Code generation: evaluating polynomials

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

August 28, 2002

Abstract

Writing a program to evaluate a given polynomial at a point can be done rather simply. If it has to be done repeatedly, efficiently, and/or accurately, a naive approach may not suffice. There is in fact a huge design space of how to approach this problem [12]. Depending upon one's criteria, it may be worthwhile to use automated program-generation programs to write source code. Such generated code can be structured to exploit whatever special knowledge of the polynomial being evaluated may be available early, as well as the design requirements. This knowledge can include its degree, some or all of its coefficients (used in pre-computing coefficients of an auxiliary polynomial), special knowledge of the point at which it is evaluated (real or complex), special knowledge of the computer on which it is run (pipe-line depth and number of arithmetic units), the required error tolerance, and whether auxiliary information is needed, such as the first or second derivative of the polynomial. It is possible also to indicate exactly the number of arithmetic operations used, and in some cases, other information describing the efficiency of the code. In this paper we describe some program-generating programs that may be of assistance. All the code is available in an on-line appendix. Although the implementation language is ANSI standard Common Lisp, the target language is (your choice) of Lisp or C.

1 Introduction

Most practitioners of scientific programming view evaluating a polynomial as quite rudimentary and not worth deep examination. Merely typing into Fortran something like `y=3*x^2+4*x+1` would seem to do it, though many would immediately re-write this as `y=1+x*(4+x*3)` to save a multiplication.

Writing a program that takes an array of double-float coefficients, the integer degree of the polynomial, and the point at which it is to be evaluated, is not that much harder, and Horner's rule provides a good method. Given a programming language that deals nicely with data aggregates like lists, (e.g. Lisp) one can define a Horner's rule program

```
(defun ev(p x) (reduce #'(lambda( s r)(+ r (* x s))) p))
```

such that `(ev '(3 4 1) 7)` evaluates the given polynomial at `x=7.` to 176.

Do you care how accurate the answer is? If you are evaluating this polynomial to find a root using a Newton iteration, you will need to know its derivative. One can, in general, evaluate a derivative as part of the same computation.

Our ability to satisfy such specific needs lies at the core of many more sophisticated routines. Most polynomial zero-finding routines eventually evaluate the polynomial.

Backing off a bit from the specific objective of polynomials, we will examine code generation as a tradeoff between program re-use (taking a program from a library) and special single-purpose programs. Our objective is to provide a program framework, but not the object code *per se*. The framework is re-used to generate single- (or at least special-) purpose code.

2 Horner's Rule, embroidered, then improved

The basic notion of implementing Horner's rule has been shown already: we write a loop that executes the appropriate adds and multiplies.

A modest change to that `ev` program is the one below. `ev-s` reminds us that it is like `ev` but symbolic. We use `*s` and `+s` to denote symbolic multiply and add routines which can either produce numeric answers if given numbers, or symbolic formulas. The `newname` program essentially renames a subexpression and places the name/value pairs on a list of variables that is then trotted out to produce a frame for the bindings by the `framevars` program.

```
(defun ev-s(p x)
  (framevars(reduce #'(lambda( s r)(newname(+s r (*s x s)))) p)))
```

allows us to feed in a *symbolic* `x` and a list of coefficients `(a b c d)` to generate the straight-line program below.

```
(let* ((w0 (+ b (* a x)))
       (w1 (+ c (* x w0)))
       (w2 (+ d (* x w1))))
    w2)
```

Knowing at run-time the values for `a,b,c,d`, we can execute this for its value. *Note the absence of a loop index or any branches.*

Observe that we can re-use the temporary names. A program for "live-dead analysis" detects that we are free to re-use `w0`, and provides an improved version of the result:

```
(let (w0)
  (setf w0 (+ b (* a x)))
  (setf w0 (+ c (* x w0)))
  (setf w0 (+ d (* x w0)))
  w0)
```

Observe that the program generation can use partial information simplified away as soon as possible. If some names are constant: `(ev-s '(1 2 c d) 3)` becomes

```
(let (w0) (setf w0 (+ 15 c)) (setf w0 (+ d (* 3 w0))) w0)
```

or even `(ev-s '(1 2 3 4 5) 6)` which simply returns `1865`.

Under some circumstances and some measures of complexity, such a version of Horner's rule may be optimal. But these may not be the circumstances at hand.

The usual nested multiplies and adds are optimal if the degree and coefficients as well as the evaluation point *are not known until the last moment.* If one knows the coefficients well in advance there is a substantial literature on preconditioning or adaptation of coefficients [10]; we assume for this section that we are willing to perform substantial numerical computation *once while the polynomial is being preconditioned* understanding that we will save effort each of the (many) times this polynomial is subsequently evaluated. A polynomial of degree six can always be evaluated with four multiplications and seven additions, but requires the pre-condition computation of the solution of a cubic equation. There is a general method that requires $\lfloor n/2 \rfloor + 2$ multiplications and n additions[1].

Knuth's [10] Theorem E of section 4.6.4 (p. 474) asserts that a polynomial $U(x) := u_0 x^n + ... + u_n$ with real coefficients, $n \geq 3$ can be evaluated as follows:

[1]This may not actually be faster if you have 2 multipliers and you are evaluating at ordinary floating-point numbers, as suggested in section 7.

Set $y = x + c$, $w = y^2$.

