THE IMPORTANCE OF LAZY EVALUATION IN SEARCH

Nicholas Freitag McPhee

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-06 March 1991

The importance of lazy evaluation in search

Nicholas Freitag McPhee
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
mcphee@cs.utexas.edu

March 13, 1991

Abstract

Modularity is generally considered one of the most valuable tools available to aid programmers in coping with the complexity of large tasks, and higher order functions and lazy evaluation are two very powerful tools for modularizing functional programs. Unfortunately most presentations of search algorithms in the artificial intelligence literature are not modular, at least partly because they assume a strict implementation language. In this paper I will discuss the importance of laziness in creating modular implementations of search algorithms, and show that lazy evaluation allows the algorithms and their implementations to be highly modular. In particular, laziness allows the many logically separate processes, such as the acts of building and searching a tree, to remain separate at the implementation level.

Here the search problem is treated generally, presenting numerous tools useful in general tree manipulation. This treatment demonstrates that the key difference between various search algorithms is the manner in which trees are traversed, a fact that will be illustrated through the implementation of several common search methods in Miranda.

One of the search strategies implemented is A*, which is generally described as a variant of heuristic search. With this highly modular approach, however, it becomes clear that the A* search strategy is no different from heuristic search. What defines A* is in fact the heuristic used, not the way in which which is it is applied.

1 Introduction

Modularity is generally considered one of the most valuable tools available to aid programmers in coping with the complexity of large tasks, and as John Hughes discussed so clearly in [4], higher order functions and lazy evaluation are

two very powerful tools for modularizing functional programs. Unfortunately most presentations of search algorithms in the artificial intelligence literature are not modular, at least partly because they assume a strict implementation language. In this paper I will discuss the importance of laziness in creating modular implementations of search algorithms, and show that lazy evaluation allows the algorithms and their implementations to be highly modular.

The value of higher order functions and lazy evaluation when writing search routines has been touched upon in both [4] and [2], but in both cases the emphasis was on implementing specific search routines, such as alpha-beta heuristic search. Here I will treat the problem more generally, presenting numerous tools useful in general tree manipulation. This treatment will demonstrate that the key difference between various search algorithms is the manner in which trees are traversed, a fact that will be illustrated through the implementation of several common search methods in Miranda.

2 Search algorithms

A wide range of problems can be seen as inducing a tree structure where each node is a problem state and a node's children are the states immediately reachable from the parent. For example, a game might induce a tree where each node is a state of the game (e.g., the position of the pieces on the board), and that node's children are the states resulting from each of the legal moves at that point.

Given such a tree and a description of what a goal node is, the search problem is then to find some goal node. The trees involved are typically extremely large, if not infinite, and the density of goal nodes is often low, making finding one extremely difficult. Numerous search strategies have been developed in an attempt to limit this combinatorial explosion, including heuristic search, where some heuristic is used to guide the search of the tree. In domains where effective heuristics can be developed such a search procedure can be very powerful.

As a typical example of s search algorithm (for others see any of [3, 5, 8]) consider the following heuristic search algorithm presented in [1]. Here f^* is the heuristic function estimating the distance from a node to the nearest goal node.

1. Put the start node s on a list, called OPEN, of unexpanded nodes. Calculate $f^*(s)$ and associate its value with node s.
2. If OPEN is empty, exit with failure; no solution exists.
3. Select from OPEN a node i at which f^* is minimum. If several nodes qualify, choose a goal node if there is one, and otherwise choose among them arbitrarily.
4. Remove node i from OPEN and place it on a list, called CLOSED, of expanded nodes.

5. Expand node i, creating nodes for all its successors. For every successor node j of i:

   (a) Calculate $f^*(j)$.

   (b) If j is neither in list OPEN nor in list CLOSED, then add it to OPEN, with its f^* value. Attach a pointer from j back to its predecessor i (in order to trace back a solution path once a goal node is found).

