Memo-Functions In Alfl

Pradeep Varma and Paul Hudak

Research Report YALEU/DCS/RR-759
December 1989

This work is supported by the National Science Foundation CCR-8809919.

Memo-Functions In Alfl

Pradeep Varma
Paul Hudak

Department of Computer Science
Yale University
Box 2158 Yale Station
New Haven, CT 06520

Dec 1989

Abstract

Memoisation is a well known technique for increasing the efficiency of elegantly written programs. In this paper we present a memo-scheme that has been designed and implemented for Alfl, a non-strict functional language at Yale. The scheme permits a choice between user defined memo-functions and a default which is which is lazy memoisation. Allowing a user full control over the design of memo-functions causes the loss of referential transparency. The situation however is not intractable, as simple proof conditions can be given using which one can guarantee a safe behavior of these functions. The conditions and a precise description of the work via denotational semantics is given.

1 Introduction

Why memoisation? This is the first question that arises as we begin to delve in the subject. Why should we worry about memoising anything? The answer is, memoisation provides a way of increasing efficiency without compromising on elegance. It does so by avoiding unnecessary computation. Externally a memo-function is just like any other function. The difference is internal. A memo-function "remembers" the results of its applications so that if called on the same arguments again, the old answer is returned, no recomputation occurs.

An simple example to demonstrate the method is fibonacci. The following[1] is how it is usually defined.

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

It is an elegant definition. It is also terribly inefficient, exponential in time requirements. Now consider a slight variation of it.

```
fibonacci 0 == 0;
fibonacci 1 == 1;
fibonacci n == fib(n-1) + fib(n-2);
fib == memoise fibonacci;
```

This version shows a linear performance in terms of space and time. The improvement is because of the recomputation avoided by converting `fibonacci` into a memo-function via `memoise`. Assuming that this much of distortion[2] in the original definition is acceptable, we have achieved our goal of gaining efficiency without compromising style or readability.

A problem is immediately obvious. We cannot do this optimization blindly. If we have side effects in our language, then they may necessitate recomputation each time. However since we are only interested in functional languages, this becomes a non issue for us. Identifying equal expressions and avoiding recomputation is an old implementation trick. Lazy evaluation, common subexpression elimination are examples. However the amount of recomputation they avoid is limited. Identification of dynamically created equal expressions, as in the example above, requires aid from the programmer. Memo-functions can be viewed as a way of providing it.

A thing to note about the fibonacci example is that nowhere have we mentioned how the memoised version remembers the results it computes. This information is packaged in the definition of `memoise` which can be defined anywhere. Writing programs so as to get a clean break between *what* is being

[1]This and all other examples are given in Alfl syntax, a partial glossary for which is given in appendix C.
[2]Syntax can be used for further packaging. We choose otherwise for reasons discussed in section 2.

computed, and *how* it is being computed is regarded as good style[3]. Memoisation is helpful in realizing it.

All this indicates the desirability of having memoisation. What's needed next is a handle to it in a language. This forms the topic of discussion for the rest of the paper. We work with Alfl, a higher order, lazy, dynamically typed, functional language, and extend it by a predefined function called `memo`. This extended language permits a user three options in memoising. The easiest is a default, called lazy memoisation. Here the user only specifies the number of arguments he wants to cache a function on. All other details are left to the implementation. To get finer control, a user must either switch to the so called applicative caching scheme, or the non-applicative one. Applicative caching does not use the language extension. It relies instead on lazy evaluation. This is nice, minimizing language extensions is a good policy. However it is also limited. Specification of schemes such as a binary search tree, that take advantage of the dynamic behavior of a program is not possible. Doing this requires a shift to non-applicative memoisation. As the name suggests, we also exit purely functional programming upon doing so. This is unfortunate, but not intractable. We give a set of simple proof conditions, which if followed guarantee that a cached function will behave just like an uncached one. These conditions, along with a discussion of the three schemes, their merits and demerits, are given. Also given is a formal description of the extended language for reasons of portability and exact understanding. To begin, a description of the Alfl extension `memo` is given first.

