internal state or IO monads,
- Or, variables bound by the let-generators, regardless of the underlying monad.

Since checking for the first condition seems to be rather expensive, we might settle for allowing generalization in let-bound variables only, which coincides with the treatment of let-generators in the current do-notation. (Such a solution would be similar to ML's value restriction, where only "syntactically distinguishable" values are typed polymorphically [93].) Of course, a more detailed study is needed before such an approach can be adopted. We leave the exploration of this idea for future work.

7.4 Current status and related work

The mdo-notation is implemented both by the Hugs interpreter [37] and the GHC compiler [26]. Details on these implementations can be found on the web [74].

Predating our work, the need for recursive bindings in the do-notation was also discussed in the framework of Nordlander's O'Haskell language, a concurrent, object-oriented extension to Haskell [65]. O'Haskell extends the do-notation with a variety of new features. With regard to recursion, O'Haskell provides a special keyword **fix**, providing a way to specify a block of generators with mutual dependencies. The translation for fix-blocks is a simpler version of ours: No segmentation is performed and let-generators are not allowed. The translation seems to permit shadowing, but that appears to be an oversight, rather than a conscious design decision. The addition of the **fix** keyword to the do-notation in O'Haskell arose from practical programming needs; the syntax and the translation was not designed to meet a general need.

Paterson's arrow-notation supports recursive bindings as well, provided the underlying arrow comes equipped with a *loop* operator [66]. (See Section 6.4.1 for a discussion of arrows and *loop* operators.) Similar to O'Haskell, mutually dependent generators are explicitly marked, using the keyword **rec**. No segmentation is performed on recursive blocks. Currently, let-generators are not supported in the arrow-notation, but the addition of such bindings seems straightforward. We note that all variables become λ-bound after the translation in the arrow-notation, forcing monomorphic types. Hence, regardless of the support for recursive bindings, let-generators will suffer from the monomorphism problem in the arrow-notation.

7.5 Summary

In this chapter, we have described an alternative translation schema for the do-notation of Haskell, providing syntactic support for recursive bindings. The ability to bind variables recursively in the do-notation is an essential feature for value recursion as it elegantly hides the use of explicit value recursion operators.

Recalling the design goals we have set for the mdo-notation, we can conclude that our translation fulfills its purpose. To review briefly, we have aimed for syntactic and semantic agreement with the do-notation, segmentation for grouping minimally dependent sequences of statements together, and preservation of the syntactic-sugar status. Our translation achieves all these goals, except for syntactic agreement for a relatively small set of do-expressions. Since let-generators become monomorphic and shadowing is no longer allowed, any do-expression using these features will be rejected. However, we believe that neither of these restrictions will cause serious problems in practice. Also, if desired, the typing problem might be remedied by devising a solution along the lines we have described in Section 7.3.3.

It is our belief that Haskell should have just one version of the do-notation. Just like let-expressions, do-expressions should be capable of expressing both recursive and non-recursive bindings. (The type system will insist on the *MonadFix* instance only when recursive bindings are used.) However, such a change will potentially break existing programs, due to the minor incompatibilities mentioned above. Therefore, a separate notation (using the keyword **mdo**) has been adopted for the time being, possibly switching to the new translation in a future version of the Haskell standard.

Chapter 8

The IO monad and *fixIO*

The IO monad of Haskell comes equipped with a value recursion operator, namely the function *fixIO*.[1] Both the IO monad and *fixIO* are language primitives in Haskell, i.e., they cannot be defined within the language itself. Therefore, any attempt to formally reason about *fixIO* is futile without a viable semantics for computations in the IO monad. Recently, Peyton Jones introduced an operational semantics based on observable transitions as a method for reasoning about I/O in Haskell [67]. In this chapter, we build on his framework, and show how one can model *fixIO* as well.[2]

Synopsis. We start with a brief discussion of the operation of *fixIO*, showing how it fits within the rest of the IO monad. We then describe a core language based on Haskell, with basic monadic I/O facilities. We continue by giving a layered semantics for this language. Finally, we show that our model of *fixIO* satisfies the requirements for being a value recursion operator with respect to our semantics.

8.1 Introduction

Ever since Peyton Jones and Wadler showed how monads can be used to model I/O in a language with non-strict semantics, monadic I/O became the standard way of dealing with input and output in Haskell [69]. The IO monad in Haskell comes equipped with a value recursion operator, namely the function *fixIO*. As Achten and Peyton Jones point out, and as with all value recursion operators, *fixIO* "*... allows us to manipulate results [of IO computations] that are not yet computed, but lazily available*" [1, Section 4.1].

Unlike many other monads, the IO monad of Haskell is built into the language, as it cannot be defined within Haskell itself. As a consequence, *fixIO* is a language primitive

[1] The function *fixIO* is not part of the standard Haskell library [68]. Implementations, including Hugs and GHC, provide it generally in the *IOExts* library.

[2] This chapter is based on a paper that will appear in the *Journal of Theoretical Informatics and Applications* [21]. A preliminary version of the material presented in this chapter appeared in the *Fixed Points in Computer Science Workshop'2001* [20].

as well. Given we do not have direct access to the internals of the IO monad, how can we understand the operation of *fixIO*? Or, in general, how can we understand IO-based computations? Recently, Peyton Jones introduced a semantics for Haskell IO [67], similar to the monadic transition systems of Gordon [27]. In such a system, IO computations are viewed as sequences of labeled transitions. Each label indicates an effect observable in the real world, similar to those found in process calculi [61]. Peyton Jones's work used an embedding of a denotational semantics for the functional layer into the IO layer. However, it bypassed the details of this embedding. Such an approach is fine, as long as one is interested in the big picture. If, on the other hand, one wants to reason about *fixIO*, it becomes necessary to be explicit about the relationship between the IO and functional layers. One aim of this chapter is to bridge this gap.

Our semantics is structured in two layers: IO and functional. The semantics for the IO layer is based on the approach taken by Peyton Jones [67]. The semantics for the functional layer is based on the natural semantics for lazy evaluation of Launchbury [48]. A final set of rules precisely shows how these two layers interact with each other. It is this interaction that allows us to give a semantics for *fixIO*. (The material in this chapter builds directly on Peyton Jones's and Launchbury's work mentioned above. We assume that the reader is already familiar with these papers.)

8.2 Motivating examples

Although *fixIO* is just like any other value recursion operator we have seen so far, the fact that we cannot give a Haskell definition for it makes it rather mysterious. Also, the IO monad provides mutable variables, a feature that we will have to deal with explicitly. We start by considering several examples to get familiar with the operation of *fixIO*.

Example 8.2.1 Our first example shows the interaction of *fixIO* with input operations:

$$\begin{array}{l} \mathit{fixIO}\ (\lambda cs.\ \mathbf{do}\ c \leftarrow \mathit{getChar} \\ \qquad\qquad\qquad\qquad\ \ \mathit{return}\ (c : cs)) \end{array}$$

When we run this computation, a character will be read from the standard input, say a. Then, the computation will immediately deliver an infinite list of a's.[3] We will be able to pull out as many characters as we wish out of this list, following the demand-driven evaluation policy of Haskell. There are two crucial points: (i) the action *getChar*

[3] Note that, by applying the left shrinking and purity properties, we can reduce this expression to $\mathit{getChar} \gg\!= \lambda c.\ \mathit{return}\ (\mathit{fix}\ (\lambda cs.\ c{:}cs))$, guaranteeing the described behavior axiomatically. Of course, we have not yet established that these two properties hold for *fixIO*, but we will do so in Section 8.6.

is executed only once, and (ii) the computation terminates immediately after the reading is done, i.e., the infinite list is not constructed prior to its demand. In other words, the fact that the IO monad is strict in actions but not in values is preserved by *fixIO*.

Here, we also get a feel for what *fixIO* provides: It provides a means for recursively defining values resulting from IO computations. That is, it allows naming results of computations that will only be available later on. For instance, in the expression above, we were able to name the result of the computation as *cs*, before we had its value computed. In this sense, the semantics is similar to the semantics of the pure expression:

$$\mathbf{let}\ cs\ =\ \text{'a'}\ :\ cs\ \mathbf{in}\ cs$$

which is a convenient way of writing $\mathit{fix}\ \ (\lambda cs.\ \text{'a'}\ :\ cs)$, where *fix* is the usual fixed-point operator. Except, of course, in the *fixIO* case the character in the list is determined by the call to *getChar*, i.e., it depends on the actual input available when we run the computation.

Example 8.2.2 Let us revisit the *fudgets* example given by Expression 4.35. In terms of *fixIO*, the corresponding computation is given by:

$$\begin{array}{l} \mathit{fixIO}\ \ (\lambda c.\ \mathbf{do}\ \mathit{putChar}\ \ c \\ \qquad\qquad\qquad\quad \mathit{return}\ \ \text{'a'}) \end{array}$$

When run, this computation diverges as c is not yet available when requested by *putChar*. (Note that this behavior is in accordance with *mfix* as discussed in Section 4.8.)

Example 8.2.3 Here is a Haskell expression showing the interaction of *fixIO* with mutable variables:

$$\begin{array}{l} \mathit{fixIO}\ \ (\lambda\,\tilde{}\,(x,\ \_).\ \mathbf{do}\ y \leftarrow \mathit{newIORef}\ \ x \\ \qquad\qquad\qquad\qquad\qquad \mathit{return}\ \ (1{:}x,\ y)) \\ \quad \gg\!= \ \lambda(\_,\ l).\ \mathit{readIORef}\ \ l \end{array}$$

In this expression, we allocate a cell in which we store the value of the variable x, before we know what that value really is. The value of x, determined through the fixed point computation, is the infinite list of 1's. The call to *fixIO* returns the value (which is discarded) and the address of the cell that stores this cyclic structure. Outside of the call to *fixIO*, we dereference the address and get back the lazily computed list of 1's. Although this example might look superficial, it basically captures the essence of cyclic structures with mutable nodes. (See Section 9.4 for an example, where we use a similar idea to implement doubly linked circular lists in Haskell.)

Once we describe our semantics, we will revisit these examples to see how our system works in practice.

8.3 The language

In this section, we define a language based on Haskell [68], supporting monadic IO primitives, mutable variables, usual recursive definitions, and value recursion.

Notation 8.3.1 We use the following naming conventions for variables:

$$\begin{array}{rcl} c & \in & \textit{constructors} \\ x, y, z, w & \in & \textit{heap variables} \\ r, s, t & \in & \textit{mutable variables} \end{array}$$

To simplify the discussion, we syntactically distinguish between heap and mutable variables: They are drawn from different alphabets.

Definition 8.3.2 (*Terms and values.*) Terms and values are defined mutually recursively by the following grammars, respectively:

$$\begin{array}{rcl} M, N & ::= & x \\ & \mid & V \\ & \mid & M\ N \\ & \mid & \mathbf{let}\ \vec{x} = \vec{M}\ \mathbf{in}\ N \\ & \mid & \mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \vec{x}_i \to N_i\} \end{array}$$

$$\begin{array}{rl} V ::= & c\ x_1\ x_2\ \ldots\ x_i \\ \mid & \lambda x.\ M \\ \mid & \mathit{return}\ M \mid M \gg\!= N \\ \mid & \mathit{getChar} \mid \mathit{putChar}\ M \\ \mid & \mathit{fixIO}\ M \mid \mathit{update}_z\ M \\ \mid & r \\ \mid & \mathit{newIORef}\ M \\ \mid & \mathit{readIORef}\ M \\ \mid & \mathit{writeIORef}\ M\ N \end{array}$$

The function update_z, associated with the heap variable z, cannot appear in a valid input program, and it is never the result of any program either. It is only used internally, in giving a semantics to *fixIO*. We will explain its role in detail later. All other constructs have the same meaning and type as they do in Haskell [7]. Note that IO actions are values as far as the purely functional world is concerned.

For the purposes of this chapter, we only work with well-typed terms, and ignore the issues of type checking and inference. We assume that the usual Haskell rules apply to determine well typed terms. (Typing of Haskell programs has been discussed in detail in the literature [41, 68].) Notice that *return*, $\gg\!=$, *fixIO*, etc., are polymorphic constants. As usual, **let** expressions provide recursive (and possibly polymorphic) bindings.

A constructor c of arity i is treated as a function $\lambda x_1 \ldots x_i.\ c\ x_1 \ldots x_i$, which becomes a value of its own when fully applied. This case is captured by the first alternative in the definition of values, where c is assumed to have arity i. We model constants as nullary constructors, that is, numbers, characters, etc., are treated as constructors with zero arity. (As a notational hint, we will use the letter k to refer to constants.)

Remark 8.3.3 It is worth noting that the grammar we gave describes the syntax for the reduced terms of our language rather than the concrete syntax that we will allow ourselves to use. In particular, we will freely use the do-notation and pattern bindings in λ-abstractions. In each case, however, the translation to the core language will be trivial.

Definition 8.3.4 (*IO and pure terms.*) A well-typed term of type $IO\ \tau$, for some type τ, is called an IO term. All other terms are called pure.

Definition 8.3.5 (*Terminal values.*) A value is called terminal if it has one of the following forms:

- $c\, x_1\, x_2 \ldots x_i$, where c is a constructor of arity i,
- $\lambda x.M$,
- $\mathit{return}\; M$,

where M is an arbitrary term in the second and third cases.

Definition 8.3.6 (*Heaps.*) A heap is a finite partial function from heap variables to terms extended with a special black hole value $\bullet$:

$$\Gamma\ ::\ \mathit{Heap\ Variables} \rightharpoonup \mathit{Terms} \cup \{\bullet\}$$

A heap binding can be polymorphically typed. A black hole binding, such as $z \mapsto \bullet$, indicates that the variable is known but not directly accessible. Intuitively, $\bullet$ is a detectable bottom.

Notation 8.3.7 Although heaps are functions, we will allow ourselves to use the set notation freely on them: The notation $x \mapsto M \in \Gamma$ simply states that Γ maps x to M. The empty heap is denoted $\{\}$. The notation $(\Gamma, x \mapsto M)$ denotes the heap Γ extended with a new binding $x \mapsto M$. In this case, x cannot be already bound in Γ, but might appear free in M.

Since our language allows input operations, the meaning of a term might depend on the input stream it receives while being run. To accommodate this view, we have to consider terms and input streams together.

Definition 8.3.8 (*Input streams.*) An input stream is a list of characters, not necessarily finite.

Notation 8.3.9 We will use the Haskell list notation to denote input streams. [] (or "") denotes the empty input stream, i.e., the case when the input is exhausted. Otherwise, a stream is of the form $(c : I)$, where c is a character and I is an input stream.

Definition 8.3.10 (*Term and program states.*) A running program is identified by its program state, which consists of an input stream, a heap and a term state:

$$\begin{array}{llcll} (\textit{Terms States}) & P & ::= & M & \text{Current term} \\ & & | & P \mid \langle x \rangle_r & \text{Passive container} \\ & & | & \nu r.P & \text{Restriction} \end{array}$$

We use the notation $I : \Gamma : P$ to denote program states.

A term state is simply the current term under consideration, together with a number of passive containers. A passive container $\langle x \rangle_r$ represents a mutable variable named r, which holds a heap variable x. (We only store heap variables in these containers; the actual contents are stored in the heap.) Restrictions convey the scoping information for mutable variables. Notice that a program state contains enough information to capture a program in execution.

Remark 8.3.11 To reduce clutter, we will generally skip the bits of the program state that are not needed in the discussion, especially when we write our rules. That is, we will use $\Gamma : P$, if the input stream is irrelevant, and similarly $I : P$, when the heap is not needed. There is no chance of confusion, however, because we only use capital Greek letters for heaps and never skip the term state.

Definition 8.3.12 (*The functions bn and fn.*) The function *bn* takes a heap and returns all the variables bound in it, i.e., $bn(\Gamma) = \{x \mid x \mapsto M \in \Gamma\}$. The function *fn* is defined for term states and heaps. Given a term state, *fn* returns the set of free variables in it. A heap variable x is free if it is not in the scope of a λx binding. A mutable variable r is free if it is not in the scope of a νr binding. For a heap Γ, $fn(\Gamma) = \bigcup\{fn(M) \mid x \mapsto M \in \Gamma\} - bn(\Gamma)$. We treat *fn* as a variable-arity function to simplify the notation: $fn(A, B)$ means $fn(A) \cup fn(B)$, and similarly for more arguments.

Definition 8.3.13 (*Slice of a heap.*) The slice of a heap Γ, with respect to a term state P, written Γ/P, is the subset of Γ that is reachable from the free names of P. More precisely, for a given Γ and P, let

$$\begin{array}{rcl} S_0 & = & fn(P) \\ S_{i+1} & = & S_i \cup (\bigcup\{fn(M) \mid x \in S_i \wedge x \mapsto M \in \Gamma\}) \end{array}$$

and let $S = \bigcup_{i \in \mathbb{N}} S_i$. Then,

$$\Gamma/P = \{x \mapsto M \mid x \in S \wedge x \mapsto M \in \Gamma\} \tag{8.1}$$

Definition 8.3.14 (*Closed program states.*) A program state $S : \Gamma : P$ is closed if $\mathit{fn}(\Gamma) = \emptyset$, and $\mathit{fn}(P) \subseteq \mathit{bn}(\Gamma)$. (Note that if the second condition is satisfied, no mutable variable in P can be free.)

Definition 8.3.15 (*Type of a program state.*) Let $S : \Gamma : P$ be a closed program state, and let M be the term associated with P. We say that $S : \Gamma : P$ has type τ, and write $(S : \Gamma : P) :: \tau$, when M has type τ when typed in the heap Γ.

Definition 8.3.16 (*Terminal program state.*) A program state $S : \Gamma : P$ is terminal if the term associated with P is terminal (Definition 8.3.5).

8.4 Semantics

We describe the semantics of our language in layers. The IO layer takes care of input-output and manages mutable variables. The functional layer handles pure computations. A final set of rules regulate the interaction between these two layers.

Given a term, we need to be able to extract the part that is going to be executed next. We use contexts to guide this search:

Definition 8.4.1 (*Execution Contexts.*) Execution contexts are described by the following grammar:

$$\begin{array}{lrcl} (\textit{Execution Contexts}) & \mathbb{E} & ::= & [\cdot] \\ & & \mid & \mathbb{E} \gg\!= M \end{array}$$

An execution context is a term with one hole, where the hole itself is filled with a term. The notation $\mathbb{E}[M]$ denotes the context $\mathbb{E}$ filled with the term M. An empty context is one where there are no $\gg\!=$'s, as captured by the first alternative. Otherwise, the context is non-empty, i.e., it is some IO action followed by others.[4] If the context is empty, the term filling the context might be pure.

