Stream processing
with purely-functional programming

UKSystems ‘19

Jonathan Dowland
<jon.dowland@ncl.ac.uk>

Supervisor: Paul Watson

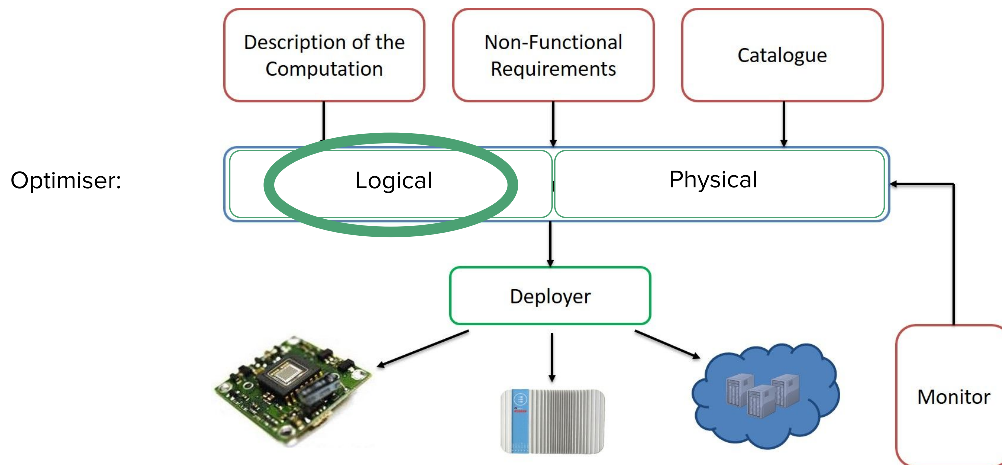
Part-time PhD student

I’ve been working for 18 months of real-time, so I’m about ⅔ through my first year in virtual time.

We’ve been building a stream-processing system called “StrIoT” as a base for exploring the question

“To what extent are the properties of purely-functional programming advantageous in the design and engineering of a stream-processing system”?

For this talk I’m going to give a brief overview of StrIoT, followed by a short demonstration of it in action, but the majority of the time I will spend on my particular area of focus within the architecture.



Original architectural plan
Your stream processing program ("description of the computation")
Using a restricted set of 8 functional operators provided by StrIoT (we'll see these as we go)

Example non-functional requirements:
Battery life of a field sensor
Minimize network traffic due to cost of network links

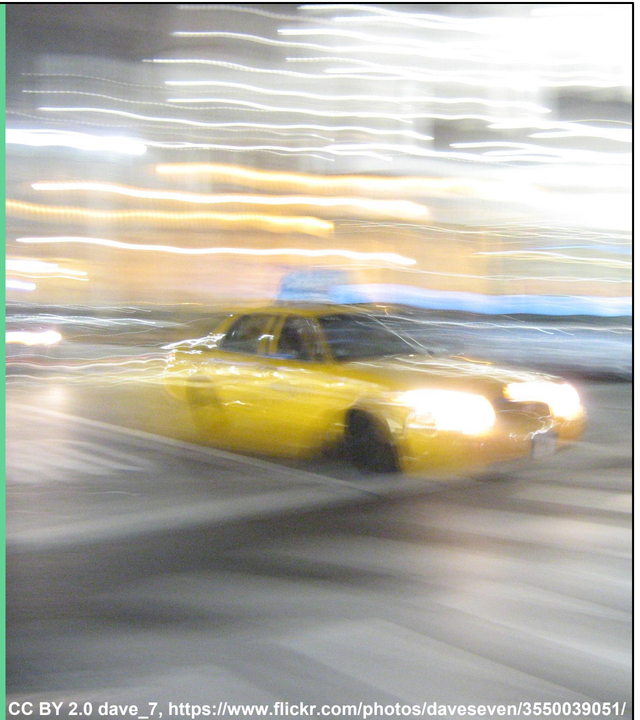
The heart of the system is the Optimiser, where the stream-processing program you provided may be rewritten to better meet the non-functional requirements.

Partitioned and deployed to heterogeneous IoT environment (described by Catalogue)

Run time performance data could be collected and fed back into the optimiser to iteratively improve the deployment

Live Demo

Profitable Taxi Journeys



A solution to the grand challenge at the 2015 DEBS conference (distributed event-based systems)
“10 most frequent routes during the last 30 minutes”
To demonstrate: usage of stream operators; stream-processing definition; partitioning, deployment, running in docker-compose
(remember to activate docker compose virtual environment)

Rewriting rules for logical optimisation

streamFilter	F	x^1	F	x^1	×	x^4	x^5	8
streamMap	9	F	11	12	13	x^4	15	16
streamFilterAcc	F	x^1	F	x^1	×	x^4	×	x^8
streamScan	x^7	x^9	x^7	x^6	×	x^4	31	
streamWindow	×	×			×	x^3	×	40
streamExpand	41	42	×	44	45	46	×	48
streamJoin	x^2	x^2	x^2	x^2	x^2	x^2	x^2	x^2
streamMerge	57	58	x^8	x^8	×	62	×	F

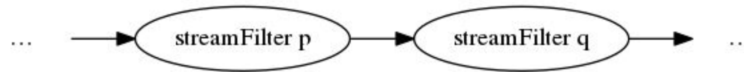
Claessen, Koen & Hughes, John (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00). ACM, New York, NY, USA, 268–279. DOI:http://dx.doi.org/10.1145/351240.351266

In order to explore the semantics of the 8 operators, as well as to synthesise some potentially useful rewrite rules, we performed a systematic pairwise comparison of 8 the operators.
for each pair, could I think of something to do with them: fuse, eliminate, swap?

