# Hash-consing Garbage Collection

Andrew W. Appel[*] Marcelo J. R. Gonçalves[†]
appel@princeton.edu mjrg@cs.princeton.edu

CS-TR-412-93

Princeton University

February 1993

**Abstract**

We describe an implementation of *hash-consing* for the Standard ML of New Jersey compiler. Hash-consing can eliminate replication among heap-allocated data, which may allow the use of fast equality checking and may also improve the locality of reference of a program. The cost of a hash table lookup for each record allocated may, however, offset any gains from the elimination of replication.

Our hash-consing scheme is integrated with a generational garbage collector. Only records that survive a garbage collection are "hash-consed," thus avoiding the cost of a table lookup for short-lived records. We discuss some issues related with the implementation of this scheme and present a performance evaluation.

[*]Supported in part by NSF Grant CCR-9200790
[†]Supported by CNPQ Grant 200579/90.8

# 1 Introduction

In heap-allocated programming languages, such as ML and Lisp, values are represented by memory records and complex objects by pointers to graph-like collections of records. One major drawback of this scheme is that a value that is used in more than one place in a program may be represented by more than one record. This may increase the space complexity of a program. It may also increase the time complexity, due to the need for structural comparison instead of a simple pointer comparison in order to determine the equality of two values (e.g. use of `equal` instead of `eq` in Lisp). Moreover, the duplication of storage for values is likely to affect the locality of reference of the program, causing an increase in cache misses and virtual memory page faults, which may further increase its running time.

Ershov [11] showed how to collapse trees to minimal DAGs by traversing trees bottom-up, using hashing to eliminate common subexpressions. Goto [13] implemented a Lisp system with a built-in hash-cons operation: his "h-cons" cells were rewrite protected and free of duplicate copies.

The technique of *hash-consing* guarantees that two identical objects will share the same records on the heap, and thus will be represented by the same pointer. The basic idea is very simple. Whenever a new record is allocated (or "consed") we Check whether there is already an identical record in the heap. If so, we avoid the allocation and simply use the existing one. Otherwise, we perform the allocation as usual. A hash table is used to search the heap for a duplicate record.

Hash-consing can be applied only to pure values, that is, values that never change during the execution of a program. For if two updatable records have identical values now, they might not be identical later, and so merging then could lead to incorrect results. This implies that it would be difficult to use hash-consing in a language like Lisp, where all cons cells are potentially updatable (using `rplaca/rplacd`), even though very few, if any, of the allocated cells are actually updated during the execution of a typical program. In ML, however, most records are immutable; the few updateable "ref-cells" are easily distinguishable at compile time and run time. Therefore, it is possible to add hash-consing to ML without changing its semantics.

Hash-consing can offer many advantages by eliminating the drawbacks of replication described above, but not without cost. First, allocation is much more expensive. Every record allocation must check for the existence of a copy of the new record and, even if this can be done in constant time, the extra cost can be quite significant. If allocation is very frequent, as is often the case in ML, this could easily make the program much slower, instead of making it faster as we would expect. Another problem is the space needed for the hash table, which can be large. This could offset the gain in space introduced by sharing; worse yet, the hash table is now competing for the cache and main memory.

In this paper we describe an experience with the implementation of a modified *hash-consing* scheme for the Standard ML of New Jersey compiler [3]. The SML/NJ compiler divides the heap in two main areas, the *newer generation*, where records are initially allocated and the *older generation* which contains the records that survive garbage collections. In our new hash-consing implementation, only those records that survive at least one garbage collection are "hash-consed". Thus, only the older generation is guaranteed not to have duplicate records. The motivation for this scheme is to avoid the cost of a hash table lookup for the large number of records that have a very short life (i.e. do not survive any garbage collections), while still taking advantage of some of the benefits of a full hash-consing scheme.

## 2 Overview of the SML/NJ Runtime System

This section contains a brief overview of the SML/NJ runtime system. For a more detailed description see Appel[1, 2].

