ACM Sigplan *Lisp Pointers* VI,4 (October-December 1993), 3-10.

# The Boyer Benchmark Meets Linear Logic

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436 (818) 986-1436 (818) 986-1360 (FAX)

This material is based upon work supported by the National Science Foundation under Grant No. III-9261682.

Of the Gabriel Lisp Benchmarks, the Boyer Benchmark ("Boyer") is the most representative of real AI applications, because it performs Prolog-like rule-directed rewriting, and because it relies heavily on garbage collection (GC) for the recovery of storage. We investigated the notion that such programs are unsuitable for explicit storage management—e.g., by means of a "linear" programming style in which every bound name dynamically occurs exactly once. We programmed Boyer in a "linear" fragment of Lisp in both interpretive-rule and compiled-rule versions, using both true linear (unshared) and reference count methods.

We found that since the intermediate result of `rewrite` is unshared, the linear interpreted version is slower than the non-linear interpreted version, while the linear compiled version is slightly faster than the non-linear compiled version. When sharing is allowed, requiring reference counts for the linear versions, the linear shared versions are not competitive with the non-linear versions, except when "anchored pointers" are used. The anchored pointer reference count version, which reclaims storage, is still 1.25X slower than the non-linear version, which reclaims no storage.

## INTRODUCTION

C programmers deride Lisp programmers as lazy, because they refuse to specify explicitly when storage is to be released. Lisp programmers rightly retort that C is unsafe, and curse the crashes of their C-based operating systems. Both camps assume that explicit storage management may be more efficient, but also more dangerous. This trade-off between efficiency and safety is particularly difficult in "production" code, where customers will not abide either inefficient or crash-prone code. Lisp systems currently offer great security, but extreme efficiency is gained only by going outside the language and its type system, where a single misstep may result in disaster. Lisp needs to allow the tuning of programs through additional programmer effort in ways that will not compromise security. We believe that the *linear* style of programming offers such a way.

Lisp implementors have long known that most list cells have a reference count of 1 [Clark77]—i.e., most cells are unshared. Unshared cells can have particularly efficient deallocation policies. Unfortunately, determining at compile time whether a cell is shared may be as expensive as ML type inference [Baker90], which is known to have exponential time complexity [Mairson90]. *Linear logic* [Girard87] [Lafont88] [Abramsky93] [Wadler91], on the other hand, exploits unshared cells by typing them as unshared, and the type system then preserves this property.

Since we are interested in understanding the advantages and disadvantages of the "linear" style of programming, we have tested this style on several different "real" programs [Baker93,Baker94a,b]. We were particularly drawn to the *Boyer Benchmark* ("Boyer"), because it is the most representative of all the Gabriel Lisp benchmarks [Gabriel85]. Boyer is a classic Lisp program, because it recursively examines and constructs lists at a great rate. Boyer proves that a formula is a tautology by continually rewriting it until it is in a normalized form, and then testing whether this normalized form is a tautology. This rewriting utilizes a database of rewrite rules (axioms), whose left-hand-sides are matched (unified) against the formula, and when a match occurs, the formula is rewritten utilizing the right-hand-side as a pattern. This rewriting is analogous to the operation of a Prolog system in which the rules act as Prolog predicates and the initial formula acts as the initial form to be "evaluated". There is a major difference, however, in that (the standard) Boyer interprets these rules on-the-fly rather than compiling them.

Our present interest in a linear version of Boyer is driven not so much by speed or space *per se*, since a *memoized* Boyer is two orders of magnitude faster and smaller than the standard Boyer [Baker92]. We are more interested in the effect of copying versus sharing on total storage requirements of a non-memoized Boyer, and in whether more explicit storage management is onerous relative to automatic garbage collection.

## STANDARD (NON-LINEAR) INTERPRETED BOYER BENCHMARK

The code in [Gabriel85] has several bugs. `assq` is not defined, and `falsep` and `truep` need fixing:

```
(defmacro assq (x y) `(assoc ,x ,y :test #'eq))

(defun falsep (x lst) (or (equal x '(f)) (member x lst :test #'equal)))

(defun truep (x lst) (or (equal x '(t)) (member x lst :test #'equal)))
```

The `prog` in the function `test` must be changed into a `let` to report the answer. Some Lisps—e.g., `xlisp`—don't provide `nil` with a property list, which Boyer requires. Rewrite rule #84 should have `greatereqp` instead of `greatereqpr`, although this rule is never used. Rule #101 should have `(t)` and `(f)` instead of `t` and `f`, respectively, in the right-hand-side, although this rule never matches.