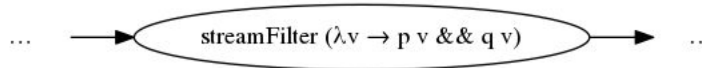
used QuickCheck to add some assurance that the rules hold (helped find some examples where re-ordering occurs that I had missed)

Yielded 24 rules

Fusion




```
streamFilter q . streamFilter p
= streamFilter (λv → p v && q v)
```



Hirzel, M., Soulé, R., Schneider, S., Gedik, B., & Grimm, R. (2014). A catalog of stream processing optimizations. DOI: 10.1145/2528412

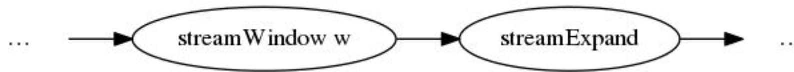
A simple example

Using equational reasoning, which is one of the powers we have thanks to pure FP

Reading the LHS right-to-left, filter p comes first, then filter q (function composition)
Possible with maps too (about 6 examples in my list)

Visual representation of the before-and-after graphs on each example slide

Elimination?



streamExpand . streamWindow w

= id

Hirzel, M., Soulé, R., Schneider, S., Gedik, B., & Grimm, R. (2014). A catalog of stream processing optimizations. DOI: 10.1145/2528412

On the face of it this looks like a viable rule, and it is in terms of the payload of the stream, but it is not “total”, because we lose stream metadata

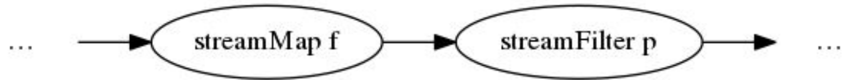
Consider a window function that batched input events into fixed lists of 5

5 input events means 5 sets of metadata: event ID, timestamps
One output event of 5 payloads in a list: which timestamp do we keep?

StreamExpand cannot recreate those timestamps

And yet this might be a useful rule. The user may not care about the metadata. And it turns out there are other non-total rules that might be effective if we know that the data we lose is not important.

Is this rule useful?




```
streamFilter p . streamMap f
= streamMap f . streamFilter (p . f)
```

Wadler, Philip (1990). "Deforestation: transforming programs to eliminate trees". Theoretical Computer Science. 73 (2): 231–248. doi:10.1016/0304-3975(90)90147-A.

For each accepted event, the former evaluates p and f once each

The latter, f twice

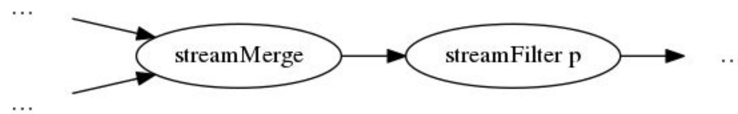
If p is highly selective, then moving the filter first will reduce the amount of list deconstruction and rebuilding that occurs beween the operators (hidden bookkeeping cost)

(the idea of removing this altogether is “deforestation”, explored by Wadler in a 1990 paper, implemented in GHC in the form of rewrite rules)

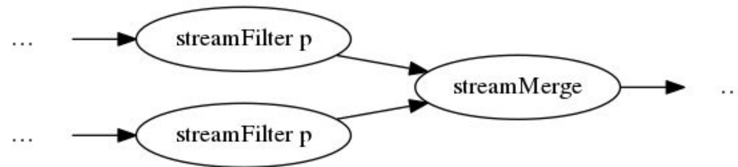
The reduction in bookkeeping cost could be larger than the cost of evaluating f twice for accepted events

We don’t know without an idea of the selectivity of p (could the user signal that to us in some way?)

filter/merge?




```
streamFilter p (streamMerge [s1,s2…])
  = streamMerge [ streamFilter p s1,
                  streamFilter p s2, … ]
```

