ACM Lisp Pointers **V**,2 (April-June 1992), 11-19.

The Buried Binding and Dead[1] Binding Problems of Lisp 1.5: Sources of Incomparability in Garbage Collector Measurements

Henry G. Baker

June, 1976[2]

Laboratory for Computer Science[3]
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

This research was supported by the Advanced Research Projects Agency of the Department of Defense, and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Abstract

Lisp has become the language of choice for many applications such as artificial intelligence programs or symbol manipulation. The original implementation of Lisp 1.5 was a concise, elegant statement of the semantics of the language. Although production Lisp systems have undergone significant development and evolution since Lisp 1.5, including the development of sophisticated compilers, there have been few significant theoretical improvements in the implementations of these systems. Most improvements, such as arrays or shallow-binding, have been made more for the sake of speed than for the sake of storage. A notable exception to this is the technique of tail recursion, which can save more than just stack space.

We believe that more can be done to reduce the storage requirements of Lisp programs. Although in many instances, the Lisp programmer can reduce the storage requirements of his program by deleting unneeded pointers as soon as possible, there is nothing he can do about systematic inefficiencies of the Lisp interpreter. This paper addresses itself to two sources of inefficiency in Lisp's variable binding mechanism—one of which is easy to detect—which prevent storage from being garbage collected long after its last reference. Implementations of Lisp which eliminate these situations should result in more economical execution for almost all Lisp programs which use a lot of storage, due to a lighter marking load on the garbage collector.

Introduction

Much work has been done to optimize the interpretation and compilation of Lisp programs. However, most of the work has involved the minimization of running time (exclusive of garbage collection time) and the minimization of the number of CONS'es. Let us define the *running time* of a Lisp program to be the execution time exclusive of garbage collection, *garbage collect time* to be the time spent garbage collecting, and *running space* to be the number of CONS'es. Each of these quantities refer to the amount of time or number of CONS'es since the program started. Let us define the *net space* at a point in a program to be the number of cells marked by the garbage collector if it were run at that point in the program. When discussing the total time or space taken by a Lisp program, we mean the running time, garbage collect time, or running space at the termination of the program; *maximal net space* will refer to the net space maximized over the life of the program, i.e., the "high water mark".

This paper discusses ways to reduce the net space of a program and hence its maximal net space. This is important because the garbage collect mark phase time for a single garbage collection is

---

[1] "Stale" was used in the original paper, but "dead" is more consistent with modern usage [Aho86].

[2] The date is correct; this manuscript was never published due to the pressure of the author's thesis work. The present version has been formatted and edited, but not revised, except where noted.

[3] Author's current address: *Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436, (818) 986-1436, FAX: (818) 986-1360.*

proportional to the net space at that point in the program. Reducing the average net space of a program will reduce the total garbage collect time; reducing the maximal net space will allow larger programs to be run in a limited address space.

We assume a paradigm Lisp interpreter similar to that found in the appendix of [McCarthy65] for the purposes of discussion. This is because most Lisp interpreters are virtually isomorphic to this one (with the possible exception of shallow-binding, which does not affect our arguments), and this interpreter is already familiar to more people than any other.

Environments and the Buried Binding Problem

An environment is intuitively an object in which the value of a variable can be looked up. In order to be more precise, however, we must enumerate all the functions which can operate on environments. The function `(lookup x e)` produces the current value of the variable *x* in the environment *e*, if there is one, and `UNDEFINED`, otherwise. There is a unique environment, called `NULL-ENVIRONMENT`, which produces `UNDEFINED` for every variable. New environments are created from old ones by the function `(bind x y e)`, which creates an environment *e'* in which the variable *x* is bound to the value *y* and the values of all other variables are inherited from *e*. The value of the variable *x* in an environment *e* can be changed to *z* by the function `(rebind x z e)`. `rebind` returns the "same" environment it was given, except that `(lookup x e)` will now produce *z* instead of whatever it produced before. (Models differ on whether rebinding a variable *x* in an environment *e* can affect the value of *x* in any other environment or not.) Environments can be compared by `(equal` e_1 e_2`)`; e_1 and e_2 are equal just when `(eq (lookup` *x* e_1`) (lookup` *x* e_2`))` is `t` for all variables *x*.

