Chains of functional primitives

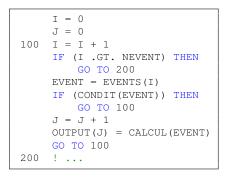
Jim Pivarski

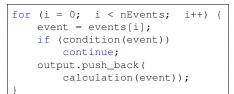
Princeton University - DIANA

May 23, 2016



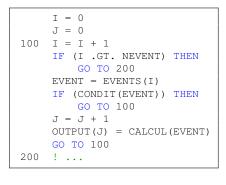


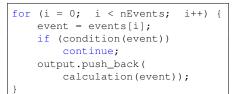




val output = events.filter(condition).map(calculation)







val output = events.filter(condition).map(calculation)

"condition" and "calculation" are user-defined functions passed as arguments to "filter" and "map."



"Go to" statements allowed extreme flexibility in program flow, usually too much, adding unwanted complexity.



"Go to" statements allowed extreme flexibility in program flow, usually too much, adding unwanted complexity.

Flow control statements (if, for) also provide more power than is often needed.

```
for (i = 0; i < nEvents; i++) {
    event = events[i];
    if (condition(event))
        continue;
    output.push_back(
        calculation(event));
}</pre>
```

```
val output = events.
    filter(condition).
    map(calculation)
```



"Go to" statements allowed extreme flexibility in program flow, usually too much, adding unwanted complexity.

Flow control statements (if, for) also provide more power than is often needed.

<pre>for (i = 0; i < nEvents; i++) {</pre>
<pre>event = events[i];</pre>
if (condition(event))
continue;
output.push_back(
calculation(event));
}

```
val output = events.
    filter(condition).
    map(calculation)
```

The map/filter functional chain says less than the for loop.

"Step through the events *in order*, skip if the condition is met, and *incrementally* grow the output list." "Remove events for which the condition holds and apply the calculation to the remainder."



```
for (i = 0; i < nEvents; i++) {
    event = events[i];
    if (condition(event))
        continue;
    output.push_back(
        calculation(event));
</pre>
```

```
val output = events.
    filter(condition).
    map(calculation)
```

The for loop body could have operated on multiple events at once, but didn't in this case. The map functional cannot ever. Compilers and runtime environments can take advantage of this knowledge for vectorization or parallelization.



```
for (i = 0; i < nEvents; i++) {
    event = events[i];
    if (condition(event))
        continue;
    output.push_back(
        calculation(event));
</pre>
```

```
val output = events.
    filter(condition).
    map(calculation)
```

- The for loop body could have operated on multiple events at once, but didn't in this case. The map functional cannot ever. Compilers and runtime environments can take advantage of this knowledge for vectorization or parallelization.
- Left: output.push_back doesn't know how large the output can be and has to dynamically allocate. Right: output.size <= events.size; allocate and trim.</p>



```
for (i = 0; i < nEvents; i++) {
    event = events[i];
    if (condition(event))
        continue;
    output.push_back(
        calculation(event));
</pre>
```

```
val output = events.
    filter(condition).
    map(calculation)
```

- The for loop body could have operated on multiple events at once, but didn't in this case. The map functional cannot ever. Compilers and runtime environments can take advantage of this knowledge for vectorization or parallelization.
- Left: output.push_back doesn't know how large the output can be and has to dynamically allocate. Right: output.size <= events.size; allocate and trim.</p>
- To repartition the for loop, the user must be involved in the index arithmetic; the functionals are more abstract.



val output = events.filter(condition).map(calculation)

could mean

- Generate inline code for condition and calculation and vectorize the calculation.
- Evaluate them in a thread execution pool.
- Launch parallel jobs on a worldwide grid.
- Construct a CUDA kernel and pass condition and calculation to the GPU.
- Construct an intermediate list after filter and before map.
- Lazy-evaluate the filter, treating it like a Python iterator (no intermediate list).

Although these choices have significant performance consequences, they are secondary to the intention expressed in that line of code.



This style is fairly common among data analysts:

- R code is full of apply/lapply/tapply, and the R users I know try to avoid for loops whenever possible.
- "Map-reduce" launched an industry around Hadoop, and functional chains are the central paradigm of Spark.
- ► Functional primitives are hidden in the SELECT, WHERE, and GROUP BY, clauses of SQL.
- LINQ, the data extraction sublanguage of .NET, is heavily functional.
- d3, a popular visualization library for Javascript, also manipulates data with functional chains.

