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

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

January 12, 2003

Abstract

Representing a potentially infinite indexed collection of data is a handy abstraction in a number of programming situations. Computational scientists routinely try to finess this notion by allocating an array of some "large enough" size, and then dynamically re-allocate if it is not large enough. A technique like this is rarely used, generally because programmers are unfamiliar with the technique; secondarily, debugging a program with such a possible "asynchronous" process in the typical poorly-equipped programming language (C, C++, Fortran, and Java come to mind) requires some delicacy.

A programmer both familiar with these technques and aware of the implementation of infinite stream/pipe abstractions in Lisp, Scheme, or functional languages, might still dismiss it as one of those tricks advocated primarily by language theorists.

In fact, the manipulation programs we discuss *are most conveniently expressed in a functional representation* and are given in ANSI Common Lisp, but we show how to use them to *generate programs in C*. Thus the stream/pipe abstraction may be used, in principle, even by those who allege that all practical programs must be written in some "efficiently compiled language" like C (or with minor changes, Fortran, etc.).

We illustrate some program applications at the heart of the intersection of symbolic and numeric computing, including power series, sparse high-precision floating-point arithmetic, solution of ODEs, and simulations.

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 computer science students is *streams*[1] or the term we will adopt: *pipes*[2]. The use of the term "stream" used in Scheme is pre-empted in many programming languages, including Common Lisp, by built-in objects representing a source or sink of characters or bytes, typically associated with file input or output. The basic idea of a pipe is to represent a sequenced collection by a pair: the first element in the collection, plus a "promise" or a suspended function. This promise can be executed to produce another pair: the second element, plus a promise that can compute a pair consisting of the third element and a promise, etc. If this is the first time the reader has encountered this idea, you are probably doubtful of its use. We hope to change your mind.

[1] (in the language of Abelson/Sussman [1])
[2] suggested in Norvig [5]

Even programmers who have encountered pipes[3] do not usually view them as one of their first lines of abstractions to point to a neat solution in scientific computing contexts, where other languages and kinds of abstractions dominate.

The problem is perhaps deeper though: many scientific application programmers *have never encountered the pipe idea in their education.* This is probably intertwined with the fact that they are typically taught programming at a simpler level, using a language where it is rather difficult to even think of the notion of a "promise." It is not our objective in this paper to argue that functional programming should replace the more common imperative style for scientific computation. Here we will simply point out that the *result* of pipe-based *program-generation programs* can, in any case, be imperative style C or Fortran or Java code. Our argument is that there may be advantages for *thinking in the pipe framework*: this can be used even if the ultimate requirement for the program is that it be written in an imperative language. Since the choice of programming idioms is in large part of matter of taste, this paper is intended to offer a novel taste.

We note up front that the programming language community has studied the transformation of sequences to loops but mostly in a more abstract setting. The interested reader may start with SICP [1] and continue to Richard Waters' accessible work on "Series"[10] which has over 50 references. Waters' project was considered for inclusion in the Common Lisp standard, but was instead made available separately in a large Common Lisp package. This provides for the conversion of series Lisp expressions to conventional loop expressions (also in Lisp). Waters has also codified the requirements for a series expression to be transformed fully to a loop, basically requiring that, to be converted, the output of a series computation must be "on-line" — depending only on a fixed length prefix of the input series, which must also be on-line. We do not go beyond this work in converting from series to loops, but suggest instead that the transition from series (in Lisp) to expansion in place for a fixed number of terms (and perhaps expressed in C) may be of some practical interest.

In particular, while Waters' code is quite powerful, in our particular examples some of Waters' formal restrictions can be lifted since we do not in general require arbitrary-length computations, nor are we necessarily producing Lisp code.

2 Introductory examples

We will be using ANSI Common Lisp for the presentation of the programs; C or Fortran fans: please bear with us. As previously stated, the resulting generated programs can be in C.[4] All the programs needed are contained in the Appendix.

As our first example we show how to compute a pipe of the natural numbers in various ways. We use `make-pipe`, a program which evaluates its first argument but encapsulates its second argument into a promise: a suspended program that can be executed at a later time. Explanation of the details can be found in the previously cited references [1, 5]. In the interests of space we will make no excuses for using ANSI Common Lisp; there are excellent tutorials and texts on this language, and the grammar, perhaps more than any other language, follows from very simple rules.

