Chapter 1

Introduction

The term *garbage collection* denotes a class of memory management techniques which free programmers from the chore of keeping track of and reclaiming memory. Garbage collection is used routinely in the implementation of languages such as Lisp and ML which are oriented towards symbolic computation and functional programming, respectively. Garbage collection is only infrequently used in the implementation of "general purpose" languages such as C, Pascal and PL/I.

A mechanism is known for garbage collection C applications which do not cooperate with the collector is known [1], but this mechanism is relatively expensive. This report identifies issues which arise when extending the C language [10] to support a garbage collected heap and to cooperate with the garbage collector by identifying to the collector those pointers which refer to the garbage collected heap. The report proposes modifications to the language which address these issues and which modify the syntax and semantics of C as little as possible. The proposal degrades the performance of only those expressions which manipulate references to the garbage collected heap. The proposal was partially implemented in a compiler for a C like language whose target is a virtual machine. An interpreter for the virtual machine was also implemented.

1.1 Applications

The extension of C to include a garbage collected heap is intended to increase programmer productivity by reducing the complexity of applications which use dynamically allocated memory. Access to a garbage collected heap is useful in a number of circumstances.

- In applications which manipulate complex data structures it can be difficult for programmers to determine when a region of memory can be reclaimed safely. A garbage collected language eliminates this problem by automatically reclaiming storage.

- The performance of applications which manage large numbers of simple data structures (such as character strings) can be improved with garbage collection. Rather than constantly copying these structures from place to place in the application, a programmer can pass pointers to the structures throughout the application. The garbage collector guarantees that no matter how complex the manipulations of these data structures, the storage they occupy will be reclaimed only when it is safe to do so.

- Garbage collection is useful in applications which are under development by multiple development teams and in applications which make heavy use of reuseable libraries. When dynamically allocated memory is exchanged between subsystems in these situations, it may not be clear whose responsibility it is to eventually reclaim this memory. Even when the responsibility is clear, the information may be available only in offline subsystem documentation. Programmers do not always refer to this documentation, especially when copying fragments of code from one module to another. In these situations, memory may not be freed at all or may be freed prematurely, leading to software defects which are difficult to diagnose. The use of a garbage collected heap eliminates this problem.

Chapter 2

Requirements Analysis

This chapter discusses requirements for the implementation of a garbage collected heap in a C compiler. The requirements fall into two broad categories: requirements intrinsic to various garbage collection algorithms and compatibility requirements arising from common C programming techniques. The intrinsic requirements are discussed first, in the context of a description of garbage collection algorithms.

In general, there are two kinds of garbage collection algorithms: *reference counting* and *tracing* algorithms [7]. The tracing algorithms can be further divided into *compacting* and *non-compacting* algorithms. Each of these kinds of algorithms is discussed in turn.

2.1 Reference Counting

Reference counting algorithms reserve a field in every allocated block of memory to count the number of pointers referring to the block. Every time a reference to a block is created or destroyed, the reference count for the block must be updated. If the count ever reaches zero, the block is no longer in use and can be immediately reclaimed. Reference counting algorithms are relatively easy to implement, but suffer from the performance penalty associated with constantly updating reference count fields. Another drawback of these algorithms is that they are unable to reclaim circular data structures because the reference counts in such structures never reach zero, even when the entire structure is no longer referenced by pointers outside of the structure.

Reference counting algorithms are used when the inability to reclaim circular structures is acceptable, or in conjunction with a tracing algorithm. The tracing algorithm is used as a backup for the reference counting algorithm. The backup algorithm is invoked as a last resort when a storage allocation request cannot be fulfilled, in the hope that the tracing algorithm will reclaim storage which the reference counting algorithm was unable to. Reference counting algorithms have also been proposed as optimizations to tracing algorithms to reduce the frequency with which tracing collectors are invoked [18].

In spite of their potential utility, schedule constraints prohibit the evaluation of reference counting algorithms as an implementation alternative in this report. The remainder of this report deals only with the implications of tracing garbage collection algorithms.

2.2 Tracing Algorithms

Tracing algorithms for garbage collection identify memory locations which are still in use by walking all active trees of pointers to data structures or *nodes*. As each node is encountered during the trace, it is identified as being in use, usually by turning on a *mark bit* in the node. Any node which is not encountered and marked during the tracing phase is not in use and can be reclaimed. The nature of tracing algorithms introduces constraints on any language using them. These constraints are introduced in the context of both *compacting* and *non-compacting* algorithms. Compacting algorithms move in use nodes in the garbage collected heap to a single contiguous region in the heap. Non-compacting algorithms do not do this and therefore suffer from memory *fragmentation.* Compacting collectors avoid fragmentation and simplify storage allocation as well, since unused storage is also available as a contiguous region after a collection.

2.2.1 Non-Compacting Algorithms

While memory fragmentation is undesirable, it is a characteristic of all existing memory management schemes for the C language. This being the case, it seems reasonable to assume that in terms of memory fragmentation, non-compacting garbage collectors work as well as existing memory management algorithms for C. This is not the case in general. Dynamic memory management algorithms for C make unused memory available for reuse as soon as it is identified by a `free` statement. In contrast, storage in a garbage collected heap is not available for reuse until the completion of the next garbage collection cycle. This may result in a garbage collected system unnecessarily fragmenting a large region of free memory when a manually managed heap would have reused smaller regions recently identified by `free` statements. Similar examples can be constructed where garbage collected heaps result in *less* fragmentation than do manually managed heaps. No conclusive statement can be made therefore, about the memory fragmentation performance of non-compacting garbage collection algorithms vs existing C algorithms.

If a non-compacting algorithm were used to manage a garbage collected heap for C, it would require a degree of compile time and run time support.

- Non-compacting tracing algorithms require that *every node in the garbage collected heap which is still in use be visited during the tracing phase of the collector.* To ensure this, the tracing phase does not need to chase every pointer to every node in the heap, but it must chase at least one pointer to every in use node in the heap.

- To chase pointers at all, these algorithms require that pointers to the garbage collected heap, or *garbage collected pointers* be identifiable, no matter where these pointers are stored. This is usually accomplished by either explicitly or implicitly associating data type tags with every block of memory in the system.

- Most non-compacting algorithms also require that each garbage collected pointer they chase point to the *beginning* of a garbage collected node. This is because control information such as the mark bit generally resides at the beginning of the node. While this requirement is not absolute, algorithms which allow pointers into the middle of nodes are relatively expensive [7].

2.2.2 Compacting Collectors

To ensure that an application continues to execute correctly after a compacting collection, the collector must carry out *pointer adjustment* on all pointers to nodes in the garbage collected heap. Pointer adjustment modifies all these pointers so that they refer to the new locations of the moved nodes. Compacting collectors require all of the language support which non-compacting collectors do, and strengthen one of the non-compacting requirements. To carry out pointer adjustment, compacting collectors must be able to identify and chase *every* pointer to every in use node in the heap.

2.3 Additional Requirements

Additional requirements for the implementation of a garbage collected heap for C stem from pragmatic considerations rather than from intrinsic qualities of garbage collection algorithms. The addition of a garbage collected heap is intended to enhance the C language by reducing the complexity of some programming tasks. While the addition may significantly modify the *implementation* of a C compiler, it is not intended to change significantly the syntax, semantics or performance of the language. All C applications which work correctly with a non-garbage collecting C implementation should compile and run correctly with the new implementation. Furthermore, only those parts of an application which use pointers referring to the garbage collected heap may incur a performance penalty as a result of adding a garbage collected heap to the language. This performance penalty is the expected cost of identifying garbage collected pointers to the garbage collector.