Implementing Environments—The Tree Representation

A non-null environment *e* can be represented by a pair <*bind*[*e*], *parent*[*e*]>, called a *node*, where *bind*[*e*]=*p* is a *binding post*, i.e. a pair <*variable*[*p*], *value*[*p*]>, in which *value*[*p*] can be updated. The above structure is created by the function call `(bind` *variable*[*e*] *value*[*e*] *parent*[*e*]`)`. (Here, *variable*, *value*, *bind*, and *parent* are meta functions for decomposing bindings and environments and are not available to the programmer.)

`lookup` and `rebind` can be defined relatively easily:

```
(lookup x e) =    let e'=search[x, e] in
                    if e'=NULL-ENVIRONMENT then UNDEFINED
                    else value[bind[e']]

(rebind x z e) =  let e'=search[x, e] in
                    if e'=NULL-ENVIRONMENT then ERROR
                    else value[bind[e']]:=z;
                  e

search[x, e] =    if e=NULL-ENVIRONMENT then e
                    else if x=variable[bind[e]] then e
                    else search[x, parent[e]]
```

If an environment *e'* can be reached from an environment *e* by following a finite series of *parent* pointers, then we say that *e'* is an *ancestor* of *e* and that *e* is a *descendant* of *e'*. We make the reasonable assumption that the null environment is an ancestor of every environment (excepting the null environment, itself). This constraint is automatically satisfied by any finite program, given only our primitives for interacting with environments. With these assumptions, a group of environments forms a *tree* having the null environment as its root. The unique path from an environment to the root we will call its *search path*.

We have defined the tree representation for environments; we now investigate its properties. First of all, `lookup` and `rebind` terminate for every environment due to the null-environment-ancestor condition. Secondly, `lookup` and `rebind` find only the first occurrence of a variable in the search path from a particular node; any other binding of that variable in an ancestor environment will not be seen. Therefore, the tree representation could lead to the "buried binding problem": a binding of a variable `x` in a node `e` cannot be referenced because 1) the only direct pointers to `e` are parent pointers from other environments; and 2) all paths of pointers to `e` from outside the environment tree go through environment nodes in which `x` is previously bound. Thus, `search` can never be called with the arguments `[x, e]`.

Buried bindings can be a problem if space is at a premium because the garbage collector will refuse to collect the value of the buried variable even though it cannot be referenced by `lookup` or `rebind`. This is because the garbage collector cannot distinguish between environment and other structures and it must assume that the parent pointer chains can be followed. There are two possible solutions to this problem: modify the garbage collector, or change the representation. We will investigate both of these possibilities.

A natural question arises before we delve into the details of possible modifications. Do buried bindings actually occur in real programming situations, such as in Lisp? The answer is paradoxical in that Lisp interpreters[4] which have been optimized in certain ways[5] exhibit more buried bindings than unoptimized interpreters!

Modifying the Garbage Collector

Suppose that we wish to modify the garbage collector to reclaim buried bindings. What must be changed? First, the buried bindings must be identified during the mark phase. Then, buried bindings must be deleted or spliced out of the environment during the collect phase.

In order to identify buried bindings in the mark phase, the garbage collector must know when it is marking an environment node, and give it special treatment. Upon encountering an unmarked environment node `e` by chasing other than a *parent* pointer, the garbage collector must first run along its parent path from the node to the root, accumulating a *defined variable set* for `e` which indicates which variables occur in the path. If the garbage collector tries to add a variable which is already in the defined variable set, it knows that the binding is buried as far as the environment `e` is concerned. However, it might be accessible through some other pointer. Therefore, the best thing the garbage collector can do is mark the bindings which are certainly accessible. If at the end of the mark phase of garbage collection, a binding is still not marked, it is either totally inaccessible (would not be marked by an unmodified garbage collector) or it represents a buried binding and should be reclaimed.

