Disposable Memo Functions[*] [†]

Byron Cook John Launchbury
byron@cse.ogi.edu jl@cse.ogi.edu

Department of Computer Science and Engineering
Oregon Graduate Institute

Abstract

We formalize the meaning of lazy memo-functions in Haskell with an extension to the lazy λ-calculus, Haskell's computational model. The semantics enable reasoning about memoization's effect on space and time complexity. Based on the semantics, we present a prototype implementation that requires no changes to the garbage-collector; memo-tables are simply reclaimed when no references to them remain.

1 Introduction

A *memo-function* remembers the arguments to which it has been applied, together with the result. If applied to a repeated argument, memo-functions return the cached answer rather than recomputing it from scratch. Memoization improves the time complexity of algorithms with repeated computations — but can consume vast amounts of memory. Some implementations of memoization use heuristic cache replacement policies (such as LRU or FIFO) to manage memory. Although implementations try to minimize space consumption, in their essence, memo-functions trade better runtime performance for worse space behavior.

Programmers are accustomed to these sorts of trade-offs: "Should I cache this value or is it too big to keep around?" Many programmers develop principles on which to base such decisions. But when memo-functions use heuristic purging, the old reasoning principles may no longer apply. Values might be removed from the memo-table if they have a certain type, or size, or are not used frequently enough. To make matters worse, the cache replacement policies make assumptions about evaluation strategy. However, there is one form of purging known to work well in Haskell: garbage-collection.

Can memoization be integrated into Haskell such that memo-tables are managed like other values in the heap? We show that it can. What's more, we give a formal semantics for memoization that is compatible with Haskell's underlying computational model, and where the garbage-collection rule can reclaim obsolete memo-functions and the space their tables consumed.

The key idea in this paper is to adapt Hughes' research on lazy memoization[7] and provide a polymorphic function:

```
memo :: Eval a => (a -> b) -> (a -> b)
```

When applied to a function `memo` returns an equivalent memoized function. When all references to this new function have been dropped, the garbage-collector is able to reclaim the function and its table. It is in this sense that memo functions are disposable.

To provide a concrete example of using `memo`, consider applying the factorial function to each element in a list.

```
map fact [17,8,17,17,17,8]
```

Clearly, recomputing `(fact 17)` four times is inefficient. This can be remedied by mapping a memoized version of `fact` down the list:

```
map (memo fact) [17,8,17,17,17,8]
```

Once the expression has been reduced, no references to the memoized `fact` will remain. The garbage collector can then reclaim it and its memo-table. Of course, in this trivial example we could have explicitly cached the repeated computations with a `let` but, as we will demonstrate later, this technique doesn't scale to larger programs.

We have developed a prototype implementation of the semantics, appropriately named Huggies, by extending the Hugs Haskell interpreter. Huggies demonstrates that, in theory, there need be no direct connection between the garbage-collector and `memo`, and our implementation required no changes to the Hugs garbage-collector (we discuss compacting collectors later). Using heap profiles, we will show that Huggies can garbage-collect disposable memo-functions.

2 Applications of Memoization

In this section we demonstrate the utility of memo-functions by addressing two problems in areas of recent research. Both examples involve specification languages embedded in Haskell. Although specifications are concisely expressed in pure functional languages, the resulting programs often contain computational redundancies. The programs typically must be modified to enhance their computational content. However, the very changes that improve efficiency also obscure structure and degrade maintainability.

2.1 Parsers

If interpreted directly with recursive descent, presentations of grammars often contain repeated computations. Typically an efficient parser is based on a transformed grammar. In some cases, rather than redesigning the grammar, a memoized parser can be about as fast. As a simple example, consider the grammar for arithmetic expressions:

[*]The authors are supported by a contract with Air Force Material Command (F19628-93-C-0069).

[†]In Proceedings of the June 1997 Haskell Workshop (Amsterdam)

$$\begin{array}{lcl}
\mathit{term} & ::= & \mathit{factor}\ \texttt{+}\mathit{factor} \\
 & | & \mathit{factor}\ \texttt{-}\mathit{factor} \\
 & | & \mathit{factor} \\
\mathit{factor} & ::= & \mathit{expr}\ \ \texttt{*}\ \mathit{expr} \\
 & | & \mathit{expr}\ \ \texttt{/}\ \mathit{expr} \\
 & | & \mathit{expr} \\
\mathit{expr} & ::= & \mathit{number} \\
 & | & (\mathit{term})
\end{array}$$

