Proceedings of Symposium on AI and Programming Languages, ACM *Sigplan Notices* **12**,8 (Aug. 1977), 55-59.

The Incremental Garbage Collection of Processes

Henry G. Baker, Jr. and Carl Hewitt
Massachusetts Institute of Technology
Cambridge, Massachusetts

This paper investigates some problems associated with an argument evaluation order that we call "future' order, which is different from both call-by-name and call-by-value. In call-by-future, each formal parameter of a function is bound to a separate process (called a "future") dedicated to the evaluation of the corresponding argument. This mechanism allows the fully parallel evaluation of arguments to a function, and has been shown to augment the expressive power of a language.

We discuss an approach to a problem that arises in this context: futures which were thought to be relevant when they were created become irrelevant through being ignored in the body of the expression where they were bound. The problem of irrelevant processes also appears in multiprocessing problem-solving systems which start several processors working on the same problem but with different methods, and return with the solution which finishes first. This *parallel method* strategy has the drawback that the processes which are investigating the losing methods must be identified, stopped, and reassigned to more useful tasks.

The solution we propose is that of garbage collection. We propose that the goal structure of the solution plan be explicitly represented in memory as part of the graph memory (like Lisp's heap) so that a garbage collection algorithm can discover which processes are performing useful work, and which can be recycled for a new task.

An incremental algorithm for the unified garbage collection of storage and processes is described.

Key Words and Phrases: garbage collection, multiprocessing systems, processor scheduling, "lazy" evaluation, "eager" evaluation.

CR Categories: 3.60, 3.80, 4.13, 4.22, 4.32.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0522.

1. Introduction

Processors are becoming very cheap and there is good evidence that this trend will continue in the next few years. As a result, there has been considerable interest in how to apply large numbers of processors to the solution of a single task [7]. Since efficient utilization of these hordes of processors requires a lot of communication, sorting networks have been devised [2,16] which allow every processor in an N-processor system to both send and receive a message on every clock pulse, with only the highest priority messages getting through, while providing acknowledgement of success or failure to the sender. Furthermore, the transit time through the network is only $O(\log^2 N)$ and the size of the network only $O(N\log^2 N)$. However, it is still not clear what all these processors should be communicating about.

Friedman and Wise [10] quite rightly note that applicative languages (languages without side-effects, e.g. "pure" LISP) are excellently suited for the purpose of representing many algorithms intended for execution on a host of processors since their lack of side-effects eliminates a great source of complexity in parallel execution. Thus, "Church-Rosser" theorems can be proved which ensure the invariance of the value of an applicative expression regardless of the order or relative speed of evaluation. However, we must keep in mind that this kind of parallelism does not implement the most general form of communication between processes. For example, it is not possible to implement an airline reservation system in such a language, due to its non-determinate behavior.

In this paper we consider an "eager beaver" evaluator for an applicative programming language which starts evaluating every subexpression as soon as possible, and in parallel. This is done through the mechanism of *futures*, which are roughly Algol-60 "thunks" which have their own evaluator process ("thinks"?). (Friedman and Wise [10] call futures "promises", while Hibbard [13] calls them "eventuals".) When an expression is given to the evaluator by the user, a *future* for that expression is returned which is a promise to deliver the value of that expression at some later time, if the expression has a value. A process is created for each new future which immediately starts to work evaluating the given expression. When the value of a future is needed explicitly, e.g. by the primitive function "+", the evaluation process may or may not have finished. If it has finished, the value Is immediately made available; if not, the requesting process is forced to wait until it finishes.

Futures are created recursively in the evaluation of an expression whenever our eager evaluator encounters functional application. A new future is created for each argument, resulting in the parallel (collateral) evaluation of those arguments, while the main evaluator process tackles the function position. We call the main evaluator process the *parent*, while any futures it directly creates become its *offspring*.

More precisely, a *future* is a 3-tuple (process, cell, queue), where *process* is the virtual processor initialized to evaluate this argument expression in its proper environment, *cell* is a writable location in memory which will save the value of the argument after it has been computed to avoid recomputing it, and *queue* is a list of the processes which are waiting for the value of this future. A future's process starts evaluating its expression in the given environment. If any other process needs the value of this future, and the value is not yet ready, the requesting process enters the queue of the future and goes to sleep. When the value promised by the future is ready, its process stores the value into its cell, wakes up all processes waiting in its queue, and dies. Henceforth, any process needing this future's value can find it in the future's cell without waiting or further computation.

