Selective Memoization*

Umut A. Acar Guy E. Blelloch Robert Harper

16 January 2003
CMU-CS-04-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

We present a framework for applying memoization selectively. The framework provides programmer control over equality, space usage, and identification of precise dependences so that memoization can be applied according to the needs of an application. Two key properties of the framework are that it is efficient and yields programs whose performance can be analyzed using standard techniques.

We describe the framework in the context of a functional language and an implementation as an SML library. The language is based on a modal type system and allows the programmer to express programs that reveal their true data dependences when executed. The SML implementation cannot support this modal type system statically, but instead employs run-time checks to ensure correct usage of primitives.

*This research was supported in part by NSF grants CCR-9706572, CCR-0085982, and CCR-0122581.

Keywords: Memoization, Functional Languages, Incremental Computing

1 Introduction

Memoization is a fundamental and powerful technique for result re-use. It dates back a half century [6, 20, 21] and has been used extensively in many areas such as dynamic programming [3, 8, 9, 19], incremental computation [10, 31, 11, 33, 15, 1, 34, 18, 13, 2], and many others [7, 22, 16, 24, 18]. In fact, lazy evaluation provides a limited form of memoization [17].

Although memoization can dramatically improve performance and can require only small changes to the code, no language or library support for memoization has gained broad acceptance. Instead, many successful uses of memoization rely on application-specific support code. The underlying reason for this is one of control: since memoization is all about performance, the user must be able to control the performance of memoization. Many subtleties of memoization, including the cost of equality checking and the cache replacement policy for memo tables, can make the difference between exponential and linear running time.

To be general and widely applicable a memoization framework must provide control over these three areas: (1) the kind and cost of equality tests; (2) the identification of precise dependences between the input and the output of memoized code; and (3) space management. Control over equality tests is critical, because this is how re-usable results are identified. Control over identification of precise dependences is important to maximize result reuse. Being able to control when memo tables or individual entries are purged is critical, because otherwise the user will not know whether or when results are re-used.

In this paper, we propose a framework for memoization that provides control over equality and identification of dependences, and some control over space management. We study the framework in the context of a small language called MFL and provide an implementation for it in the Standard ML language. We also prove the type safety and correctness of MFL—*i.e.,* that the semantics are preserved with respect to a non-memoized version. As an example, we show how to analyze the performance of a memoized version of Quicksort within our framework.

In the next section we describe background and related work. In Section 3 we introduce our framework via some examples. In Section 4 we formalize the MFL language and discuss its safety, correctness, and performance properties. In Section 5 we present a simple implementation of the framework as a Standard ML library. In Section 6 we discuss how the framework might be extended to allow for better control of space usage, and discuss the relationship of this work to our previous work on adaptive computation [2].

2 Background and Related Work

A typical memoization scheme maintains a memo table mapping argument values to previously computed results. This table is consulted before each function call to determine if the particular argument is in the table. If so, the call is skipped and the result is returned; otherwise the call is performed and its result is added to the table. The semantics and implementation of the memo lookup are critical to performance. Here we review some key issues in implementing memoization efficiently.

Equality. Any memoization scheme needs to search a memo table for a match to the current arguments. Such a search will, at minimum, require a test for equality. Typically it will also require some form of hashing. In standard language implementations testing for equality on structures, for example, can require traversing the whole structure. The cost of such an equality test can negate the advantage of memoizing and may even change the asymptotic behavior of the function. A few approaches have been proposed to alleviate this problem. The first is based on the fact that for memoization equality need not be exact—it can return unequal when two arguments are actually equal. The implementation could therefore decide to skip the test if the equality is too expensive, or could use a conservative equality test, such as "location" equality. The problem with such approaches is that whether a match is found could depend on particulars of the implementation and will surely not be evident to the programmer.

Another approach for reducing the cost of equality tests is to ensure that there is only one copy of every value, via a technique known as "hash consing" [12, 4, 32]. If there is only one copy, then equality can be implemented by comparing locations. In fact, the location can also be used as a key to a hash table. In theory, the overhead of hash-consing is constant in the expected case (expectation is over internal randomization of hash functions). The reality, however, is rather different because of large memory demands of hash-consing and its interaction with garbage collection. In fact, several researchers have argued that hash-consing is too expensive for practical purposes [30, 30, 5, 23]. As an alternative to hash consing, Pugh proposed lazy structure sharing [30]. In lazy structure sharing whenever two equal values are compared, they are made to point to the same copy to speed up subsequent comparisons. As Pugh points out, the disadvantage of this approach is that the performance depends on the order comparisons and thus it is difficult to analyze.

We note that even with hash-consing, or any other method, it remains critical to define equality on all types including reals and functions. Claiming that functions are never equivalent, for example, is not satisfactory because the result of a call involving some function as a parameter will never be re-used.

Precise Dependences. To maximize result re-use, the result of a function call must be stored with respect to its true dependences. This issue arises when the function examines only parts or an approximation of its parameter. To enable "partial" equality checks, the unexamined parts of the parameter should be disregarded. To increase the likelihood of result re-use, one should be able to match on the approximation, rather than the parameter itself. As an example, consider the code

```
fun f(x,y,z) = if (x > 0) then fy(y) else fz(z)
```

The result of `f` depends on either `(x,y)` or `(x,z)`. Also, it depends on an approximation of `x`–whether or not it is positive–rather than its exact value. Thus, the memo entry `(7,11,20)` should match `(7,11,30)` or `(4,11,50)` since, when `x` is positive, the result depends only on `y`.

Several researchers have remarked that partial matching can be very important in some applications [26, 25, 1, 13]. Abadi, Lampson, Lévy [1], and Heydon, Levin, Yu [13] have suggested program analysis methods for tracking dependences for this purpose. Although their technique is likely effective in catching potential matches, it does not provide a programmer controlled mechanism for specifying what dependences should be tracked. Also, their program analysis technique can change the asymptotic performance of a program, making it difficult to asses the effects of memoization.

Space management. Another problem with memoization is its space requirement. As a program executes, its memo tables can become large limiting the utility of memoization. To alleviate this problem, memo tables or individual entries should be disposed of under programmer control.

In some application, such as in dynamic programming, most result re-use occurs among the recursive calls of some function. Thus, the memo table of such a function can be disposed of whenever it terminates. In other applications, where result re-use is less structured, individual memo table entries should be purged according to a replacement policy [14, 30]. The problem is to determine what exact replacement policy should be used and to analyze the performance effects of the chosen policy. One widely used approach is to replace the least recently used entry. Other, more sophisticated, policies have also been suggested [30]. In general the replacement policy must be application-specific, because, for any fixed policy, there are programs whose performance is made worse by that choice [30].

3 A Framework for Selective Memoization

We present an overview of our framework via some examples. The framework extends a purely functional language with several constructs to support selective memoization. In this section, we use an extension to an ML-like language for the discussion. We formalize the core of this language and study its safety, soundness, and performance properties in Section 4.

The framework enables the programmer to determine precisely the dependences between the input and the result of a function. The main idea is to deem the parameters of a function as *resources* and provide primitives to explore incrementally any value, including the underlying value of a resource. This incremental exploration process reveals the dependences between the parameter of the function and its result.

The incremental exploration process is guided by types. If a value has the modal type $!\,\tau$, then the underlying value of type τ can be bound to an ordinary, unrestricted variable by the `let!` construct; this will create a dependence between the underlying value and the result. If a value has a product type, then its two parts can be bound to two resources using the `let*` construct; this creates no dependences. If the value is a sum type, then it can be case analyzed using the `mcase` construct, which branches according to the outermost form of the value and assigns the inner value to a resource; `mcase` creates a dependence on the outer form of the value of the resource. The key aspect of the `let*` and `mcase` is that they bind resources rather than ordinary variables.

<table>
<tr><th>Non-memoized</th><th>Memoized</th></tr>
<tr><td><pre>
fib:int -> int
fun fib (n)=

  if (n < 2) then n
  else fib(n-1) + fib(n-2)
</pre></td><td><pre>
mfib:!int -> int
mfun mfib (n')=
  let !n = n' in return (
    if (n < 2) then n
    else mfib(!(n-1)) + mfib(!(n-2))) end
</pre></td></tr>
<tr><td><pre>
f: int * int * int -> int
fun f (x, y, z)=
  if (x > 0) then
    fy y
  else
    fz z
</pre></td><td><pre>
mf:int * !int * !int -> int
mfun mf (x', y', z')=
  mif (x' > 0) then
    let !y = y' in return (fy y) end
  else
    let !z = z' in return (fz z) end
</pre></td></tr>
</table>

Figure 1: Fibonacci and expressing partial dependences.

Exploring the input to a function via `let!`, `mcase`, and `let*` builds a *branch* recording the dependences between the input and the result of the function. The `let!` adds to the branch the full value, the `mcase` adds the kind of the sum, and `let*` adds nothing. Consequently, a branch contains both data dependences (from `let!`'s) and control dependences (from `mcase`'s). When a `return` is encountered, the branch recording the revealed dependences is used to key the memo table. If the result is found in the memo table, then the stored value is returned, otherwise the body of the `return` is evaluated and the memo table is updated to map the branch to the result. The type system ensures that all dependences are made explicit by precluding the use of resources within `return`'s body.

As an example consider the Fibonacci function `fib` and its memoized counterpart `mfib` shown in Figure 1. The memoized version, `mfib`, exposes the underlying value of its parameter, a resource, before performing the two recursive calls as usual. Since the result depends on the full value of the parameter, it has a bang type. The memoized Fibonacci function runs in linear time as opposed to exponential time when not memoized.