Using Hutton and Meijer's parser combinators[8] the corresponding naive parser is:

```
naiveTerm =   binary factor '*' factor
          +++ binary factor '/' factor
          +++ factor
     where
     factor =   binary expr '+' expr
            +++ binary expr '-' expr
            +++ term
     expr = num +++ bracketed naiveTerm
```

Unfortunately, `naiveTerm` is slow. When parsing the expression `"(((1-2)-2)-3)"`, `naiveTerm` first traverses `"((1-2)-2)"` before discovering that the next character is not a '*'. It throws away the parse and begins anew only to repeat the process several more times. Of course, the parser and grammar could be restructured to explicitly cache intermediate values, but a simpler transformation would be to memoize `factor` and `expr`:

```
term () = trm
     where
     trm =   binary factor '*' factor
         +++ binary factor '/' factor
         +++ factor
     factor = memoParser (
             binary expr '+' expr
         +++ binary expr '-' expr
         +++ term )
     expr = memoParser (num +++ bracketed trm)

memoParser (Parser f) = Parser (memo f)
```

The memoized parser, while perhaps a bit slower than the restructured parser, has reasonable time complexity. More importantly, the program remains maintainable because the transformation has preserved the original structure.

Notice that `term` is not itself a memo-function. When applied to `()`, it returns a parser which builds local memo-functions. While parsing, the local memo-tables are bounded by the size of the input string. Once the string is parsed, `(term ())` becomes garbage, along with any memo-functions.

It is tempting to define the parser at the top-level:

```
term' = term ()
```

However, `term'` will build its local memo-functions only once. In Huggies, `term'` will persist throughout the lifetime of the program. The more `term'` is used, the more heap it will consume.

2.2 Animation Combinators

Reactive Behavior Modeling in Haskell[3] (RBMH) is a language in development at Microsoft for describing multi-media interactive animation. RBMH is a suite of combinators and simple behaviors from which programmers can build complex behaviors. The abstraction of combinators and behaviors can lead to programs with repeated computations — especially since behaviors are functions on time:

```
data Behavior a = B(Time -> (a, Behavior a))
```

In RBMH, references to behaviors are easily duplicated, resulting in redundant sampling. For example, `(+)` is overloaded on behaviors such that, when `b` and `c` are behaviors, the expression `(b + c)` reduces to:

```
B(\t -> let (x,b') = at t b
            (y,c') = at t c
        in (x + y, b' + c'))
```

where `at` returns the value of `b` at time `t`. However, `b + b` does not reduce to:

```
B(\t -> let (x,b') = at t b
        in (x + x, b' + b'))
```

as you might hope. Instead, `(at t b)` is computed twice:

```
B(\t -> let (x,b') = at t b
            (y,c') = at t b
        in (x + y, b' + c'))
```

The overloaded `(+)` could remove this redundancy by memoizing `(at t)`:

```
B(\t -> let at' = memo (at t)
            (x,b') = at' b
            (y,c') = at' c
        in (x + y, b' + c'))
```

Is `at'` disposable? Yes; once `x`, `y`, `b'`, and `c'` are computed no references to `at'` will remain.

3 Semantics

In this section, we provide an operational semantics for memoization in Haskell by extending the lazy λ-calculus. The power of the operational semantics is that it gives neither a trivial denotational meaning, nor a meaning based on stack pointers, program counters, and jumps. The semantics captures the essence of memoization at the right level of abstraction; it is detailed enough to be useful, and simple enough that it does not obscure meaning.

3.1 Lazy Semantics

The syntax of the source language is defined as:

$$\begin{array}{rll}
e \in & \mathit{Expns} & ::= x \mid v \mid c \mid e\ a \mid \\
 & & \quad \texttt{let}\ \{x_1 = e_1; \ldots\}\ \texttt{in}\ e \\
a \in & \mathit{Atoms} & ::= x \mid n \\
v \in & \mathit{Values} & ::= n \mid \lambda x.e \\
c \in & \mathit{Constants} & ::= + \mid - \mid \ldots \\
n, m, p \in & \mathit{Numbers} & ::= 1 \mid 2 \mid \ldots \\
?, \Delta \in & \mathit{Heaps} & ::= ?, x \mapsto e \mid \varnothing \\
f, g, x, y, z, t \in & \mathit{Variables} & \\
\rho \in & \mathit{Envs} & ::= \{\ldots (x, y) \ldots\} \mid \varnothing
\end{array}$$

3.2 Memo Semantics

Terms in the semantics are formed of pairs $\langle\ e \mid ?\ \rangle$, where e is an expression and $?$ is a heap. The semantic rules of the lazy λ-calculus are given in Figures 1 and 2 as a relation $\Longrightarrow$ and $\longrightarrow$ between terms. The semantics contain the following meta-operations:

$e[y/x]$: substitution of y for occurrences of x in e.

