Monadic Memoization Mixins

Daniel Brown and William R. Cook

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
{danb,wcook}@cs.utexas.edu

Abstract

Memoization is a familiar technique for improving the performance of programs: computed answers are saved so that they can be reused later instead of being recomputed. In a pure functional language, memoization of a function is complicated by the need to manage the table of saved answers between calls to the function, including recursive calls within the function itself. A lazy recursive data structure can be used to maintain past answers — although achieving an efficient algorithm can require a complex rewrite of the function into a special form. Memoization can also be defined as a language primitive — but to be useful it would need to support a range of memoization strategies. In this paper we develop a technique for modular memoization within a pure functional language. We define *monadic memoization mixins* that are composed (via inheritance) with an ordinary monadic function to create a memoized version of the function. As a case study, we memoize a recursive-descent parser written using standard parser combinators. A comparison of the performance of different approaches shows that memoization mixins are efficient for a small example.

1. Introduction

Memoization is a technique for saving the results of function calls so that subsequent calls with the same inputs do not need to be recomputed. Memoization does not change the values returned by a function — it only changes the performance characteristics of the function. Typically a memoized version of a function uses more space to store previously computed results, but it computes subsequent results more quickly.

Memoization is one of the key characteristics of algorithms for *dynamic programming*. Sophisticated memoization strategies can prune or discard the table of stored values, store values selectively, and analyze or translate inputs before storing them as memo table keys.

Memoization can be implemented in many ways. A function can be memoized by rewriting it — the new function would explicitly access and update the table of stored values. Rewriting many functions this way can be tedious. Moreover, it does not localize the common memoization behavior, thus reducing the modularity of the overall program. Modularity can allow different memoization strategies to be used for a single function.

In procedural languages memoization can be implemented as a user-defined, higher-order procedure $memo$ which produces a memoized version $memo\ f$ of any function f. This $memo$ function cannot be written in a pure functional language, but it is sometimes included as a built-in primitive. Naïve application of $memo$ will not memoize recursive calls within the function. In object-oriented languages, recursive methods can be modified using inheritance.

Inheritance is frequently viewed as specific to object-oriented programming, but the underlying concept of inheritance is the incremental modification of recursive structures, which has many other applications [4]. In particular, we show how inheritance can be used in Haskell to implement memoization.

As an alternative to explicit memoization, lazy functional languages typically support a form of memoization as a side effect of their evaluation strategy. In these systems, expressions are evaluated at most once, and only if needed. A lazy data structure that is referenced by multiple computations will be computed once and shared by all clients. This optimization can be used to memoize a function [11]. By storing the function results in an appropriate lazy data structure, each function result will be computed only once. For example, the results of the Fibonacci function can be stored in a list, where the nth item of the list is the nth Fibonacci number. This approach is the basis of the oft-cited Haskell legerdemain in Figure 1.

$$
\begin{aligned}
&\mathit{fibs} :: [\mathit{Int}]\\
&\mathit{fibs} = 0 : 1 : [a + b \mid (a, b) \leftarrow \mathit{zip}\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})]\\
&\mathit{zipFib}\ n = \mathit{fibs}\ !!\ n
\end{aligned}
$$

Figure 1. Fibonacci as a lazy list computation

The fib function accesses the nth item of a lazy data structure. The list comprehension defining the data structure contains two references to itself. Each node in the lazy list is computed once, on demand, and the other references reuse the already computed list values. One problem with this technique in general is that memoizing an algorithm requires the algorithm to be rewritten into a special form — in this case, the implementation is radically different from the standard description of the Fibonacci function.

In this paper we provide a pure functional account of memoization based on *monadic memoization mixins*. We consider several distinct cases where memoization can be applied: ordinary recursive functions, recursive functions returning monadic values, mutually-recursive functions, and monadic values with functional behavior. For ordinary functions, *monadification* [6] is used to introduce a monad parameter into the function. Recursive functions are converted to generators with open recursion [4]. A generalized memoization *mixin* is developed, which is composed using inheritance with the original function to create the memoized version. For functions returning monadic values, we show how the existing monad can be extended to support memoization. To do this, we develop a new class that relates a monad transformer to its corresponding monad. We compare the performance of the different approaches to memoization. As a final case study, we memoize a top-down parser written using standard parser combinators.

2. Memoizing Recursive Functions

Memoizing a simple recursive function is a good illustration of the technique of monadic memoization mixins. The underlying ideas behind this approach are well known, but have not been systematically studied in the context of mixins and monads in pure functional languages. As an example, consider the Fibonacci function.

$$\begin{aligned} &\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\ &\mathit{fib}\ 0 \qquad\quad = 0 \\ &\mathit{fib}\ 1 \qquad\quad = 1 \\ &\mathit{fib}\ (n+2) = \mathit{fib}\ n + \mathit{fib}\ (n+1) \end{aligned}$$

The execution time for *fib* n grows exponentially in n because *fib* is called many times with the same input.

While it is possible to rewrite *fib* to include memoization, the memoization code would be tangled with the Fibonacci computation. Instead, we explore ways to generalize *fib* so that it can be memoized by composing it with an appropriate memoization function.

2.1 Monadification

Modifying a function to use a monad is called *monadification*[6]. The new monad can be used to pass state between the recursive calls of the function – the state is the memo table which may be updated after each function call. Rather than rewrite *fib* to use an explicit state monad, *fib* can be rewritten to use a generic monad parameter, which can be bound to an appropriate state monad during memoization. The monadic version mFib of *fib* returns computations in a generic monad m:

$$\begin{aligned} &\mathit{mFib} :: \mathit{Monad}\ m \Rightarrow \mathit{Int} \to m\ \mathit{Int} \\ &\mathit{mFib}\ 0 \qquad\quad = \mathit{return}\ 0 \\ &\mathit{mFib}\ 1 \qquad\quad = \mathit{return}\ 1 \\ &\mathit{mFib}\ (n+2) = \mathbf{do}\ a \leftarrow \mathit{mFib}\ n \\ &\qquad\qquad\qquad\quad\ \ b \leftarrow \mathit{mFib}\ (n+1) \\ &\qquad\qquad\qquad\quad\ \ \mathit{return}\ (a+b) \end{aligned}$$

The recursive calls must be executed in the same monad. This has the effect of serializing the recursive calls, which will enable the memo table to be passed into the first call and the resulting table to be passed into the second call.

The original Fibonacci function can be recovered by running mFib in the identity monad:

$$\begin{aligned} &\mathit{fib}_m :: \mathit{Int} \to \mathit{Int} \\ &\mathit{fib}_m = \mathit{runIdentity} \circ \mathit{mFib} \end{aligned}$$

$\mathit{runIdentity} :: \mathit{Identity}\ a \to a$ serves two purpose here: it binds the monad parameter m in mFib to the $\mathit{Identity}$ monad and then extracts the Int from the resulting trivial computation of type $\mathit{Identity}\ \mathit{Int}$.