The most serious problem is with rule #54—`(equal (remainder y 1) (zero))`—which requires a *constant* `1` as `remainder`'s second argument. `one-way-unify1` incorrectly treats the atom `1` like a *variable*, however, so that this rule matches *any* use of `remainder`, replacing two occurrences in the benchmark with `(zero)`. After `one-way-unify1` is fixed, the `remainder` rules—#54, #56, #67, #86—are called 2,003 times instead of only twice; consing is increased by 12%, and `one-way-unify1` now calls `equal` 7,053 times instead of the previous 1,046 times. *Fixing this bug increases the time on the standard Boyer Benchmark by 10-25%.*[1] From now on, we use the term "standard Boyer" to refer to Boyer with this bug fixed, and "buggy Boyer" to refer to the code published in [Gabriel85].

```
(defun one-way-unify1 (term1 term2)                                  ; With bug fixed.
  (cond ((atom term2)
         (cond ((setq temp-temp (assq term2 unify-subst)) (equal term1 (cdr temp-temp)))
               ((numberp term2) (equal term1 term2))                  ; This clause is new.
               (t (setq unify-subst (cons (cons term2 term1) unify-subst)) t)))
        ((atom term1) nil)
        ((eq (car term1) (car term2)) (one-way-unify1-lst (cdr term1) (cdr term2)))
        (t nil)))
```

Essentially all of Boyer's time and space are used in rewriting; the resources required for tautology testing are relatively insignificant. The Boyer rewrite system includes 106 rewrite rules, each with a "left-hand-side" pattern to be matched, and a "right-hand-side" pattern which shows how a matching expression is to be rewritten. Of these 106 rules, 49 rules are tried in the standard benchmark, and only 8 rules ever succeed (#18,#21,#23,#25,#33,#34,#35,#36). Rule #23 (`if`) is attempted 21,491 times and succeeds 934 times, rule #25, #51 and #95 (`plus`) are each attempted 5,252 times and rule #25 succeeds twice. The rules are indexed by their outermost function symbol on the property lists of 58 symbols; `equal` has the most rules (14), `difference` and `lessp` have 7 rules, and 15 function names each have but one rule associated with them. The rules are tried in reverse order of their occurrence in the text, and the benchmark is sensitive to this ordering—one alternative ordering increased running time by 40%. Storage for the rules themselves account for 1,849 list cells (without sharing).

The intermediate result of rewriting in standard Boyer contains 49,747 cells, all of them unique! In other words, the classical reason for preferring non-linearity is to enable sharing, but `rewrite` produces completely unshared data structures! The reason is that `rewrite` makes a fresh copy of its argument even when no rules have been successfully applied. To determine the potential for sharing in `rewrite`, we tested code similar to the following:

```
(defun scons (a d c)              ; c is cell whose car might be a and whose cdr might be d.
  (if (and (eq (car c) a) (eq (cdr c) d)) c     ; if car&cdr match, use c instead of new cell.
    (cons a d)))                                                   ; scons should be inlined.

(defun rewrite (term)                                                  ; rewrite with sharing.
  (if (atom term) term
    (rewrite-with-lemmas (scons (car term) (rewrite-args (cdr term)) term)    ; scons v. cons.
                         (get (car term) 'lemmas))))

(defun rewrite-args (lst)                                         ; rewrite-args with sharing.
  (if (null lst) nil
    (scons (rewrite (car lst)) (rewrite-args (cdr lst)) lst)))                ; scons v. cons.
```

The use of `scons` within `rewrite` and `rewrite-args` was extremely successful: it won 171,878 times out of 173,804 tries—a 98.9% "hit rate". This use of `scons` eliminated 68% of all Boyer consing and dramatically increased sharing: the result of `rewrite` now required only 6,362 distinct cells instead of 49,747 cells, although both structures are `equal`. This reduction in storage usage produced a 23% reduction in execution time.[2] We tried `scons` within `apply-subst` and `apply-subst-lst` and got a further sharing of only 23 cells (6,339 cells), although with a slightly increased running time. We thus found that the sharing implicit in non-linear styles of programming can be valuable for Boyer, but only when it is properly exploited!

## LINEAR INTERPRETED BOYER BENCHMARK

The translation of Boyer into a "linear" fragment of Lisp is quite straightforward—whenever a bound variable occurs more than once in a context, the variable must be explicitly duplicated, and whenever a bound variable is not utilized

---

[1] Since our rule compiler performs this check correctly, all of our "compiled" times are with this bug fixed.