Partial dependences between the input and the result of a function can be captured by using the incremental exploration technique. As an example consider the function `f` shown in Figure 1. The function checks whether `x` is positive or not and returns `fy(y)` or `fz(z)`. Thus the result of the function depends on an approximation of `x` (its sign) and on either `y` or `z`. The memoized version `mf` captures this by first checking if `x'` is positive or not and then exposing the underlying value of `y'` or `z'` accordingly. Consequently, the result will depend on the sign of `x'` and on either `y'` or `z'`. Thus if `mf` is called with parameters $(1, 5, 7)$ first and then $(2, 5, 3)$, the result will be found in the memo the second time, because when `x'` is positive the result depends only on `y'`. Note that `mif` construct used in this example is just a special case of the more general `mcase` construct.

A critical issue for efficient memoization is the implementation of memo tables along with lookup and update operations on them. In our framework we support expected constant time memo table lookup and update operations by representing memo tables using hashing. To do this, we require that the underlying type τ of a modal type $!\tau$ be an *indexable type*. An indexable type is associated with an injective function, called an *index function*, that maps each value of that type to a unique integer; this integer is called the *index* of the value. The uniqueness property of the indices for a given type ensures that two values are equal if and only if their indices are equal. In our framework, equality is only defined for indexable types. This enables us to implement memo tables as hash tables keyed by branches consisting of indices.

We assume that each primitive type comes with an index function. For examples, for integers, the identity function can be chosen as the index function. Composite types such as lists or functions must be *boxed* to obtain an indexable type. A boxed value of type τ has type τ `box`. When a box is created, it is assigned a unique locations (or tag), and this location is used as the unique index of that boxed value. For example, we can define boxed lists as follows.

```
datatype α blist' =  NIL | CONS of α * ((α blist') box)
type α blist = (α blist') box
```

Based on boxes we implement hash-consing as a form of memoization. For example, hash-consing for boxed lists can be implemented as follows.

```
hCons:  !α * !(α blist) -> α blist
mfun hCons (h', t') =
  let !h = h' and !t = t' in
    return (box (CONS(h,t))) end
  end
```

The function takes an item and a boxed list and returns the boxed list formed by consing them. Since the function is memoized, if it is ever called with two values that are already hash-consed, then the same result will be returned. The advantage of being able to define hash-consing as a memoized function is that it can be applied selectively.

To control space usage of memo tables, our framework gives the programmer a way to dispose of memo tables by conventional scoping. In our framework, each memoized function is allocated its own memo table. Thus, when the function goes out of scope, its memo table can be garbage collected. For example, in many dynamic-programming algorithms result re-use occurs between recursive calls of the same function. In this case, the programmer can scope the memoized function inside an auxiliary function so that its memo table is discarded as soon as the auxiliary function returns. As an example, consider the standard algorithm for the Knapsack Problem `ks` and its memoized version `mks` Figure 2. Since result sharing mostly occurs among the recursive calls of `mks`, it can be scoped in some other function that calls `mks`; once `mks` returns its memo table will go out of scope and can be discarded.

We note that this technique gives only partial control over space usage. In particular it does not give control over when individual memo table entries are purged. In Section 6, we

Non-memoized	Memoized

`ks:  int * ((int*real) list) -> int`
`fun ks (c,l) =`
 	
`  case l of`
`    nil => 0`
`   |(w,v)::t =>`
`    if (c < w) then`
`      ks(c,t)`
`    else`
`      let v1 = ks(c,t)`
`          v2 = v + ks(c-w,t)`
`      in`
`        if (v1>v2) then v1`
`        else v2`
`      end`
	`mks:  !int * !((int*real) list) -> int`
	`mfun mks (c',l')`
	`  let !c = c' and !l = l' in return (`
	`    case (unbox l) of`
	`      NIL => 0`
	`    | CONS((w,v),t) =>`
	`      if (c < w) then`
	`        mks(!c,!t)`
	`      else`
	`        let v1 = mks(!c,!t)`
	`            v2 = v + mks(!(c-w),!t)`
	`        in`
	`          if (v1 > v2) then v1`
	`          else v2`
	`        end) end`

Figure 2: Memo tables for memoized Knapsack can be discarded at completion.

discuss how the framework might be extended so that each memo table is managed according to a programmer specified caching scheme. The basic idea is to require the programmer to supply a caching scheme as a parameter to the `mfun` and maintain the memo table according to the chosen caching scheme.

Memoized Quicksort. As a more sophisticated example, we consider Quicksort. Figure 3 shows an implementation of the Quicksort algorithm and its memoized counterpart. The algorithm first divides its input into two lists containing the keys less than the pivot, and greater than the pivot by using the filter function `fil`. It then sorts the two sublists, and returns the concatenation of the results. The memoized filter function `mfil` uses hash-consing to ensure that there is only one copy of each result list. The memoized Quicksort algorithm `mqs` exposes the underlying value of its parameter and is otherwise similar to `qs`. Note that `mqs` does not build its result via hash-consing—it can output two copies of the same result. Since in this example the output of `mqs` is not consumed by any other function, there is no need to do so. Even if the result were consumed by some other function, one can choose not to use hash-consing because operations such as insertions to and deletions from the input list will surely change the result of Quicksort.

When the memoized Quicksort algorithm is called on "similar" inputs, one would expect that some of the results would be re-used. Indeed, we show that the memoized Quicksort algorithm computes its result in expected linear time when its input is obtained from a previous input by inserting a new key at the beginning. Here the expectation is over all permutations of the input list and also the internal randomization of the hash functions used to implement the memo tables. For the analysis, we assume, without loss of generality, that all keys in the list are unique.

Non-memoized

```
fil:  int->bool * int list -> int list
fun fil (g:int->bool, l:int list) =
  case l of
    nil => nil
  | h::t =>
    let tt = fil(g,t) in
      if (g h) then h::tt
      else tt
    end

qs:  int list -> int list
fun qs (l) =

  case l of
    nil => nil
  | cons(h,t) =>
    let s = fil(fn x=>x<h,t)
        g = fil(fn x=>x>=h,t)
    in
      (qs s)@(h::(qs g))
    end
```

Memoized

```
empty = box NIL

mfil:  int->bool * int blist -> int blist
fun mfil (g,l) =
  case (unbox l) of
    NIL => empty
  | CONS(h,t) =>
    let tt = mfil(g,t) in
      if (g h) then hCons(h,tt)
      else tt
    end

mqs:  !(int blist) -> int blist
mfun mqs (l':!int blist) =
  let !l = l' in return (
    case (unbox l) of
      NIL => NIL
    | CONS(h,t) =>
      let s = mfil(fn x=>x<h,t)
          g = mfil(fn x=>x>=h,t)
      in
        (mqs !s)@(h::(mqs !g))
      end) end
```

Figure 3: The Quicksort algorithm.

Theorem 1
Let L be a list and let $L' = [a, L]$. Consider running memoized Quicksort on L and then on L'. The running time of Quicksort on the modified list L' is expected $O(n)$ where n is the length of L'.

Proof: Consider the recursion tree of Quicksort with input L, denoted $Q(L)$, and label each node with the pivot of the corresponding recursive call (see Figure 4 for an example). Consider any pivot (key) p from L and let L_p denote the keys that precede p in L. It is easy to see that a key k is in the subtree rooted at p if and only if the following two properties are satisfied for any key $k' \in L_p$.

1. If $k' < p$ then $k > k'$, and
2. if $k' > p$ then $k < k'$.

Of the keys that are in the subtree of p, those that are less than p are in its left subtree and those greater than p are in its right subtree.

Now consider the recursion tree $Q(L')$ for $L' = [a, L]$ and let p be any pivot in $Q(L')$. Suppose $p < a$ and let k be any key in the left subtree of p in $Q(L)$. Since $k < p$, by

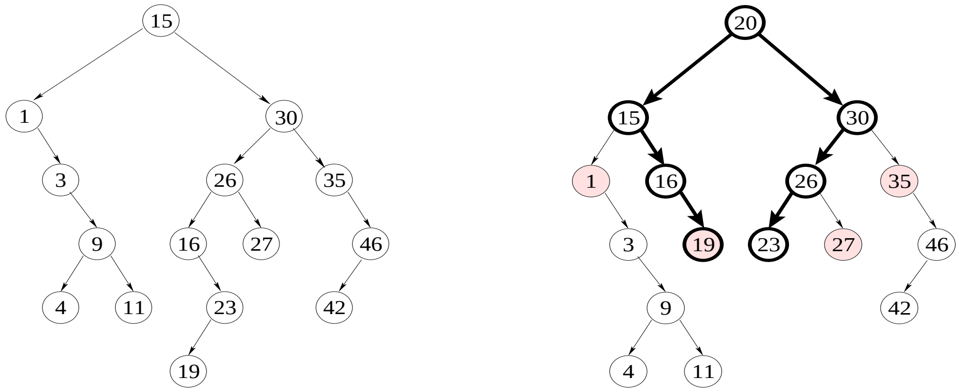

Figure 4: The recursion tree for Quicksort with inputs $L = [15, 30, 26, 1, 3, 16, 27, 9, 35, 4, 46, 23, 11, 42, 19]$ (left) and $L' = [20, L]$ (right).

the two properties k is in the left subtree of p in $Q(L')$. Similarly if $p > a$ then any k in the right subtree of p in $Q(L)$ is also in the right subtree of p in $Q(L')$. Since filtering preserves the respective order of keys in the input list, for any p, $p < a$, the input to the recursive call corresponding to its left child will be the same. Similarly, for $p > a$, the input to the recursive call corresponding to its right child will be the same. Thus, when sorting L' these recursive calls will find their results in the memo. Therefore only recursive calls corresponding to the root, to the children of the nodes in the rightmost spine of the left subtree of the root, and the children of the nodes in the leftmost spine of the right subtree of the root may be executed (the two spines are shown with thick lines in Figure 4). Furthermore, the results for the calls adjacent to the spines will be found in the memo.