The main problem with our eager interpreter is that since it anticipates which values are going to be required to compute the final result, it can be wasteful. A process may be assigned to the computation of a future whose value will never be needed; in this case, we say that the process is *irrelevant*. If there were no way of determining irrelevancy, irrelevant processes could tie up a significant amount of the system computing power. Furthermore, if a future were caught in a non-terminating evaluation, its computational power would be lost to the system forever! In the following sections, we argue that the "garbage collection" of passive storage can be extended to the reclamation of irrelevant active processes. Furthermore, this collection can be done in an *incremental* manner, eliminating the long delays required for classical garbage collection.

2. Garbage Collecting Irrelevant Futures

A classical garbage collector for passive storage proceeds by marking the *root* of the heap, and then propagating marks from marked nodes to their offspring until there is no unmarked node with a marked parent. Upon the termination of this procedure, any unmarked nodes are not accessible from the root and are therefore returned to the free list.

The key to garbage collecting *active processes* is that their process-states are addressable as vectors of words in the common address space of all the processors,[1] but marked with a special type-code. This vector stores the contents of the registers of the process. The top-level process (that assigned to the top-level future) is kept always directly accessible from the root of the heap. Suppose now that we stop all the processes at the beginning of garbage collection. As our classical collector traces the heap, it can recognize when it encounters a process. By marking a process, the collector proves that it is still relevant, hence it can be restarted when the garbage collection is finished. If a process is not marked, it is garbage collected, and a processor will not be reassigned to it.

What makes irrelevant processes go away during garbage collection is the fact that they are not accessible from static data structure; i.e. from the root of the heap. Since all relevant futures are bound in the environment structure to some program variable or temporary variable, they will be marked and retained.

(If "busy waiting" is used in an extended system having side-effects, then a process which is synchronized through busy waiting will be accessible as long as its parent is accessible, regardless of whether any other process needs its result. Hence it may not be collected even if it becomes irrelevant. This is one reason why busy waiting is not a good synchronization method.)

Garbage collection is made incremental by using some of the ideas from an earlier paper [1], which in turn is based on the work of Dijkstra [5,6] and Lamport [14,15]. The mark phase of our incremental garbage collector process employs three colors for every object—*white*, *grey*, and *black*. Intuitively, *white* nodes are not yet known to be accessible, *grey* nodes are known to be accessible, but whose offspring have not yet been checked, and *black* nodes are accessible, and have accessible offspring. Initially, all nodes (including processes) are white. A white node is made grey by *shading* it; i.e. making it "at least grey" [5], while a grey node is *marked* by shading its offspring and making the node black—both indivisible processes. Marking is initiated by stopping all processes and shading the root. Marking proceeds by finding a grey node, shading its offspring, then making that node black. When there are no more grey nodes, garbage collection is done; all still-white nodes are then emancipated and the colors white and black switch interpretations.

After garbage collection has begun, a user process can be restarted as soon as it has been blackened by the collector. Since the top-level process is pointed at directly by the root of the heap, it is restarted almost immediately. It should be obvious that when a process first becomes black, it cannot point directly at a white node. We wish to preserve this assertion. Therefore, whenever a running black process is about to violate it—by inserting into one of its registers the white component of a node it is already pointing at—it immediately shades the white node before proceeding. Furthermore, every new node the process needs is created black. The intuitive rationale behind these policies is that *so far as any black process is concerned, the garbage collection has already finished.* Furthermore, *the nodes which are found accessible by the garbage collector are exactly those which were accessible at the time the garbage collection was started.*

We prove the correctness of this garbage collector informally. The garbage collector is given a head start on all of the processes because they are stopped when it is started. When a process is restarted, it is black, and everything it sees is at least grey, hence it is in the collector's wake. Whenever a process attempts to catch up to the collector by tracing an edge from a node it can access directly, that node is immediately shaded. Therefore, it can never pass or even catch the collector. Since the collector has already traced any node a process can get its registers on, the process cannot affect the connectivity of the nodes that the collector sees. Because white or grey processes are not allowed to run, any created nodes are black, and since nodes darken monotonically, the number of white nodes must monotonically decrease, proving termination.

Our garbage collector has only one phase—the mark phase—because it uses a compacting, copying algorithm [8,4] which marks and copies in one operation. This algorithm copies accessible list structures from an "old semispace" into a "new semispace". As each node is copied. a "forwarding address" is left at its old address in the old semispace. If the Minsky copying algorithm is used [8], the collector has its own stack to keep track of grey nodes; the

[1] We assume throughout this paper that all processors are embedded in a single, global, shared address space.

