Selective memoization[*]

Umut A. Acar Guy E. Blelloch Robert Harper
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

July 22, 2002

Abstract

Although memoization is a powerful technique and can dramatically improve performance, it is often difficult to apply effectively. This is because significant performance improvements require the programmer to set up rather complex machinery specific to the application. Such machinery requires programmer control over issues such as equality tests, identification of precise dependencies, and space management.

In this paper we present a general framework for memoization that gives the programmer control over the specifics of memoization for maximum performance. The key property of the framework is that it is efficient: memoization preserves the asymptotic performance of the program. In fact the only overhead incurred is that of maintaining memo tables.

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

1 Introduction

Memoization is a fundamental and powerful technique for result re-use. It dates back a half century [7, 26, 27] and has since been used extensively in many areas such as dynamic programming [3, 9, 10, 23], incremental computations [11, 37, 13, 41, 20, 1, 42, 24, 17, 18, 2] and many others [8, 28, 21, 16, 25, 30, 31, 24]. In fact, lazy evaluation provides a limited form of memoization[22]. But it is not well suited for programs where equal arguments to functions are computed by different paths; as we will see later, this is the case for many examples such as Fibonacci or most dynamic programming algorithms.

Although memoization can dramatically improve performance and requires 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 effects of memoization. Many subtleties of memoization, including the cost of equality checking and the caching and replacement policy for memo tables, can make the difference between an exponential and a linear running time.

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

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 dependencies between the input and the output of memoized code; and (3) space management via a caching and purging scheme for memo tables. Control over equality tests is critical, because this is how re-usable results are identified. Identification of precise dependencies is important to maximize result reuse by performing "partial equality" tests. Being able to control the caching and purging scheme is critical, because otherwise the user will not know whether or when results are re-used.

Our proposal takes the form of a small language, called MFL, that makes clear the performance implications of memoization without compromising convenience or expressiveness. MFL is a purely functional language enriched with support for user-controlled, selective memoization. The operational semantics of MFL specifies the performance of programs accurately enough to determine (expected) asymptotic time bounds.[1] In particular the operational semantics makes explicit these crucial features: (1) the equality conditions for matching, and the conditions under which function calls will match in a memo table; (2) provision for partial matching of arguments to memoized functions; and (3) scoping of memo tables so that their space usage is placed under programmer control. We give several examples of the use of the language and we prove its correctness—*i.e.,* that the semantics are preserved with respect to a non-memoized version. We also briefly describe an ML library that implements MFL, albeit with run-time checks in place of certain static constraints that cannot be enforced by the ML type system. A key property of the implementation is that it requires *no* program analysis.

This work was motivated by our previous work on adaptive computation [2]. We are developing techniques for implementing dynamic, or incremental, algorithms that rely on dynamic dependency information and on memoization. In future work we intend to combine the adaptive programming mechanisms of AFL with the memoization mechanisms described here.

In the next section we describe the three issues – equality, identification of dependencies, and space management – in more detail and review the previous work. In Section 3 we introduce our framework via some examples and demonstrate how the three issues can be addressed. 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.

2 Background

A typical memoization framework maintains a cache of previously computed results. This cache is consulted before each function call. If the result is found in the cache then it is re-used; otherwise the call is performed and the result is added to the cache indexed by the function and its parameters. Although simple, the typical memoization framework requires that the programmer addresses three issues carefully: equality, caching with precise dependencies, and space management.

Equality. Any memoization scheme relies on equality tests to determine whether a call has previously been computed. The performance of a memoization scheme is therefore heav-

[1]Expected, rather than worst-case, performance is required because of our reliance on hashing, which has constant-time expected access.

ily dependent on the cost of equality checking. The broadest distinction in definitions can be made between structural equality and location/tag equality—roughly speaking `EQUAL` *vs.*. `EQ` in Lisp [14]. Using structural equality two "copies" of the same value are regarded as equal, whereas in location/tag equality they are not. On the other hand, structural equality can take time linear in the size of the structures, as compared to the constant time for location equality. Thus, if the function being memoized takes linear or sub-linear time, then the cost of the equality test will negate the advantage of memoizing. Even worse, if a function only examines part of its argument, structural equality testing could decrease its asymptotic performance. On the other hand, tag equality can miss opportunities for a match, which can lead to an exponential slowdown. For example, consider an implementation of Fibonacci in which the argument n is represented in multi-precision arithmetic, and equality is based on a tag (location) given when the argument is created. In this case the calls with $((n-1)-2)$ and $((n-2)-1)$ may generate different copies of $(n-3)$, preventing a match in the memo table.

One solution to this problem is to ensure that there is only one copy of each and every value, via a technique known as "hash consing" [15, 4, 40]. Although hash-consing is a powerful technique backed by solid theory, the reality is rather different. In fact, several researchers have argued that hash-consing is too expensive for practical purposes [38, 39, 5, 29]. The main reason for this is that many programs construct and destruct objects at immense rates, and hash-consing slows this process down by a significant constant factor. The situation is worsened by large memory demands and the interaction of hash-consing with garbage collection. In fact, even when built into a garbage collector and only used when moving data from the new to old generation, which would be ineffective in general, it has a significant cost [5]. As an alternative to hash consing, Pugh proposed lazy structure sharing[38]. In lazy structure sharing whenever two equal values are compared, they are made to point to the same copy via side effects to speed up subsequent comparisons. As Pugh points out, the disadvantage of this approach is that the performance depends on the order of 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. For example, the same partial application of the same curried function would not trigger a match.