In order to accumulate the defined variable set for an environment during the scanning of the parent path for an environment `e`, one bit per atom (variable) is allocated which is normally in the *off* state. As each binding is encountered on the scan, the variable's bit is tested. If it is *off*, the variable is not yet in the set and must be added. The bit is set to *on*, that binding is marked as accessible, and its *value* is put onto the stack for further marking. If, on the other hand, the bit was *on*, then nothing is done to that binding. In either case, the scan then proceeds with the *parent* of the current node. When the root is reached, a variable's bit will be *on* if and only if it is in the defined variable set for `e`. However, since the bits must all be *off* before the next environment node is marked, we must make one more pass and turn all of the bits *off*. Finally, the environment node `e` itself is marked.

This scheme is not as efficient as one might hope because environments close to the root will be scanned many times in the course of garbage collection, whereas in normal garbage collection, a cell can be marked (and its pointers followed) only once. It remains to be seen if more clever methods of garbage collection can solve this problem.

---

[4]Interpreted code is easier to analyze for storage utilization, but compiled code has analogous problems.

[5]E.g., through *closure-sharing* [Steele78].

Buried bindings, once identified, can be either nullified or spliced out of the environment tree. Nullifying the binding at node `e` means simply calling `(rebind` *variable*`[e] nil e)`; i.e., rebinding the variable at `e` to something harmless, so that the old value can be collected. This method does not require another phase in the garbage collector but does have the disadvantage that the storage used by the node `e` itself cannot be reclaimed. However, `lookup` could be modified to splice out nullified bindings whenever they are encountered, so that they can be picked up during the next garbage collection. Thus, this additional storage use is only temporary.

Splicing out buried bindings during the garbage collection in which they are identified might seem to require another mark-type phase. However, Guy Steele [Steele76] has suggested a clever method whereby the buried bindings can be spliced out during the collect phase. Every marked environment node encountered is checked to see if its parent is unmarked. If it is, the parent pointer is reset to the parent's parent. That node itself is checked for a marking, and so on, until the parent pointer points to a marked environment or `nil`. The collector then goes onto the next node in address order. If the node is unmarked, it is put onto the free node list. This simple scheme will work, provided only that the free list link cell is not the same as the *parent* cell in the environment node. This is because nodes may be placed onto the free list before their parent pointers have been used and we wouldn't want the parent pointer to be clobbered prematurely. Since this state of affairs can easily be arranged (e.g., link the free lists by `CAR`'s in our prototype), this method is very nearly as efficient as a normal collect phase.

The Functional Representation for an Environment[6]

The tree representation for environments can be thought of as a *shared* representation in the sense that a particular environment node stores only one binding; it gets all other bindings from another environment. Another possibility would be a *non-shared* representation, one in which each environment stores all the bindings defined for that environment.

We now present such a non-shared environment representation which we call the *functional representation*. In this representation, an environment is simply a *set* of binding posts with no more than one binding post per variable. In other words, an environment is a set of ordered pairs which forms a single-valued relation, i.e., a *function*. The definitions of each of the primitives dealing with environments is trivial. `(lookup x e)` finds the binding post for `x` in `e` and returns the associated value, or `UNDEFINED`, if none exists. `NULL-ENVIRONMENT` is the empty set. `(rebind x z e)` finds the binding post for `x` in `e` and replaces the associated value with `z`. A big change between the representations is in `bind`. `(bind x y e) = e'` has two cases: 1) if `x` is undefined in `e`, then `e'` is $e \cup \langle x,y \rangle$; 2) if `x` is bound to `w` in `e`, then `e'` is $(e - \langle x,w \rangle) \cup \langle x,y \rangle$. Thus, `bind` preserves the functionality of environments.

There are several possible implementations of the functional representation. The set of binding posts can be stored as a list, an array, or something more complex like a binary search tree or trie. It seems that all of these representations require $O(|S|)$ time and space to perform `bind`, while for some of them, `lookup` or `rebind` can be done in time $O(\log|S|)$, where $|S|$ is the cardinality of the set. For certain restricted sets of variables, the time for `lookup` is $O(1)$, i.e., constant. This is to be contrasted with the tree representation, where `bind` requires constant time and space, but `lookup` can require time much greater than $O(|S|)$! This is because some trees bind the same variable many times in a search path, which slows down accesses to other variables.

