Reproducing Programs Implement Lazy Lists

By Manfred von Thun, 21-MAR-05

Abstract: A program is called reproducing if running it creates another program which in turn is reproducing. The created program need not be identical with the program that created it. A program is called self-reproducing if running it creates another program identical to itself. This note describes how reproducing programs can be used to implement infinite or lazy lists, and how they might be used for other tasks. Importantly, these programs must not be self-reproducing.

Contents:
Self-reproducing and reproducing programs
Streams or infinite lists or lazy lists
Conventional lazy lists in Joy
Reproducing programs as lazy lists in Joy
Other reproducing programs
Concluding remarks

Self-reproducing and reproducing programs

In the programming world there has been a veritable industry for writing what are called self-reproducing programs. These are program which, when run, create a copy of themselves. Following terminology introduced by Hofstadter (19XX), these programs are often called Quines. A web search for "Quine" will find examples in many languages. Many of the examples are quite complex because the implementation language is not very suitable for the task. Other examples can be surprisingly compact.

In Joy a self-reproducing program is a quoted program [P] such that executing it with the i-combinator produces that very same program:

```
[P] i  ==  [P]
```

The simplest program that satisfies this equation is [[dup cons] dup cons]. The Joy page Recursion Theory has some discussion of this program and its relation to the y-combinator for recursion. Also included are two variant programs which on each execution by i create a program that is different from the one that created it:

```
[[false [not] infra dup rest cons] [not] infra dup rest cons]
[[0 [1 +] infra dup rest cons] [1 +] infra dup rest cons]
```

On successive executions by i the first quoted program produces others in which the initial false oscillates between true, false, true.., and the second quoted program produces others in which the initial 0 is incremented to 1, 2, 3.. for every execution. Programs like these are reproducing but not self-reproducing. Examples in other

languages abound.

The two Joy programs above are mere curiosities. But they suggest variants which promise useful programming techniques. One such technique is for implementing infinite lists or streams. A computer implementation must of course use a finite representation, so their evaluation is demand driven or lazy: expressions are only evaluated when their value is needed. Consequently they are also known as lazy lists. The next section describes a conventional approach to their implementation, and the one following that gives a conventional implementation in Joy. Then there is a section that uses reproducing programs for implementing lazy lists in Joy. The final section outlines some other possible uses of reproducing programs in Joy.

Streams or infinite lists or lazy lists

A list or vector or sequence is essentially a function which takes natural numbers as arguments and returns some value which may be a number or anything else. The function need not be defined for arguments, but if it is defined for some number then it is also defined for all its predecessors. So the function may be defined for all natural numbers and hence is infinite, or it may be defined only for an initial segment and hence be finite. If the function is defined nowhere, then the list is empty.

Finite sequences are normally implemented extensionally: the names of the members are given extensionally and in their order, and they constitute the successive values of the function which the sequence implements. This method is not possibble for infinite sequences, so they have to be implemented intensionally: some program has to be provided that can be used to compute the members. Of course that program has to be finite.

There are many useful operations on sequences, and for an implementationn some of these have to be chosen as primitives. Unfortunately what is efficient in one implementation tends to be inefficient in the other. If the implementation uses arrays of consecutive memory locations, then indexing is particularly efficient. If the implementation uses linked structures, then the first and rest operations are particularly effficient.

For infinite sequences a finite program or specification has to be found, and the choice depends on which access requirements dominate. For efficient random access the implementation must use an ordinary function, defined for all natural numbers. But for efficient sequential access another implementation recommends itself. This will be the case when the function f(n+1) is most readily computed from g(f(n)), where g(x) is particularly efficient to compute. For example, the powers of 2 (1 2 4 8 ..), the n-th can be doubled to give the next power, and the doubling function is easier to compute than exponentiation. Similarly, in the sequence of primes (2 3 5 7 11 ..) the n-th can be used as a starting point in searching for the next prime. An implementation should provide a function to find the first of the sequence and a function to compute the rest of the sequence. So between them the first and rest function characterise the sequence.

The first element of the sequence can be given literally, just as in a finite sequence.

But the rest of the sequence is infinite, and hence has to be given as a program. When that program is called it must compute the rest of the original sequence, and that of course consists of its own first element and a program to compute its ownn rest. The entire sequence thus consists of a pair of two things: a literal first value, and a program to compute the rest.