The SML/NJ compiler uses a *generational garbage collector*, with two generations. The heap is divided in three main areas: the *newer generation*, where all records are initially allocated, the *older generation*, which contains all records that survive at least one garbage collection, and the *reserve space*, between the two generations. During a garbage collection, live records in the newer generation are copied to the reserve space, starting at the side adjacent to the older generation. Once the garbage collection is completed, the filled portion of the reserve space is added to the older generation, the rest of the heap (i.e. the newer generation and the unfilled portion of the reserve space) is divided evenly in a newer generation and a reserve space and the process starts all over again. This is the most common form of garbage collection and is called a *minor collection*. Occasionally, the older generation grows too large, and must also be collected (this is a *major collection*).

The garbage collection algorithm copies records from a *fromspace* to a *tospace*. The fundamental operation is to *forward* a pointer to a record. Three cases must be considered: 1) the pointer does not point into the *fromspace*—in this case there is nothing else to do; 2) the pointer points into the *fromspace* and the record pointed to has not been copied yet—in this case the record is copied to the next free location in the *tospace*, the record's new address is stored in its first word and the pointer is set to point to the new record address; 3) the pointer points into the *fromspace* and the record pointed to has already been copied to the *tospace*—in this case the pointer is set to the new address of the record, that is stored in its first word.

Garbage collection begins by forwarding a set of root pointers. Then the pointers in the records that were copied to the reserve space are forwarded, until there are no more pointers to forward. This corresponds to a *breadth-first* traversal of the live records, with the reserve area acting as a queue for the breadth-first search algorithm.

Heap records can contain four kinds of objects: *ML records*, *arrays*, *strings* and *bytearrays*. The important distinction between these kinds is that ML records and arrays may contain pointer values, and arrays and bytearrays may be updated. Each heap record is preceded by a one word tag that contains its kind and size. The low order bit is used to distinguish between pointer and non-pointer values in ML records and arrays. All objects are aligned to word boundaries.

# 3 Hash-consing during Garbage-collection

Our implementation does hash-consing "between generations" of the generational collector. We keep a hash table with all non-updatable records in the older generation. The hash table is indexed by the contents of the records and each entry contains a pointer to the record. While forwarding a pointer to an non-updatable record in the newer generation we do a hash table lookup to check for the existence of an identical record in the older generation. If such a record exists, then no copy is made and the pointer is set to the existing copy in the older generation. Otherwise, the record is copied, inserted into the hash table, and the pointer is set to the new address. Updatable objects are copied in the usual way, without using the hash table.

In order for this scheme to work correctly, we must be careful to copy objects in a bottom-up way[11]: given a DAG, we can only forward a node (i.e. a record) after we have forwarded all of its descendants. This bottom-up forwarding can be easily achieved with the use of a depth-first algorithm to traverse the live objects in the newer generation.

The main advantage of this approach over a standard hash-consing scheme is that it avoids the cost of a hash table lookup and insert for short-lived objects, that is, those objects that do not survive any garbage collection. The premise of generational garbage collection is that most allocated objects tend to have a very short life[18]. By eliminating replication, we reduce the size of the older generation. There will be no space savings in the newer generation, but we don't lose much here since the space of this generation will be reused after each "minor" garbage collection.

A conventional hash-consing implementation can test structural equality of two objects (Lisp "`equal`") by means of a simple pointer equality test (Lisp "`eq`"). Our scheme can do this only for objects in the older generation. To compare two arbitrary objects for structural equality, we must perform structural comparison for any part of the structure resident in the younger generation; once the recursion escapes into the older generation, it can be cut off with a pointer-equality test.

