Function Inheritance: Monadic Memoization Mixins

Daniel Brown[1], **William R. Cook**[2]

[1]Northeastern University, Boston, MA USA

[2]University of Texas at Austin, Austin, TX USA

Abstract. *Inheritance is a mechanism for incrementally modifying recursive definitions. While inheritance is typically used in object-oriented languages, inheritance also has something to offer to functional programming. In this paper we illustrate the use of inheritance in a pure functional language by developing a small library for memoization. We define* monadic memoization mixins *that compose—via inheritance—with an ordinary monadic function to create a memoized version of the function. A comparison of the performance of different approaches shows that memoization mixins are efficient for a small example.*

1. Introduction

Inheritance is usually understood as specific to object-oriented programming and classes. However, at its core inheritance is a general mechanism for incrementally modifying recursive structures [Cook 1989]. Classes are one common example of recursive definitions, but inheritance also applies to types [Canning et al. 1989], functions, and modules. In this paper we illustrate the use of inheritance for functions by showing how it can be used to implement memoization in the pure functional programming language Haskell.

Memoization can be implemented in many ways. A function can be memoized by rewriting it to explicitly maintain its own memo table, but rewriting many functions this way is tedious and fails localize the common memoization behavior, which could otherwise allow modular memoization strategies. Some languages include a primitive operator $memo$, but this is beyond the user's control. Lazy functional languages indirectly support a form of memoization as a side effect of the lazy evaluation strategy.

In procedural languages memoization can be implemented as a user-defined, higher-order procedure $memo$ which produces a memoized version $memo\ f$ of a function f, but $memo$ cannot be written in a pure functional language since its internal memo table relies on side effects. Moreover, naïvely applying $memo$ to a recursive function fails to memoize recursive calls within the function.

In this paper we present *monadic memoization mixins* that compose with a recursive function using inheritance to memoize its recursive calls. The goal of this work is to illustrate a novel use of inheritance, not to develop a high-performance memoization implementation. However, we do show that performance is acceptable. In an earlier report [Brown and Cook 2007] we scaled our technique up to memoize a parser combinator library, but the details are beyond the scope of the current paper.

2. Memoizing Recursive Functions

Memoizing a simple recursive function nicely illustrates the technique of monadic memoization mixins. The underlying ideas behind this approach are well known, but they have not been systematically studied in the context of mixins and monads in pure functional languages.

Consider the Fibonacci function. The execution time for $\mathit{fib}\ n$ grows exponentially in n because fib is called exponentially often on small inputs. While it is possible to rewrite fib to include a memo table, 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.

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

2.1. Monadification

Rather than rewrite fib to explicitly use a state monad, fib can first be rewritten in monadic style over an arbitrary monad parameter. This is an automatable process called *monadification* [Erwig and Ren 2004]. This monad "hole" can then be plugged with a state monad that carries a memo table through the recursive calls of the function. The monadic version mFib of fib computes over an arbitrary monad m:

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

Monadifying a function forces the recursive calls to occur in sequence, 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. $\mathit{runIdentity} :: \mathit{Identity}\ a \to a$ 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 $\mathit{Identity}\ \mathit{Int}$.

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

2.2. Open Recursion and Inheritance

The monadified version of fib enables a memo table to be threaded through the recursive computation, but to also memoize recursive calls to fib, the self-reference must be exposed, or opened, so that it can be rebound to refer to the memoized version of the function. This is exactly what *inheritance* does in object-oriented languages [Cook and Palsberg 1989]—and the same technique can be applied to functions [Cook 1989]. To do so, we abstract the self-reference in fib as an explicit self parameter, then reconstruct fib_g using an explicit fixed point.