Consider the calls whose results are not found in the memo. In the worst case, these will be all the calls along the two spines. Consider the sizes of inputs for the nodes on a spine and define the random variables $X_1 \ldots X_k$ such that X_i is the least number of recursive calls (nodes) performed for the input size to become $\left(\frac{3}{4}\right)^i n$ or less after it first becomes $\left(\frac{3}{4}\right)^{(i-1)} n$ or less. Since $k \leq \lceil \log_{4/3} n \rceil$, the total and the expected number of operations along a spine are

$$\begin{aligned} C(n) &\leq \sum_{i=1}^{\lceil \log_{4/3} n \rceil} X_i \left(\frac{3}{4}\right)^{i-1} n, \text{ and} \\ E[C(n)] &\leq \sum_{i=1}^{\lceil \log_{4/3} n \rceil} E[X_i] \left(\frac{3}{4}\right)^{i-1} n. \end{aligned}$$

Since the probability that the pivot lies in the middle half of the list is $\frac{1}{2}$, $E[X_i] \leq 2$ for

$i \geq 1$, and we have

$$E[C(n)] \quad \leq \quad \sum_{i=1}^{\lceil \log_{4/3} n \rceil} 2 \left(\frac{3}{4}\right)^{i-1} n.$$

Thus, $E[C(n)] = O(n)$ This bound holds for both spines; therefore the number of operations due to calls whose results are not found in the memo is $O(n)$. Since each operation, including hash-consing, takes expected constant time, the total time of the calls whose results are not in the memo is $O(n)$. Now, consider the calls whose results are found in the memo, each such call will be on a spine or adjacent to it, thus there are an expected $O(\log n)$ such calls. Since, the memo table lookup overhead is expected constant time the total cost for these is $O(\log n)$. We conclude that Quicksort will take expected $O(n)$ time for sorting the modified list L'. ■

It is easy to extend the theorem to show that the $O(n)$ bound holds for an insertion anywhere in the list. Although, this bound is better than a complete rerun, which would take $O(n \log n)$, we would like to achieve $O(\log n)$. In Section 6 we discuss how a combination of memoization and adaptivity [2] may be used to reduce the expected cost of a random insertion to $O(\log n)$.

4 The MFL Language

In this section we study a small functional language, called MFL, that supports selective memoization. MFL distinguishes memoized from non-memoized code, and is equipped with a modality for tracking dependences on data structures within memoized code. This modality is central to our approach to selective memoization, and is the focus of our attention here. The main result is a soundness theorem stating that memoization does not affect the outcome of a computation compared to a standard, non-memoizing semantics. We also show that the memoization mechanism of MFL causes a constant factor slowdown compared to a standard, non-memoizing semantics.

4.1 Abstract Syntax

The abstract syntax of MFL is given in Figure 5. The meta-variables x and y range over a countable set of *variables*. The meta-variables a and b range overf a countable set of *resources*. (The distinction will be made clear below.) The meta-variable l ranges over a countable set of *locations*. We assume that variables, resources, and locations are mutually disjoint. The binding and scope conventions for variables and resources are as would be expected from the syntactic forms. As usual we identify pieces of syntax that differ only in their choice of bound variable or resource names. A term or expression is *resource-free* if and only if it contains no free resources, and is *variable-free* if and only if it contains no free variables. A *closed* term or expression is both resource-free and variable-free; otherwise it is *open*.

$$
\begin{array}{llll}
\textit{Indexable Types} & \eta & ::= & 1 \mid \texttt{int} \mid \ldots \\
\textit{Types} & \tau & ::= & \eta \mid \,!\,\eta \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu u.\tau \mid \tau_1 \to \tau_2 \\
\textit{Operators} & o & ::= & \texttt{+} \mid \texttt{-} \mid \ldots \\
\textit{Expressions} & e & ::= & \texttt{return}(t) \mid \texttt{let}\,!\,x{:}\eta\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} \mid \\
 & & & \texttt{let}\,a_1{:}\tau_1{\times}a_2{:}\tau_2\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} \mid \\
 & & & \texttt{mcase}\,t\,\texttt{of}\,\texttt{inl}\,(a_1{:}\tau_1) \Rightarrow e_1 \mid \texttt{inr}\,(a_2{:}\tau_2) \Rightarrow e_2\,\texttt{end} \\
\textit{Terms} & t & ::= & v \mid o(t_1, \ldots, t_n) \mid \langle t_1, t_2 \rangle \mid \texttt{mfun}\,f\,(a{:}\tau_1){:}\tau_2\,\texttt{is}\,e\,\texttt{end} \mid \\
 & & & t_1\,t_2 \mid \,!\,t \mid \texttt{inl}_{\tau_1+\tau_2} t \mid \texttt{inr}_{\tau_1+\tau_2} t \mid \texttt{roll}(t) \mid \texttt{unroll}(t) \\
\textit{Values} & v & ::= & x \mid a \mid \star \mid n \mid \,!\,v \mid \langle v_1, v_2 \rangle \mid \texttt{mfun}_l\,f\,(a{:}\tau_1){:}\tau_2\,\texttt{is}\,e\,\texttt{end}
\end{array}
$$

Figure 5: The abstract syntax of MFL.

The types of MFL include `1` (unit), `int`, products and sums, recursive data types $\mu u.\tau$, memoized function types, and bang types $!\,\eta$. MFL distinguishes *indexable types*, denoted η, as those that accept an injective function, called an *index function*, whose co-domain is integers. The underlying type of a bang type $!\,\eta$ is restricted to be an indexable type. For `int` type, identity serves as an index function; for `1` (unit) any constant function can be chosen as the index function. For non-primitive types an index can be supplied by boxing values of these types. Boxed values would be allocated in a store and the unique location of a box would serve as an index for the underlying value. With this extension the indexable types would be defined as $\eta ::= 1 \mid \texttt{int} \mid \tau\ \texttt{box}$. Although supporting boxed types is critical for practical purposes, we do not formalize this here to focus on the main ideas.

The syntax is structured into *terms* and *expressions*, in the terminology of Pfenning and Davies [28]. Roughly speaking, terms evaluate independently of their context, as in ordinary functional programming, whereas expressions are evaluated relative to a memo table. Thus, the body of a memoized function is an expression, whereas the function itself is a term. Note, however, that the application of a function is a *term*, not an *expression*; this corresponds to the encapsulation of memoization with the function, so that updating the memo table is benign. In a more complete language we would include case analysis and projection forms among the terms, but for the sake of simplicity we include these only as expressions. We would also include a plain function for which the body is a term. Note that every term is trivially an expression; the `return` expression is the inclusion.

4.2 Static Semantics

The type structure of MFL extends the framework of Pfenning and Davies [28] with a "necessitation" modality, $!\,\eta$, which is used to track data dependences for selective memoization. This modality does not correspond to a monadic interpretation of memoization effects ($\bigcirc\,\tau$ in the notation of Pfenning and Davies), though one could imagine adding such a modality to the language. The introductory and eliminatory forms for necessity are standard, namely $!\,t$ for introduction, and $\mathtt{let}\;!\,x{:}\eta\;\mathtt{be}\;t\;\mathtt{in}\;e\;\mathtt{end}$ for elimination.

Our modality demands that we distinguish variables from resources. Variables in MFL correspond to the "validity", or "unrestricted", context in modal logic, whereas resources in MFL correspond to the "truth", or "restricted" context. An analogy may also be made to the judgmental presentation of linear logic [27, 29]: variables correspond to the intuitionistic context, resources to the linear context.[1]

The inclusion, $\mathtt{return}(t)$, of terms into expressions has no analogue in pure modal logic, but is specific to our interpretation of memoization as a computational effect. The typing rule for $\mathtt{return}(t)$ requires that t be resource-free to ensure that any dependence on the argument to a memoized function is made explicit in the code before computing the return value of the function. In the first instance, resources arise as parameters to memoized functions, with further resources introduced by their incremental decomposition using $\mathtt{let}\times$ and `mcase`. These additional resources track the usage of as-yet-unexplored parts of a data structure. Ultimately, the complete value of a resource may be accessed using the `let!` construct, which binds its value to a variable, which may be used without restriction. In practice this means that those parts of an argument to a memoized function on whose value the function depends will be given modal type. However, it is not essential that all resources have modal type, nor that the computation depend upon every resource that does have modal type.

The static semantics of MFL consists of a set of rules for deriving typing judgments of the form $\Gamma;\Delta \vdash t : \tau$, for terms, and $\Gamma;\Delta \vdash e : \tau$, for expressions. In these judgments Γ is a *variable type assignment*, a finite function assigning types to variables, and Δ is a *resource type assignment*, a finite function assigning types to resources. The rules for deriving these judgments are given in Figures 6 and 7.

4.3 Dynamic Semantics

The dynamic semantics of MFL formalizes selective memoization. Evaluation is parameterized by a store containing memo tables that track the behavior of functions in the program. Evaluation of a function expression causes an empty memo table to be allocated and associated with that function. Application of a memoized function is affected by, and may affect, its associated memo table. Should the function value become inaccessible, so also is its associated memo table, and hence the storage required for both can be reclaimed.