   (c) If j was already on either OPEN or CLOSED, compare the f^* value just calculated for j with the value previously associated with the node. If the new value is lower, then

      i. Substitute it for the old value.

      ii. Point j back to i instead of to its previously found predecessor.

      iii. If node j was on the CLOSED list, move it back to the OPEN.

6. Go to (2).

3 Unstated assumptions

Assuming that a general purpose implementation is intended, there is reason to suspect that the presentation of this algorithm presumes an implementation language that is both higher order and strict. The assumption that higher order functions are supported is suggested, for example, by the test whether i is a goal node in Step 4, and the expansion of i to obtain its children at the beginning of Step 6. Both of these actions require the application of some function (a goal test or an generator of children) to a node of the tree. A very direct way of making these functions available to a general search algorithm is by passing them in as arguments, which requires the language support higher order functions. It is possible to implement this algorithm in a language that does not support higher order functions by just leaving these functions undefined and letting the user define them appropriately later on. This approach, however, lacks the modularity of the higher order solution if for no other reason than the implementor of these functions must know and use the names they were given by the implementor of the search routine.

That the underlying language is assumed to be strict is a slightly more subtle point, and arises from the expansion of a node to determine its successors in Step 6. As mentioned above, logically one can think of building a tree and traversing it in search of a goal node as two distinct and separate processes. In the algorithm given, however, there is no such separation of concerns and the act of searching the tree is mingled and muddled with the act of building it.

This is a feature common to most presentations of search algorithms and is almost certainly because the implicitly assumed implementation language (in most cases LISP) is strict. In a strict language it would not be feasible to first build

the tree and then search it because even in very simple search problems the resulting trees are either extremely large or infinite. Thus the act of building the entire search tree would either take a long time and a large amount of memory (very possibly more than is available) or would never terminate. Therefore the programmer (and indirectly the algorithm designer) is forced to blend the two processes and allow the search routine to explicitly control the building of the tree.

4 Advantages of lazy evaluation

In a lazy language, however, it is possible to separate the building and the searching of the tree entirely because laziness allows the search routine to implicitly control the building of the tree. Since no part of the tree is built that is not needed, no parts are built except those searched, and they are built only when needed and discarded as soon as they no longer are relevant.

Lazy evaluation also allows another separation of concerns, namely the separation of tree traversal and the search for a goal node. Every method of tree traversal defines a method of flattening a tree, namely by generating a list of the nodes in the order that they are visited in the traversal. Once the tree has been flattened to a linear list, the act of searching for a goal node becomes a simple process of finding the first node in the list satisfying the goal predicate.

Thus I will break the problem of search into three distinct phases:

1. Build the tree.
2. Traverse (and flatten) the tree.
3. Search the resulting linear list for a goal node.

Each of these actions is logically independent and, at least in a lazy language it is possible to implement each of these phases independently. This allows the code to reflect the intuitive notion that the defining feature of a search algorithm is its method of tree traversal, with everything else being subordinate. This clear separation allows the designer of search methods to concentrate on the critical issue, namely tree traversal, without being distracted by secondary matters such as testing whether or not a given node is a goal.

5 Two simple examples

In this section I'll present two very simple examples, depth first and breadth first search, as an introduction. Later more general tools will be developed and used to implement more complex search algorithms.

All implementations will be in Miranda, and unfamiliar readers are encouraged to see [6] or [2] for more information. In fact this entire document is a runnable

Miranda 'literate' script in which all text is considered comment except those lines which begin with a '>' in the first column.

First I must define the data type for trees:

```
> tree * ::=  Empty_tree | Node * [ tree * ]
```

In Miranda this defines an algebraic data type with two cases. The first is that the tree is empty, in which case there is no other interesting information to be had. The second is where the tree in fact consists of at least one node with some label or data element of type *, where * is an arbitrary type variable, and a list of immediate subtrees generated by the node's children. These properties are captured in the following definitions:

```
> label :: tree * -> *
> label (Node x ts) = x
> label Empty_tree
>     = error "Attempt to retrieve label from an empty tree."
>
> children :: tree * -> [tree *]
> children (Node x ts) = ts
> children Empty_tree
>     = error "Attempt to retrieve children from an empty tree."
```

Of the three phases mentioned in Section 4, building the tree is entirely problem specific, so for the moment I will leave it aside and assume the tree has already been built. It is important to remember, however, that because of laziness, only those parts of the tree that will be needed will actually be constructed.