2 The extension to Alfl

Alfl is extended by the addition of a predefined function `memo`. Its use is as outlined below.

```
    caching_scheme == memo scheme;
```

`Caching_scheme` takes a function and returns a memoised version of it. The way it caches results is as described by `scheme`. If

```
    scheme = n, n is a positive integer,
                then caching_scheme memoises a function by lazy
                memoisation on n arguments.
    scheme = [n, x, y, z], n is a positive integer,
                then caching_scheme memoises a function by non-applicative
                memoisation on n arguments using
                        x as the initial table,
                        y as the lookup function and
                        z as the update function.
```

[3]Separate specification of the functional and operational behaviors of programs is the subject of study in para-functional programming. Some references are [7], [8].

The number of arguments a function is to be cached on, counted from left to right and often equal to the arity, has to specified explicitly. A way to make it implicit is to insist that n is equal to the number of arguments on the left hand side of a function definition. Hughes [2] follows this method. Also, he uses a keyword *memo* to precede a function definition to indicate memoisation. There are several reasons why we choose to not take this syntactic approach. First, the effective arity of a function cannot generally be computed at compile time since Alfl is higher order, curried, and dynamically typed. Even if possible, the user may desire a different value of n than this. So fixing the value syntactically can be quite undesirable. Second, using syntax is justified when a significant amount of saving occurs in the typing effort on the keyboard. This doesn't seem to be the case here. So we would like to leave the language as simple as possible. Finally, by making the modification to the language simple, just a predefined function, the implementation effort is simplified.

3 Lazy memoisation

In 1985, John Hughes [2] came up with a scheme that relies on the knowledge that most implementations of functional languages use objects that have a unique object identity (pointers in Lisp) associated with them. A memo-function can use this identity value as a tag for looking up precomputed results. Since the test for equality of object identities (eq? test in Lisp) is quite efficiently implementable, such functions would be quite fast. Also, since identities exist for all objects including infinite lists, the function would work on any argument. However such a function is not a *full* memo function. Given two equal objects with different identities, the function would be able to resolve between them. Thus recomputation may be caused even when not needed. This is unfortunate, however the other advantages are still quite attractive. In particular the ability to work on infinite objects. No full memo scheme can work on such objects since comparing two infinite objects for equality is a potentially non terminating process. The fact that lazy memoisation is able to compare such objects or for that matter any tree or list in constant time is very welcome indeed.

How does one provide the user such memo functions to work with? One can't just provide a function like object-identity. For then referential transparency is lost as resolution between equal objects is possible. A solution is to make this scheme a default scheme of memoisation. The user no longer has to define the implementation[4], and hence does not have to deal with object-identity and related issues. Providing the user a general and efficient default scheme is a good idea in its own right. It relieves him of the burden of defining a memo scheme each time he thinks of memoising a function. This is the solution we opt for in Alfl. The <argument,value> pairs, or more correctly <argument-identity, value> pairs, are stored in hash tables. The tables interact with the garbage collector in a fashion that permits an entry to be deleted once the corresponding argument is collected. This space reclamation feature is another advantage that accrues from the fact that the scheme is not user definable. Because of implemention at a lower level, optimizations can be carried out, that are not obvious at the language level.

At this point it is interesting to note that the name lazy memoisation is really a misnomer. A lazy memo-fn f is not really lazy since (f $\perp$) is still $\perp$. Evaluation of the object-identity of arguments

[4]User defined lazy schemes are possible, but not inexpensively. See appendix B

leads to strictification. Another important point that hasn't been clarified so far is what exactly does the identity of an object mean? The example, pointers in Lisp, is good for intuition, however a formal description must be given. We formalize lazy evaluation by giving a store semantics of the language in appendices A and B.

We now present an example, taken from [2], that highlights the power of lazy memoisation. For other interesting examples, and detail, we refer the reader to the same.