Unlike conventional memoization, however, the memo table is keyed by control flow information rather than by the values of arguments to memoized functions. This is the key

[1]Note, however, that we impose no linearity constraints in our type system!

$$\frac{(\Gamma(x) = \tau)}{\Gamma; \Delta \vdash x{:}\tau}\ \textbf{(variable)} \qquad \frac{(\Delta(a) = \tau)}{\Gamma; \Delta \vdash a{:}\tau}\ \textbf{(resource)}$$

$$\frac{}{\Gamma; \Delta \vdash n : \mathtt{int}}\ \textbf{(number)} \qquad \frac{}{\Gamma; \Delta \vdash \star : 1}\ \textbf{(unit)}$$

$$\frac{\Gamma; \Delta \vdash t_i : \tau_i \ \ (1 \leq i \leq n) \qquad \vdash_o o : (\tau_1, \ldots, \tau_n)\ \tau}{\Gamma; \Delta \vdash o(t_1, \ldots, t_n) : \tau}\ \textbf{(primitive)}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \tau_1 \quad \Gamma; \Delta \vdash t_2 : \tau_2}{\Gamma; \Delta \vdash \langle t_1, t_2 \rangle : \tau_1 \times \tau_2}\ \textbf{(pair)}$$

$$\frac{\Gamma,\ f{:}\tau_1 \rightarrow \tau_2; \Delta, a{:}\tau_1 \vdash e : \tau_2}{\Gamma; \Delta \vdash \mathtt{mfun}\, f\ (a{:}\tau_1){:}\tau_2\ \mathtt{is}\, e\, \mathtt{end} : \tau_1 \rightarrow \tau_2}\ \textbf{(fun)}$$

$$\frac{\Gamma, f{:}\tau_1 \rightarrow \tau_2; \Delta, a{:}\tau_1 \vdash e : \tau_2}{\Gamma; \Delta \vdash \mathtt{mfun}_l\, f(a{:}\tau_1){:}\tau_2\ \mathtt{is}\, e\, \mathtt{end} : \tau_1 \rightarrow \tau_2}\ \textbf{(fun value)}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma; \Delta \vdash t_2 : \tau_1}{\Gamma; \Delta \vdash t_1\, t_2 : \tau_2}\ \textbf{(apply)}$$

$$\frac{\Gamma; \emptyset \vdash t : \eta}{\Gamma; \Delta \vdash\ !\, t :\ !\, \eta}\ \textbf{(bang)}$$

$$\frac{\Gamma; \Delta \vdash t : \tau_1}{\Gamma; \Delta \vdash \mathtt{inl}_{\tau_1+\tau_2} t : \tau_1 + \tau_2}\ \textbf{(sum/inl)} \qquad \frac{\Gamma; \Delta \vdash t : \tau_2}{\Gamma; \Delta \vdash \mathtt{inr}_{\tau_1+\tau_2} t : \tau_1 + \tau_2}\ \textbf{(sum/inr)}$$

$$\frac{\Gamma; \Delta \vdash t : [\mu u.\tau/u]\tau}{\Gamma; \Delta \vdash \mathtt{roll}(t) : \mu u.\tau}\ \textbf{(roll)} \qquad \frac{\Gamma; \Delta \vdash t : \mu u.\tau}{\Delta \vdash \mathtt{unroll}(t) : [\mu u.\tau/u]\tau}\ \textbf{(unroll)}$$

Figure 6: Typing judgments for terms.

to supporting selective memoization. Expression evaluation is essentially an exploration of the available resources culminating in a resource-free term that determines its value. Since the exploration is data-sensitive, only certain aspects of the resources may be relevant to a particular outcome. For example, a memoized function may take a pair of integers as argument, with the outcome determined independently of the second component in the case that the first is positive. By recording control-flow information during evaluation, we may use it to provide selective memoization.

For example, in the situation just described, all pairs of the form $\langle 0, v \rangle$ should map to the same result value, irrespective of the value v. In conventional memoization the

$$\frac{\Gamma;\emptyset \vdash t : \tau}{\Gamma;\Delta \vdash \mathtt{return}(t) : \tau}\ \textbf{(return)}$$

$$\frac{\Gamma;\Delta \vdash t : !\,\eta \quad \Gamma, x{:}\eta;\Delta \vdash e : \tau}{\Gamma;\Delta \vdash \mathtt{let}\ !\,x{:}\eta\ \mathtt{be}\ t\ \mathtt{in}\ e\ \mathtt{end} : \tau}\ \textbf{(let!)}$$

$$\frac{\Gamma;\Delta \vdash t : \tau_1 \times \tau_2 \quad \Gamma;\Delta, a_1{:}\tau_1, a_2{:}\tau_2 \vdash e : \tau}{\Gamma;\Delta \vdash \mathtt{let}\ a_1{:}\tau_1 \times a_2{:}\tau_2\ \mathtt{be}\ t\ \mathtt{in}\ e\ \mathtt{end} : \tau}\ \textbf{(let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\Gamma;\Delta & \vdash & t : \tau_1 + \tau_2 \\ \Gamma;\Delta, a_1{:}\tau_1 & \vdash & e_1 : \tau \\ \Gamma;\Delta, a_2{:}\tau_2 & \vdash & e_2 : \tau\end{array}}{\Gamma;\Delta \vdash \mathtt{mcase}\ t\ \mathtt{of}\ \mathtt{inl}\ (a_1{:}\tau_1) \Rightarrow e_1 \mid \mathtt{inr}\ (a_2{:}\tau_2) \Rightarrow e_2\ \mathtt{end} : \tau}\ \textbf{(case)}$$

Figure 7: Typing judgments for expressions.

memo table would be keyed by the pair, with the result that redundant computation is performed in the case that the function has not previously been called with v, even though the value of v is irrelevant to the result! In our framework we instead key the memo table by a "branch" that records sufficient control flow information to capture the general case. Whenever we encounter a `return` statement, we query the memo table with the current branch to determine whether this result has been computed before. If so, we return the stored value; if not, we evaluate the `return` statement, and associate that value with that branch in the memo table for future use. It is crucial that the returned term not contain any resources so that we are assured that its value does not change across calls to the function.

The dynamic semantics of MFL is given by a set of rules for deriving judgments of the form $\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'$ (for terms) and $\sigma, l{:}\beta, e \Downarrow^{\mathsf{e}} v, \sigma'$ (for expressions). The rules for deriving these judgments are given in Figures 8 and 9. These rules make use of branches, memo tables, and stores, whose precise definitions are as follows.

A *simple branch* is a list of *simple events* corresponding to "choice points" in the evaluation of an expression.

$$\begin{array}{llcl} \textit{Simple Event} & \varepsilon & ::= & !v \mid \mathtt{inl} \mid \mathtt{inr} \\ \textit{Simple Branch} & \beta & ::= & \bullet \mid \varepsilon \cdot \beta \end{array}$$

We write $\beta\frown\varepsilon$ to stand for the extension of β with the event ε at the end.

A *memo table*, θ, is a finite function mapping simple branches to values. We write $\theta[\beta \mapsto v]$, where $\beta \notin \mathrm{dom}(\theta)$, to stand for the extension of θ with the given binding for β. We write $\theta(\beta)\uparrow$ to mean that $\beta \notin \mathrm{dom}(\theta)$.

A *store*, σ, is a finite function mapping *locations*, l, to memo tables. We write $\sigma[l \mapsto \theta]$, where $l \notin \mathrm{dom}(\sigma)$, to stand for the extension of σ with the given binding for l. When $l \in \mathrm{dom}(\sigma)$, we write $\sigma[l \leftarrow \theta]$ for the store σ that maps l to θ and $l' \neq l$ to $\sigma(l')$.

Term evaluation is largely standard, except for the evaluation of (memoizing) functions and applications of these to arguments. Evaluation of a memoizing function term allocates a fresh memo table, which is then associated with the function's value. Expression evaluation is initiated by an application of a memoizing function to an argument. The function value determines the memo table to be used for that call. Evaluation of the body is performed relative to that table, initiating with the null branch.

Expression evaluation is performed relative to a "current" memo table and branch. When a `return` statement is encountered, the current memo table is consulted to determine whether or not that branch has previously been taken. If so, the stored value is returned; otherwise, the argument term is evaluated, stored in the current memo table at that branch, and the value is returned. The `let!` and `mcase` expressions extend the current branch to reflect control flow. Since `let!` signals dependence on a complete value, that value is added to the branch. Case analysis, however, merely extends the branch with an indication of which case was taken. The `let×` construct does not extend the branch, because no additional information is gleaned by splitting a pair.

4.4 Soundness of MFL

We will prove the soundness of MFL relative to a non-memoizing semantics for the language. It is straightforward to give a purely functional semantics to MFL by an inductive definition of the relations $t \Downarrow^{\mathsf{t}}_{\mathsf{p}} v$ and $e \Downarrow^{\mathsf{e}}_{\mathsf{p}} v$, where v is a *pure value* with no location subscripts (see, for example, [28]). We will show that (Theorem 5) memoization does not affect the outcome of evaluation as compared to the non-memoized semantics. To make this precise, we must introduce some additional machinery.

The *underlying term*, t^-, of a term, t, is obtained by erasing all location subscripts on function values occurring within t. The *underlying expression*, e^-, of an expression, e, is defined in the same way. As a special case, the *underlying value*, v^-, of a value, v, is the underlying term of v regarded as a term. It is easy to check that every pure value arises as the underlying value of some impure value. Note that passage to the underlying term or expression obviously commutes with substitution. The *underlying branch*, β^-, of a simple branch, β, is obtained by replacing each event of the form $!\, v$ in β by the corresponding underlying event, $!\, (v^-)$.

