Compiling functional pipe/stream abstractions into conventional programs: Software Pipelines

Richard Fateman
Computer Science Division, EECS
University of California, Berkeley

June 1, 2000

Abstract

Representing a potentially infinite indexed collection of data is a handy abstraction in a number of programming situations, even though it is ignored in scientific programming, or if noticed at all is derided as a silly functional programming trick. We show how the stream/pipe abstraction can sometimes be converted into efficient C, Java or Lisp programs.

1 Introduction

Representing a potentially infinite indexed collection of data is a handy abstraction in a number of programming situations.

A cute language facility familiar to many students is streams (in the language of Abelson/Sussman [1]) or pipes (in the language of Norvig [4]). The basic idea is to represent a collection by a pair: the first element, plus a "promise" or a suspended function. This promise can be executed to produce a pair: the second element, plus a "promise" that can compute a pair, etc.

This notion is usually discarded in scientific computing because it is embedded in a functional programming framework that has not been popular in scientific computation. We are not going to argue this case here, but will point out that the result of program-transformation programs here can be C or Fortran or Java code, and thus the results need not be expressed in functional languages.

2 Examples

Using Common Lisp syntax, and the programs in the Appendix, we can compute a pipe of the natural numbers in various ways.

```
(defun integers-from (n)(make-pipe n (integers-from (+ 1 n))))
(setf natnums (integers-from 0))
```

Here is an alternative

```
(setf natnums (make-pipe 0 (map-pipe #'1+ natnums)))
```

The value of such an expression is printed as

```
(0 . # <Interpreted Function (unnamed) @ #x2050ddb2>)
```

which is just Lisp's way of saying it begins with 0 but is otherwise too complicated to print in detail. But after "forcing" a few terms, it becomes

```
(0 1 2 3 4 . #<Closure (:INTERNAL MAP-PIPE 0) @ #x2050dfea>).
```

Later it might be

```
(0 1 2 3 4 5 6 7 8 9 . #<Closure (:INTERNAL MAP-PIPE 0) @ #x2050e2f2>)
```

etc.

A pipe of Fibonacci numbers can be constructed by

```
(setf fibs  ;; usual fibonacci stream
    (make-pipe 1 (make-pipe 1
                            (map-pipe #'+ fibs (tail-pipe fibs)))))
```

and a pipe of factorials is

```
(setf facts
      (make-pipe 1 (map-pipe #'(lambda(r s)(* r s))
                             (integers-from 2)
                             facts)))
```

A neat use of this idea is the representation and computation of formal power series. This system is motivated by the notion that many real functions of a real variable can be represented in a region by a sum such as $\exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{2!} + \cdots$. Such a sum, when truncated at a given point and evaluated for small enough x is a good approximation for the exponential function. Generalizing this and treating power series as a formal algebraic system is a common theme in computer algebra systems. The notion of error in the approximation is soft-pedalled in the process; as is traditional, we will ignore error terms too.

A series like $a + bx + cx^2 + \cdots$ is encoded as a pipe beginning `(a b c ....)`. Here are some simple programs which assume that we have agreed upon a particular variable (say x) for all our series.

```
;Adding power series by recursively traversing two pipes

(defun ps-add (x y)
    (cond ((empty-pipe x) y)
          ((empty-pipe y) x)
          (t(make-pipe (+ (head-pipe x)(head-pipe y))
                       (pss-add (tail-pipe x)(tail-pipe y))))))

;;Multiplying them

(defun ps-mult (s1 s2)
  (make-pipe (* (head-pipe s1)(head-pipe s2))
             (ps-add
              (scale-pipe (tail-pipe s1)(head-pipe s2))
              (ps-mult s1 (tail-pipe s2)))))

(defun scale-pipe
  ;; multiply each element in f by the scalar s
  (s f)(map-pipe #'(lambda(x)(* x f)) s))
```

For example, `(ps-mult '(1 2 4 5) '(1 2 4 5))` produces a stream beginning with `(1 4 12 26)`.

Power series are not the only seriously interesting application for pipes; see for example Vuillemin's work on constructive real numbers via continued fraction computation [6].

Later in this paper we mention another potential application where pipes can be used for long numbers (bigfloats); in this case the items in the stream are successively less significant digits. [5, 2].

