Squinting at Power Series

Rob Pike
(after Doug McIlroy)
Computing Sciences Research Center
Bell Labs
Lucent Technologies
rob@plan9.bell-labs.com

February 3, 2000

Background

A summary of this amazing paper:

M. Douglas McIlroy, ‘‘Squinting at Power Series’’,
Software−Practice and Experience,
July 1990, Vol. 20, No. 7, pp. 661-683.

Building on work by Kahn and MacQueen.

Power series are infinite, so standard call-by-value programming can’t handle them.

Idea: Channel can represent streams of data, and recurrence relations can translate directly into compositions of streams and processes.

Note: Our programming here is formal symbol manipulation; no attempt to guarantee convergence of series.

Power Series

A power series is a stream of coefficients, with exponents given implicitly by ordinal position. For example,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \cdots$$

is represented as a stream of rationals:

$$1,\ 1,\ \frac{1}{2},\ \frac{1}{6},\ \frac{1}{24}, \ldots$$

We will label power series with capital letters, e.g. F, and use functional form $F(x)$ when the variable must be named. F_i represents coefficient i:

$$F(x) = \sum_{i=0}^{\infty} F_i x^i$$

The representation of F as a channel will be in the program font: `F`. We will assume the coefficients of a series are rationals, defined by a type `rat`. Thus our representation of a stream is

```
F: chan of rat;
```

Rationals

Declarations of necessary pieces.

```
type rat: struct of {
    num: int;
    den: int;
};
type ps: chan of rat;

ratmk: prog(i: int, j:int) of rat; # make i/j
ratadd: prog(r:rat, s:rat) of rat; # r + s
ratsub: prog(r:rat, s:rat) of rat; # r - s
ratmul: prog(r:rat, s:rat) of rat; # r * s
ratdiv: prog(r:rat, s:rat) of rat; # r / s
ratprint: prog(r: rat);

psmk: prog() of ps; # make power series
```

Example: print a power series

```
psprint := prog(F: ps) {
    for(;;)
        ratprint(<-F);
};
```

Warmup

An exercise to get started. The sum of two series is by coefficient:

$$F(x)+G(x) = \sum_{i=0}^{\infty} (F_i+G_i)x^i$$

This translates easily into code:

```
# calculate power series S = F + G
do_psadd:= prog(F:ps, G:ps, S:ps) {
    for(;;)
        S <-= ratadd(<-F, <-G);
};
```

This gives us a process to do the work. But we want a *channel*, so we need to wrap up the loop:

```
# return a power series S = F + G
psadd:= prog(F:ps, G:ps) of ps {
    S:= psmk();
    begin prog() {
        for(;;)
            S <-= ratadd(<-F, <-G);
    }(); # spawn nameless prog, no args
    become S; # return S to caller
};

S := psadd(F, G); # to construct a sum
```

Recap

```
# return a power series S = F + G
psadd:= prog(F:ps, G:ps) of ps {
    S:= psmk();
    begin prog() {
        for(;;)
            S <-= ratadd(<-F, <-G);
    }(); # spawn nameless prog, no args
    become S; # return S to caller
};

S := psadd(F, G); # to construct a sum
psprint(S); # how to print the sum
psprint(psadd(F, G)); # shorter way
```

To add the two power series, we must create a new process to combine the coefficients and send the new result out a fresh channel.

The result must be a channel, not a process, so we wrap the copy loop in a function that creates the channel, starts the process, and returns the channel.

Differentiation

Is easy! Recall

$$F'(x) = \frac{d}{dx}F(x) = \frac{d}{dx}\sum_{i=0}^{\infty} F_i x^i = \sum_{i=1}^{\infty} i\ F_i x^{i-1}$$

Shifting the indices to get the right hand side in proper form,

$$F'(x) = \sum_{i=0}^{\infty} (i+1) F_{i+1} x^i .$$

We can write this as a *recurrence relation* that expresses the elements of the new series in terms of the old one:

$$F'_i = (i+1) F_{i+1}$$

Once we know the x^3 term of F, we can calculate the x^2 term of F', etc.

Differentiation: code

Translate the recurrence into code, elementwise:

```
psderiv:= prog(F:ps) of ps {
    D:= psmk();
    begin prog() {
        <-F;  # discard constant term
        n:= 1;
        for(;;) {
            f:= <-F;
            D <-= ratmk(n*f.num, f.den);
            n = n+1;
        }
    }();
    become D;
};
```

Note the shift in indices (from $i = 0$ to $i = 1$ in the sums, i to $i + 1$ in the recurrence) is accomplished by absorbing an element of the stream.

Integration is easy too, and involves emitting an extra element. Exercise for the reader.

