Manchester Functional Programming Group

Document	**MFPG-Report-0001**
Author(s)	**David Lester**
Title	**Distributed Garbage Collection of Cyclic Structures**
Status	**Draft**
Revision	**1.1**
Last Revised	**92/09/16 11:24:38**

Distributed Garbage Collection of Cyclic Structures

David Lester

Functional Programming Group,
Department of Computer Science, Manchester University,
Oxford Road, Manchester M13 9PL, UK.

Electronic mail: dlester@cs.man.ac.uk

Abstract

One of the key features of modern high level programming languages is the automatic allocation and reclaimation of space for data structures. This paper concentrates on the problem of providing these facilities satisfying the following constraints:

- The programming language permits cyclic data structures;
- The architecture is a distributed multi-processor; and
- Global synchronization is not required.

Keywords: Cyclic Data structures, Distributed Algorithms, Functional Programming, Garbage Collection

1 Introduction

1.1 Weighted Reference Counting

This garbage collector – invented independently by Bevan [1987] and Watson and Watson [1987] – associates a weight with each pointer in the heap, and a reference count with each node in the heap.

The problem with this collector is that it is not possible to detect and reclaim cyclic data structures.

1.2 Cyclic Refernce Counting

In his DPhil thesis, Hughes [1983] discusses an extention to the straightforward refernce counting technique that enables it to detect and reclaim cyclic structures that become garbage.

The problem with this algorithm is that it requires the decrement reference count messages to arrive in a particular order, *and this is hard to arrange in a distributed system.*

2 A Solution

Arranged as it is above, one possible line of attack is to see if there is any possibility to synthesize a new collector from the two we have presented.

2.1 Three nodes

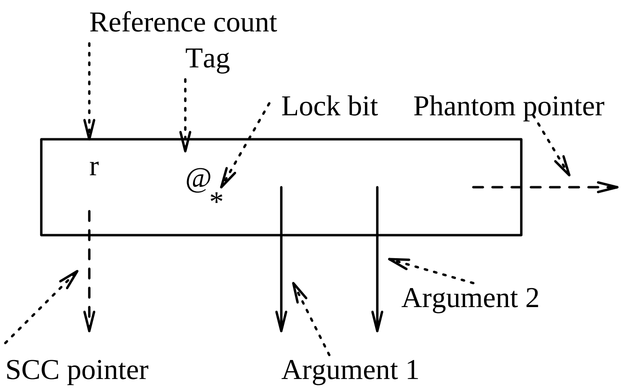

Figure 1: Application node

In a real machine we would need a range of constant nodes (numbers, functions, *&c.*) as well as constructors, and the following three types of special nodes.

The application node is shown in Figure 1. Besides the usual information – the tag and two pointers (both weighted) – there are the following extra fields:

Reference Count This is the nodes reference count and is used as in the Bevan/Watson/Watson schemes.

SCC Pointer This is similar to Hughes cyclic reference count, in that all of the nodes in a strongly connected component point have this field set to point to the same real node, which is itself a member of the component. This pointer is *not* a weighted pointer.

Phantom Pointer This is used to point to a node that will deal with incoming pointers from outside of the strongly connected component. It *is* a weighted pointer. It may be nil, in which case there are no external references to the node, or it is a singleton node.

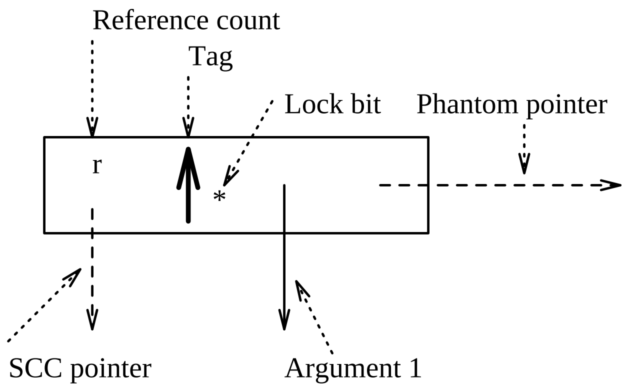

Figure 2: Indirection node

Ordinary indirection nodes, as shown in Figure 2, have the same extra fields as application nodes.

Finally, in Figure 3 we have *phantom indirection nodes*, that are used to insulate strongly connected components from each other. The idea is that when we need to find a way to divide the weight of indivisible pointers into a strongly connected component, these phantoms provide us with way to do this. Both the next node and the phantom pointer are weighted pointers.

In Figure 4 we have an example of the graph that might be produced by the following program:

```
letrec x = S (K x) in x
```

3 Creating Cycles

Because we wish to show that the collector can deal with cycles, we had better provide a way to create cycles. The simplest way to do this is by

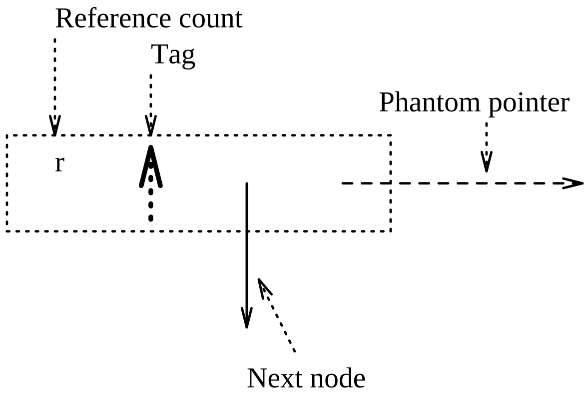

Figure 3: Phantom indirection node

using the "knot-tying" Y combinator[Peyton Jones 87, Page 219]. Let us first consider the case where the root of the expression `Y x` and `x` are in different strongly connected components. Recall that we can detect this by seeing whether the node `x` and `n` share the same SCC pointer.

Given that `n` and `x` aren't in the same component, we are about to create a new cycle. This requires us to insert a phantom node between the two original components. This is shown in the lower part of Figure 5. The phantom node has an initial reference count of infinity, and its phantom pointer is nil.

To this phantom node we link the application node which represents the cycle. The application node is (initially) the only node in the cycle, so its SCC pointer points to itself. The application node's phantom pointer points at the new phantom node.

The other case to consider is where the original root of the expression (`n`) and the argument `x` are in the *same* strongly connected component. This situation is shown in Figure 6.

Because we don't wish to deal with the pointer to the root of the redex, we turn the original node into an ordinary indirection node, and create a new application node for the sub-cycle created by the Y combinator. The new node shares the SCC pointer of the node `n` and is given the nil phantom pointer. Having seen how to create cycles, we now look at ways to copy pre-existent pointers.