3 Good news / Bad news

This notion of running a computation with objects of this kind is unpopular, perhaps because it does not appear in common texts, or ignored because it seem to be an esoteric abstraction. Sometimes it is deemed unsuitable for implementation reasons:

1. Popular efficient programming languages (C, C++, Fortran) don't support full-fledged function objects which are usually used for implementing pipes. Therefore the pipe abstraction is discarded in the belief that it cannot be used in efficient programs.

2. Since today's microprocessors are increasingly built around hardware pipelines of arithmetic units, dynamic data structures and procedure calls are avoided.

3. (Specifically for the bigfloat idea) Rather than burping out more digits (or bigits) on demand, a more plausible approach which might take advantage of typical parallel/pipeline architectures would be to try to compute in machine precison whether additional precision is required for the solution of the problem at hand. If precision P (greater than machine precision) is required, then just one more pass in P-precision may do the job. This second pass (if it is in fact necessary) is usually so much more expensive than the first pass that the cost of the first pass is negligible.

Rejoinder:

The continuation- or stream-based computation is a mechanism for representing a computation much as a source-code listing represents a program.

Rather than treating it as an execution recipe that must be followed slavishly, interpretively, including the manipulation of one or more thunks, realize that it is an abstraction that can be compiled into sequences of operations in the same way that any other program can be compiled. Indeed, we can use this formalism to explore software pipelines by analogy with the hardware pipeline concept.

Ideally the result of specifying a computation in a suitable abstract language can improve efficiency by allowing the programmer to compose algorithms at a high level, choosing the appropriate methods. This can then be followed by a translation to low-level code appropriate for the execution environment, and with suitable low-level optimizations[1].

There are at least two ways of approaching this definition/compilation situation.

The traditional approach is to take an utterance and compile it in through the stages of lexical, syntactic, and semantic analysis, eventually preparing it for execution via assembly, linking and loading binary code).

A less traditional approach is to look at the code and run it through an analyzer which will attempt to partially execute it symbolically. For example, consider a program like this:

```
(defun h (x)
  (let ((ans nil))
    (setf ans  (make-pipe x (map-pipe #'(lambda(r s)(* r s))
```

[1]It is a curious failure of software technology that high-level languages which presumably allow the programmer to specify *exactly* what should be computed seem to be compiled to less efficient code. It is also curious that programmers who rail against some high-level languages as being "too inefficient" will then ignore other far more significant speedup factors available to the assembly language programmer. The trade-off on efficiency may just be an excuse to use some familiar language which by coincidence is "just efficient enough".

```
                                    (integers-from (+ x 1))
                                    ans)))

    ans))
```

The casual reader may not immediately see what is being computed here, but typing `(h 1)` results in a pipe whose contents print as `(1 2 6 24 ...)` and indeed it computes factorials.

Let's slightly modify our basic function. `hs` is like `h`, except `*s` and `+s` and `integers-from-s` are used. These `-s` functions work as simplifiers on prefix algebraic expressions when they are given symbolic rather than numeric input. For example, `(+s '(+ x y) 'y)` is `(+ x (* 2 y))`. For numeric data, they work the same as +, *, etc.

```
(defun hs (x) ;; x, x*(x+1), x*(x+1)*(x+2), ...
  (let ((ans nil))
    (setf ans  (make-pipe x (map-pipe #'(lambda(r s)(*s r s))
                                      (integers-from-s (+s x 1))
                                      ans)))
    ans))
```

Now `(hs 'z)` returns `(z .  #<closure (:internal h 0) @ #x2055c9b2>)` which is really an abbreviation for the pipe beginning `z, z*(z+1), z*(z+1)*(z+2), ...`

The function `piece-pipe` [or `pp`] takes a pipe `p` and an integer `n` and makes an ordinary list of the first `n` elements: `(piece-pipe (hs 'z) 3)` is `(z (* z (+ 1 z)) (* z (+ 1 z) (+ 2 z)))` and `(pp (hs 1) 3)` is `(1 2 6)`.

