

Monads for Incremental Computing

Functional Pearl

Magnus Carlsson
OGI School of Science & Engineering
Oregon Health & Science University
magnus@cse.ogi.edu

Abstract

This paper presents a monadic approach to incremental computation, suitable for purely functional languages such as Haskell. A program that uses incremental computation is able to perform an incremental amount of computation to accommodate for changes in input data. Recently, Acar, Blleloch and Harper presented a small Standard ML library that supports efficient, high-level incremental computations [1]. Here, we present a monadic variant of that library, written in Haskell extended with first-class references. By using monads, not only are we able to provide a purely functional interface to the library, the types also enforce “correct usage” without having to resort to any type-system extension. We also find optimization opportunities based on standard monadic combinators.

This is an exercise in putting to work monad transformers with environments, references, and continuations.

Categories and Subject Descriptors

D.1 [Software]: Programming Techniques

General Terms

Algorithms, Design, Languages

1 Introduction

It is often a tedious task to translate a library from an imperative language into a purely functional language. Types need to mirror where different effects may happen, and here, the standard approach is to use monads [16]. The reward is that the interface of the purely functional library is more precise about what effects different operations have. It can also happen that monads lead us to an interface that is simpler, and more precisely captures how the operations can be correctly combined. The exercise described in this paper is an example of this.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 ACM 1-58113-487-8/02/0010 ...\$5.00

Our starting point is the elegant mechanism for high-level incremental computation presented in [1]. The paper comes with a 69-line Standard ML [11] library that “under the hood” maintains a dependency graph to support efficient recomputation due to incremental changes in the input. Using this library, a programmer can develop applications that support incremental computing on a high level, without having to deal with the intricacies of checking if recomputation is needed, or keeping the necessary data dependencies consistent. This is demonstrated by examples in [1] and in the following sections.

We will show how the ML library can be turned into a purely functional Haskell [13] module, using first-class references.

In the next section, we give an example of incremental programming, and present the ML library, which is our starting point. Section 2 develops a monadic interface in Haskell, based on considerations on how the interface should be used, and on the ML interface. Section 3 puts the monadic interface to a more serious test by using it for two larger examples: the incremental Quicksort, and a spreadsheet embryo. We describe the key parts of the monadic implementation in Section 4, and Section 5 concludes.

1.1 The ML Library

Using the library from [1], we can express the following incremental computation in Standard ML:

```
let val m      = mod (op=) (fn d => write(d,1))
    val mplus1 = mod (op=) (fn d =>
                            read(m,fn v =>
                                write(d,v+1)))
in change (m,2);
  propagate()
end
```

This defines `m` to be a *modifiable* integer, with the value 1. It also defines the modifiable `mplus1`, to be whatever value `m` has, plus 1. We then change the value of `m` to 2, and instruct the library to propagate that change to all dependent modifiables. The incremental library uses the comparison operation we supplied (`op=`) to decide that the new value of `m` really is different from the old. It keeps track of the fact that `mplus1` must be recomputed, since it read the value of `m`. The recomputation is done by applying `fn v => write(d,v+1)` to the modified value of `m`. As a result, `mplus1` will change to 3. This concludes our first, somewhat trivial example of an incremental computation.

The complete signature of the ML library is given in Figure 1. Note that Acar *et al.* talks about *adaptive* computation, but in what

```
signature ADAPTIVE =
sig
  type 'a mod
  type 'a dest
  type changeable
  val mod: ('a * 'a -> bool) ->
    ('a dest -> changeable) -> 'a mod
  val read: 'a mod * ('a -> changeable) -> changeable
  val write: 'a dest * 'a -> changeable
  val change: 'a mod * 'a -> unit
  val propagate: unit -> unit
  val init: unit -> unit
end
```

Figure 1. Interface for the SML adaptive library

follows, we will mainly use the more traditional term *incremental* computation.

The type `'a mod` stands for a modifiable (variable) of type `'a`. These modifiables are defined by `mod` and `write`, changed by `change` and `propagate`, and read by `read`. The types prevent a user from directly reading a modifiable: since the values of modifiables might change, it only makes sense to read them while defining values of other modifiables.¹ The vehicle here is the type `changeable`, which represents a computation that might depend on other modifiables, and thus might need to be recomputed. Changeables are expressions that read zero or more modifiables in a continuation-passing style, and finally write to the modifiable being defined or updated. It is the job of the *destination* parameter (of type `'a dest`), provided by `mod`, to keep track of which modifiable should be written. In the example we saw, the destination parameters were called `d`. There is one more operation in the ML library: `init`. This is a meta operation used to initialize the library.

1.2 Correct usage

For the library to work correctly, each changeable expression must use its destination exactly once, in a write operation (this is called *correct usage* in [1]). Unfortunately, the types of the operations are not precise enough to prevent incorrect usage. Here are two examples (we will abstract over the comparison operations in the following as `cmp`, `cmp'` etc):

```
mod cmp (fn d => write(d,1); write(d,2)): Incorrect—
the destination is used in two write operations.
```

```
mod cmp (fn d => write(d,
mod cmp' (fn d' => write(d,1)))): Incorrect—the desti-
nation d is used more than once and d' is not used in any
write operation.
```

The implementation of the ML library given in [1] does checks at runtime that catch some incorrect uses, but not all. To ensure correct usage statically, a special-purpose functional language with a modal type system is presented in [1].