Finally, to retain the "flavour" of C as a "low level" language, a garbage collected C should provide for bypassing pointer identification mechanisms *between garbage collections.* A knowledgeable programmer should be allowed to bypass these mechanisms for performance or other purposes, provided that at the time of the next garbage collection, all pointers to the garbage collected heap are correctly identified.

2.4 Summary of Requirements

The requirements discussed in this chapter are summarized in this section.

Essential Requirements:

- Pointers to the garbage collected heap must be identified to the garbage collector. Non-compacting collectors require that at least one pointer to every garbage collected node be identified to the collector. Compacting collectors require that every reference to every node in the heap be identified to the collector.

- All pointers identified to the garbage collector must refer to the beginnings of nodes in the garbage collected heap.

Other Requirements:

- All applications which run correctly using a non-garbage collected compiler should run correctly when compiled with the enhanced compiler.

- Only those portions of an application which use the garbage collected heap may incur a performance penalty as a result of adding the garbage collected heap to the language.

- References to the garbage collected heap may be `cast` as other data types between garbage collections, provided that all such references are identified to the collector during garbage collections.

Chapter 3

Related Research

Garbage collection is usually associated with *dynamically typed* languages such as Scheme and Smalltalk, and is not usually associated with *statically typed* languages such as C and Pascal. Dynamically typed languages associate data types with *values* such as blocks of memory, while statically typed languages associate data types with *identifiers* such as variables and procedures [2]. Statically typed languages can be further classified as languages which *cooperate* with a garbage collector by identifying references to the garbage collected heap, and languages which are *uncooperative* and provide no such identification.

This chapter discusses existing implementation techniques for garbage collection in dynamically typed languages and in both classes of statically typed languages. Each technique is briefly evaluated as to whether it might be useful in a cooperative, garbage collected C. Techniques identified as useful in creating a garbage collected C are examined in more detail in the next chapter. Techniques which useful only as optimizations to garbage collected languages are not considered in the remainder of this report. The thrust of this report is the establishment of basic functionality rather than peformance optimization. The consideration of optimizations is deferred to future research which will compare the performance of the proposal in this report to the performance of Boehm's technique for uncooperative environments [1].

3.1 Dynamically Typed Languages

Dynamically typed languages can associate data types with values either *explicitly*, by prefixing every value with a small data type tag, or *implicitly* by using some other mechanism to identify values. Garbage collectors in these languages use this data type information to locate garbage collected pointers within values. This section discusses mechanisms for explicit and implicit data typing and discusses storage management mechanisms used by dynamically typed languages.

3.1.1 Explicit Data Typing

On many machines a small, explicit tag causes the components of many data structures to become non-word aligned, increasing the cost of accessing these components. Two optimizations have been developed to address this problem.

- The obvious optimization is to make the data type tag occupy an entire machine word. This solves the alignment problem, but increases the memory cost of the tag, especially for small data structures. In many applications this cost is unacceptable. Empirical results for applications written in Scheme for instance, show that as many as one half of all of memory consists of one and two word data structures [6] [5]. The addition of one word tags to these data structures increases the memory cost of these applications by 25 - 50%.
- The second optimization uses processor and memory hardware support to associate a small data type tag with every *word* in memory [12]. This tag is used to identify small, common data types such as garbage collected pointers and conventional data type tags are used for larger or less common data structures. This mechanism significantly reduces the memory cost of tags as well as the cost of type checking. The disadvantage of this optimization is that it is usually possible only in computer architectures designed for dynamically typed languages.

The C language is a statically typed language and at first glance, does not appear to require the use of data type tags. Adding a garbage collector to C however, requires that pointers to garbage collected nodes be identified to the garbage collector at run time. This identification mechanism is a kind of dynamic typing, since it distinguishes values which are pointers to the garbage collected heap from other kinds of values. Implementing this identification mechanism using hardware supported tags is possible only when designing compilers for architectures which provide this support. A more generally applicable data typing mechanism is to prefix values with type information sufficient to identify pointers to the garbage collected heap.

3.1.2 Implicit Data Typing

There are alternatives to explicit data type tags, namely *implicit* typing through either typed pointers [12] or through storing all data of a given type in a particular region of memory [17]. These alternatives are useful when optimizing an implementation, since their cost is sometimes less than that of explicit tags. These alternatives are equivalent to explicit tags though, in the sense that they also associate data type information with values. Since explicit tags are simpler than these alternatives, explicit tags are the only typing mechanism considered in this report.

```
struct x {
  int a;
  struct y b;
  struct z c;
  };
struct y {
  int a;
  int b;
  struct z c;
  };
struct z {
  int a;
  int b;
  };
```

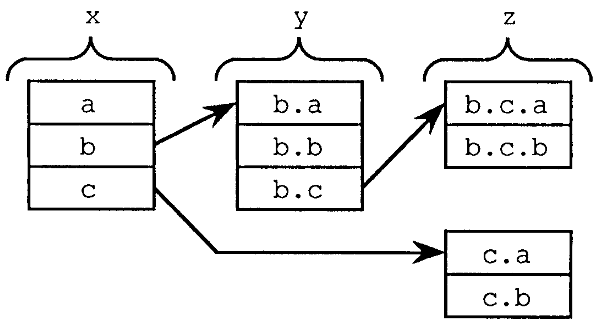

Figure 3.1: Nested Data Structures

3.1.3 Nested Data Structures

The construction of nested data structures in some dynamically typed languages and in all garbage collected languages is complicated by the fact that pointers must refer to the beginnings values. When using nested structures, programmers frequently need to make pointers refer to the nested components of a structure. This is accomplished in most dynamically typed languages by representing nested structures as multiple, linked nodes (see Figure 3.1). An exception to this rule is the Symbolics 3600 architecture, which supports both typed pointers and tagged memory [13]. This architecture allows pointers into garbage collected nodes by tagging such pointers as *locative* pointers. When the garbage collector encounters such a pointer, it chases it and then searches backwards in memory for a word tagged as the beginning of the node.

The representation for nested data structures in Figure 3.1 is very different from the representation used by C compilers. It will become clear in the next chapter that the use of this representation would make a garbage collected C incompatible with existing implementations to a large degree. The Symbolics alternative to this representation is available only on architectures which provide tagged memory. Other alternatives to this representation are proposed in the next chapter.

3.1.4 Other Storage Optimizations

Chase [3] evaluates a number of storage optimizations which have been proposed for dynamically typed languages. These optimizations eliminate the overhead of garbage collection for values which at compile time, can be proven to become garbage at a predictable point in the application. These values are on the stack or in a manually managed heap instead of in the garbage collected heap. These optimizations are not a primary means of storage management. Instead, they are optimizations which assume the existence of a variety of memory management mechanisms and identify to the compiler the least expensive mechanism which

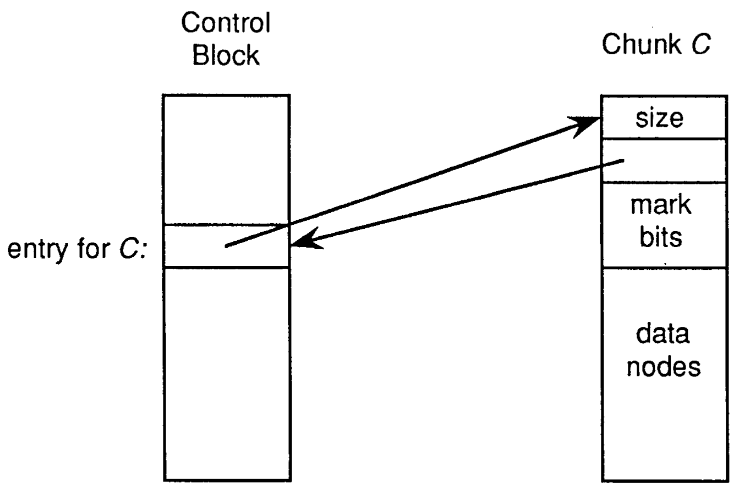

Figure 3.2: Garbage Collection in an Uncooperative Environment

can be used in a given circumstance. These optimizations may be applicable to a garbage collected C, but are not considered further in this report.