8.4.1 IO layer

Figure 8.1 gives the transition rules for the IO layer. A rule is a (possibly labeled) transition from a program state to another. The label '$!c$' indicates that the character c is printed

[4] Other authors use the term *evaluation context* for this concept [23]. We prefer the term execution, since a non-empty context can only be filled by an IO action which is going to be executed next.

$$\begin{array}{rcll} \mathbb{E}[\mathit{putChar}\ c] & \xrightarrow{!c} & \mathbb{E}[\mathit{return}\ ()] & (\mathit{PUTC}) \\ (c : I) : \mathbb{E}[\mathit{getChar}] & \xrightarrow{?c} & I : \mathbb{E}[\mathit{return}\ c] & (\mathit{GETC}) \end{array}$$

$$\mathbb{E}[\mathit{return}\ N \gg= \ M] \longrightarrow \mathbb{E}[M\ N] \qquad (\mathit{LUNIT})$$

$$\frac{r \notin \mathit{fn}(\mathbb{E}[\mathit{newIORef}\ M]) \ \wedge \ x \notin \mathit{bn}(\Gamma)}{\Gamma : \mathbb{E}[\mathit{newIORef}\ M] \longrightarrow (\Gamma, x \mapsto M) : \nu r.(\mathbb{E}[\mathit{return}\ r] \mid \langle x \rangle_r)} \quad (\mathit{NEWIO})$$

$$\mathbb{E}[\mathit{readIORef}\ r] \mid \langle x \rangle_r \longrightarrow \mathbb{E}[\mathit{return}\ x] \mid \langle x \rangle_r \quad (\mathit{READIO})$$

$$\frac{y \notin \mathit{bn}(\Gamma)}{\Gamma : \mathbb{E}[\mathit{writeIORef}\ r\ N] \mid \langle x \rangle_r \longrightarrow (\Gamma, y \mapsto N) : \mathbb{E}[\mathit{return}\ ()] \mid \langle y \rangle_r} \quad (\mathit{WRITEIO})$$

$$\frac{z \notin \mathit{bn}(\Gamma)}{\Gamma : \mathbb{E}[\mathit{fixIO}\ M] \longrightarrow (\Gamma, z \mapsto \bullet) : \mathbb{E}[M\ z \gg= \ \mathit{update}_z]} \quad (\mathit{FIXIO})$$

$$(\Gamma, z \mapsto \bullet) : \mathbb{E}[\mathit{update}_z\ M] \longrightarrow (\Gamma, z \mapsto M) : \mathbb{E}[\mathit{return}\ z] \quad (\mathit{UPDATE})$$

Figure 8.1: Semantics: IO layer

on standard output, and the one labeled '$?c$' indicates that the next character from the input stream (which happens to be c) is consumed.[5]

To simplify the notation, we use a couple of conventions in writing our rules (which are going to be formalized in Section 8.4.4). Rather than a verbal explanation, we will consider several illustrative examples:

Example 8.4.2 Consider the program state

$$\texttt{"ab"} : \Gamma : \mathit{getChar} \gg= \mathit{putChar}$$

for some heap Γ. The term state consists of the single term $\mathit{getChar} \gg= \mathit{putChar}$. When we match this term to the context grammar given in Definition 8.4.1, we see that there are two possibilities. Either we can have the empty context, filled with the term $\mathit{getChar} \gg= \mathit{putChar}$, or the context $[\cdot] \gg= \mathit{putChar}$, filled with the term $\mathit{getChar}$. Upon inspection of our rules, we see that only the second has a chance of matching a rule, namely *GETC*. Since the *GETC* rule requires the input stream to be of the form $(c : I)$,

[5] Note that this is the same convention as we have used for the execution of *fudgets* in Section 4.8.

we have to make sure that we have a non-empty stream. Because "ab" is not empty, the *GETC* rule is applicable. Hence, we end up with the transition:

$$\texttt{"ab"} : \Gamma : \mathit{getChar} \gg\!= \ \mathit{putChar} \ \xrightarrow{?\mathsf{a}} \ \texttt{"b"} : \Gamma : \mathit{return}\ \text{'a'} \gg\!= \ \mathit{putChar}$$

Note that the *GETC* rule does not make use of the heap, hence it is not even mentioned. The heap is simply carried across unchanged.

Example 8.4.3 Consider what happens when we continue the preceding example. Again, there are two possible choices for the context. The empty context, filled with the term *return* 'a' $\gg\!=$ *putChar*, or the context $[\cdot] \gg\!=$ *putChar*, filled with the term *return* 'a'. Unlike the preceding case, however, the first choice matches the *LUNIT* rule, while the second one does not match any. Since the *LUNIT* rule does not constrain the input stream or the heap in any way, it is applicable. Hence, we end up with the transition:

$$\texttt{"b"} : \Gamma : \mathit{return}\ \text{'a'} \gg\!= \ \mathit{putChar} \quad \longrightarrow \quad \texttt{"b"} : \Gamma : \mathit{putChar}\ \text{'a'}$$

Since *PUTC* rule does not make use of the input stream or the heap, it does not explicitly mention them. They are both simply copied. It should now be obvious that the next transition is:

$$\texttt{"b"} : \Gamma : \mathit{putChar}\ \text{'a'} \ \xrightarrow{!\mathsf{a}} \ \texttt{"b"} : \Gamma : \mathit{return}\ ()$$

and there are no more transitions from this state, as none of the rules match.

Example 8.4.4 Consider the program state $I : \Gamma : \mathit{newIORef}\, 5 \gg\!= \mathit{readIORef}$, for some I and Γ. The only matching choice for the context is $[\cdot] \gg\!= \mathit{readIORef}$, with the term *newIORef 5* filling the hole. The *NEWIO* rule applies. To satisfy the precondition of this rule, we have to pick variables r and x such that $r \notin \mathit{fn}(\mathit{newIORef}\, 5 \gg\!= \mathit{readIORef})$ and $x \notin \mathit{bn}(\Gamma)$. We simply pick fresh variables to satisfy these requests. Let us call them r and x for simplicity. We end up with the transition:

$$\begin{array}{l} I : \Gamma : \mathit{newIORef}\ 5 \gg\!= \ \mathit{readIORef} \\ \qquad \longrightarrow \quad I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{return}\ r \gg\!= \ \mathit{readIORef}\ \mid \langle x \rangle_r) \end{array}$$

Example 8.4.5 We will continue with the previous example. Clearly, we want to apply the *LUNIT* rule, but it is not clear how we get over the restriction νr. If we look at the *LUNIT* rule, we see that only a term in context is specified (as in all rules except *READIO* and *WRITEIO*). The convention we adopt in this case is the following: If a rule only mentions a term in a context in the term state position, then we consider the term associated with the current program state and try to match it. Any remaining restrictions, passive containers, etc., are copied along. In this case, we obtain:

$$
\begin{aligned}
& I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{return}\ r \gg\!= \ \mathit{readIORef}\ \mid \langle x\rangle_r) \\
& \qquad \longrightarrow \quad I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{readIORef}\ r\ \mid \langle x\rangle_r)
\end{aligned}
$$

Example 8.4.6 Finally we show how to handle rules that have both a term in context and a passive reference mentioned in their left hand sides, namely the *WRITEIO* and *READIO* rules. Continuing the previous example, we see that the *READIO* rule needs to be applied, which requires a term of the form *readIORef r* next to a passive container named r. In this case, our convention is the following: If a rule mentions a term in context next to a passive container, then a program state matches it if and only if we can show that the term associated with it matches the term in context, and we are next to the corresponding passive container. In our case, we get the following transition:

$$
\begin{aligned}
& I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{readIORef}\ r\ \mid \langle x\rangle_r) \\
& \qquad \longrightarrow \quad I : (\Gamma,\ x \mapsto 5) : \nu r.(\mathit{return}\ 5\ \mid \langle x\rangle_r)
\end{aligned}
$$

Remark 8.4.7 The careful reader must have noticed that it is not necessarily the case that we will always have the required passive container positioned nicely. For example, if we start with the program state

$$[\,]\ :\ \{\}\ :\ \mathit{newIORef}\ 0 \gg\!= \ \lambda r.\ \mathit{newIORef}\ 1 \gg\!= \ \lambda s.\ \mathit{readIORef}\ r$$

we will end up with:

$$[\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.(\mathit{readIORef}\ r\ \mid \langle y\rangle_s) \mid \langle x\rangle_r)$$

Clearly, we want to apply the *READIOREF* rule here as well. Alas, the rule does not match. In these cases, we will need to use structural rules, which provide means for transforming the program state into an equivalent one such that there is an applicable rule. Structural rules are covered in Section 8.4.4.

Some comments about the *FIXIO* rule are in order. The function *fixIO* is modeled after knot tying recursion semantics. We first create a new heap variable, called z, whose value is not yet known. This is achieved by binding it to $\bullet$. Then, we call the function and pass it the argument z, and proceed normally. If the evaluation of this function needs to know the value of z, the derivation will get stuck with a detected black hole. Otherwise, z could be passed around, stored in data structures, etc.: Note that it is just a normal heap variable. Once the function call completes, we update the heap variable z by the result, effectively tying the knot by an application of the *UPDATE* rule. In summary, z holds the value of the entire computation, which might in turn depend lazily on its own value, i.e., it is recursively defined.

Although the rules of our IO layer are quite similar to those given by Peyton Jones [67], the following differences are worth mentioning:

- We keep track of the input stream explicitly, rather than assuming that standard input will be consulted whenever a *getChar* is executed,

- As in the natural semantics of Launchbury [48], we keep track of a separate global heap to store values of variables,

- Unlike Peyton Jones's semantics, our reference cells only store heap variables, rather than arbitrary terms. This restriction is necessary in order to model sharing implied by lazy evaluation.

8.4.2 Functional layer

Our rules for the functional layer, given in Figure 8.2, follow Launchbury's natural semantics for lazy evaluation closely [48]. Note that none of the rules in this layer mention the input stream, as it is irrelevant at this layer. Also, we use the notation $\Downarrow$, rather than $\longrightarrow$, for reductions. Compared to the IO layer, where we have a small step semantics, the rules in the functional layer encode a big step natural semantics.

$$\Gamma : V \ \Downarrow\ \Gamma : V \qquad (\mathit{VALUE})$$

$$\frac{\Gamma : M \ \Downarrow\ \Delta : \lambda y.M' \qquad (\Delta, w \mapsto N) : M'[w/y] \ \Downarrow\ \Theta : V}{\Gamma : MN \ \Downarrow\ \Theta : V} \quad (\mathit{APP})$$

$$\frac{(\Gamma, x \mapsto \bullet) : M \ \Downarrow\ (\Delta, x \mapsto \bullet) : V}{(\Gamma, x \mapsto M) : x \ \Downarrow\ (\Delta, x \mapsto V) : V} \quad (\mathit{VAR})$$

$$\frac{(\Gamma, \hat{x}_1 \mapsto \hat{M}_1 \cdots \ \hat{x}_n \mapsto \hat{M}_n) : \hat{N} \ \Downarrow\ \Delta : V}{\Gamma : \mathbf{let}\ \ x_1 = M_1 \cdots \ x_n = M_n \ \ \mathbf{in}\ \ N \ \Downarrow\ \Delta : V} \quad (\mathit{LET})$$

$$\frac{\Gamma : M \ \Downarrow\ \Delta : c_k\, \vec{x}_k \qquad \Delta : M_k[\vec{x}_k/\vec{y}_k] \ \Downarrow\ \Theta : V}{\Gamma : \mathbf{case}\ M\ \mathbf{of}\ \ \{c_i\, \vec{y}_i \rightarrow M_i\} \ \Downarrow\ \Theta : V} \quad (\mathit{CASE})$$

Figure 8.2: Semantics: Functional layer

Compared to Launchbury's natural semantics [48], some minor differences worth mentioning are:

- We introduce a new black hole binding,
- The *APP* rule is generalized to application of terms to terms, rather than terms to just variables. Correspondingly, we do not need to perform the normalization pass,
- We perform renaming in the *LET* rule, rather than the *VAR* rule.

In the *APP* rule, we require $w \notin bn(\Gamma)$. In the *LET* rule, we rename all bound variables $x_1 \ldots x_n$ to $\hat{x}_1 \ldots \hat{x}_n$ so that there will not be any name clashes in the heap when we do the additions. Similarly, the term $\hat{M}_i$ denotes the term M_i, where each occurrence of x_i is replaced by $\hat{x}_i$. (Similarly for $\hat{N}$.) The *VAR* rule is not applicable if the variable being looked up is bound to $\bullet$ in the heap. If this case ever occurs, the derivation will simply terminate with failure, corresponding to a detectable black hole.

We refrain from going into details of this layer, as such systems are rather well studied in the literature. The interested reader is referred to Launchbury's original exposition [48], and Sestoft's work on abstract machines based on such systems [78].

8.4.3 The marriage

$$\frac{\Gamma : M \Downarrow \Delta : k}{\Gamma : \mathbb{E}[\mathit{putChar}\ \ M] \longrightarrow \Delta : \mathbb{E}[\mathit{putChar}\ \ k]} \quad (\mathit{PUTCEVAL})$$

$$\frac{\Gamma : M \Downarrow \Delta : r}{\Gamma : \mathbb{E}[\mathit{readIORef}\ \ M] \longrightarrow \Delta : \mathbb{E}[\mathit{readIORef}\ \ r]} \quad (\mathit{READIOEVAL})$$

$$\frac{\Gamma : M \Downarrow \Delta : r}{\Gamma : \mathbb{E}[\mathit{writeIORef}\ \ M\ \ N] \longrightarrow \Delta : \mathbb{E}[\mathit{writeIORef}\ \ r\ \ N]} \quad (\mathit{WRITEIOEVAL})$$

$$\frac{\Gamma : M \Downarrow \Delta : V}{\Gamma : \mathbb{E}[M] \longrightarrow \Delta : \mathbb{E}[V]} \quad (\mathit{FUN})$$

Figure 8.3: Semantics: Marriage of layers. All these rules are subject to the side condition that M is not a value.

Given separate semantics for the IO and functional layers, we need to specify exactly how they interact. There are two different kinds of interaction. First, whenever we try to reduce a term of the form, say, *putChar M*, we first need to consult the functional layer to reduce the term M to a character. The IO layer will then perform the output. (Note that the *PUTC* rule of the IO-layer only applies when the argument to *putChar* is a constant.) We need similar rules for *readIORef* and *writeIORef* as well. The first three

$$\frac{s \notin \mathit{fn}(P)}{\Gamma : \nu r.P \;\equiv\; \Gamma[s/r] : \nu s.P[s/r]} \quad (\mathit{ALPHA1})$$

$$\frac{y \notin \mathit{fn}(\Gamma, M, P) \wedge y \notin \mathit{bn}(\Gamma)}{(\Gamma, x \mapsto M) : P \;\equiv\; (\Gamma, y \mapsto M)[x/y] : P[x/y]} \quad (\mathit{ALPHA2})$$

$$\frac{x \notin \mathit{bn}(\Gamma) \;\wedge\; x \notin \mathit{fn}(\Gamma,\, P)}{\Gamma : P \equiv (\Gamma,\; x \mapsto M) : P} \quad (\mathit{HEAPEXT})$$

$$\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & (\mathit{COMM}) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\mathit{ASSOC}) \\
\nu r.\nu s.P & \equiv & \nu s.\nu r.P & (\mathit{SWAP})
\end{array}$$

$$\frac{r \notin \mathit{fn}(Q, \Gamma/Q)}{\Gamma : (\nu r.P) \mid Q \;\equiv\; \Gamma : \nu r.(P \mid Q)} \quad (\mathit{EXTRUDE})$$

Figure 8.4: Semantics: Structural rules, Part I

rules in Figure 8.3 take care of this interaction. The second kind of interaction allows handling of applications, **let** and **case** expressions, and variable lookups. This interaction is provided by embedding the functional world into the IO world, as modeled by the *FUN* rule. In all these rules, M is assumed to be a non-value: The functional layer is consulted to reduce M to a value.

8.4.4 Structural rules

Finally, we need a set of structural rules to shape our derivations. As discussed in Remark 8.4.7, structural rules do not perform evaluation steps as do the other rules, but they might be necessary in order to transform a program state to an equivalent one such that one of the transition rules can apply.

The first set of structural rules, presented in Figure 8.4, state that certain program states are equivalent to others. As usual, we mention input streams and heaps only when they are relevant. The *ALPHA* rules state that heap and mutable variables can be renamed at will, i.e., we do not distinguish program states that differ only in the names of variables. (Substitution on heaps is defined as $\Gamma[x/y] = \{z \mapsto M[x/y] \mid z \mapsto M \in \Gamma\}$.) Note that we do not need a side condition of the form $s \notin \mathit{bn}(\Gamma)$ in *ALPHA1*, since only heap variables can be bound in the heap.

The *HEAPEXT* rule states that we can add new bindings, as long as they do not interfere with existing bindings. See Section 8.6 for an example use of this rule.[6] The rules

[6] We can also add a garbage collection rule to get rid of unreachable heap variables and passive containers. We will avoid such a rule for the sake of brevity, as it is not essential for our current purposes.

$$\frac{P \xrightarrow{\alpha} Q}{\Gamma : P \xrightarrow{\alpha} \Gamma : Q}\ (\mathit{HEAPIN}) \qquad \frac{P \xrightarrow{\alpha} Q}{I : P \xrightarrow{\alpha} I : Q}\ (\mathit{STREAMIN})$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}\ (\mathit{PAR}) \qquad \frac{P \xrightarrow{\alpha} Q}{\nu r.P \xrightarrow{\alpha} \nu r.Q}\ (\mathit{NU})$$

$$\frac{\Gamma : P \equiv \Delta : P' \qquad \Delta : P' \xrightarrow{\alpha} \Theta : Q' \qquad \Theta : Q' \equiv \Sigma : Q}{\Gamma : P \xrightarrow{\alpha} \Sigma : Q}\ (\mathit{EQUIV})$$

Figure 8.5: Semantics: Structural rules, Part II. The label α ranges over empty transitions as well.

COMM, *ASSOC* and *SWAP* state obvious equivalences. Finally *EXTRUDE* shows how we can manipulate the scoping of reference variables. The side condition in the *EXTRUDE* rule guarantees that no dangling references will be created. (See Example 8.5.4 for details.)

The second set of structural rules, presented in Figure 8.5, formalize our conventions in applying the rules. The first four rules simply state that we can concentrate on the relevant bits of the derivation and add the extra bits later on. And finally, *EQUIV* states that we only need to consider program states up to equivalence when performing transitions.

Example 8.4.8 We will reconsider the example discussed in Remark 8.4.7. Recall that we had the program state:

$$[\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.(\mathit{readIORef}\ \ r \ \mid \langle y \rangle_s) \mid \langle x \rangle_r)$$

By applying *EXTRUDE, ASSOC, COMM, ASSOC* and *READIOREF* rules (and by appropriate applications of the rules in Figure 8.5 to enable them), we get:

$$\begin{array}{ll}
\equiv & [\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.((\mathit{readIORef}\ \ r \ \mid \langle y \rangle_s) \mid \langle x \rangle_r)) \\
\equiv & [\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.((\mathit{readIORef}\ \ r \ \mid \langle x \rangle_r) \mid \langle y \rangle_s)) \\
\longrightarrow & [\,] : \{x \mapsto 0,\ y \mapsto 1\} : \nu r.(\nu s.((\mathit{return}\ \ x \ \mid \langle x \rangle_r) \mid \langle y \rangle_s))
\end{array}$$

There are no matching rules for the resulting program state. We can apply structural rules again, but none will give us a program state where a non-structural rule can apply.

Remark 8.4.9 One can extend $\equiv$ to an equivalence relation on program states, simply by adding rules to make it reflexive and transitive. However, the current definition of $\equiv$ given in Figure 8.4 is simply too crude to be useful for this purpose. Intuitively, we want to be able to identify program states if their "observable behavior" are the same [27, 57, 71]. We leave the exploration of this idea for future work.

8.4.5 Meaning of program states

The meaning of a closed program state is its derivation:

Definition 8.4.10 (*Derivations.*) Let $I : \Gamma : P$ be a closed program state. The derivation for $I : \Gamma : P$ is a sequence of labeled transitions, where at each step a rule is applied. Structural rules can be applied at any time, as long as they trigger the application of a non-structural rule. The derivation continues until there are no applicable rules.