The main problem with our scheme is that garbage collection time may be considerably increased, and this may offset the gains from the reduction of replication in the older generation.

```
forward(R) =
    mark record pointed to by R
    for i := 1 to length(R) do
        if R[i] is a pointer into fromspace
        and record pointed to by R[i] is not marked then
            R[i] ← forward(R[i])
        endif
    endfor
    { all of R's fields are already forwarded }
    copy record pointed to by R to location NEXT
    R[1] ← NEXT
    NEXT ← NEXT + length(R)
    return R[1]
```

Figure 1: The Depth-First Search forwarding algorithm. `R` is a pointer to a record in the *fromspace*. `NEXT` is a pointer to the next free location of the reserve space. `length(R)` returns the length of the record pointed to by R.

# 4 A Depth-First Search Collector Algorithm

As explained in the previous section, a hash-consing garbage collector must traverse the heap "bottom-up." This suggests the use of a depth-first traversal algorithm. Since the SML/NJ compiler used a breadth-first algorithm, our first task was to rewrite the garbage collector.

Our depth-first search algorithm is not quite conventional (see Figure 1). To *forward* a record $R$ that is not yet marked we mark the record and recursively forward the fields of $R$. Only then we copy the record $R$ to to-space. A conventional depth-first copying algorithm copies the unforwarded record before recursively forwarding its fields. We choose this approach because it makes it posible to fit a stack in to-space. On the other hand, a conventional depth-first copy can correctly handle cycles; our algorithm may find marked records which do not yet have a forwarding pointer installed. We discuss cyclic structures below. For now, we will assume that the algorithm simply does not forward pointers to marked records, leaving those pointers pointing into from-space.

A nice property of the breadth-first algorithm is that it can use the records just copied as an implicit queue and, thus, does not require any auxiliary storage. Depth-first algorithms require either an auxiliary stack or a complicated pointer-reversal scheme [14]. However, if we use the scheme described above, we need not allocate any additional memory area for the stack. It turns out that there is just enough space in the to-space to put the stack. As shown in Figure 2, the stack grows downward from the high end of the to-space, while records are

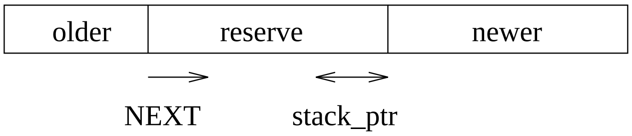


Figure 2: Location of the depth-first stack

copied starting at the low end. By ensuring that each record is popped from the stack *before* it is copied to to-space, we can guarantee that the stack occupies memory that would have been unused anyway.

In order to better explain the implementation of the stack in to-space, we present a non-recursive version of the depth-first search algorithm in Figure 3. Besides a pointer to the record which is currently being processed (`R`), the algorithm uses three other pointers: `NEXT`, which is a global variable that points to the next free location in the to-space; `current_field`, which points to the field of the record pointed to by `R` that is currently being processed and `stack-ptr` which points to the top of the stack. The bottom of the stack is location `MAX` which is the highest address of the destination region.

The stack starts at the end of to-space and grows backwards. It is not difficult to show that the stack pointer and the $NEXT$ pointer (which marks the end of the copied data) never overlap, as long as the to-space is large enough to hold all the live data. Each stack item is two words long, and each record (including its kind-and-size descriptor) is at least two words long. For each live record, there is at most one item on the stack or one copy in the to-space, but never both simultaneously.

Suppose that at a given time the stack contains $n$ records. By the algorithm of Figure 3, a record is copied to the reserve space only after all of its fields have been processed and at this point the record will not be pushed on the stack again (this is guaranteed by the statement `if current_field < length(R)`). Since the size of the stack is $2 * n$ words, the `stack-pointer` points to location `MAX` $- 2 * n$. Assuming that there is enough space to copy all records, `NEXT` can not be greater than `MAX` $- 2 * n$, since the size of each record is at least two words. It is always the case on a minor collection that there is enough space to copy all records, since the newer generation and the reserve space have the same size. On major collections, however, since the area used as the reserve space may be of size less than the *fromspace* (the *older generation*), it is possible that the pointers overlap. But this means that the garbage collector has run out of memory and the program execution is aborted.