3.2 Cooperative, Statically Typed Languages

Cooperative, statically typed languages cooperate with a garbage collector to the extent that they provide run time typing information sufficient to identify references to the garbage collected heap. The only examples of these languages are the Fortran and Pascal compilers on the Symbolics 3600 machines [14]. While these languages are nominally statically typed, they associate exactly the same kind of data type information with runtime values as do dynamically typed languages, because the Symbolics hardware does this automatically. The existence of these languages demonstrates that a cooperative garbage collected C is possible. Since the implementations use dynamic typing all of the time though, they offer little insight into how to avoid the cost of dynamic typing in portions of an application which do not use garbage collected values.

3.3 Uncooperative, Statically Typed Languages

Uncooperative, statically typed languages do not associate type information with values at all. Garbage collection in such languages is particularly difficult, since one of the key requirements of all collection algorithms is not satisfied - the requirement that garbage collected pointers be identified to the collector.

A recent approach addresses uncooperative languages in a novel way [1] (see Figure 3.2). In this technique, memory in the garbage collected heap is allocated in *chunks*, each of which holds nodes of a fixed size. Chunks are always aligned on fixed power-of-two memory boundaries and contain header information indicating the size of the chunk, the size of nodes in the chunk, a mark bit for each node in the chunk and a pointer to the *heap control block entry* for the chunk. The heap control block contains a pointer to the beginning of every chunk in the garbage collected heap.

During a garbage collection, the collector scans memory and interprets every word as a pointer to the heap. The collector masks the low order bits in the pointer to make it refer to the beginning of the chunk for the node and chases the pointer to the heap control block. If this pointer actually points into the control block and if its target points back to the chunk in question, then the original word in memory actually refers to the contents of a chunk of storage in the garbage collected heap. The collector then checks the chunk control information to ensure that the pointer actually points to the beginning of a node in the chunk and if this is the case, turns on the mark bit for the node. This technique guarantees that the mark bit is turned on only in legitimate control fields at the beginning of nodes in the garbage collected heap. The technique does *not* protect against incorrectly interpreting random data as a reference to a node and turning on the mark bit for nodes which would otherwise have been reclaimed.

Boehm's technique is expensive, both because every word in memory must be scanned for potential pointers, and because random data which happens to have the same bit pattern as a pointer to a garbage collected node can keep otherwise unreferenced nodes from being reclaimed. Furthermore, the technique is incapable of employing a compacting collector because it would be unacceptable to carry out pointer adjustment on all words in memory which happen to have the same bit pattern as a pointer to a garbage collected node. In spite of these drawbacks, Boehm reports that the cost of garbage collection in an uncooperative environment is acceptable.

Chapter 4

Implementation Alternatives

This chapter discusses implementation alternatives for a cooperative, garbage collected C language. The two requirements for this effort which are most difficult to address are the requirement to identify garbage collected pointers and the requirement to support all of the operations and coding styles which the C language now supports. This chapter proposes a mechanism to address the former requirement and evaluates this mechanism using the other requirements for the garbage collected language. This chapter examines only alternatives which do not require special hardware. The C language is expected to run on all stock architectures and few such architectures support tagged memory. Assuming such support in a garbage collected C would restrict the utility of the language by restricting the kinds of machines which could run programs in the garbage collected language.

4.1 Garbage Collected Pointers

In general, the problem of predicting the path of values through variables in a program is undecidable. Compilers therefore, cannot be expected to distinguish pointer variables containing garbage collected pointers from pointer variables containing normal pointers at compile time. To make this distinction, some additional mechanism is required. There would appear to be two alternatives for identifying pointers which refer to the garbage collected heap.

- A new class of data types could be created, namely the class of *garbage collected pointers*. These pointers would be typed pointers which always refer to the garbage collected heap. The compiler could then have to somehow identify these pointers to the collector.

- All pointers in the application could be explicitly or implicitly tagged to indicate whether or not they referred to the garbage collected heap. The compiler would then identify *all* pointers to the garbage collector and the collector would decide which ones actually referred to the heap.

The advantage of the latter approach is that programmers need not be aware of which pointers do or do not refer to the garbage collected heap. The disadvantage of this approach is that the compiler must identify all pointers to the garbage collector. As will become apparent later in this chapter, this would result in a performance penalty being associated with any section of code which makes use of any pointer, not just those sections making use of garbage collected pointers. This penalty makes this alternative unacceptable.

The former approach requires somewhat more effort of programmers, but guarantees that existing code and applications suffer no performance penalty as a result of extending the language. For this reason, it is the garbage collected pointer approach which has been adopted in this report. Once identified to the compiler, the compiler must be able to identify garbage collected pointers in all of C's storage classes to the garbage collector. The remainder of this section examines mechanisms for managing scalar variables containing these pointers in existing storage classes and in the garbage collected heap. Section 4.2 deals with identifying these pointers in aggregate variables.

4.1.1 Processor Registers

Explicitly associating type information with processor registers is very expensive. Implicit typing can be associated with registers, either by dedicating some registers to contain only garbage collected pointers, or by restricting the *kinds* of garbage collected pointers which can reside in processor registers when the garbage collector is invoked. Compilers for non-compacting collectors need only guarantee that every garbage collected pointer in a processor register have a copy in memory which is identified to the garbage collector. This way, a non-compacting collector can ignore the processor registers.

Compilers using compacting collectors are more complex. Such compilers must be able to identify *all* garbage collected pointers, including those in processor registers. Compilers for machines with very few processor registers may find it too expensive to dedicate one or more registers to hold garbage collected pointers exclusively. These compilers can use knowledge of when a collector can be invoked to guarantee that no garbage collected pointers will reside in processor registers at the time of a collection. Serial garbage collectors[1] can be invoked only when the application allocates memory in the garbage collected heap. If exception handlers are prohibited from such allocations, the compiler is guaranteed that no garbage collection will take place in the time between calls to memory allocation primitives or other subroutines. Compilers for compacting collectors on serial machines can therefore store garbage collected pointers in registers as long as the contents of these registers are flushed to memory before calls to any subroutines.

[1]Applications using serial garbage collectors allocate memory in the garbage collected heap until the heap is full. The garbage collector is then invoked, and runs until all of the unreachable nodes in the heap have been reclaimed. The application is then resumed to continue processing. This report deals exclusively with serial garbage collectors.

4.1.2 Static Data

Garbage collected pointers in static storage are easily dealt with by allocating all such pointers in a separate memory segment. When the C application is linked, the linker can accumulate all such segments into a single segment for the garbage collector to search.

4.1.3 Stack Based Data

Stack based scalar variables can be dealt with either by maintaining a separate stack for garbage collected pointers or by associating sufficient type information with stack frames to identify garbage collected pointers. A separate stack for garbage collected pointers is attractive because it simplifies code emission for expressions involving these pointers. Temporary pointers can be pushed on to and popped off of the garbage collected stack as easily as other data is manipulated on existing C stacks.