$\hat{v}$: α-renaming with fresh variables

$$\langle\ \texttt{memo}\ f \mid ?\ \rangle \longrightarrow \langle\ \lambda x.\ \mathsf{access}\ t\ x \mid ?,\ t \mapsto (f, \varnothing)\ \rangle \qquad (\mathit{memo})$$

$$\frac{\langle\ e \mid ?\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ \mathsf{access}\ t\ x \mid ?,\ x \mapsto e\ \rangle \longrightarrow \langle\ \mathsf{access}\ t\ x \mid \Delta,\ x \mapsto e'\ \rangle} \qquad (\mathit{access}^e)$$

if e is not a value

$$\langle\ \mathsf{access}\ t\ x \mid ?,\ x \mapsto n\ \rangle \longrightarrow \langle\ \mathsf{access}\ t\ n \mid ?,\ x \mapsto n\ \rangle \qquad (\mathit{access}^n)$$

$$\langle\ \mathsf{access}\ t\ n \mid ?,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ \rho(n) \mid ?,\ t \mapsto (f, \rho)\ \rangle \qquad (\mathit{access}^{\in n})$$

if $n \in \mathit{dom}\ \rho$

$$\langle\ \mathsf{access}\ t\ n \mid ?,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ z \mid ?,\ t \mapsto (f, \rho \cup \{(n, z)\}),\ z \mapsto f\ n\ \rangle \qquad (\mathit{access}^{\notin n})$$

if $n \notin \mathit{dom}\ \rho$

$$\langle\ \mathsf{access}\ t\ x \mid ?,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ \rho(x) \mid ?,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho)\ \rangle \qquad (\mathit{access}^{\in \lambda})$$

if $x \in \mathit{dom}\ \rho$

$$\langle\ \mathsf{access}\ t\ x \mid ?,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ z \mid ?,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho \cup \{(x, z)\}),\ z \mapsto f\ x\ \rangle \qquad (\mathit{access}^{\notin \lambda})$$

if $x \notin \mathit{dom}\ \rho$

Figure 3: Semantics of Memo

$$\langle\ (\lambda x.e)\ y \mid ?\ \rangle \longrightarrow \langle\ e[y/x] \mid ?\ \rangle \qquad (\mathit{app}^\beta)$$

$$\frac{\langle\ e \mid ?\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ e\ y \mid ?\ \rangle \longrightarrow \langle\ e'\ y \mid \Delta\ \rangle} \qquad (\mathit{app}^e)$$

$$\langle\ y \mid ?,\ y \mapsto v\ \rangle \longrightarrow \langle\ \hat{v} \mid ?,\ y \mapsto v\ \rangle \qquad (\mathit{var}^v)$$

where $\hat{v}$ is v with all bound variables renamed to fresh names

$$\frac{\langle\ e \mid ?\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ y \mid ?,\ y \mapsto e\ \rangle \longrightarrow \langle\ y \mid \Delta,\ y \mapsto e'\ \rangle} \qquad (\mathit{var}^e)$$

$$\langle\ \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \mid ?\ \rangle \longrightarrow \langle\ e' \mid ?,\ x \mapsto e\ \rangle \qquad (\mathit{let})$$

$$\langle\ n\ \texttt{+}\ m \mid ?\ \rangle \longrightarrow \langle\ p \mid ?\ \rangle \qquad (\mathit{plus}^n)$$

where $p = m + n$

$$\frac{\langle\ e \mid ?\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ n\ \texttt{+}\ e \mid ?\ \rangle \longrightarrow \langle\ n\ \texttt{+}\ e' \mid \Delta\ \rangle} \qquad (\mathit{plus}^l)$$

$$\frac{\langle\ e_1 \mid ?\ \rangle \longrightarrow \langle\ e_1' \mid \Delta\ \rangle}{\langle\ e_1\ \texttt{+}\ e_2 \mid ?\ \rangle \longrightarrow \langle\ e_1'\ \texttt{+}\ e_2 \mid \Delta\ \rangle} \qquad (\mathit{plus}^r)$$

Figure 1: Lazy λ-calculus Semantics

The lazy λ-calculus models Haskell's evaluation strategy by sharing computations and halting reduction when outside λs are encountered. The sharing is achieved through two constraints on reduction:

$$\langle\ e \mid ?,\ x \mapsto e'\ \rangle \Longrightarrow \langle\ e \mid ?\ \rangle \qquad (\mathit{gc})$$

if x is not reachable from e

$$\frac{\langle\ e \mid ?\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ e \mid ?\ \rangle \Longrightarrow \langle\ e' \mid \Delta\ \rangle} \qquad (\mathit{reduce})$$