`(analyze-pipe (hs 'z) n)` produces an expression which can be considered the body of a function which references the value of `z` as a global object, and computes the list as a straight-line program. For `n=3` we get this:

```
(let* ((w0 (+ 1 z))
       (w1 (* z w0))
       (w2 (+ 2 z))
       (w3 (* w1 w2)))
  (list z w1 w3))
```

or automatically translated into a C-like program segment by the program `p2i` (for prefix-to-infix):

```
{
        double w0=1+z;
        double w1=z*w0;
        double w2=2+z;
        double w3=w1*w2;
        list(z,w1,w3)
}
```

On the face of it, given a global value for `z`, or attaching a header by which `z` is bound to some parameter, this computes the right pieces for each element in the list. In the lisp version we can set it up so that efforts to find values in that returned list beyond those that are precomputed invoke the routine for the defining pipe.

Notice that we are generating names as necessary for things we have computed and will need again.

There are several problems with this solution technique.

We are ignoring for the moment the prospect that names such as `w1` can be "captured" by this process: we assume that no human programmer will be using the names `w0, w1,` as elements of the stream. As we shall see, the computer can choose names that only it has access to.

For the fans of Algol-like languages, we have printed out the type "double" but perhaps this is not the floating-point type of interest. An editing of the transformation program in the appendix can change the details, or one could take the result source code and run an edit script over that.

What then is the objective of this demonstration up to this point?

> If we express our computation abstractly as a pipe, we can easily convert the computation of a prefix of it (automatically) to a conventional and probably quite efficient program, using pipes either not at all (C-code) or using pipes "when we run out of precomputed data" (Lisp-code). The restriction on the C-like program above assumes that we need only (a fixed in advance) maximum number of elements from the pipe.

4 Problems

4.1 Name conflicts

Recall that we are using certain names for variables, such as w0, w1, ... and we are not in a position to forbid their use by the programmer. If a programmer does use such a name there is a potential for conflict. One quick fix is to use a "real" Common Lisp program `gensym` which generates a unique name distinct from any the programmer has previously mentioned (and by keeping them separate) different from any variable the programmer can mention in the future. The names chosen by the computer tend to be longer and less euphonious. When displayed they look something like `#:w83`. How do we know when to pick a name? It is really quite simple: Each time we compute the head of a pipe and it turns out to be more complicated than a symbol or a constant, we replace it by a name produced by `gensym`, while keeping track of all such symbols and the values they represent, on a list. If we then create a program segment to assign, in order, the values for these variables, then we are set.

4.2 More tricks and complications

It may seem that we are losing some efficiency in these programs because every time we return a value it is a compound value (a list in Lisp). Clearly we could also put together some other data structure such as an array or vector.

Lisp provides an alternative, that of a multiple-value return. The idea behind this is efficiency: instead of taking n values and making a list of them in order to return them as a unit, Lisp allow one to return all n values (presumably in registers or on a stack) without using permanent storage. Instead of writing `(list a b c)`, we write `(values a b c)`.

Sometimes we know that, while up to n terms may be needed, we do not need them all at some early stage. In fact it is the likelihood that we will only rarely need all n elements that makes pipe-analysis especially tempting. and therefore we do not need to test at runtime. The code generated can then be simpler, in some sense, although we are faced with the prospect of continuing the computation after interrupting it. Some languages are happy providing such a facility, but C-style languages are not. Here we suggest keeping all intermediate data as globals, using up as many gensym names as needed. Returning to our previous example,

```
(lambda(i)
  (block nil
    (setf w0 (+ 1 z))
    (if (= i 1) (return w0))
    (setf w1 (* z w0))
    (if (= i 2) (return w1)) ;both w0 and w1 will be set, etc.
    (setf w2 (+ 2 z))
```

```
    (if (= i 3) (return w2))
    (setf w3 (* w1 w2))
    (if (= i 4) (return w3))
    (if (> i 4) (error))))
```

Naturally if we know that `w0` and `w1` have been set, we can generate code that looks like:

```
(lambda(i)
  (block nil
    (if (<= i 2) (return w1)) ;both w0 and w1 will be set, etc.
    (setf w2 (+ 2 z))
    (if (= i 3) (return w2))
    (setf w3 (* w1 w2))
    (if (= i 4) (return w3))
    (if (> i 4) (error))))
```

and in fact we can generate n different program segments, where each knows that the previous segments have all been run. In this case segment `p2` might be simply:

```
    (setf w2 (+ 2 z))
```

If we use the "real" `gensym` Common Lisp program, the variable names would be guaranteed to not repeat. Again, any of this could be printed out as a section of C-like code.