2.2 Open Recursion and Inheritance

For memoization to affect recursive calls, the self-reference in *fib* must be exposed, or opened, so that it can be rebound to refer to the memoized version of *fib*. This is exactly what *inheritance* does in object-oriented languages [5]; the same technique can be applied to functions [4]. To do so, we abstract the self-reference in *fib* as an explicit *self* parameter, then reconstruct fib_g using an explicit fixed point:

$$\begin{aligned} &\mathit{gFib} :: (\mathit{Int} \to \mathit{Int}) \to (\mathit{Int} \to \mathit{Int}) \\ &\mathit{gFib}\ \mathit{self}\ 0 \qquad\quad = 0 \\ &\mathit{gFib}\ \mathit{self}\ 1 \qquad\quad = 1 \\ &\mathit{gFib}\ \mathit{self}\ (n+2) = \mathit{self}\ n + \mathit{self}\ (n+1) \end{aligned}$$

$$\begin{aligned} &\mathit{fib}_g :: \mathit{Int} \to \mathit{Int} \\ &\mathit{fib}_g = \mathit{fix}\ \mathit{gFib} \end{aligned}$$

Functions like gFib used to specify a fixed-point are called *generators* [5]. They have types of the form $a \to a$. Generators will appear frequently, so we introduce a type function to simplify their types:

$$\mathbf{type}\ \mathit{Gen}\ a = a \to a$$

Now gFib can be given the simpler type $\mathit{Gen}\ (\mathit{Int} \to \mathit{Int})$.

Inheritance works by composing generators before computing the fixed point. For memoization, the memoized Fibonacci function will have the form $\mathit{fix}\ (\mathit{memo} \circ \mathit{gFib})$ for an appropriate memo and generator gFib. This has the effect of binding self-reference in gFib to the memoized version of the function. In this context, memo is a *mixin* [2].

Object-oriented languages support open recursion implicitly: every recursive definition implicitly defines a generator which can be inherited using special syntax. The same thing could be supported in Haskell. The syntax memo **inherit** *fib* could be defined to mean $\mathit{fix}\ (\mathit{memo} \circ \mathit{gFib}_0)$ where gFib_0 is the generator of *fib*. As we shall see in Section 7, there is a significant performance penalty for using explicit fixed-points in Haskell to implement inheritance; direct support for inheritance could improve this situation.

2.3 Monadic Fibonacci Generator

The versions of *fib* with open and monadic recursion are combined to create a monadic Fibonacci generator. Since open recursion and monadification are orthogonal operations, they can be performed in either order to yield the same result:

$$\begin{aligned} &\mathit{gmFib} :: \mathit{Monad}\ m \Rightarrow \mathit{Gen}\ (\mathit{Int} \to m\ \mathit{Int}) \\ &\mathit{gmFib}\ \mathit{self}\ 0 \qquad\quad = \mathit{return}\ 0 \\ &\mathit{gmFib}\ \mathit{self}\ 1 \qquad\quad = \mathit{return}\ 1 \\ &\mathit{gmFib}\ \mathit{self}\ (n+2) = \mathbf{do}\ a \leftarrow \mathit{self}\ n \\ &\qquad\qquad\qquad\qquad\quad\ \ b \leftarrow \mathit{self}\ (n+1) \\ &\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (a+b) \end{aligned}$$

$$\begin{aligned} &\mathit{fib}_{gm} :: \mathit{Int} \to \mathit{Int} \\ &\mathit{fib}_{gm} = \mathit{runIdentity} \circ (\mathit{fix}\ \mathit{gmFib}) \end{aligned}$$

The three functions fib_g, fib_m, and fib_{gm} all behave the same as *fib*.

2.4 Memoization Mixin

A memoized version f_M of a function f has a standard pattern, based on a table of previous results. The function call $f_M(x)$ first checks if x has a value in the table, and if so returns the stored result. If not, it computes $f(x)$ and then stores the result in the table. In a pure functional language, an explicit memo table is passed as an input to f_M (and to the recursive call $f_M(x)$), and the possibly updated table is returned as a result. This kind of computation is naturally expressed using the State monad with the memo table as the state. However, various kinds of tables or state-like monads might be used, so we parameterize the memo function by two accessor functions to check whether a value has already been computed, and to store new values that are computed. These two functions constitute a dictionary interface $\mathit{Dict}\ a\ b\ m$, where a is the key type, b is the value type, and m is the state monad:

$$\begin{aligned} \mathbf{type}\ \mathit{Dict}\ a\ b\ m = (&a \to m\ (\mathit{Maybe}\ b), \\ &a \to b \to m\ ()) \end{aligned}$$

Given a dictionary, the memo mixin is easily defined, as shown in Figure 2. Following a convention from object-oriented programming [9], the argument of a mixin is called super.

While it would be desirable to encapsulate check and store within some type class for memo tables and stateful monads, this approach does not work in cases where multiple dictionaries have the same type (see Section 6).

```
memo :: Monad m ⇒ Dict a b m → Gen (a → m b)
memo (check, store) super a = do
  b ← check a
  case b of
    Just b    → return b
    Nothing → do b ← super a
                      store a b
                      return b
```

Figure 2. Memoization Mixin

2.5 Memoized Fibonacci

Finally, the *memo* mixin is combined with the generator of *fib* to create the memo function *memoFib*. Notice that the particular representation for the memo table is still unspecified.

```
type Memoized a b m = Dict a b m → a → m b
memoFib :: Monad m ⇒ Memoized Int Int m
memoFib dict = fix (memo dict ∘ gmFib)
```

The type *Memoized a b m* represents the memoized version of a function type $a \to b$ abstracted over a memo dictionary. One way to specify the memo dictionary's table is with a standard *Data.Map* object with *lookup* and *insert* operations:

```
mapDict :: Ord a ⇒ Dict a b (State (Map a b))
mapDict = (check, store) where
  check a    = gets (lookup a)
  store a b = modify (insert a b)
memoMapFib :: Int → State (Map Int Int) Int
memoMapFib = memoFib mapDict
```

The function *memoMapFib* is memoized with a Map to store computed values. Since *memoMapFib* exposes the stateful monad that carries the memo table, a client can reuse the same table across separate uses of the function, given that the client is written to handle a state monad. On the other hand, if the client doesn't need this kind of reuse and only wants to memoize recursive calls, a simpler version can be defined with the same interface as *fib*:

```
runMemoMapFib :: Int → Int
runMemoMapFib n = evalState (memoMapFib n) empty
```

The function *evalState* :: *State s a* → *s* → *a* runs the stateful computation *memoMapFib n* with the initial state *empty*, an empty map, and returns the *Int* result of that computation.

For efficiency, the memo table might instead be implemented as an array. Haskell provides a variety of array types; to use one for memoization, an appropriate pair of accessors must be defined. (The details of using the *MArray* array type and the *ST* monad aren't relevant to our discussion, but we include the code in full for completeness.)

```
arrayDict  ::  (MArray arr (Maybe b) m, Ix a, Ord a)
                ⇒ a → arr a (Maybe b) → Dict a b m
arrayDict size arr = (check, store) where
  check a    = if a > size then return Nothing
                                  else   readArray arr a
  store a b = if a > size then return ()
                                  else   writeArray arr a (Just b)
```

With *arrayDict* in hand, a memoized *fib* with an array memo table is easily defined:

```
newSTArray :: Ix i ⇒ (i, i) → e → ST s (STArray s i e)
newSTArray = newArray
runMemoArrayFib :: Int → Int → Int
runMemoArrayFib size n = runST (do
  arr ← newSTArray (0, size) Nothing
  memoFib (arrayDict size arr) n)
```

In summary, *fib* was memoized by monadifying, opening recursion, and then composing with a memo mixin. The memo mixin is parameterized by functions that interact with a memo table within a stateful monad. Next, we consider memoizing a function that is already defined in a monadic style.