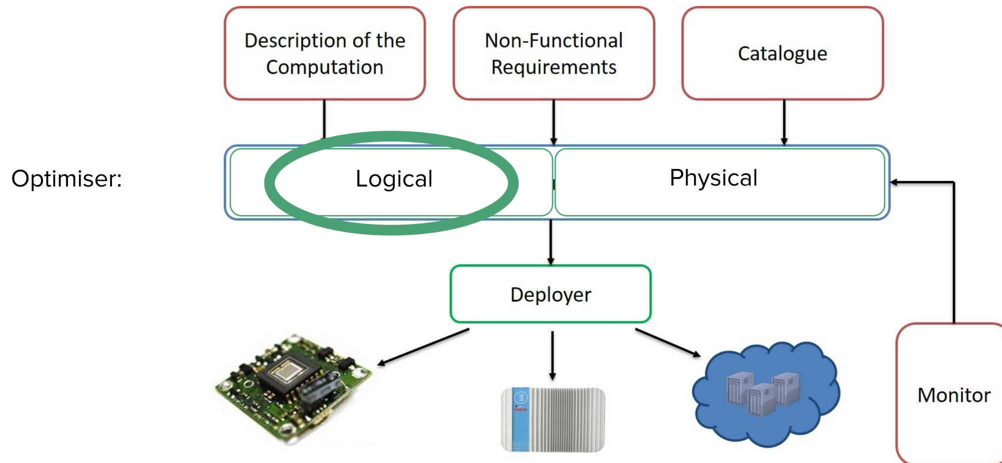


This looks like it would be practically useful: move filtering nearer to the source, prevent unneeded data from being propagated over network links

Unfortunately this is not generally applicable because it does not preserve order (reordering is bounded by the size of the input list to streamMerge, in this case 2)

If we knew that this doesn't matter, then we could use it anyway

Or we could construct some scaffold around this to re-order after the map



To recap my current area of focus is on logical optimisation

I'm going to talk about an aspect of the design of this, the encoding of rules

Graph encoding

https://github.com/snowleopard/alga

```
Four constructors:

        Empty              Overlay (Graph α) (Graph α)
        Vertex α           Connect (Graph α) (Graph α)
```

```
Overlay (Connect (Vertex 1) (Vertex 2))
        (Connect (Vertex 2) (Vertex 3))
```

Mokhov, Andrey (2017). Algebraic Graphs with Class (Functional Pearl). In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017). ACM, New York, NY, USA, 2–13. DOI:http://dx.doi.org/10.1145/3122955.3122956

A brief aside on how graphs are represented in the Optimiser
We use a 3rd party graph library "alga" which provides an algebra for graphs authored by Andrey Mokhov, Semi-coincidentally from Engineering, Newcastle University

The use of an algebra was attractive due to the prospect of reasoning over graphs

Four grammar rules: 2 terminal, 2 production; one rule only introduces edges (connect)

It is not possible to construct "invalid" graphs (for some classes of invalid: e.g. edges that go nowhere) using these rules

Example of a simple path encoded using these constructors

Notice how "Vertex 2" appears in two locations in the example

Rule encoding

```
streamFilter q . streamFilter p = streamFilter (λv → p v && q v)
```

```
filterFuse :: StreamGraph -> Maybe (StreamGraph -> StreamGraph)

filterFuse (Connect (Vertex a@(StreamVertex i Filter (p:_) ty _))
                (Vertex b@(StreamVertex _ Filter (q:_) _ _))) =

  let c = StreamVertex i Filter ["λp q x → p x && q x", p, q] ty ty
  in  Just (removeEdge c c . mergeVertices (`elem` [a,b]) c)

filterFuse _ = Nothing
```

Filter fusion example from earlier: in a pseudo-haskell encoding in the green area of the slide

We could build an optimiser where the rules were encoded within strings in such a way, and we build a parser for them. (The approach used by a “calculator” in the latest Richard Bird book, that I have considered)

We would need to support a subset of Haskell; the less we support, the harder/more awkward it is to encode the rules; the more we support, the more complex a parser we need. We don’t want to write a full-blown Haskell parser.

An alternative approach being considered: rewrite rules are simple functions

The left-hand side is encoded using pattern matching against the Graph constructors from the previous slide

In the event of a match, we cannot return a graph to be spliced into the parent stream graph by the applicator-function: because elements of the graph (such as a given vertex) might occur outside of that sub-graph (as in the previous slide example, where Vertex 2 occurred twice in the algebraic expression)

Instead return a function that takes a graph argument - the global streamgraph - and can perform a transformation (such as a global substitution)

In the event that the rewrite rule is not applicable (in this case the pattern does not match), return Maybe type Nothing (success return value is wrapped in Maybe type Just)

Further work

Minimal Viable PoC for first paper

Finish design + build the Optimiser

Encode NFRs

Rewrites for other purposes (performance monitoring)

Varying the operators

Runtime reconfiguration

Questions

Thank you!

Jonathan Dowland
<jon.dowland@ncl.ac.uk>

Discussion around the rule encoding: there's no canonical representation of a given graph in the algebra (e.g. any element of a graph could have "Overlay empty" adjacent to it, or "Overlay g g" equivalent to just g), so pattern matching in general might not apply where it "should", but for the restricted case of graph we are considering here, I have proven (to my satisfaction) that we can "normalize" a graph such that a pattern will match when it "should"

Pattern matching also looks less pleasant/legible once covered in decorators/unpacked list constructors

(pattern matching an optional feature here; a rewrite rule could be written to StreamGraph -> Maybe (StreamGraph -> StreamGraph) and not use pattern matching at all)

In danger of designing a system around simple 2-node rules and not considering the needs of more complex rules that might be more useful (such as: introducing windowing over a large sub-graph)

Q: if you could go back to the beginning and start again, would you still use Haskell?
A: yes.