[2] The `scons` function was inlined in every version that we timed.

in a context, the variable must be explicitly killed. It is interesting to note the places in Boyer where these explicit duplications and kills must be inserted. A portion of Boyer copies and performs substitutions according to an "association list" which records the mapping of a variable name to its value. Obviously, if the variable occurs more than once in the expression, its associated value must be copied when it is substituted; similarly, if the variable does not occur, then its associated value must be destroyed. Unfortunately, since the substitution patterns with their variable occurrences have not been compiled, the number of occurrences of each variable is not known in advance, and therefore each value must be copied at least one extra time before it is discovered that it will not be needed and must then be explicitly destroyed. This is an interesting case of non-linearity in an interpreted pattern being reflected back into non-linear behavior in the interpreter itself!

A similar situation occurs elsewhere in Boyer. When the database of rewrite rules is consulted, the set of rules indexed by a particular function symbol must be "copied", even though none of the rules may match. Furthermore, once a matching rule has been found, the rest of the rules for that function symbol are explicitly discarded, since the new function symbol to be rewritten is most likely different from the previous function symbol. Since most rule matches fail, and fail before a significant fraction of the rule has been examined, it is most efficient to keep only one copy of each rule, which copy is decomposed during matching and then reconstructed for later use.

The non-functional style of updating the variable association list in `one-way-unify1` was changed into a functional style. In particular, the tasks of `one-way-unify1` were split into three functions: `one-way-unify1`, which simply matches shapes, `one-way-unify2`, which constructs an association list, and `check-alist`, which checks that duplicate variables have equal values. This new style is slower than the non-functional style, but only slightly, since shapes very seldom match, in which case the construction of association lists is not necessary.

The strictly linear (unshared) style for Boyer was 1.58X slower than the non-linear Boyer, which is due primarily to the fact that linear Boyer has to recirculate many argument values as multiple result values which can be quite expensive. Furthermore, the linear interpreted Boyer performed one excess copy for each substitution performed, because it could not tell that there were no more occurrences of the variable.

We then programmed a traditional reference count implementation "underneath" our linear Boyer, by having `dup` increment reference counts instead of copying, having `kill` decrement reference counts instead of recycling, and having `dlet*` adjust reference counts on cells it is dissolving. This simple reference count implementation is not competitive, being about 2.9X as slow as the non-linear Boyer, and 1.62X slower than the strictly linear Boyer. This slowdown is worse than expected, since the copying involved during substitution is replaced by a simple reference count increment. One reason for this difference is the change from the built-in `cons` cell to a `defstruct` cell; type testing for `defstruct` cells is substantially slower in Coral Common Lisp than testing for `consp`. Another reason is that most reference count updates involve a type test in addition to the increment itself. Additionally, the bulk of the work in Boyer is *unsuccessful* pattern matching, so the costs involved in manipulating reference counts on the patterns and values far exceeds the slight savings during substitution. Finally, the next call to `rewrite` eliminates the sharing created by substitution.

We then used the concept of "anchored pointers" [Baker93AP] to further optimize our linear reference counted Boyer. An *anchored pointer* is similar to a "derived" or "weak" pointer in that it does not protect its target from being reclaimed. However, if the target is already protected by an *anchor*, then the anchored pointer will always point to an accessible target. Anchored pointers can be more efficient than normal pointers for traversing data structures because reference counts do not have to be manipulated. Anchored pointers can be safely used only within a dynamic context in which the anchor holds, and any anchored pointer which escapes this context must first be *normalized* by incrementing the target's reference count.

Anchored pointers are perfect for most of the manipulations involved in interpreted Boyer. The rules themselves are anchored, and during the pattern match, the expression to be matched is anchored, as well. However, the argument and result of `rewrite` are normal pointers, because we want the argument to `rewrite` to be consumed in the production of its result, which is also normal. The arguments to the function `apply-subst` are anchored pointers, but its value is normal, because it is newly consed.

The linear reference counted Boyer with anchored pointers was more than twice as fast as the simple reference counted Boyer, and 1.37X faster than the strictly linear Boyer without reference counts. The linear reference counted Boyer was 1.42X slower than the standard non-linear Boyer. When sharing (`scons`) was used, the linear reference counted Boyer was 1.28X slower than the standard non-linear Boyer using `defstruct` cells.

## NON-LINEAR COMPILED BOYER BENCHMARK

While the standard Boyer with its interpreted rules is a good benchmark for Lisp prototyping, it is less effective at predicting performance on a production program, which would use compiled rules [Boyer86]. A compiled rule can be more efficient, since it knows in advance the pattern to be matched, and does not have to access the list cells of either pattern. If all of the rules which are indexed to a particular function symbol are compiled together, then the optimizations familiar to any Prolog implementor can be achieved.

