The Non-stop Spineless Tagless G-machine
Draft

R.L. While*
Dept. of Computer Science,
University of Western Australia,
Nedlands,
Perth,
Western Australia 6009
email: lyndon@cs.uwa.edu.au

A.J. Field
Dept. of Computing,
Imperial College,
180 Queen's Gate,
London
SW7 2BZ
email: ajf@doc.ic.ac.uk

August 27, 1996

Abstract

We describe a technique for incorporating Baker's incremental garbage collection algorithm into the Spineless Tagless G-machine on stock hardware. This algorithm eliminates the stop/go execution associated with bulk copying collection algorithms, allowing the system to place an upper bound on the time taken to perform a store operation. The implementation is based on the manipulation of code-pointers and is considerably more efficient than previous software implementations of Baker's algorithm.

1 Introduction

A significant drawback of automatic garbage collection is its detrimental effect on the responsiveness of a program. While the system is collecting garbage it is unavailable to user programs for useful work, and this can cause a program to appear 'dead' for extended periods of time. This problem is caused largely by the fact that garbage collection algorithms typically collect all of the system's free store in one go, an operation that takes time at least proportional to the number of live objects in the system[1]. The problem has been exacerbated by the tendency towards larger memories and it has precluded the use of garbage-collected programming languages for many interactive or real-time applications.

An algorithm that avoids this problem, at least in the context of copying collection schemes[6], is Baker's algorithm[4]. The philosophy behind this algorithm is to amortise the overhead of a garbage collection across the set of allocation requests issued by the user program (the *mutator*), in essence performing a series of small 'garbage collections' rather than one major collection. When the system is unable to satisfy an allocation request, it performs the minimum amount of work necessary to generate some free store and immediately resumes the mutator. 'The minimum amount of work necessary to generate some free store' involves scavenging the roots of the computation and the work-space of the

*Part of this work was done whilst an employee of INMOS Ltd.
[1]More for some algorithms.

mutator: the remainder of to-space is then available to be allocated, while from-space is condemned. At each subsequent allocation the system traces a few more objects before returning to the mutator. The number of objects traced at each allocation (the *mark-ratio*) is set to ensure that the garbage collection is completed before the free store is exhausted again. An upper bound can then be placed on the time taken for an allocation request to be served by the system. In almost all cases, the slowest allocation is the original one that failed, setting off the collection.

It is also necessary to introduce a mechanism to prevent pointers to old copies of objects being propagated by the mutator while a garbage collection is in progress. This involves checking each pointer-load performed by the mutator to see if the pointer refers to a part of the store that has been made obsolete by the garbage collection (i.e. from-space): if it does, the object pointed-to is traced and the pointer is updated to refer to the new copy. This mechanism is known as the *read-barrier*. Implementing the read-barrier gives the mutator the same view of memory as it would have if the entire garbage collection was performed in one go: the mutator essentially executes in to-space, with from-space pointers being finessed before they can be de-referenced.

A read-barrier implementation also offers a further advantage to the system[7][9]. With a read-barrier, objects are traced by two different mechanisms during a garbage collection. Active objects are traced by the mutator when it attempts to access them and trips over the barrier: passive (though of course live) objects are traced by the garbage collector itself when it is triggered by an allocation request. Thus active objects are naturally distinguished from passive objects and we can improve the dynamic locality of a program by grouping them together. This can reduce the paging costs of a program dramatically and can also improve its cache performance.

The principal disadvantage of Baker's algorithm is the CPU overhead of checking every pointer-load in the program. There have been three main approaches to implementing the read-barrier.

In software This involves inserting extra instructions around each pointer-load in the program. It carries a typical overhead of 40-50% of the execution time of a program[16].

Using virtual-memory [1] This approach uses the machine's virtual memory protection mechanism to lock down obsolete copies of objects. Any attempt to access such an object causes a trap to the run-time system, the object is traced, the lock is removed and the mutator is resumed. The overhead is very sensitive to the trapping architecture of the system: an average range is around 13-63%[16].

In hardware [8] A small amount of extra hardware on a processor enables pointer-loads to be checked in parallel with normal execution. Again a trap to the run-time system occurs when the mutator trips over the barrier. Typical overheads with this approach are 9-11%[16].