Cheney algorithm [4] uses a "scan pointer" to linearly scan the new semispace, while updating the pointers of newly moved nodes by moving the nodes they point to. The correspondence between our coloring scheme and these algorithms is this: *white* nodes are those which reside in the old semispace; *grey* nodes are those which have been copied to the new semispace, but whose outgoing pointers have not been updated to point into the new semispace (i.e. have not yet been encountered by the scan pointer in the Cheney algorithm); and *black* nodes are those which have been both moved and updated (i.e. are behind the scan pointer). When scanning is done (i.e. there are no more grey nodes and all accessible nodes have been copied), the old and new semispaces then interchange roles. Reallocating processors is simple; all processors are withdrawn at the start of garbage collection, and are allocated to each process as it is blackened. Thus, when the garbage collection has finished, all and only relevant processes have been restarted.

The restriction that white processes cannot run can be relaxed under the condition that *a white process may not cause a black node to point to a white one*. This can only happen if the white process is trying to perform a side-effect on a black node. If operations of this type are suspended until either the process either becomes black or is garbage-collected, then proper garbage collector operation can be ensured, and convergence guaranteed. Under these conditions, white processes create only white cells. When a white process is encountered by the garbage collector, it must stop and allow itself to be colored black before continuing.

The notion that processes must be marked as well as storage may explain some of the trouble that Dijkstra and Lamport had when trying to prove their parallel garbage collection algorithm correct [5,6,14,15]. Since their algorithm does not mark a user process by coloring it black (thereby prohibiting it from directly touching white nodes), and allows these white processes to run, the proof that the algorithm collects only and all garbage is long and very subtle (see [15]).

3. Coroutines and Generators

One problem with our "eager beaver" evaluator is that some expressions which have no finite values will continue to be evaluated without mercy. Consider, for example, the infinite sequence of squares of integers 0,1,4,9,... We give below a set of LISP-like functions for computing such a list.

```
losq ≡ (λx. (cons (* x x) (losq (+ x 1))))   ; Compute an element.

cons ≡ (λx y. (λz. (if (= z 'car) x y)))     ; Define CONS function.

car ≡ (λx. (x 'car))                         ; Ask for 1st component.

cdr ≡ (λx. (x 'cdr))                         ; Ask for 2nd component.

list-of-squares ≡ (losq 0)                   ; Start the recursion.
```

The evaluation of "(losq 0)" wIll start off a future evaluating "(cons …)", which will start up another future evaluating "(losq 1)", and so forth. Since this computation will not terminate, we might worry whether anything useful will ever get done. One way to ensure that this computation will not clog the system is to convert it into a "lazy" computation [9] by only allowing it to proceed past a point in the infinite list when someone forces it to go that far. This can be easily done by performing a lambda abstraction on the expression whose evaluation is to be delayed. Since our evaluator will not try to further evaluate a λ-expression, this will protect its body from evaluation by our eager beavers.

```
losq' ≡
 (λx. (cons  (* x x)
             (λz. ((losq' (+ x 1)) z)))) ; Protect from early evaluation.
```

However, this "hack" is not really necessary if we use an *exponential scheduler* for the proportion of effort assigned to each process. This scheduler operates recursively by assigning 100% of the system effort to the top-level future, and whenever this future spawns new futures, it allocates only 50% of its allowed effort to its offspring. While a process is in the waiting queue of a future, it lends its processing effort to the computation of that future. However, a future which finishes returns its effort to helping the system—not its siblings. Now the set of futures can be ordered according to who created whom and this ordering forms a tree. As a result of our exponential scheduling, the further down in this tree a future is from the top-level future, the lower its priority in scheduling. Therefore, as our eager beavers produce more squares, they become exponentially more discouraged. But if other processes enter the queue for the square of a large number, they lend their encouragement to its computation.

In an evaluator which uses call-by-future for CONS, the obvious program for MAPCAR (the LISP analog of APL's parallel application of a function to a vector of arguments) will automatically do all of the function applications in parallel in a "pipe-lined" fashion. However, due to the scheduler the values earlier In the list will be accorded more effort than the later ones.

Because this scheduler is not omniscient, system effort will still have to be reallocated by the garbage collector as it discovers irrelevant processes and returns their computing power to help with still relevant tasks.