If n is even then set $z = (u_n * y + \alpha_0)y + \beta_0$ else $z = u_n * y + \beta_0$. We can then write

$$u(x) = (\ldots((z(w - \alpha_1) + \beta_1)(w - \alpha_2) + \beta_2)...)(w - \alpha_m) + \beta_m$$

for suitable real parameters, where $m = \lfloor n/2 \rfloor - 1$. In fact, it is possible to select these parameters so that $\beta_m = 0$. The details of implementation require finding the zeros of $u(x)$ or associated polynomials. Because of the need for algebraic computation, we will not pursue this path in our code generation. (In particular, the next scheme requires only rational preconditioning and only slightly more arithmetic at runtime.)

S. Winograd is credited with this next technique, initially proposed for a monic (leading coefficient unity) polynomial. It uses only rational operations (not root-finding) and requires about $n/2 + \log_2 n - 1$ multiplications and $(5/4)n$ additions.

To begin with, one computes x^2, x^4, up to x^{2^l} where $l = \lfloor \log_2 n \rfloor$.

Then the basic idea is to compute $u(x)$ of degree $n = 2^l - 1$ as follows. Let $m = 2^{l-1} - 1$. Then write $u(x) = (x^{m+1} + \alpha)v(x) + w(x)$ where $v(x)$ and $w(x)$ are monic of degree m and $\alpha = 1 - u_m$. Note that $x^{(m+1)}$ is one of those pre-computed powers of x. For this to be feasible, the coefficients of v and w should be numeric so that "synthetic division" can be used to find their coefficients. What if n is not so coincidentally one less than a power of 2? If n is 2^l, then compute $u_0 + x*(u'(x))$ where u' is now of degree $2^l - 1$ and monic, thus fitting into this scheme. The final situation is n is between powers of two, say $m = 2^l < n < 2^{(l+1)} - 1$ in which case we write $u(x) = x^m v(x) + w(x)$ where v is of degree $n - m$ and w is of degree m. The one-coefficient overlap in the polynomials allows us to make w monic, and so computing v and w can be done by this algorithm, once again. (See Knuth [10], new in 3rd edition of volume 2, exercise 65, section 4.6.4).

By actual generation of code we observe that for a monic polynomial of degree 15 the code uses 21 adds, 7 multiplies and 3 squarings. This compares to 15 adds and 14 multiplies for Horner's rule (Since it is monic, we save one multiply).

It is implausible to use this technique on *symbolic* polynomials because the result of synthetic division would not condense down to simple numbers. Instead we show a particular numerical example. For illustration, we once again print out a C-like form (here, slightly edited), in this case for a particular monic degree-15 polynomial:

```
 (setf r (win65'(1 -2 3 -4 5 -6 7 -8 9 -10 11 -12 13
-14 15 -16) 'x))

{double w0, w1, w2, w3, w5, w6, w9, w10, w12 w13;
      w0=x^2;  w1=w0^2;  w2=w1^2; //w2=x^8
      w3=(-2)+x;
      w3=x+w3*(2+w0);
      w5=2+x;
      w6=20+x;
      w5=w6+((-6)+w0)*w5;
      w3=w5+w3*(4+w1);
      w9=6+x;
      w10=104+x;
      w9=((-14)+w0)*w9+w10;
      w12=202+x;
      w13=82620+x;
      w0=w13+((-406)+w0)*w12;
      w1=w0+w9*((-28)+w1);
      w2=w1+(8+w2)*w3;
```

```
w2;
}
```

Using this program for an arbitrary (non-monic) polynomial is simple enough: At code-generation time, divide the coefficients through by u_m, the leading coefficient, and as a last step, multiple the result by u_m.

Each of the polynomial evaluation schemes has different properties with respect to round-off errors, and the code necessary to generate error bounds, if enabled, roughly triples the number of operations needed here. There seems to have been almost no systematic study of the error propagation behavior of these "improved" polynomial evaluation formulas, although it is commonly conceded that they tend to have, at least in some cases, substantially higher errors. Experiments with our programs could certainly aid in the statistical evaluation of the likelihood of large errors being encountered. In fact if knowledge of the errors is required, it is plausible that using Horner's rule would provide lower error and ultimately faster computation. In another section we describe how one might satisfy a requirement of minimal error (full machine precision).

For a description of how error bounds can be computed, see Appendix II.

3 Other code generation

The determinant of an $n \times n$ matrix may be considered to be a polynomial in n^2 variables x_{ij}, $1 \leq i, j \leq n$. Knowing that we can evaluate this polynomial with only about $2n^3/3$ arithmetic operations is rather remarkable because it is a polynomial which, if fully expanded, would have $n!$ monomial terms with n variables in each term.

We can use much the same tools we have for used for Horner's rule, and described in the next section, to generate straight-line programs. Here is the code result to compute a 4 by 4 determinant using a kind of symbolic Gaussian elimination (Knuth [10] section 4.6.4):

```
 (det-s '((a b c d)(e f g h)(i j k l) (m n o p)))

(let (w1 w6 w7 w8 w0 w10 w5 w2 w3 w15 w4)
  (setf w0 (/ -1 a))
  (setf w1 (+ f (* b e w0)))
  (setf w2 (+ g (* c e w0)))
  (setf w3 (+ h (* d e w0)))
  (setf w4 (+ j (* b i w0)))
  (setf w5 (+ k (* c i w0)))
  (setf w6 (+ l (* d i w0)))
  (setf w7 (+ n (* b m w0)))
  (setf w8 (+ o (* c m w0)))
  (setf w0 (+ p (* d m w0)))
  (setf w10 (/ -1 w1))
  (setf w5 (+ (* w10 w2 w4) w5))
  (setf w4 (+ (* w10 w3 w4) w6))
  (setf w2 (+ w8 (* w10 w2 w7)))
  (setf w3 (+ w0 (* w10 w3 w7)))
  (setf w15 (/ -1 w5))
  (setf w4 (+ w3 (* w4 w2 w15)))
  (* a w1 w5 w4))
```

This program assumes that certain pivot elements will be non-zero. This requirement can be avoided in at least two ways.