This discussion must also be tempered by the knowledge that the tree representation can be augmented with *value cells* to give a *shallow-binding representation* [Baker76] which can reduce `lookup` time to a constant at the expense of yet another time—*context switching time*, which is constant for either the unadorned tree representation or the functional representation. We note that the tree representation has a buried binding problem if and only if the shallow-binding tree representation has it; therefore, we will not discuss shallow-binding any further.

---

[6]A modern term for this representation is "acquaintance list/vector", after its use in Hewitt's *Actor* model [Hewitt77].

The functional representation cannot exhibit the buried binding problem because any environment binds a variable to only one value. However, it has serious efficiency problems. First, it may not be any faster to access than the tree representation. Second, creating new environments is no longer a trivial operation, as it is in the tree representation. Finally, the storage for the sets themselves is not insignificant, and since it is not shared, as it is in the tree representation, it may take up more room in some cases than the tree representation.

The $64,000[7] question is: can a representation for environments be found which has good creation and access times and which solves the buried-binding problem?

Hacking Environments

Any solution to the buried binding problem depends upon the programmer adhering to certain constraints in accessing and modifying environments. For example, some "a-list" Lisp systems give the programmer access to the current environment and allow him to apply `CDR` to that environment to get its parent environment. This kind of hacking can expose buried bindings and prohibits any attempt at there elimination.

To insure the inviolability of environments, we advocate the creation of a new Lisp datatype, the *environment*. The Lisp programmer would have access to environments only through the functions: `null-environment`, `bind`, `rebind`, and `eval`. `(null-environment)` would produce an environment with no bindings; `(bind x y e)` would produce a new environment which had the same bindings as *e* except that the atom *x* would be bound to *y*; `(rebind x z e)` would change the binding of *x* in the environment *e* to the value *z* using a side-effect, and would return the changed environment *e*; `(eval x)` would evaluate the expression *x* in the current environment and `(eval x e)` would evaluate the expression *x* in the environment *e*.

Dead[8] Bindings

A binding of a variable to a value at a point in the interpretation of a program is said to be *dead* if it will never be referenced again, yet it would not be reclaimed by a normal garbage collection at that point. By this definition, we can see that buried bindings are also dead bindings. There are many other instances of dead bindings, however.

A trivial example of a dead binding is that of a subroutine having one parameter which never references it. In this case, the binding of the parameter to the argument need never occur, since it would be immediately dead. A less trivial case would be a subroutine of one parameter, which parameter was used to determine one of a set of alternative bodies for that subroutine. If none of the alternative bodies referenced the parameter, the parameter binding would become dead immediately after the choice of alternatives had been made. Dead bindings tie up a lot of space by holding onto storage that could otherwise be reclaimed by the garbage collector. If the bindings were nullified or spliced out as soon as they were no longer needed, rather than at the end of interpretation of the form in which they were bound, the space savings could be significant.

The primary producer of dead bindings, however, is the *functional value* or *upward funarg*. A functional value is a pair consisting of a lambda-expression and an environment. Functional values are created by evaluating a `(function … )` form in Lisp. The environment of the functional values is the environment at the time the form was evaluated. Intuitively, this environment is to be used for finding the values of the free variables in the lambda-expression. Usually, however, there are only a few free variables, whereas the environment structure can define a large number.

The more trivial examples of dead bindings discussed earlier do not cause as much trouble as functional values because the binding usually becomes dead only a short time before it is released,

---

[7]From a TV quiz show which was popular when circuits were still segregated.

[8]"Stale" was the term which appeared in the original paper; "dead" is more consistent with modern usage [Aho86].

at subroutine return time, when it can be garbage-collected. Functional values, on the other hand, can be created at large return-stack depths, yet can be passed back all the way to be invoked at a very shallow stack depth. In this way, extremely large environment structures can be in effect at very shallow return point stack depths. In most cases, only a very small part of these large environment structures will ever be referenced.