Precise Dependencies. The second problem is to ensure that the results are cached with respect to their true dependencies. The issue arises in two contexts: (1) when the function examines only parts of its inputs, and (2) when it examines a user-determined approximation of its input. In the first case the unexamined parts of the input should be disregarded. In the second, it may be better to use approximations, rather than the inputs themselves, so as to increase the likelihood of re-use. As an example, consider the code

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

The result depends only on part of the input, *i.e.*, either `(x,y)` or `(x,z)`. Furthermore, the result does not depend on the exact value of `x` but an approximation of `x` – whether or not it is positive. 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 [33, 32, 25, 1, 16, 18]. Abadi, Lampson, Lévy [1], and Heydon, Levin, Yu [18] have suggested program analysis methods for tracking dependences for this purpose. Although their methods are likely effective in catching potential matches, their method have three main disadvantages. First, their scheme can efficiently support only a predetermined set of approximations [18]. Second, it does not give the user control over the granularity of the program analysis. Third, it does not provide user with a strong performance guarantee. For example, Abadi, *et. al.*'s method requires labeling all the elements of a data structure[2] before evaluating the function and keeping track of what labels the output depend on. They assume the labels are used at the finest-grain and point out that other coarser grained labeling schemes might be used. The labeling policy, however, would have to be described in a "meta-language". The finest-grain labeling policy can change the asymptotic behavior of a program by labeling parts of a data structure that is otherwise unexamined – for example, a function that returns the head of a list will now take linear time. Similarly, Heydon, *et. al.*'s system may require the same expression be evaluated multiple times due to the their dependency propagation mechanism, making it difficult for the user to analyze the running time.

Space management. Another problem with memoization is its space requirement. In general memo tables can become very large very quickly, limiting their utility. Many researchers have suggested that to alleviate space problems old entries should be flushed [19, 39]. But it is not clear what policy to use to flush old entries. One widely used approach is to purge the least recently used entry. Other, more sophisticated, policies have also been suggested [39]. In general the replacement policy must be application-specific [39]; for any fixed policy, there are programs whose performance is made worse by that choice. Moreover, it is difficult for the programmer to know when a given entry would be replaced in a given situation, making it hard to assess performance.

3 Overview

We present an overview of our framework through some examples and demonstrate how the framework addresses the issues described in Section 2. The main point is this: Previous work tried to support memoization efficiently without help from the programmer. In the case of equality tests, this requires one to build a global hash-consing mechanism, because it is undecidable to determine what values should be hash-consed for optimal performance. In the case of space management, this requires one to adapt a caching scheme that is based on heuristics such as the least-recently-used policy, or profiling information, which are difficult to analyze and do not work well for certain applications. In the case of dependency identification, this requires costly program analysis techniques that can adversely effect the asymptotic performance of the program. Our proposal simplifies the support system significantly by enabling the programmer to control the specifics of memoization.

The language and its type system ensures that the programs are safe (do not incur run-time type errors) and sound (do not change their behavior). The three principle ideas behind the language are: (1) present the programmer with one simple notion of equality and enable him to choose which objects are equal; (2) allow the programmer to choose on what aspects of the argument the result of a function depends; (3) associate a memo table with each memoized function definition so that the programmer may dispose of memo tables by scoping. In the rest of this section we will show examples demonstrating these ideas.

[2]Since they use the pure lambda-calculus it is actually all the lambda's in an expression.

We will use an ML-like syntax with some simple extensions. The core of this language is formalized in Section 4 — the presentation in this section is rather informal.

Non-memoized	Memoized
fun fib (n:int)=	mfun mfib ($\underline{n}$:!int) =
	let !n' = $\underline{n}$ in return (
if (n < 2) then n	if (n' < 2) then n'
else fib(n-1) + fib(n-2)	else mfib(!(n'-1)) + mfib (!(n'-2)))
fun sum (n:int*int)=	mfun msum ($\underline{n}$:(!int*!int))=
let (n1,n2) = n in	letX ($\underline{n1}$,$\underline{n2}$) = $\underline{n}$ in
case (n1 > 0) of	mcase ($\underline{n1}$> 0) of
true => 0	true => return (0)
| false =>	| false => let !n1' = $\underline{n1}$ and !n2' = $\underline{n2}$ in
(n1+n2)	return (n1'+n2')
end	end
	end

Figure 1: The memoized Fibonacci, and expressing partial dependencies.

Precise Dependencies. The framework enables the programmer to choose the particular dependencies between the input of a function and its result, and ensures that all such dependencies are correctly revealed. The main technique is to introduce a distinction between *resources* and *variables*. Resources are restricted variables that may not occur in the result of a memoized function. Parameters to memoized functions are resources. Taken together, this implies that the programmer must explicitly "touch" parameters before using them to compute the result of the function.

There are several ways to touch a resource, depending on its type. If the code depends on the full value of the resource, it should be assigned the modal type $!\,\tau$. The value of a resource of this type is accessed using the `let!` construct, which binds it to an ordinary, unrestricted variable. If, on the other hand, the code depends only on some aspect of the value of a resource, its value may be explored incrementally. This is possible when the resource is either of product or sum type. In the case of a (binary) product type we may split the resource into two resources, one for each component of the pair, using the `letX` primitive. In the case of a (binary) sum type we may case analyze the outermost form of the value, branching according to its summand, using the `mcase` primitive.

The dependencies are recorded during the process of exploring the structure of a resource. The `let!` records the full value, the `mcase` records the branch taken (or the kind of the sum), and `letX` records nothing. This dependency information is used to key the memo table when a `return` primitive is encountered. If this sequence of dependencies have been taken before, the stored value is returned, otherwise the body of the `return` is evaluated and the result is stored with respect to the dependencies. The type system ensures that all dependencies are made explicit by precluding the use of resources within the `return`'s body.