# 5 Dealing with Cycles

The depth-first search algorithm given above does not handle cycles properly. It relies on the fact that, when copying record $R$, all of $R$'s children have already

```
forward(Root) =
    R ← Root
    stack_ptr ← MAX
    current_field ← 1
    mark record pointed to by R
    loop
        { R points to a marked record }
        while current_field < length(R)
        and R[current_field] is not a pointer into fromspace
        and record pointed to by R[current_field] is not marked do
            current_field ← current_field + 1
        endwhile
        if current_field < length(R) then
            push(R, current_field)
            R ← R[current_field]
            mark record pointed to by R
            current_field ← 1
            continue
        endif
        { all of R's fields are already forwarded }
        copy record pointed to by R to location NEXT
        R[1] ← NEXT
        NEXT ← NEXT + length(R)
        if stack is empty then
            return R[1]
        else
            new_address ← R[1]
            (R, current_field) ← pop()
            R[current_field] ← new_address
            current_field ← current_field + 1
        endif
    endloop
```

Figure 3: The Depth-First Search forwarding algorithm. `R` is a pointer to a record in the *fromspace*. `NEXT` is a pointer to the next free location of the reserve space. `MAX` is the highest address in the reserve space. `length(R)` returns the length of the record pointed to by R.

```
done ← false
while not done do
    done ← true
    for each ref-cell in the ref-list do
        if cell has been moved to to-space then
            for all pointers p in the cell do
                forward(p)
            endfor
            remove the ref-cell from the list
            done ← false
        endif
    endfor
endwhile
```

Figure 4: Forwarding ref-cells.

been forwarded; thus we were able to look up the (already forwarded and hashed) contents of $R$ to determine if a copying was even necessary. Clearly, we cannot use such a simple scheme when $R$ is its own descendant.

Every cycle, however, must contain at least one updateable record ("ref cell"). The ref cells should not be hash-consed anyway. Our solution to the cycle problem takes advantage of this. The idea is to stop the search on records that have been updated and use these records later as roots for the forwarding algorithm. When copying ref-cell $R$, its children will not necessarily have been forwarded; but we don't need to look up the forwarded contents of $R$ prior to copying, since we don't hash-cons ref cells.

The only problem now is how to find these updated ref-cells, but it turns out that the compiler already keeps a list of these cells (which it needs to handle updates to old objects in a generational garbage collector[1]) so this problem is very neatly solved.

The method works as follows. First we scan the ref list, marking all the updated ref-cells. Then we proceed with the usual forwarding of all the roots. Whenever a marked record is reached during the depth-first search, the record is copied to *to-space*. The pointers in that record are not forwarded, however. Later, after all roots have been forwarded, we scan the ref list again and forward the pointers of all records in the list that have been copied to *to-space*. Since new ref-cells may be copied in this step, several passes over the ref-list may be required, until no new ref-cell is copied. Figure 4 describes the last step of the algorithm.

The major drawback of this method is that it can have complexity proportional to the square of the number of distinct ref-cells in the list, in the worse case. [1]

An alternative approach is to copy all updated ref-cells to to-space in the beginning of a garbage collection and then use these records as roots for the depth-first collector. The problem with this approach is that not all updated ref-cells are live, which may result in many non-live records being copied. Therefore we have not pursued this alternative.

The list of updated cells is not available on major collections. But since we do not have to "hash-cons" during major collections we can use another method to handle the cycles problem on these collections. The idea is simply to copy a record with the pointer to the old copy, and fix all the incorrect pointers later in a second pass. These pointers are easy to identify after the collection is completed since they are the only ones which point into *from-space*. Therefore, when we are processing a record that contains a pointer to a marked record — i.e., we close a cycle — we copy the record without forwarding the pointers to marked records. The marked record is copied after it is popped from the stack.

