TECHNICAL REPORT NO. 546

Recursion is a Computational Effect

by

Daniel P. Friedman
Amr Sabry

December 2000

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
BLOOMINGTON, INDIANA 47405-4101

Recursion is a Computational Effect

Daniel P. Friedman and Amr Sabry*

December 20, 2000

Abstract

In a recent paper, Launchbury, Lewis, and Cook observe that some Haskell applications could benefit from a combinator mfix for expressing recursion over monadic types. We investigate three possible definitions of mfix and implement them in Haskell.

Like traditional fixpoint operators, there are two approaches to the definition of mfix: an unfolding one based on mathematical semantics, and an updating one based on operational semantics. The two definitions are equivalent in pure calculi but have different behaviors when used within monads.

The unfolding version can be easily defined in Haskell if one restricts fixpoints to function types. The updating version is much more challenging to define in Haskell despite the fact that its definition is straightforward in Scheme. After studying the Scheme definition in detail, we mirror it in Haskell using the primitive `unsafePerformIO`. The resulting definition of mfix appears to work well but proves to be unsafe, in the sense that it breaks essential properties of the purely functional subset of Haskell. We conclude that the updating version of mfix should be treated as a monadic effect. This observation leads to a safe definition based on monad transformers that pinpoints and exposes the subtleties of combining recursion with other effects and puts them under the programmer's control to resolve as needed.

The conclusion is that Haskell applications that need the functionality of mfix can be written safely in any Haskell dialect that supports the multi-parameter classes necessary for defining monad transformers. No other extensions to standard Haskell are needed, although some syntactic abstractions and libraries can make the task of writing recursive monadic bindings much more convenient.

*Supported by National Science Foundation Grant number CCR-9733088. Work started at the University of Oregon.

1 Introduction

In Haskell computational effects are isolated to a monadic sublanguage [32]. The isolation is useful to preserve properties of the purely functional sublanguage: for example, the call-by-name denotational semantics of Haskell coincides with the call-by-need implementation despite the presence of effects [27]. But the isolation also implies that not all the constructs of the functional sublanguage are available in the monadic sublanguage. Most notably the monadic sublanguage lacks *recursion* over the results of monadic actions.

Recently Launchbury *et al.* [19] argue that this lack of recursion somewhat limits the language. Applications that would be naturally written as systems of mutually recursive stream computations cannot use the monadic infrastructure. They suggest the inclusion of a new monadic combinator mfix that can be used to express recursion over the results of monadic actions but delegate the topic to future research.

In this paper, we investigate three definitions of mfix and show how to implement them in Haskell.

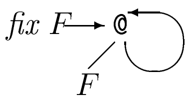
Recursion: Unfolding or Updating The literature explains recursion in one of two ways. The mathematical view [7] maintains that a recursive definition ($\mathit{fix}\ F$) is equivalent to the infinite unfolding ($F(F(F\ldots))$). The operational view (which dates back to Landin's SECD machine [17, 18] and is currently implemented in Scheme) explains a recursive definition ($\mathit{fix}\ F$) as the following sequence of steps:

1. Take a fresh cell whose address is x.

2. Use x as a spurious argument for F and evaluate the application ($F\ x$) to produce a result v.

3. Update the contents of address x with v; return v as the result.

If the functional F performs no computational effects, the two views of recursion are equivalent. This is formally investigated by Ariola *et al.* [2, 3, 4, 6] and empirically validated by Rozas's [26] compiler that can rewrite one form of definition to the other using generic analyses and optimizations. Informally one notes that if the application of F is known not to perform computational effects, then one may confuse ($F\ x$) with the value v, and perform the update to x *before* evaluating ($F\ x$). With this simplification, the operational definition of ($\mathit{fix}\ F$) reduces to evaluating the following graph:



which is just an efficient representation of the unfolding definition [31].

But like many equivalences that hold in pure calculi, the equivalence of two views of recursion does not hold in a language with computational effects, *e.g.*, Scheme or the monadic sublanguage of Haskell. We therefore investigate both approaches. In fact, we define and implement three fixpoint constructs that can be used to define recursive monadic bindings:

- mfixU: This combinator unfolds recursive definitions and is restricted to function types. Its behavior is different from the next two combinators, which are based on the updating view of recursion.

- letrecM: This construct uses unsafe primitives to realize the updating view of recursion. We show that it can be used to violate the "purity" of Haskell.

- mfixM: This last combinator is a monadic version of letrecM. It is the correct updating combinator that should be used to define recursive monadic bindings in Haskell.

Embedding in Haskell The combinator mfixU is almost straightforward to define in Haskell if one is willing to restrict the fixpoint operator to function types. Otherwise it diverges when applied in any interesting situation.

Before attempting to define the updating versions of mfix in Haskell, we study the idea in the context of Scheme. Scheme has long provided an updating version of recursion in the presence of effects and the Scheme community is aware of many of the subtleties of combining recursion and computational effects. In particular, it is folklore in the Scheme community that the update used in the definition of recursion can be exploited to define full-fledged reference cells in the presence of call/cc [8]. This idea that computational effects can duplicate the update used in defining recursion provides the insight needed to show that letrecM can be used to break fundamental properties of the purely functional subset of Haskell. Intuitively the updating step in the definition of recursion may be duplicated by operations in the list monad and performed several times in an unspecified order.

These problems suggest that recursion is itself an effect that should be confined to the monadic sublanguage. Thus, we define a combinator mfixM that exposes recursion as an explicit monadic effect. The intuition is to use a state monad to manipulate the location needed to implement the recursion. But since we are taking the fixpoint of a functional which itself performs computational effects in some monad, we now have *two* monads to take care of. Fortunately, it is always possible to combine these two monads into one using monad transformers [22].

In many situations, the combination of a monad m and the state monad used for recursion is straightforward. But in general, the combination may require a non-standard operation to *lift* computations from the monad m to the combined monad. In the continuation monad, it is possible to lift call/cc in a way that either makes the updates that happen after the continuation is captured visible when the continuation is invoked or not. In the list monad (which models non-deterministic computations), it is possible to make the updates that happen in one branch of the non-deterministic computation visible in other branches or not.

By using monad transformers, the subtleties of combining recursion and other computational effects become better-understood, explicit, and under the programmer's control. In particular, it is no longer possible for the recursion effects to happen in unspecified orders, since they are confined to the monadic sublanguage. Moreover, programmers can customize how recursion interacts with other monadic actions.

Recursion and Effects Although the implementation of recursion in the presence of arbitrary computational effects has long been known, the semantics of such a combination has long been poorly understood.

Indeed most languages impose restrictions on the combination of recursion and computational effects. In SML the right-hand side of a recursive declaration must be syntactically restricted to a function, which guarantees that its evaluation will not cause any computational effects. In Objective Caml, the guarantee is achieved with a slightly weaker restriction: the language also allows recursive declarations of constructor applications. In Haskell recursion is generally not available for monadic types, except for some well-behaved monads like the state monad.