As an example, consider the Fibonacci function `mfib`, whose code is given in Figure 1 (underlined symbols are resources). The function touches its parameter and computes the nth Fibonacci number by two recursive calls. Since the full value of the parameter n is used, it has the type `! int`. As another example, the function `msum` shows how partial dependencies can be expressed. We would like to return 0 when `n1` is positive, and the sum of `n1` and `n2` otherwise. Thus, the result will depend on the sign of `n1` in one case, and on

Non-memoized	Memoized

	`datatype α blist = NIL`
	`                  | CONS of (α * ((α blist) box))`
	`type α boxedlist = (α blist) box`
	`mfun hash-cons (h:!(α box), t:!(α boxedlist)) =`
	`  let !h' = h  in`
	`    let !t' = t  in`
	`      return (box (CONS(h',t')))`
	`  end end`

`datatype α tree = EMPTY`
`                | NODE of α * (α tree * α tree)`
`fun compare(a:!int,b:int) = ...`
`fun search (t:int tree, k:int)`
`  case t of`
`    EMPTY => nil`
`  | NODE(key,(left,right)) =>`
`      case compare (k,key) of`
`        EQUAL => [key]`
`      | LESS => let r = search (left,k) in`
`          key::r`
`        end`
`      | GREATER => let r = search (right,k) in`
`          key::r`
`        end`
	`datatype α tree = EMPTY`
	`                | NODE of α * (α tree * α tree)`
	`fun compare(a:!int,b:!int) =...`
	`mfun msearch (t:(!int) tree, k:  !int)`
	`  mcase t  of`
	`    EMPTY => return (!(box NIL))`
	`  | NODE(key,(left,right)) => let !key' = key  in`
	`      case compare (k,key) of`
	`        EQUAL => return (!box [key'])`
	`      | LESS => let !r = msearch (left,k) in`
	`          return (!(box (key'::r)))`
	`        end`
	`      | GREATER => let !r = msearch (right,k) in`
	`          return (!(box (key'::r)))`
	`        end end`

Figure 2: Hash-consing is a special form of memoization.

both `n1` and `n2` in the other. Indeed, the function `msum` touches `n1` and `n2` only when `n1` is positive. There are many other examples where support for partial dependencies is critical. For example, in scientific computing, if the result of a particular operations not introduce a big error term, then an approximation of a multi-precision floating point number can be used instead of the number itself. In this case, it is preferable to depend on the approximation than the number itself; since the approximation is coarser, this can increase result re-reuse. This sort of examples are easily expressed in our framework.

Equality. To manage equality testing we extend the language with the type τ `box` whose values are "boxed", or "labeled", values of type τ. A unique label is associated with each box when it is created, and this label is used for testing equality. Equality is only defined on boxes (*i.e.*, their labels) and primitive types. These labels and primitive types can be matched in a hash table in expected constant time. Other forms of equality such as structural equality can easily be built by the programmer. For example, the function `hash-cons` in Figure 2 shows how to implement hash consing in our framework. The code takes a head item and a boxed list, where each tail of the list is boxed. It then touches both the head and the tail and return the list of the new item and the list. Since the function is memoized, if the function is ever called with two values that are already `hash-cons`'d, then the same list will be returned.

As mentioned in Section 2, the main problem with hash consing is that, when not controlled, it should be applied to the whole program to be effective. On the other hand, such "global" hash consing in unnecessary in many situations. For example, a function is more likely to return the same result then some other function. In this case, there is no need to force that the result from these two functions go through the same hash-consing scheme. Furthermore, even though two function may be likely to return the same result, the results may be passed to different set of functions and thus there would not be any use in forcing the return results to match. As an example, consider performing a search on a binary search tree. The function `search` shows an implementation for this where the path

Non-memoized

```
fun ks (c:int, l:(int*real) list) =

  case c <= 0 of
    true => 0
  | false =>
      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
```