1. We could use run-time tests to check for and take advantage of the zero-pivot situation. (However, generating conditional statements is somewhat contrary to our straight-line program philosophy). Yet such manipulation is always possible for any non-singular matrix by the use of row and/or column operations.

2. Since determinant calculations need not have divisions, we can use a division-free algorithm.

Here's a division free determinant procedure using expansion by minors. Running this on the same example produces:

```
 (det-sx '((a b c d)(e f g h)(i j k l) (m n o p)))

(let (w3 w4 w5 w0 w7 w1 w2)
  (setf w0 (+ (* -1 l o) (* k p)))
  (setf w1 (+ (* -1 l n) (* j p)))
  (setf w2 (+ (* -1 k n) (* j o)))
  (setf w3 (+ (* h w2) (* -1 g w1) (* f w0)))
  (setf w4 (+ (* -1 l m) (* i p)))
  (setf w5 (+ (* -1 k m) (* i o)))
  (setf w0 (+ (* e w0) (* -1 g w4) (* h w5)))
  (setf w7 (+ (* -1 j m) (* i n)))
  (setf w1 (+ (* e w1) (* -1 f w4) (* h w7)))
  (setf w2 (+ (* e w2) (* -1 f w5) (* g w7)))
  (+ (* a w3) (* -1 d w2) (* c w1) (* -1 b w0)))
```

re-expressed in C, using our prefix-to-infix program `p2i`

```
{double w0, w1, w2, w3, w4, w5, w6, w7;
      w0=-l*o+k*p;
      w1=-l*n+j*p;
      w2=-k*n+j*o;
      w3=h*w2-g*w1+f*w0;
      w4=-l*m+i*p;
      w5=-k*m+i*o;
      w0=e*w0-g*w4+h*w5;
      w7=-j*m+i*n;
      w1=e*w1-f*w4+h*w7;
      w2=e*w2-f*w5+g*w7;
      a*w3-d*w2+c*w1-b*w0;
      }
```

This result is refined somewhat from our first program which generated expressions that were, in some cases, simply re-computing known results (for sub-sub-minors). We therefore changed the program framework so that before generating a new assignment (via `newname`, described below) allocated a new variable, our program checked to see if the value was already available under an existing name.

The determinant procedure can take into account certain structural information to shorten the generated program such as this example with several zero entries:

```
(det-s '((a b c d)(0 f g h)(0 j k l) (0 n o p)))

(let (w2 w4 w0 w6 w3)
  (setf w0 (/ -1 a))
```

```
(setf w0 (/ -1 f))
(setf w2 (+ k (* g j w0)))
(setf w3 (+ l (* h j w0)))
(setf w4 (+ o (* g n w0)))
(setf w0 (+ p (* h n w0)))
(setf w6 (/ -1 w2))
(setf w3 (+ w0 (* w3 w4 w6)))
(* a f w2 w3))
```

Ultimately the size of the generated programs becomes unwieldy: they grow as n^3. At some point computing larger determinants, or writing programs that must receive the dimensions as a parameter would be done via closed, looping subroutines. It is feasible although not particularly attractive in this case to generate a combined program, where certain levels are compiled as loops and subcases eventually are compiled open. For large matrices computation of numerical determinants by expansion by minors is a loser as the number of minors grows exponentially, and Gaussian elimination is only cubic. How does it look for small matrices?

As a side effect of our program generation, we can routinely keep count of the number of arithmetic operations which are generated in the straight-line code. Thus the first determinant program uses 14 adds, 23 multiplies and 2 divides. The second, expansion by minors, uses 17 adds and 40 multiplies. However 12 of the multiplies are by -1.

This kind of analysis can be used to (for example) generate several variants of a program and pick the one which is better by some weighted criterion of operation counts. It is also possible to add tools to keep track register scheduling or pipeline activity, but this requires that we pay careful attention to particular machine architectures. As has been shown in the ATLAS (Automatically Tuned Linear Algebra Software) project [2], there are many issues related to the deep memory hierarchies on today's high-performance computers that cannot be addressed in a machine-independent high-level language.

4 Tools for program generation

We use three components in our program generation. They interact in several key ways to push forward as much computation as possible using as much information as is available at compile-time.

- Partial evaluation: If a compiler is offered a program that looks like $x := a + 3 + 4$ it is generally clever enough to change that to $x := a + 7$, if not something even simpler based on knowledge of a and what is subsequently done with x. Partial evaluation of arithmetic expressions in our context removes all arithmetic that can be done at compile time to constants. It also includes unwrapping all loops with a fixed index set (the upper and lower bounds are known at compile time). When arithmetic must be partially evaluated, we indicate this by substituting in our generation program symbolic versions of arithmetic. For example, `(*s ....)` for `(* ...)`. Arguably the symbolic version can *always* be used because its semantics devolves to ordinary arithmetic, given numeric data.

- Generating intermediate variables: In the course of running some of these generation procedures, plausible points are found when an expression that is generated will likely be re-used. At this point we propose to make up a name for this expression as a "temporary" and manage the use of that name until it is no longer needed. Essentially all that is necessary for the programmer is to wrap `(newname ...)` around a generated computed expression. As an additional feature, we can analyze the expression being computed and keep track, for each temporary, of how many additions, multiplications, etc. are required to compute its value.