In contrast to most languages, Scheme allows arbitrary expressions in recursive declarations. The only attempt we know of to mathematically study the semantics of letrec in Scheme is an unpublished note by Duba and Felleisen (1991) but even that study is restricted to the pure fragment of Scheme.

By recasting both recursion and effects into the world of monads, this work reduces the problem of understanding the combination of recursion and computational effects to the problem of combining monads. The latter problem is rather well-understood mathematically and has already been studied extensively [10, 14, 22, 23, 30].

Outline The next section reviews the use of monads in Haskell and motivates the need for mfix. Section 3 investigates the unfolding definition mfixU, and provides an example of its use. In Section 4, the updating definition of recursion in Scheme is analyzed. Section 5 mirrors the Scheme definition in Haskell using unsafe primitives. Section 6 gives the monad-transformer-based definition of mfixM. Section 7 provides three complete examples of the use of mfixM. Section 8 discusses related work and Section 9 concludes.

2 Monadic Effects in Haskell

We begin by reviewing the existing support of monads in Haskell and then motivate the need for something like mfix.

2.1 Definition

Monads can be characterized in several equivalent ways. In Haskell, they are defined using a type constructor and two functions `return` and `>>=`. These functions mediate between the worlds of values (which have no effects) and computations (which may have computational effects): `return` coerces a value to a trivial computation and `>>=` glues two computations sequentially. In Haskell syntax, the type constructor and its associated operations are collected in a type class as follows:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Individual monads (*e.g.*, input/output, state, continuation, etc.) are *instances* of the `Monad` type class. We discuss definitions for three of the monads that we use in the paper.

2.2 The Continuation Monad

The continuation monad can be defined as follows:

```
data Cont ans a = CPS ((a -> ans) -> ans)
unCont (CPS ecps) = ecps

instance Monad (Cont ans) where
  return e = CPS (\k -> k e)
  (CPS e1) >>= e2 =
      CPS (\k -> e1 (\v -> unCont (e2 v) k))
```

The type of monadic computations is parameterized by the type `ans` of answers.

To use the continuation monad in interesting ways, we also define functions that manipulate the continuation in non-standard ways:

```
abort :: ans -> Cont ans a
abort e = CPS (\_ -> e)

callcc :: ((a -> Cont ans b) -> Cont ans a) ->
          Cont ans a
callcc f =
  CPS (\k -> unCont (f (\a -> CPS (\_ -> k a))) k)

prompt :: Cont ans ans -> ans
prompt (CPS ecps) = ecps id
```

As a simple example, the following expression evaluates to 10:

```
prompt (return 5 >>= \i ->
        abort (i + 5) >>= \_ ->
        return 42)
```

Since the `>>=` notation gets quickly cumbersome, Haskell provides the `do` form as syntactic sugar. Using the `do` notation we can instead write:

```
prompt (do i <- return 5
           abort (i + 5)
           return 42)
```

2.3 The List Monad

The list monad supplied by the Haskell standard library is defined as follows:

```
instance Monad [] where
  return e = [e]
  e1 >>= e2 = concat (map e2 e1)
```

The monad models non-deterministic computations, which may return zero or more results. All possible answers to a computation are propagated in a list data structure that serves as the type of monadic computations. What makes the list monad interesting from our perspective is that it relies on a pun between lists used as data structures and lists used as monadic computations. This pun will prove problematical in the presence of updates due to recursion.

2.4 The State Monad

If we were to implement the state monad in Haskell, it would have the following definition:

```
data ST s a = ST (s -> (s,a))
unST (ST f) = f

instance Monad (ST s) where
  return e = ST (\s -> (s,e))
  (ST e1) >>= e2 =
    ST (\s0 -> let (s1,v1) = e1 s0
               in unST (e2 v1) s1)
```

The type of monadic computations is a function mapping an input store to an output store and a value. The monad arranges for the store to be passed around, which is rather inefficient.

Instead of this inefficient implementation, most Haskell systems provide a built-in state monad. The built-in state monad hides the type `s` of the store, uses a global store, and implements the associated operations using destructive updates [20]. In Hugs, the state monad operations for creating, reading, and updating reference cells are called `newSTRef`, `readSTRef`, and `writeSTRef`.

The built-in state monad in Hugs provides two additional constructs: fixST and runST. The first construct is a fixpoint operator that specializes mfix to the state monad. To explain its semantics, we add it to our functional implementation of the state monad:

```
fixST :: (a -> ST s a) -> ST s a
fixST f =
  ST (\s0 -> let (s1,v) = unST (f v) s0
             in (s1,v))
```

The combinator fixST propagates the store as usual: the function `f` takes the incoming store, and the resulting store is returned. The recursion is only performed on the value part of the result `v`, which is passed back into `f`.

The second construct provided by Hugs is runST, which *encapsulates* a computation. To explain its semantics, we also add it to our functional implementation of the state monad. Assume that `emptyStore` is defined appropriately, then the semantics of runST is:

```
runST :: ST s a -> a
runST (ST f) = let (_,v) = f emptyStore
               in v
```

Intuitively runST takes a computation, runs it in an initial empty store, ignores the resulting store, and returns a value of type `a`. Close examination of runST reveals that its use might introduce *dangling* pointers. For example `v` could be a function whose body mentions references created during the computation; by returning the value and ignoring the store, these references become dangling. To address this problem, the type system is augmented to track the lifetimes of references and reject the encapsulation of terms which may import or export references.

The detailed analysis of encapsulation and its justification [20, 21, 24, 29] are not necessary for our purposes, but we must be aware of the augmented type system. In that augmented type system, computations in the built-in state monad have type `(ST rho a)` where `rho` is a type variable used for enforcing the encapsulation of references. The type of the store is still abstract and the associated operations are still implemented using destructive updates. The only change from the previous description is that `rho` infects all operations. Thus, a cell holding values of type `a` has type `(STRef rho a)`.

2.5 Recursion in Monads

One of the strengths of Haskell is that it naturally supports modeling of deterministic networks of processes. Such networks are usually specified as systems of recursive equations over streams [15], a task for which Haskell is ideal. More generally, one might want to specify the networks as systems of recursive equations over a monadic type. For example, by specifying hardware circuits as systems of recursive equations over an abstract monad of circuits, it would be possible for different implementations of the abstract monad to be used to generate net-lists for low-level tools to manipulate, or to generate logical formulae for input to a theorem prover, or simply to execute the code for simulation or testing [19]. Unfortunately, Haskell does not have constructs for defining recursive equations over a monadic type.

To remedy this situation, Launchbury *et al.* [19] propose that Haskell be extended with a special fixpoint operator for monadic types:

```
mfix :: Monad m => (a -> m a) -> m a
```

As mentioned in Section 2.4, the state monad already has a specialized version of mfix called fixST. Indeed for many monads, it is not too hard to define specialized versions of mfix. The challenge is to find a generic definition of mfix that works for all monads.