Is it possible to design the interface of the library so that correct usage can be statically ensured, without any type-system extensions? In what follows, we will see how a monadic framework can guide us toward a solution.

¹Therefore, any interesting adaptive program using this interface must ultimately rely on side effects to make the values of modifiables observable.

2 Translation into Haskell

How should we approach the problem of translating the ML library into the purely functional Haskell? Here, readers might stall and say that there is not much we can do before we have looked at the *implementation* behind the interface in Figure 1. Naturally, we will do this in due time. However, it is useful to start out by taking a step back and just look at the interface, the types of its operations, and use patterns.

2.1 Use patterns for changeables

Changeables are written in a continuation-passing style (CPS) in which a number of modifiables are read, followed by a write operation to a destination. Here is an example:

```
m = mod cmp (fn d => read(m1, fn v1 =>
  let val x = f v1 in
  read(m2, fn v2 =>
    let val y = g v1 v2 in
    write(d,(x,y))
  end)
end))
```

This style makes the order of the read operations explicit, and the ML library observes these to build a dependency graph that describes the work needed to recompute `m` given changes of `m1` or `m2`. If `m1` changes, the subexpression `fn v1 => ...` needs to be recomputed, which involves calling `f` and `g`. If `m2` changes, but not `m1`, we only need to recompute `fn v2 => ...`, which only involves calling `g`.

2.2 The relation between CPS and monads

We can easily adopt the CPS for our Haskell library, but we actually benefit from using a monadic style instead. There is a close correspondence between continuations and monads [16]. In CPS, operations take a continuation, which is of some “answer” type `A`. This continuation is possibly parameterized by some value that the operations provide. Operations of type $(\alpha \rightarrow A) \rightarrow A$ correspond to a monadic operation $C\alpha$, and we can think of these types as synonyms. As an example, assuming we have the CPS operations `op1` and `op2`, we can define an adding combinator `op3` (from now on, we will use Haskell syntax):

```
op3 k = op1 (\a -> op2 (\b -> k (a + b)))
```

The combinator can be expressed in a monadic style instead, eliminating the continuation parameter:

```
op3 :: C Integer
op3 = op1 >>= \a -> op2 >>= \b -> return (a + b)
```

Here we have assumed the following types for the operations:

```
op1 :: C Integer
op2 :: C Integer
(>>=) :: C a -> (a -> C b) -> C b
return :: a -> C a
```

The last two operations are the *bind* and *unit* operations that every monad has. So far, the monadic style might seem complicated compared to CPS, where we didn’t have to use `bind` and `unit`. But the benefit is that we have eliminated the continuation parameter, thereby preventing derived operations from using non-local jumps.

For example, using CPS, nothing prevents us from defining an operation `bad` that ignores the continuation parameter and instead invokes some other continuation that happens to be in scope:

```
bad k = op1 (\a -> op2 (\b -> k' (a + b)))
```

By hiding the continuation inside the abstraction barrier of a monad, we can keep better control over how it is used. Now, this is good news! It indicates that we are approaching a solution that prevents the incorrect usage examples given in Section 1.2. A key observation is that the destination parameter, in conjunction with the write operation, plays the role of a continuation that can be hidden from the user.

2.3 Changeables as monads

Let us look at the types of the operations from Figure 1 that involve changeables. We will adapt these types to Haskell by capitalizing type constructors and using curried functions:

```
mod    :: (a -> a -> Bool) ->
         (Dest a -> Changeable) -> Mod a
read  :: Mod a -> (a -> Changeable) -> Changeable
write :: Dest a -> a -> Changeable
```

From what we have learned, we see that the `read` operation stands out as a candidate for a monadic operation that we can name `readMod`:

```
readMod :: Mod a -> C a
```

Here, `C a` is the monadic type that corresponds to the CPS type `(a -> Changeable) -> Changeable`.

What is the monadic analog to `write`? Answer: nothing at all! As we already mentioned, we want the write operation to be part of the continuation that we hide from the user. Another way of saying this is that instead of providing the write operation to the programmer, we put it into the `mod` operation itself. This makes sense, since the write operation concludes every correct changeable. Intuitively, it should be almost effortless to push it “over the edge”, effectively factoring it out. We can try to capture the correct usage pattern by providing the derived operation `correctMod` instead of `mod`:

```
correctMod :: (a -> a -> Bool) -> C a -> C (Mod a)
correctMod cmp c = mod cmp (\d -> c (\a -> write d a))
```

We will refine this in a moment; by this definition we merely try to convey the intuitive relation with the original `mod` operation. Note that we need a monadic type for `correctMod`: otherwise, the effects in its argument `c` cannot be performed. So `correctMod` executes `c`, returning a new modifiable. Note also that since `correctMod` is an operation in `C`, it is possible to create modifiables inside the creation of other modifiables. This is particularly useful for defining recursive data structures that contain modifiables, as we shall see in Section 3.1.

Now, it is time to look at the remaining operations that let us change the values of a number of modifiables, and then to propagate the changes. These are highly imperative in nature and access and manipulate the underlying dependency graph. Therefore, it makes sense to look for monadic types for these operations. Should they be operations in the `C` monad? This would mean that changeable expressions could invoke `change` and `propagate`, which could invoke other changeable expressions, possibly resulting in infinite loops.

```
newtype A a
newtype C a
newtype Mod a
instance Monad A
instance Monad C

class Monad m => NewMod m where
  newModBy :: (a -> a -> Bool) -> C a -> m (Mod a)
instance NewMod A
instance NewMod C

newMod    :: (NewMod m, Eq a) => C a -> m (Mod a)
readMod   :: Mod a -> C a

change    :: Mod a -> a -> A ()
propagate :: A ()
```

Figure 2. Interface for the Haskell adaptive library.