- Simplification: This takes two forms. During partial evaluation, simplified expression values are computed. There is actually some tension as to how to best do this. In our examples here, `(- a b)` is

sometimes preferable to what, in most cases, works better for traditional computer algebra simplification `(+ b (* -1 a))`. This latter expression is more simplified because it uses only the most basic two operations plus a constant (which happens to be negative). The second form of simplification is contraction of the number of names needed during a computation. After the whole program is generated it may appear that we need only a subset of the names. That is, some names are dead (out of use) and can be re-used instead of generating new ones. While a good compiler may notice this, we find that doing live-dead analysis reduces the textual size of our programs, though at the cost of a somewhat mysterious program.

5 Expanded adaptive code

Consider now the situation of a programmer who wishes to write only in the most abstract level. All the code is written with generality and flexibility in mind. The proper execution of this code will require combining some information, typically the types of input, with the specification of the algorithm. This is a common idea, used even in the earliest Fortran compilers and passed down to today where most programmers do not think twice about the fact that $1 + 2$ and $1.0 + 2.0$ generate different code for the same "+" symbol. This notion of generic operations has been elaborated upon *ad nauseum* in the modern programming language design literature; users seeking the most efficient code possible tend to (wisely or not) heap derision on such languages.

Compile-time code expansion and partial evaluation can force such code into specialized forms in which the efficiency is foremost. This is done rather routinely by compilers choosing among well-known numeric types. For other types (for example, interval or rational arithmetic), older languages typically need preprocessing to make proper choices. Even with newer object-oriented languages, some programmers feel that any choices that must be made at run-time inevitably lead to inefficiency. Yet not all the specialization and choice is available as type information. Partial evaluation can make use (in principle) of *any* information, including perhaps special values of data, known array sizes, and even the number of available floating-point functional units in the pipeline of a targetted host machine.

Consider the (somewhat simplified for exposition) situation of sparse multi-term floating-point representation. This is a representation in which vectors of some format (typically d-digit double-precision) floats can be used to represent higher precision. An n-term number is a sum of d-digit numbers, not necessarily contiguous in exponent range. For example, if we were dealing with 3-digit decimals with 2-decimal digit exponents,, 123.0e90+456.0e-90 would be a legitimate 2-term number. We can produce n-term numbers for any fixed n [15] and perform arithmetic on them. There is a side computation which tells us how many terms we will need in the result; we can also provide specialized routines for multiplying 2- by 4-term numbers to produce 8-term numbers. Let us refer to this multi-term product as `mt&2&4`. Let us assume that multi-term numbers are denoted by `(mt x n)`. We can then generate a very specific Horner's rule for a degree-3 polynomial of double-floats evaluated at a 2-term number:

```
(ev-mt '(a b c d) '(mt x 2))
(let (w0)
  (setf w0 (mt+1&3 b (mt*2&1 x a)))
  (setf w0 (mt+1&6 c (mt*2&4 x w0)))
  (setf w0 (mt+1&9 d (mt*2&7 x w0)))
  w0)
```

In general the length of these multi-term numbers are overestimates on the size because adding two numbers of n and m terms may produce as many as $n + m$ terms, more typically $\max(n, m)$ terms, and possibly as few as 0 terms if they cancel. Thus the programs might substantially benefit from keeping track of the actual number of non-zero terms.

6 Complex polynomials

If the coefficients of a polynomial are complex, $U(x) := u_0x^n + ... + u_n = (a_0 + ib_0)x^n + ...$ then one can evaluate S using other programs as $U(x) := (a_0x^n + ... + a_n) + i(b_0x^n + ...b_n)$.

What, however, if we are computing $U(z)$, for a complex number z? We could do complex arithmetic as supported by many programming languages, but then the symbol "*" between a real and a complex counts for two floating-point "real" multiplications. If we use Horner's rule then we are using about $4n$ real multiplies and $3n$ additions. Or we could use about $2n$ multiplies and $2n$ additions in a scheme cited by Knuth [10] page 468, and rendered in Lisp this way:

```
(prc3-s '(a b c d) 'z)

(let (w1 w4 w5 w2 w0)
  (setf w0 (realpart z))
  (setf w1 (imagpart z))
  (setf w2 (* 2 w0)); 2*x
  (setf w0 (+ (expt w0 2) (expt w1 2))) ; x^2+y^2
  (setf w4 (+ b (* a w2)))
  (setf w5 (+ c (* -1 a w0)))
  (setf w2 (+ w5 (* w2 w4)))
  (setf w0 (+ d (* -1 w0 w4)))
  (+ w0 (* z w2)))
```

The previously mentioned "adaptive" varying-length system is orthogonal to this development, and could be used as well. Since the computation depends on $r = x^2 + y^2$ where the argument is $z = x + iy$, this routine is somewhat sensitive to the ratio of x and y. For small y, r may not be distinguished from x^2, so this technique has potentially less accurate results.