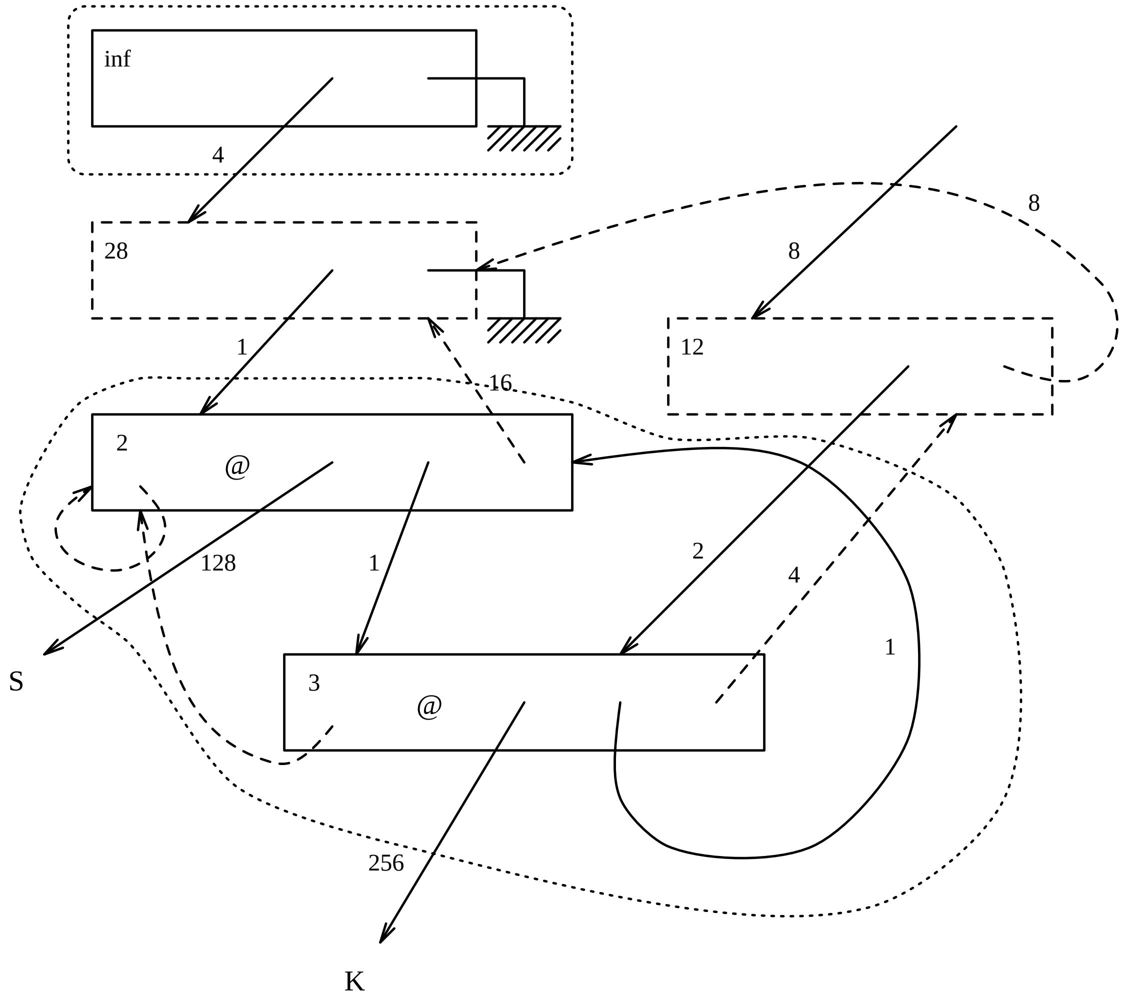

Figure 4: Example of a graph

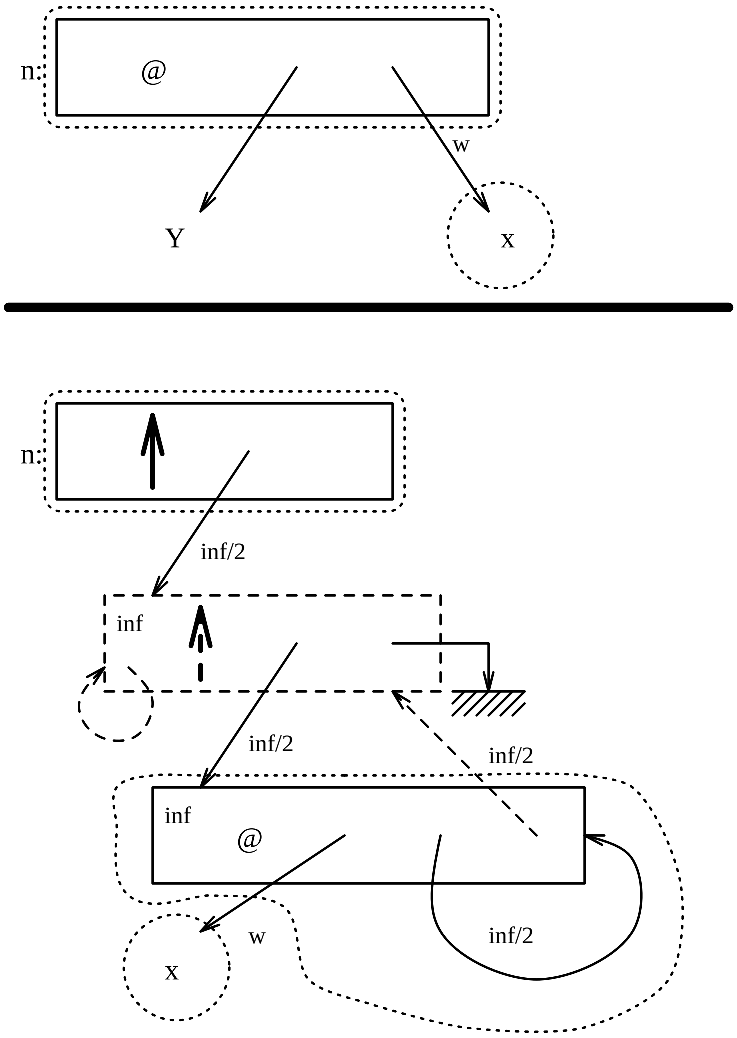

Figure 5: Y: creating a cycle

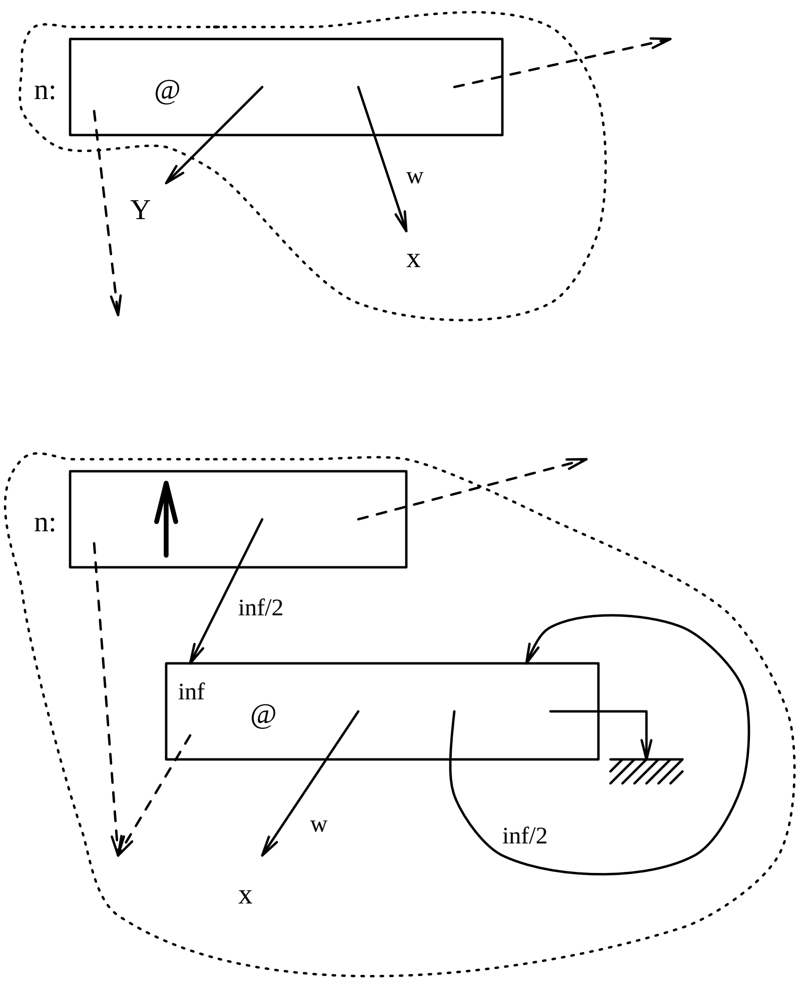

Figure 6: Y: extending a cycle

4 Making copies of pointers

In this section we describe the different cases that arise when we wish to copy a pointer. Again we assume that we are implementing a lazy functional language. Since the graphical behaviour of any combinator's reduction can be modelled with a combination of `S`, `K`, and `Y` combinators, that is all that intend to exhibit.