To fix the incorrect pointers we can make a sequential pass over the *to-space*, or we can keep a list of records that need to be fixed and fix all the records in the list. Of course, the second approach is preferable since it is much faster. But in this case there is a problem with the space for the list of records. If there is not enough memory to keep this list, the only alternative is to make the sequential pass. A combination of the two approaches is also possible. If there is space to keep part of the list, we can fix the records in the list and start scanning *to-space* sequentially from the address after the highest address in the list. The only requirement is that records are added to the list in order of increasing addresses, which is easy to do. We can achieve some speed-up in the sequential pass if we mark the tag of the records that must be fixed. Then we do not need to read all the words in *to-space*, but only the tags of the records. In any case, this method can be potentially slow if the number of cycles is large compared to the number of records in the heap.

The two methods that we have described above can be potentially slow in some cases. However, for programs that are mostly functional, both methods can be expected to be reasonably efficient, since we would not have neither a large number of ref cells, nor a large number of cycles.

# 6 Hashing

Once we have the depth-first collector and the appropriate method to deal with cycles, adding the hash-table lookup becomes very simple. We need only change

[1] The size of the ref-list is proportional to the number of updated operations and not to the number of updated ref-cells. That is, a ref-cell may appear several times in the list if it is updated several times. This can make the size of the ref-list potentially large, even if the number of distinct ref-cells in the list is actually small. However, it is easy to remove the duplicate entries from the list when we make the first pass marking the ref-cells, so that its size is reduced to be proportional to the number of updated ref-cells.

the part of the algorithm that copies a record. Before copying a non-updatable record we do a lookup on the value of that record. If an already collected record with the same value is found, then we store the address of the already collected record in the first word of the record that is being collected and proceed without copying the record. If the lookup is unsuccessful, the record is entered into the hash table and copied. Updatable records are simply copied, without neither a table lookup nor entry into the table.

The implementation requires a modification of the following three lines of the Depth-first collector algorithm presented in Figure 3:

```
copy record pointed to by R to location NEXT
R[1] ← NEXT
NEXT ← NEXT + length(R)
```

these lines are replaced by:

```
X ← lookup(R)
if X is not null then
   R[1] ← X
else
   copy record pointed to by R to location NEXT
   R[1] ← NEXT
   enter record pointed to by NEXT in the hash table
   NEXT ← NEXT + length(R)
endif
```

We have used an open hashing scheme [15] to implement the hash table. A second hash function is used to handle collisions. The two hash values are computed as a function of the first few words of a record. Each table entry is either empty or contains a pointer to a record on the heap. During minor collections, there is no need to delete nor to change the value of a table entry, but only to insert new values. On major collections, however, because we garbage collect the older generation, we would have to update the hash table, since records may have their address changed or they may simply not survive the garbage collection. We have chosen to simply discard the old hash table and rehash everything. This also makes it possible to change the size of the hash table to adapt to changes in the live-data size of the program. After the major collection is completed, we compute a new size for the hash table and scan the older generation sequentially hashing all non-updatable records.

The implementation of the modified garbage collector takes about 930 lines of C code, compared to about 530 lines of the standard garbage collector.

## 7 Performance Evaluation

In order to evaluate the performance of our hash-consing scheme, we have measured the running time of some sample programs, and compared it to the running time of the same programs when compiled with the SML/NJ compiler without using hash-consing. The experiments were made on a DECstation 5000/240, with 64Mbytes of main memory, using version 0.75 of the SML/NJ compiler. A brief description of the benchmark programs is given next.

**Boyer** The Boyer theorem-proving benchmark [12] translated into SML by Kai Li and Andrew Appel.

**Knuth-Bendix** An implementation by Gerard Huet of the Knuth-Bendix completion algorithm, translated into SML by Xavier Leroy.