7 Parallel computation

Many of today's fast processors have multiple arithmetic units which make it possible to pipeline the processing of non-interfering operations. That is, one can compute $x := y*z$ and $a := b*c$ either simultaneously or very nearly so, as long as the result of the first computation is not needed for the second. (By comparison, $x := x * y$ followed by $a := b * x$ is slower because the second multiplication (of $b * x$) must wait for the first to complete.) Our Horner's rule computation has the character of this second sequence: we compute serially a multiplication and addition. This is not a requirement though. We can break the computation of $U(x)$ into two parts, the evaluation of the even-degree terms $e(x^2)$ and the odd-degree terms $f(x^2)$ then $e + x * f$ provides the same result, but e and f can be computed independently of each other (at least after x is squared). Note that in the following, `w1` and `w2` can be computed without reference to each other. This can be generalized for more processors or a deeper pipeline.

```
(pnneval4-s '(a b c d e f g) 'x)

(let (w1 w2 w0)
  (setf w0 (expt x 2))
  (setf w1 (+ c (* a w0)))
  (setf w2 (+ d (* b w0)))
  (setf w1 (+ e (* w0 w1)))
  (setf w2 (+ f (* w0 w2)))
  (setf w0 (+ g (* w0 w1)))
```

```
  (+ w0 (* x w2)))
```

8 How accurate is the result?

If error accumulation is of primary concern, and the cost of preconditioning is not of any concern, then there is a method requiring no more than n (= degree of the polynomial) multiplies and adds. The preconditioning would consist of finding all roots $\{r_i\}$of the polynomial and computing the product $\prod x - r_i$. This would have to be done with some attention to multiplicities, and could take into account complex-conjugate pairs. This may beg the question if the reason we are computing the value of the polynomial is to find its roots. Nevertheless, consider that we may nevertheless need this result. The major question is how to set an *a prior* bound for the accuracy needed for each r_i? The answer[2] is "sort of" that we need twice the precision P of x. Each root can be written as $r_h + r_l$ where r_h and r_l are each of precision P. Then $(x - r)$ should be computed as $((x - r_h) - r_l)$ where the subtraction $x - r_h$ can at worst cancel entirely, in which case the low-order bits must then be used for the term in the product. Why do we say "sort of"? Because if the root is $(1 + 2^{-200})$, $P = 53$, and $x = 1$, we do not want r_l to be the "next" 53 bits. It must be the next *significant* 53 bits, in this case 2^{-200}. Shewchuk's sparse arithmetic [15] provides this kind of information. The accuracy of this result is essentially full accuracy: the product is the (rounded) answer to machine precision.

If we have no preconditioning available and no *a priori* reason to prefer any particular range on the evaluation point, Horner's rule may be a good choice. In the next program illustrated, taken from an implementation of polynomial root-finding according to the method suggested in W. Kahan's "Notes on Laguerre Iteration" [7]. we may wish to compute other information. Here we return a collection of values including

- p the value of the polynomial $u(x)$,
- e its error. $e > |p - u(x)|$ (assuming the arithmetic parameter `eps` is set to a number reflective of the relative error in the representation (e.g. 2^{-53} for IEEE 754 double)),
- q is approximately $u'(x)/u(x)$, the derivative divided by the value.
- r is approximately $q^2 - u''(x)/u(x)$

  (We can also generate subsets of this information). The basic code `preval` from which this is developed is also provided in the Appendix I program file, and can be used for execution or text comparison purposes. The basic code assumes the input polynomial's coefficients are contained in an array. Frankly, in the context in which we wrote the basic Lisp code, it is unlikely that one would know the degree of the polynomial much beforehand: one generally needs a standard program for a polynomial of arbitrary degree.

  Nevertheless, if one knows ahead of time that a polynomial of specific degree will be evaluated (say in a constrained design situation, or an fixed approximation of a function) we can generate code that looks like this:

```
 (pn-s '(a b c d) 'x)

(let (w1 w7 w4 w2 w0)
  (setf w0 (abs x))
  (setf w1 (* 0.7d0 (abs a)))
  (setf w2 (+ b (* a x)))
```

[2] Thanks to J. Demmel for this suggestion

```
(setf w1 (+ (* w0 w1) (abs w2)))
(setf w4 (+ w2 (* a x)))
(setf w2 (+ c (* x w2)))
(setf w1 (+ (* w0 w1) (abs w2)))
(setf w7 (+ w4 (* a x)))
(setf w4 (+ w2 (* x w4)))
(setf w2 (+ d (* x w2)))
(setf w0 (+ (abs w2) (* w0 w1)))
(values w2   ;; p e q r
   (* eps (+ (* 2.31d0 (+ w0 (* -1 (abs w2)))) w0))
   (/ w4 w2)
   (+ (expt w4 2) (/ (* -2 w7) w2))))
```

It is not difficult to produce a formula for accumulating the maximum error committed in a sequence of arithmetic calculations by, in effect, evaluating the expression using interval arithmetic. (See Appendix II) Horner's rule benefits from particular simplifications that can be made in (for example) assuming that you are evaluating at a point x which is exactly the number represented in the machine. In general some thought can short-cut the sequence of operations needed to simplify the interval operations.

9 Dealing with overflow

We would like to be able to say that we can generate code that deals especially well with overflow. We can't because we don't have control of your run-time environment. We don't have any special insight into scaling of the polynomials. Sorry.

10 Evaluating polynomials at regular intervals

If we wish to evaluate a polynomial $u(x)$ of degree n at many points in arithmetic progression $u(x_0 + jh)$, $0 \leq j \leq N$, there is a scheme where the arithmetic can be reduced eventually to n additions for each additional point. We mention at the outset that this scheme depends upon repeatedly adding together increments that are not in general exactly representable, and so this process may result in an increase in the accumulated error as the number of additional points increases. Some attention can be paid to either restarting the computation periodically, or running it "backwards" from the endpoint with negative increments.