The partial *access functions*, $t \mathbin{@} \beta$ and $e \mathbin{@} \beta$, where β is a simple branch, and t and e are variable-free (but not necessarily resource-free), are defined as follows. The definition may be justified by lexicographic induction on the structure of the branch followed by the

$$\frac{}{\sigma, \star \Downarrow^{\mathsf{t}} \star, \sigma}\ \textbf{(unit)} \qquad \frac{}{\sigma, n \Downarrow^{\mathsf{t}} n, \sigma}\ \textbf{(number)}$$

$$\frac{\sigma, t_1 \quad \Downarrow^{\mathsf{t}} \quad v_1, \sigma_1 \quad \ldots \quad \sigma_{n-1}, t_n \quad \Downarrow^{\mathsf{t}} \quad v_n, \sigma_n}{\sigma, o(t_1, \ldots, t_n) \Downarrow^{\mathsf{t}} \mathtt{app}(o, (v_1, \ldots, v_n), \sigma_n}\ \textbf{(primitive)}$$

$$\frac{\begin{array}{c}\sigma, t_1 \quad \Downarrow^{\mathsf{t}} \quad v_1, \sigma' \\ \sigma', t_2 \quad \Downarrow^{\mathsf{t}} \quad v_2, \sigma''\end{array}}{\sigma, \langle t_1, t_2 \rangle \Downarrow^{\mathsf{t}} \langle v_1, v_2 \rangle, \sigma''}\ \textbf{(pair)}$$

$$\frac{(l \notin \mathrm{dom}(\sigma))}{\sigma, \mathtt{mfun}\, f\ (a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end} \Downarrow^{\mathsf{t}} \mathtt{mfun}_l\ f\,(a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end}, \sigma[l \mapsto \emptyset]}\ \textbf{(fun)}$$

$$\frac{(l \in \mathrm{dom}(\sigma))}{\sigma, \mathtt{mfun}_l\ f\,(a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end} \Downarrow^{\mathsf{t}} \mathtt{mfun}_l\ f\,(a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end}, \sigma}\ \textbf{(fun val)}$$

$$\frac{\begin{array}{rcl}\sigma, t_1 & \Downarrow^{\mathsf{t}} & v_1, \sigma_1 \\ \sigma_1, t_2 & \Downarrow^{\mathsf{t}} & v_2, \sigma_2 \\ \sigma_2, l\mathtt{:}\bullet, [v_1, v_2/f, a]\ e & \Downarrow^{\mathsf{e}} & v, \sigma' \\ \multicolumn{3}{c}{(v_1 = \mathtt{mfun}_l\ f\,(a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end})}\end{array}}{\sigma, t_1\ t_2 \Downarrow^{\mathsf{t}} v, \sigma'}\ \textbf{(apply)}$$

$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, !\ t \Downarrow^{\mathsf{t}}\ !\ v, \sigma'}\ \textbf{(bang)}$$

$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \mathtt{inl}_{\tau_1+\tau_2} t \Downarrow^{\mathsf{t}} \mathtt{inl}_{\tau_1+\tau_2} v, \sigma'}\ \textbf{(case/inl)} \qquad \frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \mathtt{inr}_{\tau_1+\tau_2} t \Downarrow^{\mathsf{t}} \mathtt{inr}_{\tau_1+\tau_2} v, \sigma'}\ \textbf{(case/inr)}$$

$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \mathtt{roll}(t) \Downarrow^{\mathsf{t}} \mathtt{roll}(v), \sigma'}\ \textbf{(roll)} \qquad \frac{\sigma, t \Downarrow^{\mathsf{t}} \mathtt{roll}(v), \sigma'}{\sigma, \mathtt{unroll}(t) \Downarrow^{\mathsf{t}} v, \sigma'}\ \textbf{(unroll)}$$

Figure 8: Evaluation of terms.

size of the expression.

$$\begin{array}{rcl}
t \mathbin{@} \beta & = & e \mathbin{@} \beta \\
\multicolumn{3}{r}{(\textit{where}\ t = \mathtt{mfun}\, f\ (a\mathtt{:}\tau_1)\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end})} \\
\\
\mathtt{return}(t) \mathbin{@} \bullet & = & \mathtt{return}(t) \\
\mathtt{let}\ !\ x\mathtt{:}\tau\ \mathtt{be}\ t\ \mathtt{in}\ e\ \mathtt{end} \mathbin{@} \beta\hat{}\,!v & = & [v/x]e \mathbin{@} \beta \\
\mathtt{let}\, a_1\mathtt{:}\tau_1 \times a_2\mathtt{:}\tau_2\ \mathtt{be}\ t\ \mathtt{in}\ e\ \mathtt{end} \mathbin{@} \beta & = & e \mathbin{@} \beta \\
\mathtt{mcase}\, t\ \mathtt{of}\ \mathtt{inl}\ (a_1\mathtt{:}\tau_1)\ \Rightarrow\ e_1 \mid \mathtt{inr}\ (a_2\mathtt{:}\tau_2)\ \Rightarrow\ e_2\ \mathtt{end} \mathbin{@} \beta\hat{}\mathtt{inl} & = & e_1 \mathbin{@} \beta \\
\mathtt{mcase}\, t\ \mathtt{of}\ \mathtt{inl}\ (a_1\mathtt{:}\tau_1)\ \Rightarrow\ e_1 \mid \mathtt{inr}\ (a_2\mathtt{:}\tau_2)\ \Rightarrow\ e_2\ \mathtt{end} \mathbin{@} \beta\hat{}\mathtt{inr} & = & e_2 \mathbin{@} \beta
\end{array}$$

$$\frac{\sigma(l)(\beta) = v}{\sigma, l{:}\beta, \texttt{return}(t) \Downarrow^{\mathsf{e}} v, \sigma} \;\textbf{(return, found)}$$

$$\frac{\begin{array}{c}\sigma(l) = \theta \quad \theta(\beta)\uparrow \\ \sigma, t \Downarrow^{\mathsf{t}} v, \sigma' \\ \sigma'(l) = \theta'\end{array}}{\sigma, l{:}\beta, \texttt{return}(t) \Downarrow^{\mathsf{e}} v, \sigma'[l \leftarrow \theta'[\beta \mapsto v]]} \;\textbf{(return, not found)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & !\,v, \sigma' \\ \sigma', l{:}!v \cdot \beta, [v/x]e & \Downarrow^{\mathsf{t}} & v', \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{let}\ !\,x : \eta\ \texttt{be}\ t\ \texttt{in}\ e\ \texttt{end} \Downarrow^{\mathsf{e}} v', \sigma''} \;\textbf{(let!)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & v_1 \times v_2, \sigma' \\ \sigma', l{:}\beta, [v_1/a_1, v_2/a_2]e & \Downarrow^{\mathsf{e}} & v, \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{let}\ a_1 \times a_2\ \texttt{be}\ t\ \texttt{in}\ e\ \texttt{end} \Downarrow^{\mathsf{t}} v, \sigma''} \;\textbf{(let}\times\textbf{)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & \texttt{inl}_{\tau_1+\tau_2} v, \sigma' \\ \sigma', l{:}\texttt{inl} \cdot \beta, [v/a_1]e_1 & \Downarrow^{\mathsf{e}} & v_1, \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{mcase}\ t\ \texttt{of}\ \texttt{inl}\ (a_1 : \tau_1) \ \Rightarrow\ e_1 \mid \texttt{inr}\ (a_2 : \tau_2) \ \Rightarrow\ e_2\ \texttt{end} \Downarrow^{\mathsf{t}} v_1, \sigma''} \;\textbf{(case/inl)}$$

$$\frac{\begin{array}{rcl}\sigma, t & \Downarrow^{\mathsf{t}} & \texttt{inr}_{\tau_1+\tau_2} v, \sigma' \\ \sigma', l{:}\texttt{inr} \cdot \beta, [v/a_2]e_2 & \Downarrow^{\mathsf{e}} & v_2, \sigma''\end{array}}{\sigma, l{:}\beta, \texttt{mcase}\ t\ \texttt{of}\ \texttt{inl}\ (a_1 : \tau_1) \ \Rightarrow\ e_1 \mid \texttt{inr}\ (a_2 : \tau_2) \ \Rightarrow\ e_2\ \texttt{end} \Downarrow^{\mathsf{t}} v_2, \sigma''} \;\textbf{(case/inr)}$$

Figure 9: Evaluation of expressions.

This function will only be of interest in the case that $e \mathbin{@} \beta$ is a `return` expression, which, if well-typed, cannot contain free resources. Note that $(e \mathbin{@} \beta)^- = e^- \mathbin{@} \beta^-$, and similarly for values, v.

We are now in a position to justify a subtlety in the second `return` rule of the dynamic semantics, which governs the case that the returned value has not already been stored in the memo table. This rule extends, rather than updates, the memo table with a binding for the branch that determines this `return` statement within the current memoized function. But why, after evaluation of t, is this branch undefined in the revised store, σ'? If the term t were to introduce a binding for β in the memo table $\sigma(l)$, it could only do so by evaluating the very same `return` statement, which implies that there is an infinite loop, contradicting the assumption that the `return` statement has a value, v.

Lemma 2
If $\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'$, $\sigma(l)@\beta = \texttt{return(t)}$, and $\sigma(l)(\beta)$ is undefined, then $\sigma'(l)(\beta)$ is also undefined.

An *augmented branch*, γ, is an extension of the notion of branch in which we record the bindings of resource variables. Specifically, the argument used to call a memoized function is recorded, as are the bindings of resources created by pair splitting and case analysis. Augmented branches are inductively defined by the following grammar:

$$\begin{array}{lcll} \textit{Augmented Event} & \epsilon & ::= & \mathtt{(}v\mathtt{)} \mid \mathtt{!}\,v \mid \langle v_1, v_2\rangle \mid \mathtt{inl(}v\mathtt{)} \mid \mathtt{inr(}v\mathtt{)} \\ \textit{Augmented Branch} & \gamma & ::= & \bullet \mid \epsilon \cdot \gamma \end{array}$$

We write $\gamma \frown \epsilon$ for the extension of γ with ϵ at the end. There is an obvious *simplification* function, γ°, that yields the simple branch corresponding to an augmented branch by dropping "call" events, $\mathtt{(}v\mathtt{)}$, and "pair" events, $\langle v_1, v_2\rangle$, and by omitting the arguments to "injection" events, $\mathtt{inl(}v\mathtt{)}$, $\mathtt{inr(}v\mathtt{)}$. The *underlying augmented branch*, γ^-, corresponding to an augmented branch, γ, is defined by replacing each augmented event, ϵ, by its corresponding underlying augmented event, ϵ^-, which is defined in the obvious manner. Note that $(\gamma^\circ)^- = (\gamma^-)^\circ$.