This paper describes a new technique for implementing the read-barrier in software that carries a typical CPU overhead of less than 10% of a program's execution time, comparable with or better than previous figures published for hardware implementations. It is based on a system of untagged garbage collection similar to that employed in the Spineless Tagless G-machine[11][12][13]: we shall in fact describe the scheme in the context of the STG machine. Each object carries with it pointers to code to evacuate and scavenge the object during a garbage collection. This code is created for each object-type by the compiler: as the code knows the detailed structure of the object (e.g. which arguments are pointers and which are immediate values), it executes without interpretive loops and so is particularly efficient. Moreover, by manipulating these code-pointers during garbage collection, each object can implement its own individual read-barrier by tracing itself and its arguments if there is a danger of obselete pointers being propagated. The overhead of such a read-barrier is solely in manipulating these code-pointers and is significantly lower than in previous software implementations.

The next section describes the salient features of the Spineless Tagless G-machine and the untagged garbage collection scheme it uses for bulk copying garbage collection. Section 3 describes how the code for each object is modified to introduce the read-barrier. Section 4 discusses the overheads of this modification. Section 5 concludes the paper and outlines further work.

2 The Spineless Tagless G-machine

The Spineless Tagless G-machine[11][12][13] is a model for the compilation of lazy functional languages. It is derived from the G-machine[2][10] and the Spineless G-machine[5] (hence the name). The STG machine has been developed and implemented by Simon Peyton-Jones and the GRASP team at Glasgow University.

The principal distinguishing feature of the STG machine lies in its representation of program objects. Program objects take several forms: they have different (program-level) types, they have different layouts of pointers and immediate values, they may be evaluated or unevaluated, etc. The traditional way of distinguishing different types of objects is to encode the required information into a 'tag' on each object and to test the tag before performing operations on the object. In particular, when the value of an object is demanded, the mutator tests the tag and only enters the object if it has not previously been evaluated. When an object is evaluated, it is updated with its result (laid out in a standard fashion to make it available to other objects) and its tag is set to reflect the fact that it has been evaluated.

In contrast, the STG machine represents all program objects as closures with no tags. The first field of each closure is a pointer (the *info pointer*) to a statically-allocated *info table* that contains several code-pointers implementing the various operations that will be performed on the closure. The most important of these is the entry-code-pointer: when the value of a closure is demanded, the mutator simply enters the closure (jumps to its entry-code-pointer) without performing any test. Evaluated closures are updated as before and their info pointer is set to an info table whose entry-code performs an immediate return when called.

The most important feature of the STG machine for our purposes is that a closure has some control over its own operational behaviour, via the code-pointers in its info table. Section 3 describes how a closure can manipulate these pointers to implement its own individual read-barrier.

2.1 Garbage Collection on the Spineless Tagless G-machine

The original STG machine implementation used bulk copying garbage collection. The copying process involves two operations on closures.

Evacuation The closure is copied from from-space into to-space, leaving a forwarding address in the old copy and returning the address of the new copy. The closure is placed on a queue of closures in to-space waiting to be scavenged.

Scavenging Each closure to which the closure points is evacuated, unless this has already taken place, and a pointer to the new copy is substituted for the pointer to the old copy.

The STG machine implements these two operations as code-pointers in the info table of each closure. Evacuation and scavenging code is generated individually at compile-time for each closure-type in the program: the compiler knows the exact layout of the closure and can generate 'unfolded' code that is particularly efficient. It is also trivial to generate efficient evacuation code for special closure-types such as indirections and static closures.

Figure 1 illustrates the sort of code that is generated[2]. It shows the layout of a closure with two pointers and two immediate values, together with its info table[3], the code for evacuating the closure and the code for scavenging the closure. Notice that the three code-pointers are accessed by offsets from the info pointer: the importance of this will be seen later. The macro instructions used in the Figure (and subsequent figures) are defined in Appendix A.