Simple inspection of our rules reveals that we have a deterministic system modulo the structural rules. That is, given a program state there is at most one non-structural rule that can apply to it.

Definition 8.4.11 (*Effect of a derivation.*) The effect of a derivation is the concatenation of its transition labels. Empty transitions do not contribute to the effect.

The effect of a program state is simply a (possibly infinite) list, where each element is of the form '$?c$' or '$!c$' for some character c.

Notation 8.4.12 As usual, $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$. We will shorten multiple steps of derivations using the notation $I : \Gamma : P \xrightarrow{\alpha}{}^* I' : \Gamma' : P'$.

Definition 8.4.13 (*Divergent and normal program states.*) A closed program state $I : \Gamma : P$ is called divergent if the derivation starting from $I : \Gamma : P$ either

- continues indefinitely (i.e., we never run out of non-structural rules to apply),
- or, gets stuck in a non-terminal program state (Definition 8.3.16) where no non-structural rule applies.

Otherwise, $I : \Gamma : P$ is called normal.

Example 8.4.14 It is easy to come up with divergent terms. For instance, one can show that the derivation for:

$$I : \Gamma : \mathbf{let}\ \mathit{loop} = \mathit{putChar}\ \text{'a'} \gg \mathit{loop}\ \mathbf{in}\ \mathit{loop} \qquad (8.2)$$

diverges, since we never run out of rules to apply. However, the derivation for:

$$I : \Gamma : \textbf{let } x = x \textbf{ in } x \tag{8.3}$$

will diverge by getting stuck. The *FUN* rule will never fire, because there are no reductions for this term in the functional layer. (Notice that the first application of the *VAR* rule will result in $I : (\Gamma,\ x \mapsto \bullet) : x$, but no other rule will apply since the *VAR* rule is only applicable when the binding is not a black hole.) Similarly, a derivation can get stuck via the use of the *FIXIO* rule (which introduces a black hole binding in the heap). A final possibility is the application of the *GETC* rule when the input stream is empty.

Lemma 8.4.15 (*Derivations for normal program states.*) Let $I : \Gamma : P$ be a normal program state. The derivation starting at this state will take the form

$$I : \Gamma : P \xrightarrow{\alpha}{}^{*} I' : \Delta : Q$$

where I' is a suffix of I. Furthermore, Q can be transformed using only the structural rules to the form $\nu\vec{r}.(N \mid C)$, where N is a terminal value (Definition 8.3.5), and C is a number (possibly zero) of parallel passive containers. The restrictions encoded by $\vec{r}$ cover all passive containers in C.

Proof (Sketch.) By definition 8.4.13, our proof obligation reduces to establishing that Q can be transformed into the required $\nu\vec{r}.(N \mid C)$ form. By inspection of the structural rules, we see that the rule *EXTRUDE* can be repeatedly used to move restrictions to the top, obtaining the required form. (*ALPHA* rules can be used to resolve naming conflicts, if any.) To see the correspondence between restrictions and the passive containers, just notice that they are introduced together by *NEWIO*, they are never removed, and all rules respect the scoping of ν bindings. □

Observation 8.4.16 Note that derivations apply to both pure and IO terms. A derivation either diverges, or ends up with an abstraction or a saturated constructor application for a pure term, or with a term of the form *return* M for an IO term.

Proposition 8.4.17 (*Derivations for IO terms in contexts.*) Let $I : \Gamma : \nu\vec{r}.(\mathbb{E}[M] \mid C)$ be a closed program state, where M is an IO term. The derivation starting at this state will either diverge, or take the form:

$$\begin{aligned} I : \Gamma : \nu\vec{r}.(\mathbb{E}[M] \mid C) &\xrightarrow{\alpha}{}^{*} \quad I' : \Delta : \nu\vec{r'}.(\mathbb{E}[\mathit{return}\ \ N] \mid C') \\ &\xrightarrow{\beta}{}^{*} \quad I'' : \Theta : \nu\vec{r''}.(\mathit{return}\ \ O\ \mid C'') \end{aligned}$$

where I' is a suffix of I, and I'' is a suffix of I'.

Proof By inspection of our rules, we see that if the derivation for $\Gamma : \nu\vec{r}.(\mathbb{E}[M] \mid C)$ terminates, then so must the derivation for $\Gamma : \nu\vec{r}.(M \mid C)$. Hence, by the previous lemma, it must do so in the required intermediate form. The form of the final state is again guaranteed by the previous lemma. □

To be able to talk about strictness (Equation 2.1), we need to say what $\bot$ means for the type $IO\ \tau$:

Definition 8.4.18 (*Silent derivations.*) A derivation is silent if its effect is empty.

Definition 8.4.19 (*Bottoms of IO.*) A closed program state $(I : \Gamma : M) :: IO\ \tau$ is a bottom element ($\bot$) for the type $IO\ \tau$, iff the derivation for $I : \Gamma : M$ silently diverges.

Example 8.4.20 It is easy to see that Program State 8.2 is not a $\bot$ of IO, but Program State 8.3 is. While they both diverge, the former is not silent.

Definition 8.4.21 (*Strict functions.*) Let Γ be a heap and M be a term such that the program state $([\,] : \Gamma : M) :: \tau \to IO\ \sigma$ is closed. M is strict, if, for all I and $\Delta \supseteq \Gamma/M$, $x \notin bn(\Gamma)$, the derivation for

$$I : (\Delta, x \mapsto \bullet) : M\ x$$

is silently divergent.

8.5 Examples

We revisit the examples given in Section 8.2, and show how our semantics can handle them. In these examples, we will use the letters $a, b, \ldots$ to represent heap variables as well. To save space, we will apply the structural rules silently.

Example 8.5.1 We will revisit Example 8.2.1. We first remove the *do* notation in favor of explicit $\gg\!=$'s:

$$fixIO\ (\lambda cs.\ getChar \gg\!= \lambda c.\ return\ (c : cs))$$

To reduce clutter, we will not write the input stream explicitly. We have:

$\{\} : fixIO\ (\lambda cs.\ getChar \gg\!= \lambda c.\ return\ (c : cs))$

$\longrightarrow^*$ (FIXIO - FUN)

$\{z \mapsto \bullet,\ a \mapsto z\} : getChar \gg\!= \lambda c.\ return\ (c : a) \gg\!= update_z$

$\xrightarrow{?ch}$ (GETC – assume input stream has *ch* in front)

$\{z \mapsto \bullet,\ a \mapsto z\} : \mathit{return}\ \mathit{ch}\ \gg\!=\ \lambda c.\ \mathit{return}\ (c\ :\ a)\ \gg\!=\ \mathit{update}_z$

$\longrightarrow^*$ (LUNIT - FUN)

$\{z \mapsto \bullet,\ a \mapsto z, b \mapsto \mathit{ch}\} : \mathit{return}\ (b\ :\ a)\ \gg\!=\ \mathit{update}_z$

$\longrightarrow$ (LUNIT)

$\{z \mapsto \bullet,\ a \mapsto z, b \mapsto \mathit{ch}\} : \mathit{update}_z\ (b{:}a)$

$\longrightarrow$ (UPDATE)

$\{z \mapsto b : a,\ a \mapsto z, b \mapsto \mathit{ch}\} : \mathit{return}\ z$

The derivation terminates with a terminal program state at this point. Hence the initial program state is normal. The final heap contains the cyclic structure that represents the infinite list of *ch*'s: The character that was read by *getChar*. In case elements of this list are demanded in a context, the usual demand-driven rules modeled by our semantics would let us produce enough elements to satisfy the need. If the input stream is empty to start with, the derivation will simply block at the point where the *GETC* rule is applied, and wait forever, i.e., the derivation will diverge by getting stuck.

Example 8.5.2 Showing that Example 8.2.2 diverges is fairly easy. We have:

$\{\}\ :\ \mathit{fixIO}\ (\lambda c.\ \mathit{putChar}\ c\ \gg\!=\ \lambda d.\ \mathit{return}\ \text{'a'})$

$\longrightarrow^*$ (FIXIO - FUN)

$\{z \mapsto \bullet,\ a \mapsto z\} : \mathit{putChar}\ a\ \gg\ \lambda d.\ \mathit{return}\ \text{'a'}$

And now, we need to apply the *PUTCEVAL* rule to reduce the variable a to a character. The functional layer first reduces a to z using the *VAR* rule, but gets stuck at that point, as z is bound to $\bullet$ in the heap and the *VAR* rule does not apply anymore.

Example 8.5.3 We now reconsider Example 8.2.3, which involves reference cells. Again, removing *do*-notation and simplifying the patterns, we get:

$$\begin{aligned}&\mathit{fixIO}\ (\lambda t.\ \mathit{newIORef}\ (\mathit{fst}\ t)\ \gg\!=\ \lambda y.\\ &\qquad\qquad \mathit{return}\ (1\ :\ \mathit{fst}\ t,\ y))\\ &\quad \gg\!=\ \lambda u.\ \mathit{readIORef}\ (\mathit{snd}\ u)\end{aligned}$$

Since there are no calls to *getChar*, the input stream does not matter. That is, we will simply copy the same input stream through all transitions in our derivation. Therefore, we simply do not write it explicitly in what follows.

We will first consider the *fixIO* call. To save space, we will abbreviate *newIORef* to *new* and *readIORef* to *read*:

$$\{\} : \mathit{fixIO}\ (\lambda t.\ \mathit{new}\ (\mathit{fst}\ t) \gg\!=\ \lambda y.\mathit{return}\ (1 : \mathit{fst}\ t,\ y))$$

$\longrightarrow^*$ (FIXIO - FUN)

$$\{z \mapsto \bullet,\ a \mapsto z\} : \mathit{new}\ (\mathit{fst}\ a) \gg\!=\ \lambda y.\ \mathit{return}\ (1 : \mathit{fst}\ a,\ y) \gg\!=\ \mathit{update}_z$$

$\longrightarrow$ (NEWIO)

$$\{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto \mathit{fst}\ a\} :$$
$$\nu r.(\mathit{return}\ r \gg\!=\ \lambda y.\mathit{return}\ (1 : \mathit{fst}\ a,\ y) \gg\!=\ \mathit{update}_z\ \mid \langle b \rangle_r)$$

$\longrightarrow^*$ (LUNIT - FUN)

$$\{z \mapsto \bullet,\ a \mapsto z,\ b \mapsto \mathit{fst}\ a,\ c \mapsto r\} :$$
$$\nu r.(\mathit{return}\ (1 : \mathit{fst}\ a,\ c) \gg\!=\ \mathit{update}_z\ \mid \langle b \rangle_r)$$

$\longrightarrow^*$ (LUNIT - UPDATE)

$$\{z \mapsto (1 : \mathit{fst}\ a, c),\ a \mapsto z,\ b \mapsto \mathit{fst}\ a,\ c \mapsto r\} : \nu r.(\mathit{return}\ z\ \mid \langle b \rangle_r)$$

When we consider the original expression, it is not hard to see that we will have:

$\longrightarrow$ (LUNIT - FUN)

$$\{z \mapsto (1\ :\ \mathit{fst}\ a,\ c)\ ,\ a \mapsto z,\ b \mapsto \mathit{fst}\ a,\ c \mapsto r,\ d \mapsto z\} :$$
$$\nu r.(\mathit{read}\ (\mathit{snd}\ d)\ \mid \langle b \rangle_r)$$

$\longrightarrow$ (READIOEVAL)

$$\{z \mapsto (e,\ f),\ a \mapsto z,\ b \mapsto \mathit{fst}\, a,\ c \mapsto r,\ d \mapsto (e,\ f),\ e \mapsto 1 : \mathit{fst}\ a,\ f \mapsto r\} :$$
$$\nu r.(\mathit{read}\ r\ \mid \langle b \rangle_r)$$

$\longrightarrow$ (READIOREF)

$$\{z \mapsto (e,\ f),\ a \mapsto z,\ b \mapsto \mathit{fst}\, a,\ c \mapsto r,\ d \mapsto (e,\ f),\ e \mapsto 1 : \mathit{fst}\ a,\ f \mapsto r\} :$$
$$\nu r.(\mathit{return}\ b\ \mid \langle b \rangle_r)$$

Now, if we chase the value of b in the heap, we see that we will end up with a cyclic structure effectively representing the infinite lists of 1's, as intended. The most interesting step in this derivation is the application of the *READIOEVAL* rule. The function *snd* is a short hand for **case** over the pairing constructor. The *VAR* rule in the functional layer arranges for sharing, resulting in an abundance of variables in the resulting heap. Notice that, abusing the notation slightly, in the above derivation $(1 : \mathit{fst}\, a,\ c)$ refers to a function application: the pairing constructor applied to the terms $1\ :\ \mathit{fst}$ and c. In the last two lines, however, $(e,\, f)$ is a value, i.e., in this case, the pairing constructor applied to the right number of arguments.

Example 8.5.4 This example demonstrates the importance of the side condition of the *EXTRUDE* rule. Consider:

$$\begin{array}{l} \textbf{do}\ j \leftarrow \mathit{new}\ 5 \\ \quad\ \ k \leftarrow \mathit{new}\ j \\ \quad\ \ l \leftarrow \mathit{read}\ k \\ \quad\ \ \mathit{read}\ l \end{array}$$

By removing the *do*-notation, we get:

$$\mathit{new}\ 5 \gg\!= \ \mathit{new} \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}$$

We will try to give a derivation for this expression, ignoring the side condition of the *EXTRUDE* rule. Again the input stream is irrelevant, and hence ignored:

$\{\}\ :\ \mathit{new}\ 5 \gg\!= \ \mathit{new} \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}$
$\longrightarrow$ (NEWIOREF)
$\{x \mapsto 5\} : \nu j.(\mathit{return}\ j \gg\!= \ \mathit{new} \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}\ \mid \langle x\rangle_j)$
$\longrightarrow^*$ (LUNIT-NEWIOREF)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\nu k.(\mathit{return}\ k \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}\ \mid \langle y\rangle_k) \mid \langle x\rangle_j)$
$\longrightarrow$ (COMM)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x\rangle_j \mid \nu k.(\mathit{return}\ k \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}\ \mid \langle y\rangle_k))$
$\longrightarrow$ (EXTRUDE – *incorrect application*)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x\rangle_j) \mid \nu k.(\mathit{return}\ k \gg\!= \ \mathit{read} \gg\!= \ \mathit{read}\ \mid \langle y\rangle_k)$
$\longrightarrow^*$ (LUNIT - READ - LUNIT)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x\rangle_j) \mid \nu k.(\mathit{read}\ y \mid \langle y\rangle_k)$
$\longrightarrow$ (READIOEVAL)
$\{x \mapsto 5,\ y \mapsto j\} : \nu j.(\langle x\rangle_j) \mid \nu k.(\mathit{read}\ j \mid \langle y\rangle_k)$

And now we are stuck! The mutable variable j is not visible at this point. Since we were not careful in applying the extrude rule, we have created a dangling reference. Let us construct the slice when the rule is applied:

$$S_0 = \{y\},\quad S_1 = \{y,\, j\},\quad S_2 = S_1 = S_\infty$$

By Equation 8.1, the slice is: $\{y \mapsto j\}$. Since $j \in \mathit{fn}(\{y \mapsto j\})$, *EXTRUDE* is not applicable. The side condition prevents the creation of the dangling reference.

8.6 Properties of *fixIO*

Equipped with the semantics we have presented so far, we are now in a position to look at the properties of *fixIO*.

Strictness. Consider Equation 2.1, and let Γ be a heap where f is properly bound. Assuming f is strict (Definition 8.4.21), we will have:

$$I : \Gamma : \mathit{fixIO}\ f \ \longrightarrow I : (\Gamma,\ z \mapsto \bullet) : f\ z \ggg= \ \mathit{update}_z$$

by a single application of the *FIXIO* rule. The current context specifies that the application $f\ z$ should be evaluated. By Definition 8.4.21, the derivation will silently diverge. But then, by Definition 8.4.19, this divergence implies that *fixIO* f is $\perp$.

Example 8.6.1 Using **if** as a shorthand for **case** over the boolean type, consider:

$I : \{\} : \mathit{fixIO}\ (\lambda x.\ \textbf{if}\ x\ ==\ 0\ \textbf{then}\ \mathit{return}\ 1\ \textbf{else}\ \mathit{return}\ 2)$
$\longrightarrow$ (FIXIO - FUN)
$I : \{z \mapsto \bullet,\ a \mapsto z\} : \textbf{if}\ a\ =\ 0\ \textbf{then}\ \mathit{return}\ 1\ \textbf{else}\ \mathit{return}\ 2 \ggg= \ \mathit{update}_z$
... detected black hole ...

In the last step, the *FUN* rule is not applicable because there are no reductions for the current term in the functional layer.

Example 8.6.2 Consider the following non-strict function:

$$\lambda x.\ \mathit{return}\ x :: \mathit{Char} \rightarrow \mathit{IO}\ \mathit{Char}$$

Notice that it returns a computation successfully. Of course, if the result of the fixed-point computation is used, it will still diverge, but for a different reason:

$I : \{\} : \mathit{fixIO}\ (\lambda x.\ \mathit{return}\ x) \ggg= \ \mathit{putChar}$
$\longrightarrow$ (FIXIO - FUN)
$I : \{z \mapsto \bullet,\ a \mapsto z\} : \mathit{return}\ a \ggg= \ \mathit{update}_z \ggg= \ \mathit{putChar}$
$\longrightarrow$ (LUNIT - UPDATE - LUNIT)
$I : \{z \mapsto a,\ a \mapsto z\} : \mathit{putChar}\ z$
... detected black hole ...

The last step diverges, because the *VAR* rule will get stuck trying to reduce z to a character.

Example 8.6.3 Consider the function:

$$\lambda a.\ \mathit{putChar}\ \text{'q'} \gg \mathbf{if}\ a == 1\ \mathbf{then}\ \mathit{return}\ 1\ \mathbf{else}\ \mathit{return}\ 2$$

which is not strict according to our semantics. Here is the derivation for it:

$I : \{\} : \mathit{fixIO}\ (\lambda a.\ \mathit{putChar}\ \text{'q'} \gg \mathbf{if}\ a == 1\ \mathbf{then}\ \mathit{return}\ 1\ \mathbf{else}\ \mathit{return}\ 2)$
$\longrightarrow^{*}$ (FIXIO - FUN)
$I : \{z \mapsto \bullet,\ a \mapsto z\} :$
$\quad \mathit{putChar}\ \text{'q'} \gg \mathbf{if}\ a == 1\ \mathbf{then}\ \mathit{return}\ 1\ \mathbf{else}\ \mathit{return}\ 2 \gg= \ \mathit{update}_z$
$\xrightarrow{!q}$ (PUTC)
$I : \{z \mapsto \bullet,\ a \mapsto z\} : \mathbf{if}\ a == 1\ \mathbf{then}\ \mathit{return}\ 1\ \mathbf{else}\ \mathit{return}\ 2 \gg= \ \mathit{update}_z$
... detected black hole ...

Before getting stuck, we see the character q printed, which is the correct behavior.