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

Here is an alternative

[3] Including the thousands who annually take an introductory course at MIT, Berkeley or other schools using Abelson/Sussman [1]

[4] For the skeptic who holds that the only "real" portable programming language is C, and all programs must be written in such a standard, we note that free ANSI Standard Common Lisp implementations requiring only a C compiler, are available for almost any imaginable platform. From this perspective, Lisp is C. For Fortran fans, the differences between generating Fortran and C are easily overcome.

```
(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 the promise is a function which is 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  ;; the usual Fibonacci stream 1 1 2 3 5 ...
    (make-pipe 1 (make-pipe 1
                            (map-pipe #'+ fibs (tail-pipe fibs)))))
```

and a pipe of factorials might be

```
(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. Such a representation is motivated by the notion that many real functions of a real variable can be represented as an approximate polynomial expansion in a region by a sum such as $\exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots$ near $x = 0$. Such a sum, when truncated at a given point and evaluated carefully for small enough x is a good approximation for the exponential function. Generalizing this and treating power series as a *formal algebraic system*[5] is a common theme in computer algebra systems such as Maple and Mathematica.

Here are some simple programs which assume that we have agreed upon a particular variable and expansion point (say x and 0) 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))
                       (ps-add (tail-pipe x)(tail-pipe y))))))

;;Multiplying them

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

[5]The notion of error from truncation of the series or by finite arithmetic approximation, vital to consider in analytic applications, is usually ignored in the algebraic processing. Following tradition, we will ignore error terms too.

```
              (scale-pipe (tail-pipe s1)(head-pipe s2))
              (ps-mult s1 (tail-pipe s2)))))

(defun scale-pipe  (s f)
  ;; multiply each element in f by the scalar s
 (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 [9]. See also van der Hoeven's paper [8] for a more complicated but asymptotically faster lazy power series multiplication program based on Karatsuba's algorithm.

Lisp is capable of dealing with symbols as well as numbers. Unless the program requires arithmetic on the series data itself, we can encode a series with symbolic coefficients like $a + bx + cx^2 + \cdots$ as a pipe beginning `(a b c ....)`. We will return to non-numeric pipes subsequently.

How might we proceed to extend applications to numerical scientific applications?

As an example one can set up the solution of single (or systems of) ordinary differential equations. A naive, yet sometimes sufficient, approach (fixed step size, fixed order) solver is given in the fourth-order Runge-Kutta formula which produces a step-wise solution to $dy/dx = f(x, y)$ starting from an initial (x, y) pair, and continuing at step s

```
(defun rk4(f x y step)
  (make-pipe
   y
   (rk4 f
 (+ x step)
 (let* ((h  (/ step 2))
(k1 (* step (funcall f x y)))
(k2 (* step (funcall f (+ x h)(+ y (/ k1 2)))))
(k3 (* step (funcall f (+ x h)(+ y (/ k2 2)))))
(k4 (* step (funcall f (+ x step)(+ y k3)))))
   (+ y (* 1/6(+  k1 k4 (* 2 (+ k2 k3))))))
 step)))
```

We revisit this particular pipe with *symbolic* inputs subsequently: Can we make this or similar programs run faster by open-coding it a few steps?

Also, 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. [7].

3 Good news / Bad news

The notion of running a computation with potentially infinite objects of this kind is rarely taught in introductory courses (except those using the Abelson/Sussman text [1]).

As mentioned earlier, we suspect that many programmers are simply unaware of pipes/streams and that of those aware of the technique it is deemed unsuitable for implementation reasons:

1. Popular efficient programming languages (C, C++, Fortran, Java) don't support full-fledged function objects. Without these features available for implementing pipes, programs are difficult. With these features, programmers may discarded pipes in the (perhaps mistaken) belief that using these features leads to inefficient programs.

2. Since today's microprocessors are increasingly built around hardware pipelines of arithmetic units, programmers avoid dynamic data structures and procedure calls in the belief that maintaining the hardware pipeline is vital.

3. (Specifically for using a series as successively lower-order digits in a bigfloat) Rather than burping out more digits (or "bigits") on demand, a more plausible approach today is to take advantage of typical parallel/pipeline architectures. One would try to compute as much as possible first in machine precison whether or not it is suspected that additional precision is required for the solution of the problem at hand. If it is determined that precision P (greater than machine precision) is required or at least advisable for checking, 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/stream/pipe -based computation is a mechanism for representing a *computation* much as a source-code listing represents a *program*.