A single *mixed* stack has advantages and disadvantages when compared to separate stacks. The use of *mixed* stacks is complicated by the need to store temporary pointers in specific locations within the stack to ensure that the garbage collector can find them. An additional difficulty in using a mixed stack with a compacting collector is that arguments to functions are usually passed on the stack. In a mixed stack, arguments which are garbage collected pointers must either be identified as such by control information in the called function's stack frame, or these arguments must be copied into secure locations within the called function's stack frame. On the other hand, the use of a mixed stack may be justified on machines with so few processor registers that dedicating a register to hold the garbage collected stack pointer is out of the question.

Mechanisms for identifying garbage collected pointers in a mixed stack frame are exactly the mechanisms used to identify these pointers in data structures, since any stack frame can be viewed as a data structure. These mechanisms are discussed in Section 4.2.

4.1.4 Dynamically Managed Heap

Most dynamic memory management systems embed control information between dynamically allocated blocks [11]. This information usually describes the size of the current block, the size of the previous block and the allocation status (busy or free) of the current block. Both sizes are needed to find the beginnings of adjacent blocks when the application frees a block. A side effect of this control information is that it can be used to traverse all dynamically allocated blocks. If control information is associated with each block indicating the locations of garbage collected pointers within the block, the garbage collector can find all garbage collected pointers in a dynamic heap by traversing the heap.

The kind of control information necessary to identify garbage collected pointers in a block is discussed in the Section 4.2.

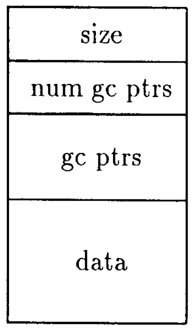

Figure 4.1: Node Structure for Dynamically Typed Languages

4.1.5 The Garbage Collected Heap

All garbage collection algorithms require that nodes in the garbage collected heap be associated with control information indicating each node's size. This information can be used to scan garbage collected heaps as well. Not all garbage collection algorithms need to scan the heap, but all need to chase garbage collected pointers contained by in use nodes in the heap. This chasing is carried out in the marking phase of tracing collectors, by chasing all such pointers when a node is marked. The locations of garbage collected pointers in nodes in the garbage collected heap is encoded using the techniques of the next section.

4.2 Data Structures and Nested Data Structures

This section of the report reviews the mechanisms for identifying garbage collected pointers in dynamically typed languages. This mechanism is found to be unfit for the garbage collected C compiler and an alternative mechanism is proposed.

4.2.1 Dynamically Typed Languages

The structure for control information identifying garbage collected pointers which was used by Cohen and Nicolau [8] is sufficient for almost all garbage collection algorithms for dynamically typed languages (see Figure 4.1). This structure requires that all garbage collected pointers be moved into a contiguous region at the beginning of each node and offers only limited support for nested data structures. In the sections which follow, this format for control information is referred to as the *standard* node format. The *mark* bit in the standard format is the sign of the `size` field. A positive `size` is unmarked and a negative `size` indicates a marked node.

Moving garbage collected pointers into a contiguous region can easily be accomplished in a compiler in a manner which is largely transparent to programmers. Such movement

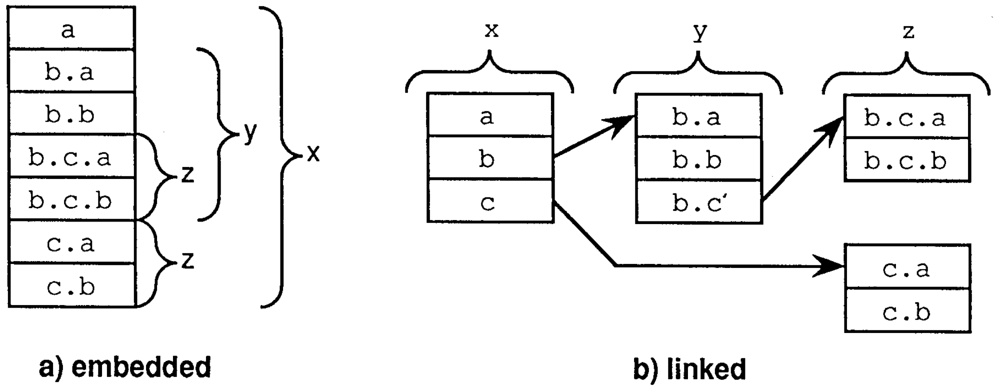

Figure 4.2: Two Implementations for Nested Data Structures

however, disregards the convention that C compilers preserve the order of components of a data structure. Preserving this ordering is important if programmers using the garbage collected compiler are to be able to link their programs with subroutine libraries designed for "normal" C compilers. Preserving the order of components is also important when the application in question is a memory-mapped device driver and the C programmer hopes to overlay a data structure on device registers through judicious pointer-wizardry.

It is not clear that these objections to pointer movement are sufficient to condemn the technique. It could be argued that it is extremely unlikely that data structures containing garbage collected pointers would be overlaid on device registers or passed to foreign language libraries. It could also be argued that on those occasions, the most that pointer movement would cost the programmer is the cost of copying the data structure to an equivalent one which contains no garbage collected pointers.

The standard node format supports nested data structures in the sense of Figure 4.2(a), but pointer movement makes obtaining any pointer to a nested structure impossible with this format, even between garbage collections. When there is only a single region for garbage collected pointers in a node, pointers from the main structure are intermixed with pointers from nested structures. This means that nested structures which contain both garbage collected pointers and other data types no longer reside in a contiguous region of storage and treatment of the nested structures as nodes themselves is impossible. This report refers to the representation for data structures in Figure 4.2a as an *embedded* representation.

The representation for nested structures in Figure 4.2b is referred to as a *linked* representation. Obtaining a pointer to a nested structure using a linked representation is possible using the standard node format. While well suited to garbage collection the linked representation is an even sharper deviation from C implementation practice than is the pointer movement technique. Nested structures using a linked representation are more costly to construct than normal C structures and do not support to the use of binary copying and initialization procedures. An attempt to copy all components of a linked structure to another

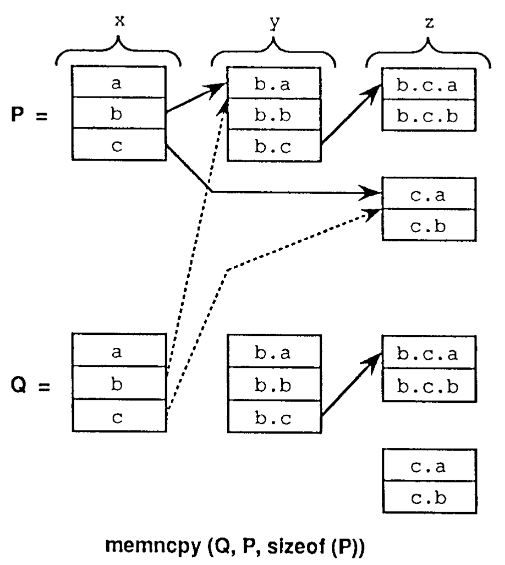

Figure 4.3: memncpy Applied to a Nested, Linked Structure

structure with memncpy or any other binary copying procedure would succeed in copying the upper-most level of information and would succeed in linking the upper-most structure to the original structure's nested components (see Figure 4.3). This modification of nested structure after the application of a binary operation is a substantial deviation from accepted C programming practice.

In summary, the pointer movement required by the standard node format may be acceptable in a garbage collected C compiler. The treatment of nested structures using this node format is not acceptable.