This does not mean that the collector only works for Turner's combinators [?].

We will merely consider the rewriting of the `S` and `K` combinators. Rewriting the `S` combinator (in the general case) involves:

1. creating copies of the pointers to the three arguments;
2. creating two application nodes;
3. rewriting the root appplication node and deleting the first pointer in the spine.

The `K` combinator is simpler, in that no new nodes are required to build the body.

We can determine the strongly connected components of the newly created nodes of an `S` combinator rewrite, by looking at the strongly connected components of the arguments. If these differ from that of the root of the redex, then the new nodes are singleton nodes, *i.e.* each of them is in a strongly connected component of its own. In this case the pointers from the application node are *external pointers*, because they are external from the strongly connected component of the argumnet.

The alternative is that the argument and the rot of the redex *are* in the same strongly connected component. This implies that the new application is also within the same component, and therefore its pointers are said to be *internal* to the strongly connected component. (See the discussion in [Hughes 83, Page 109]).

The ability to determine to which strongly connected component to assign the new application nodes is a property of functional languages that is unlikely to be present in general.

4.1 Creating external pointers

In Figure 7 we see the usual situation: we must copy a pointer whose source and target are in different strongly connected components of the graph. If

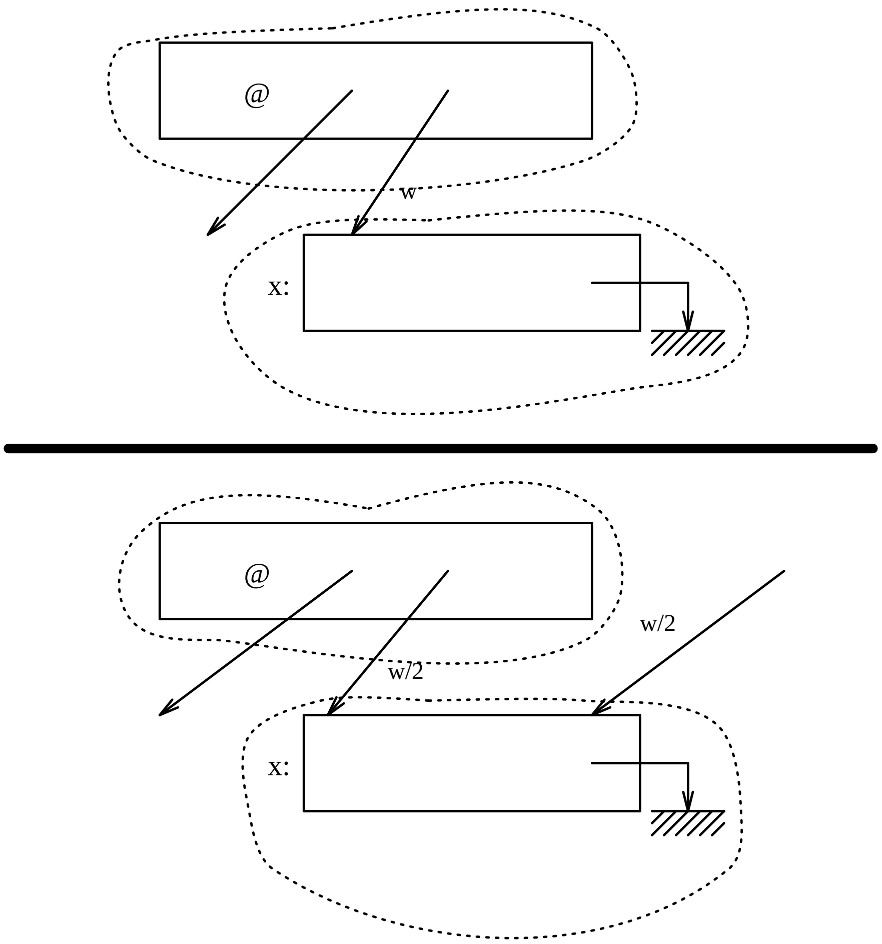

Figure 7: Copying a pointer: the usual case

the weight `w` is divisible then we divide the weight by two and the original pointer and the copy share the weight.

Sometimes this will not be possible, because the weight of the pointer is minimal *i.e.* it is 1. In this case we insert an ordinary indirection node between the original pointer and its target. We can now make the copy of the original pointer point at the indirection node, giving it a weight of `inf/2`. This is shown in Figure 8.

As we shall see later cycles are "insulated" from the rest of the graph by *phantom indirection nodes*. It follows that sometimes a pointer will point at one of these nodes. There is no complication here: we simply half the weight of the original pointer giving half to the copy (see Figure 9). In the event of a pointer of weight 1, we insert an ordinary indirection node and this then permits the weight to be divided again. (not illustrated, but as Figure 8).

4.2 Creating internal pointers

Making a copy of a pointer (lying entirely within a strongly connected component) for use inside the same strongly connected component is straightforward. We simply half the weight of the original pointer and assign the same weight to the copy pointer. In the event that the pointer weight is already 1, then we create a new ordinary indirection node between the two points copying the dotted pointer to the root node of the cycle. This is illustrated in Figure 10, where we see that the new indirection node's SCC pointer is the same as that of the other nodes in the cycle.

There is another problem for us to consider if we are not implementing a functional language: we may need to make a copy of an external pointer for use as an internal pointer. This may then be used to extend or create cycles in completely arbitrary ways, so we don't permit this.

4.3 Creating external pointers for nodes with phantoms

Now suppose that we wish to create a pointer to a node `x`, to be used outside the strongly connected component, and that the spine application node is within the same component as `x`. There are two cases to consider: when the node `x` already has a phantom node; and when a phantom node will need to be created.

In Figure 11 we see the situation where the node `x` has a phantom node which it points to with a weight of `w`. Provided `w` is not one the transforma-