A pretty explanation of the process can be found by looking at the theory of divided differences; here we will refer once again to Knuth, section 4.6.4 (Note: the 3rd edition corrects an error in the relevant solution to problem 7 of the 2nd edition). We consider the kind of program that can be generated in symbolically tabulating the computation. First we are set to compute some specified number of terms t in a vector, given the polynomial $f(x) := U(x)$ of known degree. We will compute $\{f(x_0), f(x_0+h), f(x_0+2h), \ldots, f(x_0+(t-1)h)\}$ How much of this needs to be known at "compile" time? We require specific information on the degree of U. We benefit from knowing a specific number t, but we can remain open to generating code without it, too. If both are supplied we can generate a symbolic polynomial tabulation code like this (we have added some comments):

```
(tab2-s '(a b c) 'x 'h 6); polyn a*x^2+b*x+c, any x, any h, 6 answers

(let (w0 w1 w2 w3)
  (setf w0
  ;; w0 is the name for a function to eval the polynomial
```

```
      #'(lambda (thevar)
          (let (w0)
            (setf w0 (+ b (* a thevar)))
            (setf w0 (+ c (* thevar w0)))
            w0)))
(setf w1 (funcall w0 x))  ;eval f(x)
(setf w2 (+ h x))
(setf w3 (funcall w0 w2)) ;eval f(x+h)
(setf w2 (+ h w2))
(setf w0 (funcall w0 w2)) ;eval f(x+2h)
(setf w0 (+ w0 (* -1 w3)));construct differences
(setf w3 (+ w3 (* -1 w1)))
(setf w0 (+ w0 (* -1 w3)))
(vector (prog1 w1 (incf w1 w3) (incf w3 w0)) ;; expanded out
        (prog1 w1 (incf w1 w3) (incf w3 w0)) ;; prog1 returns
        (prog1 w1 (incf w1 w3) (incf w3 w0)) ;; first item.
        (prog1 w1 (incf w1 w3) (incf w3 w0)) ;; (incf a b)
        (prog1 w1 (incf w1 w3))              ;; is like a=a+b
         w1));; vector is of f(x) ... f(x+(t-1)*h)
```

If the number of terms to be used is not known when the program is being generated, the explicit expansion of the vector t times is not possible. In that case the clause beginning `vector` can be replaced by a loop that looks like this:

```
(loop for count from 0 to (1- n) collect
      (prog1 w1 (incf w1 w3) (incf w3 w0)))
```

The idea here is to compute $f(n)$, for $0 \leq n \leq N$ in a loop relying on the ordering of the sequence $0, \ldots, n, \ldots$ to use pre-computed values of $f(j)$, or associated differences of these values with $j < n$ to assist in computing $f(n)$. It is certainly possible to consider some auxiliary functions for a shortcut to computing $f(n)$.

This tabulation method can be generalized in various ways for n being a vector of indices and f really being $f(x, n)$ with x a vector of parameters. This is especially useful if a *large* number of values are required of a function defined on a grid. This is an especially common situation with plotting software, and suggests that the generalization where n is a pair of coordinates is useful. The accuracy requirements for plotting are expected to be somewhat lower than in other numerical computation since we can rarely resolve on a plot anything near the precision provided by floating-point representations. Thus we are generally willing to trade off some accuracy for considerable speed in this situation (We remove all n multiplies for each additional data-point).

This idea is not restricted to polynomials. As an example of non-polynomial adjacent functions, consider computing $\sin(n\theta)$ and $\cos(n\theta)$ for fixed θ and for $0 \leq n \leq N$.

For $N > 3$ the best is using a recurrence based on two previous values, which for each additional sin and cos pair takes just two adds and two multiplies. Starting with $\sin(0) = 0$ and $\cos(0) = 1$ and using the relationships

$$\begin{aligned} \cos(n\theta) &= 2\cos(\theta)\cos((n-1)\theta) - \cos((n-2)\theta) \\ \sin(n\theta) &= -2\sin(\theta)\sin((n-1)\theta) + \cos((n-2)\theta) \end{aligned}$$

we can construct a simple program that first computes $2\cos(\theta)$ and $-2\sin(\theta)$, and then writes out values at some (presumably substantial number of) arguments that are multiples of θ. A slight generalization allows us to compute $\sin(n * \theta + d)$ for given d.

The simple systematic approach based on divided differences does not immediately apply to the case where the short-hand recurrence is a product or more elaborate relation such as used in our simple sin/cos example. A more elaborate theory and a practical implementation on multi-term recurrences has been developed by E. Zima and his collaborators calling the technique "computing with chains of recurrences" [4, 9]. Somewhat in common with our approach in this paper, they are ultimately targeting the generation of programs. The discussion below follows Bachmann's development [4].

A basic recurrence (BR) f is a function represented as a triple: $f = \{\phi, \odot, f_1\}$ where ϕ is an expression, $\odot$ is the operator + or *, and f_1 is a function of one argument. f is itself a function of one argument.

If $\odot$ is +, the BR is defined as

$$f(i) = \phi + \sum_{0 \le j \le i-1} f_1(j)$$

If $\odot$ is *, the BR is defined as

$$f(i) = \phi \prod_{0 \le j \le i-1} f_1(j)$$

The notation is extended to define chains of recurrences (CR). A CR allows for more operations by extending $f = \{\phi, \odot, f_1\}$ to

$$f = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \phi_k, f_k\}$$

where the extra elements are incorporated via nested BR as

$$\{\phi_0, \odot_1, \{\phi_1, \odot_2, \cdots \phi_k, f_k\} \cdots\}$$

An extensive elaboration consists of three parts, each utilizing computer algebra, simplification, and symbolic manipulation of trees, just as we have been doing here.

