Hamming Numbers, Lazy Evaluation, and Eager Disposal

C K Yuen
DISCS, NUS
Kent Ridge, Singapore 0511

Abstract:

The generation of Hamming numbers is used as a case study for discussing storage management for transient data structures in language processors.

A well known problem originally posed by Richard Hamming, frequently used as example in textbooks teaching recursive programming [1][2][3][4][5][6][7], requires a program to generate a strictly increasing sequence of integers whose prime factors are 2, 3 and 5, i.e., 1 2 3 4 5 6 8 9 10 12 15 16 .... Below we present a solution written in BaLinda Lisp, a parallel Lisp dialect developed by the author's research group[8][9], though this code version is strictly sequential:

```
(DEFUN Hamming (I)
      (LET ((Out (MINIMUM X Y Z)))
           (SETQ (ARRAY Output I) Out)
           (COND ((= Out X) (SETQ X (* 2 (ARRAY Output OutX)))
                            (SETQ OutX (+ 1 OutX))))
           (COND ((= Out Y) (SETQ Y (* 3 (ARRAY Output OutY)))
                            (SETQ OutY (+ 1 OutY))))
           (COND ((= Out Z) (SETQ Z (* 5 (ARRAY Output OutZ)))
                            (SETQ OutZ (+ 1 OutZ)))))
      (Hamming (+ 1 I)))
```

To start, X, Y, Z, OutX, OutY, OutZ and I are all set to 1 and (Hamming 1) is called. Versions in other languages, iterative or recursive, can be readily found.

Now let us imagine that it is necessary to have a parallel solution because, say, more complex processing than a simple multiplication is needed to generate a new Hamming number. It is possible to make the three COND expressions of the above program execute in parallel simply by adding the prefix FUTURE to the first two. However, that would not be a good solution since a task start/terminate overhead is paid just to obtain one value. Further, most of the time only one new Hamming number is needed, and it makes no sense to generate three new tasks with every iteration.

It would be more desirable if three tasks exist throughout the program's execution, but they operate in a stop-go fashion, with each task being signaled by the main program to compute a new number as needed. This is achieved in the following BaLinda Lisp program:

```
(DEFUN Two (OutX)
      (IN 2)
      (OUT 2 (* 2 (ARRAY Output OutX)))
      (Two (+ 1 OutX)))

(DEFUN Three (OutY)
      (IN 3)
```

```
      (OUT 3 (* 3 (ARRAY Output OutY)))
      (Three (+ 1 OutY)))

(DEFUN Five (OutZ)
      (IN 5)
      (OUT 5 (* 5 (ARRAY Output OutZ)))
      (Five (+ 1 OutZ)))

(DEFUN Hamming (I)
      (COND ((NULL X) (IN 2 ? X))
      (COND ((NULL Y) (IN 3 ? Y))
      (COND ((NULL Z) (IN 5 ? Z))
      (LET ((Out (MINIMUM X Y Z))
           (SETQ (ARRAY Output I) Out)
           (COND ((= Out X) (OUT 2)
                            (SETQ X NIL)))
           (COND ((= Out Y) (OUT 3)
                            (SETQ Y NIL)))
           (COND ((= Out Z) (OUT 5)
                            (SETQ Z NIL))))
      (Hamming (+ 1 I)))

(FUTURE Two 1)
(FUTURE Three 1)
(FUTURE Five 1)
(SETQ X 1)
(SETQ Y 1)
(SETQ Z 1)
(Hamming 1)
```