Memoized

```
mfun mks (c:!int,l:  !((int*real) boxedlist)) =
  let !c' = c  in
    case c' <= 0 of
      true => return (0)
    | false => let !l' = l in return (
        case (unbox l') of
          nil => 0
        | (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 3: Memo-tables for memoized Knapsack can be discarded at completion.

taken during the search is returned in the form of a list. The function `msearch` shows a memoized version of `search`. The function `msearch` performs hash-consing only for its own result. In many application it is common that a search is repeated after a small modification to the tree. If the modification did not change the path taken, it is advantageous to return the same list. For example, if the result is passed to a function that finds the length of the list then the result will be found in the memo the second time. Furthermore, using a scoping technique described below the memo table for `msearch` could be discarded when the search-modify cycle is over.

Space management. In our framework every memoized function is assigned its own table. Thus, when the function goes out of scope its memo table becomes garbage and its space can be reclaimed. This allows the programmer to manage space through conventional scoping mechanisms. For example, many dynamic programming algorithms re-use results between recursive calls of the same function. But it is unlikely that results are re-used across independent calls of that function. In this case, the programmer can scope the memoized dynamic programming algorithm inside of an auxiliary function so that the memo table of the algorithm is discarded as soon as the auxiliary function returns. As an example, consider the Knapsack Problem shown in Figure 3. The function `mks` shows the memoized solution to the Knapsack Problem. A critical aspect of this algorithm is that the result re-use occurs among recursive calls and re-use between two independent calls to `mks` are unlikely to share results. Thus `mks` 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.

4 MFL

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 dynamic dependencies 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 MFL programs are efficinent by demonstrating that memoization causes a constant factor slowdown even in the worst case, where no results are re-used. For the sake of brevity we do not formalize "boxing" for user-controlled, constant-time equality checking. Although this is crucial for a practical language for selective memoization, the techniques for adding such a mechanism are well-understood, and would only distract from the main points studied here.

4.1 Abstract Syntax

$$\begin{array}{llll}
\textit{Types} & \tau & ::= & \texttt{int} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu u.\tau \mid \tau_1 \to \tau_2 \mid \texttt{!}\,\tau \\
\textit{Op's} & o & ::= & \texttt{not} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots \\
\textit{Expr's} & e & ::= & \texttt{return}(t) \mid \texttt{let}\,\texttt{!}\,x{:}\tau\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} \mid \\
 & & & \texttt{let}\,a_1{:}\tau_1 \times a_2{:}\tau_2\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} \mid \\
 & & & \texttt{mcase}\,t\,\texttt{of}\,\texttt{inl}\,(a_1{:}\tau_1) \Rightarrow e_1 \mid \texttt{inr}\,(a_2{:}\tau_2) \Rightarrow e_2\,\texttt{end} \\
\textit{Terms} & t & ::= & v \mid o(t_1,\ldots,t_n) \mid \langle t_1, t_2\rangle \mid \texttt{mfun}\,f\,(a{:}\tau_1){:}\tau_2\,\texttt{is}\,e\,\texttt{end} \mid \\
 & & & t_1\,t_2 \mid \texttt{!}\,t \mid \texttt{inl}_{\tau_1+\tau_2}t \mid \texttt{inr}_{\tau_1+\tau_2}t \mid \texttt{roll}(t) \mid \texttt{unroll}(t) \\
\textit{Values} & v & ::= & x \mid a \mid n \mid \texttt{!}\,v \mid \langle v_1, v_2\rangle \mid \texttt{mfun}_l\,f\,(a{:}\tau_1){:}\tau_2\,\texttt{is}\,e\,\texttt{end}
\end{array}$$

Figure 4: The abstract syntax of MFL.

The abstract syntax of MFL is given in Figure 4. The meta-variables x and y range over a countable set of *variables*. The meta-variables a and b range over 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*.

The syntax is structured into *terms* and *expressions*, in the terminology of Pfenning and Davies [35]. 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. 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 [35] with a "necessitation" modality, $!\,\tau$, which is used to track data dependencies 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 readily imagine adding such a modality to the language. The introductory and eliminatory forms for necessity are standard, namely $!\,t$ for introduction, and $\texttt{let}\,!\,x{:}\tau\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end}$ 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 [34, 36]: variables correspond to the intuitionistic context, resources to the linear context.[3]

The inclusion, $\texttt{return}(t)$, 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 $\texttt{return}(t)$ requires that t be resource-free to ensure that any dependency 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 $\texttt{let}\times$ and `case`. 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 5 and 6.

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 complete value(s). This is the key 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.

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

Var, Res
$$\frac{(\Gamma(x) = \tau)}{\Gamma; \Delta \vdash x{:}\tau} \qquad \frac{(\Delta(a) = \tau)}{\Gamma; \Delta \vdash a{:}\tau}$$

Numbers
$$\frac{}{\Gamma; \Delta \vdash n : \texttt{int}}$$

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

Pairs