Skipping for the moment the flattening phase, the searching phase takes a list and a goal test and returns... what? There seem to be two choices: return only the first goal node found, returning some sort of error perhaps when no goal node was found, or return a list of all goal nodes. The second option seems wasteful since one usually cares only about the first goal node. There are, however, circumstances where having several goal states, or options, available is desirable. Once again laziness makes this multiple goal solution feasible, because no more goal states will actually be found than are called for. Therefore if we only need one solution, we can take the head of the solution list, and lazy evaluation ensures that no others will be computed. Thus we have the flexibility of access to all goal nodes while only paying for the ones we use. This decision is reflected in the following data types and definition of the function `search`:

```
> search_method * == tree * -> goal_test * -> [*]
> goal_test * == * -> bool
>
> search :: [*] -> goal_test * -> [*]
> search = converse filter
```

Finally we come to the flattening phase, where the type `flattener` is defined, and simple depth first and breadth first flattening are implemented directly:

```
> flattener * == tree * -> [*]
>
> simple_depth_flatten :: flattener *
> simple_depth_flatten (Node x ts)
>       = x:(concat (map simple_depth_flatten ts))
> simple_depth_flatten Empty_tree = []
>
> simple_breadth_flatten :: flattener *
> simple_breadth_flatten tree
>       = do_sbf [tree]
>         where
>         do_sbf ((Node x children):ts)
>             = x:(do_sbf (ts ++ children))
>         do_sbf (Empty_tree:ts) = do_sbf ts
>         do_sbf [] = []
```

These definitions are very similar to those one might see in other functional languages, although once again in strict languages these definitions would have to be altered to be of use on large or infinite trees.

Now defining simple depth first and breadth first search routines is straightforward:

```
> simple_depth_first_search :: search_method *
> simple_depth_first_search = search . simple_depth_flatten
>
> simple_breadth_first_search :: search_method *
> simple_breadth_first_search = search . simple_breadth_flatten
```

6 Generalized flatteners

Having decided that the method of flattening the tree is the critical element in a search problem, and having shown how lazy evaluation allows us to isolate the

flattening process, can we generalize these two simple flatteners? Both flatteners keep a list of nodes still to be expanded (often called the open list). This list is kept implicitly in the depth first case via the call stack, and explicitly in the breadth first case as the argument to `do_sbf`. In both cases the flatten process starts with just the root node on the open list, and it ends when the open list is empty. Also, the next node to explore is always the head of the open list in both methods. The only difference between the two is that in the depth first case a current node's children are added the beginning of the open list and in the breadth first case they are added to the end.

Factoring out these common features would yield a generalization that includes a significant number of traversal methods, but there would also be methods of interest not within this paradigm. An example is the algorithm presented in Section 2 which makes use of a list of nodes already explored (often called a closed list). So instead let us generalize further and assume that a flattener has internal state of some arbitrary type, and that a method of flattening trees is defined by this state and four operations:

1. Initializing the state of the flattener.
2. Determining from the current state when the flattening process is completed.
3. Retrieving the next node to explore from the current state.
4. Incorporating a list of nodes (the children of the current node) into the current state to yield a new state.

