REFERENCE COUNTING CAN MANAGE THE CIRCULAR ENVIRONMENTS OF MUTUAL RECURSION*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 73

REFERENCE COUNTING CAN MANAGE THE CIRCULAR
ENVIRONMENTS OF MUTUAL RECURSION

DANIEL P. FRIEDMAN
DAVID S. WISE

MAY, 1978

Keywords: Storage management, threads, closure, Y-combinator, funarg, LISP, lexical scoping, ISWIM, applicative programming.

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS75-06678 A01 and MCS77-22325.

To appear in Information Processing Letters.

Introduction

In this note we advance reference counting [3] as a storage management technique viable for implementing recursive languages like ISWIM [14 , 2] or pure LISP [15] with the labels construct for implementing mutual recursion from SCHEME [19]. Labels is derived from letrec [13 , 9] and displaces the label operator [15], a version of the paradoxical Y-combinator [4]. The key observation is that the requisite circular structure (which ordinarily cripples reference counts [12 , p.412]) occurs only within the language--rather than the user--structure, and that the references into this structure are well-controlled.

Two sorts of function bindings are provided in SCHEME, and both require that the environment be attached to the function definition to make a function closure, whenever that function is bound as a value. The first, akin to the funarg structure of LISP, occurs whenever a lambda-expression is encountered by the evaluator. It results from a lambda-expression being bound to a variable, which usually occurs as an argument to another (e.g. maplist [15]). This case includes the so called "upward" as well as "downward" funarg. The second results from a labels structure, the only expression in an applicative subset of SCHEME which can generate circular structures.

In LISP 1.5[15] a function closure is represented by a triple composed of the keyword funarg, a function definition (usually a LAMBDA-expression), and an environment. In Figure 1 we have altered this representation slightly, saving a node and distinguishing the reference to the environment, which may (Figure 1a) or may not (Figure 1b) be counted in the reference count scheme. In this representation a closure is an environment list (or association list [15]) preceded by two items: a keyword (funarg or βfunarg) and a function definition.

We shall differentiate closures built from labels with the symbol βfunarg because the reference-count system treats them specially. The interpretation of either closure by the language interpreter is the same; they are created and destroyed differently. The bindings we establish through βfunarg (so-called because of the BETA notation of Sussman and Steele [19] and suggestive of the backward environment pointer) are significant because they allow the programmer to express mutual recursion of locally-defined functions. This facility reduces the programmer's burden by reducing global bindings (and the extra restrictions that go with them) while improving the performance of the code because active environments remain small (along with storage space and access cost).

Using labels

A typical labels expression might appear like a prototype of the form:

```
labels:<
  (f ~ definition-of-f
   g ~ definition-of-g
   h ~ definition-of-h)  expression> .
```

Its meaning is that of the evaluation of expression in an environment which includes the bindings of the function identifiers f, g, and h to their corresponding definitions (closed lambda-expressions). Moreover (and most significantly), these closures refer to that same environment so that occurences of the identifiers f, g, or h in any of the definitions or in expression refer to these newly created function bindings and nothing else. Other identifiers have the meaning determined by their bindings in the non-local environment which envelops the labels form.

The interpretation of labels is most similar to an ALGOL 60 block. Closures establish "static" or "lexically-scoped" bindings for the identifiers f, g, and h for all lexical occurences of these identifiers within the labels form. In contrast, many interpreters (e.g. LISP 1.5, APL, and SNOBOL4 [18]) provide "dynamic" bindings under which the meaning of identifiers within a function definition depends on the eventual environment of function invocation, regardless of where the definition occurred. (LISP 1.5 even allows the user to explicitly close function definitions at any time in between by invoking it's function closer, function.)

The binding of an ordinary function identifier *f* appears in Figure 1a. The environment referenced there never includes the new binding of f, itself, because the closure is formed *before* that binding; this new binding properly augments that environment. Figure 1b illustrates a function binding established through *labels*. In this case the closure does include the functional binding through a structure (the association list or a-list) shown in Figure 2. The structure there added to G in order to construct E results from the *labels* form presented above.

Since the structure between E and G in Figure 2 is derived from interpretation of one *labels* form, it is constructed all at once. At creation there are, therefore, no shared references *into* that new piece of the association list. Furthermore, each use of this structure in the interpreter to resolve a reference (*assoc* in LISP) involves a traversal which will be completed before E is dereferenced. Together these facts imply that when

It is important that this traversal does not alter the structure, nor does it create any sharing. If the desired value is found then either it does not contain a circular reference, or it is *copied immediately* and the copied value is returned. Since the only circular references in our scheme arise from *labels* environments, the interpreter need only check to see if the value to be returned starts with *βfunarg* If it does not, then the value may be borrowed and a shared reference may be returned without delay.