Let us make the design decision that the imperative `change` and `propagate` operations should be prevented inside changeables. This suggests that these operations live in another monad, which we can call `A` for Adaptive. The monadic types for `change` and `propagate` will then be:

```
change    :: Mod a -> a -> A ()
propagate :: A ()
```

How do we create modifiables to start with? The type of `correctMod` only allows for the creation of modifiables inside other modifiables! Why not allow creation of modifiables in the `A` monad too?

```
correctModA :: (a -> a -> Bool) -> C a -> A (Mod a)
```

Experienced Haskell programmers will immediately spot that this type is a candidate for an *overloaded* operation. Let us define a class `NewMod`, and assume we have `NewMod` instances for the `A` and `C` monads. The overloaded operation has type

```
NewMod m => (a -> a -> Bool) -> C a -> m (Mod a)
```

Not only does this allow us to use the same name in both `A` and `C` for creating modifiables, it also enables us to derive more operations that can be used in both monads.

While we have our overloading hat on, our attention is drawn to the comparison operation of type `a -> a -> Bool`. In cases like this, there is a tradition in Haskell libraries [12] to provide two operations, one which is conveniently overloaded in the `Eq` class, and one which gives precise control over the comparison operator. For example, we have the `nub` and `nubBy` functions from the `List` library:

```
nubBy :: (a -> a -> Bool) -> [a] -> [a]
nub   :: (Eq a) => [a] -> [a]
nub   = nubBy (==)
```

We will follow this tradition, and provide the convenient operation `newMod` with which users don’t have to specify the comparison operation. Thus, we end up with a library interface that we summarize in Figure 2.

3 Examples in Haskell

Before we throw ourselves into the implementation details behind the interface in Figure 2, let us “dry run” it against some examples.

In an example similar to `op3` in Section 2.2, we can define a monadic operation `op4`:

```
op4 = op1 >>= \a ->
      op2 >>= \b ->
        let c = a + b
        in
          op3 >>= \_ ->
            return c
```

Using the `do` notation, the `bind` operation and `lambda` is combined into a left arrow, and we get a shorter notation for operations for which we are only interested in their effects (`op3`):

```
op4 = do a <- op1
        b <- op2
        let c = a + b
        op3
        return c
```

Figure 3. Example of Haskell's `do` notation

This will reveal if we can combine the operations in a useful way, and we will let the examples suggest extensions to the interface. For our examples, we will use Haskell's *do notation* [13], which provides convenient syntactic support for monadic programming. Figure 3 explains the `do` notation briefly.

A first example immediately comes to mind: let us translate the incremental ML computation from Section 1.1. Since value declarations in ML are effectful, we put them in a `do` command sequence:

```
do m      <- newMod (return 1)
  mplus1 <- newMod (do v <- readMod m
                    return (v+1))

  change m 2
  propagate
```

Typically, Haskell programmers avoid parentheses around a potentially large expression that is the last argument of some function by using the low-precedence operator `$`. Using this style, the example becomes

```
do m      <- newMod $ return 1
  mplus1 <- newMod $ do v <- readMod m
                    return (v+1)

  change m 2
  propagate
```

It is instructive to compare this with the ML code in Section 1.1. Note how the destination parameters have disappeared, and that the changeables don't use any `write` operations. It is obvious that by eliminating the `write` operation, changeable expressions get a more declarative feel.

On the other side, we no longer hide the fact that the creation of modifiables is an effect. Finally, the continuation passing style used in the changeables has been replaced by the monadic style. Thus, monads offer the one stylistic glue that holds the Haskell example together.

After this example, we have enough confidence to try some larger examples.

```
datatype 'a list' = NIL
                  | CONS of ('a * 'a list' mod)

(* mod1 : ('a list' dest -> changeable) ->
   'a list' mod *)
fun mod1 f = mod (fn (NIL,NIL) => true
                 | _ => false) f

(* filter' : ('a -> bool) -> 'a list' mod ->
   'a list' mod *)
fun filter' f l =
  let fun filt(l,d) = read(l, fn l' =>
      case l' of
        NIL => write(d, NIL)
      | CONS(h,r) =>
          if f(h) then write(d,
              CONS(h, mod1(fn d => filt(r,d))))
          else filt(r, d))
      in mod1(fn d => filt(l, d))
      end
  end

(* qsort' : int list' mod -> int list' mod *)
fun qsort'(l) =
  let fun qs(l,rest,d) = read(l, fn l' =>
      case l' of
        NIL => write(d, rest)
      | CONS(h,r) =>
          let
            val l1 = filter' (fn x => x < h) r
            val g  = filter' (fn x => x >= h) r
            val gs = mod1(fn d => qs(g,rest,d))
          in qs(l,CONS(h,gs),d)
          end
      end
  in mod1(fn d => qs(l,NIL,d))
  end
```

Figure 4. Incremental Quicksort in ML.

```
data List' a = NIL | CONS a (Mod (List' a))
  deriving Eq

filter' :: (Eq a, NewMod m) =>
  (a -> Bool) -> Mod (List' a) ->
  m (Mod (List' a))
filter' f l = newMod (filt l)
  where
    filt l = do
      l' <- readMod l
      case l' of
        NIL -> return NIL
        CONS h r ->
          if f h then CONS h 'liftM'
            newMod (filt r)
          else filt r

qsort' :: (Ord a, NewMod m) =>
  Mod (List' a) -> m (Mod (List' a))
qsort' l = newMod (qs l NIL) where
  qs l rest = do
    l' <- readMod l
    case l' of
      NIL -> return rest
      CONS h r -> do
        l <- filter' (<h) r
        g <- filter' (>=h) r
        gs <- newMod (qs g rest)
        qs l (CONS h gs)
```

Figure 5. Incremental Quicksort in Haskell.