Rather than treating one of the Lisp pipe-based programs as an execution recipe that must be followed literally including the manipulation of one or more promises[6], we should realize that the pipe is an abstraction that can be compiled into sequences of operations in the same way that any other program structure can be compiled ultimately into machine language. Indeed, we can use this viewpoint of our 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 abstract operations. This can then be followed by a translation to low-level code appropriate for the execution environment, and with suitable low-level optimizations[7].

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

1. 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.

2. The "quick fix" approach used in several texts [1, 5] is to use "memoization," which is to say the first time any element of a pipe is computed, it is remembered so the cost of subsequent "re-computation" is reduced to that of the data access. An efficient implementation as a vector (rather than the usual linked list) is sometimes argued as a way to make this efficient. Unfortunately, this is unconvincing if the elements are computed or accessed only once or very few times: storing them at all is wasteful.

3. A less traditional approach is to look at the program at compile time and run it through an analyzer which will attempt to partially execute and simplify code for each element, symbolically. The resulting simplified code can be executed later without reference to the pipe abstraction, and hence without any particular additional (real or imagined) extra burden. A useful collection of papers on partial evaluation appears in a recent ACM Computing Surveys [3], and an online bibliography by Sestof [6] provides pointers to the field as of late 1998.

   Partial evaluation at the symbolic-numeric interface is what we will pursue for the remainder of this paper.

[6] Some readers may have heard the term *thunk* for a related but dissimilar concept in Algol 60

[7] It is a paradox of software technology that high-level languages including those in computer algebra systems, which presumably allow the programmer to specify *exactly* what should be computed, seem to be compiled to *less efficient* code than intermediate-level language like C. It is also curious that programmers who rail against some high-level languages as being "too inefficient" will program in C, and thereby ignore other far more significant speedup factors available to the assembly language programmer. We suspect that the trade-off on efficiency of languages may be, for some programmers, merely an excuse to use a familiar language which by coincidence is "just efficient enough".

For this mapping of an abstraction to code to provide a substantive benefit, the elements must themselves be parameterized by some value(s) known only at run-time (otherwise one might as well compute at compile time all the elements one expects to need, and store them in an array of constants!) For example, first consider a program like this:

```
(defun h (x)
  (let ((ans nil))
    (setf ans  (make-pipe x (map-pipe #'(lambda(r s)(* r s))
                                      (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. This is hardly interesting for us, since this could all be precomputed at compile-time before any accesses are made.

Let's slightly modify our basic function to add a parameter. `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` (short 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. There is a difference between the Lisp and the C version in that we can set up Lisp so that the list is continuable; that no programming efforts are needed to find the fourth and later values in that returned list. For C, we are restricted to the prespecified n.

Notice that we are generating names as necessary for values we have computed, but only for those that we will need again.

We will address several plausible objections to this solution technique in the next section.

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 as given and run a post-processing 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 possibly quite efficient program, using pipes either not at all (C-code) or using pipes only "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 and some solutions

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 create a program segment to assign, in order, the values for these variables, then we are set.

4.2 Other 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 implemented more efficiently 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. We might learn this at analysis/compilation time or at some early run-time stage. In fact it is likely that we will only rarely need "all" n elements in some particular case that makes pipe-analysis especially tempting. 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.

4.3 Generalization to conditional code

This kind of code seems fairly easy to control when we have a mapping from one or more 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? Simply this: 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 ∞ where `min(x,`∞`)` is `x`.

Our design works this way: Given 2 ordered pipes[8], 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))` $\rightarrow$ `(if a (min x b)(min x c))`,
`(min x inf)` $\rightarrow$ `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)
```

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

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

This simplification requires 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))
```

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

```
      (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>))
```

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

5 Live-dead analysis and composition of series

We first introduce the example of composition of power series such as $U = \{u_0, u_1, u_2, ...\}$ and $V = \{0, v_1, v_2, ...\}$ in this section, to motivate some further analysis.

5.1 Composition

Series composition where we compute $S(x) = U(V(x))$ as a power series, is a cute illustration of software pipe code-generation from a stream specification. (Note [4] this make sense if $v_0 = 0$).

In the program below, generated by composition, seven terms are available explicitly, although if we leave the program as given in its Lisp form rather than converting to C, more can be found by simply asking for them, and hence running the associated promise.