Functions like $gFib$ that are used to specify a fixed-point are called *generators*. They have types of the form $a \to a$, described by the type $Gen\ a$.

$$\begin{aligned}
&\mathbf{type}\ Gen\ a = a \to a\\
&gFib :: Gen\ (Int \to Int)\\
&gFib\ self\ 0 \quad\ = 0\\
&gFib\ self\ 1 \quad\ = 1\\
&gFib\ self\ (n+2) = self\ n + self\ (n+1)\\
&fib_g :: Int \to Int\\
&fib_g = fix\ gFib
\end{aligned}$$

Inheritance works by composing generators before computing the fixed point. For memoization, the memoized Fibonacci function will have the form $fix\ (memo \circ gFib)$ for an appropriate generator $memo$ with $gFib$. This has the effect of binding self-reference in $gFib$ to the memoized version of the function. In this context, $memo$ is a *mixin* [Bracha and Cook 1990].

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 $fix\ (memo \circ gFib_0)$ where $gFib_0$ is the generator of fib. There is a significant performance penalty for using explicit fixed-points in Haskell to implement inheritance, but 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:

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

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 an entry in the table and, if so, returns the stored result. If not, it computes $f(x)$ and 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 calls), and the updated table is returned with the 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

```
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 1. Memoization Mixin

has already been computed, and to *store* new values that are computed. These two functions constitute a dictionary interface $Dict\ a\ b\ m$, where a is the key type, b is the value type, and m is the state-like monad:

```
type Dict a b m = (a → m (Maybe b), a → b → m ())
```

Given a dictionary, the *memo* mixin is easily defined, as shown in Figure 1. Following a convention from object-oriented programming [Goldberg and Robson 1983], the argument of the mixin is called *super*.

While it is desirable to encapsulate *check* and *store* within a type class for memo tables and stateful monads, this approach does not work nicely in cases where multiple dictionaries have the same type.

2.5. Memoized Fibonacci

Finally, the *memo* mixin is combined with the monadic generator for *fib* to create the memoized 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 of type $a \to b$, abstracted over a memo dictionary. One way to instantiate the dictionary's memo 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 could reuse the same table across separate uses of the function, if desired. On the other hand, if the client does not 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 improved efficiency, the memo table might instead be implemented as an array, which Haskell provides a variety of. 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 generic 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

Effectful computations in Haskell are often structured using monads. Since a monadic function is just a pure function from input values to output computations, we can reuse the approach in the previous section to memoize functions that are already monadic: the memo table will just map inputs to computations delivering outputs.

But this memo table is not always the one we want. For example, tables for stateful computations will map inputs a to state transformers $s \to (b, s)$, failing to use the input state s in the cache lookup. Tables for the reader monad have the same problem. But sometimes caching the output computation is exactly what we want:

$a \to \mathit{Maybe}\ b$	caches the successful result $\mathit{Just}\ b$ or failure $\mathit{Nothing}$
$a \to \mathit{Error}\ e\ b$	caches the successful result $\mathit{Right}\ b$ or error $\mathit{Left}\ e$
$a \to [b]$	caches the sequence of possible results $[b]$
$a \to \mathit{Writer}\ w\ b$	caches the accumulated output w along with the result b

Since Haskell represents effects explicitly, our mixin-based approach to memoization immediately applies to these kinds of effectful computations. In this section we illustrate how to memoize a nondeterministic computation expressed in the list monad.

First consider a function that computes the fringe of a tree: given a tree with values at its leaves, *fringe* computes the left-to-right traversal of the leaves.

$$\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\ \mathbf{deriving}\ (\mathit{Show}, \mathit{Eq})\\
&\mathit{fringe} :: \mathit{Tree}\ a \to [a]\\
&\mathit{fringe}\ (\mathit{Leaf}\ a) \quad = [a]\\
&\mathit{fringe}\ (\mathit{Fork}\ t\ u) = \mathit{fringe}\ t +\!\!+ \mathit{fringe}\ u
\end{aligned}$$

Now consider the preimage of *fringe*: *unfringe* maps a list representing the fringe of a tree to the set of trees with that fringe. Since *unfringe* computes a set of possible trees, it is naturally expressed as a nondeterministic computation using the list monad. It maps singleton lists to leaves, and it maps larger lists to a set of trees by recurring on all binary partitions of the list and combining every pair of resulting trees with *Fork*:

$$\begin{aligned}
&\mathit{unfringe} :: [a] \to [\mathit{Tree}\ a]\\
&\mathit{unfringe}\ [a] = [\mathit{Leaf}\ a]\\
&\mathit{unfringe}\ as\ = \mathbf{do}\ \{(l, k) \leftarrow \mathit{partitions}\ as; t \leftarrow \mathit{unfringe}\ l; u \leftarrow \mathit{unfringe}\ k;\\
&\qquad\qquad\qquad \mathit{return}\ (\mathit{Fork}\ t\ u)\}
\end{aligned}$$

We use the function *partitions* which computes all of the binary partitions of a list:

$$\begin{aligned}
&\mathit{partitions} :: [a] \to [([a], [a])]\\
&\mathit{partitions}\ as = [\mathit{splitAt}\ n\ as \mid n \leftarrow [1\,..\,\mathit{length}\ as - 1]]
\end{aligned}$$

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. Notice that monadification introduces a monad m separate from the existing list monad: the nondeterminism effect provided by the list monad and the stateful effect required by memoization are independent and require different monads. The result, *gmUnfringe*, is openly recursive via *self* and is parameterized over a monad m:

$$\begin{aligned}
&\mathit{gmUnfringe} :: \mathit{Monad}\ m \Rightarrow \mathit{Gen}\ ([a] \to m\ [\mathit{Tree}\ a])\\
&\mathit{gmUnfringe}\ \mathit{self}\ [a] = \mathit{return}\ [\mathit{Leaf}\ a]
\end{aligned}$$