$$\frac{\Gamma; \Delta \vdash t_1 : \tau_1 \quad \Gamma; \Delta \vdash t_2 : \tau_2}{\Gamma; \Delta \vdash \langle t_1, t_2 \rangle : \tau_1 \times \tau_2}$$

Fun
$$\frac{\Gamma,\ f{:}\tau_1 \to \tau_2; \Delta, a{:}\tau_1 \vdash e : \tau_2}{\Gamma; \Delta \vdash \texttt{mfun}\, f\ \texttt{(}a\,{:}\,\tau_1\texttt{)}\,{:}\,\tau_2\ \texttt{is}\, e\, \texttt{end} : \tau_1 \to \tau_2}$$

FunVal
$$\frac{\Gamma, f{:}\tau_1 \to \tau_2; \Delta, a{:}\tau_1 \vdash e : \tau_2}{\Gamma; \Delta \vdash \texttt{mfun}_l\, f\,\texttt{(}a\,{:}\,\tau_1\texttt{)}\,{:}\,\tau_2\ \texttt{is}\, e\, \texttt{end} : \tau_1 \to \tau_2}$$

Apply
$$\frac{\Gamma; \Delta \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma; \Delta \vdash t_2 : \tau_1}{\Gamma; \Delta \vdash t_1\, t_2 : \tau_2}$$

Bang
$$\frac{\Gamma; \bullet \vdash t : \tau}{\Gamma; \Delta \vdash \texttt{!}\, t : \texttt{!}\, \tau}$$

Inl and Inr
$$\frac{\Gamma; \Delta \vdash t : \tau_1}{\Gamma; \Delta \vdash \texttt{inl}_{\tau_1+\tau_2} t : \tau_1 + \tau_2} \qquad \frac{\Gamma; \Delta \vdash t : \tau_2}{\Gamma; \Delta \vdash \texttt{inr}_{\tau_1+\tau_2} t : \tau_1 + \tau_2}$$

Roll and Unroll
$$\frac{\Gamma; \Delta \vdash t : [\mu u.\tau/u]\tau}{\Gamma; \Delta \vdash \texttt{roll(}t\texttt{)} : \mu u.\tau} \qquad \frac{\Gamma; \Delta \vdash t : \mu u.\tau}{\Gamma; \Delta \vdash \texttt{unroll(}t\texttt{)} : [\mu u.\tau/u]\tau}$$

Figure 5: Typing for terms

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 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.

Return $$\frac{\Gamma; \bullet \vdash t : \tau}{\Gamma; \Delta \vdash \texttt{return}(t) : \tau}$$

Let! $$\frac{\Gamma; \Delta \vdash t : \,!\,\tau \quad \Gamma, x{:}\tau; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \texttt{let}\,!\,x\,\texttt{be}\,t\,\texttt{in}\,e\,\texttt{end} : \tau}$$

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

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

Figure 6: Typing judgments for expressions.

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 7 and 8. 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.

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

We write $\beta\widehat{\ }\varepsilon$ 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.[4] 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 `case` expressions extend the current branch to

[4]In the presence of "box" to support "pointer equality," memo tables would be required to respect value equivalence.

Number $$\sigma, n \Downarrow^{\mathsf{t}} n, \sigma$$

PrimOp
$$\frac{\begin{array}{rcl} \sigma, t_1 & \Downarrow^{\mathsf{t}} & v_1, \sigma_1 \\ \sigma_1, t_2 & \Downarrow^{\mathsf{t}} & v_2, \sigma_2 \\ & \vdots & \\ \sigma_{n-1}, t_n & \Downarrow^{\mathsf{t}} & v_n, \sigma_n \\ \multicolumn{3}{c}{v = \mathtt{app}(o\mathtt{,}(v_1, \ldots, v_n)\mathtt{)}} \end{array}}{\sigma, o(t_1, \ldots, t_n) \Downarrow^{\mathsf{t}} v, \sigma_n}$$

Pair
$$\frac{\begin{array}{rcl} \sigma, t_1 & \Downarrow^{\mathsf{t}} & v_1, \sigma' \\ \sigma', t_2 & \Downarrow^{\mathsf{t}} & v_2, \sigma'' \end{array}}{\sigma, \langle t_1, t_2 \rangle \Downarrow^{\mathsf{t}} \langle v_1, v_2 \rangle, \sigma''}$$

Fun
$$\frac{(l \notin \mathrm{dom}(\sigma), \quad \sigma' = \sigma[l \mapsto \emptyset])}{\sigma, \mathtt{mfun}\, f\ \mathtt{(}a\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end} \Downarrow^{\mathsf{t}} \mathtt{mfun}_l\, f\, \mathtt{(}a\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\, e\, \mathtt{end}, \sigma'}$$

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

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

Bang
$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \mathtt{!}\, t \Downarrow^{\mathsf{t}} \mathtt{!}\, v, \sigma'}$$

Inject
$$\frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \mathtt{inl}_{\tau_1+\tau_2} t \Downarrow^{\mathsf{t}} \mathtt{inl}_{\tau_1+\tau_2} v, \sigma'} \qquad \frac{\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'}{\sigma, \mathtt{inr}_{\tau_1+\tau_2} t \Downarrow^{\mathsf{t}} \mathtt{inr}_{\tau_1+\tau_2} v, \sigma'}$$

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

Figure 7: Evaluation of terms

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.

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

Return

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

Let!

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

Let$\times$

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

Case

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

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

Figure 8: Evaluation of expressions

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 the pure fragment of MFL by an inductive definition of the relations $t \Downarrow^{\mathsf{t}}_{\mathsf{p}} v$ and $e \Downarrow^{\mathsf{e}}_{\mathsf{p}} v$. Here t, e, and v are "pure" in the sense that they may not involve subscripted function values. The *underlying term*, t^-, of an MFL term, t, is obtained by erasing all location subscripts on function values occurring within t.

The soundness of MFL consists of showing that evaluation with memoization yields the same outcome as evaluation without memoization.

Theorem 1 (Soundness)
If $\emptyset, t \Downarrow^{\mathsf{t}} v, \sigma$, where $\bullet; \bullet \vdash t : \tau$, then $t^- \Downarrow^{\mathsf{t}}_{\mathsf{p}} v^-$.

Proof: The full proof is given in Appendix A. The statement of the theorem must be strengthened considerably to account for both terms and expressions, and to take account of non-empty memoization contexts. The proof then proceeds by induction on evaluation. ■

It is easy to show that the non-memoizing semantics of MFL is type safe, using com-