A worst-case situation for dead bindings produced by functional values can be imagined. It is well known that Lisp's CONS cells can be exactly simulated by functional values; actually doing so in Lisp 1.5 will cause any by the most trivial program to quickly run out of storage. This is not because the number of Lisp cells used to implement functional values is excessive; in fact, the simulation below requires just 7 CONS cells (in our paradigm interpreter, anyway) for one simulated CONS cell. What is wrong is that the functional values keep pointers not only to the simulated cell's CAR and CDR, but also to the value of every other variable which existed in the context in which the cell was created! It is this storage inefficiency caused by dead bindings rather than the constant factor inefficiency which makes functional values so useless in Lisp that some Lisp's do not even support them.

```
(defun cons (x y)
  (function
   (lambda (fn new me)
     (cond ((eq fn 'car) x)
           ((eq fn 'cdr) y)
           ((eq fn 'rplaca) (prog2 (setq x new) me))
           ((eq fn 'rplacd) (prog2 (setq y new) me))))))

(defun car (x) (x 'car nil x))

(defun cdr (x) (x 'cdr nil x))

(defun rplaca (x y) (x 'rplaca y x))

(defun rplacd (x y) (x 'rplacd y x))
```

Simulation of Lisp's CONS cell by a FUNARG

The difficulty with dead bindings is in realizing when they have become dead. In lexically-scoped languages like Algol or PL/I, the free/bound status of a variable in an expression gives a good fail-safe method for proving that a variable binding is dead. If there are no more free occurrences of that variable in the rest of the expression being evaluated, any binding of that variable in the current environment is dead. On the other hand, a variable binding may be dead and yet the variable may still occur free in the expression—e.g., within a conditional arm that will not be executed.

The whole concept of "free/bound" loses its meaning in a dynamically-scoped language like Lisp, because the scope of a variable binding changes with every different invocation of a subroutine. No purely syntactic test can determine whether a variable is in the scope of a binding or not.

In general, the deadness of a binding at a point in the interpretation of an expression is undecidable, regardless of whether lexical or dynamic scoping is used. Therefore, heuristics have to be developed which eliminate as many dead bindings as possible (without, of course, eliminating non-dead ones!).

Tail Recursion

Tail recursion is the name given to a programming technique which is used to reduce the size of the return-point stack and marginally decrease execution time in Lisp-type interpreters. Tail recursion works by noticing that the machine language sequence "`push` *e*`, pc and jump to` *x*; `pop` *e*`, pc`", where *e* is the current environment pointer and `pc` is the current program counter, can be replaced by the single instruction "`jump to` *x*", for all programs which satisfy stack (LIFO) discipline. The idea is that if the last thing a subroutine A does is to call subroutine B, then simply jumping to B will make B return directly to A's caller when it returns instead of back to A. Stack space is saved by not having to push the pair <A's environment, return address to A> onto the stack.

The most dramatic use of tail recursion is in certain recursive programs which call themselves as the last part of their body. In this case, no return-point stack space is used at all—the program *iterates* instead of recursing.

Consider the following trivial Lisp 1.5 program:

```
((label[9] f
   (lambda (x)
     (cond ((zerop x) x)
           (t (f (sub1 x))))))
 n)
```

This program recursively decrements `x` from `n` to zero and returns the value `0`. A tail-recursing interpreter would use only a bounded (independent of `n`) amount of return stack space in executing this program, because when `f` recurses, the value returned by the lower level will be that returned by the upper levels, so only one return address is ever pushed onto the stack.

The size of the environment used in the interpretation is another matter. At the bottom level of the recursion, when `x` is equal to 0, the tree environment will have `n` and `f` bound and `n+1` bindings of `x`, each one burying the previous one. Thus, doing tail recursion does not save environment space, only return-point stack space.