3.1 The incremental Quicksort

Acar *et al.* present an incremental version of Quicksort in [1], which we show in Figure 4. This version accepts as input a variant of the standard list type that allows the tail of a list to be modified (the type `'a List'`). This data structure is particularly useful for input lists to which we want to append data incrementally. Indeed, the authors show that the incremental Quicksort can insert a new element appended to its input list of length n in expected $O(\log n)$ time.

A version that uses the Haskell variant of the incremental library is shown in Figure 5. Let us go through the principal differences between the two variants.

The conservative comparison operation. ML-Quicksort uses a conservative comparison operation in constructing modifiable lists in the `modL` definition. The comparison treats all modifiable lists as different unless they are empty. Although this conservative comparison operation may trigger changeables to be recomputed more often than needed, it has the benefit of being computable in constant time.

Haskell-Quicksort instead appeals to the overloaded equality operation derived in the definition of `List'`. This equality operation in turn consults underlying equality operations on the elements of the list, and on modifiables. How do we test if two modifiables are equal? We cannot really hope for semantic equality here, since modifiables depend on the changeable computations. Moreover, semantic equality would not be a pure operation, since modifiables can be modified! Let us make the design decision that we only provide the conservative equality operation for modifiables, which is true if and only if the arguments are the *same* modifiable. This leads us to our first extension of the interface in Figure 2. We state that there is an instance declaration for `Mod a` in `Eq`, which does not depend on equality for `a`:

```
instance Eq (Mod a)
```

For Haskell programmers, this instance might come across as a weird member of the `Eq` class. Usually, `Eq` instances are not conservative; rather, they tend to lump elements together in equivalence classes. However, the most important property of an `Eq` instance is that its equality operation forms an equivalence relation, and this is indeed the case with our instance for modifiables.

The comparison operation in Haskell turns out to be less conservative than its ML counterpart. Not only does the Haskell operation identify empty lists as equal, it returns true for lists whose heads are equal and whose tails are the same modifiable. Still, it runs in constant time for lists over basic types such as `Int` and `Char`.

The changeable expressions. Again, we see that the monadic version eliminates all destination variables, continuations, and write operations. The cleanup is significant, since `filt` and `qs` no longer need destination parameters.

Monadic styles. Since `newMod` is a monadic operation (in contrast to `mod`), we cannot directly apply the `CONS` constructor to it in the recursive call in `filt`. The standard remedy is to *lift* the constructor function to a monadic function using `liftM`:

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f ma = do a <- ma
             return (f a)
```

Finally, we observe that the types of `filter'` and `qsort'` are both overloaded monadic operations in the `NewMod` class. Thus they can be used both in the outer-level `A` monad, and in changeable expressions living in the `C` monad. Note that the `newMod` operation is used inside `filt` which in turn is inside a `newMod` operation in `filter'`. This is one example in which it is necessary to create modifiables in the changeable monad `C`.

3.2 An embryonic spreadsheet

A classic example of a program that benefits from incremental computation is a *spreadsheet*. A large spreadsheet can have thousands of cells, containing numbers and formulas referring to other cells. When a user updates a cell, an efficiently implemented spreadsheet program only redraws that cell and other cells that depend on it. Let us see how the basic mechanism behind a spreadsheet program can be implemented using the incremental library.

We start by defining a datatype that captures the abstract syntax for expressions of cells.

```
data Expr v =
  Const Integer | Add (Expr v) (Expr v) | Cell v
  deriving Eq
```

With this type, we intend to express integer constants, addition of expressions, and references to *values* of other cells. What should the type of a cell's value be? Since the content of a cell might be changed by the user, it should be a modifiable:

```
type Value = Mod Integer
```

Now, we can write an evaluator for our expressions. Since expressions might refer to other cells, the value of an expression might change. Therefore, we put the evaluator in the `C` monad:

```
eval :: Expr Value -> C Integer
eval (Const i) = return i
eval (Add a b) = return (+) 'ap' eval a 'ap' eval b
eval (Cell m) = readMod m
```

Here, we have used the left-associative infix operator `ap`, which can be seen as the function application operation lifted to a monad:

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf ma = do
  f <- mf
  a <- ma
  return (f a)
```

By using `ap` and `return`, we lift the addition function to the `C` monad and apply it to the results of the recursive calls to the evaluator.

The evaluator is a straightforward example of a monadic interpreter [16]. The interesting case is when we encounter a `Cell`, where we have to use `readMod` to access the value of the cell.

We can now use the evaluator to create a *cell* containing a modifiable expression:

```

type Cell = Mod (Expr Value)

newCell :: NewMod m => m (Cell, Value)
newCell = do
  c <- newMod $ return (Const 0)
  v <- newMod $ readMod c >>= eval
  return (c,v)

```

The `newCell` operation creates and returns a new cell with an initial value of zero. But not only does it return the cell, it also returns its value. This value is calculated using `eval`, and since it is modifiable, it will track future modifications of the cell.