Purity. Consider Equation 2.2, where we will use a **let** expression to capture *fix*:

$$\mathit{fixIO}\ (\mathit{return}\ \cdot\ h) = \mathit{return}\ (\mathbf{let}\ a = h\ a\ \mathbf{in}\ a)$$

Assume Γ is a heap such that $([\,] : \Gamma : h) :: \tau \to \tau$. On the left hand side, we have:

$I : \Gamma : \mathit{fixIO}\ (\mathit{return}\ \cdot\ h)$
$\longrightarrow^{*}$ (FIXIO - FUN)
$I : (\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : (\mathit{return}\ \cdot\ h)\ a \gg= \ \mathit{update}_z$
$\longrightarrow$ (LUNIT)
$I : (\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : \mathit{update}_z\ (h\ a)$
$\longrightarrow$ (UPDATE)
$I : (\Gamma,\ z \mapsto h\ a,\ a \mapsto z) : \mathit{return}\ z$

Considering the right-hand-side, we immediately get: $I : \Gamma : \mathit{return}\ (\mathbf{let}\ a = h\ a\ \mathbf{in}\ a)$.

We should now prove that these two program states are equivalent, i.e., that the rules in our system cannot tell them apart. Such an argument would require a notion of program state equivalence that is more general than what our structural rules provide. Intuitively, the program states above will be considered equivalent if we can show that

$$I : (\Gamma,\ z \mapsto h\ a,\ a \mapsto z) : z \quad \equiv \quad I : \Gamma : \mathbf{let}\ a = h\ a\ \mathbf{in}\ a$$

Note that the second program state reduces to $I : (\Gamma,\ z \mapsto h\ z) : z$. Hence, the equivalence is clear provided we adopt a compaction rule that gets rid of the indirection via a in the first heap. To formalize this argument, we need a precise definition of program state equivalence and a proof system for showing when two program states are the same. We leave the development of a such a system for future work.

Left shrinking. Consider Equation 2.3, where we will refer to the computation a as q to avoid confusion with heap variables. For the left hand side we get:

$$I : \Gamma : \mathit{fixIO}\ (\lambda x.\ q \gg= \ \lambda y.\ f\ \ x\ \ y)$$
$$\longrightarrow^* \text{(FIXIO - FUN)}$$
$$I : (\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : q \gg= \ \lambda y.\ f\ \ a\ \ y \gg= \ \mathit{update}_z$$

On the right hand side, we have $I : \Gamma : q \gg= \ \lambda y.\ \mathit{fixIO}\ (\lambda x.\ f\ \ x\ \ y)$. Now, if the derivation for q diverges, both derivations will diverge in the exact same way, that is both sides are equivalent. Otherwise, by Lemma 8.4.15, we will have:

$$I : \Gamma : q \xrightarrow{\alpha} I' : \Delta : \nu\vec{r}.(\mathit{return}\ \ qv\ \ \mid C)$$

The C on the right hand side captures the passive containers that might be introduced in the derivation for q, along with the associated restrictions $\nu\vec{r}$. Since these containers will get copied in exactly the same way, we do not show them explicitly in the following discussion. Using the *HEAPEXT* and *EXTRUDE* rules silently, the left hand side yields:

$$I : (\Gamma,\ z \mapsto \bullet,\ a \mapsto z) : q \gg= \ \lambda y.\ f\ \ a\ \ y \gg= \ \mathit{update}_z$$
$$\xrightarrow{\alpha}^* \text{(ASSUMPTION)}$$
$$I' : (\Delta,\ z \mapsto \bullet,\ a \mapsto z) : \mathit{return}\ \ qv \gg= \ \lambda y.\ f\ \ a\ \ y \gg= \ \mathit{update}_z$$
$$\longrightarrow^* \text{(LUNIT, FUN)}$$
$$I' : (\Delta,\ z \mapsto \bullet,\ a \mapsto z,\ b \mapsto qv) : f\ \ a\ \ b \gg= \ \mathit{update}_z$$

Let us look at the right hand side:

$$I : \Gamma : q \gg= \ \lambda y.\ \mathit{fixIO}\ (\lambda x.\ f\ \ x\ \ y)$$
$$\xrightarrow{\alpha}^* \text{(ASSUMPTION - LUNIT)}$$
$$I' : (\Delta,\ b \mapsto qv) : \mathit{fixIO}\ (\lambda x.\ f\ \ x\ \ b)$$
$$\longrightarrow^* \text{(FIXIO - FUN)}$$
$$I' : (\Delta,\ b \mapsto qv,\ z \mapsto \bullet,\ a \mapsto z) : f\ \ a\ \ b \gg= \ \mathit{update}_z$$

Hence, the left shrinking property holds for *fixIO*. We conclude that, with respect to our semantics, *fixIO* is a legitimate value recursion operator for the IO monad.

Other properties. As pointed out in Corollary 3.1.7, neither strong sliding nor right shrinking properties hold for Haskell's IO monad. Both of them fail with respect to the semantics we have given in this chapter as well. (Application of our rules to functions used in Propositions 3.1.5 and 3.1.6 suffices to show the failure in both cases.) We believe that sliding and nesting properties should hold, both for Haskell's IO monad and our semantics. We leave the construction of proofs for these properties for future work.

8.7 Summary

In this chapter, we have described an operational semantics for a non-strict functional language extended with monadic IO, references, and value recursion, improving on our earlier work [20]. Our contributions are: (i) we show how a purely functional language and its semantics can be embedded into a language with monadic I/O primitives and references, (ii) we model sharing explicitly at all levels, giving an account of call by need in both the functional and the IO layers, and (iii) we provide a semantics for *fixIO* and show that it is a value recursion operator.

Our work can be extended in several ways. Addition of threads and synchronized variables seems to be fairly easy [67]. The difficulty, however, lies in adding support for asynchronous exceptions [56]. Although exceptions can be modeled nicely in the IO layer, we currently do not see a complementary way of capturing them in the functional layer using our method.

More work is needed in formalizing our arguments. Of the highest importance is the development of a notion of program equivalence, and tools for reasoning about program states which may contain symbolic terms. In this direction, program equivalence based on observational behavior seems to be the right framework [27, 61].

One important issue we have side-stepped in this chapter is that of parametricity. How do we know that the constants of our language (i.e., *return*, $\gg\!=$, *fixIO*, *newIORef*, etc.) are parametric? To talk about parametricity, we first need to define what it means for two program states to be related. Our earlier attempts at stating and establishing parametricity failed, mainly due to the lack of an appropriate notion of program equivalence. Pitts's work on observational equivalence and parametric polymorphism [71] can be used as a basis for such a work, although it is not immediately clear how to accommodate for references and input/output operations. Similarly, Launchbury and Peyton Jones discuss parametricity of constants for manipulating references in the context of the state monad of Haskell [52], but their results are not directly applicable in our framework due to differences in the notion of reference variables, handling of the heap, and the additional complexity introduced by input/output.

Chapter 9

Examples

In this chapter, we will consider a number of practical programming examples, illustrating the use of value recursion operators and the mdo-notation.[1]

Synopsis. Starting with the famous *repmin* problem, we consider applications in sorting networks, screen layout in GUI's, interpreters, cyclic graphs, and the implementation of logical variables.

9.1 The *repmin* problem

The *repmin* problem is concerned with the replacement of all the numbers in a binary tree by their minimum. The challenge is to do so in a single pass [6, 16]. In 1984, Richard Bird devised a beautiful solution to this problem, exploiting laziness and cyclic definitions:

```
data Tree α = L α | B (Tree α) (Tree α) deriving Show

copy             :: Tree Int → Int → (Tree Int, Int)
copy (L a)     m = (L m, a)
copy (B l r) m = let (l', ml) = copy l m
                     (r', mr) = copy r m
                 in (B l' r', ml `min` mr)

repmin   :: Tree Int → Tree Int
repmin t = let (t', m) = copy t m in t'
```

Here's an example run:

```
Main> repmin (B (L 11) (B (L 2) (L 3)))
B (L 2) (B (L 2) (L 2))
```

[1] Before proceeding with the examples in this chapter, the reader may want to review our motivating circuit modeling example, covered in Sections 1.2 and 7.1.

The single pass solution is achieved by the clever use of recursion in the let-expression of the function *repmin*. By virtue of the recursive binding, the function *copy* simultaneously computes and replaces all the leaves with m, the minimum value in the tree.

Benton and Hyland take the problem one step further [5]. What if we also want to perform an effect, such as printing the values stored in the nodes during this single traversal as well? It is easy to modify *copy* to achieve this effect:

$$\begin{aligned}
&\mathit{copyPrint} && :: \mathit{Tree\ Int} \rightarrow \mathit{Int} \rightarrow \mathit{IO}\ (\mathit{Tree\ Int},\ \mathit{Int})\\
&\mathit{copyPrint}\ (L\ a)\quad m && = \mathbf{do}\ \mathit{print}\ a\\
& && \phantom{= \mathbf{do}\ }\mathit{return}\ (L\ m,\ a)\\
&\mathit{copyPrint}\ (B\ l\ r)\ m && = \mathbf{do}\ (l',\ \mathit{ml}) \leftarrow \mathit{copyPrint}\ l\ m\\
& && \phantom{= \mathbf{do}\ }(r',\ \mathit{mr}) \leftarrow \mathit{copyPrint}\ r\ m\\
& && \phantom{= \mathbf{do}\ }\mathit{return}\ (B\ l'\ r',\ \mathit{ml}\ \text{`}\mathit{min}\text{`}\ \mathit{mr})
\end{aligned}$$

But, it is not clear at all how to modify *repmin* accordingly. Obviously, the attempt:

$$\mathit{copyPrint}\ t\ m \gg\!= \ \lambda(t',\ m).\ \mathit{return}\ t'$$

is flawed, since m is no longer recursively bound! We need to tie the recursive knot with an appropriate value recursion operator. In this particular case, the appropriate operator is the one for the IO monad, i.e., *fixIO* of Chapter 8:

$$\begin{aligned}
&\mathit{repminPrint} && :: \mathit{Tree\ Int} \rightarrow \mathit{IO}\ (\mathit{Tree\ Int})\\
&\mathit{repminPrint}\ t && = \mathit{fixIO}\ (\lambda^{\sim}(t',\ m).\ \mathit{copyPrint}\ t\ m)\\
& && \phantom{= \mathit{fixIO}\ } \gg\!= \ \lambda(t',\ m).\ \mathit{return}\ t'
\end{aligned}$$

Or, using the mdo-notation:

$$\begin{aligned}
&\mathit{repminPrint} && :: \mathit{Tree\ Int} \rightarrow \mathit{IO}\ (\mathit{Tree\ Int})\\
&\mathit{repminPrint}\ t && = \mathbf{mdo}\ (t',\ m) \leftarrow \mathit{copyPrint}\ t\ m\\
& && \phantom{= \mathbf{mdo}\ }\mathit{return}\ t'
\end{aligned}$$

hiding the explicit call to *fixIO*, considerably improving readability.

Note that we can accommodate arbitrary effects during the traversal of the original tree, as long as the underlying monad comes equipped with a value recursion operator. To illustrate, consider the following variation of the *repmin* problem, demonstrating the use of value recursion for the *list* monad (Section 4.3). Consider the data type:

$$\mathbf{data}\ \mathit{Exp} = C\ \mathit{Int} \mid A\ \mathit{Exp}\ \mathit{Exp}$$

representing simple arithmetic expressions formed out of integer constants and additions. The problem is to find all possible pair-swaps of a given expression. A swapping is defined to be the exchange of any two constants, not necessarily distinct.[2] Solving the swappings problem is not a terribly hard task. Here, we present a particularly neat solution, illustrating the use of value recursion for the *list* monad:

```
replace           :: Int → Exp → [(Exp,Int)]
replace x (C y)   = [(C x, y)]
replace x (A l r) = [(A l′ r, y) | (l′, y) ← replace x l]
                    ++ [(A l r′, y) | (r′, y) ← replace x r]

pairSwaps   :: Exp → [Exp]
pairSwaps e = mdo (e′, m) ← replace n e
                  (e″, n) ← replace m e′
                  return e″
```

The call *replace* x e creates copies of e, where each copy has one of its constants replaced by x. Each replaced constant is returned along with the corresponding copy. (If there are n constants in e, the call to *replace* will return n copies.) For instance:

```
replace 1 2         ⟹  [(1, 2)]
replace 1 (2 + 3)   ⟹  [(1 + 3, 2), (2 + 1, 3)]
```

The function *pairSwaps* makes two successive calls to *replace*, threading the input expression through. The first call replaces each constant with n (yet to be computed), determining the respective values for m. The second call completes the swapping by substituting m's, and by computing the values of n needed in the first call. Each pairing of m and n corresponds to a possible swapping. The cyclic dependence between m and n achieves the required swapping quite neatly.

Here is an example run for the input $(1+2)+3$, using appropriate functions for parsing and printing:

```
Main> display (pairSwaps (parse "(1 + 2) + 3"))
[(1 + 2) + 3, (2 + 1) + 3, (3 + 2) + 1,
 (2 + 1) + 3, (1 + 2) + 3, (1 + 3) + 2,
 (3 + 2) + 1, (1 + 3) + 2, (1 + 2) + 3]
```

The value recursion operator used implicitly in the definition of *pairSwaps* is the one given by Equation 4.4. Recall that we have considered an infinite family candidate operators for the *list* monad in Section 4.3 (see Equation 4.13). We have argued that these

[2] For instance, the only possible swapping of 1 is 1, while that of $1+2$ are $1+2$, $2+1$, $2+1$, and $1+2$. The two $2+1$'s are considered different, corresponding to the swappings of 1–2 and 2–1. It is easy to see that an expression with n constants will have n^2 swappings, one for each pair of constants.

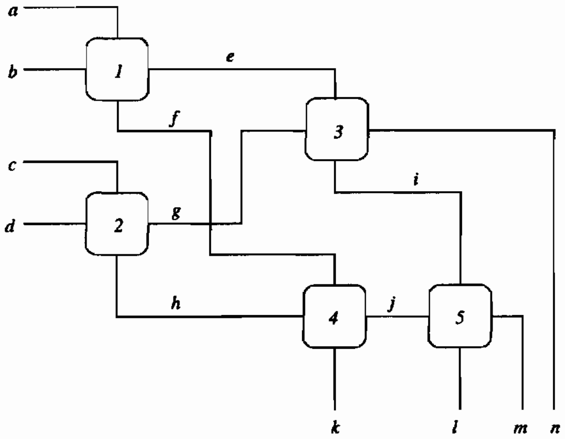
candidates behave strangely, violating the mandatory left shrinking property. We take this opportunity to show that they yield weird results for the swapping problem as well. For instance, the use of mfix_1 yields:

```
Main> display (pairSwaps1 (parse "(1 + 2) + 3"))
[(1 + 2) + 3, (2 + 1) + 3, (2 + 2) + 1,
 (2 + 2) + 3, (1 + 2) + 3, (1 + 2) + 2,
 (3 + 2) + 2, (1 + 3) + 2, (1 + 2) + 3]
```

producing illegal swappings such as $(2+2)+1$. The failure of the left-shrinking property causes unwanted interference when the constants are paired.[3]

9.2 Sorting networks and screen layout in GUI's

A sorting network is a collection of comparators, connected in such a way that the output of the network is always the sorted permutation of its input [15]. For instance, the following network can sort four numbers:



For each comparator, the wire to its right carries the maximum of its inputs, while the lower one carries the minimum. In this particular example, a, b, c, and d are the inputs, while k, l, m, and n are the outputs.

How can we implement a sorting network so that we not only get the values sorted, but also a transcript of the operations performed during sorting? We want each comparator unit to report on the operation it performed while sorting took place. The *output* monad

[3] In certain cases, the operation of the value recursion operator for the *list* monad can be understood in terms of the usual translation rules for list-comprehensions [89], using symbolic substitution for variables that occur recursively [18, Sections 1 and 6.3]. The details, although not terribly important, might be enjoyable for the curious reader, providing some more insight about the behavior of *mfix* for the *list* monad.

(Section 4.5) springs to mind. We can translate the sorting network above almost literally into the following Haskell code:

```
newtype Out α = Out (α, String)

instance Monad Out where
  return x           = Out (x, “”)
  Out ~(x, s) ≫= f  = let Out (y, s′) = f x in Out (y, s ++ s′)

instance Show α ⇒ Show (Out α) where
  show (Out (v, s))  = show v ++ s

comp          :: Int → (Int, Int) → Out (Int, Int)
comp i (a, b) = Out ((a ‘max‘ b, a ‘min‘ b), “\nUnit ” ++ show i ++ msg)
    where msg = (if a < b then “: swap: ” else “: pass: ”) ++ show (a, b)

sort4               :: (Int, Int, Int, Int) → Out (Int, Int, Int, Int)
sort4 (a, b, c, d) = do (e, f) ← comp 1 (a, b)  -- unit 1
                        (g, h) ← comp 2 (c, d) -- unit 2
                        (n, i) ← comp 3 (e, g) -- unit 3
                        (j, k) ← comp 4 (f, h) -- unit 4
                        (m, l) ← comp 5 (i, j) -- unit 5
                        return (k, l, m, n)
```

Here is a sample run:

```
Main> sort4 (23, 12, -1, 2)
(-1,2,12,23)
Unit 1: pass: (23,12)
Unit 2: swap: (-1,2)
Unit 3: pass: (23,2)
Unit 4: pass: (12,-1)
Unit 5: swap: (2,12)
```

What happens if we want to observe the output in some different order? For instance, we might want to see the output of the third unit after the fifth. Intuitively, it must be sufficient to move the third line after the fifth in the definition of *sort4*, obtaining:

```
sort4 (a, b, c, d) =  do (e, f) ← comp 1 (a, b) -- unit 1
                         (g, h) ← comp 2 (c, d) -- unit 2
                         (j, k) ← comp 4 (f, h) -- unit 4
                         (m, l) ← comp 5 (i, j) -- unit 5
                         (n, i) ← comp 3 (e, g) -- unit 3
                         return (k, l, m, n)
```

Alas, this modification is illegal: The variable *i* is unbound when used in the fourth line. Luckily, value recursion fits the bill. All we need to say is that the variable *i* used by the 5th unit is the one that is defined by the 3rd, which can be handled by an mdo-expression. As we have seen in Section 4.5, the corresponding *mfix* is given by:

```
instance MonadFix Out where
    mfix f = let Out (a, s) = f a in Out (a, s)
```

With this declaration and the use of the keyword **mdo**, *sort4* will work as expected, delivering the output of the third unit after that of the fifth.

A similar phenomenon occurs in GUI based programming, where the order of monadic actions implicitly determines the screen layout. To illustrate, consider the following simple example, taken from Thiemann's work on a CGI library for Haskell [85]:

```
do f1 ← inputField (fieldSize 10)
   f2 ← inputField (fieldSize 10)
   submitButton (someAction f1 f2)
```

The corresponding GUI will have two input fields side by side, followed by a submit button. What happens if we want to place the submit button to the left of the input fields? Since the ordering of the statements in the do-expression determines the position of the GUI elements, we would like to move the call to *submitButton* to the first line, textually preceding the calls to *inputField*. As Thiemann also points out, such a move would require the use of an mdo-expression, since the variables *f1* and *f2* will no longer be visible when used as arguments to *someAction*.

9.3 Interpreters

Suppose you are designing an interpreter for a language that has let-bindings for introducing local bindings. Operationally, the expression **let** *v* = *e* **in** *b* denotes the same expression as *b*, where *e* is substituted for all free occurrences of the variable *v*. The abstract syntax of your language might include:

```
data Exp = ... | Let Var Exp Exp
```

Assuming the language is applicative, the natural choice for implementation would be the environment monad (Section 4.6). In this setting, the section of the interpreter that handles the let-expressions might look like:

```
eval (Let v e b) = do ev ← eval e
                      inExtendedEnv (v, ev) (eval b)
```

where *inExtendedEnv* simply extends the environment with the binding $v \mapsto ev$ before passing it on. This approach yields a satisfactory implementation.