- Recursively converting expressions (when possible) to CR form. For example, the polynomial x^3 with $x_0 = 0$ and $h = 1$ is represented by $f = \{0, +, 1, +, 6, +, 6\}$
- Direct evaluation of a given CR over a range following the recipe of multiplications and additions specified by the expression itself. Thus `(creval` f `5)` produces the list of values (0, 1, 8, 27, 64). The single general 10-line Lisp program `creval` in Appendix I can be used to interpret *any* CR, once initialized, by marching the recurrence in a loop. (This is done esentially by modifying the elements of the CR vector in place).
- Converting the CR evaluation of a particular CR to an open-coded C-language program. The program for a particular CR of fixed length and operations will still retain parameters such as h and x_0 or perhaps programs to compute the initial CR coefficients, which can be left to run-time. Such a program can be easily compiled. In our case we can use Lisp into assembler, or by generating C or Fortran code as an intermediary. This latter course can take advantage of the technology embedded in vendors' compilers for their own special hardware that might not otherwise be apparent in the direct compilation method.

As a special case, a CR compilation generated by `creval-s`, given the appropriate differencing scheme representing a polynomial, generates essentially the same evaluation schema illustrated above in our tabular generation programs.

For details beyond those in the our program (Appendix I), as well as acess to downloadable binary we refer the interested readers to programs provided by the references [4].

11 Related work

Numerous authors have explored the generation of code from computer algebra systems into Fortran, C, C++, Fortran 90, or other numerical systems for various purposes. See for example a recent survey by Prentice and Wester [14] or Borst et al [5] for additional references. Some related work by Lanam was done at Berkeley [11]. These code generators typically provide opportunities to automate the production of plug-in subroutines (e.g. for root-finders or quadrature), or symbolic derivatives of expressions for optimization, or special pre-processing (e.g. to generate calls to interval arithmetic subroutines) or to make use of template-based simplified programs. Most often these provide one-to-one translations of single (simplified) computer algebra expressions to one or perhaps a few Fortran assignment statements. While this can be useful, our current perspective is that we should aim for a different goal, one that goes beyond producing "mathematically correct" expressions to more carefully crafted code. This perspective is reflected more in other papers by Geddes [6] and by Kahan and Fateman [8].

Indeed, without taking into account more aspects of the programming environment, such code will seem inadequate because the numeric computation environment is subject to "non-mathematical" constraints: numerical roundoff and error accumulation, arithmetic pipe-line efficiency, memory conflict, subroutine linkages, etc. Thus our goal must extend to providing numerically sophisticated and highly efficient sections of code.

There are efforts to take advantage of the C++ template concept in code generation, most notably by Veldhuizen [16]. The advantage here is that any sufficiently compliant C++ system can take advantage of this style of coding. The disadvantage is the complexity of the code that must be written. Modern languages have complex syntax trees; analyzing them in an already-compromised static language is an unreasonable burden. It is as though we were forced to write our code generator programs without proper scope, without state, without floating-point numbers, and without the possibility of diagnostic messages.

Evaluating polynomials at n points which constitute n^{th} roots of unity (complex numbers or elements in a finite field) are equivalent to a Fast Fourier Transform (FFT), and can be done (subject to a variety of restrictions) using $O(n \log n)$ operations. There is a vast and still-growing literature on the FFT, much of which can be twisted into a polynomial evaluation framework. For example, one can construct sets of points other than roots of unity which are in principle amenable to this approach [1]. The practicality of these methods for a particular data set, while also scaling-down to modest degree polynomials or to use special arithmetic (exact integer arithmetic is one of our favorites) is in some doubt.

12 Conclusion

Our interest here has been in generating compact correct efficient code that goes beyond providing the commonplace for a computer algebra system (CAS). Any of the popular systems are able to map an expression into the syntax of some "mathematically similar" Fortran or C code segment. The code appearance, especially for very large or complex code segments, is no guarantee of computational equivalence under the rules of arithmetic used in compilers and implemented by hardware. Unfortunately, apparent mathematical syntactic equivalency does not always provide the needed correspondence.

It is our obligation to provide as much cleverness as possible in the generation of sequential (or parallel) arithmetic processing. We can encode in a CAS at least some of the knowledge necessary to generate more efficient or more accurate code than would be likely by a human programmer. There are a number of levels at which this can be done.

In this paper we have described software that will generate computationally appropriate ("correct"), efficient, and specific code for some basic computations. Our goal is to raise the level of abstraction in scientific programming task while improving the quality of the ultimate executable computation. We consider quality to be a measure of code speed and error control, as well as mathematical correctness. Unfortunately,

this cannot be done solely by program library re-use, but seems to benefit from a symbolic code-generation phase. A key to this is a kind of symbolic partial evaluation.

13 Acknowledgments

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

Appendix I

The code referred to in this paper is written in ANSI standard Common Lisp, and is available as the following files:

```
 Polynomial evaluation
www.cs.berkeley.edu/~fateman/papers/poly.cl
 Pipe/Stream evaluation and the newname facility
www.cs.berkeley.edu/~fateman/papers/pipe.cl
 Code related to J. Shewchuk's multi-term arithmetic
www.cs.berkeley.edu/~fateman/papers/jrs.cl
 Live-Dead analysis
www.cs.berkeley.edu/~fateman/papers/lda.cl
 Determinant calculation
www.cs.berkeley.edu/~fateman/papers/determ.cl
 A Simple simplifier for algebraic expressions
www.cs.berkeley.edu/~fateman/papers/simpsimp.cl
```

Appendix II

Local error analysis on sequences of arithmetic operations can provide a rigorous bound on the computations in this paper. If we can generate the additional sequence of operations to compute the bounds and interleave them with the computation (something that a human programmer would find tedious, if not prone to blunders), we can provide a different quality of program.