3. Memoizing Monadic Functions

A function that returns monadic values can be memoized just like an ordinary function: the memo table simply contains monadic values that have been previously computed. To memoize, a new monad is inserted by monadification, such that all recursive calls are evaluated in the new monad.

Consider a function that computes the fringe of a tree: given a tree with values at its leaves, *fringe* computes a list of values representing the pre-order traversal of the leaves.

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
  deriving (Show, Eq)
fringe :: Tree a → [a]
fringe (Leaf a)     = [a]
fringe (Fork t u) = fringe t ++ fringe u
```

The preimage (of singleton sets of output values) of *fringe* is the function *unfringe* which computes the set of trees that have a given fringe. Functions like *unfringe* that produce a set of results are naturally written using the list monad, which supports iteration over multiple sub-results while combining these to produce a list of final results.

```
unfringe :: [a] → [Tree a]
unfringe [a] = [Leaf a]
unfringe as  = do
  (l, k) ← partitions as
  t        ← unfringe l
  u        ← unfringe k
  return (Fork t u)
```

The function *partitions* computes the various binary partitions of a list:

```
partitions :: [a] → [([a], [a])]
partitions as = [splitAt n as | n ← [1 . . length as − 1]]
```

3.1 Monadification of *Unfringe*

The function *unfringe* can be transformed to add a monad parameter and open recursion in the same way *fib* was transformed in Sections 2.1 & 2.2.

Even though *unfringe* uses a list monad, the list monad doesn't carry state, so we can't use it to memoize *unfringe*; more importantly, list computations are the object of memoization, so this wouldn be the wrong approach! Therefore a new monad parameter is introduced independently from the existing list monad. The recursive calls to *unfringe* are run in this new monad. The result, *gmUnfringe*, is openly recursive via *self* and is parameterized over a monad *m*:

```
gmUnfringe :: Monad m ⇒ Gen ([a] → m [Tree a])
gmUnfringe self [a] = return [Leaf a]
gmUnfringe self as  =
  liftM concat (sequence (do    -- In []
    (l, k) ← partitions as
return (do        -- In m
  ts ← self l
  us ← self k
  return (do      -- In []
    t ← ts
    u ← us
    return (Fork t u)))))
```

This straightforward but inelegant monadification of *Unfringe* produces a function written in two monads: [] and m. Computation in the two monads is interleaved, making the resulting code difficult to understand. In the next section we show how to coordinate the two monads with monad transformers.

Note that *unfringe* uses the list monad for two purposes: to iterate over the partitions, and to iterate over the results of recursive calls. Monadification separates the two uses. The first use of the list monad, to iterate over partitions, must be lifted and the results concatenated to produce the final result.

The monad generator *gmUnfringe* can be run by closing the recursion and binding the monadic parameter m to the Identity monad:

```
unfringe_gm :: [a] → [Tree a]
unfringe_gm = runIdentity ∘ fix gmUnfringe
```

The result, $unfringe_{gm}$, behaves the same as *unfringe*.

The memo mixin and accessors defined in Sections 2.4 & 2.5 apply to *gmUnfringe* just as they did for *gmFib*:

```
memoUnfringe :: (Ord a, Monad m)
               ⇒ Memoized [a] [Tree a] m
memoUnfringe access = fix (memo access ∘ gmUnfringe)
runMemoUnfringe :: Ord a ⇒ [a] → [Tree a]
runMemoUnfringe l =
  evalState (memoUnfringe mapDict l) empty
```

4. Memoization via Monad Transformers

The unbound monad parameter required for memoization can also be introduced by lifting a monad to a monad transformer. For example, *unfringe* uses the list monad, so it can easily be lifted into a monad *ListT m*, for some monad m; the newly-introduced monad parameter becomes the memoization monad. The benefit of doing this is that the resulting function *gmUnfringeT* is defined similarly to *unfringe* and avoids the interleaved monads found in *gmUnfringe*:

```
gmUnfringeT :: Monad m
             ⇒ Gen ([a] → ListT m (Tree a))
gmUnfringeT self [a] = return (Leaf a)
gmUnfringeT self as  = do
  (l, k) ← ListT (return (partitions as))
  t      ← self l
  u      ← self k
  return (Fork t u)
```

As in *gmUnfringe*, there are still two different uses of lists: one for the function being defined, and another for *partitions as*. The latter is lifted into the monad transformer via *ListT* ∘ *return*.

4.1 Transformer-based Memoization Mixin

It might seem reasonable to define a memoization mixin by lifting the memo table operations into the monad transformer. This strategy produces the following memo mixin:

```
memo_X :: (Monad m, MonadTrans t, Monad (t m))
        ⇒ Dict a b m → Gen (a → t m b)
memo_X (check, store) super a = do
  b ← lift (check a)
  case b of
    Just b   → return b
    Nothing → do b ← super a
                 lift (store a b)
                 return b
```

Unfortunately, $memo_X$ misbehaves. In the case of *ListT*, it memoizes the first value of the computation and ignores the rest. The problem is that the memo mixin must capture the monadic value that comes from the transformed monad, rather than interleaving with the *effect* of this monad. This problem can be seen in the type of $memo_X$: The memo table defined by *Dict a b m* stores values of type b, not monads of type *t Id b*. In the case of *ListT m b*, the memo table stores values of type b, but it should store values like $[b]$.

To define a memoization mixin that works with monad transformers, there must be a connection between the transformer t and the monad n to which it is related. This relationship can be expressed by a type class that specifies an isomorphism between the transformer and its corresponding monad:

```
class (MonadTrans t, Monad n) ⇒
        TransForMonad t n | t → n, n → t where
  toTrans   :: m (n a) → t m a
  fromTrans :: t m a → m (n a)
```

To define *TransForMonad* we use a common Haskell extension called *functional dependencies* [14]: the syntax $t \rightarrow n$ specifies that the type t uniquely determines n, and the second clause $n \rightarrow t$ says that n determines t. In general, the compiler uses the relationships specified by functional dependencies to infer types for functions like *toTrans* and *fromTrans*, where some variables occur only in the output type. The bidirectional specification between n and t creates the isomorphism between a monad and its transformer. A relevant example of this relationship is the list monad [] and its transformer *ListT*:

```
instance TransForMonad ListT [] where
  toTrans   = ListT
  fromTrans = runListT
```

The transformer-based memo mixin is defined by adapting the basic memo mixin to work within the transformed monad, while capturing values of the outer monad:

```
memo_T :: (TransForMonad t n, Monad m)
        ⇒ Dict a (n b) m → Gen (a → t m b)
memo_T dict f =
  toTrans ∘ memo dict (fromTrans ∘ f)
```

Note that the memo table now stores computations of type *n b*, where n is the monad associated with the transformer t.

The monad transformer version of the memo mixin can now be used with transformer-based functions like before:

```
type MemoizedT a n t b m =
  Dict a (n b) m → a → t m b
memoUnfringeT :: Monad m
               ⇒ MemoizedT [a] [] ListT (Tree a) m
memoUnfringeT dict = fix (memo_T dict ∘ gmUnfringeT)
runMemoUnfringe_T :: Ord a ⇒ [a] → [Tree a]
runMemoUnfringe_T a =
  evalState (runListT (memoUnfringeT mapDict a))
            empty
```

An aside: it is unfortunate that the type synonym $MemoizedT$ must include explicit arguments for both n and t. It would seem that either variable could be inferred from the other by the functional dependencies specified by $TransForMonad$, but GHC's type synonyms aren't designed to use the information specified by functional dependencies in this way. A more flexible type synonym feature might allow two new uses: a way to compute types that are uniquely determined by other types and variables via functional dependencies, and pattern matching, as in $(n\ b)$ below.