4. Time and Space

"Lazy" evaluation [9] (call-by-name) using "evaluate-once" thunks is an optimal strategy for evaluating applicative expressions on a single processor, in the sense that the minumum number of reductions (procedure calls) are made [17,3]. However, when more than one processor is available to evaluate the expression, it is not clear what strategy would be optimal. If nothing is known about the particular expression being evaluated, we conjecture that any reasonable strategy must allocate one processor to lazy evaluation, with the other processors performing eager evaluation. We believe that our "eager beaver" evaluator implements this policy, and unless the processors interfere with one another excessively, a computation must always run faster with an eager evaluator running on multiple processors than a lazy evaluator running on a single processor. If there are not enough processors to allocate one for every future, then we believe that our "exponential scheduling" policy will dynamically allocate processor effort where it is most needed.

Although the universal creation of futures should reduce the time necessary to evaluate an expression when an unbounded number of processors are available, we must consider how the space requirements of this method compare with other methods. The space requirements of futures are hard to calculate because under certain schedules, future order evaluation approximates call-by-value, while with other schedules, it is equivalent to call-by-name (but evaluated only once). In the worst case, the space requirements of futures can be arbitrarily bad, depending upon the relative speed of the processors assigned to non-terminating futures.

5. The Power of Futures

The intuitive semantics associated with a future is that it runs asynchronously with its parent's evaluation. This effect can be achieved by either assigning a different processor to each future, or by multiplexing all of the futures on a few processors. Given one such implementation, the language can easily be extended with a construct having the following form: "(EITHER $<e_1>$ $<e_2>$ ... $<e_n>$)" means evaluate the expressions $<e_i>$ in parallel and return the value of "the first one that finishes". Ward [18] shows how to give a Scott-type lattice semantics for this construct. He starts with a power-set of the base domain and gives it the usual subset lattice structure, then extends each primitive function to operate on sets of elements from the base domain in the obvious way, and finally defines the result of the EITHER construct to be the *least upper bound* (*LUB*) of all the $<e_i>$ in the subset lattice. The EITHER construct is approximated[2] by spawning futures for all the $<e_i>$, and polling them with the parent process until the first one finishes. At that point, its answer is returned as the value of the "EITHER" expression, and the other futures become inaccessible from the root of the heap.

We give several examples of the power of the "EITHER" construct:

```
(multiply x y) ≡ (EITHER  (if x=0 then 0 else ⊥)
                          (if y=0 then 0 else ⊥)
                          (* x y))

(integrate exp bvar) ≡
  (EITHER                 (fast-heuristic-integrate exp bvar)
            (Risch-integrate exp bvar))
```

The first example is that of a numeric product routine whose value is zero if either of its arguments are zero, even if the non-zero argument is undefined. The second example is an integration routine for use in a symbolic manipulation language like Macsyma, where there is a relatively fast heuristic integration routine which looks for common special cases, and a general but slow decision procedure called the Risch algorithm. Since the values of both methods are guaranteed to be the same (assuming that they perform integration properly), we need not worry about the possibility of non-determinacy of the value of this expression (i.e. non-singleton subsets of the base domain in Ward's lattice model).

One may ask what the power of such an "EITHER" construct is; i.e. does it increase the expressive power of the language in which it is embedded? A partial answer to this question has been given with respect to "uninterpreted" schemata. Uninterpreted schemata answer questions about the expressive power of programming language constructs which are implicit in the language, rather than being simulated. For example, one can compare the power of recursion versus iteration in a context where stacks cannot be simulated. Hewitt and Patterson [11] have shown that uninterpreted "parallel" schemata are strictly more powerful than recursive schemata. The essense of this difference is that parallel schemata can simulate non-deterministic computation without bogging down in some infinite branch by following all branches in parallel.

Also, Ward [18] has shown that the "EITHER" construct strictly increases the power of the λ-calculus in the sense that there exist functions over the base domain which are inexpressible without "EITHER", but are trivially expressible with it.

6. Shared Databases

The advantage that garbage collection has over the explicit killing of processes becomes apparent when parallel processes have access to a shared database. These databases are usually protected from inconsistency due to simultaneous update by a mutual exclusion method. However, if some process were to be killed while it was inside such a database, the database would remain locked, and hence unresponsive to the other processes requesting access.

