January 1985

When chasing your tail saves time

Richard B. Kieburtz
When Chasing Your Tail Saves Time

Richard B. Kieburtz

Technical Report No. CS/E 85-007
When chasing your tail saves time

Richard B. Kieburst
Oregon Graduate Center
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006 USA

Several investigators have observed experimentally that in evaluating functional language programs by graph reduction, the time required for evaluation depends rather strongly on whether or not a cyclic data structures are used to represent recursively-defined expressions. The result is surprising, and the purpose of this note is to explain it.

In functional-language programs, recursive definitions can be given for values of ground types, as well as for function values. This is most often used to express the solution of a system of equations, for instance by writing

\[
\text{letrec } z = H \; x \; \text{in } G \; z
\]

where \( H \) is an expression of type \( \alpha \rightarrow \alpha \); \( z \) has the type \( \alpha \), and \( G \) of type \( \alpha \rightarrow \beta \), produces a final answer from the 'solution' to the recursion equation. This style of programming furnishes an elegant and direct way to express many algorithms.

The recursive definition of \( z \) can be expressed as a graph in either of two ways. A direct representation builds a cyclic expression graph,

\[
\begin{align*}
x & \Rightarrow \Psi \\
& \rightarrow H
\end{align*}
\]

in which \( \Psi \) is the explicit application operator.

A slightly less direct representation is also possible, and can avoid the construction of a cyclic graph. The desired solution to the equation

\[
z = H \; x
\]

is the least fixpoint of the operator \( H \), and can be expressed by

\[
z = Y \; H
\]

---

\(^1\)This point was raised at the 1985 Aspenås (Sweden) Workshop on Implementation of Functional Programming Languages, in discussions with Jon Fairbairn, Will Stoye, and Hamilton Richardson.
where the least fixpoint combinator, Y, is defined by the equation

\[ Y H = H(Y H) \quad (2) \]

A direct implementation of Y can be given by the graph-rewriting

\[ \begin{align*}
\text{Y} & \quad \Rightarrow \quad \text{Y} \\
\text{H} & \quad \text{H}
\end{align*} \]

which constructs a directed, acyclic graph. It corresponds to a single step of unfolding of the recursive definition (1).

In choosing an implementation strategy, it is tempting to evaluate recursive definitions by using the acyclic expansion of \( Y H \). A pragmatic reason favoring such a choice is that management of dynamically allocated heap storage can be simplified if it is known that cyclic graphs cannot occur. However the temptation is seductive; there is a virtue to be lost. That virtue is sharing of computed values.

The fixpoint of a recursive definition is understood to be the limit of a sequence of its approximations,

\[ Y H = \lim_{n \to \infty} H^n(1) \]

The cyclic graph representation ensures that the value of each element of this sequence is available for use in computing the next element. The result of each partial evaluation is shared among all references made to the expression it evaluates. This property is not true of the acyclic expansion of \( Y H \), because in forming the graph representing the right-hand side of (2), the expression \( Y H \) is reconstructed, rather than simply copying a reference to the expression.

As a simple example, consider the following definition for the sequence of natural numbers,

\[ \text{nats} = 0. \text{map succ nats} \]

in which the dot denotes list construction. When represented as a cyclic graph, the first few values are produced (by lazy evaluation, of course) in the following sequence of graph rewrites:

\[ \begin{align*}
0 & \quad \Rightarrow \quad 0 \\
\text{map succ} & \quad \text{map succ} \\
1 & \quad \Rightarrow \quad 1 \\
\text{map succ} & \quad \text{map succ}
\end{align*} \]

At each stage of elaboration, the value of the preceding element of the sequence is used in calculating the next one.
Contrast this now with the following series of steps of the acyclic evaluation:

Note that each value is produced separately from the preceding one, by a series of applications of the successor function. In consequence, we see that the complexity of evaluating an initial subsequence of nats is $O(n)$ by unwinding the cyclic graph, but $O(n^2)$ by unfolding the acyclic graph! (Space complexity may or may not undergo a similar increase; in the diagrams above, multiple copies of the expression $H$ are shown, but the several references to $H$ could obviously share one common copy.)

One might wonder at this point whether the phenomenon just illustrated is an instance of the general case, or merely a contrived example. It is general, although the complexity blowup may be even worse.

The use of a list structure to represent a sequence of values is necessary when the object defined by a fixpoint is of a ground type. (For an object of a function type, it is possible to use programmed control flow, i.e. a while loop, to produce a sequence of values.) If succeeding values never depended upon previous ones then no complexity increase would occur, but that case is trivial, as such values do not require recursive definition. If each value depends upon exactly one preceding value, then its computation requires as much time as would computation of the sequence of all of its predecessors in the presence of value sharing. Time to compute a sequence of length $n$ takes $n$ times longer than with sharing.

If each value depends upon the values of two or more of its predecessors (as do the elements of a Fibonacci sequence), then the complexity blowup is worse -- linear time to exponential time. The inverse of this phenomenon -- a complexity improvement when value sharing is
introduced into a recursive evaluation — has been noted by John Hughes in making an argument for implementing memo functions\(^2\).