Figure 2: Lazy λ-calculus with Garbage-collection

- Functions can only be applied to atomic values like integers and variables. This prevents arbitrary expressions being substituted into function bodies. For example, the reduction $(\lambda x.x + x)(3 + 3) \longrightarrow (3 + 3) + (3 + 3)$ is not allowed because it would duplicate the computation $3 + 3$.
- Before replacing a variable with its heap value, the value must be in weak head normal form. For example, the expression $3 + 3$ must be reduced in the heap first before substituting it for x: $\langle\ x + x \mid x \mapsto 3 + 3\ \rangle \longrightarrow \langle\ x + x \mid x \mapsto 6\ \rangle \longrightarrow \langle\ 6 + 6 \mid x \mapsto 6\ \rangle$

We maintain the invariant that all binding sites bind distinct names. Whenever a term is suplicated (in a (var^a) rule) we α-rename with fresh names.

To define memoization we extend the λ-calculus with the constant `memo` and language construct access:

$$\begin{array}{lll} c \in \mathit{Constants} & ::= \ldots & \mid\ \texttt{memo} \\ e \in \mathit{Expns} & ::= \ldots & \mid\ \mathsf{access}\ t\ x \end{array}$$

Initial terms should not contain access expressions — access should only be introduced by reduction.

Figure 3 extends the semantics with rules for `memo` and access. The semantics use the additional meta-operations:

$\mathbf{x} \in \mathbf{X}$: set membership. The sets contain mixtures of variable names and numbers (i.e. atomic values).

$\rho(x)$: the corresponding value for x in the memo-table ρ

$\rho \cup \{(x,y)\}$: extension of the memo-table ρ with the tuple (x,y); x may not appear in the domain of ρ

The basic point is that if a memo-function is applied to an argument already in the table, then the already-computed value is returned. If the function is applied to a new value then it is placed into the table along with a reference to the answer.

When `memo` is applied to a variable y (and notice that `memo` can only be applied to variables) the computation is replaced by ($\lambda x.$access t x), where t is a reference to the tuple $(y, \varnothing)$ in the heap. When an `access` expression is reduced, the memo-table is potentially updated.

The behavior of memo-functions depends critically on the sharing of the lazy λ-calculus. Performance is lost if reductions can duplicate expressions containing applications of `memo`. For example, the expression $(\lambda f. f\ 5$ + $f\ 5)$(`memo` g) should evaluate to (`access` t 5 + `access` t 5) and not ((`memo` g) t 5 + (`memo` g)t 5).

We are now able to answer this paper's thesis. We have defined memo-tables as local variables inside of memo-functions. Therefore, when a memo-function becomes garbage the corresponding memo-table should be garbage too. Let $E[\bullet]$ be an expression containing $\bullet$. If $\langle\ E[$`let` f = `memo` g `in` $e'] \mid ?\ \rangle \Longrightarrow \langle e \mid \Delta,\ f \mapsto \lambda x.$access $t\ x,\ t \mapsto (f, \rho)\ \rangle$ and ($\lambda x.$access t x) is not reachable from e, then $\langle\ e \mid \Delta,\ f \mapsto \lambda x.$access $t\ x,\ t \mapsto (f,\rho)\ \rangle \ \Longrightarrow\ \langle\ e \mid \Delta\ \rangle$. Because *memo* is the only rule introducing references to t, and the access rules do not duplicate references to t, then all references to t are in the form `access` t x. Therefore, if `access` t x is not reachable from e, then t is not reachable from e; and both f and t can be removed from the heap by an application of (gc).

In this paper we do not explore the properties of the extended semantics. A proof showing that `memo` f is denotationally equivalent to a strict f would be a very strong result — and one we would like to pursue.

It might also be fruitful to abstract the semantics of `memo` over the computational model. By specifying the requirements of the underlying model, rather than relying on the particular properties of the lazy λ-calculus, the meaning of memoization could be applicable in many settings. For example, can the semantics be adapted to SML, or a strictness neutral language like Henk?

3.3 Example Reduction

Figure 4 provides a top-level reduction sequence for the term $\langle$`let` g = `memo` f `in` g 5 + g 5 $\mid ?\ \rangle$ The arrows are annotated with the rules of the sub-reduction used to establish each reduction. The (gc) step is important. After the expression has been reduced to $\langle\ 16 + z \mid \ldots \rangle$ any extra heap space is thrown away. We assume $f\ 5\ =\ 16$ for concreteness.