Our rule compiler compiles each rule separately, and does not therefore take advantage of important inter-rule optimizations. The compiler consists of two parts—a pattern match compiler, and a pattern-directed constructor, both classical. Our non-linear rule compiler uses sharing for subexpressions of the input which appear in the output, as well as for multiple copies in the output. Each compiled rule is a function with one argument—the expression to be rewritten—and 2 results—a truth value which is false if no match occurred, and the rewritten expression. Thus, when `(funcall <rule> <exp>)` fails, it returns `nil,<exp>`. The non-linear compiled rules are similar to the linear compiled rules below, except that upon failure the input expression is not reconstituted.

The rule compiler eliminates the consing required to construct association lists during pattern matching, which saves 69,208 conses, or about 27% of the total, and also runs 1.32X faster than the interpreted version. We also tested a "shared cons" (`scons`) compiled version, which eliminated another 171,878 conses, leaving 13,372 conses. This non-linear shared compiled version ran 1.34X faster than the non-`scons` compiled version.

## LINEAR COMPILED BOYER BENCHMARK

When compiling rules into a *linear* Lisp, additional optimizations are possible. The number of occurrences of a variable in a pattern can be determined in advance, as can the number of additional occurrences required during the actual rewriting. Furthermore, since a strictly linear pattern matcher destroys the portion of the expression which matches, the cells recovered during this decomposition can be reused for the construction of the rewritten expression. In this case, the cell is never actually put back onto the freelist, but is immediately reused by the constructor code.

Our linear rule compiler also compiles each rule separately. The compiler consists of *three* parts—a pattern match compiler, a pattern-directed constructor, and a copy analyzer which determines how many copies of a matched variable are kept or produced for the constructor. The pattern-directed constructor is classical, except that it is given a list of names of cells it is allowed to reuse during construction. The pattern match compiler is also classical, except that when a non-match occurs, the input expression must be explicitly re-constituted. The copy analyzer is novel in the linear style, which requires that any copying and deletion be made explicit. Our linear rule compiler is strictly linear, and does not use reference counts or sharing, so that all copying is "deep" copying.

A complete rule is a more convenient modularization for the linear style than is the separation of matching and construction found in the interpreted Boyer. Complete rules allow the compiler to more efficiently reuse cells reclaimed from the input expression in the construction of the output expression. Below is an example of the compiled version of the rule `(equal (foo x y x) (bar y y))`:

```
(defun rule-foo-1 (e)                        ; compiled from rule (equal (foo x y x) (bar y y))
  (if-atom e (values nil e)
    (dlet* (((fn . e1) e))                              ; Acquire first cell of input list.
      (if-atom e1 (values nil `(,fn ,@e1))            ; These conses reuse the input cells.
        (if-neq 'foo fn (values nil `(,fn ,@e1))      ; These conses reuse the input cells.
          (dlet* (((x1 . e2) e1))                      ; Acquire second cell of input list.
            (if-atom e2 (values nil `(,fn ,x1 ,@e2))   ; These conses reuse the input cells.
              (dlet* (((y1 . e3) e2))                   ; Acquire third cell of input list.
                (if-atom e3 (values nil `(,fn ,x1 ,y1 ,@e3))             ; Reuse input cells.
                  (dlet* (((x2 . e4) e3))              ; Acquire fourth cell of input list.
                    (if-natom e4 (values nil `(,fn ,x1 ,y1 ,x2 ,@e4))     ; Reuse input cells.
                      (if-nequal x2 x1 (values nil `(,fn ,x1 ,y1 ,x2 ,@e4))     ; Reuse cells.
                        (let* ((y1 y2 (dup y1)))                 ; Acquire copy of input y.
                          (kill fn) (kill x1) (kill x2)          ; Start freeing resources.
                          (values t `(bar ,y1 ,y2)))))))))))))) ; These conses are optimized.
```

This compiled code for a rewrite rule bears a close resemblance to concurrent programs which perform "two-phase locking" [Gray78]. In two-phase locking, a program's first phase acquires all the resources it needs, so that during its second phase, it will be guaranteed to succeed. In the code above, the rule first "acquires" the entire input expression that matches the pattern, followed by the "acquisition" of a duplicate of the y variable. If at any time during the first phase the "resources" cannot be acquired, then the rule can fail without "deadlocking". The second phase is guaranteed to succeed, because y has already been copied, and the number of cells required for the right-hand-side of the rule (3) is fewer than those appearing in the left-hand-side (4). If we make the analogy between these rewrite rules and chemical reactions which can happen in parallel [Berry92], then the two-phase locking analogy becomes very intuitive.