The above program will, upon detection of the need for a new input number, cause its computation to start, and then wait for it to become available before computing the next output. However, it might be better if each new number is computed in anticipation of its being needed very soon. This can be achieved by a few simple modifications:

```
(DEFUN Two (OutX)
      (OUT 2 (* 2 (ARRAY Output OutX)))
      (IN 2)
      (Two (+ 1 OutX)))

(DEFUN Three (OutY)
      (OUT 3 (* 3 (ARRAY Output OutY)))
      (IN 3)
      (Three (+ 1 OutY)))

(DEFUN Five (OutZ)
      (OUT 5 (* 5 (ARRAY Output OutZ)))
      (IN 5)
      (Five (+ 1 OutZ)))

(DEFUN Hamming (I)
      (LET ((Out (MINIMUM X Y Z))
           (SETQ (ARRAY Output I) Out)
           (COND ((= Out X) (IN 2 ? X)
                            (OUT 2)))
           (COND ((= Out Y) (IN 3 ? Y)
                            (OUT 3)))
```

```
            (COND ((= Out Z) (IN 5 ? Z)
                             (OUT 5))))
      (Hamming (+ 1 I)))

(SETQ (ARRAY Output 1) 1)
(SETQ (ARRAY Output 2) 2)
(FUTURE Two 1)
(FUTURE Three 1)
(FUTURE Five 1)
(SETQ X 1)
(SETQ Y 1)
(SETQ Z 1)
(Hamming 1)
```

Curiously, this program actually turns out to be slightly simpler than the previous one.

Now let us consider an alternative, functional programming approach, which specifies that the program that outputs the Hamming sequence also takes it as the input, multiplying it by 2, 3 and 5 to produce three new sequences, which are then merged to produce the Hamming sequence itself:

```
Seq = MERGE3 (MAP (* 2) Seq)
             (MAP (* 3) Seq)
             (MAP (* 5) Seq)
```

where MAP is just a function that takes an input sequence and applies a function to all its elements to make another sequence.

The very conception of this functional program embodies the idea of lazy evaluation: the input sequences to the MAP and MERGE3 functions only exist partially; new elements are evaluated as they are needed. Thus, MERGE3 tries to obtain the next output element by making a demand on one of the MAP functions, which then makes a demand on Seq, receiving whereupon an earlier element which already exists.

What has perhaps received too little attention is the issue of eager disposal: the three input sequences to MERGE3 need not be retained; elements, once used, may be immediately freed. But in fact, the Hamming sequence itself can be treated in the same way. Elements can be sent for output and their storage released, as soon as all three MAP functions have used them. It would be difficult for a functional program, which describe the relation between input and output data structures, to express such eager disposal, at least not without extensions that provide for additional specification of actions relating to individual parts of structures.

Now let us return to the earlier, non-functional programs. They describe processing at a more detailed level, and while it may be inferred from the code that elements of the array Output are being generated with the use of earlier elements, the relation between the input and output is less easy to see compared with what is in the functional program. On the other hand, the program gives clearer indications for storage management: The elements X, Y and Z are overwritten as soon as they have been chosen as the the values of Out, and with OutX, OutY and OutZ non-decreasing in each recursion, element I of Output will not be needed once all three of them exceed I.

To consider the issue of eager disposal a bit further, let us return to the very first program, and produce another version:

```
(DEFUN Hamming (I X Y Z OutX OutY OutZ)
      (LET ((Out (MINIMUM X Y Z))
            (NewX (= Out X))
            (NewY (= Out Y))
            (NewZ (= Out Z)))
           (SETQ (ARRAY Output I) Out)
           (Hamming (+ 1 I)
                    (COND ((NULL NewX) X)
                          (T (* 2 (ARRAY Output OutX))))
                    (COND ((NULL NewY) Y)
                          (T (* 3 (ARRAY Output OutY))))
                    (COND ((NULL NewZ) Z)
                          (T (* 5 (ARRAY Output OutZ))))
                    (COND ((NULL NewX) OutX)
                          (T (+ 1 OutX)))
                    (COND ((NULL NewY) OutY)
                          (T (+ 1 OutY)))
                    (COND (NULL NewZ)
                          (T (+ 1 OutZ))))))
```

That is, instead of using assignments, we achieve changes between iterations by passing expressions down as function arguments. Though we would have a sequence of different values of I, X, Y, etc., we would still have eager disposal if the language processor has been optimized for tail recursion: When Hamming is recursively called, the previous locations of the seven arguments are all released because there is no further use for them. Note also that the above program is quite "functional" in style: it uses no side effect other than defining a new set of values for Output. Once defined, values are not changed.