**Lex** A lexical-analyzer generator written by James S. Mattson and David R. Tarditi[4].

**Life** Reade's implementation of the game of Life [16].

**Mandelbrot** A program to generate Mandelbrot sets.

**Simple** A spherical fluid-dynamics program [9, 10], translated into SML by Lal George.

**VLIW** A VLIW scheduler written by John Danskin.

**Yacc** A LALR(1) parser generator, written by David Tarditi [17] processing the grammar of Standard ML.

Figure 5 gives the running time for each program, both with and without the use of hash-consing. Times are normalized to the execution time without hash-consing. The total running time is divided into garbage-collection time (*gc-time*) and non-garbage-collection time (*non-gc-time*). The results show that the hash-consing scheme slows down garbage collection by a factor of up to four times in the worst case, but for most programs the figure is of about two times or less. In many cases there is a small improvement in *non-gc-time* but, with the exception of `Boyer` and `Knuth-Bendix`, the gains are insufficient to compensate the overhead of garbage collection. `Yacc` has the worst performance, being 22 percent slower. The Boyer benchmark showed the highest increase in performance, almost 10 percent. This agrees with experiments made by Baker [5], who shows that function memoization and hash-consing can greatly improve the performance of the Boyer program (memoization seems to be more important than hash-consing).

The SML/NJ compiler uses a parameter—the *heap ratio*—that controls the ratio of total heap memory to live data size. This parameter is used to adjust the size of the heap after a major garbage collection. The *heap ratio* determines

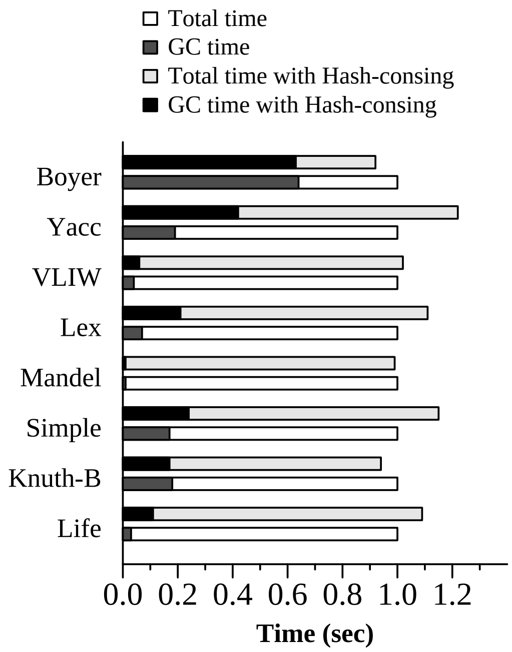


Figure 5: Running time for the programs in the benchmark with and without the use of hash-consing. For each program, times are normalized to the execution time without hash-consing.

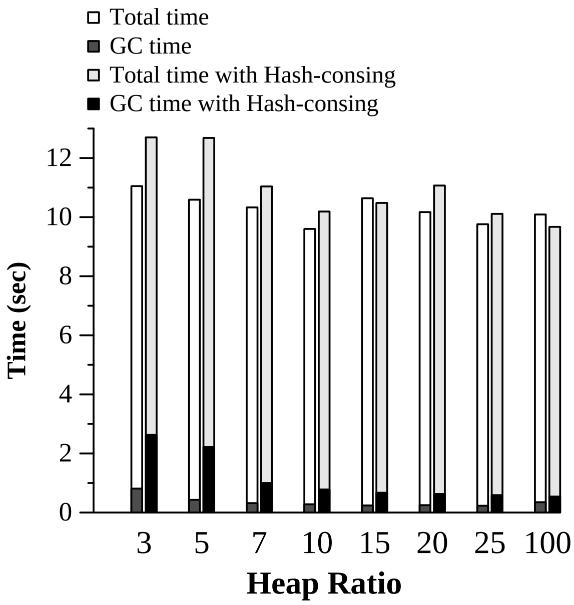


Figure 6: Effect of different heap ratios on the running time of `Lex`.

the frequency of collections and, thus, may have a great impact in the running time of a program. The default value of five was used in the measurements of Figure 5.