4.2.2 Alternate Node Formats

Figure 4.4 illustrates an alternate format for control information in data structures which avoids the problems of the standard format[2]. This format assumes an embedded representation for nested structures and associates a relatively verbose description of a data structure with each instance of the structure. Since applications tend to contain fewer data types than instances, the description of each data structure is stored in static storage and is associated with instances of the type as a pointer which preceeds the first word in the instance. The control information is just a tag for each word in the structure indicating whether or not the word is a garbage collected pointer. This representation is simple, but can become costly when applied to large arrays of data structures or to variable length arrays.

The information in the tags in this format could be compressed by counting consecutive

[2]The @ characters in the figure indicate garbage collected pointers in the same way that * characters do in normal C. A new character is required for garbage collected pointers to distinguish these pointers from pointers to other kinds of data.

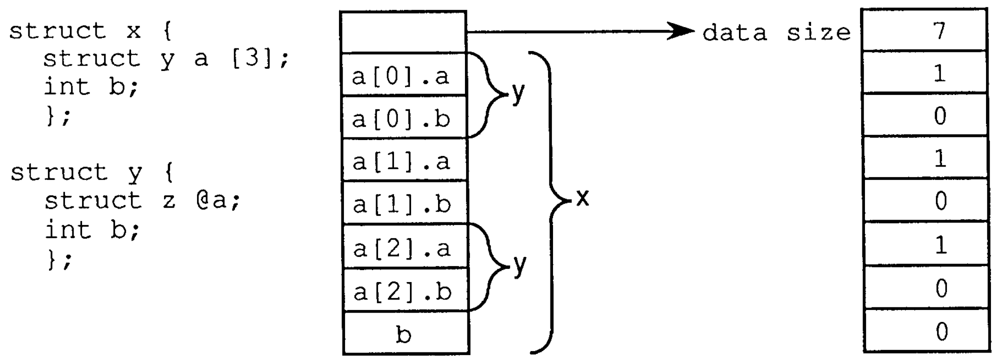

Figure 4.4: An Alternative Node Format

garbage collected pointers and counting the distance between blocks of garbage collected pointers[3] Arrays of data structures may be better addressed by adding the concept of a *multiplier* block which indicates that a subset of the control information applies repeatedly. For example, a compressed format with multiplier blocks could be organized like this:

```
<control info> = <dsize> <cisize> <sblock> ... <sblock>
      <sblock> = <multiplier> <msize> <sblock> ... <sblock>
               = <sbsize> <gcblock> ... <gcblock>
     <gcblock> = <num_gc_ptrs> <num_words_between_gc_ptrs>

       <dsize> = size of data portion of data structure
      <cisize> = size of control information block
  <multiplier> = number of times to repeat a sequence of sub-blocks
       <msize> = size of control info region affected by multiplier
      <sbsize> = size of the sub-block
```

In this representation, the control information block consists of the size of the data structure, the size of the control information block and a number of sub-blocks. Each sub-block is either a size field and a number of pairs indicating blocks of garbage collected pointers, or a multiplier field affecting a number of nested sub-blocks.

Variable sized arrays, while not explicitly allowed in C, are frequently used by C programmers. When they are used, they may appear by themselves, or they may be the last element of a data structure. This is accomplished by allocating an array of unary length and dynamically allocating enough memory to hold the array of desired length. References beyond the unary-length array are detected by almost no compilers and the net effect is that of a variable length array. These arrays can be accommodated in either the compressed or uncompressed format by recognizing data structures which have the potential to become variable length arrays and flagging the control information for these structures as being variable length arrays. Upon encountering such nodes, the garbage collector would simply continue repeating the multiplier for the variable length element in the structure until all memory in the structure had been examined.

[3]This is similar to the standard node format for dynamically typed languages.

This format for control information deals with the problems identified with the standard node format.

- Pointer movement in the alternate format takes place only within each nested data structure. In fact if no pointer movement is desired, the uncompressed control information format is not affected but the compressed format may grow slightly longer.

- Pointers to nested structures are possible because nested structures are located in contiguous blocks of memory.

- If garbage collected pointers to nested structures are desired, the top level structure and each nested structure can be prefixed with an offset indicating the location of the beginning of the node. This representation withstands the application of binary manipulations like `memncpy`, since offsets can safely be copied between structures. The representation fails though, when `memset` is used to initialize a structure, since the binary initialization destroys the offsets within the structure.

The one drawback of the new format is that it does not deal very well with very small data structures, because the pointer preceeding the data structure causes a relatively large increase in the size of the structure. To an extent, this is a problem with all formats for control information on stock architectures. The problem can be addressed by using small data type tags and incurring the associated memory alignment performance penalty.

4.2.3 Unions

The proposed compiler needs to treat unions specially only when the unions contain garbage collected pointers. When this is the case, the alternate format must be modified as in Figure 4.5. Each union containing a garbage collected pointer must be prefixed with a pointer to the current data type of the union. The data type information for the entire node specifies only the locations of these pointers within the node. The compiler must emit code to set the data type pointer for unions containing garbage collected pointers every time an assignment is made to a member of the union. This may result in a significant performance penalty being associated with the use of unions containing garbage collected pointers.

The C language does not define the result of a reference to a member of a union when the union was written to last through a *different* member. It seems fair therefore, to instruct the garbage collector to interpret the union as an instance of the last member which was written to. C programmers are already aware of the potential for abuse of unions and this potential is unfortunately increased by the addition of garbage collection to C. Perhaps the performance penalty associated with the use of unions containing garbage collected pointers will make programmers less willing to use this error prone construct.

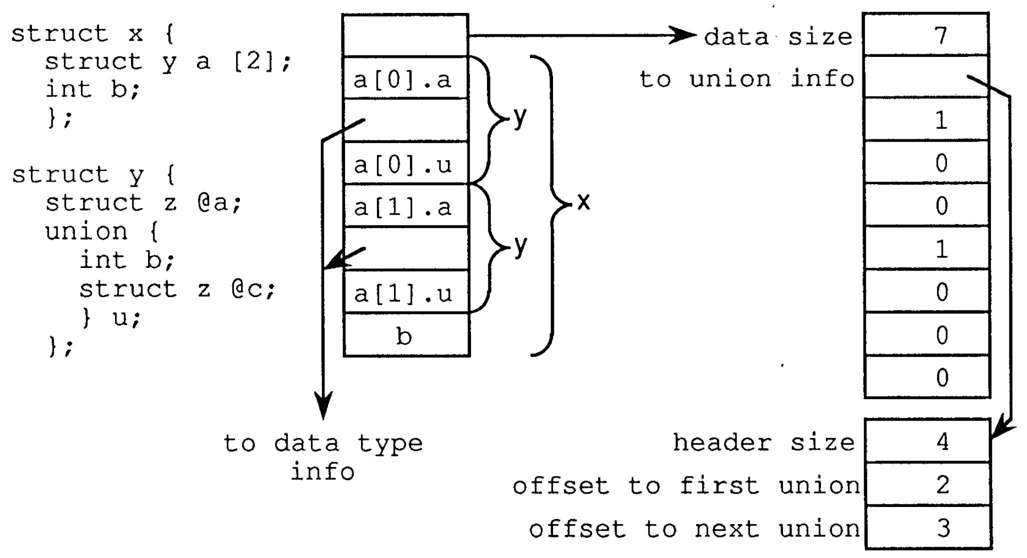

Figure 4.5: An Uncompressed Node Format for Unions

4.3 Allocation and Deallocation

Implicit in this discussion of associating type information with values is the assumption that the association takes place when memory is allocated. This prohibits the use of user-replaceable procedures to manage memory because the user procedures would have no access to the type information required by the garbage collector. Allocation and deallocation of memory in any garbage collected language *must be the responsibility of the language* so that the compiler can emit code to guarantee the initialization of data type information in allocated nodes. This is a significant deviation from existing C compilers.