```
(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>))
```

Observe that as a consequence of writing this out as a straight-line program we can tell that there are 4 additional operations between computing `w0` and P`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 by computing differences of the indexes we can observe it is a cubic algorithm.

5.2 Live-dead analysis

A conventional compiler optimization, used to keep down the number of machine registers in use, is live-dead analysis. This is an attempt to take note of the variables which are set and then used, but then not used again: perhaps we don't really need all those names in the program above. In the absence of aliasing (a way of accessing values by different names) the computation is logically rather simple. A program (LDA) in the Appendix does this work (in about 15 lines of Lisp).

Run on the example above, LDA shows the sequence of computations above could re-use some of the names. For example since `w2` is not used again after it is used in `w4`, we can re-use it in place of `w5` and have one fewer variable.

After such processing, the output program then contains only 41 unique variables. The program above needs some more modification because sequentially *rebinding* the same names is not usually legal. We therefore changed the program LDA generation scheme to establish names and then issue a sequence of assignment statements. Such a Lisp program (here we show for only 4-deep composition) looks like this:

```
(let (w0 w3 w2 w7 w10 w1 w5)
  (setf w0 (* u1 v1))  (setf w1 (* u2 v1))  (setf w2 (* v1 w1))
  (setf w3 (* u1 v2))  (setf w2 (+ w2 w3))  (setf w5 (* u3 v1))
  (setf w5 (* v1 w5))  (setf w7 (* u2 v2))  (setf w5 (+ w5 w7))
  (setf w1 (* v2 w1))  (setf w10 (* u1 v3)) (setf w5 (* v1 w5))
  (setf w1 (+ w1 w10)) (setf w5 (+ w5 w1))
  (append (list u0 w0 w2 w5)
          #<interpreted closure (:internal combine-runs) @ #x46a4d5a>))
```

Or in some other kind of language like C we can generate

```
{ double w0, w3, w2, w7, w10, w1, w5;
  w0=(u1*v1);w1=(u2*v1);w2=(v1*w1);w3=(u1*v2);
  w2=(w2+w3);w5=(u3*v1);w5=(v1*w5);w7=(u2*v2);
  w5=(w5+w7);w1=(v2*w1);w10=(u1*v3);w5=(v1*w5);
  w1=(w1+w10);w5=(w5+w1);
list(u0,w0,w2,w5)
}
```

5.3 ODEs revisited

Using a "symbolic" modification of the Runge-Kutta code, we can generate what amounts to an infinite sequence of code segments to march it to any number of steps. Applying `analyze-pipe` to write out code explicitly for the first 3 steps, we get code that looks like this (after some light editing for readability):

```
{double w5, w4, w7, w2, w13, w1, w0, w9;
       w0=s+x0;
       w1=0.5*s;
       w2=f(x0,y0);
       w2=s*w2;
       w4=x0+w1;
       w5=s*f(w4,y0+0.5*w2);
       w4=s*f(w4,y0+0.5*w5);
       w7=s*f(s+x0,y0+w4);
       w2=(1/6)*(2*(w5+w4)+w2+w7)+y0;
       w9=s+w0;
       w9=f(w0,w2);
       w9=s*w9;
       w1=w0+w1;
       w13=s*f(w1,0.5*w9+w2);
       w1=s*f(w1,0.5*w13+w2);
       w0=s*f(s+w0,w1+w2);
       w9=w2+(1/6)*(w9+w0+2*(w13+w1));
```

```
list(y0,w2,w9);
}
```

Why bother? Given some sample codes as in the Appendix, we suspect that the transparency of pipes may provide the programmer prototyping ODE scientific codes a useful abstraction. Note, for example, that this code does not need to mention storage allocation. There is no explicit iteration either.

One can consider using this Runge-Kutta "self-starting" method to initiate a sequence of values which are then used for other solution methods. One can also continue with RK by changing the interval between steps at will (although in such a case we would recommend returning a pipe of (x, y) values instead of just y values. If, in anticipation of a relatively smooth curve, we use Hamming's method to more rapidly compute a solution, we use (x, y', y) triples. It may be plausible to include a measure of local error, making the stream a collection of quadruples. Additionally, we can build upon the pipes to (say) interpolate between terms or extrapolate. Further detailed discussion of the *many* variations possible in numerical ODE solution is beyond the scope of this paper. Any modern numerical analysis text will treat this topic in some detail.

Although it is not evident in the displayed C code, the Lisp code (with suitably modified interpretation of arithmetic) could be executed on vector arguments. Although we have not done so, a parallel-C version could also be generated. The very same routine could also be used as a simple-minded program to solve a system of initial value problems.

We do not expect that compile-time analysis such as suggested here will yield superior fully-general code compared to that which has already been produced in sophisticated programs[9]. In particular, methods in which local error estimates affect the course of the program cannot be pre-generated as straight-line code. Note that popular high-quality ODE programs, even those that are supposed to be user-friendly[10] are rife with special flags, or choices for the user to make based on (sometimes limited) evidence.

This leaves us the possibility that creation of special purpose programs via code generation will be viable in some contexts, where (in effect) the settings of those flags can be known in advance, or a repeatedly executed special code can be fitted to a situation of interest.

6 Other operations, and Horner's rule

Returning once again to power series, there are a variety of other algorithms, most notably power series division and reversion, that can be implemented nicely as pipes. Including these in nice packaged facility might then provide all the tools needed for the elementary application taught in introductory differential equations courses: power series solution of ordinary differential equations. Although this is not a currently popular numerical technique, (certainly compared to the vastness of the ODE libraries), it occasionally reappears. In any case the basic algorithms are described by Knuth [4].

6.1 Sparseness in a series or a polynomial

We use the task of evaluation of a power series at a scalar point to raise the issue of sparseness. First consider evaluation of a series at a point s that is small enough that terms far out in the series, multiplied by s^n will be negligible contributors to the value.