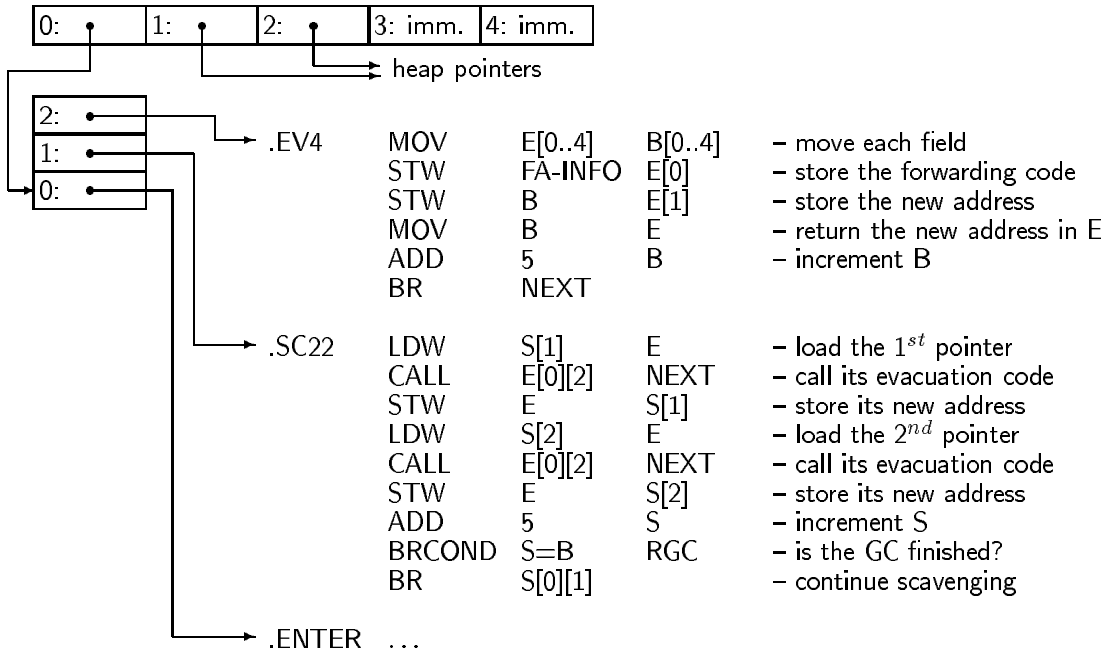

Figure 1: The layout of a closure with two pointers and two immediate values. B is the free-space pointer, S points to the closure when it is being scavenged, E points to the closure when it is being evacuated, NEXT holds the return address for an evacuation and RGC holds the return address for a garbage collection.

3 Incremental Garbage Collection

Baker-style incremental garbage collection has two essential features.

1. Each time the mutator claims some space from the heap, the garbage collector is called to scavenge k closures on the collector queue. k is called the *mark-ratio* and it must be large enough to ensure that all of the live closures in from-space are evacuated and scavenged before to-space is filled up. This is the mechanism by which most closures are garbage collected.

2. There must be no possibility of from-space references being propagated into to-space (beyond those that arise naturally in closures on the collector queue and that will be scavenged by the garbage

[2]Figure 1 is a more detailed version of Figure 4 from Section 7.1 of [13].
[3]The info table will usually contain other fields, but these are not immediately relevant.

collector). This requires the implementation of a *read-barrier*, that ensures that the workspace of the mutator contains only to-space pointers. Thus the mutator gets the same view of memory as it would have under a stop-and-copy scheme.

The first of these requirements is simple to implement. At each allocation request, the mutator checks if there is a garbage collection in progress: if there is, it arranges for k closures to be scavenged and jumps to the scavenging code of the first closure on the collector queue. It must also arrange for the stacks (and other pointers outside the heap) to be scavenged incrementally in similar fashion. We shall not discuss this aspect of the scheme further.

For the STG machine, we can interpret Requirement 2 above as the following invariant:

> *The currently-active closure (CAC), and all of the closures to which it can refer (both free variables in its own fields and bound variables on the stack), must lie in to-space.*

This implies that the workspace of the CAC contains no references to from-space, thus there is no danger of such references being propagated. (We assume that any registers holding local values will have been loaded from the workspace.)

In operational terms, there are two major elements to implementing this requirement.

- If the CAC demands the value of a closure, that closure must have been scavenged before its code is entered.
- If the CAC returns to its caller, the stack frame that the caller's continuation will use must have been scavenged before its code is re-entered.

The implementation of these two operations is described in Sections 3.1 and 3.2 respectively. Special treatment required for the update stack and the CAF list is dealt with in Sections 3.3 and 3.4 respectively.