Each of these operations can be implemented as a function that will be passed as an argument to a general function that generates a flattening method from these parameters. If the type of the tree being flattened is `tree *`, and the type of the flattener's internal state is `**`, then the types of these operations are:

1. `gen_start_state :: tree * -> **`
2. `is_empty_state :: ** -> bool`
3. `next_node :: ** -> tree *`
4. `insertion_method :: insertion_method * **`

where an insertion method has type:

```
> insertion_method * ** == ** -> [tree *] -> **
```

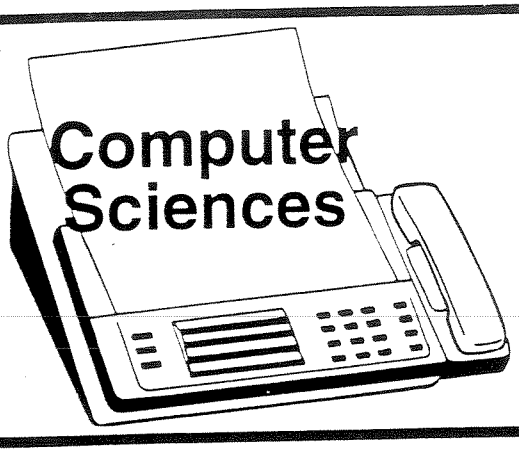
Now one can define a general function to generate flatteners by:

```
> generate_flattener
>       :: (tree * -> **) -> (** -> bool) -> (** -> tree *) ->
>            insertion_method * ** -> flattener *
> generate_flattener gen_start_state is_empty_state
>                    next_node insertion_method
>       = map (label . next_node) .
>            takewhile ((~) . is_empty_state) .
>            iterate (next_state insertion_method next_node) .
>            gen_start_state
>
> next_state insertion_method next_node state
>       = insertion_method state (children (next_node state))
```

As defined here flattening a tree consists of four separate processes. First the tree to be flattened is converted into an initial state for the flattener. Then `iterate` is used to create the (possibly infinite) list of states the flattener passes through while traversing the tree. Then the longest prefix of this list consisting entirely of non-empty states is obtained using `takewhile`. Each state in the resulting list then corresponds to a specific node, namely the next node to be explored at that point, so extracting from each state first the node, and then the label, yields the desired flat list of labels. The fact that these four processes are logically separate is captured nicely through the composition of four independent functions generated from the higher order arguments.

Having presented `generate_flattener` as a general function for creating flatteners, can it be used, for example, to generate depth and breadth first flatteners? It can, but before we do so let us remember that these methods of flattening shared a common internal state, namely that of an open list, and in fact they differed only in their insertion methods. So first let us use `generate_flattener` to define an intermediate class of flatteners, namely those whose state is just an open list, which we can further specify class to obtain depth and breadth first flatten.

```
> gen_open_list_flattener
>       :: insertion_method * [tree *] -> flattener *
> gen_open_list_flattener
>       = (generate_flattener (:[]) (=[]) hd) . ( . tl)
>
> depth_flatten = gen_open_list_flattener (converse (++))
> breadth_flatten = gen_open_list_flattener (++)
>
> depth_first_search = search . depth_flatten
```



Department of Computer Sciences
The University of Texas at Austin
Taylor Hall 2.124
Austin, TX 78712-1188
(512) 471-7316
FAX (512) 471-8885

FAX COVER SHEET

To: School of Cog. Sciences Hampshire Coll.

Attn: Roger Bellin

From: Diane Dunkelberger

Number of Pages to follow : 17

☒Notes: make ck to. U of Texas at Austin
Address is on web homepage.

D—

9595

If you have trouble receiving this FAX, please call (512) 471-7316

```
> breadth_first_search = search . breadth_flatten
```

These definitions make the relationship between these two methods of tree traversal remarkably clear, especially when compared to the earlier "traditional" definitions of `simple_depth_flatten` and `simple_breadth_flatten`.

7 Heuristic search

While it is nice that `generate_flattener` can be used to generate both depth first and breadth first flattening routines, neither strict depth first or breadth first search is very useful in general. The former has the advantage of requiring little memory, but has a tendency to get lost searching unproductive paths. Breadth first search, on the other hand, is guaranteed to find a solution if one exists, but can require an enormous amount of memory to maintain its open list. This is the approach taken in the heuristic search algorithm presented in Section 2.