The pair is most conveniently constructed from the first value and the next function. For the infinite list of all natural numbers these are 0 and the successor function, for the list of powers of two they are 1 and the doubling function. To construct the pair one can use a program. Friedman and Felleisen (1984, p 181) give a recursive Lisp program for doing just this. The program is called str-maker (for stream maker), and the only other primitives needed are str-first and str-rest for computing the first and the rest of a stream. The next section describes an implementation in Joy whose core is modelled on their Lisp programs.

Conventional lazy lists in Joy

This section and the next consists of a run of Joy that has been extensively commented. Note that several standard libraries are loaded. The first part consists of the definitions of the three primitives needed for building a stream, and for accessing its first an its rest.

```
JOY  -  compiled at 14:53:58 on Mar 14 2003 (NOBDW)
Copyright 2001 by Manfred von Thun
usrlib  is loaded
inilib  is loaded
numlib  is loaded
agglib  is loaded
seqlib  is loaded

        DEFINE
            str-maker  ==  #  n [N]  =>  [n [N(n) [N] str-maker]]
                dupd dup [i] dip
                [str-maker] cons cons
                [] cons cons;
            str-first == first;
            str-rest  == second i.
```

Here follow definitions of several streams that will be used later. The first three are streams of numbers, and the fourth is a reminder that the elements of a stream do not have to be numbers. Also included are two of the actual streams that have defined. Observe that in each case the stream consists of two parts: a first value and a quotation for computing the rest of the stream. Note also that the quotation recursively uses the str-maker operator, but the recursive call only occurs when the quotation is executed.

```
        DEFINE
            str-ints == 0 [succ] str-maker;
            str-pows == 1 [2 *] str-maker;
            str-prim == 2 [succ [prime not] [succ] while] str-maker;
            str-brak == [] [[] cons] str-maker.

        str-ints.
[0 [1 [succ] str-maker]]
```

```
        str-pows.
[1 [2 [2 *] str-maker]]
```

Here are a several definitions of operations on streams that will be used later. The names correspond to similar list operations in Joy. Note that the all are defined in terms of just the two primitives str-first and str-rest, and otherwise plain Joy.

```
        DEFINE
            str-uncons ==  [str-first] [str-rest]  cleave;
            str-third ==   str-rest str-rest str-rest str-first;
            str-tenth ==  10 [str-rest] times str-first;
            str-drop == [str-rest] times;
            str-n-th == str-drop str-first;
            str-take ==    #  S n  =>  [S1 S2 .. Sn]
                [null] [pop pop []] [[str-uncons] dip pred] [cons] linrec.
```

Finally, here are examples of these defined operations applied to several of the streams.

```
        str-brak.
[[] [[[]] [[] cons] str-maker]]
        str-brak  str-third.
[[[[]]]]

        str-pows.
[1 [2 [2 *] str-maker]]
        str-pows str-tenth.
1024

        str-ints 5 str-take.
[0 1 2 3 4]

        str-pows 10 str-take.
[1 2 4 8 16 32 64 128 256 512]

        str-prim 10 str-drop 10 str-take.
[31 37 41 43 47 53 59 61 67 71]
```

Reproducing programs as lazy lists in Joy

This section is like the previous one, except that the lazy lists are implemented as reproducing programs. Recall from the first section the following reproducing Joy program:

```
        [[0 [1 +] infra dup rest cons] infra dup rest cons]
```

Since the combination "dup rest cons" occurs frequently, it is convient to introduce an abbreviation "dureco". This is the first definition below. The other three definitions are the three primitives for building a stream and for accessing its first and rest. The prefix "rep-" is a reminder that in this implementation a lazy list is a reproducing program - for the rest operator, to be precise.

```
        DEFINE
            dureco == dup rest cons;
            rep-maker == # n [N] => [[n [N] infra dureco] [N] infra dureco]
                [infra dureco] cons cons  dureco;
```

```
            rep-first == first first;
            rep-rest  == i.
```

Here follow definitions of the same streams as in the previous section. Note that the definions are identical, except for the prefix "rep-" instead of "str-". Two of the constructed streams are shown. Note in particular that, in contrast to the corresponding definitions in the previous section, they do not contain a recursive reference to the operator rep-maker - this is due to the reproducing implementation.

```
        DEFINE
            rep-ints == 0 [succ] rep-maker;
            rep-pows == 1 [2 *] rep-maker;
            rep-prim == 2 [succ [prime not] [succ] while] rep-maker;
            rep-brak == [] [[] cons] rep-maker.

        rep-ints.
[[0 [succ] infra dureco] [succ] infra dureco]

        rep-pows.
[[1 [2 *] infra dureco] [2 *] infra dureco]
```