3.1 Entering a Closure

The first thing that happens to a closure during a garbage collection is that it is evacuated and placed on the queue of closures in to-space waiting to be scavenged. At some subsequent time, the garbage collector scavenges the closure. If the closure is entered between being evacuated and being scavenged, there is a danger of from-space references being propagated into to-space. We avoid this by arranging for the closure to scavenge itself if it is entered while it is on the collector queue.

We extend the info table of the closure with two more 'scavenging pointers' and we cause the closure to scavenge itself when it is entered by side-effecting its info pointer when it is evacuated, as shown in Figure 2. After evacuation, the closure will either be entered by the mutator or scavenged by the garbage collector.

- If the closure is entered by the mutator first, the mutator enters at .MUTSC (offset 0 from the new info pointer) rather than at .ENTER. .MUTSC scavenges the closure (that it can access via NODE, that always points to the CAC), resets its info pointer and drops through to the 'normal' evaluation code. The garbage collector will eventually enter the closure at .SKSC22 (offset 1 from the info pointer), skipping over it with no effect.
- If the closure is scavenged by the garbage collector first, the garbage collector enters the closure at .SC22 (offset 1 from the info pointer), scavenges the closure and resets its info pointer so that any subsequent mutator entry will occur in the normal fashion at .ENTER (offset 0 from the info pointer).

All of this code is illustrated in Figure 2, with the instructions to adjust the info pointer highlighted in bold. In effect, the two segments of scavenging code co-operate to guarantee that the closure is scavenged exactly once *before* it is entered.