The solution we propose is for the database to always keep a list of pointers to the processes which it has currently inside. In this way, an otherwise irrelevant process will be accessible so long as it is inside an accessible database. However, the moment it emerges, it will be forgotten by the database, and subject to reclamation by garbage collection. The *crowds* component of a *serializer*, a synchronization construct designed to manage parallel access to a shared database [12], automatically performs such bookkeeping.

7. Conclusions

We have presented a method for managing the allocation of processors as well as storage to the subcomputations of a computation in a way that tries to minimize the elapsed time required. This is done by anticipating which subcomputations will be needed and starting them running in parallel, before the results they compute are needed. Because of this anticipation, subcomputations may be started whose results are not needed. and thus our method identifies and revokes these allocations of storage and processing power through an incremental garbage collection method.

The scheme presented here assumes that all of the processors reside in a common, global address space, like that of C.mmp [19]. Since networks of local address spaces look promising for the future, methods for garbage collecting those systems need to be developed.

There are currently no plans to implement this method due to the lack of access to suitable hardware. However, it could be implemented on systems like C.mmp [19] in a straightforward manner.

8. Acknowledgements

Some of the early thinking about call-by-future was done several years ago by J. Rumbaugh.

9. References

1. Baker, H.G., Jr. "List Processing in Real Time on a Serial Computer". AI Working Paper 139, MIT AI Lab., Feb. 1977, also *CACM 21*,4 (April 1978), 280-294.
2. Batcher, K.E. "Sorting Networks and their Applications". *1968 SJCC*, April 1968, 307-314.
3. Berry, Gerard and Levy, Jean-Jacques. "Minimal and Optimal Computations of Recursive Programs". ACM *POPL4*, Jan. 1977, 215-226.
4. Cheney, C.J. "A Nonrecursive List Compacting Algorithm". *CACM 13*,11 (Nov. 1970), 677-678.

[2] This implementation is only an approximation because only singleton sets of elements of the base domain can ever be returned.

Proceedings of Symposium on AI and Programming Languages, ACM *Sigplan Notices* **12**,8 (Aug. 1977), 55-59.

5. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M. "On-the-fly Garbage Collection: An Exercise in Cooperation". Dijkstra note EWD496, June 1975.
6. Dijkstra, E. W. "After Many a Sobering Experience". Dijkstra note EWD500.
7. Erman, L.D. and Lesser, V.R. "A Multi-level Organization for Problem Solving using Many, Diverse, Cooperating Sources of Knowledge". *Proc. IJCAI-75*, Sept. 1975, 483-490.
8. Fenichel, R.R., and Yochelson, J.C. "A LISP Garbage-Collector for Virtual-Memory Computer Systems". *CACM 12*,11 (Nov. 1969), 611-612.
9. Friedman, D. P. and Wise, D. S. "Why CONS should not evaluate its arguments". In S. Michaelson and R. Milner (*eds.*), *Automata, Languages and Programming*, Edinburgh University Press, Edinburgh (1976), 257-284.
10. Friedman, D. P. and Wise, D. S. "The Impact of Applicative Programming on Multiprocessing". *1976 International Conference on Parallel Processing*, 263-272.
11. Hewitt, C. and Patterson, M. "Comparative Schematology". *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*, June 1970.
12. Hewitt, C. and Atkinson, R. "Parallelism and Synchronization in Actor Systems". ACM *POPL4*, Jan. 17-19, 1977, L.A., Cal., 267-280.
13. Hibbard, P. "Parallel Processing Facilities". In *New Directions in Algorithmic Languages*, (*ed.*) Stephen A. Schuman, IRIA, 1976, 1-7.
14. Lamport, L. "Garbage Collection with Multiple Processes: An Exercise in Parallelism". Mass. Comp. Associates, CA-7602-2511, Feb. 1976.
15. Lamport, L. "On-the-fly Garbage Collection: Once More with Rigor". Mass. Comp. Associates, CA-7508-1611, Aug. 1975.
16. Moravec, H. P. "The Role of Raw Power in Intelligence". Unpublished ms., Stanford, Cal., May 1976.
17. Vuillemin, Jean. "Correct and Optimal Implementations of Recursion in a Simple Programming Language". *JCSS 9* (1974), 332-354.
18. Ward, S. A. "Functional Domains of Applicative Languages". MAC TR-136, Project MAC, MIT, Sept. 1974.
19. Wulf, W., *et al*. "HYDRA: The Kernel of a Multiprocessor Operating System". *CACM 17*,6 (June 1974), 337-345.