```
signature MEMO =
sig
  type 'a res
  type 'a memoized

  val init:  unit -> unit                                             (* Initialize the library *)
  val expose:  'a res -> 'a                                           (* Expose a resource *)

  val memoize':  ('a res -> 'b memoized) -> ('a -> 'b)            (* Memo a recursive function *)
  val memoize:  (('a -> 'b) -> 'a res -> 'b memoized) -> ('a -> 'b) (* Memo a non-recursive function *)
  val letBang:  ('a -> int) -> (unit -> 'a) -> ('a -> 'b memoized) -> 'b memoized        (* let!  *)
  val letX: (unit -> 'a*'b) -> (('a res*'b res) -> 'c memoized) -> 'c memoized          (* let* *)
  val mcase:  ('a -> bool) -> (unit -> 'a) ->
              ('a res -> 'b memoized) -> ('a res -> 'b memoized) -> 'b memoized   (* Memoized case *)
  val return:  (unit -> 'a) -> 'a memoized
end
```

Figure 9: The signature of the memoization library

pletely conventional techniques. It follows that the memoizing semantics is also type-safe, for if not, there would be a closed value of a type τ that is not canonical for that type. However, erasure preserves and reflects canonical forms, hence, by the Soundness Theorem, MFL must also be type safe.

4.5 Performance

We show that memoization slows down an MFL program by no more than a constant factor (expected) even when no results are re-used. The proof of this claim is relatively straightforward. We will bound the overhead of memoization with respect to a non-memoizing version of MFL's operational semantics. Imagine modifying the operational semantics so that 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 the 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 lookup (or update) with a branch β and let $|\beta|$ be the length of the branch. Using nested hash tables, a lookup (or update) can be performed in expected $O(|\beta|)$ time. But 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 `case`. Furthermore, each evaluation of `let!` and `case` can be charged by exactly one `return`. Thus, we conclude that $T' = O(T)$ in the expected case.

5 Implementation

A key property of the framework is that it accepts a simple implementation. In this section we present a brief tour of an interface for the library in the Standard ML language. The implementation itself along with code for the examples presented in Section 3 can be found in Appendix B. In Standard ML, it is not possible to capture statically the syntactic distinction between resources and variables, and also between terms and expression, therefore we use run-time checks to ensure correct usage of primitives. We describe how to incorporate these run-time checks to the library in Appendix B.

Figure 9 shows the signature of the library. The interface is somewhat different than the MFL language in several aspects. First, we introduce a resource type `res` for resources to enforce correct usage. Resources are created only by the library. A parameter of a memoized function is turned into a resource at a function call and more resources can be created by touching other resources using `letX` and `mCase`. The value of a resource can be accessed via the `expose` primitive. Second, we do not have an explicit bang (`!`) type. Rather, the library allows any value to be touched. All memoized expression have return type `'a memoized`. All terms that are examined by expressions occur in the suspended form `unit -> 'a`. This is used in checking for correct usage by setting certain flags at run-time before forcing the value of the term.

The primitives `memoize'` and `memoize` are used to memoize functions whose bodies are memoized expressions. The typing requirement dictate that we distinguish between recursive and non-recursive function. The primitive `memoize'` takes a non-recursive function and returns a memoized version of the function. The parameter to `memoize'` must be a function from a `res` type to a `memoized` type. The primitive `memoize` takes a recursive function and returns a memoized version of the function. One subtle issue is that the parameter to `memoize` must recursively call its *memoized* version and therefore it should take this memoized version as a parameter.

The primitive `letBang` is used to *touch* a value. Its first parameter is a hash function of type `'a -> int` that maps the value touched to a hash index; this index is used for memo-table lookup and extensions. The second parameter is a term. The third parameter to `letBang` is the body, which is a memoized expression.

The primitive `letX` is used to touch tuples. The first parameter is term of a product type, and the second parameter is a the body in the form of a function that operates on the two parts of the tuple. Note that the two parts are resource types. The primitive `mCase` is a primitive for touching sum types. Due to type system limitations, however, its implementation is somewhat contrived. The first parameter is a function that indicates which branch to take and its second parameter is a term that supplies a value. The third and the fourth parameters are the body of the two branches. The primitive `return` takes a body and returns the result of the body in memoized form.

6 Discussion and Future Work

In this paper we describe a framework for selective memoization under programmer control. The key property of the framework is that it is general yet efficient. This efficiency guarantee comes with some limitations particularly in determining the precise dependencies and space management.

For the case of space management, our technique allows the programmer to control the life-span of complete memo tables. It may be preferable to manage the life span of individual memo-table entries as well; our proposal does not address this issue. As for identifying precise dependencies, we do not capture "deep" dependencies that the work of Abadi *et. al.* [1] and that of Heydon *et. al.* [18] does. In fact, we only keep track of "local" (or "shallow") dependencies – that is we do not propagate the dependencies of a callee to the caller. For example, consider the functions `fun swap (x,y) = (y,x)` and `fun second (x,y) = let (x',y') = swap (x,y) in x' end`. These can be written in our framework as