The partial access functions $e \mathbin{@} \gamma$ and $t \mathbin{@} \gamma$ are defined for closed expressions e and closed terms t by the following equations:

$$\begin{array}{rcl} t \mathbin{@} \gamma \frown \mathtt{(}v\mathtt{)} & = & [t, v/f, a]e \mathbin{@} \gamma \\ (\textit{where } t = \mathtt{mfun}\, f\ \mathtt{(}a\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\, e\ \mathtt{end}) \end{array}$$

$$\begin{array}{rcl} e \mathbin{@} \bullet & = & e \\ \mathtt{let\ !}\, x\mathtt{:}\tau\ \mathtt{be}\, t\ \mathtt{in}\, e\ \mathtt{end} \mathbin{@} \gamma \frown \mathtt{!}v & = & [v/x]e \mathbin{@} \gamma \\ \mathtt{let}\, a_1\mathtt{:}\tau_1 \times a_2\mathtt{:}\tau_2\ \mathtt{be}\, t\ \mathtt{in}\, e\ \mathtt{end} \mathbin{@} \beta \frown \langle v_1, v_2\rangle & = & [v_1, v_2/a_1, a_2]e \mathbin{@} \beta \\ \mathtt{mcase}\, t\ \mathtt{of\ inl\ (}a_1\mathtt{:}\tau_1\mathtt{)} \Rightarrow e_1 \mid \mathtt{inr\ (}a_2\mathtt{:}\tau_2\mathtt{)} \Rightarrow e_2\ \mathtt{end} \mathbin{@} \beta \frown \mathtt{inl(}v\mathtt{)} & = & [v/a_1]e_1 \mathbin{@} \beta \\ \mathtt{mcase}\, t\ \mathtt{of\ inl\ (}a_1\mathtt{:}\tau_1\mathtt{)} \Rightarrow e_1 \mid \mathtt{inr\ (}a_2\mathtt{:}\tau_2\mathtt{)} \Rightarrow e_2\ \mathtt{end} \mathbin{@} \beta \frown \mathtt{inr(}v\mathtt{)} & = & [v/a_2]e_2 \mathbin{@} \beta \end{array}$$

Note that $(e \mathbin{@} \gamma)^- = e^- \mathbin{@} \gamma^-$, and similarly for values, v.

Augmented branches, and the associated access function, are needed for the proof of soundness. The proof maintains an augmented branch that enriches the current simple branch of the dynamic semantics. The additional information provided by augmented branches is required for the induction, but it does not affect any `return` statement it may determine.

Lemma 3
If $e \mathbin{@} \gamma = \mathtt{return(}t\mathtt{)}$, *then* $e \mathbin{@} \gamma^\circ = \mathtt{return(}t\mathtt{)}$.

A *function assignment*, Σ, is a finite mapping from locations to well-formed, closed, pure function values. A function assignment is *consistent with* a term, t, or expression, e, if and only if whenever $\mathtt{mfun}_l\, f\,\mathtt{(}a\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\, e\ \mathtt{end}$ occurs in either t or e, then $\Sigma(l) = \mathtt{mfun}\, f\ \mathtt{(}a\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\, e^-\ \mathtt{end}$. Note that if a term or expression is consistent with a function assignment, then no two function values with distinct underlying values may have the same label. A function assignment is consistent with a store, σ, if and only if whenever $\sigma(l)(\beta) = v$, then Σ is consistent with v.

A store, σ, *tracks* a function assignment, Σ, if and only if Σ is consistent with σ, $\mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma)$, and for every $l \in \mathrm{dom}(\sigma)$, if $\sigma(l)(\beta) = v$, then

1. $\Sigma(l) \mathbin{@} \beta^- = \texttt{return}(t^-)$,

2. $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$,

Thus if a branch is assigned a value by the memo table associated with a function, it can only do so if that branch determines a `return` statement whose value is the assigned value of that branch, relative to the non-memoizing semantics.

We are now in a position to prove the soundness of MFL.

Theorem 4

1. If $\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'$, Σ is consistent with t, σ tracks Σ, $\emptyset; \emptyset \vdash t : \tau$, then $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$ and there exists $\Sigma' \supseteq \Sigma$ such that Σ' is consistent with v and σ' tracks Σ'.

2. If $\sigma, l{:}\beta, e \Downarrow^{\mathsf{e}} v, \sigma'$, Σ is consistent with e, σ tracks Σ, $\gamma^\circ = \beta$, $\Sigma(l) \mathbin{@} \gamma^- = e^-$, and $\emptyset; \emptyset \vdash e : \tau$, then there exists $\Sigma' \supseteq \Sigma$ such that $e^- \Downarrow^{\mathsf{e}}_{\mathsf{p}} v^-$, Σ' is consistent with v, and σ' tracks Σ'.

Proof: The proof proceeds by simultaneous induction on the memoized evaluation relation. We consider here the five most important cases of the proof: function values, function terms, function application terms, and return expressions.

For function values $t = \texttt{mfun}_l\, f\,(a{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end}$, simply take $\Sigma' = \Sigma$ and note that $v = t$ and $\sigma' = \sigma$.

For function terms $t = \texttt{mfun}\, f\,(a{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end}$, note that $v = \texttt{mfun}_l\, f\,(a{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end}$ and $\sigma' = \sigma[l \mapsto \emptyset]$, where $l \notin \mathrm{dom}(\sigma)$. Let $\Sigma' = \Sigma[l \mapsto v^-]$, and note that since σ tracks Σ, and $\sigma(l) = \emptyset$, it follows that σ' tracks Σ'. Since Σ is consistent with t, it follows by construction that Σ' is consistent with v. Finally, since $v^- = t^-$, we have $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$, as required.

For application terms $t = t_1\, t_2$, we have by induction that ${t_1}^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} {v_1}^-$ and there exists $\Sigma_1 \supseteq \Sigma$ consistent with v_1 such that σ_1 tracks Σ_1. Since $v_1 = \texttt{mfun}_l\, f\,(a{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end}$, it follows from consistency that $\Sigma_1(l) = {v_1}^-$. Applying induction again, we obtain that ${t_2}^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} {v_2}^-$, and there exists $\Sigma_2 \supseteq \Sigma_1$ consistent with v_2 such that σ_2 tracks Σ_2. It follows that Σ_2 is consistent with $[v_1, v_2/f, a]e$. Let $\gamma = (v_2) \cdot \bullet$. Note that $\gamma^\circ = \bullet = \beta$ and we have

$$\begin{array}{rcl}
\Sigma_2(l) \mathbin{@} \gamma^- & = & {v_1}^- \mathbin{@} \gamma^- \\
& = & (v_1 \mathbin{@} \gamma)^- \\
& = & ([v_1, v_2/f, a]e)^- \\
& = & [{v_1}^-, {v_2}^-/f, a]e^-.
\end{array}$$

Therefore, by induction, $[{v_1}^-, {v_2}^-/f, a]e^- \Downarrow^{\mathsf{e}}_{\mathsf{p}} {v'}^-$, and there exists $\Sigma' \supseteq \Sigma_2$ consistent with v' such that σ' tracks Σ'. It follows that $(t_1\, t_2)^- = {t_1}^-\, {t_2}^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} {v'}^-$, as required.

For return statements, we have two cases to consider, according to whether the current branch is in the domain of the current memo table. Suppose that $\sigma, l{:}\beta, \texttt{return}(t) \Downarrow^{\mathsf{e}} v, \sigma'$ with Σ consistent with $\texttt{return}(t)$, σ tracking Σ, $\gamma^\circ = \beta$, $\Sigma(l) \mathbin{@} \gamma^- = (\texttt{return}(t))^- = \texttt{return}(t^-)$, and $\emptyset; \emptyset \vdash \texttt{return}(t) : \tau$. Note that by Lemma 3, $(\Sigma(l) \mathbin{@} \beta)^- = \Sigma(l) \mathbin{@} \beta^- = \texttt{return}(t^-)$.

For the first case, suppose that $\sigma(l)(\beta) = v$. Since σ tracks Σ and $l \in \mathrm{dom}(\sigma)$, we have $\Sigma(l) = \mathtt{mfun}\, f\ (a\!:\!\tau_1)\!:\!\tau_2\ \mathtt{is}\, e^-\ \mathtt{end}$ with $e^-\ @\ \beta^- = \mathtt{return}(t^-)$, and $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$. Note that $\sigma' = \sigma$, so taking $\Sigma' = \Sigma$ completes the proof.

For the second case, suppose that $\sigma(l)(\beta)$ is undefined. By induction $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$ and there exists $\Sigma' \supseteq \Sigma$ consistent with v such that σ' tracks Σ'. Let $\theta' = \sigma'(l)$, and note $\theta'(\beta) \uparrow$, by Lemma 2. Let $\theta'' = \theta'[\beta \mapsto v]$, and $\sigma'' = \sigma'[l \leftarrow \theta'']$. Let $\Sigma'' = \Sigma'$; we are to show that Σ'' is consistent with v, and σ'' tracks Σ''. By the choice of Σ'' it is enough to show that $\Sigma'(l)\ @\ \beta^- = \mathtt{return}(t^-)$, which we noted above.

■

The soundness theorem (Theorem 5) for MFL states that evaluation of a program (a closed term) with memoization yields the same outcome as evaluation without memoization. The theorem follows from Theorem 4.

Theorem 5 (Soundness)
If $\emptyset, t \Downarrow^{\mathsf{t}} v, \sigma$, *where* $\emptyset; \emptyset \vdash t : \tau$, *then* $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$.

Type safety follows from the soundness theorem, since type safety holds for the non-memoized semantics. In particular, if a term or expression had a non-canonical value in the memoized semantics, then the same term or expression would have a non-canonical value in the non-memoized semantics, contradicting safety for the non-memoized semantics.