```
gmUnfringe self as =
  liftM concat (sequence (do
    (l, k) ← partitions as
    return (do {ts ← self l; us ← self k;
                return (do {t ← ts; u ← us; return (Fork t u)})})))
```

The monadic generator $gmUnfringe$ can be run without memoization 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)
runMUnfringe :: Ord a ⇒ [a] → [Tree a]
runMUnfringe l = evalState (memoUnfringe mapDict l) empty
```

Obtaining $gmUnfringe$ from $unfringe$ is straightforward, but it produces a function that is difficult to understand because it is written in two monads, $List$ and m, whose use must be interleaved. One way to avoid this interleaving is to combine the two monads into one and express $gmUnfringe$ more uniformly in the combined monad. In the next section we show how to achieve this using monad transformers.

4. Memoization via Monad Transformers

Monadifying a function that is already written in monadic style produces code that is difficult to understand because it interleaves uses of two different monads. In the last section, we identified certain monads where our approach from Section 2 properly memoizes functions with that effect—$Maybe$, $Error\ e$, $List$, and $Writer\ w$—and in this section we show how using the corresponding monad transformers is an elegant alternative to modification. We continue the $unfringe$ example from the last section to illustrate our approach with the $ListT$ monad transformer[1].

We obtained the monad parameter m in $gmUnfringe$ by monadifying the list computation $unfringe$. A more elegant way to introduce a new monad into an already monadic computation is to use monad transformers [Liang et al. 1995]: lift $unfringe$ from the list monad to the transformed monad $ListT\ m$, leaving m to be bound to the memoization monad when composed with an appropriate memo mixin. This produces the function $gmUnfringeT$:

```
gmUnfringeT :: Monad m ⇒ Gen ([a] → ListT m (Tree a))
gmUnfringeT self [a] = return (Leaf a)
```

[1] $ListT$ fails to be a proper monad transformer; we discuss this at the end of the section.

```
gmUnfringeT self as = do
  (l, k) ← ListT (return (partitions as))
  t      ← self l
  u      ← self k
  return (Fork t u)
```

The benefit of using monad transformers is that $gmUnfringeT$ is defined similarly to $unfringe$ and avoids the interleaving of the two monads found in $gmUnfringe$.