However, the tail-recursing interpreter has still helped us solve the problem of the extra dead bindings of `x` in the previous problem piling up. Whereas the normal interpreter would put pointers to all the intermediate environments onto the return-point stack, the tail-recursing interpreter builds the environment without keeping pointers to all the ancestor environments. In this way, the extra bindings of `x` are converted from being simply dead to being buried. Since buried bindings are easier to detect than other kinds of dead bindings, the technique of tail recursion has helped us make progress towards a solution in this case.

In general, the techniques of tail-recursion and buried-binding reclamation, consistently applied, will allow "iterative style" programs to be interpreted with a bounded net use of storage. In other words, these techniques allow iterative-style programs to recurse to an unbounded depth while holding onto only a bounded number of free storage cells. (This statement is actually false as it stands, because a terminating program which runs for n steps must use $O(\log n)$ space. However we are assuming that data types like integers take only one free storage cell—a good approximation for most Lisp programs.)

It is well known that recursion is a stronger programming technique than iteration; any iterative program can be rewritten in a uniform way to use recursion instead, while the reverse is not true. However, we would like a stronger reducibility than simple expressibility—we would like to say that recursion requires no more than a constant factor more in running time or net space than iteration. This is why tail recursion and the elimination of buried bindings is so important; they allow us to claim this stronger reducibility.

Tail recursion and dead binding elimination would allow a Lisp program which has been converted by Fischer's algorithm [Fischer72] to a continuation-passing style to use no more than a constant factor more maximal net space than the original program. This would strengthen the expressibility

---

[9] In Common Lisp, this expression would be written as follows:

```
(labels ((f (x)
           (cond ((zerop x) x)
                 (t (f (1- x))))))
 (f n))
```

equivalence that Fischer claims to a maximal net space as well as running time equivalence. It is not clear what would happen to garbage collect time.

The `EVAL` Problem

If the function `eval` is used with a single argument in Lisp 1.5, that argument is interpreted as an S-expression which is to be interpreted in the "current" environment. The value computed by this interpretation becomes the value of the application of `eval`. The problem with this construct is that the S-expression can be created at run-time, thus eliminating the possibility of pre-computing its list of free variables. To be semantically correct, then, the interpreter must keep bindings for all variables which might be referenced in the argument to `eval`. The number of different variables which might be needed is bounded by the number of different variables used in the program, assuming that `eval` is not being called from inside another `eval`! Buried bindings can thus be deleted from the environment without harm, but no other binding can be proven dead a priori because it might possibly be referenced in an S-expression which will be `eval`'ed.

`eval` is not the only construct which prevents us from determining free variables and therefore dead bindings. `apply` with two arguments, i.e., with an implied environment, has the same troubles. This is because `(apply` *`f a`*`)` is essentially the same as `(eval (cons` *`f a`*`))`. However, there is one more construct which can cause the problem. This is the implied `eval` that takes place when the `CAR` of a form cannot be recognized as either a primitive function, a lambda-expression, a label-expression, or a closure. In these cases, (f a_1 a_2 … a_n) reduces to ((eval f) a_1 a_2 … a_n). Here, the expression `f` can compute a lambda-expression having any free variables, whatsoever.

Lexical versus Dynamic Scoping of Variables

In a lexically-scoped language, the depth (height?) of the environment tree is bounded by the depth of lexical nesting of the program text. Hence, the buried binding problem in these languages cannot grow to the vast proportions that can be achieved in dynamically-scoped languages. In particular, the variables from one invocation of a recursive routine do not stack on top of one another, burying their previous incarnations, but stack "beside" one another. Therefore, if the intermediate environments are not pointed to by the return-point stack, they can be reclaimed by a normal garbage collection. As a result, tail recursion in lexically-scoped languages is enough to allow bounded storage interpretation of iterative style programs without requiring a reclamation scheme for buried bindings.

We do not know the reasons why the architects of Lisp chose dynamic rather than lexical variable scoping. If it was to save time or the storage required to hold the environment needed for functional arguments or values, these small savings are gained at the likely expense of many dead bindings and the large time lost in tracing over them at garbage collect time.