A much more fruitful approach in domains where a reasonable heuristic can be found is heuristic search. In this approach a heuristic is used to estimate the distance from the current node to the nearest goal node, and at each point the search process explores the node on the open list with the lowest heuristic value. This can be done using just an open list as was done for the previous two methods, but it is common to also maintain a closed list containing those nodes already visited to avoid repeatedly visiting nodes with the same label.

Just as we were able to define a subclass of flatteners whose state was an open list, we are also able to define a subclass of flatteners whose state consists of both an open and a closed list:

```
> op_cl_state * == ([tree *], [tree *])
>
> op_cl_flatten
>       :: insertion_method * (op_cl_state *) -> flattener *
> op_cl_flatten
>       = (generate_flattener gen_start is_empty next) .
>             ( . move_to_closed)
>         where
>         gen_start tree = ([tree], [])
>         is_empty (open_list, closed_list) = (open_list = [])
>         next (open_list, closed_list) = hd open_list
>         move_to_closed (open, closed)
>               = (tl open, hd open:closed)
```

All that remains to define heuristic flatten, and thus heuristic search, is to define a heuristic insertion procedure. What properties must this insertion procedure have? Since `op_cl_flatten` assumes the next node to be explored is the head of the open list, this new insertion procedure must maintain an open list ordered in increasing heuristic value. For simplicity it also seems reasonable to require that there not be two nodes with the same label in the union of the open and closed lists. Lastly, if a new node is found with the same label as one already on the open or closed list, but with a lower heuristic value, then the old node should be removed from the appropriate list and the new one added to the open list.

It is clear that computing heuristic values of tree nodes has nothing intrinsic to do with the act of searching a tree, and we would like to be able to separate these logically distinct processes in our implementation. Presentations of heuristic search algorithms, such as the one in Section 2, mix these actions, calling for the heuristic values to be computed as part of the search process. In a strict language this is necessary because strict evaluation makes it impossible to compute in advance the heuristic values of only those nodes that will be explored. Lazy evaluation again gives us an advantage and allows us to keep these issues separate because laziness guarantees to only compute the heuristic values of those nodes that are actually explored.

So let us assume that the heuristic values of the tree's nodes have already been computed and adjoined to the labels. Thus, in heuristic flatten, node labels are going to contain at least two pieces of information: the original label and that node's heuristic value. We could just assume that these labels will have some fixed type, but it is more general to pass our heuristic insertion function two functions that allow it to access the the necessary components of the label. This allows us to use heuristic search on trees with more complex label structure without any modification.

In light of these decisions, it is now possible to define `heuristic_search` using the following definitions of `heuristic_insert` and `heuristic_flatten`.

```
> heuristic_search label_key heur_key
>       = search . (heuristic_flatten label_key heur_key)
>
> heuristic_flatten label_key heur_key
>       = op_cl_flatten (heuristic_insert label_key heur_key)
>
> heuristic_insert label_key heur_key
>       = foldr insert_node
>         where
>         ek = label_key . label
>         hk = heur_key . label
>         insert_node node (open, closed)
```

```
>             = (add_node hk node new_open, closed),
>                   if on_open & node_val < open_val
>             = (open, add_node hk node new_closed),
>                   if on_closed & node_val < closed_val
>             = (open, closed), if on_open \/ on_closed
>             = (add_node hk node open, closed), otherwise
>               where
>               node_val = hk node
>               (on_open, open_val, new_open)
>                   = find_occurrence ek hk node open
>               (on_closed, closed_val, new_closed)
>                   = find_occurrence ek hk node closed
>
> add_node heur_key node list
>       = takewhile (( <= heur_key node ) . heur_key) list
>             ++ [node] ++
>             dropwhile (( <= heur_key node ) . heur_key) list
>
> find_occurrence label_key heur_key node list
>       = (in_list, val, new_list)
>         where
>         matches = filter (( = label_key node ) . label_key) list
>         in_list = (matches ~= [])
>         val = heur_key (hd matches)
>         new_list = list -- matches
```

8 A*

A* is specific kind of heuristic search in which two different heuristic functions are employed: one to estimate the cost of getting from the start state to the current state, and the other to estimate the cost of getting from the current state to the nearest goal state. If these heuristics satisfy certain monotonicity properties, then A* is guaranteed to find an optimal goal state (if one exists) in the sense that no other goal state can be reached more cheaply from the start state.