A false start to defining an appropriate memo mixin for transformer-based functions is to simply lift the memo table operations into the transformed monad:

```
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 only memoizes the first result of the computation, ignoring the rest. The problem is evident in the type of $memo_X$: the memo table defined by $Dict\ a\ b\ m$ stores values of type b instead of computations $[b]$.

This problem arises because the transformed monad interleaves the effects of the two monads: the call to $super\ a :: t\ m\ b$ delivers a result of type b from the monad $t\ m$, whereas we want a result of type $m\ b$ from the monad embodied by the transformer t. Thus, we must shift perspective from transformed monad $t\ m$ to the monad n, where $t\ m\ a \cong n\ (m\ a)$—although many monad transformers are not of this shape, those for the monads we address in Section 3 all are. We encode this shift with a type class that maps between a monad 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)
```

The type class $TransForMonad$ uses a common Haskell extension called *functional dependencies* [Jones 2000]: the clause $t \to n$ specifies that the type t uniquely determines n in instances of this class, which the type checker will use to infer types for functions like $fromTrans$ where n does not occur in the function's input types. Since we also have the dependency $n \to t$, $TransForMonad$ expresses a bijection between certain monad transformers and their underlying monads. Most instance of this class simply add and remove datatype constructors:

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

$TransForMonad$ enables us to define the transformer-based mixin. We use the original memo mixin as a generator $Gen\ (a \to n\ (m\ b))$ so that it will capture results of type $m\ b$, and we then wrap it to obtain a generator $Gen\ (a \to t\ m\ b)$:

$$\begin{aligned}
&memo_T :: (TransForMonad\ t\ n, Monad\ m) \Rightarrow \\
&\qquad Dict\ a\ (n\ b)\ m \to Gen\ (a \to t\ m\ b) \\
&memo_T\ dict\ f = toTrans \circ memo\ dict\ (fromTrans \circ f)
\end{aligned}$$

The transformer-based mixin memoizes transformer-based functions, as before:

$$\begin{aligned}
&\mathbf{type}\ MemoizedT\ a\ n\ t\ b\ m = Dict\ a\ (n\ b)\ m \to a \to t\ m\ b \\
&memoUnfringeT :: Monad\ m \Rightarrow MemoizedT\ [a]\ []\ ListT\ (Tree\ a)\ m \\
&memoUnfringeT\ dict = fix\ (memo_T\ dict \circ gmUnfringeT) \\
&runMUnfringe_T :: Ord\ a \Rightarrow [a] \to [Tree\ a] \\
&runMUnfringe_T\ a = evalState\ (runListT\ (memoUnfringeT\ mapDict\ a))\ empty
\end{aligned}$$

Throughout this section we have been abusing the $ListT$ "monad transformer" because the functor $ListT\ \ m$ is only a monad when m is a *commutative* monad [Jones and Duponcheel 1993], but the particular instances we have in mind for m—a state-like monad—are usually not commutative. However, the "memoization effect" does not necessarily bring along the full functionality of state—and, intuitively, it appears to be commutative: switching the order in which the results of a function are cached makes no semantic difference. So we expect that our $Dict$ abstraction can be refined into a proper axiomatization of a commutative "memoization monad", but we leave this as future work.

5. Memoizing Mutual Recursion

Our approach so far handles recursive functions, possibly written in monadic style, but computations are often spread across multiple, mutually recursive functions. Memoizing mutually recursive functions requires maintaining a collective state with a memo table for each function—all of which might have different types. In this section we extend our technique to apply to a pair of mutually recursive, non-monadic functions; extending to many monadic functions would then be straightforward.

Consider mutually recursive functions f and g (with different types):

$$\begin{aligned}
&f :: Int \to (Int, String) \\
&f\ 0 \qquad\quad = (1, \texttt{"+"}) \\
&f\ (n+1) \quad = (g\ (n, fst\ (f\ n)), \texttt{"-"} +\!\!+ snd\ (f\ n)) \\
&g :: (Int, Int) \to Int \\
&g\ (0, m) \quad\ = m + 1 \\
&g\ (n+1, m) = fst\ (f\ n) - g\ (n, m)
\end{aligned}$$