This compiled linear Boyer uses strict copying, so each call to `dup` causes its argument to be "deep" copied. Since most people expect this copying to be expensive, we collected statistics for this Boyer benchmark. There were 1,868 calls to `dup` on non-atoms, with a mean size of 24.5 cells, a max size of 1801 cells and a standard deviation of 64.1 cells. The total number of cells copied is 45,784, which accounts for most of the 49,747 cells which appear in the result of `rewrite`. The rest of the cells (3,963) come from incremental additions by the successful rules—e.g., the 934 successful applications of rule #23 add 934*4=3,736 additional cells.[3] Since the 8 successful rules cause neither deletions (`kill`'s) nor incremental reductions in the cell count, every cell is accounted for.

A major speed difference between the standard (non-linear) Boyer and our linear Boyer is the speed of `tautologyp`. The non-linear `tautologyp` takes about 12 times longer than the linear `tautologyp`. The reason for this discrepancy is instructive. The linear `tautologyp` on an argument of 49,747 cells takes only 14% longer than `kill` on the same argument, and the absolute difference is only marginally greater than the time for non-linear `tautologyp`. Thus, the time for linear `tautologyp` is just the time to test a tautology, plus the time to reclaim all the cells of its argument.

## RESULTS

The following results were obtained on an Apple Mac+ with 4 MBytes of memory and a 68020 accelerator running Coral Common Lisp version 1.2. The different versions use the following labels:
NL — non-linear (garbage collected) using standard `cons` cells.
NL3 — non-linear (garbage collected) using `defstruct` cells.
L — linear (deep copy) using standard `cons` cells.
LRC — linear sharing using reference counts; uses `defstruct` cells.
LRCAP — linear sharing using reference counts and anchored pointers; uses `defstruct` cells.

Note that compiled versions (C) have the bug fixed (BF), and strict linear (L) versions cannot represent sharing (S). Interpreted versions (I) do not count the number of cells used to represent the rules themselves (1,849 cells).

| Memory Management | Interpreted (I) Compiled (C) | Bug (B) Bug Fixed (BF) | Unshared (US) Shared (S) | `rewrite` Time w/o GC | Result Size | Total Allocated |
|---|---|---|---|---:|---:|---:|
| NL | I | B | US | 24.967 s. | 48,139 cells | 226,436 cells. |
| NL | I | B | S | 21.167 s. | 6,362 cells | 58,566 cells. |
| NL | I | BF | US | 28.050 s. | 49,747 cells | 254,458 cells. |
| NL | I | BF | S | 24.417 s. | 6,337 cells | 82,580 cells. |
| NL | C | BF | US | 21.250 s. | 49,747 cells | 185,250 cells. |
| NL | C | BF | S | 15.900 s. | 6,337 cells | 13,372 cells. |
| NL3 | I | BF | S | 25.367 s. | 6,337 cells | 82,580 cells. |
| NL3 | C | BF | S | 16.533 s. | 6,337 cells | 13,372 cells. |
| L | I | BF | US | 44.400 s. | 49,747 cells | 50,204 cells. |
| L | C | BF | US | 20.783 s. | 49,747 cells | ~52,000 cells. |
| LRC | I | B | US | 72.000 s. | 48,139 cells | 49,880 cells. |
| LRCAP | I | B | US | 33.133 s. | 48,139 cells | ~66,000 cells. |
| LRCAP | I | B | S | 27.333 s. | 6,362 cells | ~8,000 cells. |
| LRCAP | I | BF | US | 39.767 s. | 49,747 cells | 66,208 cells. |
| LRCAP | I | BF | S | 32.400 s. | 6,337 cells | 7,544 cells. |
| LRCAP | C | BF | S | 20.700 s. | 6,337 cells | 7,478 cells. |

One of the more interesting comparisons is between NL3,C,BF,S at 16.533 secs. and LRCAP,C,BF,S at 20.7 secs. Both of these tests use exactly the same code, except that the reference count updating and cell recycling has been removed from the non-linear version. The reference counted version is 1.25X slower than the non-linear version, meaning that fully 1/5 of all of the work went into reference count updating and cell recycling. Recycling the cells themselves takes only about 1/4 of that difference, or about 1/20 of the total effort. The rest of the difference, or about 15% of the total effort, is devoted to reference count updating. This test performed 7,584 normalizations, of which 7,082 were non-atoms; this number is probably a minimum for the amount of sharing which actually occurs Thus, although anchored pointers reduce reference count overhead, it is still a substantial fraction of the total effort.

The difference between NL3,C,BF,S at 16.533 secs. and NL,C,BF,S at 15.9 secs. is entirely due to the representation of the cells and the type testing. The NL3 version uses `defstruct` cells, while the NL version uses the more efficient built-in `cons` cells.