5 Generalization to conditional code

This kind of code seems fairly easy to control when we have a mapping from one or pipes with elements $\{u_i\}$ $\{v_i\}$ to the elements $\{s_i = f(i, \{u_j\}_{j=0}^{k}, \{v_j\}_{j=0}^{k}, \{s_j\}_{j=0}^{i-1})\}$. and the functions are relatively simple arithmetic. What if we are attempting to do something in which the elements $\{s_i\}$ require conditional computation? A standard kind of operation is that of merging two or more ordered pipes. That is given inputs such that $u_j \leq u_k$ if $j \leq k$, produce a pipe such that all the elements of all input streams are contained in the sorted result.

The standard way of producing such a stream in Lisp might be rendered about like this:

```
(defun merge-pipe (p1 p2 compare) ;;; a sorted pipe
  (cond ((empty-pipe p1) p2)
        ((empty-pipe p2) p1)
        (t (let ((h1 (head-pipe p1)) (h2 (head-pipe p2)))
             (if (funcall compare h1 h2)
                 (make-pipe h1 (merge-pipe (tail-pipe p1) p2 compare))
                 (make-pipe h2 (merge-pipe (tail-pipe p2) p1 compare)))))))
```

Why does this presents a barrier to our analysis program? It is that in order to analyze the pipe elements and set up a pre-computation schedule we must symbolically execute the `compare` operation on the inputs. We do not know what this function might be or how to simplify its application on its inputs.

Fortunately we do not have to give up quite yet. We consider as a prototype that the comparison function is `min`. As should be clear, we could generalize to any other function providing a complete ordering. In the case of `min` for programming purposes we must also have a maximal element serving as ∞. The important property is that `min(x,`∞`)` is `x`.

Our design works this way: Given 2 ordered pipes[2], U and V, the first result element of W is $w_0 = \min(u_0, v_0)$. We know that the next element, w_1 can be computed as the minimum of 4 elements: two are the next elements in the U and V stream, namely u_1 and v_1. The next two are the first elements of two "singleton" streams of one element each: {if($w_0 = u_0$) then v_0 else ∞}, and {if($w_0 = v_0$) then u_0 else ∞}. This scheme generalizes so that starting with 2 streams the nth element combines $2n$ streams. For example, the next element w_2 is the minimum of 6 elements.

In reality the complexity can be reduced by applying some simplifications. This can always be done as shown here. We use Lisp notation where `(if a b c)` means "if a then b else c".

If we repeatedly use these rules:
`(min x (if a b c))` → `(if a (min x b)(min x c))`,
`(min x inf)` → `x`.
we can prove that these expressions are equivalent:

```
1. (min u1 (if (= w0 v0) u0 inf))
2. (if (= w0 v0) (min u1 u0) (min u1 inf))
3. (if (= w0 v0) u0 u1)
```

Similarly, these two are equivalent:

```
4. (min v1 (if (= w0 u0) v0 inf))
5. (if (= w0 u0) v0 v1)
```

Tthe second term w_1 in the result sequence can be reduced in steps to

```
 (if (= w0 v0)  (min v1 u0)  (min v0 u1))
```

This does require that we insist the inputs are all distinct so that if `(= w0 v0)` is true, then `(= w0 u0)` is false,

This is now fairly clear. If the first element is `v0`, the second is the minimum of `u0` and `v1`. If the first element is `u0`, the second is the minimum of `v0` and `u1`.

The result of sufficient `min` and `if` simplification should allow one to merge `(a 1 4 b)` and `(2 r s 3)` to `(a 1 2 r s 3 4 b)` without knowing what `a, b, r` or `s` are, exactly.