3 Unfolding Recursion

We explore a definition of mfix based on unfolding. This definition is called mfixU.

3.1 Types

The usual equation for defining fixpoints in a call-by-name (need) language like Haskell is:

```
fix e = e (fix e)
```

Adapting this equation directly for mfixU does not make sense from a typing perspective. Recall that the type of mfixU is:

```
mfixU :: Monad m => (a -> m a) -> m a
```

An obvious way to proceed is to rewrite the fixpoint equation using the monadic combinators instead of function application:

```
mfixU e = mfixU e >>= e
```

or using the do notation:

```
mfixU e = do v <- mfixU e                                  (*)
             e v
```

This at least makes sense as far as the types are concerned, but is completely useless, as it diverges in any interesting situation. This follows, since the combinator >>= is usually *strict* in its left argument.

3.2 Delaying Evaluation

All is not lost, however. The problem with equation `(*)` is reminiscent of the problem of defining the Y-combinator in a call-by-value language. The remedy is to explicitly delay the argument with an η-expansion [25]. We employ the same approach here.

Instead of computing fixpoints of arbitrary types `a`, we only compute fixpoints at types `(a -> m b)` for which the η-expansion makes sense:

```
type Fun a m b = a -> m b

mfixU :: Monad m =>
           (Fun a m b -> m (Fun a m b)) ->
           m (Fun a m b)
mfixU e = do v <- return (\a -> do v <- mfixU e
                                   v a)
             e v
```

The unfolding of the recursion `(mfixU e)` is delayed using a function `(\a->...a)` to avoid the non-termination problem at the expense of restricting fixpoints to function types. Otherwise this definition is identical to the one labeled `(*)`.

3.3 Example

We illustrate how mfixU can be used to define recursive functions in the presence of effects. Our example is a variant of the familiar factorial function that keeps track of the number of recursive calls performed:

```
type FactT rho = Int -> ST rho (Int,Int)

factF :: FactT rho -> ST rho (FactT rho)
factF fact =
  do b <- newSTRef 0
     return
       (\n -> if n == 0
              then do nc <- readSTRef b
                      return (1,nc)
              else do (pr,nc) <- fact (n-1)
                      writeSTRef b (nc+1)
                      return (n*pr, nc+1))

factST :: ST rho (FactT rho)
factST = mfixU factF
```

The type of the factorial function is `(FactT rho)`: it is a function that takes an integer and returns a computation that uses the store and evaluates to a pair of integers: the result of the mathematical factorial and the number of calls. The factorial function is defined as the fixpoint of the functional `factF`. The functional performs a computational effect by allocating a reference cell `b` and every call to factorial updates that reference cell.

Since our variant of factorial performs effects, it can only be called from within a monadic expression. For example, one might write:

```
runST (do fact <- factST
          v1 <- fact 5
          v2 <- fact 5
          v3 <- fact 5
          return (v1,v2,v3))
```

which calls factorial three times and collects the results in a tuple. This expression evaluates to:

```
((120,5),(120,5),(120,5))
```

which clearly shows the unfolding nature of `mfixU`: each of three calls to `(fact 5)` allocates a new reference cell.

The example has another reasonable behavior that might even be more desirable in some situations: allocate the reference cell *once* when `factST` is first computed. This behavior is achievable in Scheme using the updating version of recursion. For example, the expression:

```
(letrec
  ((fact
     (let ((b (box 0)))
       (lambda (n)
         (if (= n 0)
             (let ((nc (unbox b)))
               (values 1 nc))
             (call-with-values
               (lambda () (fact (- n 1)))
               (lambda (pr nc)
                 (set-box! b (+ nc 1))
                 (values (* n pr) (+ nc 1)))))))))
  (let ((callFact
         (lambda (n)
           (call-with-values
             (lambda () (fact n))
             (lambda (vr vc) (cons vr vc))))))
```

```
    (let* ((v1 (callFact 5))
           (v2 (callFact 5))
           (v3 (callFact 5)))
      (list v1 v2 v3))))
```

evaluates to:

```
((120 . 5) (120 . 10) (120 . 15))
```

Interestingly, replacing mfixU by fixST in the Haskell code also produces the answer:

```
((120,5),(120,10),(120,15))
```

which shows that the behavior of fixST is consistent with the updating version of recursion.

4 Scheme

We now investigate the updating versions of mfix. But before embarking on a definition within Haskell, and to have some concrete starting point, we look at the implementation of recursion in Scheme. The analysis explains some of the subtleties in the informal definition of Landin [17, 18].

4.1 Definition

The Scheme report [1] allows recursive definitions of the form:

```
(letrec ((x1 e1) ... (xn en)) e)
```

where there are no syntactic restrictions on the expressions `e1 ... en`. In other words, the evaluation of the right-hand sides may perform any computational effects.

For simplicity we focus our attention on the case where there is only one right-hand side and the body is just `x`. The semantics of such a use of letrec is made precise by the following expansion [1]:

```
(letrec ((x e)) x)
==>
(let ((x (void)))
  (let ((v e))
    (begin (set! x v) v)))
```

We use this definition as a more accurate specification of the updating definition of recursion than the informal specification given by Landin [17, 18].

4.2 Assignment Conversion

As the denotational semantics of Scheme shows, the language uses *implicit* locations: every variable binding implicitly allocates a location, and every use of a variable implicitly dereferences the associated location [1]. The use of locations can be made explicit using a translation, called *assignment conversion* [16], which is a standard part of many Scheme compilers. It is typically explained as follows:

```
(lambda (x)
  ... x ... (set! x v) ...)
==>
(lambda (x)
  (let ((y (box x)))
    ... (unbox y) ... (set-box! y v) ...))
```

The translation introduces reference cells to hold the values of assigned variables. It is important that *every* occurrence of `x` gets replaced by the expression `(unbox y)`: evaluating the expression `(unbox y)` once and sharing its value is incorrect. The translation simplifies the later phases of the compiler because values may now be freely substituted for variables without checking whether the variables are assigned.

Applying this transformation to the definition of letrec, we get:

```
(letrec ((x e)) x)
==>
(let ((x (box (void))))
  (let ((v e*))
    (begin (set-box! x v) v)))
```

where we have no direct way of expressing the translation of `e` without access to its parse tree. The notation `e*` stands for the result of substituting every free occurrence of `x` by `(unbox x)` and every occurrence of an expression of the form `(set! x e')` by `(set-box! x e')`. Attempts to express the first part of the substitution by:

```
((lambda (x) e) (unbox x))
```

are incorrect since under the call-by-value parameter-passing mechanism the value of `x` would be needed too soon. Even if we assumed a call-by-need parameter-passing mechanism like in our eventual target language Haskell, the use of the function application would still be wrong: the first time the value of `x` is encountered, the expression `(unbox x)` would be evaluated and all subsequent uses of `x` would share that value.