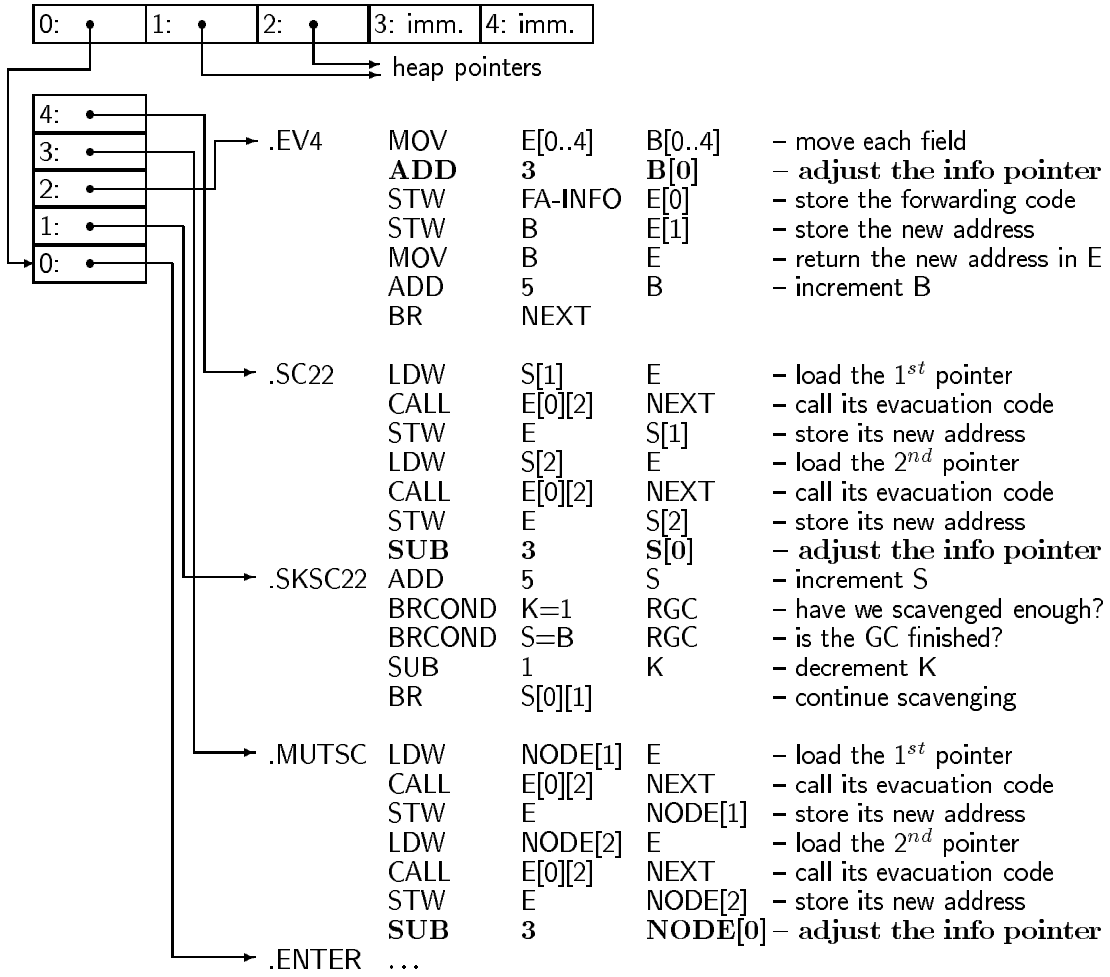

Figure 2: The layout of a closure with two pointers and two immediate values under the incremental regime. NODE points to the closure when it is active and K counts down (from k) the number of closures to be scavenged.

Note that it is possible to abstract the mutator-scavenging code from Figure 2 so that similarly-shaped closures can share it via the subroutine call mechanism (as has already been done with the normal evacuation and scavenging code).

3.2 Returning to a Closure

A similar scheme is used to scavenge the (pointer-)arguments on the stack when the CAC returns to its caller. The code for each closure is augmented with a subroutine call before each of its return points, in the same way as .MUTSC precedes .ENTER in Figure 2. This is shown in Figure 3. Note that in this case, there is no indirection through the info table, so we are forced to incorporate a subroutine call to ensure that the required decrement is uniform across all closure types (the CAC may not know the layout of its caller). The subroutines .SCSFm scavenge the stack frame of the closure, decrement the next return address on the stack and branch back to the 'normal' return code. The code for the .SCSFm can be generated at compile-time in 'unfolded' form, avoiding the cost of maintaining a loop counter.

```
.SCSF2   LDW     SUA[0]    E       – load the top location
         CALL    E[0][2]   NEXT    – call its evacuation code
         STW     E         SUA[0]  – store its new address
         LDW     SUA[1]    E       – load the second location
         CALL    E[0][2]   NEXT    – call its evacuation code
         STW     E         SUA[1]  – store its new address
         SUB     RET_DEC   SUB[0]  – decrement the next return address
         BR      RGC

         CALL    SCSF2     RGC     – scavenge the top m stack locations
.RET_i   ...
```

Figure 3: Incremental stack-scavenging code for a closure with two pointers on the A-stack. SUA is the stack-pointer for the A-stack, RET_DEC is the return-address-decrement and SUB is the stack-pointer for the B-stack. This code appears whenever a closure demands the value of another closure.

At the start of a garbage collection, the CAC's stack frame is scavenged and the top return address on the stack is decremented so that it points to its scavenging subroutine call. This sets up the invariant of stack garbage collection:

> *The stack frame of the CAC has been scavenged and the return address of its caller has been decremented so that the caller's stack frame will be scavenged before its code is re-entered.*

This invariant is maintained by each execution of the subroutines .SCSFm. The size of the stack frame and the offset of the return address at the time the garbage collection is initiated can either be calculated in advance and stored in a table (with one entry for each allocation request in the program) or determined on-the-fly by run-time abstract interpretation on the value of the stack-pointer.

The stack is also scavenged 'bottom-up' by the garbage collector at each allocation to ensure that the whole stack is scavenged before the end of the collection. Both stack-scavenging mechanisms maintain pointers into the stack to record how far they have progressed to avoid problems when they meet.

There should be no problem in combining this mechanism with vectored returns.

3.3 The Update Stack

The code-pointers in update frames resemble return addresses in all respects relevant to garbage collection. The update stack is therefore collected using the technique described in Section 3.2 for the return stack.

An inevitable consequence of performing garbage collection incrementally is that the optimisation described in Section 10.7 of [13] cannot be used: we cannot postpone garbage collecting the whole of the update stack until the end of the garbage collection, as its size cannot be predicted in advance and 'incrementality' would be jeopardised. The loss of this optimisation represents a (small) dynamic space cost of the scheme.