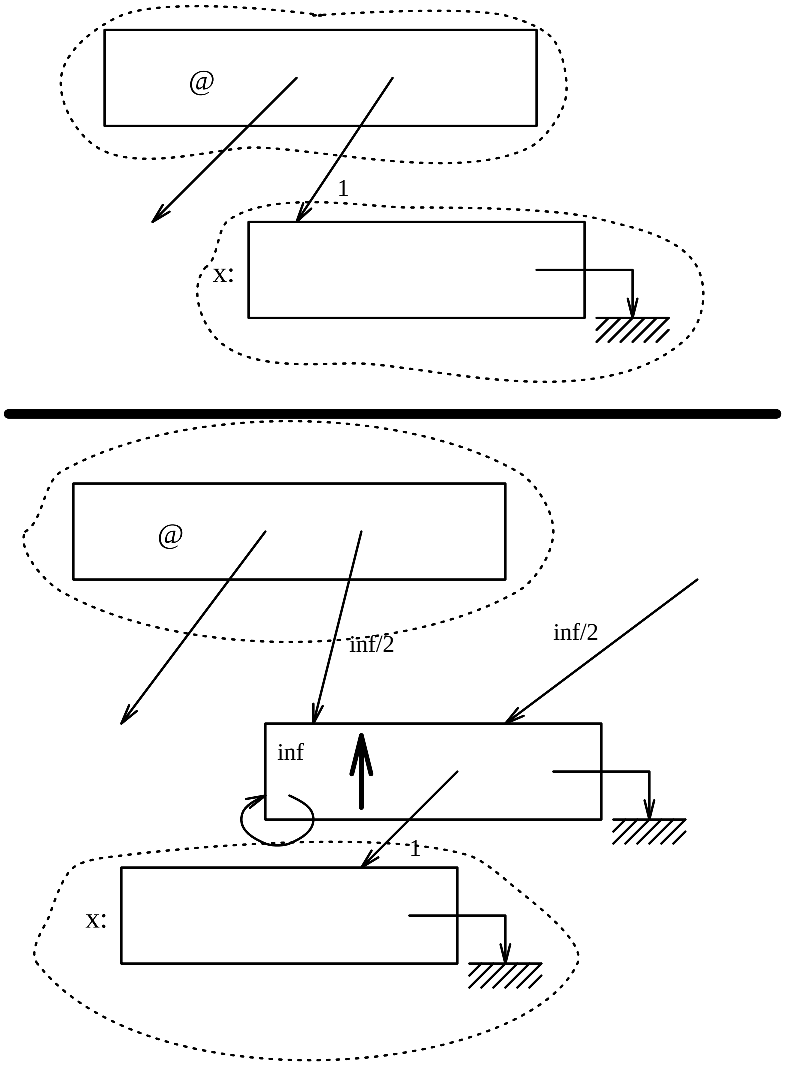

Figure 8: Copying a pointer of weight 1

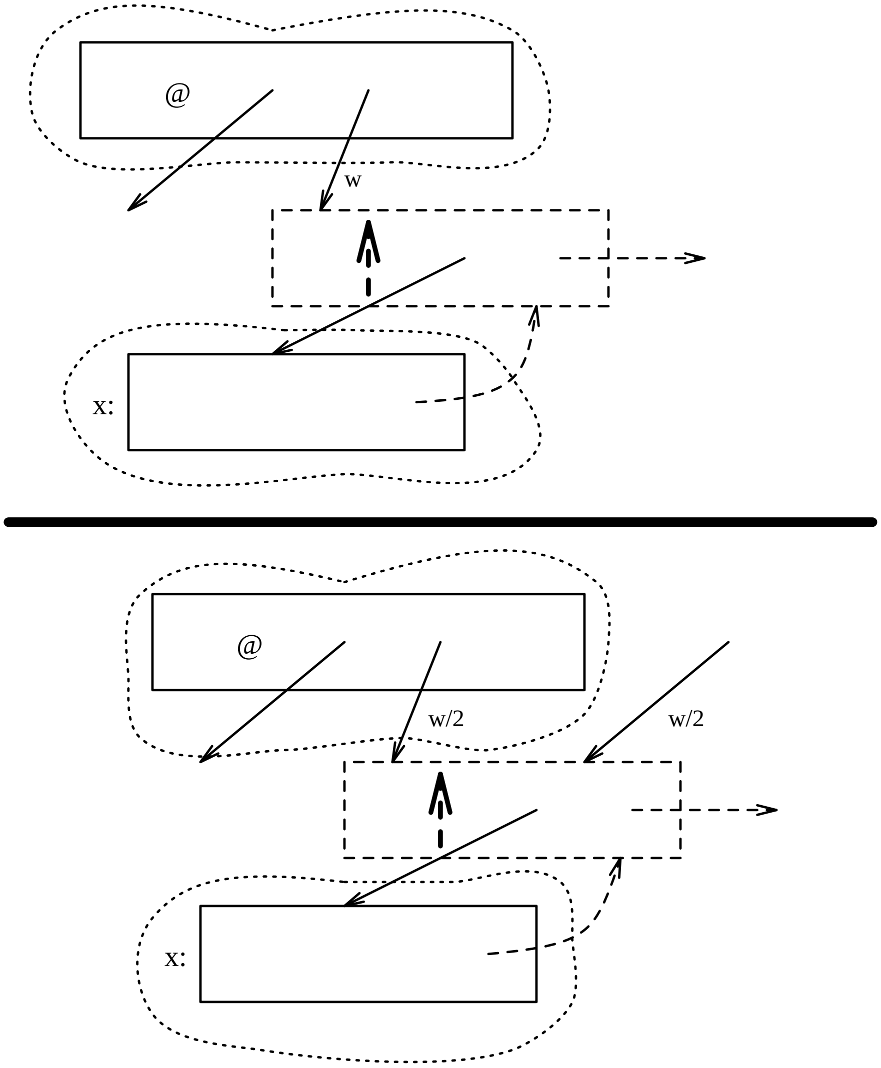

Figure 9: Copying a pointer to a cycle

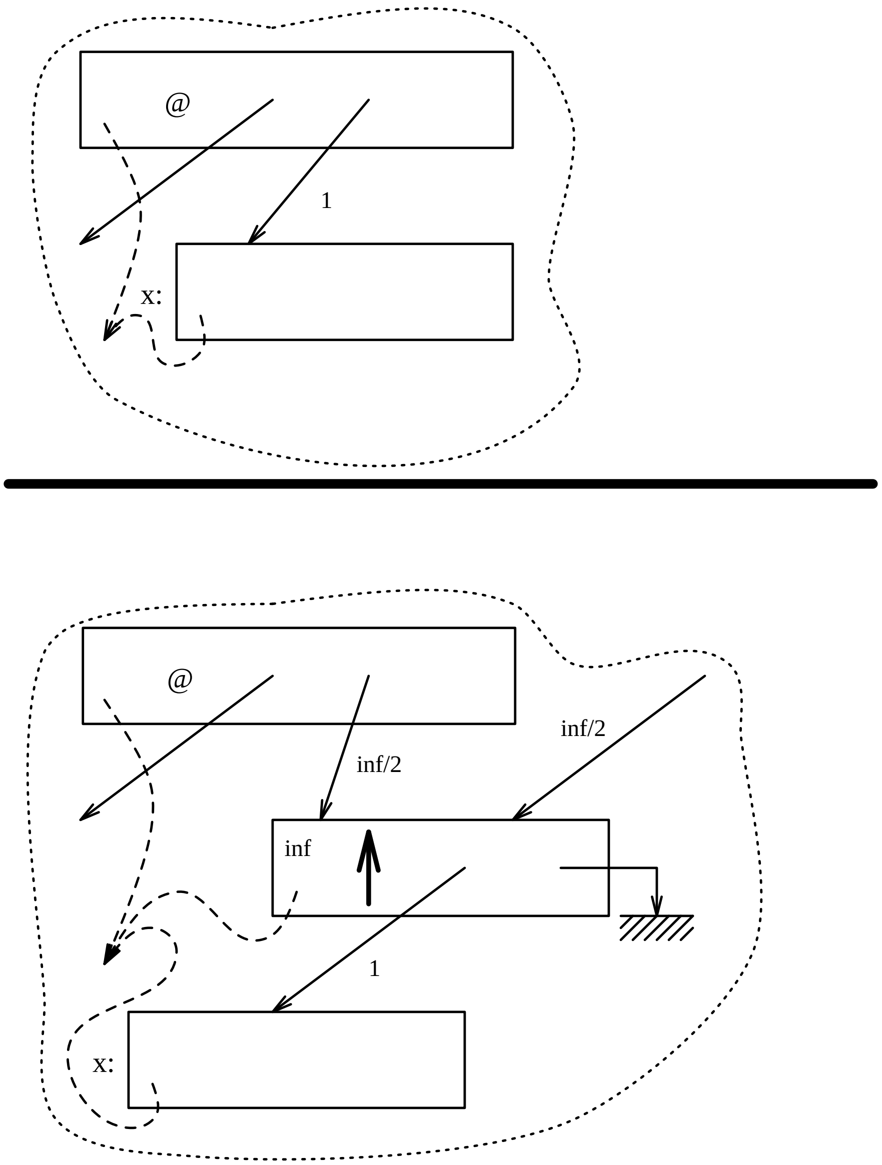

Figure 10: Copying a pointer of weight 1 within a cycle

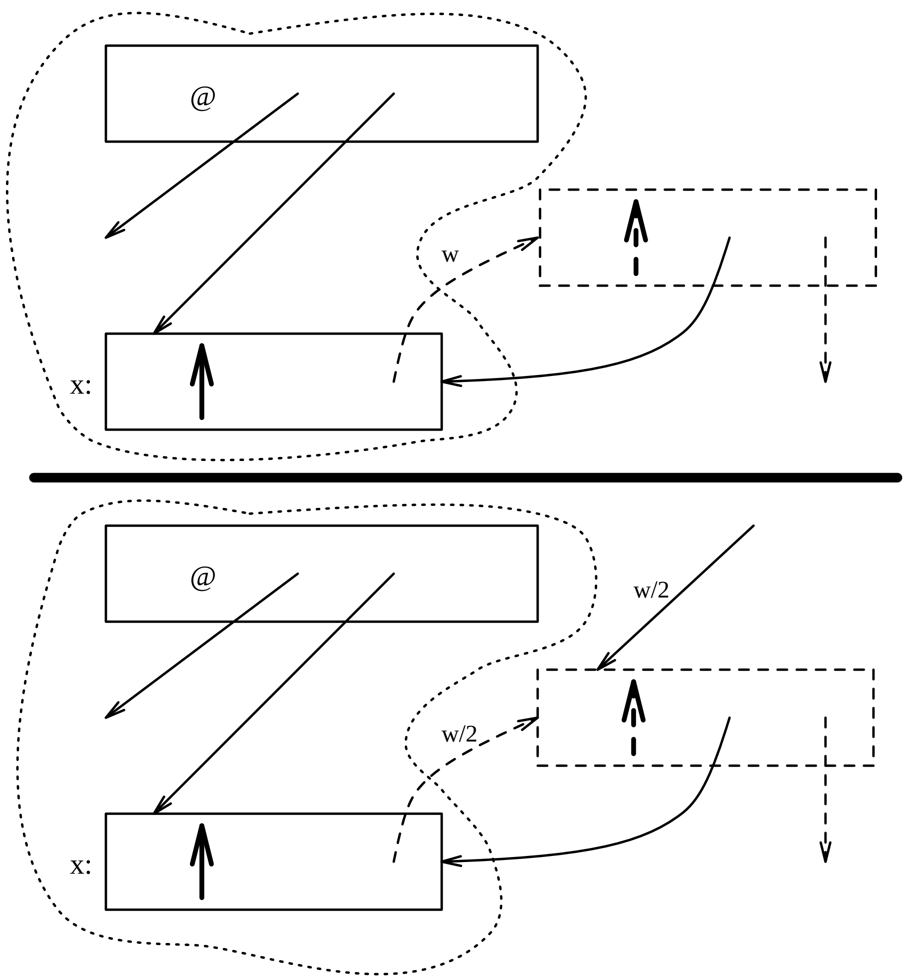

Figure 11: External pointer for a node with a phantom

tion of the graph is shown as shown in Figure 11. Of course we must have a way to deal with the case where this weight is 1; this is shown in Figure 12.

A new phantom node has been created which replaces the one originally used as a phantom by `x`. This new node will have a divisible weight, and so the new external pointer can be given the weight: one quarter infinity.

4.4 Creating external pointers for nodes without phantoms

The final case we must consider is that of creating external pointers to nodes without phantom indirection nodes pointing at them.

The first observation is that – in a fuctional language setting – there must be a phantom node between the root of the redex and the strongly connected component that contains `x`. The initial graph is shown in Figure 13. The simple situation, in which the weights of both `w1` and `w2` are divisible, is shown in Figure 14.

In Figure 15 we show the transformation to take when the weight `w1` is indivisible.

Of course it may be that the weight `w2` is indivisible, in which case we have the transformation shown in Figure 16. Finally, for those who like complexity, Figure 17 shows what happens when both `w1` and `w2` are indivisible.

This concludes the various ways to copy pointer that will be needed to implement a functional language.

5 Deleting Graph Nodes

In this section we shall see how to delete pointers and graphs. The most important criteria for graph nodes is as follows: *A 'real' graph node is garbage whenever:*

- *it has a null phantom pointer and a reference count of 0; or*
- *the weight of the pointer from a node's phantom node equals the node's reference count.*

A phantom node is garbage whenever:

- *the weight of the pointer from the node pointed to by the phantom node equals the phantom's reference count.*