How can we try this out? It turns out that there is a major flaw in our monadic interface: there is no way to “run” any computations in the `A` monad. Moreover, the modifiables are rather useless as it stands, since they can only be observed while defining other modifiables. How should we allow for modifiables to be observed? For a fully-fledged, interactive spreadsheet program, changes to a cell will trigger rendering operations in the user interface. For our purposes, it will suffice if we can print strings to the screen. This suggests that we need some means of putting I/O operations in the `C` and `A` monads, and that we need a way to invoke `A` computations from the top level in a Haskell program, that is, from the `IO` monad:

```

class InIO m where
  inIO :: IO a -> m a

instance InIO A
instance InIO C

inA :: A a -> IO a

```

Now, we can expand our spreadsheet example into a complete adaptive program. Let us first define a *cell observer* that creates a cell, and also observes its value by printing to the screen:

```

newCellObserver :: NewMod m => String -> m (Cell, Value)
newCellObserver l = do
  (c,v) <- newCell
  newMod $ do
    x <- readMod v
    inIO $ putStrLn (l ++ " = " ++ show x)
  return (c,v)

```

Here, `putStrLn` is a standard operation for printing a string to the screen. When we create a cell with `newCellObserver` applied to some label, its initial value will be printed. Moreover, whenever the value of the cell changes, the new value will be printed. This can happen if we change the expression in the cell, or if the cell refers to some other cell value that changes.

Let us use `newCellObserver` in a small program to create two cells, and then change their values to observe the effects:

```

1 main = inA $ do
2   (c1,v1) <- newCellObserver "c1"
3   (c2,v2) <- newCellObserver "c2"
4   change c2 (Add (Cell v1) (Const 40))
5   propagate
6   change c1 (Const 2)
7   propagate

```

The creation of the two cell/value pairs in lines 2 and 3 result in their initial values being printed to the screen:

```

c1 = 0
c2 = 0

```

We then change the expression of `c2` to be the value of `c1` plus 40. The propagation of this change on line 5 triggers the new value of `c2` to be printed:

```
c2 = 40
```

Finally, the change of `c1`, once it’s propagated on line 6, triggers changes in the values of both cells, so two more lines are being output:

```

c1 = 2
c2 = 42

```

What happens if we introduce a circularity in the spreadsheet? In our example, this can be done by adding the following operations:

```

8   change c1 (Cell v2)
9   propagate

```

The answer is not too surprising: the library will loop in search for a fixpoint. In this case, there is no fixpoint, and the loop will be infinite.

3.3 Optimizing the `ap` combinator

Let us reconsider the definition for `ap` that we defined in conjunction with the evaluator. Suppose that the expression `mf` reads the value of a modifiable which has changed. This triggers a recomputation of `mf`, yielding a new value for `f`. The library will recompute everything that comes after the binding of `f`, in particular, `ma`. But this is a waste, since `ma` doesn’t have `f` as a free variable! We can isolate the changes in `mf` from `ma` by introducing an auxiliary modifiable:

```

ap :: C (a -> b) -> C a -> C b
ap mf ma = do
  m <- newModBy (\_ _ -> False) mf
  a <- ma
  f <- readMod m
  return (f a)

```

The changed value from `mf` has now been captured inside `m`. We carefully avoid looking at this value until *after* evaluating `ma`.

In the original definition of `ap`, changes inside `ma` did not trigger re-computation of `mf`, since `mf` was evaluated before `ma`. The optimized version is more symmetric, in that there are no dependencies in any direction between the arguments. By using the optimized version in the evaluator, we achieve a fine-grained incremental computation: a changed cell value propagates up the spine of the expression tree, but no other subtrees of the expression need to be recomputed.

It is an interesting exercise to develop the spreadsheet example into a more complete spreadsheet program, but we leave it for now.

4 Implementation

Most of the implementation behind the interface in Figure 2 closely follows the ML implementation in [1]. In this section, we will give a short recapitulation of the mechanism implemented in the library, and then present our implementation, focusing on parts that differ from the ML implementation. The complete Haskell implementation is available online at [3].

The implementation relies on the following principal data structures:

Dependency graph: Each modifiable acts as a node, and carries a list of dependent changeables (acting as edges). The edges are created dynamically: during evaluation of a changeable, an edge is added to the dependency list of each modifiable it reads.

Evaluation queue: When a modifiable is changed by the `change` operation, all its edges are moved to an evaluation queue, so that dependent modifiables can be updated. This queue is processed when the `propagate` operation is invoked. It can happen that the evaluation of a changeable in the queue results in a change of its corresponding modifiable. In this case, the modifiable's edges are in turn added to the queue.

Ordered list: Each edge in the dependency graph is annotated with a time interval, so that changeables in the evaluation queue (which is a priority queue) can be processed in the same order they were created. The evaluation queue is also pruned of edges whose time intervals are contained in those of other edges in the queue. The ML library uses an efficient implementation of ordered lists due to Dietz and Sleator [4] to get apt operations on time intervals.

The notion of time used in the library does not correspond to anything like real time or computation time. Rather, it is used to capture the precise order in which modifiables are read and written.