---

[3] The statistics of Quicksort [Baker94QS] support the cell reuse optimization far better than these Boyer statistics do.

 

## CONCLUSIONS

To summarize our conclusions:

- Boyer Benchmark is a kind of worst case for the linear style
- linear styles generally use less storage—even when deep copying
- strict copying linear style can be remarkably efficient, due to the elimination of most run-time decisions
- linear style forces all work to be done by the processor, including trivial copies usually made by the memory
- reference counting, even after being minimized by linearity and anchored pointers, can still be expensive
- reference counting may require hardware help in memory systems to be competitive
- the true cost of reference count updating is a store-to-store increment, *plus a type check*
- linear style requires heavy use of multiple returned values, which must therefore be efficient
- linear style definitely requires a linearity checking tool

If one wishes to execute the fastest possible Boyer benchmark, one should compile the rules [Boyer86] and utilize memoization [Baker92]. Such a scheme requires only about 1% of the time and space of the standard Boyer, and is clearly superior. However, we wanted to estimate the performance to be expected from problems that could not be successfully memoized. Rule compilation provides an overall performance improvement of perhaps 1.4X. Since rule matches are hundreds of times more common than rule rewrites, one can focus most optimizations on matches. Compiling rule matches saves the cost of destructuring the pattern as well as the expression, and about doubles the speed of this operation. Significant additional performance improvements are possible through compiling all patterns as a group, but we did not pursue it because this optimization is orthogonal to issues of storage management.

The lack of sharing and explicit copying was not as expensive as we had expected. The explicit copying, compiled rule Boyer is 1.02X faster than the unshared non-linear compiled Boyer, is only 1.004X slower than the shared, reference-counted Boyer, and is only 1.31X slower than the fastest shared, non-linear Boyer, which does no storage reclamation at all. There are several reasons for this efficiency in the strict linear (unshared) version. First, the standard Boyer does not preserve sharing, anyway. Second, the static, *a priori* knowledge that a cell is unshared means that when it is dissolved, the cell can be immediately recycled without any checking. Third, linearity allows for reallocation of a recycled cell by the rule compiler without it ever appearing on the freelist. Fourth, recycling during dissolution means that most cells are recycled by mutator traversal rather than requiring a separate collector traversal. Thus, data structures must be heavily shared before explicit copying becomes inefficient.

Of course, sharing can be remarkably efficient for Boyer, since a hash cons table implementation of Boyer uses only 146 cells to represent the intermediate result of rewriting, compared with 48,139 cells[4] for an unshared representation [Baker92]! However, our current results show that the sharing which is important for this compression is obtained by exploiting "may share" information (exploited by `scons` and hash consing) rather than "must share" information (traditional non-linear sharing) which utilizes 6,337 cells—almost an order of magnitude smaller than 49,747, but 1.6 orders of magnitude more than 146. It therefore appears that the scheme suggested in [Baker92LLL] which uses both reference-counted linearity and hash consing may be an optimum point for Boyer-like symbolic computations.

Our tests do not settle the question about which kind of storage management is best for Boyer. They do seem to indicate that unless reference counts can dramatically reduce cache misses, reference counting will need hardware help in order to be competitive with other techniques. We could not measure the "high water mark" storage requirements for the non-linear versions which rely on tracing garbage collection for cell reclamation, but we believe that linear and reference-counted versions show substantial reductions in the high water mark compared with non-linear versions.

Most traditional Lisp optimizations have focussed on minimizing execution time for both a computation and its garbage collector. With modern RISC architectures having one- and two-level caches, however, minimizing time requires minimizing space, because an "off-chip" access can cost upwards of 100 times as much as an on-chip access. The linear style does seem to minimize storage requirements, and should produce fewer cache misses. Our timings could not reflect this, however, since our experiments were performed on an architecture without a cache.

## REFERENCES

Abramsky, S. "Computational interpretations of linear logic". *Theor. Comp. Sci. 111* (1993), 3-57.

Baker, H.G. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". ACM *Lisp & Funct. Progr.* (1990), 218-226.

Baker, H.G. "The Buried Binding and Dead Binding Problems of Lisp 1.5: Sources of Incomparability in Garbage Collector Measurements". ACM *Lisp Pointers V*,2 (Apr.-June 1992), 11-19.

[4]This number is not 49,747 because [Baker92] did not include the `one-way-unify1` fix discussed earlier in this paper.