Note that, our *eval* function cannot deal with recursive bindings, i.e., in the expression *Let v e b*, *v* is not visible in *e*. What happens if we lift this restriction? All we need is a way to extend the environment with the binding $v \mapsto ev$ in the call to *eval e*, before we actually know what *ev* is. The following mdo-expression expresses the required dependency:

$$\begin{aligned} eval\ (Let\ v\ e\ b) = \mathbf{mdo}\ & ev \leftarrow inExtendedEnv\ (v,\ ev)\ (eval\ e) \\ & inExtendedEnv\ (v,\ ev)\ (eval\ b) \end{aligned}$$

In contrast, consider how we might solve this problem *without* using value recursion. Assuming *Val* denotes the data type for the values our language can process, and the following declaration of environments:

$$\mathbf{data}\ Env\ \alpha = Env\ ([(Var,\ Val)] \rightarrow \alpha)$$

we are forced to implement recursive let-expressions as follows:

$$\begin{aligned} eval\ (Let\ v\ e\ b) = Env\ (\lambda env.\ \mathbf{let}\ & Env\ f = eval\ e \\ & ev \quad\ \ = f\ ((v,\ ev):env) \\ & Env\ g = eval\ b \\ \mathbf{in}\ & g\ ((v,\ ev):env)) \end{aligned}$$

Although it will perform the required task, this solution is hardly satisfactory. First of all, we had to reveal how environments are actually implemented, defeating the whole point of the monadic abstraction. As a result, our code will only work with that particular implementation; switching to a different representation will require changes in the interpreter. The code is no longer easy to understand or maintain.

On the other hand, our first implementation using the mdo-notation is quite simple to understand, concise, and not tied to any particular representation of environments.

9.4 Doubly linked circular lists with mutable nodes

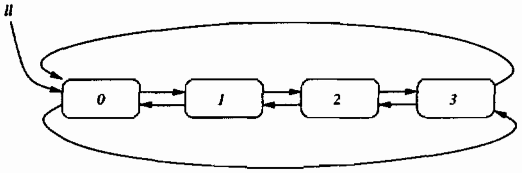
Consider a simple implementation of doubly linked circular lists in Haskell. For this example, we will store a mutable boolean flag at each node, a *True* value indicating that the node is already visited in a particular traversal. We use the internal state monad to gain access to mutable variables [52]. The nodes in our circular lists have the following structure:

$$\mathbf{newtype}\ Node\ s\ \alpha = N\ (STRef\ s\ Bool,\ Node\ s\ \alpha,\ \alpha,\ Node\ s\ \alpha)$$

consisting of the mutable flag, the pointer to the previous node, the data item, and the pointer to the next node. Given two nodes *b* and *f*, a new node in between is created by the following function:

```
newNode       :: Node s α → α → Node s α → ST s (Node s α)
newNode b c f = do v ← newSTRef False
                   return (N (v, b, c, f))
```

Here is a simple example of a circular list, and its rendering in Haskell using the function *newNode*. Note that the use of the mdo-expression is essential in expressing the cyclic structure:[4]



```
ll :: ST s (Node s Int)
ll = mdo n0 ← newNode n3 0 n1
         n1 ← newNode n0 1 n2
         n2 ← newNode n1 2 n3
         n3 ← newNode n2 3 n0
         return n0
```

Traversing a given doubly linked list simply amounts to following the links until we reach a node that has been visited before:

```
data Direction = Forward | Backward deriving Eq

traverse                  :: Direction → Node s α → ST s [α]
traverse dir (N (v, b, i, f)) =
      do visited ← readSTRef v
         if visited
            then return []
            else do writeSTRef v True
                    let n = if dir == Forward then f else b
                    is ← traverse dir n
                    return (i:is)
```

Here's a sample run:

[4] A more traditional technique would rely on creating dummy initial link values for at least one of the nodes, and explicitly overwriting them when the rest of the structure is created. This "clunky" approach is often seen in the formation of cyclic objects in imperative languages, such as Java. Perhaps an mdo-like construct could help there also.

```
Main> runST (ll >>= traverse Forward)
[0,1,2,3]
Main> runST (ll >>= traverse Backward)
[0,3,2,1]
```

The inverse function that takes a non-empty list and constructs a doubly linked circular list out of its elements further illustrates the use of value recursion:

```
encircle        :: [α] → ST s (Node s α)
encircle (x:xs) = mdo c ← newNode l x f
                      (f, l) ← encircle' c xs
                      return c

encircle'          :: Node s α → [α] → ST s (Node s α, Node s α)
encircle' p []     = return (p, p)
encircle' p (x:xs) = mdo c ← newNode p x f
                         (f, l) ← encircle' c xs
                         return (c, l)
```

We have:

```
Main> runST (encircle "hello world" >>= traverse Backward)
"hdlrow olle"
Main> runST (encircle "hello world" >>= traverse Forward)
"hello world"
```

Similar techniques might be useful in the functional implementation of graph algorithms as well [45]. In general, programs manipulating stateful objects with cyclic dependencies can benefit from value recursion. For instance, Nordlander shows how to use value recursion to express layered networking protocols in the context of his O'Haskell language [65, Section 4.2].

9.5 Logical variables

In a tutorial paper on monads and effects, Benton, Hughes and Moggi suggest the following exercise on programming with monads [4, Exercise 55]:

> Prolog provides so-called *logical variables*, whose values can be referred to before they are set. Define a type *LVar* and a monad *Logic* in terms of *ST*, supporting operations:

```
newLVar   :: Logic s (LVar s α)
readLVar  :: LVar s α → α
writeLVar :: LVar s α → α → Logic s ()
```

> where s is again a state-thread identifier. The intention is that an *LVar* should be written exactly once, but its value maybe read *beforehand*, between its creation and the write—lazy evaluation is at work here. Note that *readLVar* does not have a monadic type, and so can be used anywhere.

Clearly, we will need to use value recursion in implementing *newLVar*, allowing us to access the value of a logical variable before it is actually set. There is a small problem, however. How do we determine the *scope* of a logical variable, i.e., how do we make it available to the rest of the computation? We solve this problem by using the continuation monad transformer, a clever trick suggested to us by John Hughes.[5] Using this idea, the *Logic* monad looks like:

```
data Logic s α = Logic {unL :: forall τ. (α → ST s τ) → ST s τ}

instance Monad (Logic s) where
  return a        = Logic (λk. k a)
  Logic f ≫= g   = Logic (λk. f (λa. unL (g a) k))
```

A logical variable is nothing but a value and a pointer to it. To read, we simply project the value. To write, we update the mutable cell:

```
newtype LVar s α = LVar (STRef s α, α)

readLVar             :: LVar s α → α
readLVar (LVar (_, v)) = v

writeLVar                :: LVar s α → α  → Logic s ()
writeLVar (LVar (r, _)) a = Logic (λk. do writeSTRef r a
                                          k ())
```

The magic that makes logical variables work is hidden in *newLVar*:

```
newLVar :: Logic s (LVar s α)
newLVar = Logic (λk. mdo r ← newSTRef (error "unbound LVar!")
                         a ← k (LVar (r, v))
                         v ← readSTRef r
                         return a)
```

[5] An alternative would be to use the type *newLVar* :: (*LVar s a* → *Logic s b*) → *Logic s b*, requiring the user to explicitly specify the scope, as in: *newLVar* (λ*v*. ...). In that case, the *ST* monad itself would serve as the *Logic* monad, without any need for continuations.

Here is how *newLVar* works. We allocate a new mutable variable, r, and form the pair $(r,\ v)$, where v is the value that will eventually be stored in r. This pair is passed to the continuation k, representing the remainder of the computation, i.e., the scope of the new logical variable. Before returning the result of this call, we simply read the mutable variable r, determining the actual value of v. (Note that the computation represented by k is expected to call *writeLVar* on the newly created logical variable, setting its final value.) The mdo-expression implicitly uses the function *fixST*, the value recursion operator for Haskell's internal state monad (see Section 4.4). The final bit of machinery we need is a simple run method to extract values:

```
runLogic   :: (forall s. Logic s τ) → τ
runLogic f = runST (unL f return)
```

Here are a some simple examples demonstrating the use of *LVar*'s:

```
t1 = do v ← newLVar
        let val = readLVar v
        return val
```

```
t2 = do v ← newLVar
        let val = [0, 6 .. readLVar v]
        writeLVar v 42
        return val
```

```
t3 = do v ← newLVar
        let v1 = readLVar v
        writeLVar v 43
        let v2 = readLVar v
        writeLVar v 42
        let v3 = readLVar v
        return (v1, v2, v3)
```

```
t4 = do s ← newLVar
        c ← newLVar
        let sVal = readLVar s
            cVal = readLVar c
        writeLVar s "test"
        writeLVar c '1'
        return (cVal : sVal)
```

We have:

```
Main> runLogic t1 :: Int
Program error: unbound LVar!
Main> runLogic t3
(42,42,42)
```

```
Main> runLogic t2
[0,6,12,18,24,30,36,42]
Main> runLogic t4
"1test"
```

In *t1*, we never write to v, hence its value is left undefined. All calls to *writeLVar* except the last will be ignored,[6] as demonstrated by *t3*. Finally, *t4* shows that we can use variables with different types in the same computation.

Claessen and Ljunglöf show how one can use logical variables to embed a typed functional logic programming language in Haskell [13]. Similar to our implementation, they

[6] Of course, every call to *writeLVar* will be performed when the computation is run, in the given order. However, all calls to *readLVar* will return the last value written, regardless of their order. For all practical purposes, logical variables behave as constants, whose values can be used before they are set.

use the *ST* monad to get access to typed mutable variables. However, they only allow access to logical variables inside their version of the *Logic* monad. (In our terms, their *readLVar* function has the type $LVar\ s\ \alpha \rightarrow Logic\ s\ \alpha$.) It might be interesting to combine their work with ours, allowing logical variables to be used anywhere, and hence providing a more flexible embedding. We leave the exploration of this idea for future work.

9.6 Summary

In this chapter, we illustrated the use of value recursion operators and of the recursive do-notation. We admit that some of our examples might seem a little contrived, with the notable exception of the circuit modeling example of Section 1.2. However, it is our hope that readers will be able to relate these examples to their own work, spotting further applications for value recursion. Here are some common cases to watch for:

- Programs dealing with data flow equations. Assuming the underlying model is monadic, any feedback loop or a cyclic dependency would signal the need for a value recursion operator to tie the recursive knot. Our circuit modeling example is an instance of this problem.

- Stateful objects with mutual dependencies. Again, if a monadic interface is used, mutual dependencies will require the use of value recursion. Our implementation of doubly linked circular lists, and the network programming example in O'Haskell (see Section 10.1 for a brief discussion) are examples of this kind.

- Programs that combine several phases and use recursion to eliminate multiple traversals of data structures, similar to the *repmin* problem of Section 9.1. If any one of the eliminated phases require monadic effects, value recursion becomes the tool for expressing the required cyclic dependence.

- Monadic programs where a particular ordering of effects forces us to use variables that will only become available later, similar to the sorting networks or GUI design examples of Section 9.2.

Chapter 10

Epilogue

In this thesis, we have studied the interaction between two fundamental notions in programming languages: Recursion and effects. As we have seen, cyclic definitions in the presence of monadic effects can be understood in terms of value recursion operators, whose behavior can be characterized by means of a number of equational properties. It is our belief that these properties capture the essence of the interaction satisfactorily. Of course, the extent to which our axiomatization is successful will only be determined by practice. Our properties could be deemed appropriate if they rule out useless definitions of value recursion operators, and admit only those that are meaningful in practical programs. It is still too early to come to a decisive conclusion in this regard, but we hope that our work will be useful for both researchers and practitioners, especially as monads become more and more pervasive in functional programming.

We conclude our exposition of value recursion by briefly reviewing the related work, and pointing out some future research opportunities.

10.1 Related work

The interaction between recursion and shared computations has been extensively studied by Hasegawa [32, 33]. Sharing is a commutative effect, i.e., the order of computations does not matter.[1] As we have explored in the first part of Chapter 6, recursion in commutative monads can be understood in terms of traces in symmetric monoidal categories. Hasegawa shows that giving a trace over a cartesian closed category is the same as giving a fixed-point operator for it (see Theorem 6.2.4). This result is remarkable, as it provides an escape from the usual domain-theoretic view, increasing the level of abstraction considerably. As Hasegawa himself points out, however, when the underlying effect is non-commutative, we can no longer stay in the monoidal world.

[1] Think of a recursive let-expression in Haskell. The order of bindings is irrelevant; equations can be swapped around without changing the result.

Paterson introduced *loop* operators for handling value recursion in *arrows* [36, 66]. Although Paterson notes that some of the axioms are too strong for many practical cases, he does not weaken his axiomatization to accommodate accordingly. On the syntactic side, Paterson's work introduced a convenient notation for programming with arrows in Haskell, providing support for recursive bindings as well. However, rather than letting the translation figure out recursive segments as we do in the mdo-notation, Paterson prefers using an explicit keyword, **rec**, asking the programmers to mark recursive blocks explicitly. (The **rec** keyword is modeled after O'Haskell's handling of recursive bindings, reviewed below.)

Building on our initial paper of monadic fixed-points [18], Benton and Hyland take Hasegawa's work one step further by generalizing the notion of trace to premonoidal categories [5]. (It turns out that Benton and Hyland's axiomatization and Paterson's work on arrows are essentially the same, although developed independently and presented in slightly different contexts.) Similar to Paterson's *loop* axioms, Benton and Hyland's axiomatization is too strong for many monads as well. As we have seen in the second half of Chapter 6, their sliding and right tightening laws are simply not satisfiable in many practical cases (see also Section 3.1). As a consequence, their work can explain value recursion for the state monad (and those monads that embed into it, such as output and environments), but not exceptions, lists, or the I/O monad. In general, any monad that is based on a sum-like data type will fail to satisfy their requirements. In any case, we consider Paterson and Benton and Hyland's work as an important step toward a categorical account of value recursion.

Friedman and Sabry [25] approach value recursion form an entirely different angle. Rather than considering individual monads separately, they consider recursion itself as a computational effect, following an operational definition: Allocate a reference cell, evaluate the body, and update the cell with the result. (This process is essentially how Scheme models recursion, as we have briefly covered in Section 5.3.) Since recursion is performed in the combined monad, it is the users' responsibility to translate original problems and values to and from this combined world. That is, to model value recursion in a monad m, they end up using a function:

$$mfixM :: (STM\ s\ m\ \alpha \to STM\ s\ m\ \alpha) \to STM\ s\ m\ \alpha$$

where STM is the state monad transformer.[2] Furthermore, all the morphisms of the base monad have to be lifted into this "state enriched" world as well, and this is where

[2] Note that *mfixM* accepts a function from computations to computations, rather than from values to computations as in the case for *mfix*. This change of view is necessary for implementing the allocate-evaluate-overwrite model.

the interaction between particular effects and recursion has to be addressed by the user. Unlike us, however, they do not postulate any properties, hence it is up to the user to come up with correct liftings. As Friedman and Sabry observe themselves, their method is rather inconvenient to use from a programming perspective, compared to our mdo-expressions and direct handling of recursion in the given monad. Unfortunately, a similar comparison is not immediately possible from a theoretical point of view, as the approaches are fundamentally different.

From a practical point of view, much greater similarity to our work is found in Nordlander's O'Haskell language. O'Haskell is an object oriented extension of Haskell, designed for addressing issues in reactive functional programming [65]. One application of O'Haskell is in programming layered network protocols. Each layer interacts with its predecessor and successor by receiving and passing information in both directions. In order to connect two protocols that have mutual dependencies, one needs a recursive knot-tying operation. Since O'Haskell objects are monadic, value recursion is employed in establishing such connections. O'Haskell adds a keyword **fix** to the do-notation, whose translation is a simplified version of our mdo-notation. The O'Haskell work, however, does not try to axiomatize or generalize the idea any further.

Carlsson and Hallgren discuss a variety of *loop* operators in the context of their work on stream based programming using *fudgets* [29]. Although the intended semantics of their *loop* operators is quite similar to those of value recursion operators, the types and the mechanics are somewhat different. For instance, one of their operators has the type:

$$\mathit{loopLeftF} :: F\ (\mathit{Either}\ \alpha\ \beta)\ (\mathit{Either}\ \alpha\ \gamma) \to F\ \beta\ \gamma$$

which, intuitively, ties the recursive loop over α, resulting in a fudget from β to γ. Carlsson and Hallgren use loop operators only in the framework of fudgets, without generalizing to arbitrary monads, or studying their behavior more abstractly.

The circuit modeling example we have seen in Section 1.2 is discussed in detail in Claessen's recent dissertation [12]. Although Claessen points out the need for an appropriate looping combinator, he does not pursue the monadic approach any further. Instead, he introduces the notion of *observable sharing*, which is a non-conservative[3] extension to Haskell [14]. (Briefly, observable sharing allows programmers to determine whether a circuit component is reached via a feedback loop, solving the infinite unfolding problem.) Claessen argues that "*...loop combinators are unfortunate because they introduce extra clutter in the code that is hard to motivate*" [12]. We believe that our mdo-notation addresses Claessen's concerns perfectly, relieving the programmers from error-prone and

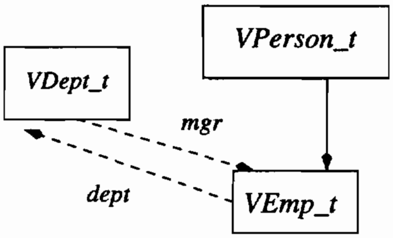
[3] Since the addition of observational sharing violates referential transparency, the resulting language is no longer pure. That is, the law: **let** $x = M$ **in** $N \equiv N[\mathit{fix}\ (\lambda x.\ M)/x]$ no longer holds.

cumbersome uses of explicit looping combinators. In addition, the monadic approach has the obvious advantage of keeping the underlying language pure, providing a nice and clean semantic framework.

Turbak and Wells introduce the *cycamore* data type, which is aimed at simplifying the use of cyclic structures in declarative languages [86]. The basic idea is to associate each node in a cycamore with a global unique identifier, similar to our doubly linked list example of Section 9.4. They consider implementations in both ML and Haskell, and the Haskell version makes use of references in the state monad to implement unique identifiers. As expected, Turbak and Wells employ value recursion in order to express the required cyclic structure.

10.2 Future Work

Although we have concentrated on applications in functional programming, value recursion certainly makes sense in other programming paradigms as well. One future research direction to explore is the problem of creating cyclic structures in imperative languages. Such structures arise quite frequently in practice. For instance, the following example presents an opportunity in IBM's data manipulation language for its DB2 database system [11]:[4]



```
create type VDept_t as
      (name Varchar(20)) mode db2sql;

create type VPerson_t as
      (name Varchar(40)) mode db2sql;

create type VEmp_t under VPerson_t as
      (dept Ref(VDept_t)) mode db2sql;

alter type VDept_t
     add attribute mgr Ref(VEmp_t);
```

In this example, the user creates three types: department, person, and employee. Each department has a name and a manager. Each person is identified by a name. Finally, each employee is a `VPerson_t`, which further has a (reference to a) particular department. While the `create type` directives for `VPerson_t` and `VEmp_t` reflect the structure correctly, the `VDept_t` type cannot be created with both of its required attributes. Clearly, the difficulty arises as the `VEmp_t` type is not yet visible when `VDept_t` is created. The final `alter type` directive remedies the situation in a roundabout fashion, adding the missing attribute.