Of course the presence of substitution is not a problem if the transformation is done within the compiler where one has access to the parse tree of `e`. The transformation just cannot be expressed as a source-to-source translation.

4.3 Special Form or Combinator

Instead of the letrec special form, it is tempting to use a combinator `schemeFix` to make the correspondence with mfix more direct. One might attempt the following natural definition:

```
(schemeFix f)
==>
(letrec ((x (f x))) x)
```

but this is incorrect since after assignment conversion, it produces:

```
(let ((x (box (void))))
  (let ((v (f* (unbox x))))
    (begin (set-box! x v) v)))
```

where it is clear that the location `x` is dereferenced prematurely.

Another possibility for a combinator version proposed by Honsell *et al.* [13] and described as "essentially equivalent to the one suggested by Landin" is:

```
exception Undefined

fun updatingFix f =
  let val x = ref (fn _ => raise Undefined)
      val _ = (x := (fn a => f (!x) a))
  in !x
  end
```

The definition is written in ML. The type of `updatingFix` is:

```
(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

which shows that the definition of `updatingFix` restricts fixpoints to function types. More importantly by using an η-expansion around the application of the functional, the computational effects of `f` end up being duplicated on every recursive call. Rewriting the factorial example of Section 3.3 in ML using `updatingFix` produces:

```
((120,5),(120,5),(120,5))
```

which shows that `updatingFix` implements the unfolding semantics like mfixU.

To avoid the problems with the combinator versions, we focus on the special form letrec and its Haskell counterpart letrecM.

4.4 Recursion and Continuations

Continuations are the most powerful computational effect: not only can they represent other computational effects [11], but any equation that is invalidated by any computational effect is also invalidated within the continuation monad [28]. Much insight about the interaction of recursion and computational effects can be gained by studying how letrec interacts with call/cc.

We begin by examining the situation abstractly. Consider the following expression:

```
(letrec ((x (call/cc (lambda (k) e)))) x)
```

This expression is equivalent to:

```
(let ((x (void)))
  (let ((v (call/cc (lambda (k) e))))
    (begin (set! x v) v)))
```

which in turn is equivalent to:

```
(let ((x (void)))
  (call/cc (lambda (k')
    (let ((v ((lambda (k) e)
               (lambda (z)
                 (k' (begin (set! x z) z))))))
      (begin (set! x v) v)))))
```

The first equivalence follows from the definition of letrec. The second equivalence is justified by the following axiom describing the behavior of call/cc [28]:

$$E[\mathsf{call/cc}\ \lambda k.e] = \mathsf{call/cc}\ \lambda k'.E[(\lambda k.e)\ (\lambda z.k' E[z])]$$

In the axiom, E can be any evaluation context (even the empty one) which intuitively represents part of the continuation. The call/cc in the left-hand side of the equation captures that part of the continuation and another call/cc is used to capture the rest of the continuation and bind it to k'. The continuation bound to k is a procedure that when invoked aborts its calling context, but re-installs the part of the continuation represented by E as well as the rest of the continuation represented by k'.

A close examination of the last Scheme code fragment reveals an important point: the assignment used to tie the recursive knot has been exported to the user code as part of the continuation. This suggests that judicious manipulation of the continuation may cause the update used to tie the recursive knot to happen more than once. It is part of folklore in the Scheme community that this can be exploited to simulate reference cells.

The following code illustrates the idea:

```
(define boxi
 (lambda (v)
  (letrec
    ((box_def
     (call/cc (lambda (dk)
      (list
        (lambda (x) x)
        v
        (letrec
          ((ms
           (lambda (msg)
             (case msg
               [read box_def]
               [write (lambda (nv)
                        (call/cc (lambda (rk)
                          (dk (list rk nv ms)))))]
             ))))
          ms))))))
   ((car box_def) box_def))))

(define unboxi
  (lambda (b)
    (cadr ((caddr b) 'read))))

(define setboxi
  (lambda (b nv)
    (begin (((caddr b) 'write) nv) (void))))
```

In the code, a reference cell is represented as a list of three elements: the first element is a procedure that forwards the reference to clients, the second element contains the current value of the reference, and the third provides two "methods" to read the contents of the reference and to update it with a new value. There are two uses of call/cc: the first one captures the update that is used in the definition of letrec in order to implement updates to the reference. The second call/cc arranges for the new updated reference cell to be returned to the caller of `setboxi`.

To see the code in action, consider the following test expression:

```
(define test
  (lambda (n)
    (let* ((x (boxi n))
           (f (lambda () (unboxi x)))
           (g (lambda ()
                (setboxi x
```

```
                (add1 (unboxi x))))))
      (begin
        (setboxi x (* 2 (unboxi x)))
        (g)
        (f)))))
```

The evaluation of `(test 6)` produces 13, which is the expected result from a correct implementation of references. What is important for our purposes is that continuation effects interfered with the definition of recursion and caused the update used to define the recursion to happen more than once. This idea will be exploited again when we move to Haskell.

5 A Haskell Implementation

The study of recursion in Scheme resulted in the following two facts:

- If the use of locations becomes explicit and we assume that the right-hand side does not include assignments to the variable being defined, the definition of letrec is:

  ```
  (letrec ((x e)) x)
  ==>
  (let ((x (box (void))))
    (let ((v e[x:=(unbox x)]))
      (begin (set-box! x v) v)))
  ```

  The expression `e[x:=(unbox x)]` denotes the substitution of every free occurrrence of `x` by `(unbox x)`.

- Computational effects may interfere with the above definition of recursion in unexpected ways. In particular it is possible for the assignment used to define the recursion to happen more than once under certain conditions.

We use the first fact to guide the implementation of a construct letrecM in Haskell. Then we use the second fact to show that letrecM breaks the purity of Haskell.

5.1 Types

Our goal is to define a construct:

```
letrecM x = e in e'
```

The types of the components are as expected. Assuming the type of `x` is `a`, then the expression `e` should have type `(m a)`. The expression `e'` should have type `(m b)` for some `b`.

5.2 Unsafe Primitives

The intent is to extend Haskell with a construct letrecM whose definition mirrors the definition of letrec in Scheme. Since this latter definition uses references freely, letrecM cannot be defined in the Haskell source language; the definition must be relegated to the compiler or runtime system using unsafe primitives.

In the Hugs runtime system, we can define unsafe primitives for manipulating references as follows:

```
ref :: a -> IORef a
ref e = unsafePerformIO (newIORef e)

deref :: IORef a -> a
deref e = unsafePerformIO (readIORef e)

setref :: IORef a -> a -> ()
setref e1 e2 = unsafePerformIO (writeIORef e1 e2)
```

The combinator `unsafePerformIO` enables us to use computational effects outside of the monadic sublanguage and is inherently dangerous (it can be used to define a function that casts between any two types [19]). But for the time being we do not worry about this since we only intend to use these primitives to define another primitive construct.

Again for simplicity we focus on the simple case of letrecM, which has one right-hand side, and whose body is just a variable. Rewriting the Scheme definition using monadic combinators instead of function composition, we get:

```
letrecM x = e in return x
==>
let x = ref (error "Undefined")
in do v <- e[x:=(deref x)]
      return (seq (setref x v) v)
```

The function `seq` is a recent addition to Haskell: it is a sequencing operator that is strict in its first argument. Naturally we assume that the unsafe primitives do not directly appear in `e`. Hence it is sufficient to do the first part of the substitution.

5.3 Breaking Purity

As Section 4.4 showed there is the possibility that some computational effects may duplicate the update used to define the recursion. Indeed we can use letrecM and the continuation monad to simulate reference cells much like in Scheme. (See Section 7.3 for a variant of this code in Haskell using mfixM instead of letrecM.) This is not a disturbing fact in itself since the use of reference cells is still constrained by the continuation monad. However by doing the same trick in the list monad, and exploiting the pun between lists used as data structures and lists used as monadic computations, it is possible for the use of references to happen unpredictably.

Consider the following expression:

```
letrecM xs = [2:xs, 3:xs]
in return xs
```

Using rewriting steps based on the call-by-need calculus [5], we can symbolically follow the evaluation as follows:

```
= by definition of letrecM

let x = ref (error "Undefined")
in do v <- [2 : (deref x), 3 : (deref x)]
      return (seq (setref x v) v)

= expanding do; list monad

let x = ref (error "Undefined")
in concat
    (map (\v -> [seq (setref x v) v])
         [2 : (deref x), 3 : (deref x)])

= definitions of map; concat

let x = ref (error "Undefined")
in [ let v2 = 2:(deref x)
      in seq (setref x v2) v2,
     let v3 = 3:(deref x)
      in seq (setref x v3) v3]
```

The symbolic evaluation shows that the result is a list of two expressions that each evaluate to a list of integers. The update used to implement the recursion has been mapped over the outermost list. As we show, judicious use of lazy evaluation can cause the updates to happen unpredictably.

For example, consider the following sequence of demands:

- Demand the first element of the first list of numbers: this assigns `v2` to `x`, and returns 2.
- Then demand two elements from the second list of numbers: this assigns `v3` to `x`, and returns 3 and 3.
- Then go back and demand the second element of the first list of numbers. This forces the evaluation of `(deref x)`. Since `x` has just been overwritten to `v3`, this returns a list whose first element is 3.

Had we switched the last two steps by keeping the demands from the first list of numbers contiguous, the evaluation of (deref x) would have returned a list whose first element is 2.

Using this idea, it is possible to construct a Haskell context that breaks the commutativity of addition:

```
addLR m n = m+n
addRL m n = n+m

context add =
   let [ is1 , is2 ] =
         letrecM xs = [2:xs, 3:xs]
         in return xs
       (v1a:v1bl) = take 2 is1
       [v2a,v2b] = take 2 is2
       [v1b] = take 2 v1bl
   in v1a + add v2a v1b
```

Evaluating (context addLR) gives the answer 8 but evaluating (context addRL) gives the answer 7. This is unacceptable!

5.4 Recursion is an Effect

What should be by now becoming obvious is that the common implementation of recursion using updates makes recursion a computational effect. When used freely this effect interacts in unpredictable ways with other effects. For the Haskell language, the solution is standard: confine the recursion effect into a monad. This motivates the definition of mfixM.

6 A Correct and Safe Definition

The basic idea is to define a state monad in which one can perform the update needed to implement the recursion. Then one defines a *monad transformer* that takes any monad m and combines it with the state monad to yield a new monad, in which one can perform the effects in m and define an updating version of recursion. We now turn to the details.

6.1 Monad Transformers

Our presentation of monad transformers closely follows that of Liang *et al.* [22].

A monad transformer t takes a monad m and produces a new monad (t m). For example, a state monad transformer could take a continuation monad and produce a state-and-continuation monad in which one may use both assignments and jumps.

The challenge in defining monad transformers is what to do with non-trivial computations in the underlying monad `m`. Thus when applying the state monad transformer to the continuation monad, the functions `abort`, `callcc`, and `prompt` need to be *lifted* to operate in the combined state and continuation monad.

We postpone the definition of the function `lift` for now and just state the required signature of monad transformers:

```
class (Monad m, Monad (t m)) => MonadT t m where
  lift :: m a -> t m a
```

The definition states that if both `m` and `(t m)` are known to be monads, then `t` is a monad transformer. Additionally a monad transformer must have a function `lift` of the appropriate type to map computations in `m` to computations in `(t m)`. The definition is a type class of two parameters, which is non-standard Haskell.

6.2 Combining monads

Having the definition of monad transformers in place, we define a specific monad transformer that takes an arbitrary monad `m` and combines it with a state monad for implementing recursion.

Computations in a monad combining state and `m` are described using the following datatype:

```
data STM s m a = STM (s -> m (s,a))
unSTM (STM f) = f
```

The computations are state transformers as usual: they take an input store `s` and return an output store and a value. What is different here is that the output store and value are not returned directly. Instead what is returned is a computation in monad `m` whose result is the output store and value.

To keep things modular, we introduce a class that describes the type of stores that are acceptable arguments to `STM`:

```
type Loc = Int

class Store s a | s -> a where
  alloc :: s -> a -> (Loc,s)
  look :: s -> Loc -> a
  upd :: s -> Loc -> a -> s
```

The type `(Store s a)` describes stores represented using type `s` (perhaps a list of locations and values), and whose contents are all of type `a`. The operations we require for stores are standard and have the obvious meaning. For convenience the class definition uses the *functional dependency* `s->a`. Functional dependencies were recently added to Hugs as a way to resolve the ambiguity problems of multi-parameter classes.

Putting things together, we can build a new monad from a monad `m` and a store `(Store s a)` as follows:

```
instance
  (Monad m, Store s a) =>
   Monad (STM s m) where
  return e = STM (\s -> return (s,e))
  (STM e1) >>= e2 =
    STM (\s0 -> do (s1,v1) <- e1 s0
                   unSTM (e2 v1) s1)
```

In the new monad, computations have type `(STM s m a)`. The definitions of `return` and `>>=` are more subtle than they appear because of overloading. For example the definition of `return` appears to refer to itself, but careful examination of the types reveals that in fact they are different occurrences of the same overloaded name. The `return` being defined is for the monad `(STM s m a)`; the `return` on the right is for the monad `m`.

Since `(STM s m)` is a monad for any monad `m`, it follows that `(STM s)` is a monad transformer:

```
instance
  (Monad m, Monad (STM s m)) =>
   MonadT (STM s) m where
  lift ma = STM (\s -> do v <- ma
                          return (s,v))
```

To complete the definition of the monad transformer, we provide a default definition of `lift`. As the definition shows, `lift` takes a computation `ma` in the underlying monad, an initial store, executes the computation of the underlying monad, and returns the old store unchanged irrespective of what happened during the execution of `ma`. As we will soon see, this may not be appropriate for all operations.

6.3 Adding Recursion to a Monad

Not surprisingly, the use of monadic effects to implement the update needed for recursion affects the types. Consider the definition of letrecM once again:

```
letrecM x = e in return x
==>
let y = ref (error "Undefined")
in do v <- e[x:=(deref y)]
      return (seq (setref y v) v)
```

Assume the variable x has some type a. The variable y must have type (IORef a). Using the unsafe version of deref, the expression (deref y) has type a and the substitution makes sense. If we instead use a state monad to implement deref, the type of (deref y) would be (ST ...) and the substitution would be ill-typed. To regain typing, we change the type of x to match the type of (deref y). This changes the type of the fixpoint construct. Intuitively the use of monads forces all effects to infect the types, and clients of the recursion combinator must now be aware of these effects.

Another consequence of using monadic style is that the unsafe side-effecting expression (deref y) is replaced by a state transformer: a function that takes a store and looks up the value of y in that store. This means that we can safely express the substitution as a function application since duplicating or sharing a value is equivalent in the call-by-need calculus [5].

Everything is now in place to define mfixM. Given a monad m, we first use the (STM s) monad transformer to add state to m, and then implement mfixM in the resulting monad. The class RecMonad describes this resulting monad.

```
class Monad m => RecMonad m a where
  mfixM :: (m a -> m a) -> m a
```

The type of mfixM reflects the fact that we have made the effects used to implement recursion explicit. In more detail, the functional whose fixed point we are taking has type (m a -> m a) where m is the combination of the original monad and the state monad. The domain of this functional has type (m a) because it involves a computation that reads the location holding the (initially undefined) result of the recursion. The range of this functional has type (m a) because the functional performs computational effects in the original monad and these operations have been lifted to the combined monad, and because the body of the functional may access the location holding the result of the recursion. The entire application of mfixM to a functional has type (m a) because the application of the functional performs effects and because mfixM itself performs an assignment to the location holding the result of the recursion.

Putting these ideas together, we have:

```
instance
  (Monad m, Store s a) =>
   RecMonad (STM s m) a where
  mfixM f =
    STM (\s ->
          let (x,s') = alloc s (error "Undefined")
              derefx = STM (\s ->
                              return (s, look s x))
          in do (s'',v) <- unSTM (f derefx) s'
                return (upd s'' x v, v))
```

The definition of mfixM uses the operations `next`, `alloc`, `look`, and `upd`, that come with the store type. It closely follows the operational definition of recursion but uses proper monadic style instead of unsafe primitives.

6.4 Correctness

We do not prove any formal results about mfixM. Informally we argue for the correctness of mfixM as follows. The updating definition of recursion due to Landin and implemented in Scheme is taken for granted. Starting from the Scheme definition, mfixM is derived in three steps:

1. Assignment conversion to make the use of locations explicit.
2. Translation of expressions including assignments to the state monad.
3. Combining the state monad with another monad.

Because each of these steps is well-understood, we argue that the definition of mfixM correctly implements the updating definition of recursion. The next section shows that it behaves as expected with several complex examples.

7 Examples

We now present examples that use mfixM. Since mfixM has a different type than mfixU and letrecM, it is not possible to directly reuse old examples. Instead a certain amount of work is needed. In more detail, the previous examples assumed that the recursion had no effect or had an implicit effect. Now the recursion has an explicit monadic effect and all the code that uses recursion must be converted to monadic style.

7.1 Functional Store

In the examples we refer to a type `FuncStore`, which is an instance of the class `(Store s a)` that implements stores. For completeness we include the code:

```
type FuncStore a = [(Loc,a)]

instance Store (FuncStore a) a where
  alloc s e = let i = length s + 1
              in (i, (i,e):s)
  look s i =
    case lookup i s of
      Just v -> v
      Nothing -> error "Can't look"
```

```
  upd s i e =
    case s of
      [] -> error "Can't upd"
      (j,e'):s | i==j -> (i,e) : s
      (j,e'):s -> (j,e') : upd s i e
```

7.2 Factorial

We revisit the factorial function that uses a reference cell to count the number of times it is called. We rewrite this code in monadic style to use mfixM. The relevant type definitions are:

```
type FactStore rho = FuncStore (FactT rho)
type STST rho = STM (FactStore rho) (ST rho)
data FactT rho = FactT (Int -> STST rho (Int,Int))
unFactT (FactT e) = e
```

First we define the store used for the recursion. The store is used to hold a value describing the modified factorial function. This value has type `(FactT rho)`: it takes an integer and returns a computation that may perform side-effects and return a pair of integers. The first element of the pair is the result of factorial and the second is the number of calls. The computation occurs in the monad described by `STST`, which augments the built-in `ST` monad with recursion.

Next we need to decide how the operations in the built-in state monad interact with the recursion store. The default behavior of `lift` provides no interaction and is appropriate:

```
lnewSTRef :: a -> STST rho (STRef rho a)
lnewSTRef = lift . newSTRef

lreadSTRef :: STRef rho a -> STST rho a
lreadSTRef = lift . readSTRef

lwriteSTRef :: STRef rho a -> a -> STST rho ()
lwriteSTRef r = lift . (writeSTRef r)
```

The case of `runST` is slightly complicated because of the encapsulation provided by `runST`:

```
lrunST :: (forall rho. STST rho a) -> a
lrunST sf =
  let (STM f) = sf
      (_,e) = runST (do (_,e) <- f []
                        returnST ([],e))
  in e
```

To run a computation in the combined monad, we first pass the empty recursion store to get a computation in the underlying state monad. Unfortunately this computation cannot be directly passed to `runST` since the result of that computation contains a recursion store that mentions the universally quantified type variable `rho`. To get around this, we explicitly run the state computation and throw away the recursion store.

With these preliminary steps done, the actual factorial function is straightforward to write. Syntactically it is almost identical to the version in Section 3.3, but of course the minor syntactic differences have major semantic consequences. This point is however important since it suggests that the overhead in using mfixM is not much of a burden.