However, the eager disposal of Output in the above program still requires code analysis to determine when an element will no longer be needed. Another modification can be made, at some cost to program clarity, such that the disposal occurs in the course of garbage collection:

```
(DEFUN Hamming (I)
      (LET ((Out (MINIMUM X Y Z)))
           (PRINT Out)
           (SETCAR Output Out)
           (SETCDR Output (LIST 0))
           (SETQ Output (CDR Output))
           (COND ((= Out X) (SETQ X (* 2 (CAR OutX))
                            (SETQ OutX (CDR OutX))))
           (COND ((= Out Y) (SETQ Y (* 3 (CAR OutY))
                            (SETQ OutY (CDR OutY))))
           (COND ((= Out Z) (SETQ Z (* 5 (CAR OutZ))
                            (SETQ OutZ (CDR OutZ)))))
      (Hamming (+ 1 I)))

(SETQ Output (LIST 0))
(SETQ OutX Output)
(SETQ OutY Output)
(SETQ OutZ Output)
(SETQ X 1)
(SETQ Y 1)
(SETQ Z 1)
(Hamming 1)
```

The four pointers Output, OutX, OutY and OutZ point into a list whose tail grows with each recursion to accommodate a new Hamming number. The pointers are moved forward each time they get used, and after all four pointers move past an element, its location ceases to be accessible and can be garbage collected. It is not hard to produce a parallel version for the above. There is however not much point in imposing a more functional style on this program. It was found that this program, running on only one processor (Transputer), executes considerably faster than the earlier ones, partly because it does not use the ARRAY data structure, and partly because it uses less space and consequently reduces other overheads.

What is the message we are trying to convey? While not a particularly new one, it is nevertheless worth repeating. Though certain programming languages allow a nice global program specification of a algorithm to be produced, implementation efficiency often requires a more detailed description that helps the language processor to manage its hardware resources better. Eager disposal is just one example of such needs. For some earlier discussions on other efficiency issues, see [10][11]

Acknolwedgement: I am grateful to M D Feng, a graduate student, for carrying out execution experiments on the BaLinda Lisp programs contained in the paper.

References

[1] E W Dijkstra, A Discipline of Programming, Prentice-Hall, 1976, pp. 129-134.

[2] P Henderson, Functional Programming: Application and Implementation, Prentice-Hall, 1980, pp. 235-237.

[3] H Abelson and G J Sussman, Structure and Interpretation of Computer Programs, MIT Press, 1985, p. 271.

[4] D A Turner, "Functional Programs as Executable Specifications", in C A R Hoare and J C Shepherdson, Mathematical Logic and Programming Languages, Prentice-Hall, 1985, pp. 43-45.

[5] B J MacLennan, Functional Programming: Practice and Theory, Addison-Wesley, 1990, pp. 320-322.

[6] R Bird and P Wadler, Introduction to Functional Programming, Prentice-Hall, 1988, pp. 188-189.

[7] C Reade, Elements of Functional Programming, Addison-Wesley, 1989, pp. 286-288.

[8] M D Feng and C K Yuen, A Transputer-Based Implementation Parallel Lisp Implementation, ACM Comp Sc Conf, March 1992, Kansas City, pp. 83-90.

[9] C K Yuen, M D Feng, W F Wong and J J Yee, Lisp: Languages and Architectures, Chapman and Hall, 1992.

[10] C K Yuen, What Model of Programming for Lisp: Sequential, Functional or Mixed? ACM SIGPLAN Notices, vol. 26, No. 10, October 1991, pp. 83-92.

[11] Arvind, R S Nikhil and K K Pingali, I-Structures: Data Structures for Parallel Computing, ACM Trans on Prog Lang Sys, vol. 11, October 1989, pp. 568-632.