To understand how this aspect of the library works, let us consider the following program fragment:

```
m1 <- newMod $
  do v <- readMod m
     if v > 0 -- A
       then do m2 <- newMod $ do v <- readMod m
                    return (1 / v) -- B
           ...
     else ...
```

We assume that there is already a modifiable `m` created. The program fragment will add the nodes `m1` and `m2` to the dependency graph. Since both the newly created modifiables read `m`, the library adds the changeable starting at `A` and the changeable at `B` as edges to the list of dependent changeables of node `m`.

Now, suppose that the value of `m` is changed to zero. The library will move all the edges of `m` to the evaluation queue. This schedules recomputation of the changeables `A` and `B`. Now, we see that the recomputed value of `m1` will follow the `else` branch of the conditional, so `m2` is no longer part of the value of `m1`. It would thus not only be a waste to recompute `B`; it would raise a division-by-zero exception!

So the library needs to ensure that `B` does not get recomputed. We observe that the `B` changeable is contained within the `A` changeable. Therefore, the library will consider containing changeables for recomputation before contained. As it recomputes a changeable, it will prune all contained changeables from the evaluation queue. In our example, this ensures that `A` is recomputed and that `B` is pruned.

The library uses the time intervals to decide when one queued changeable is contained within another. During recomputation, an arbitrary number of time stamps might need to be created within a given time interval. Also, efficient deletion of time stamps within a time interval is needed. The ordered-list data structure by Dietz and Sleator precisely meets these requirements.

4.1 Monad transformers and parameterized modules

In order to make the library as reusable as possible, we have actually implemented a more general interface than Figure 2 shows. The more general approach uses *monad transformers* [10], and these enable us to employ the incremental library not only on top of the `I/O` monad, but on any underlying monad that provides first-class references. Therefore, we would like to parameterize the interface with respect to this underlying monad. Unfortunately, Haskell lacks the possibility to express modules that are parameterized in any way, unlike Standard ML or Cayenne [2]. Recently, a design for first-class modules in Haskell has been proposed [15], but it is not yet incorporated in any of the mainstream compilers or interpreters [5, 6].

Without the ability to parameterize our interface over the underlying monad, what do we do? We could declare a record type with fields corresponding to the operations in the interface, along the lines of [14]. However, we would still need to parameterize every new type declared in the interface by the underlying monad.

With no completely satisfying solution at hand, we resort to the traditional, verbose solution: to parameterize the individual definitions in our interface as necessary. To make the impact as small as possible, we capture the needed properties of the underlying monad in the class `Ref`:

```
class Monad m => Ref m r | m -> r where
  newRef  :: a -> m (r a)
  readRef :: r a -> m a
  writeRef :: r a -> a -> m ()
```

With the class `Ref m r`, we can capture all monads `m` with an associated *reference* type `r`. In our definition, we have used two extensions of Haskell: *multi-parameter* type classes [8], and *functional dependencies* [7], which in conjunction have turned out to be a very versatile tool. The functional dependency `m -> r` tells us that the reference type `r` is uniquely determined by the monad type `m`. This helps resolving ambiguities when we define monadic operations that need references internally. Here is a trivial example, that creates a reference and immediately reads it:

```
ex :: Ref m r => m Integer
ex = newRef 42 >>= readRef
```

This can be used in any monad which is an instance of the `Ref` class. Thanks to the functional dependency of `r` on `m`, the type of the reference can also be determined, although it is a completely internal affair to `ex`.

First-class references are the monadic equivalent of the ML type `ref`, and its associated operations `ref`, `!`, and `:=`, for creating, reading and assigning references. The standard libraries of Haskell do not have any instances for first-class references, but since the introduction of *lazy functional state threads* and the `ST` monad [9], most implementations provide libraries with first-class references.

The underlying monad `m`, its reference type `r`, and the context `Ref m r` will show up everywhere in the interfaces that we will present. Therefore, for the sake of clarity, we will take the liberty of presenting elided versions of type definitions and signatures without `m`, `r`, and `Ref m r`. As an example, this allows us to write

```
newtype OL e a
newtype R e
order :: R e -> R e -> OL e Ordering
```

```

newtype PQ a

empty    :: PQ a
insert   :: Ord a => a -> PQ a -> PQ a
insertM  :: Monad m =>
  (a -> a -> m Ordering) -> a -> PQ a -> m (PQ a)
min      :: PQ a -> Maybe (a, PQ a)

```

Figure 6. Interface for priority queues.

```

newtype OL e a
instance Monad (OL e)

newtype R e

insert    :: R e -> e -> OL e (R e)
spliceOut :: R e -> R e -> OL e ()
deleted   :: R e -> OL e Bool
order     :: R e -> R e -> OL e Ordering
base      :: OL e (R e)
inOL      :: OL e a -> m a
inM       :: m a -> OL e a

```

Figure 7. Interface for ordered lists, parameterized over `Ref m r`.

instead of

```

newtype OL m r e a
newtype R m r e
order :: Ref m r =>
  R m r e -> R m r e -> OL m r e Ordering

```

4.2 Priority queues and ordered lists

The ML library relies on two other libraries that implement priority queues and ordered lists. We give Haskell interfaces for these interfaces in Figure 6 and 7.

For ordered lists, we have provided a monad-transformer based interface, again relying on the `Ref` class. Operations on ordered lists whose elements are of type `e` live in the monad `OL e`. This monad is an instance of the *environment monad transformer* [10], modeled by parameterizing the underlying monad by the environment type. Our environment is a triple carrying the current size, the maximum size, and the base record of the ordered list:

```
newtype OL e a = OL ((r Integer, r Integer, R e) -> m a)
```

(Remember that we have assumed that this type is also parameterized over `m` and `r`.) The current size and maximum size of the ordered list can change, which is why the environment has references to these.