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 $(Int \to (Int, String), (Int, Int) \to Int)$. As in the case for *fib*, these non-monadic functions must be monadified to introduce a monad parameter. The result is a generator on a pair of functions monadified over m:

$$
\begin{aligned}
&\textbf{type } MFuns\ m = (Int \to m\ (Int, String), (Int, Int) \to m\ Int)\\
&gmFG :: Monad\ m \Rightarrow Gen\ (MFuns\ m)\\
&gmFG\ \sim(f, g) = (f', g')\ \textbf{where}\\
&\quad f'\ 0 \qquad\quad = return\ (1, \texttt{"+"})\\
&\quad f'\ (n+1) \quad = \textbf{do}\ \{a \leftarrow f\ n; b \leftarrow g\ (n, fst\ a); return\ (b, \texttt{"-"} +\!\!+ snd\ a)\}\\
&\quad g'\ (0, m) \qquad = return\ (m+1)\\
&\quad g'\ (n+1, m) = \textbf{do}\ \{a \leftarrow f\ n; b \leftarrow g\ (n, m); return\ (fst\ a - b)\}
\end{aligned}
$$

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 *lazy pattern* $\sim(f, g)$ prevents 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 update functions, so we settle here for tuples.

The function f is easily recovered from $gmFG$ by taking its fixed-point, projecting, and running the result through the identity monad. The resulting function f_{gm} behaves the same as f.

$$
\begin{aligned}
&f_{gm} :: Int \to (Int, String)\\
&f_{gm} = runIdentity \circ (fst\ (fix\ gmFG))
\end{aligned}
$$

To memoize f and g, a memoization mixin must be composed with each generator separately, yet each memo mixin must read and write to disjoint parts of the same state. The memo mixin for f and g is a function on pairs, which uses a pair of dictionaries to access the store:

$$
\begin{aligned}
&memoFGMixin :: (Monad\ m, Monad\ m') \Rightarrow\\
&\qquad\qquad (Dict\ a\ b\ m, Dict\ a'\ b'\ m') \to Gen\ (a \to m\ b, a' \to m'\ b')\\
&memoFGMixin\ (df, dg)\ (f, g) = (memo\ df\ f, memo\ dg\ g)\\
&memoF :: Monad\ m \Rightarrow\\
&\qquad (Dict\ Int\ (Int, String)\ m, Dict\ (Int, Int)\ Int\ m)\\
&\qquad\quad \to Int \to m\ (Int, String)\\
&memoF\ dicts = fst\ (fix\ (memoFGMixin\ dicts \circ gmFG))
\end{aligned}
$$

One method to represent memo tables for mutual recursion is to maintain a map for each function, providing access to the ith map by projection and injection functions:

$$\textbf{type } Accessor\ a\ b = (b \to a, a \to b \to b)$$

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

$$
\begin{aligned}
&\mathit{selMap} :: \mathit{Ord}\ a \Rightarrow \mathit{Accessor}\ (\mathit{Map}\ a\ b)\ s \rightarrow \mathit{Dict}\ a\ b\ (\mathit{State}\ s)\\
&\mathit{selMap}\ (\mathit{proj}, \mathit{inj}) = (\mathit{check}, \mathit{store})\ \mathbf{where}\\
&\quad \mathit{check}\ a\ \ = \mathit{gets}\ (\mathit{lookup}\ a \circ \mathit{proj})\\
&\quad \mathit{store}\ a\ b = \mathit{modify}\ (\lambda s \rightarrow \mathit{inj}\ (\mathit{insert}\ a\ b\ (\mathit{proj}\ s))\ s)
\end{aligned}
$$

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, and let $acc_{i/n} = (proj_{i/n}, inj_{i/n})$. (These could easily be written by hand or generated with metaprogramming.)

$$
\begin{aligned}
&\mathit{runMemoF} :: \mathit{Int} \rightarrow (\mathit{Int}, \mathit{String})\\
&\mathit{runMemoF}\ n = \mathit{evalState}\ (\mathit{memoF}\ \mathit{dicts}\ n)\ (\mathit{empty}, \mathit{empty})\\
&\quad \mathbf{where}\ \mathit{dicts} = (\mathit{selMap}\ \mathit{acc}_{1/2}, \mathit{selMap}\ \mathit{acc}_{1/2})
\end{aligned}
$$