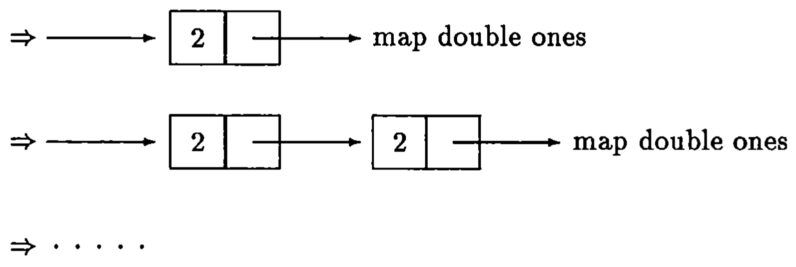
3.1 Example: Manipulation of cyclic structures

Infinite objects are most efficiently dealt with as cyclic structures. It is highly desirable that operations manipulating these objects maintain them in this form. As this example demonstrates, lazy memoisation can be quite helpful.

```
ones == 1^ones;
twos == map double ones;
double x == 2 * x;
```

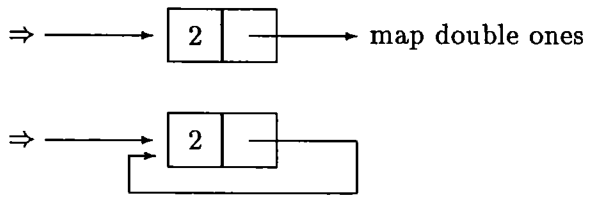
When twos is computed, a non-cyclic structure representing the infinite list twos is created. As more and more of twos is evaluated, the space required by this structure also increases.

map double ones



If map is memoised lazily then a cyclic structure is created. The second application of map is identical to the first, hence the same cons cell is returned. The space requirements reduce to a constant value, and the time performance also improves.

map double ones



⇒ *same*

3.2 Insufficiency of lazy memoisation

The ideal scenario would be if just a default scheme were sufficient for all applications. Unfortunately this is not so. Often the user has sufficient knowledge of a problem to be able to design optimal schemes. He must be allowed a way of expressing this. Another problem is that lazy memoisation can be hard to reason about. Being dependent on the implementation notion of object identities, it requires the user to have knowledge of the implementation for effective use. Optimizations such as common subexpression elimination can make this process harder. Since the scheme is not really lazy and strictifies a function, it also necessitates reasoning about termination of arguments for correct usage. These reasons lead us to consider alternative schemes for Alfl.

4 Applicative Caching

Building a caching scheme does not necessarily require an extension to a language. Alfl has lazy evaluation, we are free to make full use of this fact. If we build a structure that contains all the function applications we are interested in, repeated lookups will cause only singular computation. This is what lazy evaluation guarantees . The following examples demonstrates this. Following Keller and Sleep's terminology [1], we call such caches applicative caches.

A stream cache.

```
cache f == {enumerate f i == (f i)^(enumerate f (i + 1));
            result @ x == nth x table;
            table == enumerate f 1;
           };
```

`cache` takes a function and returns a corresponding memo-function. The table here is an infinite list of the function applications `(f 1)^(f 2)^...` Each time a cached function is applied on an argument i, the i^{th} element of the list gets picked up. The first call forces the evaluation, later ones simply reuse it.

Another example is a vector cache.

```
cache_gen n f == {table == mkv n f;
                  result @ x == (x > n) → f x, table x;
                 };
caching_scheme == cache_gen n;
```

Here cache_gen is a cache generator function that when given n, the size of the table, returns a caching_scheme. Upon application to a function f, this caching scheme returns a memo-function with a vector table of size n bound within it. Each slot in the vector is occupied by `(f i)`. Since the language is lazy, none of these entries is initially evaluated. Upon application to an argument x (within range of vector), the corresponding entry is evaluated. This evaluation occurs only once. Any further accesses simply read off the computed value.

The time cost of accessing a value in stream cache is a linear function of the argument. For the vector cache, the cost is a constant. Getting linear performance from the fibonacci function discussed in the introduction requires that memoise be a vector cache.

Applicative caches are useful, but they do have their limitations. One problem is that they work only on arguments from enumerable domains. Mapping a domain to integers can be a difficult, sometimes even impossible task. Also, the structure of a table must be completely determined at the time the cached function is generated. It cannot depend, and hence cannot take advantage of, the order in which arguments are applied. This means that dynamic schemes such as an association list cannot be implemented. Overcoming these limitations requires that we consider another user definable memoisation technique, non-applicative caching.