```
(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)
```

[9] see for example `www.netlib.org` for a collection of such programs.

[10] such as the 7 programs in the MATLAB ODE suite

```
        (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)))))))
```

This may not be the right pipe organization if in fact the terms far out in the series are non-negligible contributors. Then we are really dealing with what would be more plausibly be considered polynomial arithmetic. In such a case it is expected that the pipe will be finite in length and it turns out to be convenient for most operations to arrange the pipe (or simply a list since it is finite) so that the highest degree is first. A modern view of computer algebra implementation of polynomial arithmetic 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` We will revisit this in the next section.

Other aspects of generation of polynomial evaluation code are explored in Fateman [2]. In particular, code can be developed to exploit the special structure of the polynomial being evaluated. Surprisingly, we can make use of even such a limited datum as its degree. If we know its coefficients, we can pre-compute coefficients of auxiliary polynomials, find roots, factor. We can consider special knowledge of the point(s) at which it is evaluated (real or complex), and special knowledge of the computer on which it is run (pipe-line depth and number of arithmetic units) [2].

6.2 Multi-term floating point 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 exactly by only two numbers: (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 [7, 4]. As shown by Shewchuk [7], 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" 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 [7].

6.3 Random walks

A final example in our code file, suggested by Tunc Simsek, uses pipes for random walks. This is of some interest in simulations. Consider a pipe of random directions (say, restricted to one unit +/- real or imaginary in the complex plane. We can build a pipe trivially to accumulate the location after each step, or the distance from the original after each step. Code generation to run this out to some length is not going to work with our conventions here: we must change the programs to recognize that our random number generation is not a function[11] Under this interpretation `(+ (random 4)(random 4))` is simplified to `(* 2 (random 4))`, which is not acceptable. By adding an extra argument which changes each time we call `random`, we avoid this difficulty.

See the file `pipe.cl` for details.

7 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 source code that can be translated line-by-line into assembly-language code necessary for a computation. 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 conventional sophisticated numerical code compilers. For best performance, the resulting source code would be optimized yet again, by a traditional

[11] If we were to treat the pseudo-function `(random 4)` as a true function, we would believe it would return the same value each time!

compiler, in the context of a particular machine model, taking into account the number of registers, the available parallelism, cache performance, and other considerations.

8 Appendix

The code referred to in this paper is available in the following files:

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

9 Acknowledgments

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

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 polynomial evaluation." (submitted for publication).

[3] O. Danvy, R. Gluck, P. Thiemann. "1998 symposium on partial evaluation" *ACM Computing Surveys 30*, no 3 (Sept., 1998), 285–290. See also other papers in that issue.

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

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

[6] P. Sestof "Bibliography on partial evaluation and mixed computation (BibTeX format) (last update 1998) ftp://ftp.diku.dk/pub/diku/dists/jones-book/partial-eval.bib.Z.

[7] 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 p. 141–150.

[8] van der Hoeven, J. "Lazy multiplication of formal power series," *ISSAC '97* Maui, Hawaii. p. 17–20.

[9] 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.

[10] Richard C. Waters, "Automatic Transformation of Series Expressions into Loops," *ACM Trans. on Prog. Lang. and Sys. 13* No. 1, (January 1991) p. 52–98.