4 Implementation

Huggies provides a proof-of-concept, and is far from optimal. Huggies clearly implements our semantics, and is a starting point from which future refinements can be based.

The function `memo` is written in Haskell as the composition of several non-standard primitives. The primitives represent the pieces of the semantics that are impure.

4.1 Memoizing in Standard Haskell

Before extending the language with `memo`, it is useful to see how we might memoize functions in standard Haskell. Using the example in Section 1 we can build a new function, `fastMap`, that uses a memo-function while traversing the list and assumes that the list contains integers between 0 and 96:

```
fastMap :: (Int -> b) -> [Int] -> [b]
fastMap f list = runST (
    do g <- memoST f
       mapM g list
    )

memoST :: (a -> b) -> ST s (a -> ST s b)
memoST f =
  do t <- newArray (0,97) Nothing
     return (\x -> accessST t x)
  where
  accessST t x =
    do let w = x `mod` 97
       a <- readArray t w
       case a of
         Nothing -> do let z = f x
                       writeArray t w (Just z)
                       return z
         Just y -> return y
```

The example can now be re-written as:

```
fastMap fact [17,8,17,17,17,8]
```

In this expression, `(fact 17)` is calculated only once. What's more, the expression is purely functional.

4.2 Non-standard Primitives

4.2.1 Memo-tables

Fast memoization depends critically on an efficient implementation of environment lookup ($\in$ in the semantics). However, Haskell's purity stands in the way of fast polymorphic environments written within the language. Therefore, we have added the environment type `Env` to Huggies:

```
newEnv    :: (a -> a -> ST s Bool)
                -> ST Mem (Env s a b)
readEnv   :: Env s a b -> a -> ST Mem (Maybe b)
writeEnv  :: Env s a b -> a -> b -> ST Mem ()
accessEnv :: Env s a b -> a -> b -> ST Mem b
```

The underlying implementation is a hash-table where we hash on based the value of integers, booleans, characters and floating point numbers, and the location of other values. Because environments depend on the state of global memory, `ST`'s first parameter is instantiated to the constant `Mem`.

The function `newEnv`, when passed a stateful equality function, returns a new environment. Subsequent reads and writes to the environment use the parameterized equality function to determine where in the environment the read or write should be performed.

The function `accessEnv` is an efficient composition of `readEnv` and `writeEnv`; but it can be thought of as:

```
accessEnv e x y =
  do r <- readEnv e x
     case r of
       Nothing -> do writeEnv e x y
                     return y
       Just y' -> return y'
```

Notice that our implementation of `Env` and the garbage-collector are tightly coupled — environments would have to be rebuilt after each garbage-collection if Hugs compacted memory. But the interaction between `Env` and the garbage-collector is not visable from the defintion of `memo`.

$$
\begin{aligned}
& \langle\ \texttt{let}\ g = \texttt{memo}\ f\ \texttt{in}\ g\ 5 + g\ 5 \mid ?\ \rangle \\
\xrightarrow{let}\ & \langle\ g\ 5 + g\ 5 \mid ?,\ g \mapsto \texttt{memo}\ f \rangle \\
\xrightarrow{plus^l, app^e, var^a, memo}\ & \langle\ g\ 5 + g\ 5 \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \varnothing)\ \rangle \\
\xrightarrow{plus^l, app^e, var^v}\ & \langle\ (\lambda x.\mathsf{access}\ t\ x)\ 5 + g\ 5 \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \varnothing)\ \rangle \\
\xrightarrow{plus^l, app^\beta}\ & \langle\ \mathsf{access}\ t\ 5 + g\ 5 \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \varnothing)\ \rangle \\
\xrightarrow{plus^l, access^{\notin n}}\ & \langle\ z\ + g\ 5 \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto f\ 5\ \rangle \\
\xrightarrow{plus^l, var^v}\ & \langle\ 16\ + g\ 5 \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\xrightarrow{plus^r, app^e, var^v}\ & \langle\ 16\ +\ (\lambda x.\mathsf{access}\ t\ x)\ 5 \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\xrightarrow{plus^r, app^\beta}\ & \langle\ 16\ +\ \mathsf{access}\ t\ 5 \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\xrightarrow{plus^r, access^{\in n}}\ & \langle\ 16\ +\ z \mid ?,\ g \mapsto \lambda x.\mathsf{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\xrightarrow{gc}\ & \langle\ 16\ +\ z \mid ?,\ z \mapsto 16\ \rangle \\
\xrightarrow{plus^r, var^v}\ & \langle\ 16\ +\ 16 \mid ?,\ z \mapsto 16\ \rangle \\
\xrightarrow{plus^n}\ & \langle\ 32 \mid ?,\ z \mapsto 16\ \rangle
\end{aligned}
$$