[4] This example was pointed out to us by David Maier.

We see two opportunities with regard to our research. First of all, better syntactic support (along the lines of our mdo-notation) would help get rid of the final `alter type` directive, keeping the declaration of `VDept_t` self-contained, possibly simplifying further analyses. More importantly, if such declarations are ever given a monadic semantics, value recursion would be the right tool for modeling the cyclic dependency. Similar opportunities exist in other languages as well.

On the theoretical side, we would like to see value recursion studied in a more abstract setting. In this regard, the *trace-fixed point* correspondence, as we have studied in Chapter 6, seems to be the right direction to proceed. We would like to investigate the reasons why the axiomatization via traces turns out to be too strong, hopefully augmenting the theory to capture the practical aspects more precisely. For instance, it would be interesting to pin down the role of the right shrinking property precisely. As we have seen in Chapter 3, right shrinking property is not satisfiable whenever the $\gg\!=$ operator is strict in its first argument, and hence a weakening of the trace-based axiomatizations seems inevitable.

Several questions remain to be explored regarding the behavior of value recursion operators. For instance, we lack a reasoning principle along the lines of fixed-point induction. Recall that the fixed-point induction principle states that $P\ (\mathit{fix}\ f)$ can be established by showing that $P\ \bot \wedge \forall d.(P\ d \Rightarrow P\ (f\ d))$ holds, provided P is an admissible predicate. (The obvious generalization: $P\ \bot \wedge \forall d.(P\ d \Rightarrow P\ (d \gg\!= f)) \Rightarrow P\ (\mathit{mfix}\ f)$ is *not* sound, as it implicitly assumes an unfolding view of value recursion.) It is probably the case that one needs to formulate and prove a separate induction principle for each new *mfix*, rather than looking for a universal principle that would work for all cases. While our properties provide a framework for reasoning about terms involving *mfix*, such an induction principle might prove essential for reasoning about value recursion in general.

Another question is the automatic construction of value recursion operators for arbitrary monads. Although we have seen many "design patterns," it is still not clear how to define an appropriate operator for a given monad that will satisfy our properties. (The continuation monad seems to be the problem child in this regard.) Although it is highly unlikely that a magic recipe for automatic construction of such operators exists, it would be nice to pin down the exact conditions under which their existence (and possibly uniqueness) can be guaranteed.

The semantics we have presented in Chapter 8 for modeling monadic I/O needs some improvements to simplify reasoning with symbolic terms. Furthermore, we would like to extend our language to support more features, such as concurrency and exceptions. While concurrency seems relatively easy to support, it is not immediately clear how to extend our system to include Haskell'98 style exceptions. More importantly, it would be interesting to show that the addition of monadic I/O primitives, mutable variables, and support for value recursion preserves the parametricity principle. Also, we would like to design an accompanying abstract machine semantics, which might be useful as a basis for constructing interpreters for similar languages.

Whether the mdo-notation should eventually replace the current do-notation in Haskell is a question that will have to be answered by the Haskell community. While we believe that a single construct should handle both recursive and non-recursive cases, such a change potentially breaks existing programs, and it might be a better idea to make the switch in a future version of Haskell.

Appendix A

Fixed-point operators

In this appendix, we briefly review fixed-point operators. Our aim is to introduce the terminology we use, providing pointers to the literature for details as necessary.

In the domain theoretic semantics of programming languages, types are modeled by domains and functions are modeled by continuous maps. The meaning of a typical recursive declaration of the form **let** $x = M$ **in** N is taken to be $N\ [\mathit{fix}\ (\lambda x.M)/x]$, where

$$\mathit{fix}\ f = \bigsqcup_i f^i \perp_\alpha \tag{A.1}$$

assuming M has type α. Note that x need *not* be a function only, we might define recursive values this way as well. For instance, we have (using Haskell-like notation):

$$\textbf{let}\ \mathit{ones} = 1 : \mathit{ones}\ \textbf{in}\ \mathit{ones} \qquad \equiv \qquad \mathit{fix}\ (\lambda \mathit{ones}.\ 1 : \mathit{ones})$$

The least fixed-point theorem states that $\mathit{fix}\ f$ is the *least fixed-point* of f [76, 92]. That is, (i) it satisfies the fixed-point property: $f\ (\mathit{fix}\ f) = \mathit{fix}\ f$, and (ii) it is the *least* such value, i.e., for all x s.t. $x = f\ x$, we have $\mathit{fix}\ f \ \sqsubseteq x$. We use the name *fix* only to mean this particular fixed-point operator over domains.

The theory of fixed points is extensively studied [9, 10, 81]. It is neither possible, nor necessary for us to summarize this huge body of work here; we will simply state the results that are most relevant to our work.

Property A.1 (*Dinaturality.*) Let $f :: \alpha \rightarrow \beta$, $g :: \beta \rightarrow \alpha$. The *dinaturality*[1] property of *fix* states that:

$$\mathit{fix}\ (f \cdot g) = f\ (\mathit{fix}\ (g \cdot f))$$

[1] The term *dinaturality* refers to the fact that *fix* can be viewed as a dinatural transformation between certain functors [55, 80]. We will not need this level of detail in our work, so we skip the details.

Property A.2 (*Bekič.*) Let $f :: \alpha \times \alpha \rightarrow \alpha$. The Bekič property of *fix* states that:

$$\mathit{fix}\ (\lambda x.\ \mathit{fix}\ (\lambda y.\ f\ (x,\ y))) = \mathit{fix}\ (\lambda x.\ f\ (x,\ x))$$

Or, equivalently,

$$\mathit{fix}\ (\lambda t.\ \mathit{fix}\ (\lambda v.\ f\ (\pi_1\ t,\ \pi_2\ v))) = \mathit{fix}\ f$$

where $f :: \alpha \times \beta \rightarrow \alpha \times \beta$. It is easy to generalize to arbitrary number of variables, rather than just two; see Winskel's textbook for details [92].[2]

In Chapter 6, we consider fixed-point operators in more abstract settings, i.e., without assuming that the underlying structure is domains and continuous maps. We assume a minimal acquaintance with category theory in the following discussion [2, 70]. The basic structure we work with is a category $\mathcal{C}$ with finite products. We write $\mathbb{1}$ for the terminal object. The set of arrows between two objects A and B is denoted $\mathcal{C}(A,\ B)$. We will need the following basic definitions [33, 79]:

Definition A.3 A *fixed-point operator* is a family of functions $(\cdot)^*_A : \mathcal{C}(A,\ A) \rightarrow \mathcal{C}(\mathbb{1},\ A)$, such that for any $f : A \rightarrow A$, $f \cdot f^* = f^*$.

Definition A.4 A *parameterized fixed-point operator* is a family of functions:

$$(\cdot)^\dagger_{A,X} : \mathcal{C}(A \times X,\ X) \rightarrow \mathcal{C}(A,\ X)$$

satisfying:

- Parameterized fixed-point property: For $f : A \times X \rightarrow X$, $f \cdot \langle id_A,\ f^\dagger \rangle = f^\dagger$.
- Naturality in A: For $f : A \times X \rightarrow X$ and $g : B \rightarrow A$, $(f \cdot (g \times id_X))^\dagger = f^\dagger \cdot g$.

Definition A.5 A *Conway operator* is a parameterized fixed-point operator that further satisfies:

- Dinaturality: For $f : A \times X \rightarrow Y$ and $g : A \times Y \rightarrow X$, $(g \cdot \langle \pi_1^{A,X},\ f \rangle)^\dagger = g \cdot \langle id_A,\ (f \cdot \langle \pi_1^{A,Y},\ g \rangle)^\dagger \rangle$.
- Diagonal property: For $f : A \times X \times X \rightarrow X$, $f^{\dagger\dagger} = (f \cdot (id_A \times \langle id_X,\ id_X \rangle))^\dagger$.

The reader need not master these definitions in full, only a basic familiarity is sufficient. For the most part we will be working with *fix*, and using the dinaturality and Bekič properties given before, which are much easier to read and understand.

[2] Bekič's property appears in many different but equivalent forms in the literature [3]. The versions we have given here are the ones that are most suitable for our purposes.

Appendix B

Proofs

In the following proofs, we assume true products. In the case of lifted products, special care must be taken to ensure that the difference between $(\bot, \bot)$ and $\bot$ is not visible. The cases when the distinction does matter have been pointed out in the text. (See Warning 2.6.7 as well.)

To save space, we will shorten *return* to η in our proofs. Also note that we use the name *map* to refer to Haskell's *fmap*, i.e., $map :: (\alpha \to \beta) \to m\ \alpha \to m\ \beta$ for all monads m, defined by the equation $map\ f\ m = m \gg\!\!= \eta \cdot f$.

B.1 Proposition 2.5.2

Given Equation 2.7, establishing 2.8 is easy. We have:

$$\begin{aligned}
& \mathit{mfix}\ (\lambda(x,\ \_).\ \mathit{mfix}\ (\lambda(\_,\ y).\ f\ (x,\ y))) \\
=\ & \mathit{mfix}\ (\lambda t.\ \mathit{mfix}\ (\lambda u.\ f\ (\pi_1\ t,\ \pi_2\ u))) \\
=\ & \mathit{mfix}\ (\lambda t.\ \mathit{mfix}\ (\lambda u.\ (\lambda(x,\ y).\ f\ (\pi_1\ x,\ \pi_2\ y))\ (t,\ u))) \\
=\ & \mathit{mfix}\ (\lambda t.\ (\lambda(x,\ y).\ f\ (\pi_1\ x,\ \pi_2\ y))\ (t,\ t)) && \{\mathit{Equation}\ 2.7\} \\
=\ & \mathit{mfix}\ (\lambda t.\ f\ (\pi_1\ t,\ \pi_2\ t)) \\
=\ & \mathit{mfix}\ f
\end{aligned}$$

In the last step, we used the fact that $(\pi_1\, t, \pi_2\, t) = t$, which only holds for true products.

To show the correspondence in the other direction, let $\Delta\ x = (x, x)$, and note that Δ is strict (again thanks to true products). We have:

$$\begin{aligned}
& \mathit{mfix}\ (\lambda x.\ f\ (x,\ x)) \\
=\ & \mathit{mfix}\ (f \cdot \Delta) \\
=\ & \mathit{map}\ (\pi_1 \cdot \Delta)\ (\mathit{mfix}\ (f \cdot \Delta)) && \{\pi_1 \cdot \Delta = \mathit{id}\} \\
=\ & \mathit{map}\ \pi_1\ (\mathit{map}\ \Delta\ (\mathit{mfix}\ (f \cdot \Delta))) \\
=\ & \mathit{map}\ \pi_1\ (\mathit{mfix}\ (\mathit{map}\ \Delta \cdot f)) && \{\mathit{slide}\}
\end{aligned}$$

$$\begin{aligned}
&= \mathit{map}\ \pi_1\ (\mathit{mfix}\ (\lambda x.\ \mathit{mfix}\ (\lambda y.\ (\mathit{map}\ \Delta \cdot f)\ (\pi_1\ x,\ \pi_2\ y)))) && \{\mathit{Equation}\ 2.8\}\\
&= \mathit{map}\ \pi_1\ (\mathit{mfix}\ (\lambda x.\ \mathit{mfix}\ (\mathit{map}\ \Delta \cdot (\lambda y.\ f\ (\pi_1\ x,\ y)) \cdot \pi_2))) && \\
&= \mathit{map}\ \pi_1\ (\mathit{mfix}\ (\lambda x.\ \mathit{map}\ \Delta\ (\mathit{mfix}\ (\lambda y.\ f\ (\pi_1\ x,\ y))))) && \{\mathit{slide}\}\\
&= \mathit{map}\ \pi_1\ (\mathit{mfix}\ (\mathit{map}\ \Delta \cdot (\lambda x.\ \mathit{mfix}\ (\lambda y.\ f\ (x,\ y))) \cdot \pi_1)) && \\
&= (\mathit{map}\ \pi_1 \cdot \mathit{map}\ \Delta)\ (\mathit{mfix}\ (\lambda x.\ \mathit{mfix}\ (\lambda y.\ f\ (x,\ y)))) && \{\mathit{slide}\}\\
&= \mathit{mfix}\ (\lambda x.\ \mathit{mfix}\ (\lambda y.\ f\ (x,\ y))) &&
\end{aligned}$$

In case of lifted products (Proposition 2.5.4), the proof proceeds similarly. The last step in the first proof is not applicable, but in that case we can replace the last line with $\mathit{mfix}\ (\lambda\,\tilde{}\,(x,\ y).\ f\ (x,\ y))$, which is predicted by Equation 2.10. The second implication follows similarly.

B.2 Proposition 2.6.8

$$\begin{aligned}
&\ \mathit{mfix}\ (\lambda(x,\ y).\ f\ x \gg\!=\ \lambda z.\ \eta\ (z,\ h\ z\ (x,\ y))) && \\
&= \mathit{mfix}\ (\lambda t.\ (f \cdot \pi_1)\ t \gg\!=\ \lambda z.\ \eta\ (z,\ h\ z\ t)) && \\
&= \mathit{mfix}\ (\lambda t.\ (\lambda(u,\ v).\ (f \cdot \pi_1)\ u \gg\!=\ \lambda z.\ \eta\ (z,\ h\ z\ v))\ (t,\ t)) && \\
&= \mathit{mfix}\ (\lambda x.\ \mathit{mfix}\ (\lambda y.\ (f \cdot \pi_1)\ x \gg\!= \lambda z.\ \eta\ (z,\ h\ z\ y))) && \{\mathit{nest}\}\\
&= \mathit{mfix}\ (\lambda x.\ (f \cdot \pi_1)\ x \gg\!= \lambda z.\ \mathit{mfix}\ (\lambda y.\ \eta\ (z,\ h\ z\ y))) && \{\mathit{left\ shrink}\}\\
&= \mathit{mfix}\ (\lambda x.\ (f \cdot \pi_1)\ x \gg\!= \lambda z.\ \eta\ (\mathit{fix}\ (\lambda y.\ (z,\ h\ z\ y)))) && \{\mathit{purity}\}\\
&= \mathit{mfix}\ (\lambda x.\ (f \cdot \pi_1)\ x \gg\!= \lambda z.\ \eta\ (z,\ \mathit{fix}\ (\lambda y.\ h\ z\ (z,\ y)))) && \{\mathit{nest}\ (\mathit{fix})\}\\
&= \mathit{mfix}\ f \gg\!= \lambda z.\ \eta\ (z,\ \mathit{fix}\ (\lambda y.\ h\ z\ (z,\ y))) && \{\mathit{pure\ right}\}\\
&= \mathit{mfix}\ f \gg\!= \lambda z.\ \eta\ (\mathit{fix}\ (\lambda(x,\ y).\ (z,\ h\ z\ (x,\ y)))) && \{\mathit{nest}\ (\mathit{fix})\}
\end{aligned}$$

B.3 Proposition 2.7.1

$$\begin{aligned}
&\ \mathit{mfix}\ (\lambda(x,\ \_).\ f\ x \gg\!=\ \lambda y.\ \eta\ (q,\ y)) && \\
&= \mathit{mfix}\ (\mathit{map}\ (\lambda y.\ (q,\ y)) \cdot f \cdot \pi_1) && \\
&= \mathit{map}\ (\lambda y.\ (q,\ y))\ (\mathit{mfix}\ (f \cdot \pi_1 \cdot (\lambda y.\ (q,\ y)))) && \{\mathit{strong\ sliding}\}\\
&= \mathit{map}\ (\lambda y.\ (q,\ y))\ (\mathit{mfix}\ (\lambda y.\ f\ q)) && \\
&= \mathit{map}\ (\lambda y.\ (q,\ y))\ (f\ q) && \{\mathit{constant\ functions}\}\\
&= f\ q \gg\!=\ \lambda y.\ \eta\ (q,\ y) &&
\end{aligned}$$

The need for strong sliding is obvious, since otherwise we would have to require $f\ q = f\ \bot$ to satisfy the precedent.

B.4 Lemma 3.1.4

Recall that η is a natural transformation, that is, it satisfies the equality $\mathit{map}\ h \cdot \eta_\alpha = \eta_\beta \cdot h$ for all $h :: \alpha \to \beta$. Assume η is strict at the type α, i.e., $\eta_\alpha \perp_\alpha = \perp_{m\ \alpha}$, and $\gg\!=$ is strict in its first argument. Pick an arbitrary type β. We will show that $\eta_\beta \perp_\beta = \perp_{m\ \beta}$:

$$\begin{array}{lll}
 & \eta_\beta \perp_\beta & \\
= & \eta_\beta\ (\mathit{const} \perp_\beta \perp_\alpha) & \\
= & (\eta_\beta \cdot \mathit{const} \perp_\beta) \perp_\alpha & \\
= & (\mathit{map}\ (\mathit{const} \perp_\beta) \cdot \eta_\alpha) \perp_\alpha & \{\mathit{naturality\ of}\ \eta\} \\
= & \mathit{map}\ (\mathit{const} \perp_\beta)\ (\eta_\alpha \perp_\alpha) & \\
= & \mathit{map}\ (\mathit{const} \perp_\beta) \perp_{m\ \alpha} & \{\mathit{assumption:}\ \eta_\alpha \perp_\alpha = \perp_{m\ \alpha}\} \\
= & \perp_{m\ \alpha} \gg\!= \eta \cdot \mathit{const} \perp_\beta & \{\mathit{definition\ of\ map}\ \} \\
= & \perp_{m\ \beta} & \{\mathit{assumption:} \gg\!= \mathit{is\ left\text{-}strict}\}
\end{array}$$

B.5 Proposition 3.4.2

Given arbitrary f and g, define:

$$\begin{array}{l}
h\ x\ 1 = f\ x \\
h\ x\ 2 = g\ x
\end{array}$$

We have:

$$\begin{array}{lll}
 & \mathit{mfix}\ (\lambda x.\ f\ x \oplus g\ x) & \\
= & \mathit{mfix}\ (\lambda x.\ h\ x\ 1 \oplus h\ x\ 2) & \\
= & \mathit{mfix}\ (\lambda x.\ (\eta\ 1 \gg\!= \lambda y.\ h\ x\ y) \oplus (\eta\ 2 \gg\!= \lambda y.\ h\ x\ y)) & \\
= & \mathit{mfix}\ (\lambda x.\ (\eta\ 1 \oplus \eta\ 2) \gg\!= \lambda y.\ h\ x\ y) & \{\mathit{Eqn.}\ 3.8\} \\
= & (\eta\ 1 \oplus \eta\ 2) \gg\!= \lambda y.\ \mathit{mfix}\ (\lambda x.\ h\ x\ y) & \{\mathit{left\ shrink}\} \\
= & \mathit{mfix}\ (\lambda x.\ h\ x\ 1) \oplus \mathit{mfix}\ (\lambda x.\ h\ x\ 2) & \{\mathit{Eqn.}\ 3.8\} \\
= & \mathit{mfix}\ f \oplus \mathit{mfix}\ g &
\end{array}$$

B.6 Proposition 4.3.1

We consider each case in turn:

4.5: Right to left implication is immediate. From left to right, $\mathit{fix}\ (f \cdot \mathit{head})$ must be $\perp$, which only happens when $f \perp = \perp$. (Note that this establishes the strictness property.)

4.6: Similar to the previous case.

4.7: Simple case analysis. If $\mathit{mfix}\, f$ is $\perp$, f is strict by 4.5, and both sides reduce to $\perp$. If $\mathit{mfix}\ f$ is $[\ ]$, then $f \perp = [\ ]$ by 4.6, reducing both sides to $\perp$ again.