Figure 6 shows the impact of different values for the *heap ratio* on the running time of `Lex`. Since larger ratios lead to smaller collection time, a system whose collector is slowed by hash-consing may have better overall performance with a larger heap ratio.

Figure 7 shows the impact of hash-consing on heap usage by the programs in the benchmark. The first column gives the total amount of data copied into the older generation without hash-consing. The second column gives the same measurement with the use of hash-consing. The numbers in the third column give the amount of duplicate data found by the hash-consing collector. All numbers are in megabytes. All programs, with the exception on `Lex`, have a small reduction on their data size. In most cases, however, this reduction is quite small, less than one percent.

The next measurements show the impact of different hash table sizes on the total running time of the programs. The size of the hash table as a function of the heap size is a parameter that can be adjusted by the programmer in the same way as the *heap ratio*. The results, presented in Figure 8, show that the size of the hash table does not have a great impact in the performance of the programs in the benchmark.

| | Without hash-consing | With hash-consing | |
|---|---|---|---|
| Program | Data size(Mb) | Data size(Mb) | Duplicates (Mb) |
| Boyer | 153.0 | 130.2 | 3.340 |
| Life | 172.1 | 171.8 | 0.066 |
| Knuth-B | 181.6 | 178.5 | 0.354 |
| Simple | 406.3 | 404.1 | 0.452 |
| Mandel | 1482.8 | 1482.7 | 0.000 |
| Lex | 112.4 | 112.9 | 0.007 |
| VLIW | 235.2 | 234.9 | 0.008 |
| Yacc | 52.1 | 51.0 | 0.256 |

Figure 7: Heap Allocation and Duplicate Data

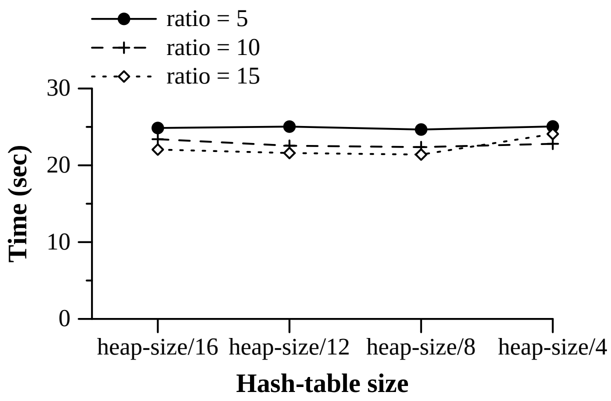


Figure 8: Effect of varying the hash table size on the running time of `Lex`.

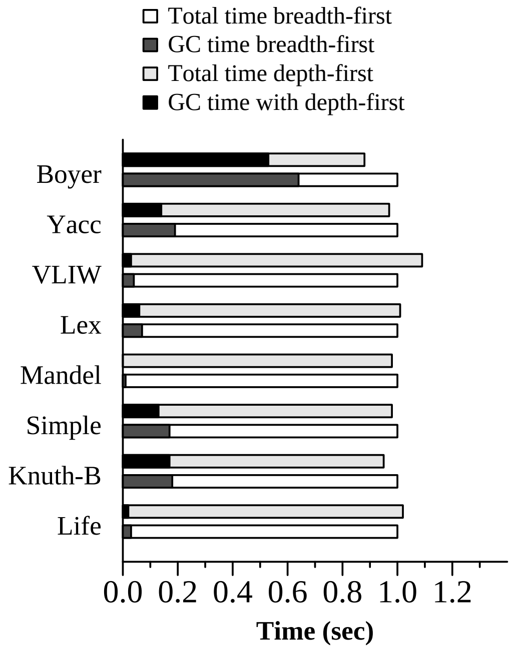


Figure 9: Performance comparison between the breadth-first and depth-first search garbage collectors. For each program, times are normalized to the execution time for the breadth-first collector.