All records in the list have type `e`, and are linked together in a cir-

```

newtype CL a
circularList :: a -> m (CL a)
next         :: CL a -> m (CL a)
previous     :: CL a -> m (CL a)
insert      :: CL a -> a -> m (CL a)
val         :: CL a -> m a
update      :: CL a -> a -> m ()
delete      :: CL a -> m ()

```

Figure 8. Interface for circular lists, parameterized over `Ref m r`.

cular fashion. Moreover, each record has a flag that tells whether it has been spliced out, and an integer that is used for the order comparison operation. The type `R` captures all this information:

```
newtype R e = R (CL (Bool, Integer, e))
```

This relies on yet another monadic library for circular lists in Figure 8, which also relies on our first-class references. The type `CL` represents an element which is a member in a circular list. Straightforward operations are provided to create an one-element list, get to the next and previous element in the list, and to insert and return a new element after an element. The value of an element can be read or updated, and the element can also be deleted from the list.

Let us return to the operations on ordered lists in Figure 7. The operations allow us to create and insert a new record after an existing record, delete all records strictly between two records, and check if a record has been deleted. The key operation is `order`, that checks the relative position of two records, and returns a value in the standard Haskell type `Ordering`:

```
data Ordering = LT | EQ | GT
```

Using this type, a record that comes before another record has the ordering `LT`. Now, why does the operation `order` have a monadic type? Since records never move around in the list, `order` ought to be a pure function! The problem is that the ordering is determined by looking at the integer references in the records, and these might change as new records are inserted in the list. The only way the integer references can be read is by the monadic `readRef`.

The imperative signature of `order` contaminates the otherwise monad-free interface for priority queues in Figure 6. The usual `insert` operation, that works for the pure ordering operation in the `Ord` class, is useless if we want to insert elements using ordered-list order. We therefore provide an additional insertion operation that allows the comparison operation to take place in a monad. The `min` operation returns the head and tail of a priority queue, or `Nothing` if the queue is empty.

4.3 Implementation of the A and C monads

By looking inside the implementation of the incremental library in [1], we see that it uses references for maintaining its evaluation queue of edges, and for keeping track of the current time. An edge corresponds to a part of a changeable computation that starts by reading an input modifiable, and ends by writing its output modifiable. That is, whenever a changeable reads a modifiable, an edge containing the continuing changeable is inserted in the dependency graph. It has a time interval which starts with the read operation of the input modifiable, and ends at the final write operation. Time is accounted here by records in an underlying ordered list, where the values of the records are insignificant.

To get the corresponding kind of state into the `A` monad, we use the environment monad transformer again, this time on top of the ordered-list monad. The environment will have references to the process queue and the current time:

```
newtype A a = A ( (r (PQ Edge), r Time) -> OL () a )
type Time = R ()
```

It turns out that the continuations that sit in the edges precisely amount to `A`-computations, which leads to the following type for

```

propagate = do
  let prop = do
        pq <- readPq
        case min pq of
          Nothing -> return ()
          Just ((reader,start,stop),pq') -> do
            writePq pq'
            unlessM (inOL $ deleted start) $do
              inOL $ spliceOut start stop
              writeCurrentTime start
              reader
            prop
        now <- readCurrentTime
        prop
        writeCurrentTime now

```

Figure 9. The propagate operation.

Edge:

```
type Edge = (A (), Time, Time)
```

With these types, it is straightforward to define the propagate operation, see Figure 9, using the same algorithm as the ML library. It repeatedly extracts the first edge from the evaluation queue. If the start time of the edge has not been deleted from the ordered list, it deletes all edges contained in its time interval, and evaluates its changeable.

Let us now turn to the definitions of the C monad and the modifiables. In the ML library, a modifiable carries references to a value, a write operation, and a list of edges. Thus, modifiables form the nodes in the dependency graph. We get the corresponding type in Haskell:

```
newtype Mod a = Mod (r a, r (a -> A ()), r [Edge])
```

In the C monad, we will capture continuations of type A () to put in edges. This leads to the following definitions:

```
newtype C a = C ( (a -> A ()) -> A () )
deC (C m) = m
```

Usually, the continuation monad comes with the operation *call with current continuation*, or `callcc` [10]:

```
callcc :: ((a -> C b) -> C a) -> C a
callcc f = C $ \k -> deC (f (\a -> C $ \k' -> (k a))) k
```

This operation exposes the current continuation to its argument, so that one later can invoke it to “jump back” to the point of `callcc`. For our purposes, the operation `cont` is more appropriate: it gives us the possibility to completely redefine what the continuation should be, in the underlying monad:

```
cont :: ((a -> A ()) -> A ()) -> C a
cont m = C m
```

We use `cont` to create the edge in `readMod`, defined in Figure 11. The locally defined `reader`, of type A (), has captured the continuation `k`, and forms a new continuation in terms of it. The new continuation reads the value of the input modifiable, and passes it to `k`. After `k` has finished, it has written to its output modifiable, so the new continuation checks the time, and forms an edge to insert in the dependency list of the input modifiable. Here, and in what follows,

```

mapRef      :: (a -> a) -> r a -> m ()
mapRef f r = readRef r >>= (writeRef r . f)

inA         :: A a -> C a
inC         :: C a -> A a

readCurrentTime :: A Time
stepTime       :: A Time
stepTime = do
  t <- readCurrentTime
  t' <- inOL $ insert t ()
  writeCurrentTime t'
  return t'

```

Figure 10. Some auxiliary functions.

```

readMod (Mod (v,_,es)) = do
  start <- stepTime
  cont $ \k -> do
    let reader = do readRef v >>= k
                    now <- readCurrentTime
                    mapRef ((reader,start,now):) es
        reader

```

Figure 11. The readMod operation.

we use a couple of auxiliary functions given in Figure 10. These let us apply a function to the value of a reference, use A-operations in the C monad and vice versa, and read and bump the current time.