5 Non Applicative Memoisation

Many caching schemes such as a simple association list, or a Binary Search Tree cannot be expressed in an applicative fashion. The table built in these schemes depends on the order in which function calls occur at runtime. Normally accomplishing this requires a rewrite of the entire program and carrying the table all over as an extra argument. However this is major surgery, we want to achieve the same while making only local changes.

It seems the only way of achieving this is to encapsulate a table as state inside a memo-function and to side-effect it in the course of program evaluation. Fine, but we lose referential transparency. Is this acceptable? How much are we willing to give to achieve the expressive power that memo-functions afford us? These questions arise as we investigate the territory between functional and imperative programming.

The minimum a user needs to specify for such a memo scheme is an initial table and a lookup_and_update function to operate on it. Though this can be made official, we prefer a slightly larger specification. We require separate definitions of the lookup and update functions. The reason is that although similar sets of proof conditions can be provided for the two specifications, the second one is more conducive to maintaining the functional discipline in an unguarded territory. This can be seen by writing a random number generator in the two schemes. Basically by removing the freedom of combining the two functions from the users hand, we reduce, or at least make difficult, the potential mischief that is possible. The following describes "memo" in Alfl enhanced with sequencing (`%force`) and setting operations (`%set`). `%force` is a sequencer that takes 2 arguments, evaluates them left to right, and returns the value of the second. `%set` is used for assignment. The formal semantics of these operations is given in appendix A. `Set` and `force` are preceded with a `%` character to emphasize the fact that these are not provided directly to the user.

```
memo [n,table,lookup,update] f ==
   @args = {result found → val,
                             %force (%set table (update table args value)) value;
           found^val == lookup table args;
           value == f args;
          };
memo n f == (intp n) & (n > 0) → lazily_memoise_fn n f,
           error ''invalid usage of memo scheme'';
```

Here *args* is a shorter syntax for $\mathbf{arg}_1\ \mathbf{arg}_2\ ..\ \mathbf{arg}_n$

As should be obvious from the above,
`lookup` :: table → *args* → Bool × value
`update` :: table → *args* → value → table

`lookup` takes a table and a sequence of arguments. It returns a value paired with a boolean flag indicating whether it is valid or not (i.e found or not). If not, then it can well be anything, a "." i.e don't care. `update` takes a table, a sequence of arguments, a value and returns an updated table.

Given the definition above, it is easy to show that if a memo scheme satisfies the following properties, it is safe i.e the cached and uncached functions behave alike.

Theorem: `f = memo [n,initial_dir,lookup,update] f` **if the following properties hold.**

Property1
(lookup init_dir *args*) = <true,value> ⇒ value = (f *args*)
                    and
(lookup init_dir *args*) $\in$ {$\perp$,<$\perp$,.>} ⇒ (f *args*) = $\perp$

Property2
(lookup (update dir *arguments* val) *args*) = <true,value> $\Rightarrow$

$$\begin{cases} (\texttt{val} = \texttt{value}) & \texttt{if } args = arguments \\ (<\texttt{true},\texttt{value}> = \texttt{lookup dir } args) & \texttt{if } args \neq arguments \end{cases}$$

and

(lookup (update dir *arguments* val) *args*) $\in \{\perp,<\perp,.>\} \Rightarrow$
   (f *args*) = $\perp$

Proof: By reasoning over the possible answers that lookup can return and by induction.

Like applicative memoisation, the name and the idea behind non-applicative memoisation is taken from Keller and Sleep [1]. However unlike applicative memoisation, there is significant divergence between the two works beyond this. We differ by disallowing the user, direct access to side effecting primitives like %force and %set. This way we are able to provide a framework of writing non-applicative caches within (extended) Alfl. We are also able to give conditions for the correctness of such a scheme, and to contain totally unrestrained imperative programming. This we believe is a better way to approach non-functional language extensions.