As before, here are the definitions of some core operators and some example applications.

```
        DEFINE
            rep-uncons ==  [rep-first] [rep-rest]  cleave;
            rep-third ==   rep-rest rep-rest rep-rest rep-first;
            rep-tenth ==  10 [rep-rest] times rep-first;
            rep-drop == [rep-rest] times;
            rep-n-th == rep-drop rep-first;
            rep-take ==    #  S n  =>  [S1 S2 .. Sn]
                [null] [pop pop []] [[rep-uncons] dip pred] [cons] linrec.

        rep-brak.
[[[] [[] cons] infra dureco] [[] cons] infra dureco]
        rep-brak  rep-third.
[[[[]]]]

        rep-pows.
[[1 [2 *] infra dureco] [2 *] infra dureco]
        rep-pows rep-tenth.
1024

        rep-ints 5 rep-take.
[0 1 2 3 4]

        rep-pows 10 rep-take.
[1 2 4 8 16 32 64 128 256 512]

        rep-prim 10 rep-drop 10 rep-take.
[31 37 41 43 47 53 59 61 67 71]
```

Finally, a reminder that the elements of a stream can be anything. So here is a stream whose members are lists of two things: the ordinal number of the member and, in this case, a power of two. Of course such a stream could have been defined in the previous section.

```
        DEFINE
            rep-n-th-pow ==  [0 1] [[succ [2 *] dip] infra] rep-maker.
```

```
        rep-n-th-pow  10 rep-drop 5 rep-take.
[[10 1024] [11 2048] [12 4096] [13 8192] [14 16384]]
```

The output from this and the preceding section was produced by running the input file jp-reprodtst.joy which produced the output file jp-reprodtst.out.

The implementation of lazy lists in this and the preceding section is different from the quite large older Joy library for lazy lists: lazlib.joy ("lazy" infinite and finite lists) with laztst.joy test file and laztst.out output. This library also contains definitions of several useful combinators, and the test file exercises them. There is no reason why the implementation in the preceding two sections could not be extended in a similar way.

Other replicating programs

Apart from the replicating lazy lists of the previous section, there are other uses of replicating programs that deserve exploration.

The replicating lists of the previous two sections were all based on a simple next function which uses the first element of the list to determine the first element of the rest of the list. This does not work well for some lists, for example the list of all (natural) logarithms of the positive integers: 0, 0.693147, 1.09862, 1.38629, 1.60944, ... A program for this list is:

```
    [ [0 1 [pop succ dup log] infra durereco]
           [pop succ dup log] infra durereco ]
```

where durereco has been defined as: dup rest rest cons. Then, as in the previous section, rep-first and rep-rest will find the first element of the list and the rest of the list. Such a list can be constructedd from three parameters, 1, [succ] and [log], by a program similar to rep-maker but somewhat more complex. A different set of parameters will then givve a different list which is also not based on a next function of its members.

There are also variant styles of lazy lists. The first line below is the style of the previous section. In the second line the program consists of two quotations, in the third of a value and above that two quotations. In all three n is a natural number, the first of the list.

```
(1)       [[n  [succ] infra dup rest cons]  [succ] infra dup rest cons]
(2)        [n  [succ] infra dup rest     ] [[succ] infra dup rest     ]
(3)         n [[succ]  dip  dup          ] [[succ]  dip  dup          ]
```

The rest of the list for styles (1) (2 (3) is formed by the i-combinator. The first element of the list in style (1) is found by first first, as in the previous section. In style (2) it is found by pop first, and in style (3) by pop pop. So for finding the rest of the list style (3) is more efficient than style (2), which is more efficient than style (1). But for other manipulations the exact opposite is the case, especially for stack manipulations.

In all the replicating programs so far the replication is, in a sense, internal.

Replication steps do not involve the stacck below the programs. But this need not be so.

First, there are replicating programs which on every execution deposit something on the stack below the replicating program. One way to do that is to use the lists of the previous section. Assume such a list is on top of the stack. To leave its first element and above that its rest on the stack, do this: rep-uncons. Alternatively, to leave the first of the rest and above that the rest, do this: rep-rest dup rep-first swap. But in neither of these does the replication and depositing occur just by the i-combinator. To achieve this, the replicating program must do its own depositing, and not rely on anything external. Here is an example, it similates radioactivity or diffusion. The internal quantity, 1.0 at the start, is halved at every step and the other half "leaks out" and is deposited on the stack below the replicating program.

```
[ [1.0 [2 / dup] infra uncons dureco]
       [2 / dup] infra uncons dureco ]
```

Again, quotations of this kind can be constructed by a variant of rep-maker; the quotation above would be constructed from the parameters 1.0 and [2 / dup].

