Monadic memoisation of non-deterministic left-recursive computations in Haskell

Samer Abdallah

August 23, 2017

Abstract

This technical note describes a Haskell implementation of a monadic memoising combinator that supports non-nondeterminism and recursion of the sort required to implement memoising parser combinators for left recursive grammars. It is a direct port of the OCaml implementation, presented by Abdallah (2017a)—in turn based on the work of Johnson (1995)—but using the standard Haskell library to provide monads and monad transformers for non-determinism, mutable state references, continuation capture, and fixed points for building recursive computations.

1 The code

The code is presented in Fig. 1. *Control.Monad.Cont* provides the continuation monad transformer *ContT*. *Control.Monad.ST* provides a monad supporting references to mutable cells; these are a required to manage the memo tables. *Control.Monad.Ref* provides *MonadRef*, a type class which presents a uniform interface to monads (such as *ST*) supporting references to mutable cells. There are several implementations of *MonadRef* in the Haskell package repository *Hackage*. The code presented here requires the one provided by the package *ref-fd*. *Data.Foldable* provides a type class which presents a uniform interface to data structures that can be folded over, including lists, sets, and associative maps. *Data.Map* and *Data.Set* provide data structures for associative maps and sets respectively.

The core of this approach is the type definition on line 10: the type constructor *NDC s n r* represents a monad supporting three computational effects: (1) a supply of references to mutable cells, ultimately provide by the *ST* monad; (2) non-determinism, provided by any instance *n* of *MonadPlus*; and (3) the ability to capture the continuation of this computation. This is done using the continuation monad transformer *ContT* with the base monad set to *ST s* (where *s* is phantom type parameter used by the *ST* monad to enforce correct scoping) and the answer type set to *n r*, where *n* can be any instance of *MonadPlus*. We can begin to understand how this is achieved by unpacking the type implied by applying the type constructor *ContT* to these particular arguments: after stripping out unimportant field names, it amounts to

$$NDC\ s\ n\ r\ a \approx (a \to ST\ s\ (n\ r)) \to ST\ s\ (n\ r)$$

```
module CPSMemo where

import Control.Monad.Cont
import Control.Monad.ST
import Control.Monad.Ref
import Data.Foldable
import qualified Data.Map as Map
import qualified Data.Set as Set

type NDC s n r = ContT (n r) (ST s)
type NDCK s n r a b = a → NDC s n r b
type Table a b = Map.Map a (Set.Set b)

instance MonadPlus n ⇒ MonadPlus (NDC s n r) where
  mzero = ContT {runContT = λ_ → return mzero}
  mplus f g = ContT {runContT = λk → liftM2 mplus (runContT f k) (runContT g k)}

run :: MonadPlus n ⇒ NDC s n a a → ST s (n a)
run m = runContT m (return . return)

memo :: (MonadPlus n, Ord a, Ord b) ⇒
         (NDCK s n r a b) → ST s (ST s (Table a b), NDCK s n r a b)
memo f = do
  loc ← newRef Map.empty
  let feed x table k = do
      let update e t = writeRef loc (Map.insert x e t)
      let consumer (res,conts) = do
          update (res, k:conts) table
          foldr' (mplus . k) mzero res
      let producer = do
          update (Set.empty, [k]) table
          y ← f x
          table' ← readRef loc
          let Just (res,conts) = Map.lookup x table'
          if Set.member y res then mzero
          else update (Set.insert y res, conts) table' ≫
               foldr' (λk → mplus (k y)) mzero conts
      maybe producer consumer (Map.lookup x table)
  return (readRef loc ≫= return . fmap fst,
          λx → readRef loc ≫= callCC . feed x)
```

Figure 1: Complete code for memoising non-deterministic and recursive (including left-recursive) monadic computations. All of the required library modules are automatically included in most installations of GHC, the Glasgow Haskell Compiler, except for *Control.Monad.Ref*, which should be installed from the Hackage package *ref-fd*. Note that other implementations of the *MonadRef* type class are available, but are not compatible with this code. See main text for a detailed description. This and supporting code can be found at https://github.com/samer--/cpsmemo in the directory `haskell`.

This means that a computation which produces an a is represented in continuation passing style (CPS) as a function which, given a continuation ready to map an a to a stateful nondeterministic computation of type r, returns a stateful nondeterministic computation of r. The essence of CPS is that this function is free to ignore or use the continuation as many times as necessary to achieve the desired effect.