In this appendix we will provide a somewhat simplified justification for the error bounding techniques. Given that real numbers A and B are represented by the nearest exactly representable floating-point numbers a and b, and operations that result in the correctly rounded answer[3], we can look at the accumulation of error by introducing intervals around a and b and watching the intervals grow. For convenience in notation and discussion we will treat the intervals as a center and error. Thus $A = a \pm \Delta a$ where Δ is a quantity related to the arithmetic rounding ($\epsilon = 2^{-53}$ for IEEE 754 double precision floats). The Δ quantities are unknown, but for the initial values we assume they are bounded by $\pm\epsilon A$, reflecting a (nearly) correct conversion of a real to a float. In the case of the argument x of the polynomial, we generally assume it is a number as exactly represented, simplifying the formulas ($\Delta x = 0$).

We can propagate errors as follows. If we are interested in AB, how far off is the product ab as represented in the computer? Consider the errors in representation (or the propagated error to this point) and the fact that in rounding, the answer for ab may suffer a rounding error of $ab(1 + \epsilon)$. We therefore add in an uncertainty of ϵab to the interval computation for $r = AB$:

$$r + \epsilon\Delta r = (a \pm \Delta a) * (b \pm \Delta b) = ab \pm ab \pm a\Delta b \pm b\Delta a \pm a\Delta b \pm \Delta a\Delta b$$

[3]Actually, truncation or any systematic bounded amount of introduced error can fit this analysis

Therefore $|\Delta r|$ is bounded by $|ab| + |a\Delta b| + |b\Delta a| + |\Delta a \Delta b|$.

For addition, with a similar consideration of the rounding of $a + b$ we have Δr is bounded by $|a| + |b| + |\Delta a| + |\Delta b|$.

It can get more than a bit tiresome to write these expressions out in a C program, and so it is appealing to generate them.

In straight Horner's rule (`pn-s`), the error propagation is particularly simple since multiplication by (exactly) x requires less arithmetic to generate error bounds, and in fact about doubles the arithmetic necessary. In the monic Winograd algorithm we are first repeatedly squaring the evaluation point which we take to be the exactly presented x (That is, $\Delta x = 0$). The result after squaring and rounding is in the interval $x^2 \pm \epsilon(x^2)$. In general $|\Delta(x^{2^k})|$ is certainly bounded by $2^k x^{2^k}$. The subsequent arithmetic for error bounds requires some 3 or 4 times as much arithmetic as needed to compute the approximate value, and so drags the performance down from slightly more than half the cost for Horner's rule, to about the same. It is possible to consider doing the error arithmetic in lower precision (hence faster) arithmetic.

References

[1] , Aho AV. Steiglitz K. Ullman JD. "Evaluating polynomials at fixed sets of points." *SIAM Journal on Computing, vol.4, no.4* Dec. 1975, pp.533–539.

[2] `http://www.netlib.org/atlas/`

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

[4] Bachmann, O.; Wang, P.S.; Zima, E.V. Chains of recurrences-a method to expedite the evaluation of closed-form functions. in ISSAC'94. Proceedings of the International Symposium on Symbolic and Algebraic Computation, Oxford, UK, 20-22 July 1994. p. 242–249.

[5] Borst, W.N., Goldamn, V.V, and van Hulzen, J.A. GENTRAN 90: A REDUCE package for the generation of Fortran 90 code. in ISSAC'94. Proceedings of the International Symposium on Symbolic and Algebraic Computation, Oxford, UK, 20-22 July 1994. p. 45–51.

[6] Geddes, K. O. "Generating numerical ODE formulas via a symbolic calculus of divided differences," ACM SIGSAM Bull. vol 33, no. 2 June, 1999, 29–42

[7] Kahan, W.M. "Notes on Laguerre's Iteration" Dec., 1992.(unpublished).

[8] Kahan, W. and Fateman, R. "Symbolic computation of divided differences", ACM SIGSAM Bull. vol 33, no. 2 June, 1999, 7–22.

[9] Kislenkov, V.; Mitrofanov, V.; Zima, E. Multidimensional chains of recurrences. ISSAC 98. Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation, Proceedings of ISSAC '98. Rostock, Germany, 13-15 Aug. 1998. p.199-206.

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

[11] Douglas H. Lanam. An algebraic front-end for the production and use of numeric programs. ISSAC'83. Proceedings of the International Symposium on Symbolic and Algebraic Computation, Proceedings of International Symposium on Symbolic and Algebraic Computation, Snowbird Utah. p. 223-227.

[12] McNamee, J.M. A Bibliography on roots of polynomials, (section on Evaluation of polynomials and derivatives) contains over 200 references prior to 1994. `http://www.elsevier.co.jp/homepage/sac/cam/mcnamee/17.htm`

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

[14] Prentice, J.K. and Wester, Michael. "Code generation using computer algebra systems," Chapter 13 in Wester, Michael J. (ed), *Computer Algebra Systems A Practical Guide*, John Wiley and Sons, 1999.

[15] 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.

[16] Veldhuizen, Todd L. "C++ Templates as Partial Evaluation," 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99).