```
type MemoizedT′ a (n b) m = TransForMonad t n
   ⇒ Dict a (n b) m → a → t m b
```

This way, $MemoizedT'\ a\ [b]\ m$ would expand to $Dict\ a\ [b]\ m \rightarrow a \rightarrow ListT\ m\ b$ by computing that the constraint $TransForMonad\ t\ [\,]$ uniquely determines t as $ListT$. However, this syntax is unfortunate since it already means something else in GHC! GHC expands $MemoizedT'\ a\ [b]\ m$ to $forall\ t\ a\ b.\ TransForMonad\ t\ [\,] \Rightarrow Dict\ a\ [b]\ m \rightarrow a \rightarrow t\ m\ b$, which is different. Given a more clever syntax, we think these two extensions to type synonyms would be useful.

5. Memoizing Mutual Recursion

A set of mutually recursive functions can be memoized by maintaining a collective state with a memo table for each individual function. In general the functions may have different types, so their corresponding memo tables may have different types as well. In this section we develop a technique to memoize a pair of mutually recursive, non-monadic functions. The parser case study we will consider later will have many mutually recursive functions, and we will show how this method generalizes.

Consider a pair of mutually recursive functions f and g (designed primarily to have different types):

```
f :: Int → (Int, String)
f 0           = (1, "+")
f (n + 1)     = (g (n, fst (f n)), "-" ++ snd (f n))

g :: (Int, Int) → Int
g (0, m)      = m + 1
g (n + 1, m)  = fst (f n) − g (n, m)
```

The technique for defining mutually recursive functions using an explicit fixed-point is standard: the fixed-point generator operates on tuples of functions. In this case, the tuple is a pair with type $(Int \rightarrow (Int, String), (Int, Int) \rightarrow Int)$. As in the case for *fib*, these non-monadic functions must be monadified to introduce a monad parameter. The result is a generator of a pair of functions parameterized by a monad m which serializes the recursive calls to f and g:

```
type MFuns m =
  (Int → m (Int, String), (Int, Int) → m Int)

gmFG :: Monad m ⇒ Gen (MFuns m)
gmFG ∼(f, g) = (f′, g′) where
  f′ 0          = return (1, "+")
  f′ (n + 1)    = do a ← f n
                     b ← g (n, fst a)
                     return (b, "-" ++ snd a)

  g′ (0, m)     = return (m + 1)
  g′ (n + 1, m) = do a ← f n
                     b ← g (n, m)
                     return (fst a − b)
```

The input, a pair of functions f and g, represents the self parameters, while f' and g' are the functions produced by the generator. The pattern (f, g) is made lazy with the $\sim$ symbol to prevent divergence. Whereas similar encodings for object-oriented languages typically use records instead of tuples, Haskell records are akward to use because they lack first-class injection functions, so we use tuples instead.

The function f is easily recovered from $gmFG$ by taking its fixed-point, projecting, and running the result through the identity monad.

```
f_gm :: Int → (Int, String)
f_gm = runIdentity ∘ (fst (fix gmFG))
```

The function f_{gm} behaves the same as f.

5.1 Memoizing Mutually Recursive Functions

To memoize mutually recursive functions, a memoization mixin must be composed with each generator individually, yet each memo mixin must read and write to a separate part of a shared state. The memo mixin for f and g is a function on pairs, which uses a pair of dictionaries to access the store:

```
memoFGMixin :: (Monad m, Monad m′)
            ⇒ (Dict a b m, Dict a′ b′ m′)
            → Gen (a → m b, a′ → m′ b′)
memoFGMixin (df, dg) (f, g) = (memo df f, memo dg g)

memoF :: Monad m
      ⇒ (Dict Int (Int, String) m,
         Dict (Int, Int) Int m)
      → Int → m (Int, String)
memoF dicts = fst (fix (memoFGMixin dicts ∘ gmFG))
```

One strategy for representing memo tables for mutual recursion is to maintain a map for each function. Access to the ith map is provided by a pair of a projection and injection functions:

```
type Accessor a b = (b → a, a → b → b)
```

The function *selMap* creates a pair of dictionary accessors given a projection/injection pair. It assumes that the components of the pair are Maps.

```
selMap :: Ord a
       ⇒ Accessor (Map a b) s → Dict a b (State s)
selMap (proj, inj) = (check, store) where
  check a   = gets (lookup a ∘ proj)
  store a b = modify (λs → inj (insert a b (proj s)) s)
```

To run the memoized version of f, the memo function $memoF$ is applied to an appropriate pair of accessors and executed in an empty state. We assume a family of projection and injection functions $proj_{i/n}$ and $inj_{i/n}$ for accessing the ith component of an n-tuple. Let $acc_{i/n} = (proj_{i/n}, inj_{i/n})$. (These could easily be written by hand or generated with metaprogramming.)

```
runMemoF :: Int → (Int, String)
runMemoF n =
  evalState (memoF dicts n) (empty, empty) where
  dicts = (selMap acc_1/2, selMap acc_1/2)
```

The shared state for f and g contains a Map for each function of the appropriate type:

```
type MemoFG = State (Map Int (Int, String),
                     Map (Int, Int) Int)
```

5.2 Tuple Operations with Template Haskell

When the tuple contains multiple recursive functions, this construction becomes tedious, but there is no way to abstract over tuple elements in standard Haskell. Template Haskell [22] provides an elegant solution by generating the $proj_{i,n}$ and $inj_{i,n}$ functions automatically.

We use psuedo-Template Haskell here to specify functions instead of defining them; their proper definitions are straightforward and not relevant to the discussion. To *specify* a Template Haskell function, we define the result of splicing it and use ellipses and subscripts to represent tuples of arbitrary length.

The Template Haskell functions needed for this example are determined by their types. First we define projection and injection functions. The integer pair (i, n) is used to indicate the ith component of a tuple of size n.

$$\$\,(proj\ (i,n)) :: (a_1, ..., a_n) \rightarrow a_i$$
$$\$\,(proj\ (i,n)) = \lambda(a_1, ..., a_n) \rightarrow a_i$$

$$\$\,(inj\ (i,n)) ::$$
$$\quad a \rightarrow (a_1, ..., a_n) \rightarrow (a_1, ..., a_{i-1}, a, a_{i+1}, ..., a_n)$$
$$\$\,(inj\ (i,n)) =$$
$$\quad \lambda a\ (a_1, ..., a_n) \rightarrow (a_1, ..., a_{i-1}, a, a_{i+1}, ..., a_n)$$

An accessor pair can now be defined in terms of $proj$ and inj:

$$\$\,(accessors\ (i,n)) :: Accessors\ a\ b\ s$$
$$accessors\ l = [|\ (\$(proj\ l), \$(inj\ l))\ |]$$

Finally, $mapAccT$ maps a function over a tuple, much like map maps a function over a list, also providing the function an accessor pair as an initial input:

$$\$\,(mapAccT\ n\ [|\ f\ |])\ (a_1, ..., a_n) =$$
$$\quad (f\ \$\,(accessors\ (1,n))\ a_1, ..., f\ \$\,(accessors\ (n,n))\ a_n)$$

The $memoFGMixin_{TH}$ is now defined simply as a call to $mapAccT$:

$$memoFGMixin_{TH} :: Gen\ (MFuns\ MemoFG)$$
$$memoFGMixin_{TH} = \$(mapAccT\ 2\ [|\ memo \circ selMap\ |])$$

This approach scales up to any number of functions in the mutually recursive definition, with potentially different types, as long as they are all unary. To create the memoized function, the mixin is applied as before.

6. Memoizing Parsers

In this section we consider a case study: memoizing monadic parsers. Parsers are commonly expressed in Haskell top-down as state monads with failure (and optionally non-determinism) [12, 18]. State monads are an instance of a more general structure we call *functional monads* — monads that encapsulate a functional behavior. State monads fit this mold because each contains a state transformation function. We will show how any functional monad — and indeed anything with a function-like behavior — can be memoized by mapping to its functional behavior, memoizing it, and mapping back into the original construct — possibly with a new type, depending on whether the memo tables need to be reused.

Since parsers tend to be mutually recursive, the case study will employ the techniques developed in Section 5; since grammars tend to require a fair number of mutually recursive parsers, those results will be generalized to arbitrary numbers of mutually recursive functions.

While the transformations in this section may appear complex, they much simpler than the transformations used in creating the Packrat parser, which is memoized with a complex lazy data structure of linked $Derivs$ structures [7]. The version presented here requires less invasive transformations and thus is closer to the original, simple parser.

We take as an example a parser for the “grammar for a trivial language” in figure 1 of [7]:

Additive	::=	Multitive ‘+’ Additive | Multitive
Multitive	::=	Primary ‘*’ Multitive | Primary
Primary	::=	‘(’ Additive ‘)’ | Decimal
Decimal	::=	‘0’ | ... | ‘9’

Our parser combinators are modeled after the monadic Packrat parsing style of section 3.3 in [7], which is very similar to Parsec [18]. We use a variant of the standard parsing monad [12] and store the parsed result in a $Maybe$ instead of a $List$. This provides a depth-first rather than breadth-first parsing strategy for deterministic parsers. Extending our technique to non-deterministic parsers should be straightforward.

```
type Parser a = StateT String Maybe a

mkParser :: (String → Maybe (a, String)) → Parser a
mkParser = StateT

runParser :: Parser a → String → Maybe (a, String)
runParser = runStateT
```

The functions $mkParser$ and $runParser$ simply map between the function that defines a parser's behavior and the parser itself.

The grammar is implemented as a collection of mutually recursive parsers:

```
additive :: Parser Int
additive = do
  a ← multitive; char '+'; b ← additive; return (a + b)
   <|> multitive
multitive :: Parser Int
multitive = do
  a ← primary; char '*'; b ← multitive; return (a ∗ b)
   <|> primary
primary :: Parser Int
primary = do
  char '('; a ← additive; char ')'; return a
   <|> decimal
decimal :: Parser Int
decimal = do
  c ← anyChar
  guard (inRange ('0', '9') c)
  return (read [c] :: Int)
```

This mutually recursive parser depends on the parsers $(<|>)$, $char$, and $anyChar$. The $(<|>)$ parser is simply $mplus$ specialized to $Parser$, which is an instance of $MonadPlus$ because $StateT\ String\ Maybe$ is one.

```
(<|>) :: Parser a → Parser a → Parser a
(<|>) = mplus
```

The $char$ parser parses a specified character, and $anyChar$ simply returns the next character to parse. $char$ is easily defined in terms of $anyChar$, and $anyChar$ interacts directly with the underlying $String$ state:

```
char :: Char → Parser Char
char c = do
  d ← anyChar; guard (c == d); return d
anyChar :: Parser Char
anyChar = do
  s ← get
  case s of
    []     → mzero
    c : s′ → do put s′; return c
```

These three parsers constitute the parsing library on top of which our parser is built; in practice these libraries are much larger with many more features!

Finally, a string from the grammar is parsed with *parse*, which runs the parser that corresponds to the start symbol in the grammar, Additive:

```
parse :: String → Maybe (Int, String)
parse = runParser additive
```

6.1 Monadifying a Single Parser

Memoizing the parsers requires the same initial transformation described in Section 5: monadify each parser and combine all of the mutually recursive parsers together into a tuple generator. First, we examine how to monadify a parser. Since a parser is a monad that encapsulates a function, monadifying a parser requires monadifying the encapsulated function and wrapping it back up as a parser again.

Monadifying such a state transformation function changes its type from $String \to Maybe\ (a, String)$ to $String \to m\ (Maybe\ (a, String))$. Thinking of m as a memoization monad carrying a memo table, this makes sense: we want the memo table to associate input strings with values in $Maybe\ (a, String)$. This will memoize both successful and failed parses, which is also what we expect. Unfortunately, the *Parser* monad can't wrap functions with that type. This motivates a generalization of the parser monad: a parser monad transformer.

```
type ParserT m a = StateT String (MaybeT m) a
mkParserT :: Monad m
    ⇒ (String → m (Maybe (a, String))) → ParserT m a
mkParserT f = StateT (MaybeT ∘ f)
runParserT :: Monad m
    ⇒ ParserT m a → String → m (Maybe (a, String))
runParserT p = runMaybeT ∘ runStateT p
```

The monad $ParserT\ m$ nicely wraps a monadified state transformation function with type $String \to m\ (Maybe\ (a, String))$.

Although *MaybeT* is not in the standard Haskell libraries, its definition is well-known. The necessary instance declarations are straight-forward and omitted here.

```
newtype MaybeT m a =
    MaybeT{runMaybeT :: m (Maybe a)}
```

So to monadify a parser with type $Parser\ \ a$, it suffices to type it as a parser transformer over an unbound monad using type $ParserT\ \ m\ \ a$. Since the parser monad is the outer monad, none of the parser code needs to be changed! That is, a $Parser\ a$ value is a $ParserT\ m\ a$ value when all of its sub-parsers are typed with $ParserT\ m$ instead of *Parser*.

Unfortunately, re-typing *every* sub-parser requires re-typing every sub-parsers' sub-parsers, and eventually requires monadifying the *entire* parsing library, which violates basic assumptions about modularity and abstraction. On the other hand, we must monadify *some* of the sub-parsers simply to support mutual recursion.

While monadifying entire parsing libraries isn't feasible in practice, we feel our method is a first step toward memoizing real-world parsers. We leave the development of more modular techniques as future work.

6.2 Transforming All of the Parsers

Monadifying each parser and applying the mutual recursion techniques developed previously, the mutually recursive parsers *additive*, *multitive*, *primary*, and *decimal* together with the parsing library *char* and *anyChar* become a six-tuple of parsers:

```
type Parsers m = (ParserT m Int,
                  ParserT m Int,
                  ParserT m Int,
                  ParserT m Int,
                  Char → ParserT m Char,
                  ParserT m Char)
```

Since the individual parser definitions remain unchanged, the tuple generator is also easily defined.