On the other hand, data type information is required only for data structures which contain garbage collected pointers. This means that the garbage collected C compiler will still correctly compile all existing C programs. In addition, C programmers are already becoming somewhat accustomed to the demise of `malloc` by C's association with C++ [16]. The object-oriented C++ contains no `malloc` either, forcing users to use the language operators `new` and `delete` in place of C's `malloc` and `free` functions. C++ also contains restrictions on the unrestricted use of overlays, since it is impossible to overlay classes.

This report proposes that two new operators be added to the C language: the `new` and `gcnew` operators. These operators are equivalent to the `new` operator in C++ and would allocate memory in dynamically allocated and garbage collected heaps respectively. In contrast to C++, this report proposes that `malloc` be retained for compatibility with older applications. These applications will work correctly using `malloc`, provided that they do *not* use the procedure to allocate data structures containing garbage collected pointers.

If it is deemed essential to support user replaceable memory managers and allow them to overlay data structures containing garbage collected pointers, the language could be modified to support this. Such overlays could be supported by *registering* values with the garbage collector garbage collector through a number of language-defined garbage collection support procedures. The desirability of these mechanisms is questionable though, since they require that programmers be unduly familiar with the details of garbage collection mechanisms in the language.

There is an additional cost associated with memory allocation for data structures containing garbage collected pointers. All of the garbage collected pointers in each newly allocated structure must be initialized to be NULL. This is necessary to prevent the garbage collector from interpreting the uninitialized fields as pointers to the garbage collected heap. This performance penalty is incurred by all dynamic and garbage collected heap allocations for values containing garbage collected pointers *and for the creation of stack frames containing such pointers.* The creation of a stack frame is a kind of memory allocation and is subject to all of the limitations of other kinds of allocation.

4.4 Bypassing Identification Mechanisms

The C language has always allowed programmers to bypass data typing mechanisms through the use of *data type casting* and through relaxed data type checking on some assignments and procedure calls. These mechanisms may still be used in the proposed language, but relaxations which affect pointers to the garbage collected heap are required to print warnings for the programmer. Programmers must be aware that relaxing data type identification mechanisms for garbage collected pointers is safe only *between* garbage collections. At the time of a garbage collection, all garbage collected pointers must be identified as such to the garbage collector.

Chapter 5

Evaluation of the Language Proposal

This chapter evaluates the proposal for a garbage collected C language against the requirements set out in Chapter 2. This report does not define all aspects of a cooperative, garbage collected language. Instead, it is a first attempt to identify and address significant design issues in such a language. The proposal and therefore this evaluation is qualitative and undoubtedly incomplete.

The proposed implementation of a garbage collected C compiler addresses the requirements outlined in Chapter 2 as follows:

- Garbage collected pointers are identified to the compiler by programmer declarations. The compiler identifies these pointers to the garbage collector by associating data type information with each instance of each data type containing a garbage collected pointer.

- Garbage collected pointers to components of a garbage collected node can be supported by the alternate node formats, but at the cost of some compatibility with C.

- All applications which ran correctly in a non-garbage collected language implementation should run correctly in the proposed implementation.

- The proposed implementation would correctly compile all existing applications and these applications would suffer no performance penalty as a result of the addition of garbage collection to the language. Furthermore, only portions of code which use garbage collected data types will incur a performance penalty as a result of the addition of garbage collection to the language.

- In the proposed implementation, garbage collected pointers can be *cast* as other data types between garbage collections, provided that all garbage collected pointers are identified to the collector during garbage collections.

- Unions containing garbage collected pointers can be accommodated by the proposed implementation, but they are more expensive and more error prone than are unions which contain no garbage collected pointers.

In summary, the proposal enhances the C language with garbage collection with few performance penalties but at some cost of compatibility with accepted C programming practice. The most costly performance penalties are associated with the use of unions the creation of data structures, including stack frames, which contain garbage collected pointers. The most glaring language incompatibilities arise when programmers use overlays which contain garbage collected pointers. The frequency with which such overlays are required may be reduced somewhat by the addition to the language of the `new` and `gcnew` operators.

Chapter 6

The INF Virtual Machine Interpreter

This chapter and Chapter 7 discuss a partial implementation of the language proposal. Schedule constraints prohibited the implementation of a true C compiler for a real machine. Instead, the author implemented a compiler for a C-like language called X which is targeted for a stack-based virtual machine. An interpreter called INF[1] was implemented for the virtual machine and the implementation closely resembles a FORTH [9] interpreter. This chapter describes the INF virtual machine and its implementation.

6.1 The Virtual Machine

The INF virtual machine consists of several stacks managed by a number of machine registers.

- A *garbage collected stack* holds garbage collected pointers exclusively and is managed by a stack pointer and a frame pointer.
- A *main stack* holds all other data types and holds most operands for stack-based instructions. This stack is also managed by a stack pointer and a frame pointer.
- A *control stack* contains return addresses for procedure calls and is managed by only a stack pointer.
- A program counter register is accessible to the programmer through control transfer instructions.

The instruction architecture is relatively simple and supports four addressing modes:

- a register offset mode specifies a constant offset from a register,
- a stack indirect offset specifies a constant offset from a pointer in the top word of the main stack,

[1]INF is Not FORTH.

- an absolute addressing mode supports static variables and
- an immediate addressing mode provides for in-line constants for some instructions.

The immediate addressing mode is useable only in a few special instructions, but any instruction which requires an operand may use the other addressing modes. The offsets in the addressing modes are usually interpreted as *word* offsets and words are four bytes long. Two byte manipulation instructions, `loadb` and `storeb` are exceptions to this rule and interpret offsets as byte offsets. When instructions use operands on a stack, the instructions pop these operands off of the stack before pushing the instruction result (if any) back on to the stack.

This instruction set does not correspond to any real processor, but was chosen, so that it *could* correspond to a simple, real processor. The instruction set was chosen with two purposes in mind:

- The set is intended to convince the reader that writing a garbage collected C compiler for a real machine can be accomplished and
- it was chosen to make writing the X compiler as simple as possible.

Were the instruction set to be used for an actual processor, it would have to be significantly extended. For instance, the current instruction set does not support floating point operations and provides limited support for common logical operations and array indexing.

Instructions are preceeded by their operands in INF source code. This syntax is the reverse of what most assemblers use and the reason for this will be made clear in the next section. The instructions supported by INF are listed in the table below. In this table, *any* indicates that any addressing mode can be used with the instruction and *immed* indicates that only an immediate addressing mode can be used.