`Merge-pipe-min` is a modestly complex program (25 lines), but most of the fiendishness comes in the form of simplification programs to assist in the partial execution of `min`. The version we have written allow us to produce this from merging two pipes of `(1.2 a 2.5)` and `(1 2 3)` to get the following pretty-good program.

```
(let* ((w0 (min a 1.2))
       (w1 (if (equal w0 a) inf a))
       (w2 (if (equal w0 2) inf 2))
       (w3 (if (equal w0 1.2) inf 1.2))
       (w4 (min 2.5 w1 w2 w3))
       (w5 (if (equal w4 2.5) inf 2.5))
       (w6 (if (equal w4 3) inf 3))
       (w7 (if (equal w4 w1) inf w1))
       (w8 (if (equal w4 w2) inf w2))
       (w9 (if (equal w4 w3) inf w3))
       (w10 (min w5 w6 w7 w8 w9 inf)))
  (append (list 1 w0 w4 w10)
          #<interpreted closure (:internal sameop) @ #x20700e22>))
```

[2]for simplicity we require that they have all distinct elements. This restriction can be lifted, although with some effort.

We are not necessarily endorsing this tactic which amounts to laying out a special-case sort/merge program. While it can make sense where some, but not complete, knowledge of the input is available, in the absence of any information, it will be difficult to beat a clever run-time sort on any but the smallest length inputs.

6 Composition

Composition of power series such as $U = \{u0, u1, u2, ...\}$ and $V = \{0, v1, v2, ...\}$ is a cute illustration of software pipe code-generation from a stream specification. (Note [3] this make sense if v0 = 0). This computes $U(V(x))$ as a power series. In the program below, seven terms are available explicitly, although more can be found by running the associated program. (These additional terms would not be available if we converted this to C).

```
(let* ((w0 (* u1 v1))       (w1 (* v1 u2))       (w2 (* v1 w1))
       (w3 (* u1 v2))       (w4 (+ w2 w3))       (w5 (* v1 u3))
       (w6 (* v1 w5))       (w7 (* u2 v2))       (w8 (+ w6 w7))
       (w9 (* w1 v2))       (w10 (* u1 v3))      (w11 (* v1 w8))
```

21 very similar lines omitted

```
       (w78 (+ w75 w76))    (w79 (+ w77 w78))    (w80 (* w1 v5))
       (w81 (* u1 v6))      (w82 (* v4 w8))      (w83 (+ w80 w81))
       (w84 (* w22 v3))     (w85 (+ w82 w83))    (w86 (* v2 w45))
       (w87 (+ w84 w85))    (w88 (* v1 w79))     (w89 (+ w86 w87))
       (w90 (+ w88 w89)))
 (append (list u0 w0 w4 w13 w29 w54 w90)
         #<closure (:internal pss-add 0) @ #x205ac29a>))
```

There is also an amusing consequence of writing this out as a straight-line program: we can tell that there are 4 additional operations between computing w0 and w4, then 9 between w4 and w13, etc. The count of operations per additional term is obvious as the count of temporary variables is given, and computing differences we find it is a cubic algorithm.

7 Other operations, and Horner's Rule

There are a variety of other algorithms, including power series division and reversion that can be implemented as streams. The basic ideas are described by Knuth [3]

Evaluation of a truncation of a power series at a scalar point is rather straightforward.

```
(defun ps-horner-pipe (p s n)
  ;; evaluate pipe p at scalar s to n terms. Return a scalar.
  ;; assumes s is ``small''
  (cond ((or (= n 0)(empty-pipe p))0)
        (t (+ (head-pipe p)
              (* s (ps-horner-pipe (tail-pipe p) s (1- n)))))))
```

;; A symbolic version of the above

```
(defun ps-horner-pipe-s (p s n)
  ;; (ps-horner-pipe-s '(a b c) 'x 3) -> (+ a (* x (+ b (* c x))))
  (cond ((or (= n 0)(empty-pipe p))0)
(t (+s (head-pipe p)
               (*s s (ps-horner-pipe-s (tail-pipe p) s (1- n)))))))
```

Another view of pipe organization which is probably more appropriate if mostly polynomial "exact" operations are being done, is to arrange the pipe so that the highest degree is first; a modern view of computer algebra suggests that most polynomials are sparse and hence we should allow for encoding coefficient/exponent pairs. That is, $r * x^5 + s * x^2 + t$ should be

```
((r . 15)(s . 2) (t . 0))
```

and evaluation looks like this

```
(defun poly-eval-list-s (p s) ;no pipes at all
  ;;example: (poly-eval-list-s '((a  43) (b  3) (c  0)) 'x)
  ;; ==> (+ c (* a (expt x 43)) (* b (expt x 3)))

  (cond ((null p) 0)
(t
 (let* ((h (car p))
(coef (car h))
(expon (cadr h)))
   (+s (*s coef `(expt x ,expon))
       (poly-eval-list-s  (cdr p) s ))))))
```

This kind of evaluation is especially interesting if we know that the terms are sorted, and the usual evaluation is at a point that is large compared the coefficients. In particular we could think of the polynomial above, evaluated at `r=3, s=2, t=9, x=10` as the number `3000000000000000 + 200 + 9 = 3.000000000000209d+15` This brings us to our next use of pipes.