An example

The power series for $\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$ is represented by a stream of ones, `1 1 1 1 ...`:

```
Ones:= psmk(); # the series for 1/(1−x)
begin prog() {
    one:= ratmk(1, 1);
    for(;;)
        Ones <-= one;
}();
```

Its derivative is regular:

```
psprint(psderiv(Ones));
```

produces `1 2 3 4 5 ...`; and given `psxmul` to multiply by `x` and `pscmul` to multiply by a constant,

```
psprint(psadd(Ones,
    psxmul(pscmul(ratmk(-1,1), Ones))));
```

prints `Ones` $+ x \cdot (-1) \cdot$ `Ones`, or `1 0 0 0 0 ...` .

But this cheats because it depends on the elements of `Ones` all having the same value; the stream gets scattered about. We need to control the splitting of streams.

Multiplication: A need for splitting streams

The product of two power series $P = F\ G$ is challenging. From the direct formula

$$\sum_{n=0}^{\infty} P_n x^n = \left[\sum_{i=0}^{\infty} F_i x^i\right]\ \left[\sum_{j=0}^{\infty} G_j x^j\right],$$

we can derive the familiar convolution for the product,

$$P_n = \sum_{i=0}^{n} F_i\ G_{n-i}, \tag{1}$$

Miserable to program. Must store n terms. Let's construct the recurrence. Write a stream F as a first term plus x times another series:

$$F = F_0 + x\,\overline{F}.$$

The tail $\overline{F}$ is a power series beginning with constant, so recur:

$$P = F\ G = P_0 + x\,\overline{P}$$

$$= F_0\ G_0 + x\,(F_0\ \overline{G} + G_0\ \overline{F}) + x^2\ \overline{F}\ \overline{G}. \tag{2}$$

Equate coefficients and we get first term

$$P_0 = F_0\ G_0,$$

followed by tail

$$\overline{P} = F_0\ \overline{G} + G_0\ \overline{F} + x\,\overline{F}\ \overline{G}.$$

Much simpler to program than the convolution (1).

Multiplication: A need for splitting (2)

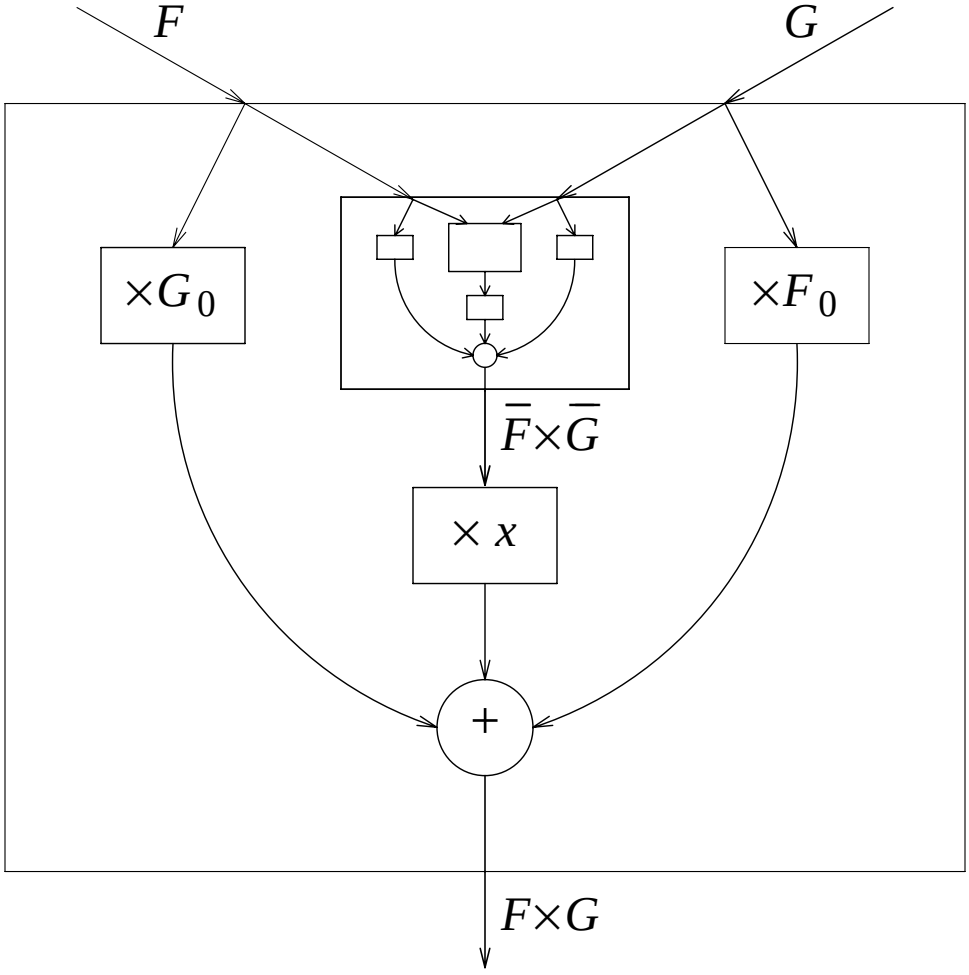
It's now easy to write the code except for a small problem. Note that the tail of the recurrence

$$\overline{P} = F_0\ \overline{G} + G_0\ \overline{F} + x\ \overline{F}\ \overline{G}$$

has $\overline{F}$ and $\overline{G}$ twice, so we must *split* the streams for $\overline{F}$ and $\overline{G}$ so each coefficient can be processed twice.