5.1 Examples

5.1.1 A Simple Association List

```
initial_table == [];
lookup [] arg == false^dont_care;
   '   ((argument^val)^table) arg == (arg = argument) → true^val,
                                                  lookup table arg;
update table arg val == (arg^val)^table;
```

In this example, the initial_table is an empty list. Updating a table simply requires consing an arg^val pair to the front of it. Lookup requires cdring down the list while checking if arg matches an argument in the stored pairs. If this runs off the end, not found is signalled by returning false^dont_care.

5.1.2 Binary Search Tree

We define a data type tree in this example. A tree can either be a node with the argument-value pair and its subtrees in it, or it can be a leaf. A leaf represents an empty tree or subtree, hence the value of the initial tree. We restrict the argument type to numbers for simplicity, generalizing to other types is not hard. The remaining details should now be fairly accessible.

```
data tree == node left_tree arg value right_tree | leaf;
initial_table == leaf;
lookup leaf arg == false^dont_care;
  '   (node l argument value r) arg ==
              (arg = argument) → true^value,
                     (arg < argument) → lookup l arg,
                                        lookup r arg;
update leaf arg val == node leaf arg val leaf;
  '   (node l argument value r) arg val ==
              (arg < argument) → node (update l arg val) argument value r,
                            node l argument value (update r arg val);
```

5.1.3 Hash Table - an example of cache composition

Caching schemes can be formed by composing preexisting ones. The following example illustrates how for a hash table consisting of a vector of buckets, each bucket implemented by a linked list.

```
hash_cache_gen (no_of_buckets^hash_fn) f ==
  {result @ x == g (hash_fn x) x;
   g == vector_cache no_of_buckets (@ x == memo association_list_cache f);
  };
```

```
hash_scheme == hash_cache_gen (n^hash_fn);
```

Here a vector cache and an association list cache (both described before) are composed together to form a hash scheme generator. For a given (`no_of_buckets`, `hash_fn`) pair, `Hash_cache_gen` returns a hashing scheme defined by these parameters. The basic computation in a function memoised by such a hashing scheme is as follows. The application (`g (hash_fn x)`) returns a bucket in the vector cache. The bucket represents a association-list cached version of `f`. This upon application to `x` returns the answer.

Note that any caching scheme could have been used instead of a vector cache or association lists. For instance changing to a scheme where buckets are managed like binary search trees only requires replacing `association_list` above by `binary_search_tree`.

5.2 Space reclamation

Each time an update occurs, a modified table is returned to be used in later applications. There is nothing preventing an update operation from reclaiming space while carrying out such modifications. For instance, it would be simple to implement a fixed size table based on say a FIFO scheme of removing entries. A more complex space reclamation scheme would be expressing the `purge` operator of Keller and Sleep. As defined in [1], `purge [f, x]` removes from the function

Appendix A: Semantics

The semantics given is for a simple language which can be considered as a core representative of modern functional languages, extended with side effects.

Abstract Syntax

k ∈ Con
x ∈ Id
eg ∈ Eqgrp
e ∈ Exp
c ∈ Com
pr ∈ Prog

pr ::= e
e ::= k | x | e_1 e_2 | lambda x.e | eg | %force c e | (e)
c ::= %set x e | (c)

```
eg ::= { result e ;
         x1 == e1;
         x2 == e2;
         ....
         xn == en;
       }
```

Standard Semantic Categories

Bas = Int + Bool	Basis Values
Fn = Locn × (Thunk → Store → (E × Store))	
Thunk = Store → (E × Store)	
E = Bas + Fn	Expressible Values
ρ ∈ Env = Id → Locn	Environments
σ ∈ Store = Locn → E	

Standard Semantic Functions

$\mathcal{K}$: Con → E
$\mathcal{E}$: Exp → Env → Store → (E × Store)
$\mathcal{C}$: Com → Env → Store → Store
$\mathcal{E}_p$: Prog → E

$$\mathcal{E}\,[\![\,k\,]\!]\,\rho\,\sigma = <\mathcal{K}\,[\![\,k\,]\!]\,,\sigma>$$