Second, whereas the program above left something on the stack after each replication, it is also possible to do the reverse: to remove something from the stack from below the program and incorporate what has been removed into the program. Here are two examples. The first is a numeric accumulator, it starts with an internal state 0 which has a number from below the stack added to it by every reproduction. The second is also an accumulator, but it starts with an empty list which has an element from below the quotation cons'ed into it by every reproduction.

```
[ [0  cons [+]     infra dureco] cons [+]     infra dureco ]
[ [[] cons [swons] infra dureco] cons [swons] infra dureco ]
```

Another variant of rep-maker could construct these, from parameter 0 and [+], or [] and [swond], respectively.

Third, there are reproducing programs that change the stack below and keep an internal count of how many times they have been called. To start, here is a program which will square what it finds below itself on the stack, but without such a count:

```
[ [[dup *] dip dup cons] [dup *] dip dup cons ]
```

And the following reproducing program does the same, except that it keeps an internal call-count of the number of times that it has performed a squaring and a reproduction:

```
[ [0 [succ] infra [dup *] dip dureco]
     [succ] infra [dup *] dip dureco ]
```

Again, a variant of rep-maker could construct such programs, in this case from [dup *].

Fourthly, here is a reproducing program which picks up quotations from below on the

stack and executes them on its own internal stack which starts off as []:

```
[ [[] uncons [swap infra] dip cons dureco]
      uncons [swap infra] dip cons dureco ]
```

If we push two qutations, [dup *] and then [2 3 +], then this program and finally do i i, then the internal stack will be [25].

Concluding remarks

Abelson and Sussman (1985 "SICP", pp 242-292), in their wonderful early book on Scheme, have a large section on streams, much of it general and independent of any implementation. But they also give a particular implementation which uses two functions, delay and force. The delay funnction creates a thunk, a parameterless lambda expression, which can be evaluated by the force function. (Folklore has it that the name "thunk" derives from the noise when such a beast is dropped onto the runtime stack.) In their simplest form these correspond very roughly to what in Joy would be creating a quotation and executing it by i. In another version force is implemented as "call by need", which is a simple example of "memoising optimisation". Here force evaluates a delayed (unevaluated) expression the first time it is needed, and simply returns that value on later occasions. Eager (non-lazy) languages such as Scheme use call by value as the default, and to implement a call by need version of force requires assignment. This uses a variable "already evaluated" which becomes part of the delayed expression. Lazy languages such as Turner's KRC and Miranda, and their modern descendant Haskell use graph rewriting as the implementation, and consequently call by need is the default.

An implementation of call by need in Joy would require a new force combinator which is like the i-combinator except that it also modifies the quotation to contains just the single value which the execution of the quotation left on top of the stack. Normally, but not necessarily, the quotation would be the Joy equivalent of a thunk: its executionn does not access the stack below. An interaction might look like this:

```
DEFINE  five == [2 3 +].

five.                   =>  [2 3 +]
five i.                 =>  5
five.                   =>  [2 3 +]   (* as before *)
five force.             =>  5         (* also changes quotation *)
five.                   =>  [5]       (* note change *)
five i.                 =>  5         (* of course *)
five force              =>  5         (* cannot simplify further *)
five.                   =>  [5]       (* first change is permanent *)
```

[Note to myself: The force funnction must change the first element of the quotation, not the list-node for the quotation. Because in the current implementation of Joy definitions are not gc'ed, this will not work if (1) the quotation is in a definition AND (2) the result value (5 above) is a list or other quotation. So the implementation of force must check that the result value is a number, char, string, set or symbol. In a future implementation of Joy the symbol table management for definitions needs to be

changed so that definitions can be gc'ed - at a cost of course. This would also be necessary if runtime changes (assignments?) are being contemplated.]

References:

H. Abelson and G.J. Sussman with J. Sussman, 1985, *Structure and Interpretation of Computer Programs*, MIT Press, Chicago, Mass.

D.P. Friedman and M. Felleisen, 1984, *The Little Lisper*, 2nd edn, Science Research Associates, Chicago.

Back to Homepage for Manfred von Thun

- Accessibility
- Privacy
- Copyright and disclaimer