	Stack Operands	Instruction Operands	Instruction	Comment
Load and Store		*any*	load	push operand on to main stack
			gcload	push operand on to garbage collected stack
	value	*any*	store	store *value* into operand
	gcvalue	*any*	gcstore	store *gcvalue* on gc stack into operand
		immed	ldconst	push operand on to main stack
		any	ldaddr	push address of operand on main stack
		any	loadb	push byte operand on to main stack
	value	*any*	storeb	store value into operand
Stack		*immed*	pop	pop words off the main stack
		immed	gcpop	pop words off the gc stack
		immed	link	create a stack frame on main stack
			unlink	discard a stack frame from main stack
		immed	gclink	create a stack frame on gc stack
			gcunlink	discard a stack frame from gc stack
	value		unlink/propagate	unlink main stack and propagate return value
	value		gcunlink/propagate	unlink gc stack and propagate return value
	value	*immed*	propagate	move a value *immed* words down main stack
	value	*immed*	gcpropagate	move a value *immed* words down gc stack
Flow of Control		*label*	call	call a procedure
			return	return from a procedure
		label	goto	unconditional transfer of control
	value	*label*	beq	branch to label if *value* = zero
	value	*label*	bne	branch to label if *value* != zero
	value	*label*	ble	branch to label if *value* <= zero
	value	*label*	bge	branch to label if *value* >= zero
	value	*label*	blt	branch to label if *value* < zero
	value	*label*	bgt	branch to label if *value* > zero
Arith.	*2 values*		+	push sum of values on stack
	2 values		-	push difference of values on stack
	value		negate	push inverse of value on stack
	2 values		*	push product of values on stack
	2 values		/	push quotient of values on stack
			BREAK	signal a breakpoint to the debugger

In addition to these instructions, the interpreter understands six *meta-instructions*:

proc(name inst1 ... instN)	defines a procedure
label name	defines a label within a procedure
variable name	defines a static variable one word long
gcvariable name	defines a static variable containing a collected pointer
constant name value	defines a named constant
include filename	loads an INF source file into the interpreter

One advantage of writing a compiler for a virtual rather than a real machine is that the compiler author can choose the target machine's instruction set. Another advantage is that debugging facilities can be implemented very easily in the virtual machine. When the INF interpreter is run interactively, the following debugging commands are available in addition to instructions and meta-instructions:

		.s	display the main stack
		.gs	display the garbage collected stack
		.cs	display the control stack
		regs	display the processor registers
from	*count*	dump	display from *from* for *count* words
		statistics	display memory usage statistics
	name	disassemble_word	disassemble procedure *name*
	addr	disassemble	disassemble procedure at *addr*
	addr	set_breakpoint	set breakpoint at *addr*
	addr	clear_breakpoint	clear breakpoint at *addr*
		list_breakpoints	list the active breakpoint
		step	single step into a procedures
		STEP	single step through procedures
		continue	continue execution after a breakpoint

Finally, the interpreter implements a number of runtime support procedures for the INF machine. Were INF a real microprocessor, these procedures would be written in X or in some other high level language. The support procedures include support for memory management (including the garbage collector) and some simple I/O procedures.

6.2 The Implementation

The INF interpreter is written in C and was inspired by the TILE FORTH [15] interpreter. The two data structures central to INF are the dictionary and the threaded space (see Figure 6.1).

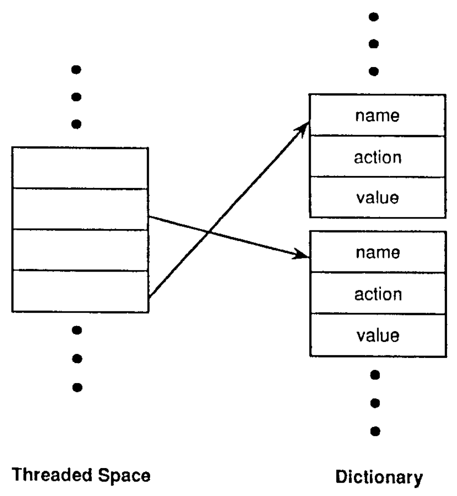

Figure 6.1: The INF Dictionary and Threaded Space

6.2.1 The Dictionary

The dictionary is an array of three word *entries.* The *name* field of each entry is a pointer to the character string name of the entry. The *action* field is a pointer to the C procedure which implements the entry and the *value* field is the value associated with the entry. Each named entity in INF (each instruction, procedure, label, variable and constant) has an entry in the dictionary. An INF entity is *executed* by the interpreter by calling the procedure in the action field of the entry.

- In entries for instructions, meta-instructions and debugging facilities, the action field points to a procedure which carries out the named operation,
- in variables and garbage collected variables, this field points to a procedure which places the *address* of the value field on the stack,
- in constants and labels the action field points to a procedure which places the *contents* of the value field on the stack, and
- in INF procedures, the action field points to a procedure which pushes the current program counter register on the control stack and replaces the program counter with the contents of the entry's value field.

6.2.2 Threaded Space

Threaded space is a large array of pointers to entries in the dictionary. INF procedures are implemented as subsets of threaded space. The value field of every procedure entry in the dictionary points to a location in threaded space. When the procedure is invoked, this pointer is loaded into the program counter register, and the INF interpreter proceeds to execute the subset of threaded space which is the INF procedure.

6.2.3 Virtual Machine Emulation

The description of the INF interpreter thus far has not mentioned any emulation of machine registers or addressing modes. These elements of INF are supported only indirectly by the interpreter. For instance, when INF executes an instruction like:

```
fp|-1 load
```

it sees the instruction as three separate pointers in threaded space. The first pointer points to the dictionary entry for the `fp` constant. When this entry is executed, the value field of the constant is pushed on the stack. This value is the address of the static C variable which is the stack frame register for the main stack. When INF executes the second entry, it pushes a zero word on the main stack and when INF executes the last entry, it calls the procedure for the `load` instruction. This procedure adds the top two words on the stack together and uses the result as a pointer to fetch a 4 byte word from memory. The top two words on the stack are discarded and the 4 byte result is pushed on to the stack. The reason for the "reversed" operand/instruction syntax is now clear. INF is a reverse polish notation calculator and needs to see the operands first in order to push them on to the stack for the instruction.

Other instructions and addressing modes are implemented using similar techniques. The `ldconst` instruction for example, is a no-op. In an instruction such as:

```
23 ldconst
```

INF loads the constant 23 on to the stack when the constant is executed. The `call` instruction is a nop as well. By the time INF encounters the `call` instruction in a sequence like

```
printf call
```

the `printf` procedure has already been executed.

6.3 Garbage Collection

The INF garbage collector is a two-space copying, compacting collector [4]. A compacting collector was chosen for the project because these collectors required the identification of all garbage collected pointers. This is a more difficult task than identifying at least one pointer to every in use node as a non-compacting collector requires. The more difficult collector was implemented to increase confidence in the ability of the C language to cooperate with a garbage collector.

The collector is part of the runtime support implemented as part of the INF interpreter, but could have been implemented in INF or in X as well. The collector is invoked automatically every time a memory allocation request in the garbage collected heap cannot be serviced. The collector can be invoked manually by calling the gc procedure.

Chapter 7

The X Compiler

The objective of this report was to extend the C language to cooperate with a garbage collector. Writing a full fledged lexical scanner, parser and code emitter for C however, was felt to be too time consuming an effort for a small term project. Instead, the new language X was invented with the semantics of C, but a simpler syntax.

The syntax of X is a parenthesized polish notation based on lexical structures recognized by the Scheme `read` primitive. The X compiler was written in Scheme, using `read` to eliminate the need for a lexical scanner. The polish notation was intended to eliminate the need for a parser, but only partially succeeded in dong so. The syntax checking component of a parser was still required by X to verify the syntax of some complex statements such as declarations.

7.1 Syntax

Figure 7.1 is an example of a recursive fibonacci function written in X. The diagram illustrates how X declarations are modelled after the public domain CDECL tool which maps cryptic C declarations into structured english. Garbage collected pointers (which are not used in the example in Figure 7.1) are declared by prefixing any `ptr` declaration with gc. For example, a pointer to a function returning a garbage collected pointer would be declared:

```
(dcl fibonacci as function ((int)) returning int)
(function fibonacci (n)
  (if (< 1 n)
      (return (+ (fibonacci (- n 1))
                 (fibonacci (- n 2))))
      (return n)))
```

Figure 7.1: The Fibonacci Function in X

```
(dcl boo as ptr to function of () returning gc ptr to int)
```

Garbage collected pointers are used in X just as any pointer would be used in C. Some other elements of syntax may not be obvious:

- Procedures must be declared before they are defined.
- Assignment uses the `set!` operator instead of the `=` operator.
- If/then/else clauses must always contain an else clause.
- All variables and data structures must be declared before they are used.