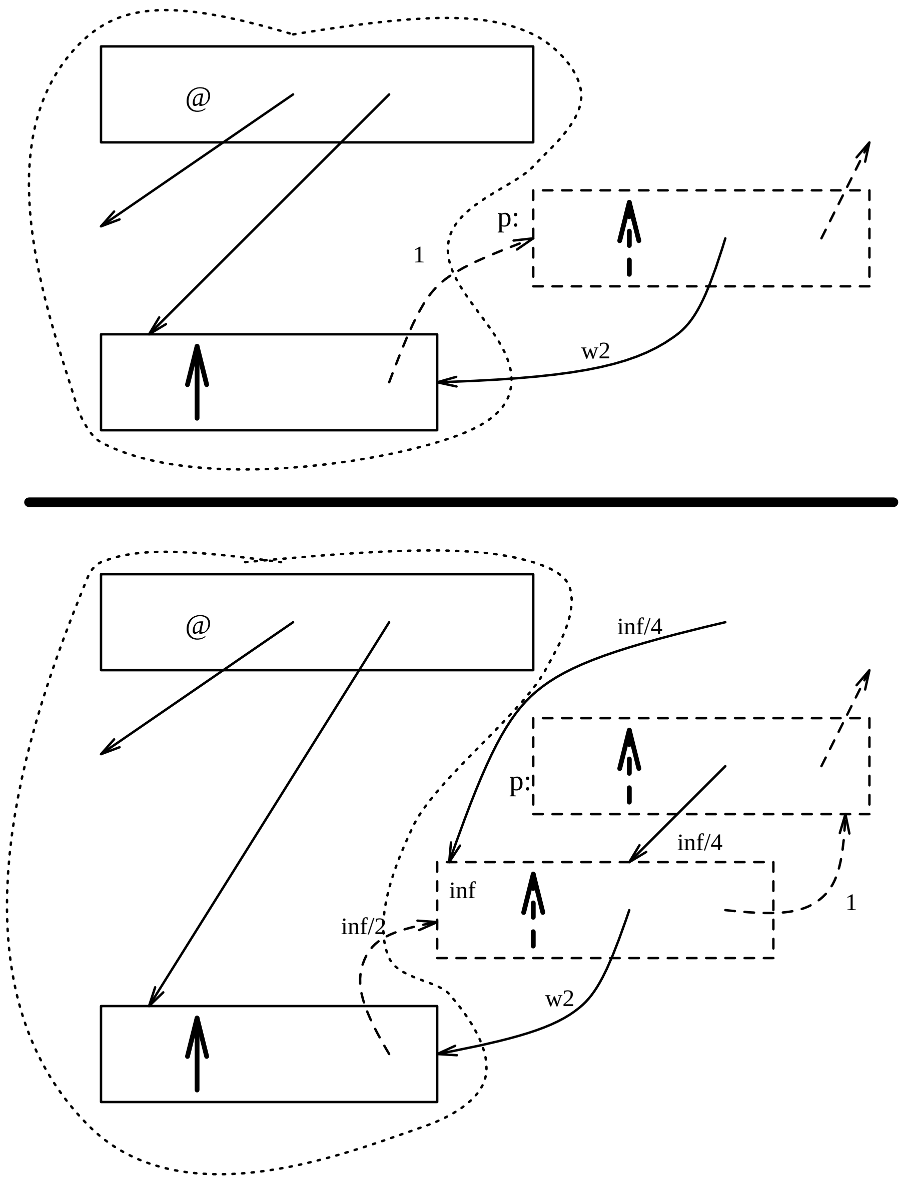

Figure 12: External pointer for a node with a phantom `w` = 1

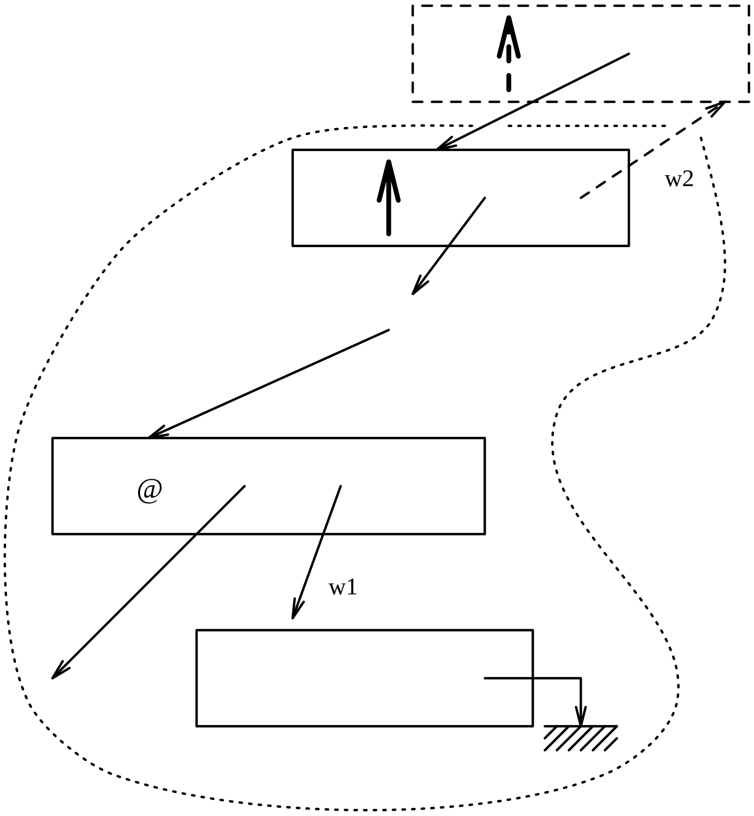

Figure 13: Initial graph

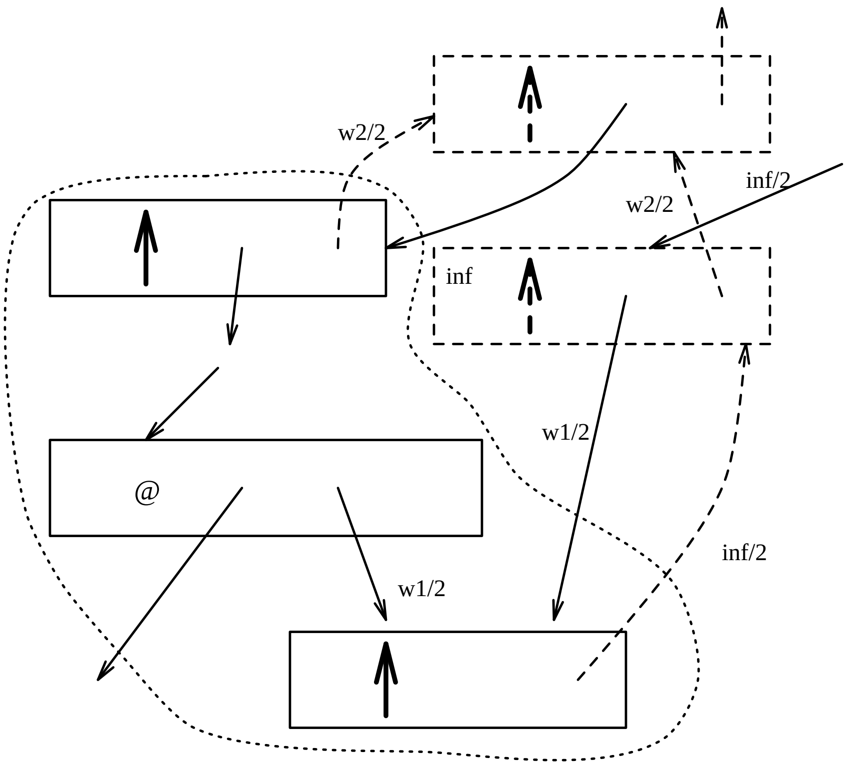

Figure 14: Creating a phantom for an external pointer: tt w1 $\neq$ 1, tt w2 $\neq$ 1