Finally, Figure 9 compares the performance of the breadth-first collector and the depth-first collector, without hash-consing. The traversal algorithm may have an impact on the locality of references of the collected data and thus affect the performance of programs [19]. Our measurements do not show any significant differences. Perhaps this is because the DECstation's four-word cache blocks are too small to hold much more than one object each (and our programs exhibit minimal virtual-memory paging).

# 8 Future Work

Perhaps the real promise of hash-consing is that it can make function memoization [8] possible. If a pure (side-effect free) function $f$ is applied to an argument $x$ yielding a result $y$, then the next time $f(x)$ is evaluated the same result will obtain. (By side-effect free, we mean that reference cells are neither read nor written.)

If the mapping $(f, x) \mapsto y$ is entered into a table, then the next time the same function is applied to the same argument, the value $y$ can be returned without evaluating $f$. This is the essence of *function memoization*.

Without a hashing scheme, searching for $(f, x)$ in the table is nontrivial, especially if $x$ is a deeply structured value. With a hash-consing scheme, in which the values $f, x$ already have been hashed to unique addresses, it should be possible to do function memoization with very little overhead beyond what is reported in this paper. To avoid hashing on all function calls ever performed, the programmer should probably specify which functions are to be memoized.

Standard ML does have side effects, however, and these might cause the memoizer to give semantically incorrect results. We propose a scheme to solve that problem. A global counter $S$ will be incremented on every read or write side effect. When $f(x)$ is first invoked, the tuple $(f, x, S)$ will be recorded. When $f$ returns the result $y$, the current value of $S$ will be compared with the original value. If they are the same, then $f(x)$ had no read or write side effects, and $f$ is guaranteed (at least on input $x$) to be a pure function. Thus, $(f, x) \mapsto y$ can be entered in the memo table. No guarantee is implied that $f$ is pure when applied to other arguments, but none is needed.

Baker[5, 6, 7] has shown that memoization (in combination with hash-consing) is extremely powerful in improving the performance of programs such as the Boyer benchmark, which proves theorems by backtracking search.

# 9 Conclusion

A hash-consing garbage collector is about a factor of two or three slower than an ordinary collector; but garbage collection takes only about 5% of execution time in a typical modern system, so the overall overhead of hash-consing may be affordable for many applications.

The hash-consing collector was not difficult to implement, and did not impinge on any part of the compiler outside the runtime system.

The space savings in most of the benchmarks we measured was not very impressive. The real utility of hash-consing may involve memoization, or other ways to let programmers make use of the ability to hash structures to integers efficiently and in a general way.

# References


[1] Andrew Appel. Simple generational garbage collection and fast allocation. *Software - Practice and Experience*, 19(2):171–183, February 1989.

[2] Andrew Appel. A runtime system. *Lisp and Symbolic Computation: An International Journal*, 3:343–380, 1990.

[3] Andrew Appel and David MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.

[4] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyser. Distributed with Standard ML of New Jersey, December 1989.

[5] Henry Baker. The Boyer benchmark at warp speed. *Lisp Pointers*, 5(3):13–14, July 1992.

[6] Henry Baker. The Gabriel 'Triangle' benchmark at warp speed. *Lisp Pointers*, 5(3), July 1992.

[7] Henry Baker. A tachy 'TAK'. *Lisp Pointers*, 5(3), July 1992.

[8] Richard S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[9] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, February 1978.

[10] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T.J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[11] A. P. Ershov. On programming of arithmetic operations. *CACM*, 1(8), 1958.

[12] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, MA, 1985.

[13] Eiichi Goto. Monocopy and associative algorithms in extended lisp. Technical Report TR 74-03, University of Tokyo, 1974.

[14] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[15] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[16] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, MA, 1989.

[17] David R. Tarditi and Andrew W. Appel. ML-Yacc, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

[18] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, Cambrigde, MA, 1986.

[19] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proc. ACM SIGPLAN'91 Conf. on Prog. Lang. Design and Implementation*, pages 177–191, June 1991.