 

Baker, H.G. "The Boyer Benchmark at Warp Speed". ACM *Lisp Pointers V*,3 (July-Sept. 1992), 13-14.
Baker, H.G. "Lively Linear Lisp — 'Look Ma, No Garbage!'". ACM *Sigplan Notices 27*,8 (Aug. 1992), 89-98.
Baker, H.G. "NREVERSAL of Fortune—The Thermodynamics of Garbage Collection". *Int'l. W/S on Memory Mgmt.*, St Malo, France, Sept. 1992, Springer LNCS 637.
Baker, H.G. "Minimizing Reference Count Updating with Deferred and Anchored Pointers". Manuscript, Nov., 1993.
Baker, H.G. "Sparse Polynomials and Linear Logic". ACM *Sigsam Bulletin* 27,4 (Dec. 1993), 10-14.
Baker, H.G. "A 'Linear Logic' Quicksort". ACM *Sigplan Notices* 29,2 (Feb. 1994), 13-18.
Baker, H.G. "Linear Logic and Permutation Stacks—The Forth Shall Be First". ACM Sigarch *Computer Architecture News* 22,1 (Mar. 1994), 34-43.
Barth, J. "Shifting garbage collection overhead to compile time". *CACM 20*, 7 (July 1977),513-518.
Boyer, R. "Rewrite Rule Compilation". TR AI-194-86-P, M.C.C., Austin, TX 1986.
Berry, G., and Boudol, G. "The chemical abstract machine". *Theor. Comp. Sci. 96* (1992), 217-248.
Carriero, N., and Gelernter, D. "Linda in Context". *CACM 32*,4 (April 1989),444-458.
Chirimar, J., *et al*. "Proving Memory Management Invariants for a Language Based on Linear Logic". *Proc. ACM Conf. Lisp & Funct. Prog.*, San Francisco, CA, June, 1992, also ACM *Lisp Pointers V*,1 (Jan.-Mar. 1992), 139.
Clark, D.W., and Green, C.C. "An Empirical Study of List Structure in Lisp". *CACM 20*,2 (Feb. 1977), 78-87.
Collins, G.E. "A method for overlapping and erasure of lists". *CACM 3*,12 (Dec. 1960), 655-657.
Francis, R.S. "Containment Defines a Class of Recursive Data Structures". *Sigplan Not. 18*,4 (Apr. 1983), 58-64.
Friedman, D.P., and Wise, D.S. "Aspects of applicative programming for parallel processing". *IEEE Trans. Comput. C-27*,4 (Apr. 1978), 289-296.
Gabriel, R.P. *Performance and Evaluation of Lisp Systems*. MIT Press, Camb., MA 1985.
Girard, J.-Y. "Linear Logic". *Theoretical Computer Sci. 50* (1987),1-102.
Gray, J.N. "Notes on Database Operating Systems", in Bayer, R., *et al*, *eds*. *Operating Systems*. Springer 1978.
Harms, D.E., and Weide, B.W. "Copying and Swapping: Influences on the Design of Reusable Software Components". *IEEE Trans. SW Eng. 17*,5 (May 1991),424-435.
Hesselink, W.H. "Axioms and Models of Linear Logic". *Formal Aspects of Comput. 2*,2 (Apr-June 1990), 139-166.
Kieburtz, R.B. "Programming without pointer variables". *Proc. Conf. on Data: Abstraction, Definition and Structure, Sigplan Not. 11* (special issue 1976), 95-107.
Lafont, Y. "The Linear Abstract Machine". *Theor. Comp. Sci. 59* (1988), 157-180.
Mairson, H.G. "Deciding ML Typeability is Complete for Deterministic Exponential Time". ACM *POPL 17* (1990).
Martí-Oliet, N., and Meseguer, J. "From Petri nets to linear logic". *Math. Struct. in Comp. Sci. 1*,1 (Mar. 1991).
Shalit, A. *Dylan™: An object-oriented dynamic language*. Apple Computer, Camb., MA, 1992.
Steele, G.L. *Common Lisp, The Language; 2nd Ed.* Digital Press, Bedford, MA, 1990.
Strom, R.E. "Mechanisms for Compile-Time Enforcement of Security". *Proc. ACM POPL 10*, Jan. 1983.
Strom, R.E., and Yemini, S. "Typestate: A Programming Language Concept for Enhancing Software Reliability". *IEEE Trans. SW Engrg. SE-12*,1 (Jan. 1986), 157-171.
Wadler, P. "Is there a use for linear logic?". *Proc. ACM PEPM'91*, New Haven, June 1991, 255-273.
Wakeling, D., and Runciman, C. "Linearity and Laziness". *Proc. Funct. Progr. & Computer Arch.*, LNCS 523, Springer-Verlag, Aug. 1991, 215-240.