Figure 4: Example Reduction

4.2.2 Equality

As described in Section 1, lazy memo-functions use a different notion of equality depending on the type of the arguments. The primitive function `eql` implements this behavior:

```
eql :: (Eval a,Eval b) => a -> b -> ST Mem Bool
```

Notice that `eql` accepts arguments of different types. We are in a precarious position with the type system. By using `eql` rather than `(==)` we have avoided a subtle error that could cause programs to crash. To see why this is true, imagine memoizing the identity function with traditional memoization:

```
memoEq :: Eq a => (a -> b) -> (a -> b)

let f = memoEq id
in (f 5, f "hello")
```

If `(f 5)` is reduced first then the subsequent reduction of `(f "hello")` might cause `access` to compare `5` and `"hello"`. Fortunately, `eql` simply returns `False`, but `(==)` is not so forgiving.

4.2.3 Unsafe State Monad Escape Operator

The final primitive added to Huggies has the same behavior as `runST` without the restrictive type:

```
unsafeST :: ST s a -> a
```

This new primitive has to be used with extreme caution. Whereas `runST` will only encapsulate referentially transparent functions, `unsafeST` makes no such distinction. This allows the state of a memo-function to flow implicitly throughout a program's execution. In the case of `memo` this effect is not observable, but a misuse of `unsafeST` could easily subvert Haskell's purity.

4.3 Defining `memo`

The functions `memo` and `access` are thus defined:

```
memo :: Eval a => (a -> b) -> (a -> b)
memo f = \x -> access t x
     where
     t = (f,emptyEnv)
     emptyEnv = unsafeST (newEnv eql)

access :: Eval a => (a -> b, Env Mem a b)
          -> (a -> b)
access (f,t) x = unsafeST (accessEnv t x (f x))
```

4.4 Recursion

Memoizing syntactically recursive functions with `memo` is clumsy:

```
fib = memo mfib
     where
     mfib 0 = 1
mfib 1 = 1
mfib n = fib (n-1) + fib (n-2)
```

That fib has the correct behavior may be seen by an application of the semantic rules, but it is far from intuitive. However, functions that are defined as the fixed point of functionals can be memoized with memoFix:

```
memoFix f = let g = f h
                h = memo g
            in h
```

Recursive functions, such as Fibonacci, can be written and then memoized as such:

```
fib m 0 = 1
fib m 1 = 1
fib m n = m (n-1) + m (n-2)

memofib = memoFix fib
```

4.5 Dangers of unsafeST

Although unsafeST makes the definition of memo possible, we are wary of its robustness and portability. Programs written with unsafeST can exhibit surprising behaviors. The state monad and runST were carefully designed to avoid functions like unsafeST — and for good reason. Hidden updatable state endangers referential transparency. As an example, we ask the question: does bad () equal bad ()? Not when defined as follows:

```
bad :: () -> Bool
bad = unsafeST (
  do v <- newVar True
     return (\x -> unsafeST (toggle v x))
  )


toggle v () =
  do x <- readVar v
     case x of
        True  -> do () <- writeVar v False
                    return False
        False -> do () <- writeVar v True
                    return True
```

After each application the value of bad () toggles between True and False. Much like bad, memoized functions contain a mutable variable that is updated during applications; however, because memo was carefully written, memoized functions should preserve equational reasoning. The values that result in expressions containing unsafeST, therefore, come with proof obligations to guarantee their safe behavior.

Notice that there is subtle interaction between unsafeST and lazy state in the definition of toggle. If toggle were re-written as

```
toggle v () =
  do x <- readVar v
     case x of
        True  -> do writeVar v False
                    return False
        False -> do writeVar v True
                    return True
```

then the mutable variable is no longer updated. In this case the return value of toggle does not depend on the application of writeVar and it is simply never performed.

5 Profiling Huggies

We have preliminary evidence that the Huggies garbage-collector reclaims the memo-tables of disposable memo-functions. Figure 5 contains heap profiles annotated with the expressions that were executed while generating them. The expressions on the left use disposable memo functions. The expressions on the right use non-disposable memo-functions. Notice that heap consumption returns to zero in the profiles on the left — indicating that the garbage-collector successfully reclaimed the space.