```
gmPars :: Monad m ⇒ Gen (Parsers m)
gmPars ~(add, mult, prim, decimal, char, any) =
  (add′, mult′, prim′, decimal′, char′, any′) where
  add′ :: ParserT m Int = do
    a ← mult; char '+'; b ← add; return (a + b)
     <|> mult
  mult′ :: ParserT m Int = do
    a ← prim; char '*'; b ← mult; return (a ∗ b)
     <|> prim
  prim′ :: ParserT m Int = do
    char '('; a ← add′; char′ ')'; return a
     <|> decimal
  decimal′ :: ParserT m Int = do
    c ← any
    guard (inRange ('0', '9') c)
    return (read [c] :: Int)
  char′ (c :: Char) :: ParserT m Char = do
    d ← any; guard (c == d); return d
  any′ :: ParserT m Char = do
    s ← get
    case s of
      [] → mzero
      c : s′ → do
        put s′
        return c
```

The input tuple $(add, mult, prim, decimal, char, any)$ represents the self parameters.

The original function can be obtained, as usual, by taking the fixed-point, selecting the function corresponding to the start symbol, and then running the unbound computation in the identity monad:

```
parse_gm :: String → Maybe (Int, String)
parse_gm = runIdentity ∘ runParserT (proj_1/6 (fix gmPars))
```

We use $proj_{1/6}$ to denote the function that projects the first element from a six-tuple. The parser $parse_{gm}$ behaves the same as *parse*.

6.3 Memoizing the Parsers

Now that the parsers are transformed into a tuple generator, they are almost ready to be memoized. The last step is to compose the memo mixin with the transformer function within each parser. This is trivial most of the parsers, but the fifth parser *char* with type $Char \to ParserT\ \ m\ \ Char$ requires some currying and poses a more general question: since the memo mixin expects a unary function, how do we memoize parsers with an arbitrary number of input parameters?

Our solution to this problem is to introduce a type class for constructs that are "like functions" but lack a direct functional representation:

```
class Functional f a b | f → a b, a b → f where
  toFun   :: f → (a → b)
  fromFun :: (a → b) → f
```

Parser types easily fit this mold:

```
instance Monad m ⇒
        Functional (ParserT m a)
                   String
                   (m (Maybe (a, String)))
  where
  fromFun = mkParserT
  toFun   = runParserT
```

And to solve the problem posed above, n-ary functions also fit the mold, reducing to $(n-1)$-ary functions:

```
instance Functional f b c ⇒
        Functional (a → f) (a, b) c where
  fromFun = (fromFun∘) ∘ curry
  toFun   = uncurry ∘ (toFun∘)
```

Thus a parser that takes n arguments has an *alternate* representation as a unary state transformation function whose input is an $n+1$ tuple, combining the n parser arguments with the state input. For example, the necessary instance for the type of the $char$ parser would be deduced as:

```
instance Monad m ⇒
        Functional (Char → ParserT m Char)
                   (Char, String)
                   (m (Maybe (Char, String)))
  where
  fromFun = (mkParserT∘) ∘ curry
  toFun   = uncurry ∘ (runParserT∘)
```

With this abstraction we can define a function $wrap$ that composes a given generator g with the function underlying something “like a function”:

```
wrap :: Functional f a b ⇒ Gen (a → b) → Gen f
wrap g = fromFun ∘ g ∘ toFun
```

The function $wrap$ can be used to extend the behavior of a parser that takes any number of arguments. For a zero-argument parser like $additive$, $wrap\ g\ additive$ has type $ParserT\ m\ Int$ when g has type $Gen\ (String \to m\ (Maybe\ (Int, String)))$. For a one-argument parser like $char$, $wrap\ g\ char$ has type $Char \to ParserT\ m\ Char$ when g has type $Gen\ ((Char, String) \to m\ (Maybe\ (Char, String)))$. For parsers with multiple arguments, the input type to g becomes a nesting of pairs: $(a,(b,c))$ for two-argument parsers, $(a,(b,(c,d)))$ for three-argument parsers, and so on.

With $wrap$, we can define a memo mixin for the six parsers, again using Template Haskell to enable the use of arbitrarily-sized tuples:

```
$ (mapT n [| f |]) (a_1, ..., a_n)          = (f a_1, ..., f a_n)
$ (applyT n) (f_1, ..., f_n) (a_1, ..., a_n) = (f_1 a_1, ..., f_n a_n)

type ParserDicts m =
  (Dict String         (Maybe (Int, String)) m,
   Dict String         (Maybe (Int, String)) m,
   Dict String         (Maybe (Int, String)) m,
   Dict String         (Maybe (Int, String)) m,
   Dict (Char, String) (Maybe (Char, String)) m,
   Dict String         (Maybe (Char, String)) m)

memoParsMixin :: (Monad m)
              ⇒ ParserDicts m → Gen (Parsers m)
memoParsMixin dicts = $(applyT 6) wraps where
  wraps = $(mapT 6 [| wrap |]) memos
  memos = $(mapT 6 [| memo |]) dicts
```

In $memoParsMixin$, the standard $memo$ function is applied to each of the dictionaries in the six-tuple $super$, and then $wrap$ is applied to each of the six resulting memo mixins to create a six-tuple of parsers generators.

Finally, to create a fully memoized parser, the memo mixin adapter is composed with the parser generator. After taking the fixed-point and selecting the start function, the parser is run with an empty memoization table. Since it is not generally useful to parse different strings with the same memo tables, a self-contained memoized parser is probably sufficient (but a reusable parser is easily definable).

```
memoParse :: Monad m
          ⇒ ParserDicts m → ParserT m Int
memoParse dicts =
  proj_{1/6} (fix (memoParsMixin dicts ∘ gmPars))
```

Given a concrete tuple of dictionaries, $memoParse$ creates a memoized parser for the grammar given at the beginning of the section. As usual, the supplied dictionaries determine the state monad for memoization and the state representation for the memo tables.

6.4 Running the Parser

As before, we can memoize using both maps and arrays. Although we only consider memo tables that are either all maps or all arrays, there is no requirement that they are all represented in the same way: some could be represented as maps, others as arrays, and the rest as something else entirely.

The first memoization state, $ParserMaps$, is a tuple containing six parser maps. Each tables maps a suffix of the input string to its parsed result paired with the remainder of the string. Since parsing may fail for an input, the table stores this output pair within a $Maybe$ type.