This is easy to see if we represent the recurrence by a Figure.

Recursive data flow for multiplication



The outer box represents the process for the tail of $F\ G$, equation (2). The similar inner box does not receive the first terms of F or G, and its output does not enter into the first two terms of the product $F\ G$. The code will use a separate process for each box, and one or more processes for each stream splitting.

Splitting

A program to split a stream must read a coefficient and deliver it twice. But, the two values may be needed at different times, so we must be ready for either order:

```
# split power series F into F0 and F1
rec do_split:= prog(F:ps, F0:ps, F1:ps) {
    f:= <-F;
    H:= psmk(); # the held branch
    select {
    case F0 <-= f:
        begin do_split(F, F0, H);
        F1 <-= f;
        become copy(H, F1);
    case F1 <-= f:
        # same, with F0 and F1 interchanged
    }
};
```

`Copy` is a trivial helper routine:

```
# calculate power series C = F
copy:= prog(F:ps, C:ps) {
    for(;;)
        C <-= <-F;
};
```

Splitting (2)

Finally we encapsulate to produce the stream pair:

```
# a pair of power series
type pspair: array[2] of ps;
pspairmk: prog(F: ps, G: ps) of pspair;

# return a pair of copies of F
# (consuming F)
split:= prog(F:ps) of pspair {
    FF:= pspairmk(psmk(), psmk());
    begin do_split(F, FF[0], FF[1]);
    become FF;
};
```

Note we use a process to buffer each term of the partial series until it's needed; each split generates another process to feed the new branch.

There is a problem with this structure, which we'll return to, but it works well enough to write our multiply code.

Multiply, at last

Now we have the tools. Here is the recurrence (2) again, as a reminder:

$$P = F\,G = P_0 + x\,\overline{P} = F_0\,G_0 + x\,(F_0\,\overline{G} + G_0\,\overline{F}) + x^2\,\overline{F}\,\overline{G}.$$

And here is the code:

```
# return power series F*G
rec psmul:= prog(F:ps, G:ps) of ps {
    P:= psmk();
    begin prog(){
        f := <-F;
        g := <-G;
        P <-= ratmul(f, g);
        FF := split(F);
        GG := split(G);
        fG := pscmul(f, GG[0]);
        gF := pscmul(g, FF[0]);
        xFG := psxmul(psmul(FF[1], GG[1]));
        for(;;)
            P <-= ratadd(ratadd(<-fG, <-gF),
                                <-xFG);
    }();
    become P;
};
```

Some fun

A similar combination of recurrence relations and stream processing can give us other tools, like substitution ($S = F(G(x))$), reversion (R such that $F(R(x)) = x$), integration, exponentiation (very clever), etc.

Example: To compute the power series for $\tan(x)$, note that

$$\frac{d}{dx}\arctan(x) = \frac{1}{1 + x^2}.$$

Build a monomial substitution operator, `psmsubst(F, c, n)`, that generates $F(cx^n)$. It multiplies each input coefficient F_i by c^i and copies it to the output followed by $n-1$ zeros. Substitute $-x^2$ into `Ones` (i.e. into $1/(1-x)$) to get $1/(1+x^2)$, integrate to get the arctangent, and revert. The resulting code,

```
psmsubst : prog(ps, rat, int) of ps;
psrev : prog(ps) of ps; # reversion

Tan:= psrev(psinteg(psmsubst(Ones,
    ratmk(-1, 1), 2), 0));
psprint(Tan);
```

prints the coefficients of the tangent series:

```
0 1 0 1/3 0 2/15 0 17/315 0 62/2835
  0 1382/155925 ...
```

a difficult calculation by traditional means.

A minor problem

There was a problem we skipped during the implementation of `split`. It is that the program is a *runaway*: it prepares to satisfy requests for coefficients before they are required. Since it creates processes to do this, it can get very far in front of the calculation, consuming resources to calculate values that will never be used.

This sort of performance problem is peculiar to concurrent programming, but it's easy to fix. We just need to change the definition of a power series into a *pair* of channels, one for control and one for data. When a value is needed, it is requested using the control channel, and the answer read on the data channel. The code is simple, and is in the paper.

This is strongly analogous with lazy evaluation.

For these and other topics, a full explanation may be found in McIlroy's paper.