$$\mathcal{E}\,[\![\,x\,]\!]\,\rho\,\sigma = (\sigma\,(\rho\,[\![\,x\,]\!]))\,\sigma$$

```
$\mathcal{E}$ [[ lambda x . e ]] $\rho\ \sigma$ =
    let < loc$_1$, $\sigma_1$> = get_new_loc ($\sigma$)
    in << loc$_1$, lambda v $\sigma$ .
                    let < loc$_1$, $\sigma_1$> = get_new_loc ($\sigma$)
                    in $\mathcal{E}$ [[ e ]] $\rho$[loc$_1$/x] $\sigma_1$[v/loc$_1$]>, $\sigma_1$ >

$\mathcal{E}$ [[e$_1$ e$_2$ ]] $\rho\ \sigma$ =
    let <f, $\sigma_1$ > = $\mathcal{E}$ [[e$_1$ ]] $\rho\ \sigma$
        < t, $\sigma_2$ > = make-thunk ($\mathcal{E}$ [[ e$_2$ ]] $\rho$) $\sigma_1$
    in (fn? f) $\rightarrow$ (snd f) t $\sigma_2$, (error "non function in application position")

$\mathcal{E}$ [[ { result e ; x$_1$ == e$_1$; x$_2$ == e$_2$; ... x$_n$ == e$_n$; } ]] $\rho\ \sigma$ =
    let < loc$_1$,$\sigma_1$> = get_new_loc($\sigma$)
        < loc$_2$,$\sigma_2$> = get_new_loc($\sigma_1$)
        ....
        <loc$_n$,$\sigma_n$> = get_new_loc($\sigma_n$)
        $\rho_{new}$ = $\rho$ [loc$_1$/x$_1$, loc$_2$/x$_2$, ...,loc$_n$/x$_n$ ]
        < t$_1$, $\sigma_{n+1}$ > = make-thunk ($\mathcal{E}$ [[e$_1$ ]] $\rho_{new}$) $\sigma_n$
        < t$_2$, $\sigma_{n+2}$ > = make-thunk ($\mathcal{E}$ [[e$_2$ ]] $\rho_{new}$) $\sigma_{n+1}$
        ....
        < t$_n$, $\sigma_{2n}$ > = make-thunk ($\mathcal{E}$ [[e$_n$ ]] $\rho_{new}$) $\sigma_{2n-1}$
        $\sigma_{new}$ = $\sigma_{2n}$ [ t$_1$/loc$_1$, t$_2$/loc$_2$, ...,t$_n$/loc$_n$ ]
    in $\mathcal{E}$ [[ e ]] $\rho_{new}$ $\sigma_{new}$

$\mathcal{E}$ [[ %force c e ]] $\rho\ \sigma$ =
    let $\sigma_1$ = $\mathcal{C}$ [[ c ]] $\rho\ \sigma$
    in $\mathcal{E}$ [[ e ]] $\rho\ \sigma_1$

$\mathcal{E}$ [[ (exp) ]] = $\mathcal{E}$ [[ exp ]]

$\mathcal{C}$ [[ %set x exp ]] $\rho\ \sigma$ =
    let < t, $\sigma_1$ > = make-thunk ($\mathcal{E}$ [[ exp ]] $\rho$) $\sigma$
    in $\sigma_1$[t/($\rho$[[ x ]])]

$\mathcal{C}$ [[ (c) ]] = $\mathcal{C}$ [[ c ]]

$\mathcal{E}_p$[[ program ]] =
    fst ($\mathcal{E}$ [[ program ]] initial_env initial_$\sigma$)
```

Auxiliary Functions

```
get_new_loc :: Store → (Locn × Store)
fst ⊥ = ⊥
fst <x,y> = x
snd ⊥ = ⊥
snd <x,y> = y
make-thunk th $\sigma$ =
    let < flag, $\sigma_1$ > = get_new_loc $\sigma$
        < val, $\sigma_2$ > = get_new_loc $\sigma_1$
    in < lambda $\sigma$ . ($\sigma$ flag) → < ($\sigma$ val), $\sigma$ >,
                                        let < value, $\sigma_1$ > = (val $\sigma$)
                                        in < value, $\sigma_1$[true/flag, value/val] >,
         $\sigma_2$ [false/flag, th/val]>
```

A treatment of pairs and vectors can be done similarly. The semantic categories Pair and Vector can be defined as follows, each consisting of a sequence of values tagged by location.