When garbage collection is performed incrementally, it is possible for a closure to be updated between being evacuated and being scavenged by the garbage collector. This causes a problem if the updated closure is smaller than the original: when the garbage collector eventually scavenges the closure, the scavenging code for the updated closure will not increment the heap-pointer B by enough to skip over the whole of the original closure. We can avoid this easily enough (but at some cost) by making all updates of closures on the collector queue via indirections, and using the spare field in indirection closures to hold the scavenging code of the original closure[4]. Note that the original closure must have been scavenged before the update, so its code will just increment S and continue. The scavenging code for indirections will then be as shown in Figure 4.

```
.SCIND    BR          S[2]                    – scavenge the original closure
```

Figure 4: The scavenging code for indirections.

Supporting updates-in-place requires more careful treatment of info pointers in updated closures that are on the collector queue in to-space. A possible extension to the scheme to allow updates-in-place is described in [14].

3.4 Other Roots Outside the Heap

The CAF list (the list of constant applicative forms) contains statically-allocated pointers into the heap. Some care has to be taken to ensure that these pointers are updated in the correct manner during a garbage collection: the simplest method is to evacuate the top-level closure of each CAF at a flip, then evacuate the rest in the normal fashion. As the maximum length of the CAF list is known at compile time, this operation does not jeopardise incrementality.

4 Performance Implications

4.1 Time Overheads

The time overheads of the scheme come in two areas: overheads on normal execution and overheads on garbage collection.

The only overhead on normal execution is that at each allocation, the mutator tests whether there is a garbage collection in progress, in order to decide whether to invoke the garbage collector.

There are three time overheads on garbage collection itself.

- The info pointer of each closure is adjusted during both evacuation and scavenging. Note that this overhead does not apply to closures that contain no pointers (they do not need to be scavenged).

[4]Note that indirections did not previously require scavenging code.

- The return address of each closure whose callee is active during the garbage collection is decremented and a subroutine call is required when its stack frame is scavenged.
- Closures, stack pointers and update frames have to be counted as they are scavenged. Note that this overhead is eliminated if the mark-ratio is set to 1.

Simulations suggest that these costs add only around 10% to the execution time of a program, considerably less than previous software implementations of Baker's algorithm[16]. This figure is borne out by statistics collected from our partial implementation of the scheme, described in Section 4.3.

Our extended info tables may complicate the optimisation described in Section 7.6 of [13] when native code is being generated. This may lead to (slight) additional time overheads in this event.

4.2 Space Overheads

The scheme carries no dynamic space overhead (but see Section 3.3), but there is an increase in the size of the compiled code for each closure. We have not yet quantified this effect. There are several trade-offs available between code-size and execution speed, for example the code pointed to by .MUTSC and .SCSFm can be designed using loops to maximise sharing or using straight-line code to maximise speed. Similarly, the information required to initiate stack garbage collection can be calculated in advance and stored in a table or determined on-the-fly using run-time abstract interpretation.

4.3 Experimental Results

We have constructed a partial implementation of the scheme on top of version 0.10 of the Glasgow Spineless Tagless G-machine compiler. The current implementation scavenges both stacks at a flip, so it does not qualify as properly incremental. It does, however, give a good indication of the costs involved.

*** Some figures/tables, etc. in here. ***

5 Conclusions

We have described a scheme for incorporating incremental garbage collection into the Spineless Tagless G-machine on stock hardware. The scheme is based on the manipulation of the code-pointers that are used to implement closure behaviour in the STG machine. Each closure creates its own, 'personal' read-barrier by manipulating its code-pointers during execution and garbage collection. The technique derives its efficiency from the twin facts that these manipulations are very cheap and that they are performed on a per-closure basis, rather than a per-pointer-load basis, as in most implementations of the read-barrier. Simulations suggest that this technique will add only around 10% to the execution time of a program: this figure is borne out by statistics from our partial implementation of the scheme on the STG machine. This cost may be more than compensated for by the other documented advantages of a read-barrier implementation, for example its beneficial effect on paging costs[7][9].

A previous application of this research to a modified version of the $< \nu, G >$-machine[3] is described in [14]. At first sight, the scheme appears to apply only to models that always employ a subroutine call when a closure is entered, irrespective of whether the closure has previously been evaluated. However, the scheme can also apply to models that employ tags to distinguish evaluated closures from unevaluated closures, provided that a closure on the queue in to-space waiting to be scavenged is distinguished from an evaluated closure. Such a closure will then expect to be entered before its contents are examined, permitting it to scavenge itself and return. The scheme can apply to any model where a closure has some

control over its evaluation/garbage collection behaviour: in the case of the STG machine, this control is exercised via the code-pointers in each closure's info table.