Using *ContT* to define the *NDC* monad means that many useful instances are automatically defined, in particular *MonadRef*, which means that the instance methods *newRef*, *readRef* and *writeRef* are available. Once instance which is not defined automatically is that for *MonadPlus*. This is easily done: *mplus* is implemented (on line 16) using the *mplus* operator of the underlying nondeterminism monad n to combine the results of applying the two alternatives f and g to the continuation k. The *mzero* is even simpler (line 15): the continuation is ignored and the *mzero* of the underlying monad n returned directly.

The type constructor *NDCK s n r* on line 11 is nothing more than the Kleisli arrow for the monad *NDC s n r*, but is useful to declare as it is objects of this type that will be memoised by the *memo* operator.

The type *Table a b* on line 12 represents the external form of the memo table for a computation in this arrow, and is an associative map from values of type a to a sets of values of type b.

The function *run* (lines 18–19) reduces a computation in the *NDC* monad to a computation in the *ST* monad which, when run, using *runST*, will produce a collection of results represented using the base instance of *MonadPlus*.

We now come to the *memo* function itself. In order to memoise a monadic computation f, the first step is to create a reference to a new mutable cell to contain the memo table for that computation (line 24). For a computation of type *NDCK s n r a b*, this table will be of type *Map a* (*Set b*, $[b \to$ *NDC s n r r*$]$), that is, a map associating with each a a set of bs and a list of continuations accepting a b. The use of *Map* and *Set* here requires that both a and b be instances of the type class *Ord* (line 21). Then, *memo* returns (monadically) two monadic computations, the first (line 39) can be used to get the current state of the memo table, but with the list of continuations stripped out, while the second (line 40) is the memoised version of f. When applied to a value x, it gets the current state of the memo table, captures the current continuation (out to the nearest enclosing *runContT*), and passes everything to the local function *feed* (lines 25–38), which handles the bulk of the processing.

If the memo table already includes an entry for x (line 38), then this is passed to *consumer*, which updates the entry by prepending the newly captured continuation k to the list of continuations associated with this entry (line 28) and then returns the monadic sum of the results of applying k to each of the values returned by f so far (line 29). If the table does not contain an entry for x, then *producer* adds one, with an empty set of results and the continuation k (line 31), and then calls f. This part of the computation is still in the *NDC s n r* monad: for each result y produced nondeterministically by f (line 32), the current state of the table entry is checked; if y is already in the set of results (line 35), then this branch of the computation fails; otherwise, the new value is added to the set (line 36) and the function returns the monadic sum of the results of applying each continuation in the stored list to the new value (line 37). The helper function *update* on line 26 simplifies the process of updating the table entry for x and storing this in cell referred to by *loc*.

2 Example: Fibonacci function

In the previous section, the mechanism for setting up recursive computations was glossed over. In the OCaml implementation of Abdallah (2017a), this required careful handling by writing recursive computation in open recursive style and then applying an explicit fixed point combinator. In Haskell, the typeclass *MonadFix* in *Control.Monad.Fix* already provides the necessary functionality, while an extension to the **do** notation, the *recursive do notation* (Erkök and Launchbury, 2000) makes this feature transparently easy to use: inside an **mdo** block, the right hand sides of any monadic binding operators ← can refer to names defined on the left hand sides using any kind of mutual recursion, just as in ordinary Haskell bindings. Since the *ST* monad already has an instance of *MonadFix*, a fast Fibonacci function can be implemented as follows.

```
import Control.Monad.Fix

memo' :: (MonadPlus n, Ord a, Ord b) ⇒ (NDCK s n r a b) → ST s (NDCK s n r a b)
memo' = fmap snd . memo

ffib  n = runST $ mdo
  fib  ← memo' (λn → if n<2 then return n else
                         liftM2 (+) (fib  (n−2)) (fib  (n−1)))
  run (fib  n)
```

Here, the *memo*' function simply discards the memo table retrieval operator returned by *memo*, as we are only interested in the memoised computation itself. The run time of this implementation is approximately linear in the argument *n*, and thanks to Haskell's unbounded *Integer* type, runs happily with *n* in the tens of thousands. When compiled using GHC version 7 and run on a 2015 MacBook Pro with a 2.7 GHz Core i5 processor and 8 GB of memory, the 10,000th Fibonacci number is computed in 70 ms, while the 20,000th number is found in 130 ms (it is 4,180 digits long).

3 Adding parser combinators

Since the *NDC* monad already provides the nondetermism required for building flexible top down parser combinators, the only other ingredient that needs to be added is a representation of the parser *state*, that is, how far through the input sequence the parser as got and what remains to be parsed. Although it would be possible to add another monadic layer to handle this, the code below takes a direct approach, representing the parser state as a list of remaining unparsed symbols, and passing this in to and out of the parsers explicitly. Thus, a parser of sequences of *a*s is an *NDCK* computation from a list of *a*s to another list of *a*s. The code is shown in Fig. 2. Parser combinators for sequencing ∗> and alternatives <|> are trivially defined, as are the primitives *term* for matching a particular symbol and *epsilon* for matching the empty sequence.