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

$$\mathbf{type}\ \mathit{MemoFG} = \mathit{State}\ (\mathit{Map}\ \mathit{Int}\ (\mathit{Int}, \mathit{String}), \mathit{Map}\ (\mathit{Int}, \mathit{Int})\ \mathit{Int})$$

6. Evaluation

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

zipFib	The traditional hand-coded lazy data structure
mixinFib/fix	Fibonacci implemented as a memo mixin as in Section 2
mixinFib/nofix	**mixinFib/fix** with harded-coded fixed-point
mixinFib/nofix/IO	**mixinFib/nofix** with $Data.Array.IO$ instead of ST
tabFib/nofix	Lazy *tabulating* function without mixins [Hinze 2000]

The performance of these implementations is summarized in Figure 2. The vertical axis is the time to compute $fib\ n$ where n increases linearly along the horizontal axis. Each computation is performed separately so that the memo tables are reconstructed each time. The results represent the total time of 200 runs on a 2.4GHz Intel Pentium 4 running Windows XP. The code was compiled with optimizations (-O) 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. Curiously, when run without optimizations in `ghci` the trends are more clearly linear. The implementation using IO arrays is the closest to linear.

In [Brown and Cook 2007] we also applied our techniques to memoize a monadic parser so we could construct Ford's Packrat parsers [2002] in a more modular way. This case study was interesting because parsers produce large nests of mutually

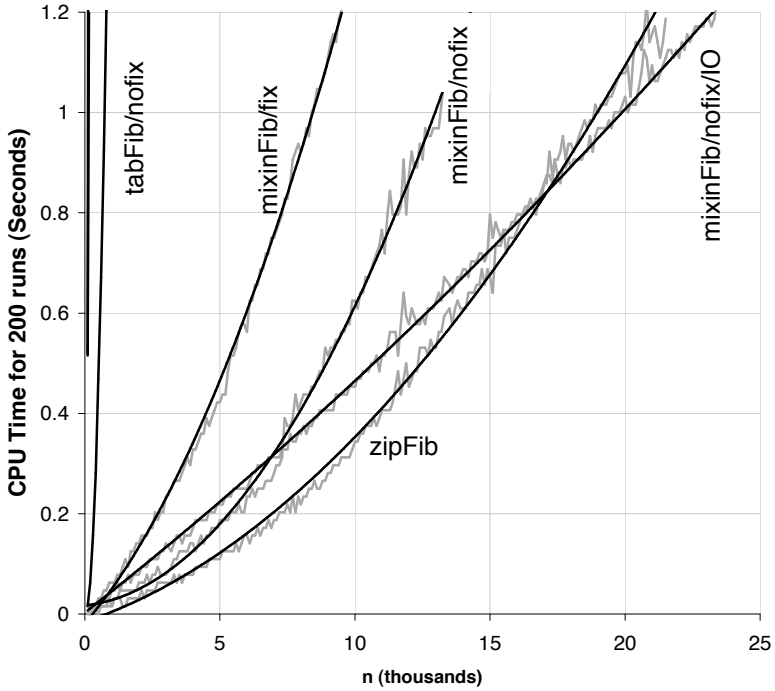

Figure 2. Performance of memoized Fibonacci (detail)

recursive functions and monadic parsers typically combine the state monad with failure or nondeterminism. As discussed in Section 3, additional work is needed to develop a general approach to memoizing computations in the state monad because our method does not make the input state available to the memo mixin. To work around this, we devised an ad-hoc method to expose the input states, and the resulting memoization mixin works but is not satisfactorily elegant or general—see the report for details. We think that broadening our technique to monads beyond those mentioned in Section 3 is a good challenge problem for future research.

7. Related Work

Memoization is an old technique [Michie 1968]. Most accounts introduce a higher-order function $memo$ to perform memoization. This approach can be implemented in procedural languages that support higher-order functions [Hall and Mayfield 1993]. A $memo$ function can also be implemented as a primitive within the implementation of a lazy functional language. For example, some versions of GHC included a $memo$ function, but it appears to have been removed from recent versions [GHC]. 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 [Cook and Launchbury 1997] or unsafe IO monad [Jones et al. 1999].