If it does then that closure is copied as a **funarg**, requiring two new nodes and a new reference (counted) to the environment; all references to the circular structure remain at the head, E in Figure 2. Most often these two nodes are recovered immediately because the closure is passed directly to the system evaluator which consumes it before a second reference can be established. (Any other use of a closure, as another function's parameter or value, occurs so rarely that actual copying of a βfunarg closure as a funarg closure will be the exception rather than the rule.) This is the only way closures may be used with so-called "first order" functions or a call-by-need [21] protocol where the only time a functional binding is sought is when it will be applied by the evaluator. With second-order functions under a call-by-value protocol (like pure LISP) these copied structures could proliferate within the system at some cost in space, but fortunately practical instances of such programs are quite rare.

Not counting circular references

The dotted edges in Figure 2 illustrate the only circular references in the system. These circularities only occur in the environment structure which is created, accessed, and destroyed exclusively by the interpreter. Since the user is not provided with field altering primitives (i.e. rplaca, rplacd, set in LISP), he has no way of constructing his own circular structures. Such restrictions occurring in languages like pure LISP have particular import for parallel processing systems [20] aside from the implications for storage management considered here.

As a result of these observations, the circular links, dotted in Figure 2, need not be counted by the reference count scheme. The reference count of the node referenced by E is one just after creation of the environment between E and G. It may rise and fall as it is "borrowed" and "erased" [3] and when it eventually falls to zero, all the nodes illustrated in Figure 2 accessible from E, but not from G are necessarily recoverable. That recovery cannot occur, however, until all use of the closures bound to f, g, and h has been completed; the accession of any of these bindings would have resulted in a copied closure including a counted reference to E, precluding its recovery, and so if E is dereferenced completely then there will never be any more access to those closures.

On recovery, the distinction between funarg and βfunarg is significant. When an environment is dereferenced and its structure is recovered, any structures (atomic identifiers and their bound values) indirectly referenced will have their reference counts decremented; if to zero then those structures will be recovered in turn. The reference count of the environment within a dereferenced funarg closure has its reference count decremented as does the function definition of the closure, but the environment within a βfunarg closure is treated differently. Such environment references, the dotted ones in Figure 2, are dereferenced but there is no reference count manipulation; the fact that a βfunarg closure is being recovered is sufficient to demonstrate that its environment is already recovered. So the βfunarg indicator tags the environment link as a sort of thread [17] with respect to storage recovery.

Implications

In this section we make several observations about reference counts and circular structures. The first is that these remarks extend to nodes of more than two fields rather naturally, although two is sufficient to argue the matter and to mimic its application in LISP. Indeed, Figure 1a or Figure 1b each illustrate a closure which probably would require only a single node in any efficient implementation, instead of the pair of nodes shown.

Second is the description of the certain sort of circular structures which may be handled by a reference counting storage manager. We specify three criteria, the first two of which guarantee that there will be no shared references (reference counts above one) anywhere within a cycle except at the head. Firstly, circular structures should be created all at once (in our case by the interpretation of labels) so that no shared reference is created before the circularity is established. Secondly, any use of a proper suffix of the cycle (a ßfunarg closure in our example) must be copied as an independent and non-circular structure rather than being borrowed or shared. Finally, in order that the storage recovery mechanism avoid the circularity, the circular link to the head of the cycle must be "tagged" (by β for instance) so that it will be treated as null on recovery. Taken together, these three restrictions assure that the cycle is referenced and dereferenced as a unit; no part of a circular structure will be created before or will be preserved after any other part.

The third observation is a degenerate case of the circular structure considered above. Figure 3 illustrates self-references to a node from within the same node. Such references need not be included in the reference counts for the same reasons that we considered before. These circular structures are created all at once by a single field assignment, they have no references except at the front (since there is nothing internal to reference), and the circular "thread" is easily recognizable without a "tag" like β (by the reflexive pointer).

Such direct self-reference may, therefore, be handled by any reference count storage manager regardless of the number of pointer fields in the node. They need not be isolated to distinguished structures like the system-managed environments, but may be allowed in user structure as well. We have used a structure like that in Figure 3 to represent infinite homogeneous lists [6] in the reference count implementation of our applicative language [11]. The structure shown here represents an infinite list of zeroes, the zero vector of infinite length.