4.5 Performance

We show that memoization slows down an MFL program by a constant factor (expected) with respect to a standard, non-memoizing semantics even when no results are re-used. The result relies on representing a branch as a sequence of integers and using this sequence to key memo tables, which are implemented as hash tables. To represent branches as integer sequences we use the property of MFL that the underlying type η of a bang type, $!\,\eta$, is an indexable type. Since any value of an indexable type has an integer index, we can represent a branch of dependencies as sequence of integers corresponding to the indices of `let!`'ed values, and zero or one for `inl` and `inr`.

Consider a non-memoizing semantics, where the `return` rule always evaluates its body and neither looks up nor updates memo tables (stores). Consider an MFL program and let T denote the time it takes (the number of evaluation steps) to evaluate the program with respect to this non-memoizing semantics. Let T' denote the time it takes to evaluate the same program with respect to the memoizing semantics. In the worst case, no results are re-used, thus the difference between T and T' is due to memo-table lookups and updates done by the memoizing semantics. To bound the time for these, consider a memo table lookup or update with a branch β and let $|\beta|$ be the length of the branch. Since a branch is a sequence of integers, a lookup or update can be performed in expected $O(|\beta|)$ time using nested hash tables to represent memo tables. Now note that the non-memoizing semantics takes $|\beta|$ time to build the branch thus, the cost of a lookup or update can be charged to the evaluations that build the branch β, *i.e.*, evaluations of `let!` and `mcase`. Furthermore, each

```
signature MEMO =
sig
  (* Expressions *)
  type 'a expr
  val return:  (unit -> 'a) -> 'a expr

  (* Resources *)
  type 'a res
  val expose:  'a res -> 'a

  (* Bangs *)
  type 'a bang
  val bang :  ('a -> int) -> 'a -> 'a bang
  val letBang:  ('a bang) -> ('a -> 'b expr) -> 'b expr

  (* Products *)
  type ('a,'b) prod
  val pair:  'a -> 'b -> ('a,'b) prod
  val letx:  ('a,'b) prod -> (('a res * 'b res) -> 'c expr) -> 'c expr
  val split:  ('a,'b) prod -> (('a * 'b) -> 'c) -> 'c

  (* Sums *)
  type ('a,'b) sum
  val inl:  'a -> ('a,'b) sum
  val inr:  'b -> ('a,'b) sum
  val mcase:  ('a,'b) sum -> ('a res -> 'c expr) -> ('b res -> 'c expr) -> 'c expr
  val choose:  ('a,'b) sum -> ('a -> 'c) -> ('b -> 'c) -> 'c

  (* Memoized arrow *)
  type ('a,'b) marrow
  val mfun:  ('a res -> 'b expr) -> ('a,'b) marrow
  val mfun_rec:  (('a, 'b) marrow -> 'a res -> 'b expr) -> ('a,'b) marrow
  val mapply:  ('a,'b) marrow -> 'a -> 'b
end

signature BOX = sig
  type 'a box
  val init:  unit->unit
  val box:  'a->'a box
  val unbox:  'a box->'a
  val getKey:  'a box->int
end
```

Figure 10: The signatures for the memo library and boxes.

evaluation of `let!` and `mcase` can be charged by exactly one `return`. Thus, we conclude that $T' = O(T)$ in the expected case.

5 Implementation

We describe an implementation of our framework as a Standard ML library. The aspects of the MFL language that relies on the syntactic distinction between resources and variables cannot be enforced statically in Standard ML. Therefore, we use a separate type for resources and employ run-time checks to detect violations of correct usage.

```
functor BuildMemo (structure Box:BOX structure Memopad:MEMOPAD):MEMO =
struct
  type 'a expr = int list * (unit -> 'a)
  fun return f = (nil,f)

  type 'a res = 'a
  fun res v = v
  fun expose r = r

  type 'a bang = 'a * ('a -> int)
  fun bang h t = (t,h)
  fun letBang b f =
    let val (v,h) = b
        val (branch,susp) = f v
    in
      ((h v)::branch, susp)
    end

  type ('a,'b) prod = 'a * 'b
  fun pair x y = (x,y)
  fun split p f = f p
  fun letx (p as (x1,x2)) f = f (res x1, res x2)

  datatype ('a,'b) sum = INL of 'a | INR of 'b
  fun inl v = INL(v)
  fun inr v = INR(v)
  fun mcase s f g =
    let val (lr,(branch,susp)) = case s of
                                   INL v => (0,f (res v))
                                 | INR v => (1,g (res v))
    in
      (lr::branch,susp)
    end
  fun choose s f g = case s of INL v => f v | INR v => g v

  type ('a,'b) marrow = 'a -> 'b
  fun mfun_rec f =
    let val mpad = Memopad.empty ()
        fun mf rf x =
          let val (branch,susp) = f rf (res x)
              val result = case Memopad.extend mpad branch of
                              (NONE,SOME mpad') =>        (* Not found *)
                                let val v = susp ()
                                    val _ = Memopad.add v mpad'
                                in v end
                            | (SOME v,NONE) => v          (* Found *)
          in result end
        fun mf' x = mf mf' x
    in
      mf'
    end

  fun mfun f = ...  (* Similar to mfun_rec *)

  fun mapply f v = f v
end
```

Figure 11: The implementation of the memoization library.

```
structure Examples =
struct
  type 'a box = 'a Box.box

  fun iBang v = bang (fn i => i) v
  fun bBang b = bang (fn b => Box.key b) b

  (** Boxed lists **)
  datatype 'a blist' = NIL | CONS of ('a * (('a blist') box))
  type 'a blist = ('a blist') box

  (** Hash-cons **)
  fun hCons' (x') =
    letx (expose x') (fn (h',t') =>
    letBang (expose h') (fn h => letBang (expose t') (fn t =>
    return (fn()=> box (CONS(h,t))))))
  val hCons = mfun hCons'

  (** Fibonacci **)
  fun mfib' f (n') =
    letBang (expose n') (fn n =>
    return (fn()=>if n < 2 then n else (mapply f (iBang(n-1))) + (mapply f (iBang(n-2)))
  fun mfib n = mapply (mfun_rec mfib') n

  (** Knapsack **)
  fun mks' mks (arg) =
    letx (expose arg) (fn (c',l') =>
    letBang (expose c') (fn c =>
    letBang (expose l') (fn l => return (fn () =>
      case (unbox l) of
        NIL => 0
      | CONS((w,v),t) => if (c < w) then mapply mks (pair (iBang c) (bBang t))
                         else let val v1 = mapply mks (pair (iBang c) (bBang t))
                                  val v2 = v + mapply mks (pair (iBang (c-w)) (bBang t))
                              in if (v1 > v2) then v1 else v2 end))))
  val mks x = mfun_rec mks'

  (** Quicksort **)
  fun mqs () =
    let val empty = box NIL
        val hCons = mfun hCons'
        fun fil f l =
          case (unbox l) of
            NIL => empty
          | CONS(h,t) => if (f h) then (mapply hCons (pair (iBang h) (bBang (fil f t))))
                         else (fil f t)
        fun qs' qs (l') = letBang (expose l') (fn l => return (fn () =>
            case (unbox l) of
              NIL => nil
            | CONS(h,t) => let val ll = fil (fn x=>x<h) t
                               val gg = fil (fn x=>x>=h) t
                               val sll = mapply qs (bBang ll)
                               val sgg = mapply qs (bBang gg)
                           in sll@(h::sgg) end))
    in mfun_rec qs' end
end
```

Figure 12: Examples from Section 3 in the SML library.

The interface for the library (shown in Figure 10) provides types for expressions, resources, bangs, products, sums, memoized functions along with their introduction and elimination forms. All expressions have type `'a expr`, which is a monad with `return` as the inclusion and various forms of "bind" induced by the elimination forms `letBang,` `letx`, and `mcase`. A resource has type `'a res` and `expose` is its elimination form. Resources are only created by the library, thus no introduction form for resources is available to the user. The introduction and elimination form for bang types are `bang` and `letBang`. The introduction and elimination form for product types are `pair`, and `letx` and `split` respectively. The `letx` is a form of "bind" for the monad `expr`; `split` is the elimination form for the term context. The treatment of sums is similar to product types. The introduction forms are `inl` and `inr`, and the elimination forms are `mcase` and `choose`; `mcase` is a form of bind for the `expr` monad and `choose` is the elimination for the term context.

Memoized functions are introduced by `mfun` and `mfun_rec`; `mfun` takes a function of type `'a res -> 'b expr` and returns the memoized function of type `('a,'b) marrow`; `mfun_rec` is similar to `mfun` but it also takes as a parameter its memoized version. Note that the result type does not contain the "effect" `expr`—we encapsulate memoization effects, which are benign, within the function. The elimination form for the `marrow` is the memoized apply function `mapply`.

Figure 11 shows an implementation of the library without the run-time checks for correct usage. To incorporate the run-time checks, one needs a more sophisticated definition of resources in order to detect when a resource is exposed out of its context (*i.e.*, function instance). In addition, the interface must be updated so that the first parameter of `letBang`, `letx`, and `mcase`, occurs in suspended form. This allows us to update the state consisting of certain flags before forcing a term.

The implementation extends the operational semantics of the MFL language (Section 4.3) with boxes. The `bang` primitive takes a value and an injective function, called the index function, that maps the value to an integer, called the index. The index of a value is used to key memo tables. The restriction that the indices be unique, enables us to implement memo tables as a nested hash tables, which support update and lookup operations in expected constant time. The primitive `letBang` takes a value `b` of bang type and a body. It applies the body to the underlying value of `b`, and extends the branch with the index of `b`. The function `letx` takes a pair `p` and a body. It binds the parts of the pair to two resources and and applies the body to the resources; as with the operational semantics, `letx` does not extend the branch. The function `mcase` takes value `s` of sum type and a body. It branches on the outer form of `s` and binds its inner value to a resource. It then applies the body to the resource and extends the branch with `0` or `1` depending on the outer form of `s`. The elimination forms of sums and products for the term context, `split` and `choose` are standard.