Finally, we give the implementation for the A-monad instance of the `newModBy` operation in Figure 12. Just as its corresponding ML operation `mod`, it defines two write operations that the changeable `c` will use. As the name indicates, `writeFirst` is only used the first time the value of the modifiable is being computed. It will update the reference `changeR` in the modifiable to point to `writeAgain`, which will be used whenever the modifiable needs to be recomputed. The `writeAgain` operation compares the newly computed value of the modifiable with the old, and if they differ, all dependent changeables are queued for recomputation by means of the auxiliary `insertPQ`:

```
insertPQ :: r [Edge] -> A ()
```

After the definition of these write operations, the changeable is executed to get its value `v`. This value is written to by using either `writeFirst` or `writeAgain`. This is the point where we were able to factor out the write operation from the changeable, as promised in Section 2.3.

5 Conclusions

We have presented an example of how a monadic framework can lead us to a safe interface to an imperative library for incremental computing, suitable for use in a purely functional language. After looking at some examples that use the monadic library, we found opportunities to optimize one of the standard monadic combinators. As a result, the library is able to remove some artificial dependencies, which in turn can result in less incremental work carried out when input changes.

Now, our monadic library has not only paid off by ensuring correct usage. Algorithms that use the `ap` combinator immediately benefit

```

instance NewMod A where
  newModBy :: (a -> a -> Bool) -> C a -> A (Mod a)
  newModBy cmp c = do
    m <- newRef (error "newMod")
    changeR <- newRef (error "changeR")
    es <- newRef []
    let writeFirst v = do
          writeRef m v
          now <- stepTime
          writeRef changeR (writeAgain now)
        writeAgain t v = do
          v' <- readRef m
          unless (cmp v' v) $do
            writeRef m v
            insertPQ es
            writeRef es []
          writeCurrentTime t
    writeRef changeR writeFirst
  inC $ do
    v <- c
    write <- readRef changeR
    inA $ write v
  return (Mod (m, changeR, es))

```

Figure 12. The `NewMod` instance for `A`.

from the fact that the arguments are independent, and thus isolated in the dependency graph.

From a Haskell library designer's point of view, we notice that the optimized version of the `ap` combinator for the `C` monad suggests that this combinator should be put in a class, and that other monads may benefit from optimized instances.

We find that Haskell as a language provides good support for monadic programming, but that the lack of parameterizable modules is a big disadvantage. Most of our interfaces include types and operations that are parameterized over an underlying monad, and the inability to capture this parameter at one single point decreases the clarity of the interfaces significantly. For these reasons, it was tempting to use Standard ML or Cayenne instead for implementation language. We decided to go for Haskell, since it is a widely used language that is purely functional. This makes it clear that all effects are captured in the monadic types. We are also pleased to see that there is ongoing work on first-class module systems for Haskell [15].

The implementation of our library is freely available for download at [3].

6 Acknowledgments

We would like to thank Dick Kieburtz, Thomas Hallgren, Bill Harrison, Sylvain Conchon, Walid Taha, and the anonymous referees for valuable feedback on this paper.

7 References

- [1] U. Acar, G. Blelloch, , and R. Harper. Adaptive functional programming. In *Principles of Programming Languages (POPL02)*, Portland, Oregon, January 2002. ACM.
- [2] L. Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.
- [3] M. Carlsson. *Adaptive* — incremental computations in Haskell. www.cse.ogi.edu/~magnus/Adaptive/, 2002.
- [4] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings. 19th ACM Symposium. Theory of Computing*, 1987.
- [5] The Glasgow Haskell compiler. www.haskell.org/ghc/.
- [6] The Hugs 98 interpreter. www.haskell.org/hugs/.
- [7] M. P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, number 1782 in LNCS, Berlin, Germany, March 2000. Springer-Verlag.
- [8] S. P. Jones, M. P. Jones, and E. Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.
- [9] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *ACM Programming Languages Design and Implementation*, Orlando, 1994.
- [10] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, Jan. 1995.
- [11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [12] S. Peyton Jones et al. The Haskell 98 libraries. haskell.org/onlineLibrary/, 1999.
- [13] S. Peyton Jones et al. Report on the programming language Haskell 98, a non-strict, purely functional language. Available from <http://haskell.org>, February 1999.
- [14] T. Sheard. Generic unification via two-level types and parameterized modules. In *International Conference on Functional Programming*, Florence, Italy, 2001. ACM.
- [15] M. Shields and S. P. Jones. First class modules for Haskell. In *9th International Conference on Foundations of Object-Oriented Languages (FOOL 9)*, Portland, Oregon, pages 28–40, Jan. 2002.
- [16] P. Wadler. The essence of functional programming. In *Proceedings 1992 Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.