The syntax of language features like arrays, unions and indirection is not defined for X because there was no time to implement these features.

7.2 The Implementation

A compiler implementation supporting garbage collection, local variables, procedures, recursion, if/then/else, assignment, integers, pointers, character strings, casting and data structures was constructed comparatively easily. The entire compiler consists of 900 lines of Scheme, excluding comments and empty lines. Symbol table support for data types was particularly straightforward. The type of a variable is simply the `cdddr` of its declaration and two variables are of the same type if their types are `equal?`.

Data structures were implemented as the uncompressed type tags discussed in Section 4.2.2. Space for these tags was allocated in all nodes in the dynamically managed and garbage collected heaps. The memory allocation primitives initialized this space with a `NULL` pointer, and code emitted for `new` and `gcnew` operations made these tag fields refer to compiler constants indicating the locations of garbage collected pointers. The current implementation of the X compiler always emits a type tag, even for data structures which contain no garbage collected pointers. This overhead affects slightly the performance of applications which make no use of the garbage collected heap. This performance penalty can be avoided by a slightly more intelligent implementation, since the INF garbage collector ignores nodes whose tag field is `NULL`. An even more intelligent implementation would define two entry points to the dynamic memory manager software, one to allocate nodes with the tag field and one without the field.

Stack frames in the implementation could have been cleared by code emitted by the X compiler, but were not. Instead, the INF interpreter's `link` and `gclink` instructions were modified to clear the stack frame they created. While this affected slightly the performance of all procedures, a real compiler would not always have the alternative of modifying the target machine's instruction set. A real compiler would emit code to clear only those stack frames which contained garbage collected pointers.

Chapter 8

Conclusions

The objective of this report was to address design issues associated with the creation of a cooperative, garbage collected C language. This objective has largely been accomplished by examining requirements for such a language, proposing implementation techniques suitable for implementing the language and implementing a portion of the proposed language. The main conclusion of this report is that a garbage collected C is possible and from a programmer's viewpoint, differs very little from existing implementations of the language. At the implementation level, significant differences do exist, especially in the treatment of data structures and unions containing garbage collected pointers. It appears that even in statically typed languages, a degree of dynamic typing is required for the language to cooperate with a garbage collector.

8.1 Future Research

Whether or not implementations using this proposal perform better than those using Boehm's technique is an open question. Boehm's technique appears to be faster than this proposal between collections, since between collections C applications incur only the performance penalty of maintaining the garbage collected heap's control segment. During garbage collection, Boehm's technique appears to perform substantially worse than would the proposal in this report, since it must examine more potential pointers and since the cost of examining each pointer appears much higher than the cost in a cooperative implementation.

Future research in this area could address two areas of concern:

- Known storage and data type tagging optimizations could be applied to this proposal and to Boehm's technique. The optimized approaches could then be compared to determine which performs better.

- Additional thought could be given to improving the compatibility of a cooperative, garbage collected C with existing implementations of the language. It may be that the creation of a new class of data types and the creation of new storage allocation operators could be avoided.

In addition, some experimentation with the existing X implementation is in order to determine the impact of the garbage collected pointer data type on the complexity of programming tasks. It could be that identifying these pointers with a separate data type is in fact a boon to programmers, since programming styles for garbage collected values may differ from those for non-collected values.

8.2 Lessons Learned

A number of additional lessons were learned throughout the design and implementation of INF and X, only some of which were directly related to the addition of garbage collection to C.

- The use of a separate stack for garbage collected pointers was not worth while. The two stacks complicated the compiler and did not eliminate the need for a mixed stack, since data structures containing garbage collected pointers were allocated on the main stack.
- The use of the INF interpreter to emulate separate compilation was clumsy because the names of character string constants were global and sometimes conflicted.
- The use of the INF virtual machine may have been a poor choice entirely. It may have been faster to have the compiler emit assembler source code for a real architecture and to use the development tools for that architecture than to reinvent these tools for the INF environment.
- The choice of the Scheme language and syntax for the compiler and the X language was well made, since the resultant compiler was simple and small.
- An attempt was made to implement tail recursion in X, but the attempt was not successful. Tail recursion in a language which passes parameters on the stack is unsafe because there is no assurance that any given procedure is invoked with the correct number of arguments. An attempt to remove non-existent arguments from the stack during tail recursion results in spectacular failures.

Bibliography

[1] Boehm, Hans-Juergen *Garbage Collection in an Uncooperative Environment* Software - Practice and Experience, Vol 18, No 9, Sep 1988, pp: 807-820

[2] Cardelli, Luca, and Wegner, Peter *On Understanding Types, Data Abstraction and Polymorphism* Computing Surveys, JVol 17, No 4, Dec 1985, pp: 471-522

[3] Chase, David R. *Safety Considerations for Storage Allocation Optimizations* Proceedings of the SIGPLAn '87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices Vol 22, No 7, Jul 1987, pp: 1-10

[4] Cheney, C. J., *A Nonrecursive List Compacting Algorithm*, Communications of the ACM, Vol 13, No 11, Nov 1970, pp: 677-678

[5] Clark, Douglas W. and Green, C. Cordell, *An Empirical Study of List Structure in Lisp*, Communications of the ACM, Vol 20, No 2, Sep 1977, pp: 78-87

[6] Clark, Douglas W., and Green, C. Cordell, *A Note on Shared List Structure in LISP*, Information Processing Letters, Vol 7, No 6, Oct 1978, pp: 312-314

[7] Cohen, Jaques, *Garbage Collection of Linked Data Structures*, ACM Computing Surveys, Vol 13, No 3, Sep 1981, pp 341-367

[8] Cohen, Jaques and Nicolau, Alexandru, *Comparison of Compacting Algorithms for Garbage Collection*, ACM Transactions on Programming Languages and Systems, Vol 5, No 4, Oct 1983, pp: 532-553

[9] James, John S. *FORTH for Microcomputers* SIGPLAN Notices, Vol 13, No 10, Oct 1978, pp: 33-39

[10] Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[11] Knuth, Donald E., *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, Addison - Wesley, Reading, Mass., 1973

[12] Moon, David A., *Garbage Collection in a Large Lisp System*, ACM Symposium on LISP and Functional Programming, Aug 1984, pp: 235-246

[13] Moon, David A. *Symbolics Architecture* Computer, Jan 1987, pp: 43-52

[14] Moon, David A. *Architecture of the Symbolics 3600* The 12th Annual International Symposium on Computer Architectures, SIGARCH Newsletter, Vol 13, No 3, Jun 1985, pp: 76-83

[15] Patel, Mikael R.K. *Threaded Interpretive Language Environment (TILE)* Department of Computer and Information Science, Linkoping University, Sweden, 1989

[16] Soustrup, Bjarne *The C++ Programming Language* Addison-Wesley, Reading, Massachussets, 1987

[17] Steele, Guy Lewis Jr., *Data Representations in PDP-10 MacLisp*, Proceedings of the 1977 MACSYMA User's Conference. NASA Scientific Technical Information Office, Washington, D.C. Jul, 1977

[18] Wise, David S. and Friedman, Daniel P., *The One-Bit Reference Count*, BIT Vol 17, No 4, pp: 351-359