```
type ParserMaps =
  (Map String         (Maybe (Int, String)),
   Map String         (Maybe (Int, String)),
   Map String         (Maybe (Int, String)),
   Map String         (Maybe (Int, String)),
   Map (Char, String) (Maybe (Char, String)),
   Map String         (Maybe (Char, String)))

parserDicts :: ParserDicts (State ParserMaps)
parserDicts = $(mapAccT' 6 [| selMap |])

runMemoMapParse :: String → Maybe (Int, String)
runMemoMapParse s =
  evalState (runParserT (memoParse parserDicts) s)
            empties
  where
  empties = (empty, empty, empty, empty, empty, empty)
```

Alternatively, we might use arrays for the memo tables instead of maps for efficiency:

```
arrayDict :: (MArray arr (Maybe b) m)
          ⇒ (a → Int)
          → Int
          → arr Int (Maybe b)
          → Dict a b m
arrayDict f size arr = (check, store) where
  check a   = readArray arr (f a)
  store a b = writeArray arr (f a) (Just b)

newSTArray :: Ix i ⇒ (i, i) → e → ST s (STArray s i e)
newSTArray = newArray

runMemoArrParse :: String → Maybe (Int, String)
runMemoArrParse s = runST (do
```

$$
\begin{array}{l}
\textbf{let } size = length\ s \\
addA \quad\ \leftarrow newSTArray\ (0, size)\ Nothing \\
multA \quad \leftarrow newSTArray\ (0, size)\ Nothing \\
primA \quad\ \leftarrow newSTArray\ (0, size)\ Nothing \\
decimalA \leftarrow newSTArray\ (0, size)\ Nothing \\
charA \quad\ \leftarrow newSTArray\ (0, 4 * (size + 1))\ Nothing \\
anyA \quad\ \ \leftarrow newSTArray\ (0, size)\ Nothing \\
\textbf{let } acc = (\ arrayDict\ length\ size\ addA, \\
\qquad\qquad arrayDict\ length\ size\ multA, \\
\qquad\qquad arrayDict\ length\ size\ primA, \\
\qquad\qquad arrayDict\ length\ size\ decimalA, \\
\qquad\qquad arrayDict\ hash\ \ size\ charA, \\
\qquad\qquad arrayDict\ length\ size\ anyA) \\
\quad hash\ (a, b) = 4 * length\ b \\
\qquad\qquad\qquad + fromJust\ (elemIndex\ a\ \texttt{"+*()"}) \\
runParserT\ (memoParse\ acc)\ s)
\end{array}
$$

7. Performance

To evaluate performance of the monadic memo mixins, we compared the simple Fibonacci function implemented in six different ways:

zipFib The traditional hand-coded lazy data structure (Figure 1).

mixinFib/fix The Fibonacci implemented as a memo mixin as in Section 2.

mixinFib/nofix To identify the effect of explicit use of *fix* versus the hard-coded recursion, we created a version of **mixinFib/fix** in which the recursion between the mixin and Fibonacci function is defined by name references, as **tabFib/nofix**, rather than an explicit call to *fix*.

mixinFib/nofix/IO A version of **mixinFib/nofix** that uses the $Data.Array.IO$ module rather than the arrays in the ST monad. It also uses hard-coded inheritance rather than explicit fixed-points.

tabFib/nofix Hinze developed a theory of *tabulating* functions that can be used for memoization [11]. A tabulation is a lazy data structure that contains a map from an index value to a result. The concept of a tabulation is defined generically for any index type by induction on the structure of the type; it is thus *polytypic*, working on many types. We defined tabulation functions for integers, so that the results can be compared with the other implementations:

$$
\begin{array}{l}
tabulateInt\ f = map\ f\ [0\,..] \\
applyInt\ list\ n = list\ !!\ n
\end{array}
$$

These functions create a table of results of the function f and then look up the results. Tabulation reduces the Fibonacci function from exponential to quadratic execution time.

$$
\begin{array}{l}
fib\ 0 = 0 \\
fib\ 1 = 1 \\
fib\ n = tabFib\ (n-1) + tabFib\ (n-2) \\
tabFib = applyInt\ (tabulateInt\ fib)
\end{array}
$$

The memoized version is not a modular extension of an original *fib* function, because the definition of *fib* calls $memoFib$. A modular version is considered next.

tabFib/fix It is possible to express the tabulating memo function as a mixin [21]:

$$
\begin{array}{l}
memoNat = applyInt \circ tabulateInt \\
fib \qquad\ = fix\ (memoNat \circ gFib)
\end{array}
$$

The function $gFib$ used to define *fib* is the Fibonacci generator defined in Section 2.2.

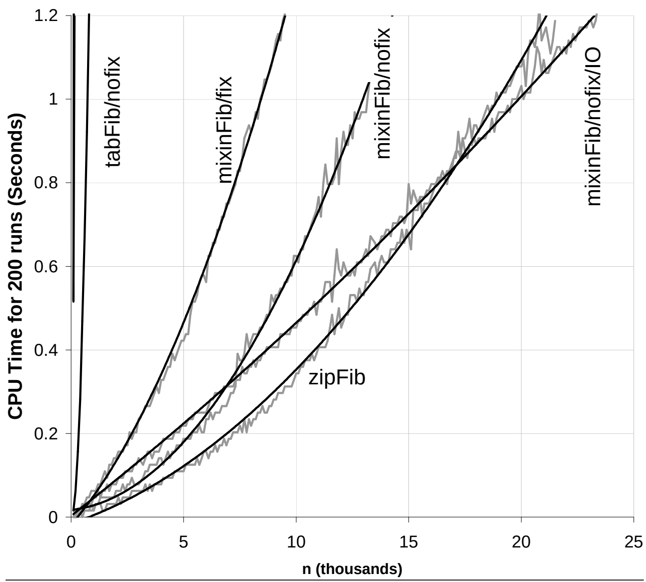

Figure 3. Performance of memoized Fibonacci (detail)

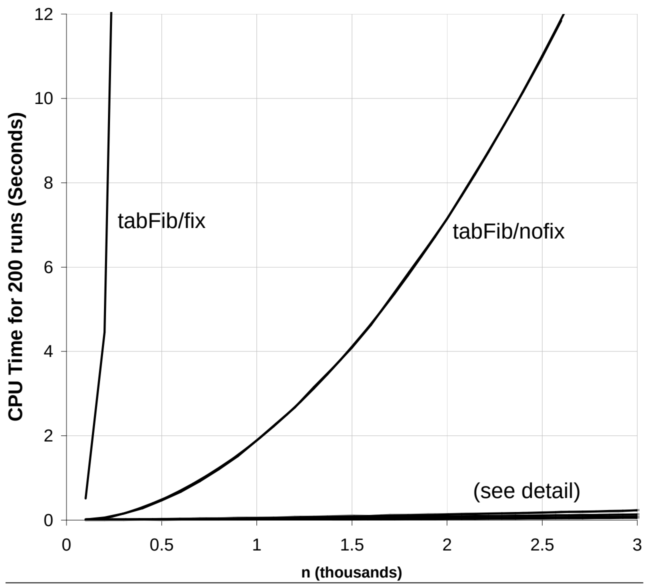

Figure 4. Performance of memoized Fibonacci (expanded view)

7.1 Performance Results

The performance of these implementations are summarized in Figures 3 & 4. The x axis defines n and the y axis is the time to compute $fib\ n$. The two charts present the same data, but with different maximum values for n and time: $(25, 1.2)$ and $(3, 12)$ respectively. Each computation is performed separately so that the memo tables for computing $fib\ \ n$ are cleared before $fib\ \ (n+1)$ is computed. The results represent the total time of of 200 runs on a 2.4Ghz Intel Pentium 4 with Windows XP. The code was compiled with optimizations (using the -O flag) in GHC 6.4.1. The performance results are relatively stable between runs.

The memoization mixins are comparable to the traditional **zipFib** version, although the latter has a slightly worse asymptotic behavior. None of the implementations exhibit true linear behavior, although the second-degree coefficients are generally small. The graphs include second-degree polynomial trend lines which match the curves closely. Interestingly, when run without optimizations in GHCi the trends are more clearly linear. The implementation using IO arrays is the closest to linear.

One clear source of overhead is the use of explicit fixed points to achieve modular memoization. For the monadic memoization mixins, **mixinFib/fix** is on average 2 times slower than **mixinFib/nofix**. This is a very significant penalty on modularity. In object-oriented languages, the performance hit of virtual methods versus non-virtual methods is typically just a few additional memory accesses.

As Hinze noted, the tabulating version, **tabFib/nofix**, is quadratic in n. This is because it traverses the list multiple times to look up cached values. Its performance is more easily seen in Figure 4. It is approximately 200 times slower than the other **zipFib**. The slower modular version, **tabFib/fix**, might be slow because it's creating a new memo table at each unrolling of the fixed-point.

Our initial investigations show that the parser performs badly when compared to the hand-written tabulating parser in [7]. Until the performance penalty for modularity is addressed, the monadic memoized parser will remain an order of magnitude slower than the lazy implementation.

8. Related Work

Memoization is an old technique [19]. Most accounts introduce a higher-order $memo$ function to perform memoization. This approach can be implemented in procedural languages that support higher-order functions [10]. A $memo$ function can also be implemented as a primitive within the implementation of a lazy functional language. For example, some versions of the Glasgow Haskell Compiler system included a $memo$ function, but it appears to have been removed from recent versions [8]. Memoization can also be defined in terms of more basic primitives which allow effects outside the normal semantics of functional languages; for example, memoization can be defined on top of an unsafe state monad [3] or unsafe IO monad [15].

The current work was motivated by a desire to develop a simpler implementation of Packrat parsers [7]. The presentation given by Ford uses Haskell's internal memoization of recursive lazy data structures. Using this approach requires rewriting the parser to use a special memoization data structure; the transformation is quite complex. While some of the complexity can be hidden from users of the library who write grammars, not all of it can: users creating a parser must also create the corresponding lazy data structures.