Finally, if *mfix f* is a cons-cell, the **case** expression must take its second branch, i.e., $\mathit{head}\ (\mathit{mfix}\ f) = \mathit{head}\ (\mathit{fix}\ (f \cdot \mathit{head}))$, which is exactly the right hand side by the dinaturality of *fix*.

4.8: Similar to the previous case, if *mfix f* equals $\perp$ or [ ], both sides yield $\perp$. Otherwise, **case** must take its second branch, i.e., $\mathit{tail}\ (\mathit{mfix}\ f) = \mathit{mfix}\ (\mathit{tail} \cdot f)$.

4.9: Consider the test expression for **case**. We have:

$$\begin{aligned}
\mathit{fix}\ ((\lambda x.\ f\ x : g\ x) \cdot \mathit{head}) &= (\lambda x.\ f\ x : g\ x)\ (\mathit{fix}\ (\mathit{head} \cdot (\lambda x.\ f\ x : g\ x)))\\
&= (\lambda x.\ f\ x : g\ x)\ (\mathit{fix}\ f)\\
&= f\ (\mathit{fix}\ f) : g\ (\mathit{fix}\ f)\\
&= \mathit{fix}\ f : g\ (\mathit{fix}\ f)
\end{aligned}$$

Hence, the **case** expression takes its second branch, yielding:

$$\begin{aligned}
\mathit{mfix}\ (\lambda x.\ f\ x : g\ x) &= \mathit{fix}\ f : \mathit{mfix}\ (\mathit{tail} \cdot (\lambda x.\ f\ x : g\ x))\\
&= \mathit{fix}\ f : \mathit{mfix}\ g
\end{aligned}$$

4.10: We will use the approximation lemma [7, 38], which states that:

$$(\forall n.\ \mathit{approx}\ n\ xs = \mathit{approx}\ n\ ys) \Rightarrow xs = ys$$

for arbitrary lists *xs* and *ys*. The function *approx* is defined as:

$$\begin{array}{lll}
\mathit{approx} & & :: \mathit{Integer} \rightarrow [\alpha] \rightarrow [\alpha]\\
\mathit{approx}\ 0 & xs & = \perp\\
\mathit{approx}\ (n{+}1) & \perp & = \perp\\
\mathit{approx}\ (n{+}1) & [\,] & = [\,]\\
\mathit{approx}\ (n{+}1) & (x{:}xs) & = x : \mathit{approx}\ n\ xs
\end{array}$$

We will prove:

$$\forall n. \forall f, g.\ \mathit{approx}\ n\ (\mathit{mfix}\ (\lambda x.\ f\ x +\!\!+\ g\ x)) = \mathit{approx}\ n\ (\mathit{mfix}\ f +\!\!+\ \mathit{mfix}\ g)$$

by induction on n, implying the required result. Base case ($n = 0$) is trivial. The induction hypothesis is:

$$\forall f, g.\ \mathit{approx}\ k\ (\mathit{mfix}\ (\lambda x.\ f\ x +\!\!+\ g\ x)) = \mathit{approx}\ k\ (\mathit{mfix}\ f +\!\!+\ \mathit{mfix}\ g) \qquad \text{(B.1)}$$

Note that the hypothesis is assumed for all f and g. This generality will be essential in establishing the induction step. We need to show:

$$\forall f, g.\ \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ (\lambda x.\ f\ x +\!\!+\ g\ x)) = \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ f +\!\!+\ \mathit{mfix}\ g)$$

Pick two arbitrary functions f', $g' :: \alpha \rightarrow [\alpha]$. It suffices to show that:

$$\mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ (\lambda x.\ f'\ x +\!\!+\ g'\ x)) = \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ f' +\!\!+\ \mathit{mfix}\ g') \qquad \text{(B.2)}$$

which we establish by case analysis on $f'\ \bot$. The cases $\bot$ and $[\,]$ are immediate. By 4.5 and 4.6, both sides reduce to $\bot$ and *approx* $(k+1)$ (*mfix* g'), respectively. If $f'\ \bot$ is a cons-cell, it follows that

$$\forall x.\ f'\ x = (\mathit{head} \cdot f')\ x\ :\ (\mathit{tail} \cdot f')\ x \tag{B.3}$$

Simple inspection of the definition of *mfix* reveals that *mfix* f' must be a cons-cell in this case as well. Hence, we have:

$$\mathit{mfix}\ f' = \mathit{head}\ (\mathit{mfix}\ f')\ :\ \mathit{tail}\ (\mathit{mfix}\ f') \tag{B.4}$$

Therefore,

$$\begin{array}{lll}
 & \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ (\lambda x.\ f'\ x +\!\!+\ g'\ x)) & \\
= & \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ (\lambda x.\ ((\mathit{head} \cdot f')\ x : (\mathit{tail} \cdot f')\ x) +\!\!+\ g'\ x)) & \{\mathit{Eqn.}\ \mathrm{B.3}\} \\
= & \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ (\lambda x.\ (\mathit{head} \cdot f')\ x : ((\mathit{tail} \cdot f')\ x +\!\!+\ g'\ x))) & \\
= & \mathit{approx}\ (k{+}1)\ (\mathit{fix}\ (\mathit{head} \cdot f')\ :\ \mathit{mfix}\ (\lambda x.\ (\mathit{tail} \cdot f')\ x +\!\!+\ g'\ x)) & \{\mathit{Eqn.}\ 4.9\} \\
= & \mathit{fix}\ (\mathit{head} \cdot f')\ :\ (\mathit{approx}\ k\ (\mathit{mfix}\ (\lambda x.\ (\mathit{tail} \cdot f')\ x\ +\!\!+\ g'\ x))) & \\
= & \mathit{fix}\ (\mathit{head} \cdot f')\ :\ (\mathit{approx}\ k\ (\mathit{mfix}\ (\mathit{tail} \cdot f')\ +\!\!+\ \mathit{mfix}\ g')) & \{\mathit{I.H.}\} \\
= & \mathit{approx}\ (k{+}1)\ ((\mathit{fix}\ (\mathit{head} \cdot f')\ :\ \mathit{mfix}\ (\mathit{tail} \cdot f'))\ +\!\!+\ \mathit{mfix}\ g') & \\
= & \mathit{approx}\ (k{+}1)\ ((\mathit{head}\ (\mathit{mfix}\ f')\ :\ \mathit{tail}\ (\mathit{mfix}\ f'))\ +\!\!+\ \mathit{mfix}\ g') & \{\mathit{Eqns.}\ 4.7,\ 4.8\} \\
= & \mathit{approx}\ (k{+}1)\ (\mathit{mfix}\ f'\ +\!\!+\ \mathit{mfix}\ g') & \{\mathit{Eqn.}\ \mathrm{B.4}\}
\end{array}$$

completing the proof.

B.7 Proposition 4.9.1

We need to show that the function *mfixErrM* satisfies strictness, purity and left shrinking properties. All cases follow from the corresponding properties of *mfixM*, and simple symbolic manipulation. We will only present the left shrinking case to illustrate the technique. To avoid confusion due to overloaded operators, we will write *returnM* and *bindM* for the morphisms of m, while *returnErrM* and *bindErrM* for those of *Err m*.

$$\begin{array}{ll}
 & \mathit{mfixErrM}\ (\lambda x.\ a\ \text{`}\mathit{bindErrM}\text{`}\ \lambda y.\ f\ x\ y) \\
= & \{\mathit{Equation}\ 4.36,\ \mathit{expand}\ \mathit{bindErrM}\} \\
 & \mathit{mfixM}\ (\lambda x.\ a\ \text{`}\mathit{bindErrM}\text{`}\ \lambda y.\ f\ (\mathit{unErr}\ x)\ y) \\
= & \{\mathit{expand}\ \mathit{bindErrM}\} \\
 & \mathit{mfixM}\ (\lambda x.\ a\ \text{`}\mathit{bindM}\text{`}\ \lambda y.\ \textbf{case}\ y\ \textbf{of} \\
 & \qquad\qquad\qquad\qquad\qquad\quad \mathit{Ok}\ q\ \rightarrow f\ (\mathit{unErr}\ x)\ q \\
 & \qquad\qquad\qquad\qquad\qquad\quad \mathit{Err}\ s\ \rightarrow \mathit{returnM}\ (\mathit{Err}\ s))
\end{array}$$

$$
\begin{array}{ll}
= & \{\textit{left shrinking on mfixM}\}\\
 & a\ \text{‘}\textit{bindM}\text{‘}\ \lambda y.\ \textit{mfixM}\ (\lambda x.\ \textbf{case}\ y\ \textbf{of}\\
 & \qquad\qquad\qquad\qquad\qquad\qquad \textit{Ok}\ q\ \rightarrow f\ (\textit{unErr}\ x)\ q\\
 & \qquad\qquad\qquad\qquad\qquad\qquad \textit{Err}\ s\ \rightarrow \textit{returnM}\ (\textit{Err}\ s))\\
= & \{\textit{Proposition } 2.6.2,\ \textbf{case}\ \textit{is a shortcut for}\ \textbf{if}\}\\
 & a\ \text{‘}\textit{bindM}\text{‘}\ \lambda y.\ \textbf{case}\ y\ \textbf{of}\\
 & \qquad\qquad\qquad \textit{Ok}\ q\ \rightarrow \textit{mfixM}\ (\lambda x.\ f\ (\textit{unErr}\ x)\ q)\\
 & \qquad\qquad\qquad \textit{Err}\ s\ \rightarrow \textit{mfixM}\ (\lambda x.\ \textit{returnM}\ (\textit{Err}\ s))\\
= & \{\textit{fold down mifxErrM on the first branch, Proposition } 2.6.1 \textit{ on the second}\}\\
 & a\ \text{‘}\textit{bindM}\text{‘}\ \lambda y.\ \textbf{case}\ y\ \textbf{of}\\
 & \qquad\qquad\qquad \textit{Ok}\ q\ \rightarrow \textit{mfixErrM}\ (\lambda x.\ f\ x\ q)\\
 & \qquad\qquad\qquad \textit{Err}\ s\ \rightarrow \textit{returnM}\ (\textit{Err}\ s)\\
= & \{\textit{fold down bindErrM}\}\\
 & a\ \text{‘}\textit{bindErrM}\text{‘}\ \lambda y.\ \textit{mfixErrM}\ (\lambda x.\ f\ x\ y)
\end{array}
$$

B.8 Proposition 6.3.5

We will need the following two lemmas:

Lemma B.8.1 Let T be a monad and *mfix* be a value recursion operator satisfying the right shrinking law. Let $f : X \rightarrow T\ (B \times X)$ and $g : B \times X \rightarrow T\ B'$. Then,

$$
\begin{array}{l}
\textit{mfix}\ (\lambda(\_,\ x).\ f\ x \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ \eta\ (w,\ \pi_2\ z))\\
\quad = \textit{mfix}\ (\lambda(\_,\ x).\ f\ x) \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ \eta\ (w,\ \pi_2\ z)
\end{array}
$$

Proof Note that the first *mfix* is at instance $B' \times X$, while the second is at $B \times X$. We reason as follows:

$$
\begin{array}{ll}
 & \textit{mfix}\ (\lambda(\_,\ x).\ f\ x \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ \eta\ (w,\ \pi_2\ z))\\
= & \textit{mfix}\ (\lambda(\_,\ x).\ f\ x \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ \eta\ (w,\ z) \gg\!= \lambda(p,\ q).\ \eta\ (p,\ \pi_2\ q))\\
= & \{\textit{slide},\ \lambda(p,\ q).\ (p,\ \pi_2\ q)\ \textit{is strict}\}\\
 & \textit{mfix}\ ((\lambda(\_,\ x).\ f\ x \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ \eta\ (w,\ z)) \cdot (\lambda(p,\ q).\ (p,\ \pi_2\ q)))\\
 & \qquad\qquad\qquad \gg\!= \lambda(p,\ q).\ \eta\ (p,\ \pi_2\ q)\\
= & \textit{mfix}\ (\lambda(\_,\ t).\ f\ (\pi_2\ t) \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ \eta\ (w,\ z)) \gg\!= \lambda(p,\ q).\ \eta\ (p,\ \pi_2\ q)\\
= & \{\textit{right shrinking}\}\\
 & \textit{mfix}\ (\lambda(\_,\ x).\ f\ x) \gg\!= \lambda z.\ g\ z \gg\!= \lambda w.\ \eta\ (w,\ \pi_2\ z)
\end{array}
$$

□

The second lemma states a variant of Equation 3.7:

Lemma B.8.2 Let $f :: \alpha \to m\ (\beta, \tau)$, $g :: \tau \to m\ \alpha$, where m is a commutative monad. Then,

$$\begin{aligned} &\mathit{mfix}\ (\lambda t.\ g\ (\pi_2\ t) \gg\!= f) \gg\!= \eta \cdot \pi_1 \\ &\quad = \mathit{mfix}\ (\lambda t.\ f\ (\pi_2\ t) \gg\!= \lambda t'.\ g\ (\pi_2\ t') \gg\!= \lambda a.\ \eta\ (\pi_1\ t',\ a)) \gg\!= \eta \cdot \pi_1 \end{aligned}$$

provided *mfix* satisfies strong sliding and nesting.

Proof (Sketch) Note that the first *mfix* is at instance $\beta \times \tau$, while the second one is at $\beta \times \alpha$. The proof first extends the recursion to $\alpha \times (\beta \times \tau)$, applies commutativity (Proposition 3.3.2), and then gets rid of the τ argument. □

To establish Proposition 6.3.5, we need to verify that the definition of trace as given by 6.21 satisfies Equations 6.14- 6.20. We consider each case in turn:

- Left tightening (6.14):

$$\begin{aligned} &\mathit{trace}\ (\lambda(a,\ x).\ g\ a \gg\!= \lambda a'.\ f\ (a',\ x)) \\ =\ &\lambda a.\ \mathit{mfix}\ (\lambda(b,\ x).\ g\ a \gg\!= \lambda a'.\ f\ (a',\ x)) \gg\!= \eta \cdot \pi_1 \\ =\ &\{\mathit{left\ shrinking\ on\ mfix}\} \\ &\lambda a.\ g\ a \gg\!= \lambda a'.\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a',\ x)) \gg\!= \eta \cdot \pi_1 \\ =\ &\lambda a.\ g\ a \gg\!= \mathit{trace}\ f \end{aligned}$$

- Right tightening (6.15):

$$\begin{aligned} &\mathit{trace}\ (\lambda(a,\ x).\ f\ (a,\ x) \gg\!= \lambda(b,\ x).\ g\ b \gg\!= \lambda b'.\ \eta\ (b',\ x)) \\ =\ &\lambda a.\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x) \gg\!= \lambda(b,\ x).\ g\ b \gg\!= \lambda b'.\ \eta\ (b',\ x)) \gg\!= \eta \cdot \pi_1 \\ =\ &\lambda a.\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x) \gg\!= \lambda z.(g \cdot \pi_1)\ z \gg\!= \lambda b'.\ \eta\ (b',\ \pi_2\ z)) \gg\!= \eta \cdot \pi_1 \\ =\ &\{\mathit{lemma}\ \text{B.8.1}\} \\ &\lambda a.\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x)) \gg\!= \lambda z.\ (g \cdot \pi_1)\ z \\ =\ &\lambda a.\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x)) \gg\!= \lambda z.\ \eta\ (\pi_1\ z) \gg\!= \lambda w.\ g\ w \\ =\ &\lambda a.\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x)) \gg\!= \eta \cdot \pi_1 \gg\!= g \\ =\ &\lambda a.\ \mathit{trace}\ f\ a \gg\!= g \end{aligned}$$

- Sliding (6.16):

$$\begin{aligned} &\mathit{trace}\ (\lambda(a,\ x).\ g\ x \gg\!= \lambda x'.\ f\ (a,\ x')) \\ =\ &\lambda a.\ \mathit{mfix}\ (\lambda(b,\ x).\ g\ x \gg\!= \lambda x'.\ f\ (a,\ x')) \gg\!= \eta \cdot \pi_1 \\ =\ &\lambda a.\ \mathit{mfix}\ (\lambda t.\ g\ (\pi_2\ t) \gg\!= \mathit{curry}\ f\ a) \gg\!= \eta \cdot \pi_1 \\ =\ &\{\mathit{Lemma}\ \text{B.8.2}\} \end{aligned}$$

$$\begin{aligned}
& \lambda a.\ \mathit{mfix}\ (\lambda t.\ \mathit{curry}\ f\ a\ (\pi_2\ t) \gg\!= \lambda t'.\ g\ (\pi_2\ t') \\
& \qquad\qquad\qquad\qquad \gg\!= \lambda x'.\ \eta\ (\pi_1\ t',\ x')) \gg\!= \eta \cdot \pi_1 \\
={} & \lambda a.\ \mathit{mfix}\ (\lambda (b,\ x').\ f\ (a,\ x') \gg\!= \lambda (b,\ x).\ g\ x \\
& \qquad\qquad\qquad\qquad\qquad \gg\!= \lambda x'.\ \eta\ (b,\ x')) \gg\!= \eta \cdot \pi_1 \\
={} & \mathit{trace}\ (\lambda (a,\ x').\ f\ (a,\ x') \gg\!= \lambda (b,\ x).\ g\ x \gg\!= \lambda x'.\ \eta\ (b,\ x'))
\end{aligned}$$

- Vanishing (6.17):

$$\begin{aligned}
& \mathit{trace}\ (\lambda (a,\ ()).\ f\ a \gg\!= \lambda b.\ \eta\ (b,\ ())) \\
={} & \lambda a.\ \mathit{mfix}\ (\lambda (b,\ ()).\ f\ a \gg\!= \lambda b.\ \eta\ (b,\ ())) \gg\!= \eta \cdot \pi_1 \\
={} & \{\mathit{constant\ functions}\} \\
& \lambda a.\ f\ a \gg\!= \lambda b.\ \eta\ (b,\ ()) \gg\!= \eta \cdot \pi_1 \\
={} & \lambda a.\ f\ a \gg\!= \lambda b.\ b \\
={} & f
\end{aligned}$$

- Vanishing (6.18): Let

$$\begin{aligned}
\mathit{asc}\ \ (x,\ (y,\ z)) &= ((x,\ y),\ z) \\
\mathit{iasc}\ \ ((x,\ y),\ z) &= (x,\ (y,\ z))
\end{aligned}$$