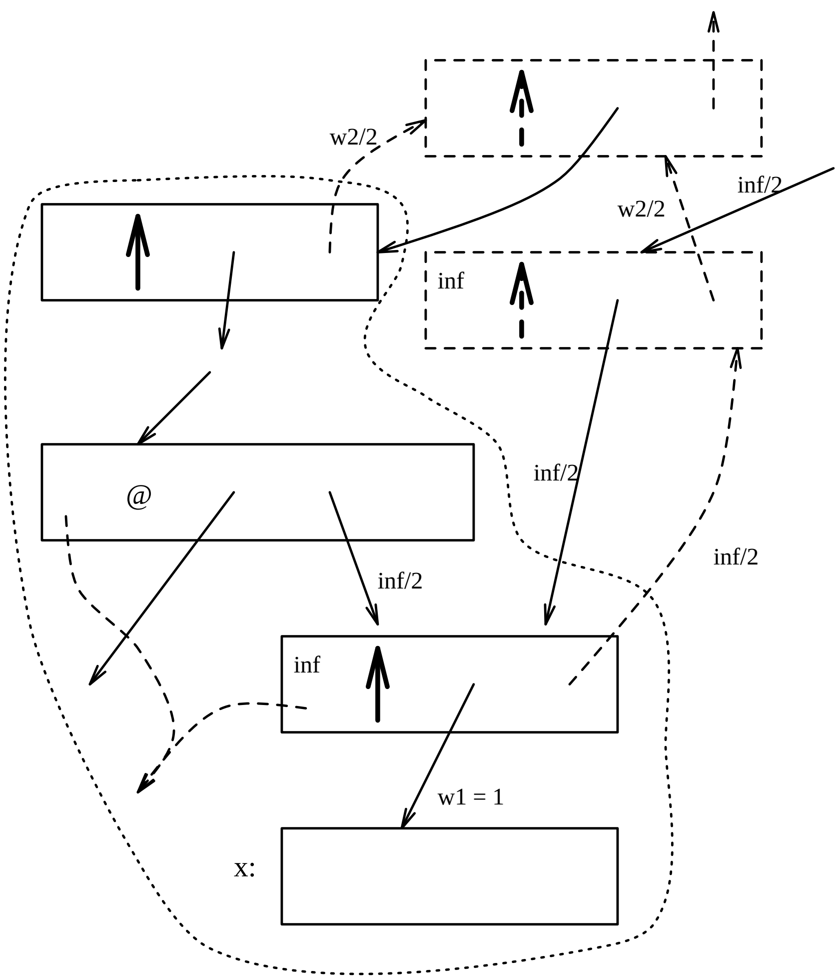

Figure 15: Creating a phantom for an external pointer: tt w1 = 1, tt w2 $\neq$ 1

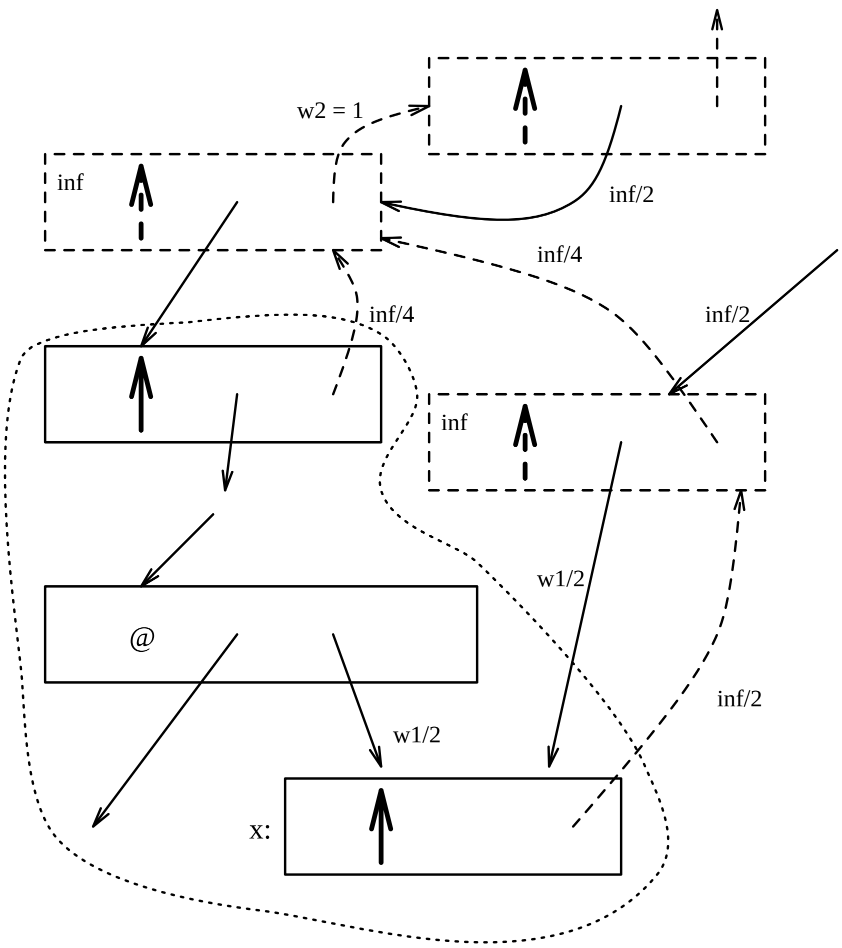

Figure 16: Creating a phantom for an external pointer: tt w1 $\neq$ 1, tt w2 = 1

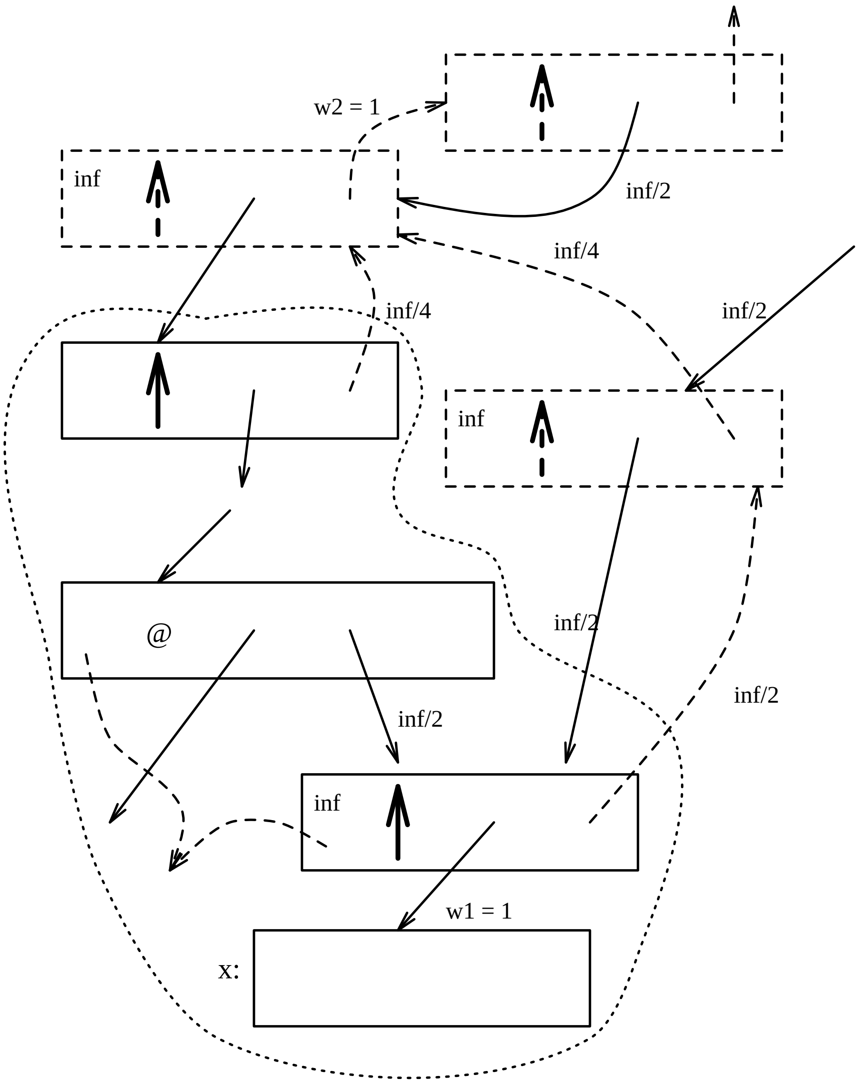

Figure 17: Creating a phantom for an external pointer: tt w1 = 1, tt w2 = 1

Once a node has been declared garbage we must recursively delete all outgoing pointers from the newly deleted node.

When the root phantom node (the one with null phantom pointer) is deleted the cycle it points to is also to be deleted.

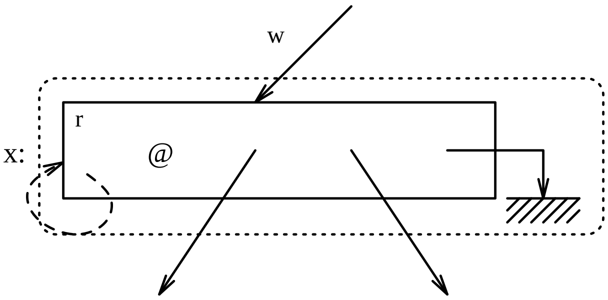

Figure 18: Deleting external pointer to real node

We begin by looking at the case of deleting a pointer to a real node. If `w` is equal to `r` then the node is garbage and we recursively delete its pointers. Alternatively, if `r` is greater than `w`, we simply subtract the weight from the reference count and stop.

The next case too consider is when the pointer is pointing to a phantom node, which is not the last one. This node (`x'`) becomes garbage when `r1` is equal to `w1` + `w2`. As this phantom node isn't the last one, we simply replace `x`'s phantom pointer by nil, subtracting `w3` from `x`'s reference count. The pointer to the phantom node `y'` is also deleted. This is shown in Figure 19.

Another case is where the pointer is pointing at the last phantom node (the phantom node that has a null phantom pointer). If the phantom node's refernce count equals `w1`+`w2`, the whole of the strongly connected component is deleted along with the phantom node `x'`. This is shown in Figure 20.

Suppose that the phantom node to be deleted doesn't have an associated 'real' node. Figure 21 shows what happens when the node `x'` is deleted.

The final case to consider is deleting an internal pointer, *i.e.* one that lies between two nodes in the same strongly connected component. This situation is illustrated in Figure 22.

The pointer is retained but its weight is now zero. Using the SCC pointer, we traverse the strongly connected component, locking each of the nodes. Using Tragan's Algorithm we analyse the locked nodes to see whether they still form a strongly connected component, or whether they are now a collection of smaller components.