```
mfun mswap a = letX (a1,a2) = a in let !x1 = a1 in let !x2 = a2 in
  return (!x2,!x1) end end end

mfun msecond a = letX (a1,a2) = swap a in let !x1 = a1 in
  return x1 end end
```

In the previous work, the function `second` will only depend on `y`. Thus as soon as `second` is called the cache will be checked using `y` and a match may be found. In our framework, the function `mswap` will be called before `return` (of `msecond`) performs a cache lookup. This could seem like a disadvantage at first, but it enables us to capture dependencies on approximations of input as discussed in Section 3. Furthermore, there are various problems with propagation of deep dependencies. One issue is that the dependencies can grow large and one may need to introduce cut points to control this [18]. Another issue is that the program analysis can change running time of a program significantly. More importantly, in earlier work, we showed that "deep" dependencies are best handled using modifiable references [2].

In the context of adaptive or incremental computing, the memoization framework presented here complements the adaptive computing technique with dynamic dependency graphs [2]. Memoization handles shallow changes well, whereas dynamic dependency graphs handles deep changes well. Therefore, we expect that these two techniques can be combined to obtain a general technique for obtaining dynamic or kinetic algorithms [12, 6]. In fact, we implemented a preliminary version of a library that combines these two techniques. The library enables one to write fairly sophisticated dynamic algorithms and also yields to performance analysis. A particular advantage of memoization in the context of the adaptive computing framework is that it accepts a simple caching and purging scheme. In fact, *time stamps* [2] can be used to determine if a result should be purged from the cache. On the other hand, combining these two techniques is nontrivial because of the interaction between memoization and modifiables – in particular it is important to isolate the side effects performed during change propagation. In future work, we are planning to investigate this proposal.

References

[1] Martin Abadi, Butler W. Lampson, and Jean-Jacques Levy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the Twenty-ninth Annual ACM-SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2002.

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

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

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

[6] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.

[7] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[8] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 2002.

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

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[11] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Conference Record of the 8th Annual ACM Symposium on POPL*, pages 105–116, January 1981.

[12] D. Eppstein, Z. Galil, and G. Italiano. Dynamic graph algorithms, 1997.

[13] 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.

[14] Eiichi Goto. Monocopy and associative algorithms in extended lisp. Technical Report TR-74-03, University of Tokyo, 1974.

[15] Eiichi Goto and Yasumasa 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.

[16] Marty Hall and J. Paul McNamee. Improving software performance with automatic memoization.

[17] A. Heydon, R. Levin, T. Mann, and Y. Yu. The vesta approach to software configuration management, 1999.

[18] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. *ACM SIGPLAN Notices*, 35(5):311–320, 2000.

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

[20] Roger Hoover. Alphonse: incremental computation as a programming abstraction. In *Proceedings of the 5th ACM SIGPLAN conference on Programming language design and implementation*, pages 261–272. ACM Press, 1992.

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

[22] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

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

[24] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1 May 1998.

[25] James Mayfield, Time Finin, and Marty Hall. Using automatic memoization as a software engineering tool in real-world ai systems.

[26] John 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.

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

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

[29] Tom Murphy, Robert Harper, and Karl Crary. The wizard of tilt: Efficient(?), convenient and abstract type representations. Technical Report CMU-CS-02-120, School of Computer Science, Carnegie Mellon University, March 2002.

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

[31] H. A. Partsch. *Specification and Transformation of Programs–A Formal Approach to Software Development.* Springer-Verlag, 1990.

[32] Maarten Pennings. *Generating Incremental Attribute Evaluators.* PhD thesis, University of Utrecht, November 1994.

[33] Maarten Pennings, S. Doaitse Swierstra, and Harald 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.

[34] Frank 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.

[35] Frank Pfenning and Rowan 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.

[36] Jeff Polakow and Frank 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.

[37] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual Symposium on POPL*, pages 315–328, January 1989.

[38] William Pugh. *Incremental computation via function caching.* PhD thesis, Department of Computer Science, Cornell University, 1987.

[39] William Pugh. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 269–276. ACM Press, 1988.

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

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

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

A Soundness

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, [35]). We will show that, under suitable conditions, 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 $\texttt{!}\,v$ in β by the corresponding underlying event, $\texttt{!}\,(v^-)$.

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 size of the expression.

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

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 $\sigma, t \Downarrow^{\mathsf{t}} v, \sigma'$, $\sigma(l) @ \beta = \texttt{return(}t\texttt{)}$, 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:

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

We write $\gamma\hat{}\epsilon$ 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, $\texttt{(}v\texttt{)}$, and "pair" events, $\langle v_1, v_2\rangle$, and by omitting the arguments to "injection" events, $\texttt{inl(}v\texttt{)}$, $\texttt{inr(}v\texttt{)}$. 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:

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

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

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 $e \mathbin{@} \gamma =$ `return(`t`)`, then $e \mathbin{@} \gamma^\circ =$ `return(`t`)`.

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 $\texttt{mfun}_l\, f\,\texttt{(}a\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\ \texttt{is}\ e\ \texttt{end}$ occurs in either t or e, then $\Sigma(l) = \texttt{mfun}\, f\ \texttt{(}a\,\texttt{:}\,\tau_1\texttt{)}\,\texttt{:}\,\tau_2\ \texttt{is}\ e^-\ \texttt{end}$. 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. $\Sigma(l) \mathbin{@} \beta^- = \texttt{return(}t^-\texttt{)}$,
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, $\bullet;\bullet \vdash t : \tau$, 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 $\bullet;\bullet \vdash e : \tau$, 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 $t = \mathtt{mfun}_l\, f\,\mathtt{(}a\,\mathtt{:}\,\tau_1\mathtt{)}\,\mathtt{:}\,\tau_2\,\mathtt{is}\,e\,\mathtt{end}$, simply take $\Sigma' = \Sigma$ and note that $v = t$ and $\sigma' = \sigma$.

For function terms $t = \mathtt{mfun}\, f\ \mathtt{(}a\,\mathtt{:}\,\tau_1\mathtt{)}\,\mathtt{:}\,\tau_2\,\mathtt{is}\,e\,\mathtt{end}$, note that $v = \mathtt{mfun}_l\, f\,\mathtt{(}a\,\mathtt{:}\,\tau_1\mathtt{)}\,\mathtt{:}\,\tau_2\,\mathtt{is}\,e\,\mathtt{end}$ 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 $v_1 = \mathtt{mfun}_l\, f\,\mathtt{(}a\,\mathtt{:}\,\tau_1\mathtt{)}\,\mathtt{:}\,\tau_2\,\mathtt{is}\,e\,\mathtt{end}$, 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 $\sigma, l{:}\beta, \mathtt{return(}t\mathtt{)} \Downarrow^{\mathsf{e}} v, \sigma'$ with Σ consistent with $\mathtt{return(}t\mathtt{)}$, σ tracking Σ, $\gamma^\circ = \beta$, $\Sigma(l) \mathbin{@} \gamma^- = (\mathtt{return(}t\mathtt{)})^- = \mathtt{return(}t^-\mathtt{)}$, and $\bullet;\bullet \vdash \mathtt{return(}t\mathtt{)} : \tau$. Note that by Lemma 3, $(\Sigma(l) \mathbin{@} \beta)^- = \Sigma(l) \mathbin{@} \beta^- = \mathtt{return(}t^-\mathtt{)}$.

For the first case, suppose that $\sigma(l)(\beta) = v$. Since σ tracks Σ and $l \in \mathrm{dom}(\sigma)$, we have $\Sigma(l) = \mathtt{mfun}\, f\ \mathtt{(}a\,\mathtt{:}\,\tau_1\mathtt{)}\,\mathtt{:}\,\tau_2\,\mathtt{is}\,e^-\,\mathtt{end}$ with $e^- \mathbin{@} \beta^- = \mathtt{return(}t^-\mathtt{)}$, 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 $\Sigma'(l) \mathbin{@} \beta^- = \mathtt{return(}t^-\mathtt{)}$, which we noted above.

■

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.

```
signature MEMO =
sig
  type 'a res
  type 'a memoized

  val init:  unit -> unit                                                  (* Initialize the library *)
  val expose:  'a res -> 'a                                                (* Expose a resource *)

  val memoize':  ('a res -> 'b memoized) -> ('a -> 'b)              (* Memo a recursive function *)
  val memoize:  (('a -> 'b) -> 'a res -> 'b memoized) -> ('a -> 'b) (* Memo a non-recursive function *)
  val letBang:  ('a -> int) -> (unit -> 'a) -> ('a -> 'b memoized) -> 'b memoized        (* let!  *)
  val letX: (unit -> 'a*'b) -> (('a res*'b res) -> 'c memoized) -> 'c memoized           (* let* *)
  val mcase:  ('a -> bool) -> (unit -> 'a) ->
              ('a res -> 'b memoized) -> ('a res -> 'b memoized) -> 'b memoized   (* Memoized case *)
  val return:  (unit -> 'a) -> 'a memoized
end
```

Figure 10: The signature of the memoization library

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

signature MEMOPAD = sig
  type 'a memopad
  type index
  val empty:  unit -> 'a memopad
  val extend:  'a memopad -> index list -> ('a option * 'a memopad option)
  val add:  'a -> 'a memopad -> unit
end
```

Figure 11: The signatures for boxes and memopads.

B Implementation

In this section, we present the code for an implementation of our framework in the Standard ML language. The interface for the library is given in Figure 10. The implementation relies on an interface for boxing/unboxing and memo tables; these are shown in Figure 11. Figure 12 shows a simple implementation of the library without run-time checks. Figure 13 demonstrates the code for the examples described in Section 3.