Since A* is a kind of heuristic search, it is usually suggested that it can be defined through some slight modification of heuristic search, namely by changing the way in which the heuristic values of the nodes are computed as they are encountered. As mentioned above, though, lazy evaluation allows us to separate the processes of computing heuristic values and searching the tree, so using our definitions we are able to use `heuristic_search` *without modification* to implement A*. All that is necessary is for us to define functions that compute the

A* heuristic values of the nodes of a tree, and then pass this new tree and the necessary keys to `heuristic_search`.

8.1 Some general tree processing tools

Before I define these functions to compute the A* heuristic values, however, it will be useful to define a number of tree processing tools similar to the list processing tools provided in Miranda.

```
> tree_map :: (* -> **) -> tree * -> tree **
> tree_map f (Node x ts) = Node (f x) (map (tree_map f) ts)
> tree_map f Empty_tree = Empty_tree
>
> tree_scanl :: (* -> ** -> **) -> ** -> tree * -> tree **
> tree_scanl op k (Node x ts)
>       = Node k (map (tree_scanl op (op x k)) ts)
> tree_scanl op k Empty_tree = Node k []
>
> tree_zip2 (Node x ts) (Node y us)
>       = Node (x,y) [ tree_zip2 t u | (t, u) <- zip2 ts us ]
>
> prune :: (* -> bool) -> tree * -> tree *
> prune test (Node x ts)
>       = Node x (filter (~= Empty_tree) (map (prune test) ts)),
>                if test x
>       = Empty_tree, otherwise
> prune test Empty_tree = Empty_tree
```

Each of these functions does roughly the same thing to trees that their list counterparts do to lists (with the counterpart of `prune` being `takewhile`), and in fact nice laws exist relating these functions to their list counterparts:

1. `map f (flatten tree) = flatten (tree_map f tree)`, where `flatten` is some arbitrary tree flattener.
2. `scanl f k (path tree) = path (tree_scanl f k tree)`, where `path` is any function that returns a connected path from the root of the tree, and `scanl` is defined as in [7], namely `scanl = scan . converse`.
3. `zip2 (path tree1) (path tree2) = path (tree_zip2 tree1 tree2)`, where again `path` is a function that returns a connected path from the root.
4. `takewhile test (path tree) = path (prune test tree)`, where `path` is as above.

As an aside, some might notice that no counterpart to the fold functions is given. Several authors have defined tree folding functions (e.g., [2]), but these definitions often differ and there's no clear notion of which is the "right" definition and why. I believe the problem arises because folding depends crucially upon the order in which the arguments are folded, and in the case of lists there are only two natural orders: left to right (corresponding to `foldr`), and right to left (corresponding to `foldl`). In the case of trees, however, there is no such simple space of choices. In fact each different way of flattening the tree represents a different possible folding of the tree. This suggests that one way to fold a tree is to first flatten it in some way and then apply one of the list folding functions to the result, and upon inspection many definitions of tree folding correspond to folding either depth first or breadth first flattening the tree. This combination of flattening and list folding doesn't capture all the possibilities, though. For example, the definition of fold over binary trees given in [2] is not equivalent to any combination of a flattener and a list folding function without the additional assumption that the function being folded in is associative.