Pair = Locn × (E × E)
Vector = Locn × E*

Appendix B: A Note on Lazy Memoisation

The meaning of object_identity is as follows.

```
object_identity :: E → Int
object_identity x =
    ((int? x) | (bool? x)) → (encode x),
                              (encode (location x))
```

```
location = fst                                   Returns locations of pairs, vectors and fns
encode :: Bool + Int + Locn → Int
            Each value is mapped to a unique integer code.
```

Now that object identity has been defined, we can make lazy memoisation user definable. Again the method used does not hand over opaque constructs to the user directly. Following the definition of non applicative memoisation,

```
lazymemo [n,table,lookup,update] f args ==
          {result found → val,
                         %force (%set table (update table ids value)) value;
           found^val == lookup table ids;
           value == f args;
           id_1 == object_identity arg_1;
           ...
           id_n == object_identity arg_n;
          };
error ''unable to pattern match on memo scheme'';
```

Any "safe" non applicative memoisation scheme will work correctly as a lazy memoisation scheme too. Such usage requires that the implementation maintain a consistent and unique identity of an object throughout the duration of a program, not a simple requirement.

Appendix C: Partial Glossary of Alfl terms

{...result...} A lexical block. The value of the overall expression is the value of the expression following the **result** clause. This is evaluated in an environment extended by the definitions in the current block.

== The equals sign used in definitions.

' Instead of rewriting the function name each time on the left hand side of a definition group, a tick mark (') can be used.

@ arg$_1$ .. arg$_n$ == body Corresponds to **lambda arg$_1$ .. arg$_n$ = body**

exp1 $\rightarrow$ exp2 , exp3 Corresponds to **if exp1 then exp2 else exp3**

= Equality predicate.

^ Pair forming operator (cons).

^^ Append operator.

nth i list Finds the **i**th element of **list**

hd list Returns the car or head of **list**.

tl list Returns the cdr or tail of **list**.

mkv n f Forms a vector of size **n**, 1 based, with each slot i initialized to (**f** i). A vector applied to an integer i returns the value stored in location i.

data Syntax to construct data types.

Further details on Alfl can be found in [5].

Acknowledgements

We thank Duke Briscoe, Richard Kelsey, Siau Cheng Khoo and Raman S Sundaresh for the many useful comments they made on earlier drafts of this report.

References

[1] R.M. Keller and R. Sleep, "Applicative Caching" *ACM Trans. Programming Languages and Systems*, Jan. 1986, pp. 88-108.

[2] J. Hughes, "Lazy Memo-Functions" in *Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS 201, New York, 1985, pp129-146.

[3] R. M. Keller and G. Lindstrom, "Parallelism in Functional Programming through Applicative loops", University of Utah, Salt Lake City.

[4] D. Michie, "Memo Functions and Machine Learning", *Nature*, April 1968, pp. 19-22.

[5] P. Hudak, "ALFL Reference Manual and Programmer's Guide", *Research Report* YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[6] P. Hudak, A. Bloss, and J. Young, "Code Optimization for Lazy Evaluation", *Lisp and Symbolic Computation: An International Journal*, 1988

[7] P. Hudak, "Exploring para-functional programming: Separating the *What* from the *How*", *IEEE Software*, 5(1):54-61, January 1988.

[8] P. Hudak, "Para-functional programming", *Computer*, 19(8):60-71, August 1986.

[9] L. Allison, *A practical introduction to denotational semantics*, Cambridge Computer Science Texts 23, Cambridge University Press, 1986.

[10] G. L. Steele Jr, G. J. Sussman, "The Art of the Interpreter or, The Modularity Complex (Parts Zero, One, and Two)", *AI Memo No. 453*, Artificial Intelligence Laboratory, Massachusetts Institute Of Technology, May 1978.

[11] A. J. Field, P. G. Harrison, *Functional Programming*, Chapter 19, International Computer Science Series, Addison Wesley Publishing Company, 1988.

[12] P. Hudak "The Conception, Evolution, and Application of Functional Programming Languages", To appear in *ACM Computing Surveys.*