The side-stacking feature, together with the ease of identifying free variables, would seem to make lexically-scoped languages far superior to dynamically-scoped languages. There are several advantages of dynamic scoping which have been overlooked, however. Dynamic scoping allows an extremely flexible coupling of independent modules at execution time. Flipping the coin over, lexical scoping ignores the problems of hooking together independent modules. The simplest lexical scoping model assumes one large, complete program text; linking is required to resolve external names of multiple modules before execution can proceed. The problem of correctly associating mutually recursive functions is not straight-forward in these languages. These requirements lead to a read-resolve-evaluate-print style of interpretation rather than Lisp's read-evaluate-print style.

The resolution phase is not a problem in systems with a clean break between program text and data. However, many artificial intelligence programs attach pieces of program to data and these pieces are dynamically linked together at run time by Lisp's dynamic variable scoping.

Thus, for systems requiring the utmost in linking flexibility, dynamic scoping can be very valuable. It is for these systems that solutions to the dead binding problem are important.

ACM Lisp Pointers **V**,2 (April-June 1992), 11-19.

XLisp — leXically scoped Lisp

For people who like this sort of thing,
This is the sort of thing they will like.

We would like to describe a variant of Lisp which is as close to Lisp 1.5 as possible for a lexically-scoped language. In other words, it makes the smallest deviation from Lisp 1.5 while being lexically-scoped.

People may complain that dynamic scoping is an inherent quality of Lisp; that Lisp wouldn't be Lisp without it. We believe that the essential qualities of Lisp are its trivial syntax, its S-expressions formed from CONS cells, its garbage collector, its atoms and property lists, and its representation of programs as S-expressions. A language with all of these features would be more Lisp-like than a language with only dynamic scoping (e.g., APL).

You may have thought that we would proceed to describe yet another language, complete with an exhaustive listings of functions, etc. However, that is not necessary—it has already been done. For XLisp is just our name for *Scheme* [Sussman75]![10]

Future Work

The buried binding problem is a precisely defined problem for which an elegant representation and/or garbage collector can probably be devised.

The general dead binding problem will require a formalism to state more precisely the timing and mechanism of the binding of variables and their reclamation. What Reynolds did for the order of interpretation issue using continuations [Reynolds72] needs to be done for the precise timing of binding and reclamation. Either a new formalism for specifying bindings, or a stronger interpretation on previous formalisms, must be used to pin down more precisely the semantics of the meta-circular interpreter of Lisp. Once this is done, the language will be strong enough to distinguish between classical and tail-recursing interpreters; between interpreters with dead binding problems and those without. When the language is strong enough to state the problem, solutions will not be far away.[11]

References

Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

Baker, H.G. "Shallow Binding in Lisp 1.5". Manuscript, April, 1976. Subsequently published in *Comm. ACM 21*,7 (July 1978),565-569.

Fischer, M.J. "Lambda Calculus Schemata". *Proc. ACM Conf. on Proving Asserts. about Programs, Sigplan Not.* (Jan. 1972).

Hewitt, C., and Baker, H.G. "Actors and continuous functionals". In Neuhold, E.J., *ed. Proc. IFIP Working Conf. on Formal Desc. of Progr. Concepts*, IFIP, Aug. 1977, 367-387.

McCarthy, John, *et al. LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, 1965.

Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages". *ACM Conference Proceedings*, 1972.

Steele, G.L. Personal communication, June, 1976.

Steele, G.L. *RABBIT: A Compiler for SCHEME*. AI-TR-474, AI Lab., MIT, May 1978.

Sussman, G.J., and Steele, G.L. "SCHEME—An Interpreter for Extended Lambda Calculus". AI Memo 349, MIT AI Lab., Dec. 1975.

---

[10]But without *reified continuations*.

[11]Steele was aware of this paper when he wrote his Master's Thesis [Steele78]; indeed, he made valuable suggestions for its improvement [Steele76]. Nevertheless, Steele's *closure-sharing* analysis [Steele78,p.60-63] often creates buried bindings where none existed before; if the goal of minimum "maximal net space" is desired, closure-sharing can be a "pessimization" rather than an optimization.