The run time checks are relatively straightforward to incorporate by enhancing the definition of the resources (this approximately doubles the number of lines). The enhancement ensures that resources are never exposed inside a `return` and never escape their scope. This can be achieved by tagging each resource with the function instance that it belongs to and then invalidating these tags at a `return`. Correct usage of a resource is ensured by checking it tag is valid before an expose.

Correct usage of terms and expressions cannot statically be enforced in SML. For example, there is no way to prevent the body of a memoized function to be a statement of the form `if (expose r) then return 0 else return 1`. This is not correct usage because `if` is not a proper expression (it does not record the branch taken). The way to get around this is to make sure that each `expose` occurs at a term position and a term position will only be gained through an expression. For example a statement of the form `mcase (fn () => expose r) ...` will signal a term position before forcing `fn () => expose r`. This is easy to implement by keeping a flag indicating a term position.

```
functor BuildMemo (structure Box:  BOX structure Memopad:MEMOPAD where type index = int):MEMO =
struct
  type 'a memoized = int list * (unit -> 'a)
  type 'a box = 'a Box.box

  type 'a res = 'a   (* More sophisticated definition is needed for checking correct usage *)
  fun expose x = x
  fun resource v = v

  fun init () = (Box.init ())

  fun memoize f = let
    val mpad = Memopad.empty ()
    fun mf rf x = let
      val (branch,thunk) = f rf x
      val result =
        case Memopad.extend mpad branch of
          (NONE,SOME mpad') => let      (* Not Found in the memo *)
              val v = thunk ()
              val _ = Memopad.add v mpad'
            in
              v
            end
         | (SOME v,NONE) => v           (* Found in the memo *)
    in
      result
    end
    fun mf' x = mf mf' (resource x)
  in
    mf'
  end

  fun memoize' f = ...  (* Similar to memoize *)

  fun letBang h t f = let
    val (branch,thunk) = f (t ())
  in
    ((h (t ()))::branch, thunk)
  end

  fun letX t f = let
    val (x1,x2) = x ()
  in
    f (resource x1, resource x2)
  end

  fun mcase h t f1 f2 = let
    val v = t ()
    val lr = h v
    val (branch,thunk) = if lr then f1 v else f2 v
  in
    if lr then (0::branch,thunk)
    else (1::branch,thunk)
  end

  fun return f = (nil,f)

end
```

Figure 12: The implementation of the memoization library (correct usage not checked).

```
(* Some utilities *)
fun letI r body = letBang (fn i => i) r body (* let!  for ints *)
fun letB r body = letBang (fn b => Box.getKey b) r body (* let!  for boxes *)

fun fib f (n:int res) =
  letI (fn()=>expose n) (fn n' => return (fn()=>
    if n' < 2 then n'
    else f(n'-1) + f(n'-2)))
val mfib = memoize fib (* Memoized Fibonacci *)

fun sum x =
  letX (fn()=>expose x) (fn (x1,x2) =>
    mcase (fn v => v >= 0) (fn()=>expose x1)
           (fn _ => return (fn()=>0))
           (fn _ => letI (fn()=>expose x1)
                   (fn x1' => letI (fn()=>expose x2) (fn x2' =>
                    return (fn()=>x1'+x2')))))
val msum = memoize' sum

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

fun hashCons (x:  ('a box * (('a box) boxedlist)) res) =
  letX (fn()=>expose x) (fn (h,t) => letB (fn()=>expose h) (fn h' => letB (fn()=>expose t) (fn t' =>
    return (fn()=>box (CONS(h',t'))))))
val mhashCons = memoize' hashCons

datatype 'a tree = EMPTY | NODE of 'a * ('a tree * 'a tree)
fun isEqual' a = letX (fn()=>expose a) (fn (a1,a2) => letI (fn()=>expose a1) (fn a1' =>
                                       letI (fn()=>expose a2) (fn a2' => return (fn()=>a1'=a2'))))
val isEqual = memoize' isEqual'
fun isLess' a = letX (fn()=>expose a) (fn (a1,a2) => letI (fn()=>expose a1) (fn a1' =>
                                     letI (fn()=>expose a2) (fn a2' => return (fn()=>a1'<a2'))))
val isLess = memoize' isLess'

fun search msearch arg = letX (fn()=>expose arg) (fn (tree,key) =>
  mcase (fn v => if (v = EMPTY) then true else false) (fn()=>expose tree)
         (fn _ => return (fn()=>box NIL))
         (fn n => let val NODE(klr) = (fn()=>expose n) () in
           letX (fn()=>klr) (fn (k,lr) =>
           letX (fn()=>expose lr) ( fn (l,r) =>
           letI (fn()=>expose k) (fn k' =>
             mcase (fn v => v) (fn()=>isEqual (k', expose key))
                   (fn _ => return (fn()=>fromList[k']))
                   (fn _ => mcase (fn v => v) (fn()=>isLess(expose key,k'))
                                  (fn _ => letB (fn()=>msearch (expose l,expose key)) (fn r =>
                                               return (fn()=>box (CONS(k',r)))))
                                  (fn _ => letB (fn()=>msearch (expose r,expose key)) (fn r =>
                                               return (fn()=>box (CONS(k',r)))))))))
           end))

fun mks' ks (args) = letX (fn()=>expose arg) (fn (c, l) =>
  letI (fn()=>expose c) (fn c' =>
    if c' <= 0 then return (fn()=>0)
    else letB (fn()=>expose l) (fn l' => return (fn () =>
      case (unbox l') of
        NIL => 0
       | CONS((w,v),t) =>
          if (c' < w) then mks(c',t)
          else let
            val v1 = mks (c',t)
            val v2 = v + mks (c'-w,t)
          in
            if (v1 > v2) then v1
            else v2
          end))))
val mks = memoize mks'
```

Figure 13: Examples of Section 3 in the SML library