The `return` primitive finalizes the branch and returns its body as a suspension. The branch is used by `mfun_rec` or `mfun`, to key the memo table; if the result is found in the memo table, then the suspension is disregarded and the result is re-used; otherwise the suspension is forces and the result is stored in the memo table keyed by the branch. The `mfun_rec` primitive takes a recursive function `f` as a parameter and "memoizes" `f` by associating it with a memo pad. A subtle issue is that `f` must calls its memoized version recursively. Therefore `f` must take its memoized version as a parameter. Note also that the memoized function internally converts its parameter to a resource before applying `f` to it.

The interface of the library provides no introduction form for resources. Indeed, all resources are created by the library inside the `letx`, `mcase`, `mfun_rec`, and `mfun`. The function `expose` is the elimination form for resources. If, for example, one would like to apply `letBang` to a resource, then he must first `expose` the resource, which "exposes" the underlying value.

Figure 12 show the examples from Section 3 written in the SML library. Note that the memoized Fibonacci function `mfib` creates a memo table every time it is called. When `mfib` finishes, this table can be garbage collected (the same applies to `mks`). For Quicksort, we provide a function `mqs` that returns an instance of memoized Quicksort when applied. Each such instance has its own memo table. Note also that `mqs` creates a local instance of the hash-cons function so that each instance of memoized Quicksort has its own memo table for hash-consing.

In the examples, we do not use the sum types provided by the library to represent boxed lists, because we do not need to. In general, one will use the provided sum types instead of their ML counterparts (for example if an `mcase` is needed). The examples in Figure 12 can be implemented using the following definition of boxed lists.

```
datatype 'a boxlist' =
         ROLL of (unit, (('a, 'a boxlist' box) prod)) sum
type 'a boxlist = ('a boxlist') box
```

Changing the code in Figure 12 to work with this definition of boxed lists requires several straightforward modifications.

6 Discussion

Space and Cache Management. Our framework associates a separate memo table with each memoized function. This allows the programmer to control the life-span of memo tables by conventional scoping. This somewhat coarse degree of control is sufficient in certain applications such as in dynamic programming, but finer level of control may be desirable for applications where result re-use is less regular. Such an application can benefit from specifying a caching scheme for individual memo tables so as to determine the size of the memo table and the replacement policy. We discuss how the framework can be extended to associate a cache scheme with each memo table and maintain the memo table accordingly.

The caching scheme should be specified in the form of a parameter to the `mfun` construct. When evaluated, this construct will bind the caching scheme to the memo table and the memo table will be maintained accordingly. Changes to the operational semantics to accommodate this extension is small. The store σ will now map a label to a pair consisting of a memo table and its caching scheme. The handling of the `return` will be changed so that the stores do not merely expand but are updated according to the caching scheme before adding a new entry. The following shows the updated return rule. Here $\mathcal{S}$ denotes a caching scheme and θ denotes a memo table. The `update` function denotes a function that updates the memo table to accommodate a new entry by possibly purging an existing entry. The programmer must ensure that the caching scheme does not violate the integrity of the memo table by tampering with stored values.

$$\frac{\sigma(l) = (\theta, \mathcal{S}) \quad \theta(\beta) = v}{\sigma, l{:}\beta, \texttt{return}(t) \Downarrow^{\mathsf{e}} v, \sigma} \quad \text{(Found)}$$

$$\frac{\begin{array}{c}\sigma(l) = (\theta, \mathcal{S}) \quad \theta(\beta) \uparrow \\ \sigma, t \Downarrow^{\mathsf{t}} v, \sigma' \\ \sigma'(l) = (\theta', \mathcal{S}) \quad \theta'' = \texttt{update}(\theta', \mathcal{S}, (\beta, v))\end{array}}{\sigma, l{:}\beta, \texttt{return}(t) \Downarrow^{\mathsf{e}} v, \sigma'[l \leftarrow \theta'']} \quad \text{(Not Found)}$$

For example, we can specify that the memo table for the Fibonacci function, shown in Figure 1, can contain at most two entries and be managed using the least-recently-used replacement policy. This is sufficient to ensure that the memoized Fibonacci runs in linear time. This extension can also be incorporated into the type system described in Section 4. This would require that we associate types with memo stores and also require that we develop a type system for "safe" `update` functions if we are to enforce that the caching schemes are safe.

Local *vs.* Non-local Dependences. Our dependence tracking mechanism only captures "local" dependences between the input and the result of a function. A local dependence of a function `f` is one that is created inside the static scope of `f`. A non-local dependence of `f` is created when `f` passes its input to some other function `g`, which examines `f`'s input indirectly. In previous work, Abadi *et. al.* [1] and Heydon *et. al.* [13] showed a program analysis technique for tracking non-local dependences by propagating dependences of a function to its caller. They do not, however, make clear the performance implications of their technique.

Our framework can be extended to track non-local dependences by introducing an application form for memoized functions in the expression context. This extension would, for example, allow for dependences of non-constant length. We chose not to support non-local dependences because it is not clear if its utility exceeds its performance effects.

Memoization and Adaptivity. The work we present in this paper was motivated by our previous work on adaptive computation [2]. We briefly discuss the relationship between memoization and adaptivity and how they can be combined to obtain efficient dynamic or incremental algorithms.

An adaptive computation maintains a dynamic dependence graph representing data and control dependences. When the input is modified, a change propagation algorithm updates the output and the dependence graph. The adaptivity mechanism handles "deep" changes efficiently. We say that a change is *deep* if it affects calls that occur at leaves of the call tree for the computation. In contrast, a change is *shallow* if it affects by a calls that occur at the roots of the call tree.

As an example consider the Quicksort algorithm that picks the first key of its input as pivot. Inserting a new key at the end of the input list is a deep change because this change will affect the last recursive calls of some filter functions and will become pivot only at the end of some sequence of recursive calls to Quicksort. In contrast, inserting a new key at the beginning of the list is a shallow change for Quicksort, because the new key will be selected as a pivot immediately by the first call to Quicksort. The adaptivity mechanism based on dynamic dependence graphs handles an insertion at the end of the input, a deep change, in expected $O(\log n)$ time [2], whereas the insertion at the beginning of the list, a shallow change, will cause a complete rerun, which takes $O(n \log n)$ time. Using memoization, however, an insertion at the beginning of the list can be handled in $O(n)$ time as showed in Section 3.

Any change can be thought of a combination of shallow and deep changes. Since memoization and adaptivity complement each other in their handling of deep and shallow changes, we would expect that a combination of these two techniques would handle general changes efficiently. For example, in Quicksort, we expect that an insertion in a random position in the list would be handled in expected $O(\log n)$ time by a combination of these two techniques.

7 Conclusion

We presented a framework for selective memoization under programmer control. The framework makes explicit the performance effects of memoization and yields programs whose running times can be analyzed using standard techniques. A key aspect of the framework is that it can capture both control and data dependences between input and the result of a memoized function. The main contributions of the paper are the particular set of primitives we suggest and the semantics along with the proofs that it is sound. We gave a simple implementation of the framework in the Standard ML language. We expect that this framework can be implemented in any purely-functional language.

References

[1] M. Abadi, B. W. Lampson, and J.-J. Levy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] J. Allen. *Anatomy of LISP*. McGraw Hill, 1978.

[5] A. W. Appel and M. J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

[6] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[7] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, Dec. 1980.

[8] N. H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[10] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

[11] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the ACM '90 Conference on LISP and Functional Programming*, pages 307–322, June 1990.

[12] E. Goto and Y. Kanada. Hashing lemmas on time complexities with applications to formula manipulation. In *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 154–158, 1976.

[13] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the 2000 ACM SIGPLAN Conference on PLDI*, pages 311–320, May 2000.

[14] J. Hilden. Elimination of recursive calls using a small table of randomly selected function values. *BIT*, 16(1):60–73, 1976.

[15] R. Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the 1992 ACM SIGPLAN Conference on PLDI*, pages 261–272, June 1992.

[16] R. J. M. Hughes. Lazy memo-functions. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, 1985.

[17] S. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[18] Y. A. Liu, S. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.

[19] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *European Symposium on Programming*, pages 288–305, 1999.

[20] J. McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[21] D. Michie. 'memo' functions and machine learning. *Nature*, 218:19–22, 1968.

[22] J. Mostov and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 165–172, Aug. 1985.

[23] T. Murphy, R. Harper, and K. Crary. The wizard of TILT: Efficient(?), convenient and abstract type representations. Technical Report CMU-CS-02-120, School of Computer Science, Carnegie Mellon University, Mar. 2002.

[24] P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, pages 91–98, 1991.

[25] M. Pennings. *Generating Incremental Attribute Evaluators.* PhD thesis, University of Utrecht, Nov. 1994.

[26] M. Pennings, S. D. Swierstra, and H. Vogt. Using cached functions and constructors for incremental attribute evaluation. In *Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, pages 130–144, 1992.

[27] F. Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166. Computer Society Press, 1995.

[28] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[29] J. Polakow and F. Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 130–144. Springer-Verlag LNCS 1581, 1999.

[30] W. Pugh. *Incremental computation via function caching.* PhD thesis, Department of Computer Science, Cornell University, August 1988.

[31] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.

[32] J. M. Spitzen and K. N. Levitt. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, 1978.

[33] R. S. Sundaresh and P. Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on POPL*, pages 1–13, Jan. 1991.

[34] Y. Zhang and Y. A. Liu. Automating derivation of incremental programs. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, page 350. ACM Press, 1998.