## A SHORT TUTORIAL ON "LINEAR" LISP

Linear Lisp is a style of Lisp in which every bound name is referenced exactly once. Thus, each parameter of a function is used just once, as is each name introduced via other binding constructs such as `let`, `let*`, etc. A linear language requires work from the programmer to make explicit any copying or deletion, but he is paid back by better error checking during compilation and better utilization of resources (time, space) at run-time. Unlike Pascal, Ada, C, and other languages providing explicit deletion, however, *a linear language cannot have dangling references*.

The `identity` function is already linear, but `five` must dispose of its argument before returning the value `5`:

```
(defun identity (x) x)

(defun five (x) (kill x) 5)            ; a true Linear Lisp would use "x" instead of "(kill x)"
```

The `kill` function, which returns *no* values, provides an appropriate "boundary condition" for the parameter `x`. The appearance of `kill` in `five` signifies *non-linearity*. (See Appendix for a definition of `kill`).

The `square` function requires *two* occurrences of its argument, and is therefore also non-linear. A second copy can be obtained by use of the `dup` function, which accepts one argument and returns *two* values—i.e., two copies of its argument. (See Appendix for a definition of `dup`). The `square` function follows:

 

```
(defun square (x)
  (let* ((x x-prime (dup x)))         ; Use Dylan-style syntax for multiple values [Shalit92].
    (* x x-prime)))
```

Conditional expressions such as `if`-expressions require a bit of sophistication. Since only one "arm" of the conditional will be executed, we relax the "one-occurrence" linearity condition to allow a reference in both arms.[5] One should immediately see that linearity implies that an occurrence in one arm *if and only if* there is an occurrence in the other arm. (This condition is similar to that for *typestates* [Strom83]).

The boolean expression part of an `if`-expression requires more sophistication. Strict linearity requires that any name used in the boolean part of an `if`-expression be counted as an occurrence. However, many predicates are "shallow", in that they examine only a small (i.e., shallow) portion of their arguments (e.g., `null`, `zerop`), and therefore a modified policy is required. We have not yet found the best syntax to solve this problem, but provisionally use several new `if`-like expressions: `if-atom`, `if-null`, `if-zerop`, etc. These `if`-like expressions require that the boolean part be a simple name, which does not count towards the "occur-once" linearity condition. This modified rule allows for a shallow condition to be tested, and then the name can be reused within the arms of the conditional.[6]

We require a mechanism to *linearly* extract both components of a Lisp cons cell, since a use of `(car x)` precludes the use of `(cdr x)`, and vice versa, due to the requirement for a single occurrence of `x`. We therefore introduce a "destructuring let" operation `dlet*`, which takes a series of binding pairs and a body, and binds the names in the binding pairs before executing the body. Each binding pair consists of a pattern and an expression; the expression is evaluated to a value, and the result is matched to the pattern, which consists of list structure with embedded names. The list structure must match to the value, and the names are then bound to the portions of the list structure as if the pattern had been *unified* with the value. Linearity requires that a name appear only once within a particular pattern. Linearity also requires that each name bound by a `dlet*` binding pair must occur either within an expression in a succeeding binding pair, or within the body of the `dlet*` itself. Using these constructs, we can now program the `append` and factorial (`fact`) functions:

```
(defun lappend (x y)                                            ; append for "linear" lists.
  (if-null x (progn (kill x) y)                                              ; trivial kill
    (dlet* (((carx . cdrx) x))                                ; disassociate top-level cons.
      (cons carx (lappend cdrx y)))))     ; this cons will be optimized to reuse input cell x.

(defun fact (n)
  (if-zerop n (progn (kill n) 1)                                             ; trivial kill.
    (let* ((n n-prime (dup n)))                             ; Dylan-style multiple-value syntax.
      (* n (fact (1- n-prime))))))
```

## APPENDIX

```
(defun kill (x)                    ; Return no values.
  (if-atom x (kill-atom x)
    (dlet* (((carx . cdrx) x))
      (kill carx) (kill cdrx) (values))))

(defun dup (x)                     ; Return 2 values.
  (if-atom x (dup-atom x)
    (dlet* (((carx . cdrx) x))
      (let* ((carx carx-prime (dup carx)) (cdrx cdrx-prime (dup cdrx)))
          (values (cons carx cdrx) (cons carx-prime cdrx-prime))))))       ; reuse input cell.
```

[5] Any use of parallel or *speculative* execution of the arms of the conditional would require strict linearity, however.

[6] Although this rule seems a bit messy, it is equivalent to having the shallow predicate return *two* values: the predicate itself and its unmodified argument. This policy is completely consistent with linear semantics.

 