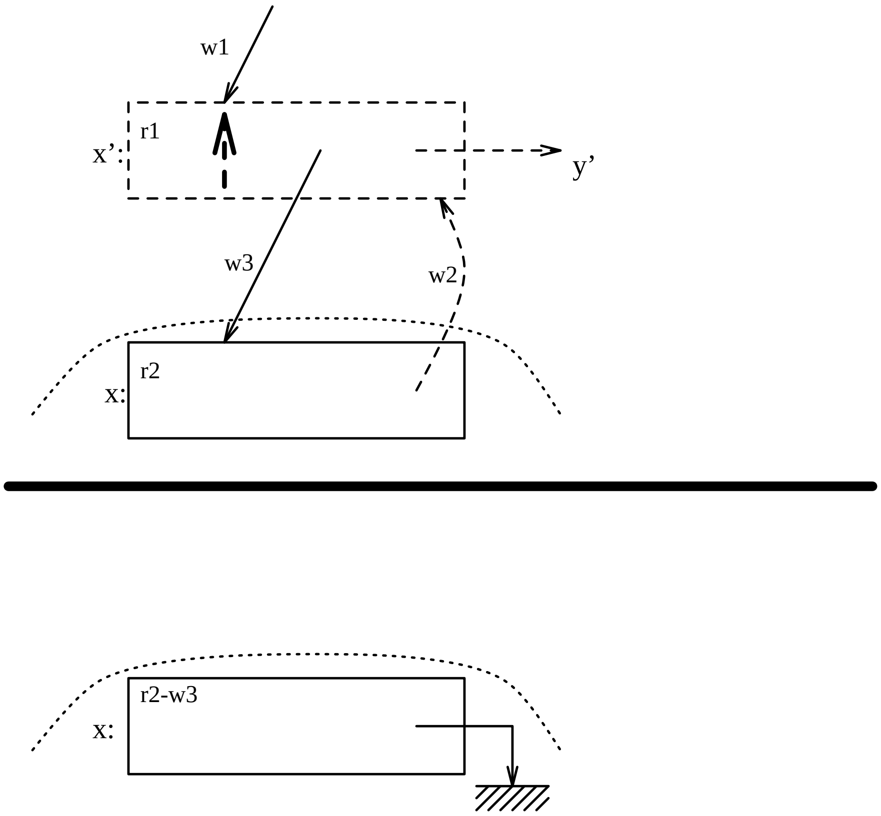

Figure 19: Deleting external pointer to phantom

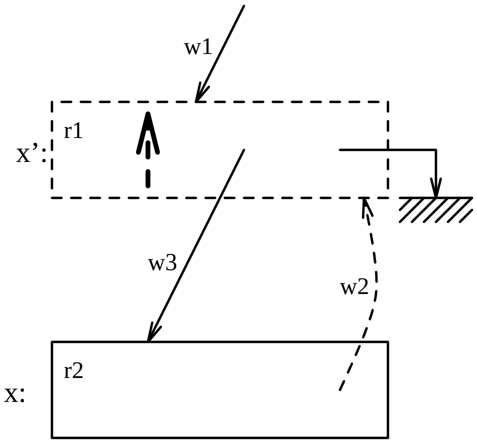

Figure 20: Deleting external pointer to the last phantom

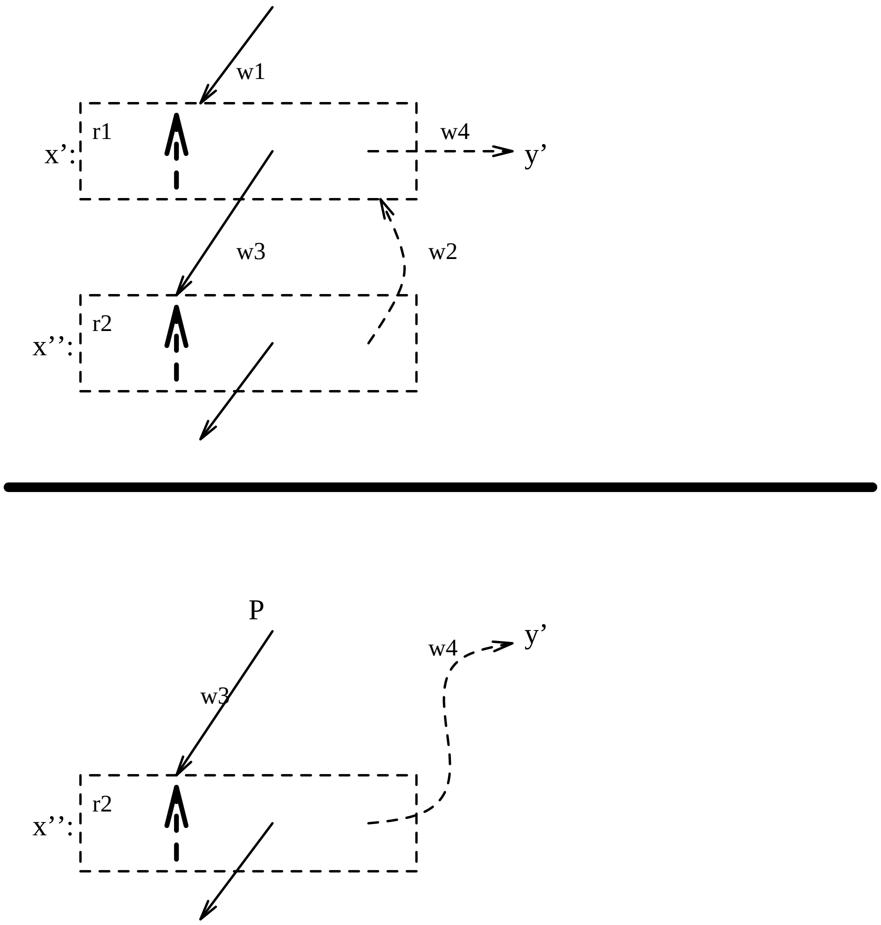

Figure 21: Deleting external pointer between two phantoms

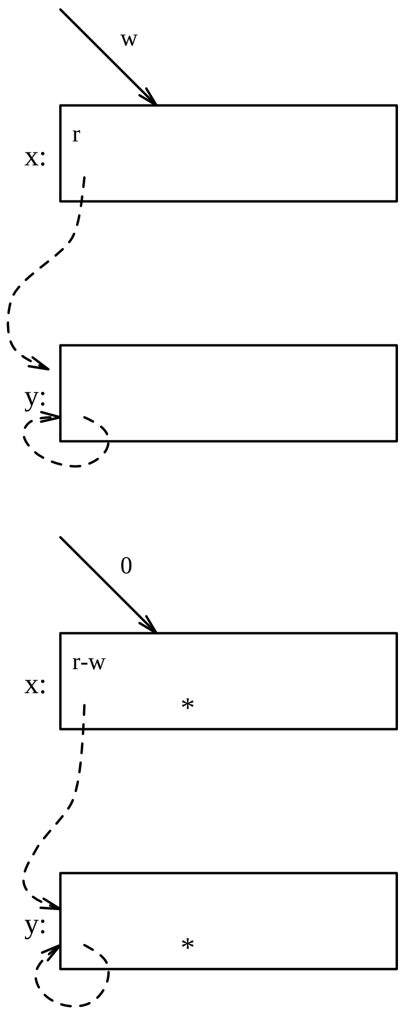

Figure 22: Deleting internal pointer

6 Conclusions and Further Work

As I have mentioned throughout this paper, I have no practical experience of the costs associated with this approach to garbage collecting. It could be that it is cheaper to ignore cyclic structures until they have built up to a sufficient extent that we stop the whole machine to stop them.

It is also possible that the efficiency of the collector could be improved by using a similar technique to the one I used in [Lester 89]. The idea is to use a compacting collector locally, and reserve the complexity of weighted reference counting for those places where the pointers really are pointing across processor boundaries. The problem I foresee with this collector is that we must mark all cycles when they are created, and treat them specially when they become distributed across the machine.

References

[1987] D.I. Bevan. Distributed garbage collection using reference counting. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, volume 2, pages 176–187, Eindhoven, The Netherlands, June 1987. Springer-Verlag LNCS 259.

[1983] R.J.M. Hughes. *The Design and Implementation of Programming languages.* Dphil thesis, Oxford University Computing Laboratory, Programming Research Group, July 1983. *Also* published as Technical Monograph PRG-40.

[1989] D.R. Lester. An efficient distributed garbage collection algorithm. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE Parallel Architectures and Languages Europe*, volume 1, pages 207–223. Springer-Verlag, 12–16 June 1989.

[1987] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.

[1987] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE Parallel Architectures and Languages*

Europe, volume 2, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer-Verlag LNCS 259.