6 Related Work

6.1 Lazy Memo-functions

Rather than defining memoization, Hughes[7] focused on the applications of lazy memo-functions and the implementation issues of obsolescence-based purging optimizations

Hughes proposed that memo-functions be defined with a language construct rather than a higher-order function. In his syntax, the keyword memo was placed in front of the memo-function's definition.

```
memo fib 0 = 1
memo fib 1 = 1
memo fib n = fib (n-1) + fib (n-2)
```

It is easier to define recursive memo-functions in this notation, but clumsy to use when developing a semantics.

The semantics presented in this paper are compatible with Hughes's original work. In fact, we have clarified issues originally raised in Hughes's paper. For example, Hughes states the following definition of map creates a local memo-function that can be garbage-collected after map has been applied:

```
map f l  = m l
 where memo m []     = []
       memo m (x:xs) = f x: m xs
```

However, the paper does not develop principles for reasoning about when a memo-function can be garbage-collected. With our semantics (extended with lists), it could be verified that m is disposable once translated into the appropriate form:

```
map f l  = memoFix m l
 where m g []     = []
       m g (x:xs) = f x: g xs
```

6.2 Memoization in LISP and SML

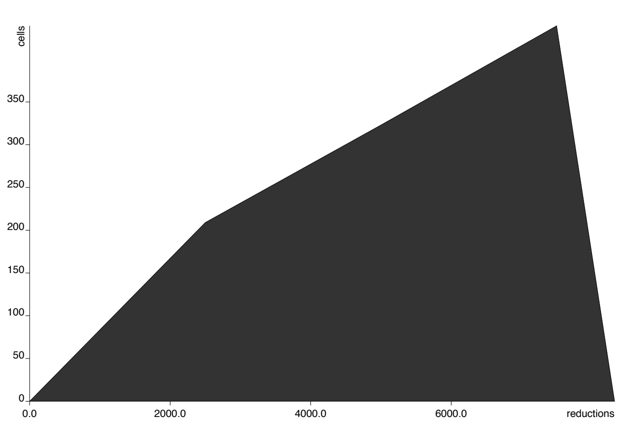
Our research mirrors similar work in LISP[5] and SML[2]. All of the implementations use higher-order functions named memo that return memo-functions.

How is our research different? The challenge that we have faced is referential transparency. Both the LISP and SML implementations were simply defined in the source language; which is easy to do because they are imperative languages (with a functional subset). In Haskell, memo must be defined outside of the language with a semantics that is formal enough to show that referential transparency holds.

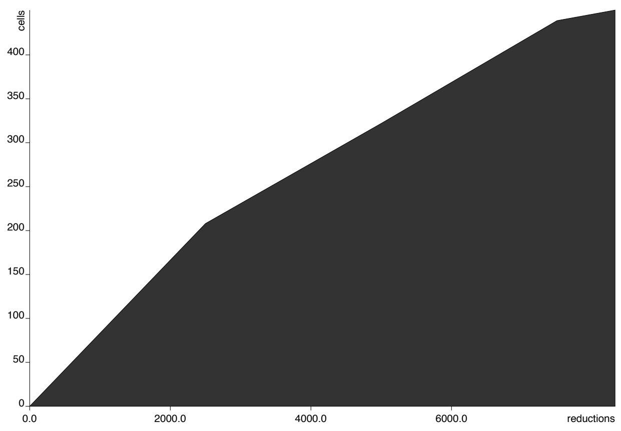
6.3 Memo Gofer

van Dalen[15] extended the Gofer interpreter with a very different notion of lazy memo-functions. As in this paper, memoization is provided with a primitive function memo. However, rather than creating a new function, memo refers to a global memo-table.

In van Dalen's Gofer, a memoized Fibonacci is defined as:

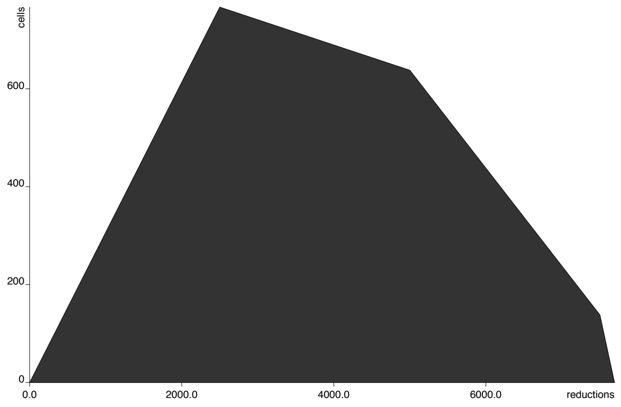


`map (memo f) [1 .. 100]`

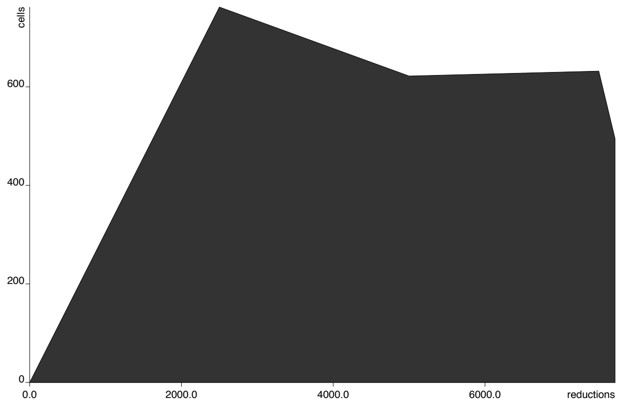


`map f' [1 .. 100]`

where `f x = x+1` and `f' = memo f` at the top-level



```
papply (expr ())
"((((((((((1-1)-1)-1)-1)-2)-2)-2)-1)-3)-4)"
```



```
papply (expr')
"((((((((((1-1)-1)-1)-1)-2)-2)-2)-1)-3)-4)"
```

Figure 5: Heap Profiles of Disposable and Non-disposable Memo-functions

```
    memofib = memo mfib
         where
         mfib 0 = 1
         mfib 1 = 1
         mfib n = memo mfib (n-1) + memo mfib (n-2)
```

Notice that, with the semantics in this paper, `memofib` would generate a separate memoized `mfib` for each recursive application.

7 Conclusions

Based on Hughes's lazy memo-functions we have given memoization a formal meaning. The semantics describe memoization at precisely the right level of abstraction. With the semantics, it is possible to reason about the space and performance of memo-functions. Perhaps future research can leverage the semantics and develop better strategies for determining when to memoize.

Huggies is useful because it provides a working prototype and clarifies implementation issues such as overloading, recursion, and polymorphism. The implementation techniques used in Huggies, or even the source code, provide a basis for future refinement.

Acknowledgements

We wish to thank Simon Peyton Jones and Zena Ariola for the original discussions that fostered this research; Bobbie Lavender, Erik Meijer, Koen Claessen, John Matthews, Mark Shields, and the "half-baked talk" attendees for their valuable suggestions; Alastair Reid for his extensive comments on both our code and paper. John Hughes, Mark Jones, and Conal Elliot, for useful insights; the referees of the Haskell Workshop for their helpful comments; and Tito Autrey for the title "Huggies".

References

[1] Ariola, Z. M., and Felleisen, M. The call-by-need lambda calculus. *Journal of Functional Programming 1*, 1 (1993), 1–000.

[2] Berry, D. The Edinburgh SML library. Tech. Rep. ECS-LFCS-91-148, Department of Computer Science, The University of Edinburgh, 1991.

[3] Elliott, C., and Hudak, P. Functional reactive animation. To appear in *The International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997).

[4] Gunter, C. A. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Science. The MIT Press, 1992.

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

[6] Hudak, P., Peterson, J., and Fasel, J. A gentle introduction to Haskell. Available at www.haskell.org, Dec. 1997.

[7] Hughes, R. J. M. Lazy memo-functions. In *The Conference on Functional Programming and Computer Architecture* (Nancy, France, Sept. 1985), Springer-Verlag.

[8] Hutton, G., and Meijer, E. Monadic parser combinators. Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[9] John Peterson, e. Report on the programming language haskell: A non-strict, purely functional language, version 1.4. Available at www.haskell.org, Apr. 1997.

[10] Jones, M. P. The implementation of the Gofer functional programming system. Tech. Rep. YALEU/DCS/RR-1030, Deptartment of Computer Science, Yale University, May 1994.

[11] Jones, S. P., and Meijer, E. Henk: A typed intermediate language. To appear in *The Workshop on Types in Compilation* (Amsterdam, The Netherlands, June 1997).

[12] Keller, R. M., and Sleep, M. R. Applicative caching: Programmer control of object sharing and lifetime in distributed implementations of applicative languages. In *The Conference on Functional Programming and Computer Architecture* (Wentworth-by-the-sea, Portsmouth, New Hampshire, Oct. 1981).

[13] Launchbury, J., and Jones, S. P. Lazy functional state threads. In *Programming Languages Design and Implementation* (Orlando, Florida, 1994), ACM Press.

[14] Michie, D. Memo functions and machine learning. *Nature*, 218 (Apr. 1968), 19–22.

[15] van Dalen, L. Incremental evaluation through memoization. Master's thesis, Department of Computer Science, Utrecht University, The Netherlands, 1992.