```
factF factLoc =
 do b <- lnewSTRef 0
    return (FactT
              (\n ->
                 if n == 0
                 then do nc <- lreadSTRef b
                         return (1,nc)
                 else do (FactT fact) <- factLoc
                         (pr,nc) <- fact (n-1)
                         lwriteSTRef b (nc+1)
                         return (n*pr, nc+1)))

test =
  lrunST (do (FactT fact) <- mfixM factF
             v1 <- fact 5
             v2 <- fact 5
             v3 <- fact 5
             return (v1,v2,v3))
```

As expected the `test` expression evaluates to:

```
((120,5),(120,10),(120,15))
```

7.3 Continuations

Our next example simulates reference cells by using continuations to exploit the update used in defining the recursion. We proceed following the same steps of the previous section. First the relevant types are:

```
type BoxStore = FuncStore Box
type STCont = STM BoxStore (Cont Int)
```

```
data Msg = Read | Write Int
type Sender = Box -> STCont Box
type Contents = Int
type Methods = Msg -> STCont Box

data Box = Box (Sender, Contents, Methods)
```

The types express the fact that the fixpoint we are interested in computing is of type `Box`, and that we are combining recursion with the continuation monad.

Next we must decide how the operations of the continuation monad interact with the recursion store. For `prompt`, the situation is much like `runST`: we supply the initial empty recursion store, run the computation in the continuation monad, throw away the recursion store, and return the resulting value. For call/cc there are two choices: if a continuation is captured, and the recursion store is updated before the continuation is invoked, should the invocation be given the original store or the updated store? The standard semantics of call/cc is that the continuation invocation takes the updated store, but the other choice is clearly "programmable" if needed. In any case the default `lift` operation is not appropriate here.

```
lprompt :: STCont Int -> Int
lprompt (STM f) =
  prompt (do (_,s) <- f []
             return s)

lcallcc :: ((a -> STCont b) -> STCont a)
           -> STCont a
lcallcc f =
  STM
    (\s0 ->
      callcc (\k ->
       unSTM (f (\a ->
                  STM (\s1 -> k(s1,a))))
       s0))
```

With these preliminary decisions, the Scheme code of Section 4.4 can be easily adapted to Haskell:

```
boxF :: Contents -> STCont Box -> STCont Box
boxF v box_def_loc =
  lcallcc (\dk ->
   return
     (Box
      (\x -> return x,
```

```
        v,
        let ms Read = box_def_loc
            ms (Write nv) =
              lcallcc (\rk ->
                dk (Box (rk, nv, ms)))
        in ms)))

boxi :: Contents -> STCont Box
boxi v =
  do box_def@(Box(send_k,_,_)) <- mfixM (boxF v)
     send_k box_def

unboxi :: Box -> STCont Int
unboxi (Box(_,_,ms)) =
  do (Box(_,v,_)) <- ms Read
     return v

setboxi :: Box -> Int -> STCont ()
setboxi (Box(_,_,ms)) nv =
  do ms (Write nv)
     return ()

test n =
  lprompt
    (do x <- boxi n
        let f () = unboxi x
            g () = do c <- unboxi x
                      setboxi x (c+1)
         in do c <- unboxi x
               setboxi x (c*2)
               g ()
               f ())
```

As expected the expression (`test 6`) evaluates to 13.

7.4 Lists

The goal here is to understand the subtleties of combining non-determinism and recursion that were mentioned in Section 5.3. We rewrite the example from that section properly using monadic style and study its evaluation. The original example was:

```
  letrecM xs = [2:xs, 3:xs]
```

```
  in return xs
```

As usual we begin by giving the types of the fixpoint and of the combined monad:

```
type ListStore = FuncStore ListT
type STList = STM ListStore []
data ListT = Base (STList ListT) | Cons Int ListT
```

The declarations state that the new monad combines recursion with the list monad and that the type of the fixpoint is `ListT`. This latter type is essentially a list of integers but it may contain references to the location holding the result of the recursion.

Next we examine how to lift the operations of the list monad. The only operation used in the example is `[]`, which takes a collection of elements and returns a non-deterministic computation whose result is one of these elements. (Again it is confusing that both the collection of the elements and the non-deterministic computation are represented using list data structures.) The lifted version should arrange for the non-deterministic computation to take the recursion store and distribute it among all possible values. The default `lift` operation achieves this:

```
llist :: [ListT] -> STList ListT
llist = lift
```

Now the original example can be rewritten as the following non-deterministic recursive computation:

```
listc :: STList ListT
listc =
  mfixM (\xs ->
          llist [Cons 2 (Base xs),
                 Cons 3 (Base xs)])
```

So far there are no surprises. However as explained in Section 5.3, once we start demanding the elements of the list, we can produce different answers. But by moving to a monadic style, the sequence of demands must be explicitly encoded and cannot happen implicitly using two versions of addition that evaluate their arguments in different orders.

To be explicit, here are two ways to extract lists of integers from the monadic computation `listc`:

```
runRecList_one :: STList ListT -> [[Int]]
runRecList_one (STM f) =
  do (rs,fp) <- f []
     return (runFP rs fp)

runRecList_two :: STList ListT -> [[Int]]
```

```
runRecList_two (STM f) =
  let svs@((s,_):_) = f []
  in map (\(_,fp) -> runFP s fp) svs

runFP :: ListStore -> ListT -> [Int]
runFP rs (Cons i fp) = i : (runFP rs fp)
runFP rs (Base (STM rlf)) =
  do (rs',fp') <- rlf rs
     runFP rs' fp'
```

The first function `runRecList_one` provides the initial empty store to the non-deterministic computation and uses the monadic combinators to distribute that store over all possible answers. This causes each possible branch of the non-deterministic computation to get its own copy of the store. No interactions happen between the various branches of the non-deterministic computation. Indeed,

```
map (take 3) (runRecList_one listc)
```

evaluates to `[[2,2,2],[3,3,3]]` as if we had distributed the fixpoint computation over the list.

In contrast, `runRecList_two` provides the empty store to the non-deterministic computation but exploits the pun between the use of lists as monadic computations and as data structures. It uses regular list operations to access the leftmost store and distributes that store to every branch of the non-deterministic computation. The result is that we get the following behavior. The expression:

```
map (take 3) (runRecList_two listc)
```

evaluates to `[[2,2,2],[3,2,2]]` where it is apparent that the second non-deterministic branch used the store supplied by the first branch. This exhibits the same behavior of interference among the branches of a non-deterministic computation that broke the purity of Haskell in Section 5.3. Here the different results only occur in explicitly different contexts, which is not a problem.

8 Related Work

The most closely related work is a concurrent investigation of mfix by Launchbury and Erkök [9]. Instead of pursuing an operational understanding of recursion based on updates like we do, they pursue an axiomatic approach. They postulate three axioms to characterize the behavior of mfix and take them as the starting point. Then they show that these axioms can be satisfied in several individual monads (called recursive monads), and that they are invariant under monad embeddings.

Although they successfully provide definitions of mfix for several individual monads, one might question whether their axioms really describe the behavior of mfix. In particular the second axiom they postulate is at odds with continuation effects. The axiom is:

```
  mfix (\x -> e >>= \y -> f x y)
= e >>= \y -> mfix (\x -> f x y)
(if x is not free in e)
```

Intuitively the axiom states that a computation `e` that does not involve the recursion variable can be lifted out of the recursive definition. A long time ago, Bawden [8] wondered in a message to the Scheme mailing list

> ... if any real compilers make this mistaken optimization?

Indeed anyone familiar with continuations would note that although `e` does not mention `x` it can perform *arbitrary* computational effects, and in particular, it is possible for `e` to capture the continuation. In this situation, the two sides of the axiom are not equal. In the left-hand side, the continuation captured by `e` is within the recursive definition and can be used to re-bind a new value to `x`. In the right-hand side, the continuation captured by `e` is outside the recursive definition and cannot interfere with it. Indeed Launchbury and Erkök do not show that the continuation monad is recursive.

Their treatment of recursion in the list monad also deserves some discussion. Our approach shows that recursion in the list monad may produce different results depending on the amount of interference among the different branches of the non-deterministic computation. Their postulated axioms are just strong enough to eliminate the interference. For example their definition of mfix in the list monad would only produce the result:

```
[[2,2,2],[3,3,3]]
```

for the example in Section 7.4.

Otherwise their definitions of mfix for the individual monads they consider appear to agree with our generic one. Formal proofs of correspondence are not immediately possible since our final mfixM has a different type. Of course specializing mfix to individual monads sometimes provides more elegant definitions though.

9 Conclusion

Motivated by a problem posed by Launchbury *et al.* [19], we have explained how in Haskell, one can combine recursion and monadic programming without any extensions to the language (other than multi-parameter classes). Our solution is to simply adapt the following operational definition of recursion to Haskell: allocate a reference cell to hold the result of the recursive declaration, evaluate the declaration, and update the cell with the result. The adaptation relies on a monad transformer that combines any monad with facilities for handling the effects of recursion.

Although this approach requires no changes to the language, it could be made more convenient and efficient with a few extensions. In particular, programming recursive definitions explicitly with mfix and its variants is not as intuitive as the μdo notation proposed by Launchbury and Erkök [9]. Also much of the code needed to implement monad transformers can be encapsulated in a library.

Other than extending the range of Haskell applications, our work finally suggests a mathematical approach to studying the poorly understood combination of recursion and computational effects. Studying recursion in combination with even as mild an effect as *sharing* is already challenging [12] and other combinations appear not to have been studied at all. Explaining recursion as a monadic effect may open the door for a more manageable approach based on combining monads. For example it may be possible to formally characterize the computational effects that do not interfere with recursion as the ones that can be lifted *naturally*.

The study of recursion and computational effects also has applications in compiling. In particular it simplifies the derivation of a call-by-need continuation-passing style transformation, and brings closer the worlds of compiling call-by-value and call-by-need languages.

Acknowledgments

We have benefited from discussions with Levent Erkök and John Launchbury.

References

[1] Abelson, H., Dybvig, R. K., Haynes, C. T., Rozas, G. J., IV, N. I. A., Friedman, D. P., Kohlbecker, E., Steele Jr., G. L., Bartley, D. H., Halstead, R., Oxley, D., Sussman, G. J., Brooks, G., Hanson, C., Pitman, K. M., and Wand, M. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11*, 1 (Aug. 1998), 7–105.

[2] Ariola, Z. M. Relating graph and term rewriting via Böhm models. *Applicable Algebra in Engineering, Communication and Computing 7*, 5 (1996), 401–426.

[3] Ariola, Z. M., and Arvind. Properties of a first-order functional language with sharing. *Theoret. Comput. Sci. 146* (1995).

[4] Ariola, Z. M., and Blom, S. Cyclic lambda calculi. In the *International Symposium on Theoretical Aspects of Computer Software (TACS)* (1997).

[5] Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. A call-by-need lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), ACM Press, New York, pp. 233–246.

[6] Ariola, Z. M., and Klop, J. W. Cyclic lambda graph rewriting. In *Proc. of the Eighth IEEE Symposium on Logic in Computer Science, Paris* (1994).

[7] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V., Amsterdam, 1984.

[8] Bawden, A. Letrec and callcc implement references. Appeared in `comp.lang.scheme`, 1988.

[9] Erkök, L., and Launchbury, J. Recursive monadic bindings. *ACM SIGPLAN Notices 35*, 9 (Sept. 2000), 174–185.

[10] Espinosa, D. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[11] Filinski, A. Representing monads. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1994), pp. 446–457.

[12] Hasegawa, M. *Models of Sharing Graphs: A Categorical Semantics of let and letrec*. PhD thesis, University of Edinburgh, 1997.

[13] Honsell, F., Mason, I. A., Smith, S., and Talcott, C. A variable typed logic of effects. *Information and Computation 119*, 1 (15 May 1995), 55–90.

[14] Jones, M. P., and Duponcheel, L. Composing monads. Tech. Rep. YALEU/DCS/RR-1004, Department of Computer Science, Yale University, Dec. 1993.

[15] Kahn, G. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress* (1974), J. L. Rosenfeld, Ed., North-Holland, pp. 471–475.

[16] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: An optimizing compiler for Scheme. In *ACM SIGPLAN Symposium on Compiler Construction* (1986), Sigplan Notices, 21, 7, pp. 219–233.

[17] Landin, P. The mechanical evaluation of expressions. *Comput. J. 6*, 4 (1964), 308–320.

[18] Landin, P. J. A λ-calculus approach. In *Advances in Programming and Non-Numerical Computation* (1966), L. Fox, Ed., Pergamon Press, pp. 97–141.

[19] Launchbury, J., Lewis, J. R., and Cook, B. On embedding a microarchitectural design language within Haskell. In the *ACM SIGPLAN International Conference on Functional Programming* (1999), ACM Press, New York, pp. 60–69.

[20] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp Symbol. Comput. 8* (1995), 193–341.

[21] Launchbury, J., and Sabry, A. Monadic state: Axiomatization and type safety. In the *ACM SIGPLAN International Conference on Functional Programming* (1997), ACM Press, New York, pp. 227–238.

[22] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), pp. 333–343.

[23] Moggi, E. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, 1990.

[24] Moggi, E., and Palumbo, F. Monadic encapsulation of effects: A revised approach. In the *Third International Workshop on Higher Order Operational Techniques in Semantics* (Oct. 1999).

[25] Reynolds, J. C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM 13*, 5 (1970), 308–319.

[26] Rozas, G. J. Taming the Y operator. In *ACM Conference on Lisp and Functional Programming* (1992), ACM Press, New York, pp. 226–234.

[27] Sabry, A. What is a purely functional language? *J. Functional Programming 8*, 1 (Jan. 1998), 1–22.

[28] Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp Symbol. Comput. 6*, 3/4 (1993), 289–360. Also in the *ACM Conference on Lisp and Functional Programming* (1992) and Tech. Rep. 92-180, Rice University.

[29] Semmelroth, M., and Sabry, A. Monadic encapsulation in ML. In the *ACM SIGPLAN International Conference on Functional Programming* (1999), ACM Press, New York, pp. 8–17.

[30] Steele, Jr., G. L. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon* (Jan. 1994), ACM, pp. 472–492.

[31] Turner, D. A. A new implementation technique for applicative languages. *Software – Practice and Experience 9* (1979), 31–49.

[32] Wadler, P. Comprehending monads. In *ACM Conference on Lisp and Functional Programming* (1990), pp. 61–78.