Although it restricts flexibility, this paradigm seems to fit data analysis well.



Switching to this paradigm requires the user to become familiar with some functional primitives.

Adopt the "there's an app for that" mentality.



	input	function	output	operation	
map	table of <i>A</i>	$f: A \rightarrow B$	table of <i>B</i>	apply <i>f</i> to each row <i>A</i> , get a table of the same number of rows <i>B</i>	
	a.k.a. "lapply" (R), "SELECT" (SQL), list comprehension (Python)				
filter	table of <i>A</i>	$f: A \rightarrow$ boolean	table of <i>A</i>	get a shorter table with the same type of rows	
	a.k.a. sin	gle brackets (R),	"WHERE"	(SQL), list comprehension (Python)	
flatMap	table of <i>A</i>	$f: A \to \\ table of B$	table of <i>B</i>	compose map and flatten, get a table of any length	
	a.k.a. "map" (Hadoop), "EXPLODE" (SQL), >>= (Haskell)				



	input	function(s)	output	operation
reduce	table of A	f:(A,A) o A	single value <i>A</i>	apply <i>f</i> to the running sum and one more element
aggregate	table of <i>A</i> , initial value <i>B</i> ("zero")	$f: (A, B) \rightarrow B$ $f: (B, B) \rightarrow B$ (increment and combine)	single value B	accumulate a counter with a different data type from the input
aggregate by key	table of $\langle K, A \rangle$, initial value <i>B</i>	$f:(A,B) \to B$ $f:(B,B) \to B$	pairs $\langle K, B \rangle$	aggregate independently for each key

a.k.a. "reduce" (Hadoop), "GROUP BY" (SQL)



More exotic functionals can handle specific cases.

For instance,

- collection.skip(n) to offset a collection by n
- > zip(collections*) to walk through collections in lock-step

can be combined to compare an event with the previous event:

zip(events, events.skip(1)).map(operation_on_pairs)

Or perform nested loops (SQL JOIN):

- cartesian to loop over all pairs i, j of a collection
- triangular to loop over pairs $i, j \ge i$ of a collection

Different functional names because the user thinks of them differently; each would have to be optimized differently, anyway.



This is very similar to what I'm doing with Histogrammar (https://github.com/diana-hep/histogrammar), which introduces a dozen functional primitives that are all variations on aggregate.

```
histogram = Bin(100, 0, 20, fill_rule, Count())
```

```
hist2d = Bin(binsX, lowX, highX, fillX,
Bin(binsY, lowY, highY, fillY, Count()))
```

```
profile = Bin(binsX, lowX, highX, fillX, Deviate(fillY))
```

```
box_whiskers = Bin(binsX, lowX, highX, fillX, Branch(
Minimize(fillY), Quantile(0.25, fillY), Quantile(0.5, fillY),
Quantile(0.75, fillY), Maximize(fillY)))
```

```
unknown_support = SparselyBin(binWidth, fillX, Count())
efficiency = Fraction(cut, Bin(100, 0, 20, fill_rule, Count()))
```

```
where all fill_rules are lambda functions.
```



To be fluent, one needs a good syntax for lambdas.

C++	<pre>[](Datum d){return sqrt(d.px*d.px + d.py*d.py);}</pre>
Scala	<pre>{d: Datum => Math.sqrt(d.px*d.px + d.py*d.py) }</pre>
Python	<pre>lambda d: math.sqrt(d.px**2 + d.py**2)</pre>
R	function (d) { $sqrt(d.px^2 + d.py^2)$ }

Lambda functions



Unlike all the rest, Python lambdas are fundamentally limited to one-line expressions (no statements, such as local assignment).

I have some code (written in Python) that extends Python's grammar to include multiline assignments like this:

```
lambda d: sqrt(d.px**2 + d.py**2)
def(d -> sqrt(d.px**2 + d.py**2))
```

or even

```
def(x -> y = sqrt(x),
    z = 2*y if y < 1,
        3*y if y < 2,
        4*y else,
        z**2)
```

which might be useful for extended functionals in Python.

. . .



I also have a suite of functional primitives with naïve implementations in the attached functional_chains.py.

19 / 20



...with comparisons to TTree.Draw strings in PyROOT.