8 Numbers

Consider the modeling of a high precision floating-point number `h` by an ordered sequences of modest precision-floating numbers. For example, using 6 decimal-digit numbers, represent `h=123,456,789,012.345678` by (123456*10^6 ,789012, 345678*10^{-6}).

These sequences can be sparse in the sense that h=123,456,000,000.345678 could be represented by (123456*10^6, 345678*10^{-6})

Suitably constrained so that we do not lose information to overflow, we can do arithmetic on such numbers without loss of precision [5, 3]. As shown by Shewchuk [5], it is possible to add sequences subject to occasional normalization, and it is possible to multiply them fairly rapidly, since with IEEE standard arithmetic one can effectively split the bigdigits to into high/low components and losslessly multiply these half-bigdigits.

How much work is it to do addition of sequences like this? If we know that $|a| \geq |b|$, then the following algorithm will produce a nonoverlapping expansion $(x,\ y)$ such that $a+b = x+y$ where x is an approximation to $a + b$ and y represents the roundoff error in the calculation of x.

```
(setf x (+ a b))  ;ordinary double-float approximate add
(setf v (- x a))  ;ordinary double-float subtract
(setf y (- b v))  ;the left-over part.
```

The sequences representing two bigfloats to be added are merged arranged so that the largest numbers in absolute value come first. If the first two bigdigits are a and b in the combined sequence, they are added as above; the left-over part y is merged into the next term.

Suppose that we have merged two bignums to produce the following sequence of 4-decimal bigdigits. (2000.0 +3.101 +0.0003) The first step gives us (2003.0 +0.101 +0.0003) and the next step gives us (2003.0 +0.1013 +0.0000)

Adding two numbers of lengths n and m will result in a merged sequence of length at most $n+m$. Some combinations may occur and there will in general be gaps in the sequence as well as cancellations. Some of the ideas for code generation used in this paper are applicable to this situation as well. For extensive details, see the references [5, 2].

9 Conclusions

This paper illustrates how symbolic computer execution can bridge the gap between clever (but apparently "inefficient") abstraction and brute-force (but apparently "efficient") code. By partially executing some of the high-level programs, software source code can be generated that is specialized into sequences of arithmetic and assignment statements. Thus the code looks very much like the optimal assembly-language code necessary for a computation. The level of execution possible in this source-to-source transformation Although this would seem to be merely the work of a compiler, the partial execution and compile-time simplification paradigm far exceeds the kind of high-level transformations to be expected from even the most optimizing compiler. For best performance, the resulting source code would be optimized in the context of a particular machine model, taking into account the number of registers, the available parallelism, cache performance, and other considerations.

10 Appendix

`www.cs.berkeley.edu/~fateman/papers/pipe.cl`

11 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley.

References

[1] Abelson, H., Sussman, G.J. and Sussman, J. *Structure and Interpretation of Computer Programs.* McGraw Hill, 1996.

[2] Fateman, Richard. Code generation for sparse floats. (in preparation).

[3] Knuth, Donald E. *The Art of Computer Programming: Seminumerical Algorithms,* vol. 2, 2nd ed. Addison-Wesley, 1981.

[4] Norvig, Peter. *Paradigms of Artificial Intelligence Programming.* Morgan Kaufman, 1992.

[5] Shewchuk, J.R. "Robust adaptive floating-point geometric predicates." *Proceedings of the Twelfth Annual Symposium on Computational Geometry, FCRC '96*, New York, NY, USA: ACM, 1996 141–150.

[6] Vuillemin, J. "Exact real computer arithmetic with continued fractions." *Proceedings of the 1988 ACM Conference on LISP and Functional Programming,* New York, NY, USA: ACM, 1988. p. 14–27. also *IEEE Trans. on Computers.* vol 38 no 8 Aug 1990 p. 1087–1105.