Then,

$$\begin{aligned}
& \mathit{trace}\ (\mathit{trace}\ (\lambda ((a,\ x),\ y).\ f\ (a,\ (x,\ y)) \gg\!= \lambda (b,\ (x,\ y)).\ \eta\ ((b,\ x),\ y))) \\
={} & \mathit{trace}\ (\mathit{trace}\ (\lambda t.\ f\ (\mathit{iasc}\ t) \gg\!= \eta \cdot \mathit{asc})) \\
={} & \mathit{trace}\ (\mathit{trace}\ (\mathit{map}\ \mathit{asc} \cdot f \cdot \mathit{iasc})) \\
={} & \mathit{trace}\ (\lambda (a,\ x).\ \mathit{mfix}\ (\lambda ((\_,\ \_),\ y).\ (\mathit{map}\ \mathit{asc} \cdot f \cdot \mathit{iasc})\ ((a,\ x),\ y)) \gg\!= \eta \cdot \pi_1) \\
={} & \mathit{trace}\ (\lambda (a,\ x).\ \mathit{mfix}\ (\mathit{map}\ \mathit{asc} \cdot (\lambda ((\_,\ \_),\ y).\ f\ (a,\ (x,\ y)))) \gg\!= \eta \cdot \pi_1) \\
={} & \{\mathit{slide,\ asc\ is\ strict}\ \} \\
& \mathit{trace}\ (\lambda (a,\ x).\ \mathit{mfix}\ (\lambda (\_,\ (\_,\ y)).\ f\ (a,\ (x,\ y))) \gg\!= \eta \cdot \mathit{asc} \gg\!= \eta \cdot \pi_1) \\
={} & \lambda a.\ \mathit{mfix}\ (\lambda (\_,\ x).\ \mathit{mfix}\ (\lambda (\_,\ (\_,\ y)).\ f\ (a,\ (x,\ y))) \gg\!= \eta \cdot \pi_1 \cdot \mathit{asc}) \\
& \qquad\qquad\qquad \gg\!= \eta \cdot \pi_1 \\
={} & \{\mathit{slide},\ \pi_1 \cdot \mathit{asc\ is\ strict}\ \} \\
& \lambda a.\ \mathit{mfix}\ ((\lambda (\_,\ x).\ \mathit{mfix}\ (\lambda (\_,\ (\_,\ y)).\ f\ (a,\ (x,\ y)))) \cdot \pi_1 \cdot \mathit{asc}) \\
& \qquad\qquad\qquad \gg\!= \eta \cdot (\pi_1 \cdot \pi_1 \cdot \mathit{asc}) \\
={} & \{\pi_1 \cdot \pi_1 \cdot \mathit{asc} = \pi_1\} \\
& \lambda a.\ \mathit{mfix}\ (\lambda (\_,\ (x,\ \_)).\ \mathit{mfix}\ (\lambda (\_,\ (\_,\ y)).\ f\ (a,\ (x,\ y)))) \gg\!= \eta \cdot \pi_1 \\
={} & \{\mathit{unnest\ triple}\} \\
& \lambda a.\ \mathit{mfix}\ (\lambda (\_,\ (x,\ y)).\ f\ (a,\ (x,\ y))) \gg\!= \eta \cdot \pi_1 \\
={} & \mathit{trace}\ f
\end{aligned}$$

- Superposing (6.19): Let *asc* be defined as above,

$$
\begin{aligned}
& \mathit{trace}\ (\lambda((c,\ a),\ x).\ f\ (a,\ x) \gg\!=\ \lambda(b,\ x).\ \eta\ ((c,\ b),\ x)) \\
= {} & \lambda(c,\ a).\ \mathit{mfix}\ (\lambda((\_,\ \_),\ x).\ f\ (a,\ x) \gg\!=\ \lambda(b,\ x).\ \eta\ ((c,\ b),\ x)) \gg\!=\ \eta \cdot \pi_1 \\
= {} & \{\mathit{slide}\} \\
& \lambda(c,\ a).\ \mathit{mfix}\ (\lambda(\_,\ (\_,\ x)).\ f\ (a,\ x) \gg\!=\ \lambda(b,\ x).\ \eta\ (c,\ (b,\ x))) \\
& \qquad\qquad \gg\!=\ \eta \cdot \pi_1 \cdot \mathit{asc} \\
= {} & \{\mathit{pure\ right\ shrinking}\} \\
& \lambda(c,\ a).\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x)) \gg\!= \lambda(b,\ x).\ \eta\ (c,\ b) \\
= {} & \lambda(c,\ a).\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x)) \gg\!= \lambda(b',\ x).\ \eta\ b' \gg\!=\ \lambda b.\ \eta\ (c,\ b) \\
= {} & \lambda(c,\ a).\ \mathit{mfix}\ (\lambda(b,\ x).\ f\ (a,\ x)) \gg\!=\ \eta \cdot \pi_1 \gg\!=\ \lambda b.\ \eta\ (c,\ b) \\
= {} & \lambda(c,\ a).\ \mathit{trace}\ f\ a \gg\!=\ \lambda b.\ \eta\ (c,\ b)
\end{aligned}
$$

- Yanking (6.20):

$$
\begin{aligned}
& \mathit{trace}\ (\lambda(a,\ a').\ \eta\ (a',\ a)) \\
= {} & \lambda a.\ \mathit{mfix}\ (\lambda(b,\ a').\ \eta\ (a',\ a)) \gg\!=\ \eta \cdot \pi_1 \\
= {} & \{\mathit{purity}\} \\
& \lambda a.\ \eta\ (\mathit{fix}\ (\lambda(b,\ a').\ (a',\ a))) \gg\!=\ \eta \cdot \pi_1 \\
= {} & \lambda a.\ \eta\ (a,\ a) \gg\!=\ \eta \cdot \pi_1 \\
= {} & \eta
\end{aligned}
$$

Bibliography

(Each entry is followed by a list of page numbers on which the citation appears. All cited URLs were last accessed in October 2002.)

[1] Achten, P., and Peyton Jones, S. Porting the Clean Object I/O Library to Haskell. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages* (2000), pp. 194–213. (98)

[2] Barr, M., and Wells, C. *Category Theory for Computing Science*, second ed. Prentice Hall International Series in Computer Science. Prentice Hall, 1995. (10, 46, 68, 75, 141)

[3] Bekič, H. *Programming Languages and their Definition. Selected Papers*, vol. 177 of *Lecture Notes in Computer Science.* Springer Verlag, 1984. (141)

[4] Benton, N., Hughes, J., and Moggi, E. Monads and effects. Lecture notes from APPSEM'00 Summer School. URL: `www.disi.unige.it/person/MoggiE/APPSEM00/BHM-revised.ps.gz`, 2000. (10, 130)

[5] Benton, N., and Hyland, M. Traced premonoidal categories (Extended Abstract). In *Fixed Points in Computer Science Workshop, FICS'02* (2002). (66, 79, 80, 81, 82, 83, 123, 135)

[6] Bird, R. S. Using circular programs to eliminate multiple traversals of data. *Acta Informatica 21* (1984), 239–250. (122)

[7] Bird, R. S. *Introduction to Functional Programming using Haskell*, second ed. Prentice Hall Europe, London, 1998. (10, 42, 47, 101, 145)

[8] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware design in Haskell. In *International Conference on Functional Programming* (Baltimore, July 1998). (2)

[9] Bloom, S. L., and Ésik, Z. Fixed-point operations on ccc's. Part I. *Theoretical Computer Science 155*, 1 (1996), 1–38. (66, 140)

[10] Bloom, S. L., and Ésik, Z. The equational logic of fixed points. *Theoretical Computer Science 179*, 1–2 (1997), 1–60. (66, 140)

[11] Carey, M. J., Chamberlin, D. D., Narayanan, S., Vance, B., Doole, D., Rielau, S., Swagerman, R., and Mattos, N. M. O-O, what have they done to DB2? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (1999), M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds., Morgan Kaufmann, pp. 542–553. (137)

[12] Claessen, K. *Embedded languages for describing and verifying hardware.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2001. (2, 4, 136)

[13] Claessen, K., and Ljunglöf, P. Typed logical variables in Haskell. In *Haskell Workshop 2000* (2000). (132)

[14] Claessen, K., and Sands, D. Observable sharing for functional circuit description. In *Asian Computing Science Conference* (1999), pp. 62–73. (136)

[15] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms*, second ed. The MIT Press, Cambridge, MA, 2001. (125)

[16] de Moor, O. An exercise in polytypic program derivation: *repmin.* Unpublished manuscript. URL: `web.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`, 1996. (122)

[17] Erkök, L., and Launchbury, J. A recursive do for Haskell: Design and implementation. Tech. Rep. CSE-00-014, Oregon Graduate Institute School of Science and Engineering, Department of CSE, OHSU, August 2000. (88, 95)

[18] Erkök, L., and Launchbury, J. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00* (September 2000), ACM Press, pp. 174–185. (6, 125, 135)

[19] Erkök, L., and Launchbury, J. A recursive do for Haskell. In *Haskell Workshop'02, Pittsburgh, Pennsylvania, USA* (Oct. 2002), ACM Press, pp. 29–37. (84)

[20] Erkök, L., Launchbury, J., and Moran, A. Semantics of *fixIO.* In *Fixed Points in Computer Science Workshop, FICS'01* (September 2001). (90, 98, 121)

[21] Erkök, L., Launchbury, J., and Moran, A. Semantics of value recursion for monadic input/output. *Journal of Theoretical Informatics and Applications 36*, 2 (2002), 155–180. (98)

[22] Espinosa, D. *Semantic Lego.* PhD thesis, Columbia University, 1995. (33, 58)

[23] Felleisen, M., and Hieb, R. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103*, 2 (1992), 235–271. (104)

[24] Filinski, A. Representing monads. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon* (New York, NY, 1994), pp. 446–457. (58)

[25] FRIEDMAN, D., AND SABRY, A. Recursion is a Computational Effect. Tech. Rep. TR546, Computer Science Department, Indiana University, Dec. 2000. (135)

[26] GHC web page. URL: `www.haskell.org/ghc`. (96)

[27] GORDON, A. D. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, Sept. 1994. (99, 112, 121)

[28] HALLGREN, T., AND CARLSSON, M. Programming with fudgets. In *Advanced Functional Programming* (1995), vol. 925 of *Lecture Notes in Computer Science*, Springer. (51)

[29] HALLGREN, T., AND CARLSSON, M. *Fudgets – Purely Functional Processes with applications to Graphical User Interfaces.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1998. (51, 136)

[30] HANKIN, C. *Lambda Calculi: A Guide for Computer Scientists*, vol. 3 of *Graduate Texts in Computer Science.* Clarendon Press, Oxford, 1994. (10)

[31] HARPER, R., DUBA, B. F., AND MACQUEEN, D. B. Typing first-class continuations in ML. *Journal of Functional Programming 3*, 4 (October 1993), 465–484. Also in ACM POPL 91, pp. 163–173. (58, 62)

[32] HASEGAWA, M. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *Typed Lambda Calculus and Applications* (1997), pp. 196–213. (66, 71, 134)

[33] HASEGAWA, M. *Models of Sharing Graphs, A categorical semantics of let and letrec.* Distinguished Dissertations in Computer Science. Springer Verlag, 1999. (66, 67, 68, 69, 70, 78, 134, 141)

[34] HUGHES, J. Global variables in Haskell. Draft paper. URL: `www.cs.chalmers.se/~rjmh/Globals.ps`. (89)

[35] HUGHES, J. Why functional programming matters. *Computer Journal 32*, 2 (1989), 98–107. (10)

[36] HUGHES, J. Generalising monads to arrows. *Science of Computer Programming 37*, 1-3 (May 2000), 67–111. (79, 135)

[37] Hugs (Haskell Users Gofer System) web page. URL: `www.haskell.org/hugs`. (96)

[38] HUTTON, G., AND GIBBONS, J. The generic approximation lemma. *Information Processing Letters 79*, 4 (2001), 197–201. (50, 145)

[39] JEFFREY, A. Premonoidal categories and a graphical view of programs. Unpublished manuscript. URL: `fpl.cs.depaul.edu/ajeffrey/premon/paper.html`, 1997. (79)

[40] JONES, M. P. First-class polymorphism with type inference. In *Proceedings of the Twenty Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)* (1997). (88)

[41] JONES, M. P. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop* (1999). (95, 101)

[42] JONES, M. P., AND DUPONCHEEL, L. Composing monads. Tech. Rep. YALEU/DCS/RR-1004, Department of Computer Science, Yale University, Dec. 1993. (33, 49)

[43] JOYAL, A., STREET, R. H., AND VERITY, D. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society 119*, 3 (1996), 447–468. (66, 68, 69, 70)

[44] KELSEY, R., CLINGER, W., AND REES, J. (Editors.) Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998), 26–76. (58, 63)

[45] KING, D. J., AND LAUNCHBURY, J. Structuring depth-first search algorithms in Haskell. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), pp. 344–354. (130)

[46] KING, D. J., AND WADLER, P. Combining monads. In *Glasgow Workshop on Functional Programming* (Ayr, July 1992), J. Launchbury and P. M. Sansom, Eds., Springer Verlag. (28)

[47] LAUNCHBURY, J. Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92* (1993), pp. 46–56. (2)

[48] LAUNCHBURY, J. A natural semantics for lazy evaluation. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina* (1993), pp. 144–154. (99, 108, 109)

[49] LAUNCHBURY, J., LEWIS, J., AND COOK, B. On embedding a microarchitectural design language within Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99)* (1999), pp. 60–69. (2, 4, 85)

[50] LAUNCHBURY, J., AND PATERSON, R. Parametricity and unboxing with unpointed types. In *European Symposium of Programming* (Apr. 1996), vol. 1058 of *Lecture Notes in Computer Science*, Springer, pp. 204–218. (10, 20)

[51] LAUNCHBURY, J., AND PEYTON JONES, S. L. Lazy functional state threads. *ACM SIGPLAN Notices 29*, 6 (June 1994), 24–35. (42)

[52] LAUNCHBURY, J., AND PEYTON JONES, S. L. State in Haskell. *Lisp and Symbolic Computation 8*, 4 (Dec. 1995), 293–341. (42, 121, 128)

[53] LIANG, S. *Modular Monadic Semantics and Compilation.* PhD thesis, Yale University, 1998. (1, 31, 33, 53, 54, 55, 61)

[54] LIANG, S., HUDAK, P., AND JONES, M. P. Monad transformers and modular interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22-25, 1995* (1995), ACM, Ed., ACM Press, pp. 333–343. (33)

[55] MACLANE, S. *Categories for the Working Mathematician*, second ed., vol. 5 of *Graduate Texts in Mathematics.* Springer Verlag, 1997. (10, 28, 68, 74, 140)

[56] MARLOW, S., PEYTON JONES, S. L., MORAN, A., AND REPPY, J. Asynchronous exceptions in Haskell. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)* (Snowbird, Utah, June 20–22 2001). (121)

[57] MASON, I. A., AND TALCOTT, C. L. Equivalence in functional languages with effects. *Journal of Functional Programming 1*, 3 (1991), 287–327. (112)

[58] MATTHEWS, J. *Algebraic Specification and Verification of Processor Microarchitectures.* PhD thesis, Oregon Graduate Institute of Science and Technology, Portland, Oregon, 2000. (2)

[59] MATTHEWS, J., COOK, B., AND LAUNCHBURY, J. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages* (1998), IEEE Computer Society Press, pp. 90–101. (2)

[60] MEIJER, E., FOKKINGA, M., AND PATERSON, R. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, J. Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1991, pp. 124–144. (20)

[61] MILNER, R. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, May 1999. (99, 121)

[62] MOGGI, E. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990. (1)

[63] MOGGI, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991). (1, 8, 10, 67)

[64] MOSSES, P. D. Semantics, modularity, and rewriting logic. In *Electronic Notes in Theoretical Computer Science* (2000), C. Kirchner and H. Kirchner, Eds., vol. 15, Elsevier Science Publishers. (1)

[65] NORDLANDER, J. *Reactive Objects and Functional Programming.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1999. (96, 130, 136)

[66] Paterson, R. A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP'01, Florence, Italy* (September 2001), ACM Press, pp. 229–240. (66, 79, 80, 83, 96, 135)

[67] Peyton Jones, S. L. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction* (2001), T. Hoare, M. Broy, and R. Steinbruggen, Eds., IOS Press, pp. 47–96. (98, 99, 108, 121)

[68] Peyton Jones, S. L., and Hughes, J. (Editors.) Report on the programming language Haskell 98, a non-strict purely-functional programming language. URL: `www.haskell.org/onlinereport`, Feb. 1999. (2, 10, 14, 20, 30, 86, 87, 89, 95, 98, 101, 101)

[69] Peyton Jones, S. L., and Wadler, P. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina* (1993), pp. 71–84. (2, 98)

[70] Pierce, B. C. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, MA, 1991. (141)

[71] Pitts, A. M. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science 10* (2000), 321–359. (112, 121)

[72] Power, J., and Robinson, E. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science 7*, 5 (1997), 453–468. (81)

[73] Power, J., and Thielecke, H. Closed Freyd- and κ-categories. In *Automata, Languages and Programming* (1999), pp. 625–634. (79, 82)

[74] Recursive monadic bindings web page. URL: `www.cse.ogi.edu/PacSoft/projects/rmb`. (96)

[75] Reynolds, J. C. Types, abstraction, and parametric polymorphism. In *Information Processing'83*, R. Mason, Ed. North-Holland, Amsterdam, 1983, pp. 513–523. (10, 20)

[76] Reynolds, J. C. *Theories of Programming Languages.* Cambridge University Press, 1998. (1, 10, 12, 140)

[77] Schmidt, D. A. *Denotational Semantics.* Allyn and Bacon, Boston, 1986. (1, 10, 32)

[78] Sestoft, P. Deriving a lazy abstract machine. *Journal of Functional Programming 7*, 3 (1997), 231-264. (109)

[79] Simpson, A., and Plotkin, G. Complete axioms for categorical fixed-point operators. In *Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science* (2000), pp. 30–41. (20, 66, 141)

[80] SIMPSON, A. K. A characterisation of the least-fixed-point operator by dinaturality. *Theoretical Computer Science 118*, 2 (1993), 301–314. (140)

[81] SMYTH, M., AND PLOTKIN, G. The category-theoretic solution to recursive domain equations, 1982. (140)

[82] STOY, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977. (1, 10, 20)

[83] TENNENT, R. D. *Semantics of Programming Languages.* Prentice Hall, New York, 1991. (1, 10, 12)

[84] THIELECKE, H. Using a continuation twice and its implications for the expressive power of `call/cc`. *Higher-Order and Symbolic Computation 12*, 1 (1999), 47–74. (58, 62)

[85] THIEMANN, P. WASH/CGI: Server-side web scripting with sessions. compositional forms, and graphics. Unpublished manuscript. URL: `www.informatik.uni-freiburg.de/~thiemann/papers`, Mar. 2001. (127)

[86] TURBAK, F., AND WELLS, J. B. Cycle therapy: A prescription for fold and unfold on regular trees. In *Proc. 3rd Int'l Conf. Principles & Practice Declarative Programming, PPDP'01* (Sept. 2001). (137)

[87] TURNER, D. A. A new implementation technique for applicative languages. *Software Practice and Experience 9*, 1 (Jan. 1979), 31–49. (10)

[88] WADLER, P. Theorems for free! In *FPCA'89, London, England.* ACM Press, Sept. 1989, pp. 347–359. (10, 20)

[89] WADLER, P. Comprehending Monads. In *LISP'90, Nice, France.* ACM Press, 1990, pp. 61–78. (2, 31, 125)

[90] WADLER, P. Monads and composable continuations. *Lisp and Symbolic Computation 7*, 1 (Jan. 1994), 39–56. (58)

[91] WADLER, P. Monads for functional programming. In *Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds., vol. 925 of *Lecture Notes in Computer Science.* Springer Verlag, 1995. (2, 10, 42, 47)

[92] WINSKEL, G. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing series. MIT Press, Feb. 1993. (140, 141)

[93] WRIGHT, A. K. Simple imperative polymorphism. *Lisp and Symbolic Computation 8*, 4 (1995), 343–355. (95, 96)

Index

Biographical Note

Levent Erkök was born in 1972, Merzifon, Turkey. He received a B.S. degree in Computer Engineering from the Middle East Technical University in Ankara, Turkey, in 1994, and an M.S. degree in Computer Science from the University of Texas at Austin in 1998. He started his Ph.D. studies at Oregon Graduate Institute later that year. His research interests lie in the theory of programming languages, especially the functional paradigm.