Given the tree utility functions defined above, several other useful utilities can be defined as well:

```
> add_value :: (* -> **) -> tree * -> tree (*, **)
> add_value f
>       = tree_map op
>         where
>         op x = (x, f x)
>
> tree_of_ancestors :: tree * -> tree [*]
> tree_of_ancestors = tree_scanl (:) []
>
> add_ancestors :: tree * -> tree (*, [*])
> add_ancestors tree = tree_zip2 tree (tree_of_ancestors tree)
```

These last two allow one to incorporate into a node's label a list of its ancestors, which is useful if you need to know not only that there is a goal node, but what path leads to it as well. Note that once again laziness allows us to keep this process separate from the search process, whereas they would have to be mingled in a strict implementation (see the algorithm in Section 2).

8.2 Implementation of A*

Now it is time to define A* search. As mentioned before, all that's necessary is to compute the value of the A* heuristic for each node and adjoin that value to the node. Once this is accomplished the tree can be searched merely by calling

heuristic_search with the appropriate keys. For the sake of illustration let us also assume that both the goal node and the path leading to it are desired, as is often the case. In these definitions the function f_heur is the heuristic that estimates a node's distance from the neartest goal. The function g_heur computes the cost of a given segment of the path.

```
> add_pair (x, y) = x + y
>
> a_star_state * == ((*, [*]), num)
>
> label_key :: a_star_state * -> *
> label_key ((label, parents), h_val) = label
>
> parents_key :: a_star_state * -> [*]
> parents_key ((label, parents), h_val) = parents
>
> heur_key :: a_star_state * -> num
> heur_key ((label, parents), h_val) = h_val
>
> add_a_star_heuristic
>      :: (* -> **) -> (** -> num) -> (** -> num) ->
>           tree * -> tree (*, num)
> add_a_star_heuristic key f_heur g_heur tree
>       = tree_zip2 tree
>             (tree_map add_pair
>                   (tree_zip2 (tree_map (f_heur . key) tree)
>                              (tree_scanl ((+) . g_heur . key)
>                                    0 tree)))
>
> prep_tree_for_a_star
>       :: (* -> num) -> (* -> num) -> tree * ->
>                tree (a_star_state *)
> prep_tree_for_a_star f_heur g_heur
>       = (add_a_star_heuristic key f_heur g_heur) .
>              add_ancestors
>         where
>         key (x, parents) = x
>
> a_star_search f_heur g_heur
>       = (heuristic_search label_key heur_key) .
>             (prep_tree_for_a_star f_heur g_heur)
```

9 Conclusions

Throughout the implementations of the various search routines there have been numerous instances where laziness has allowed us to keep logically independent processes separate in the implementation as well. In doing so lazy evaluation has allowed modularization far beyond that generally found in the search literature, and far beyond what is even possible with a strict language.

A common way around such language limitations is the design of software generation tools, and it would no doubt be possible to build such a tool that would take the specification of a search problem and generate code in some strict functional, or even imperative, language to handle that specific problem. There's no question, however, that being able to solve the problem directly using the kind of highly modular code demonstrated here is a far better solution. We have far too many tools at hand that allow us to gush forth code until we become buried in its complexity, and tools that help us reduce the amount of code are most certainly to be prized. In lazy evaluation we have a tool that allows us to write genuinely modular code so that we may reuse functions over and over again in a wide variety of contexts, and in doing so, laziness helps keep a lid on code proliferation. And the less code there is, the easier it is to understand and verify, modify and maintain.

References

[1] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume 1. William Kaufmann, Inc., 1981.

[2] Richard Bird and Philip Wadler. *Introduction to functional programming.* Prentice Hall, 1988.

[3] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley Publishing Company, 1985.

[4] John Hughes. Why functional programming matters. In D. A. Turner, editor, *Research topics in functional programming.* Addison-Wesley Publishing Company, 1990.

[5] Elaine Rich. *Artificial Intelligence.* McGraw-Hill Book Company, 1983.

[6] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings IFIP Conference on functional languages and computer architecture.* Springer-Verlag, 1985.

[7] D. A. Turner. Duality and de morgan principles for lists. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: A birthday salute to Edsger W. Dijkstra.* Springer-Verlag, 1990.

[8] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley Publishing Company, second edition, 1984.