Hinze [2000] 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, his technique of tabulation is easier to use than monadic memoization mixins, but it produces a less efficient result.

Swadi et al. [2006] independently proposed an approach to memoization similar to the one presented in this paper (and our original report). However, they reached a significantly different solution because their primary goal was to perform partial evaluation on memoized computations to increase performance. Instead of creating a memoization mixin, they create a memoizing fixed-point combinator.

The memoization technique described here depends upon monadification, which introduces a monad parameter into a recursive function. A comprehensive review of this problem and its solution was presented by Erwig and Ren [2004] although the problem was discussed earlier [Lämmel 2000].

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 [Bird and Hinze 2003]. There is no simple way to shrink the memo table in the canonical memoization of fib presented in the introduction.

Cook and Lauchbury [1997] studied "disposable memo functions" where the memo table can be garbage collected when it is no longer referenced. They present an extension to a λ-calculus with a primitive $memo$ function. They also briefly discuss 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. Our results refute their claim that "in Haskell, $memo$ must be defined outside the of the language".

8. Conclusion

This paper presents monadic memoization mixins. The main motivation for our work was to illustrate the use of inheritance in functional languages. Inheritance is usually associated with classes in object-oriented languages, but it is in fact a more general technique for modifying recursive structures—including classes, modules, functions, and types. Our monadic memo mixins memoize recursive functions, monadic functions, and mutually recursive functions, and they are efficient for a small example.

References

Bird, R. and Hinze, R. (2003). Functional pearl: Trouble shared is trouble halved. In *ACM SIGPLAN Workshop on Haskell*, pages 1–6.

Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 303–311.

Brown, D. and Cook, W. R. (2007). Monadic memoization mixins. Technical Report TR-07-11, UT Austin, Department of Computer Sciences.

Canning, P., Cook, W., Hill, W., Mitchell, J., and Olthoff, W. (1989). F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280.

Cook, B. and Launchbury, J. (1997). Disposable memo functions. In *International Conference on Functional Programming (ICFP)*, pages 310–310.

Cook, W. (1989). *A Denotational Semantics of Inheritance*. PhD thesis, Brown.

Cook, W. and Palsberg, J. (1989). A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 433–444.

Erwig, M. and Ren, D. (2004). Monadification of functional programs. *Sci. Comput. Program.*, 52(1-3):101–129.

Ford, B. (2002). Packrat parsing: simple, powerful, lazy, linear time. In *International Conference on Functional Programming (ICFP)*, pages 36–47.

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

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley.

Hall, M. and Mayfield, J. (1993). Improving the performance of AI software: Payoffs and pitfalls in using automatic memoization. In *International Symposium on Artificial Intelligence*.

Hinze, R. (2000). Memo functions, polytypically! In *Second Workshop on Generic Programming*.

Jones, M. P. (2000). Type classes with functional dependencies. In *European Symposium on Programming Languages and Systems (ESOP)*, pages 230–244.

Jones, M. P. and Duponcheel, L. (1993). Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University.

Jones, S. L. P., Marlow, S., and Elliott, C. (1999). Stretching the storage manager: Weak pointers and stable names in haskell. In *Implementation of Functional Languages*, pages 37–58.

Lämmel, R. (2000). Reuse by Program Transformation. In Michaelson, G. and Trinder, P., editors, *Functional Programming Trends 1999*. Intellect. Selected papers from the 1st Scottish Functional Programming Workshop.

Liang, S., Hudak, P., and Jones, M. P. (1995). Monad transformers and modular interpreters. In *Principles of Programming Languages (POPL)*, pages 333–343. ACM.

Michie, D. (1968). "memo" functions and machine learning. *Nature*, 218:19–22.

Swadi, K., Taha, W., Kiselyov, O., and Pasalic, E. (2006). A monadic approach for avoiding code duplication when staging memoized functions. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 160–169.