The following shows how this parsing framework can be used—it is an implementation of the grammar used by Johnson (1995) to illustrate his memoising parser combinators, which were the inspiration for the approach presented presented here and previously in other languages (Abdallah, 2017a,b). The type

```
infixl 7 *>
infixl 6 <|>

(*>) f g xs = f xs ≫= g
(<|>) f g xs = (f xs) `mplus` (g xs)
term x (y:ys) | x==y = return ys
term x _ = mzero
epsilon xs = return xs
```

Figure 2: Code for implementing parsers combinators. Apart from the assumed instance of *MonadPlus*, it is completely orthogonal to the memoisation system.

ParserGen represents a computation with the right sort of context to support the creation of memoised parsers using the *memo* or *memo'* functions defined earlier. Note how the **mdo** notation supports the recursive *np* parser and the mutually recursive *s* and *vp* parsers.

```
type Parser s n r a = NDCK s n r [a] [a]
type ParserGen s n r a = ST s (Parser s n r a)

johnson :: MonadPlus n ⇒ ParserGen s n r String
johnson = mdo
  v   ← return $ term "likes"  <|> term "knows"
  pn  ← return $ term "Kim" <|> term "Sandy"
  det ← return $ term "every" <|> term "no"
  n   ← return $ term "student" <|> term "professor"
  np  ← memo' $ det *> n <|> pn <|> np *> term "'s" *> n
  vp  ← memo' $ v *> np <|> v *> s
  s   ← memo' $ np *> vp
  return s
```

Notably lacking from the above is any commitment to a particular instance of *MonadPlus* as the base representation of nondeterminism. For parsing, a better alternative to the list monad is the following *free* monad, which, in practise, means that the collection of results is represented as a lazy tree.

```
data FreePlus a = Return a | Plus (FreePlus a) (FreePlus a) | Zero

instance Monad FreePlus where
  return = Return
  Zero       ≫= _ = Zero
  Return x ≫= f = f x
  Plus l r ≫= f = Plus (l ≫= f) (r ≫= f)

instance MonadPlus FreePlus where
  mzero = Zero
  mplus = Plus

parse :: (forall s. ParserGen s FreePlus [t] t) → [t] → FreePlus [t]
parse p xs = runST $ p ≫= run . ($ xs)
```

In operational terms, the function *parse* takes a *ParserGen*, runs it in the *ST* monad to get a *Parser*, applies the parser to the input sequence *xs* to a get computation in the *NDC s FreePlus r* monad, runs that to get a computation in the *ST* monad, and finally runs that to get a collection of tail sequences resulting from all possible parses.

It is worth noting that the parsing using this framework is *efficient*: like Johnson's (1995) parser and parsing using tabling in Prolog (Abdallah, 2017b), it is equivalent to Earley's chart parsing algorithm Earley (1970), which results in at-worst complexity of $O(N^3)$ in the length of the input sequence, depending on the level of ambiguity in the grammar.

4 Conclusions

The implementation presented here in Haskell is, in essence, the same as the OCaml version presented by Abdallah (2017a). However, language features found in Haskell but not in OCaml, combined with standard library modules for several relevant monads and monad transformers mean that the result is considerably shorter and in some ways more elegant. In particular, Haskell's typeclasses mean that most of the implementation of the *NDC* monad is derived automatically, leaving only the *MonadPlus* instance and the memoising operator *memo* itself to defined manually.

In addition Haskell's more flexible polymorphism means that there are fewer hoops to jump through to obtain 'enough' polymorphism in the implementation; for example, there is no need for a *Dynamic* module to implement a universal type to serve as the answer type of the continuation monad here.

Finally, the presence of *MonadFix* and the **mdo** notation make the use of open recursion and explicit fixed point operators unnecessary, making it possible to express recursive and mutually recursive computations in a more natural and less cluttered style.

In tests made so far, performance is on a par with the OCaml version.

References

S. Abdallah. Memoisation: Purely, left-recursively, and with (continuation passing) style. *arXiv preprint arXiv:1707.04724*, 2017a.

S. Abdallah. More declarative tabling in Prolog using multi-prompt delimited control. Technical report, Jukedeck Ltd., 2017b.

J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

L. Erkök and J. Launchbury. Recursive monadic bindings. In *ACM Sigplan Notices*, volume 35, pages 174–185. ACM, 2000.

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