A similar approach to parsing was presented by Norvig [20]. He used a procedural language in which $memo$ could be defined as a macro. Johnson extends this method to work with left-recursive grammars by transforming the parser to continuation-passing style [13].

Hinze [11] defines tabulation functions that store previous results in lazy data structures. He also transforms functions, but only needs to open recursion, not apply monadification. As a result, the technique of tabulation is easier to use than monadic memoization mixins. However, it does not produce as efficient a result. For Fibonacci, the result is a quadratic function, not the linear function in Figure 1.

Swadi et al. [23] independently proposed an approach to memoization similar to the one presented in this paper. However, their goal is different, leading them to significantly different solutions. Their primary goal is to perform partial evaluation on memoized computations to increase performance. They develop a monadic memoization combinator that is similar, but not identical, to the first one given here. Instead of creating a memoization mixin, they create a memoizing fixed-point combinator. This obscures the connection to inheritance and mixins in object-oriented languages. They do this in part because they are working in an applicative language, rather than a lazy one. They express partial evaluation via explicit *staging* of the computation. When the memoized computation is staged in an applicative language, computations are duplicated which negate the benefit of memoization. To avoid recomputation they transform the code to continuation-passing style (CPS), and then define a memoizing fixed-point combinator for this style. The staged version of the memoized function performs significantly better than the unstaged version. The work presented here has a different goal: to develop memoization techniques useful for a variety of situations, including functional and monadic computations. We base our approach on the well-known concepts of mixins and inheritance that support flexible and modular programming. They mention the situation of mutual recursion, and outline a similar solution using accessor functions. We take this approach slightly further by using template metaprogramming to generate the accessor functions automatically. Our approach does not require staging or continuations to avoid recomputation. Since the two approaches have a common starting point, it may be possible to combine them.

The memoization technique described here depends upon monadification — introducing an unbound monad parameter into a recursive function. A comprehensive review of this problem and its solution was presented by Erwig in [6], although the problem was discussed earlier [16].

Memoization by lazy data structures appears to be unique to lazy functional programming languages. The technique is powerful, but difficult to use because it is almost invisible in the resulting program [1]. The technique relies on structures implicitly maintained by the Haskell runtime that cannot be manipulated by the programmer; for example, there is no simple way to shrink the memo table in the canonical memoization of fib presented in the introduction. This approach also depends upon the implementation of the language; it can be difficult to determine exactly which situations will be memoized. These optimizations are not inherent in the underlying λ-calculus. Precise specification of lazy evaluation is a difficult problem [17].

Cook and Lauchbury studied "disposable memo functions", where the memo table can be garbage collected when it is no longer referenced [3]. They present an extension to λ-calculus with a primitive $memo$ function. They also discuss briefly the basic solution to memoization using a state monad. They show how the behavior of the extended λ-calculus can be implemented using $unsafeST$, which allows update of mutable state to be hidden within a Haskell program. They also discuss applications to parsers. Our results refute the claim that "in Haskell, $memo$ must be defined outside the of the language". The issue of disposing memo tables does not arise in the solution presented here because the memo table is an explicit value and not hidden by implementation primitives. Having an explicit memo table also allows it to be pruned, shared, or written to disk just like any other value. The technique of lazy data structures supports disposable memoization, although it requires that lazy data structure and its corresponding functions go out of scope.

9. Conclusion

This paper presents monadic memoization mixins. The work had two main motivations. One was to illustrate the use of inheritance in functional languages. Inheritance is usually associated with classes in object-oriented languages, but is in fact a general technique for modifying recursive structures — classes, modules, functions, types, etc. The other motivation was to develop a modular technique for memoization in pure functional languages and a simpler approach than Packrat parsers to memoizing top-down parsers built from combinator libraries. We developed techniques for memoizing recursive functions, mutually recursive functions, monadic functions, and general constructs that encapsulate a functional behavior. We evaluated the performance of various approaches to memoization in Haskell and showed that ours is efficient for a small example. Memoizing mutually recursive functions is awkward because Haskell lacks generic functions for manipulating tuples and records; Template Haskell provides some help, although language extensions might be more effective.

References

[1] Richard Bird and Ralf Hinze. Functional pearl: Trouble shared is trouble halved. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 1–6, New York, NY, USA, 2003. ACM Press.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 303–311, 1990.

[3] Byron Cook and John Launchbury. Disposable memo functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-97)*, volume 32,8 of *ACM SIGPLAN Notices*, pages 310–310. ACM Press, 1997.

[4] William Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[5] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 433–444, 1989.

[6] Martin Erwig and Deling Ren. Monadification of functional programs. *Sci. Comput. Program.*, 52(1-3):101–129, 2004.

[7] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[8] GHC – The Glasgow Haskell Compiler. `haskell.org/ghc`.

[9] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[10] M. Hall and J. Mayfield. Improving the performance of ai software: Payoffs and pitfalls in using automatic memoization. In *International Symposium on Artificial Intelligence*, 1993.

[11] Ralf Hinze. Memo functions, polytypically! In *Second Workshop on Generic Programming*, 2000.

[12] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1998.

[13] Mark Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417, 1995.

[14] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

[15] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In *Implementation of Functional Languages*, pages 37–58, 1999.

[16] R. Lämmel. Reuse by Program Transformation. In Greg Michaelson and Phil Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop.

[17] John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

[18] Daan Leijen. Parsec, a fast combinator parser. `http://www.cs.uu.nl/daan`.

[19] Donald Michie. "memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[20] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1), 1991.

[21] Anonymous ICFP reviewer. Private communication.

[22] Tim Sheard and Simon L. Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, pages 1–16, October 2002.

[23] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 160–169, New York, NY, USA, 2006. ACM Press.