Acknowledgements

We should like to thank Simon Peyton-Jones and the GRASP team for making their compiler available to the world, and Rana Peries for his help with its installation.

We should like to thank Paul Kelly and Denis Howe for their comments on earlier versions of this paper.

An extended abstract of this material was presented at the Massey Functional Programming Workshop, Massey University, Palmerston North, New Zealand in August 1992[15].

References

[1] A.W. Appel, J.R. Ellis and K. Li, *Real-time Concurrent Collection on Stock Multiprocessors*, Conference on Programming Language Design and Implementation, pp. 11-20, 1988.

[2] L. Augustsson, *Compiling Lazy Functional Languages, Part II*, Ph.D thesis, Chalmers University, Sweden, 1987.

[3] L. Augustsson and T. Johnsson, *Parallel Graph Reduction with the $< \nu, G >$-machine*, Conference on Functional Programming Languages and Computer Architecture, pp. 202-13, 1989.

[4] H.G. Baker, *List-processing in Real-time on a Serial Computer*, CACM 21(4), pp. 280-94, 1978.

[5] G.L. Burn, S.L. Peyton-Jones and J. Robson, *The Spineless G-machine*, Conference on Lisp and Functional Programming, pp. 244-58, 1988.

[6] C.J. Cheney, *A Non-recursive List Compacting Algorithm*, CACM 13(11), pp. 677-8, 1970.

[7] R. Courts, *Improving Locality of Reference in a Garbage-collecting Memory Management System*, CACM 31(9), pp. 1128-38, 1988.

[8] D. Johnson, *Trap Architectures for Lisp Systems*, Conference on Lisp and Functional Programming, pp. 79-86, 1990.

[9] D. Johnson, *The Case for a Read Barrier*, ASPLOS-IV, pp. 279-87, 1991.

[10] T. Johnsson, *Compiling Lazy Functional Languages*, Ph.D thesis, Chalmers University, Sweden, 1987.

[11] S.L. Peyton Jones and J. Salkild, *The Spineless Tagless G-machine*, Conference on Functional Programming Languages and Computer Architecture, pp. 184-201, 1989.

[12] S.L. Peyton Jones, *The Spineless Tagless G-machine: Second Attempt*, Workshop on the Parallel Implementation of Functional Languages, pp. 147-91, University of Southampton CSTR 91-07, 1991.

[13] S.L. Peyton Jones, *Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine*, to appear in the Journal of Functional Programming, 1992.

[14] R.L. While, *A Viable Software Read-barrier*, Departmental Report DoC 92/12, Dept. of Computing, Imperial College, 1992.

[15] R.L. While and A.J. Field, *Incremental Garbage Collection for the Spineless Tagless G-machine*, Massey Functional Programming Workshop, Massey University, New Zealand, 1992.

[16] B.G. Zorn, *Comparative Performance Evaluation of Garbage Collection Algorithms*, Ph.D thesis, University of California at Berkeley, 1989.

A Instruction Definitions

The Figures in the main body of the paper use eight macro instructions. A register reg is specified in the code by its name. An address addr is of the form A[i], where A is a register and i is an offset. An immediate value imm can be either a constant or a label.

LDW addr reg	loads the value stored at addr into reg.
STW reg addr	stores the value held in reg in addr.
STW imm addr	stores the value imm in addr.
MOV reg_1 reg_2	loads the value held in reg_1 into reg_2.
MOV $addr_1$ $addr_2$	stores the value stored at $addr_1$ in $addr_2$.
ADD/SUB imm reg	adds/subtracts the value imm to/from the value held in reg.
ADD/SUB imm addr	adds/subtracts the value imm to/from the value stored at addr.
BR addr/reg/imm	jumps to the specified point in the code.
BRCOND cond addr/reg	evaluates the boolean expression cond: if it evaluates to true, it jumps to the specified point in the code, otherwise it continues with the following instruction.
CALL addr/imm reg	loads the address of the following instruction into reg and jumps to the specified point in the code.