RatPak

Ian Daniher - PLDI - 04/29

What?

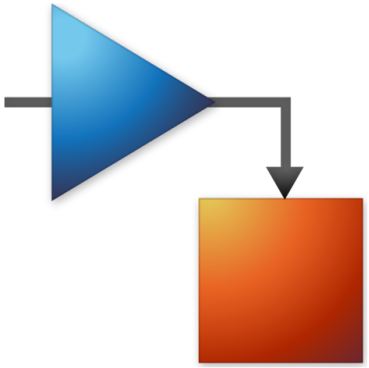
Statically typed declarative language for stream processing.

Compiles a textual definition of a process network flowgraph to Rust.

Why?


GNURadio
THE FREE & OPEN SOFTWARE RADIO ECOSYSTEM


NATIONAL INSTRUMENTS
LabVIEW



Why?

Fat, expensive, ugly.

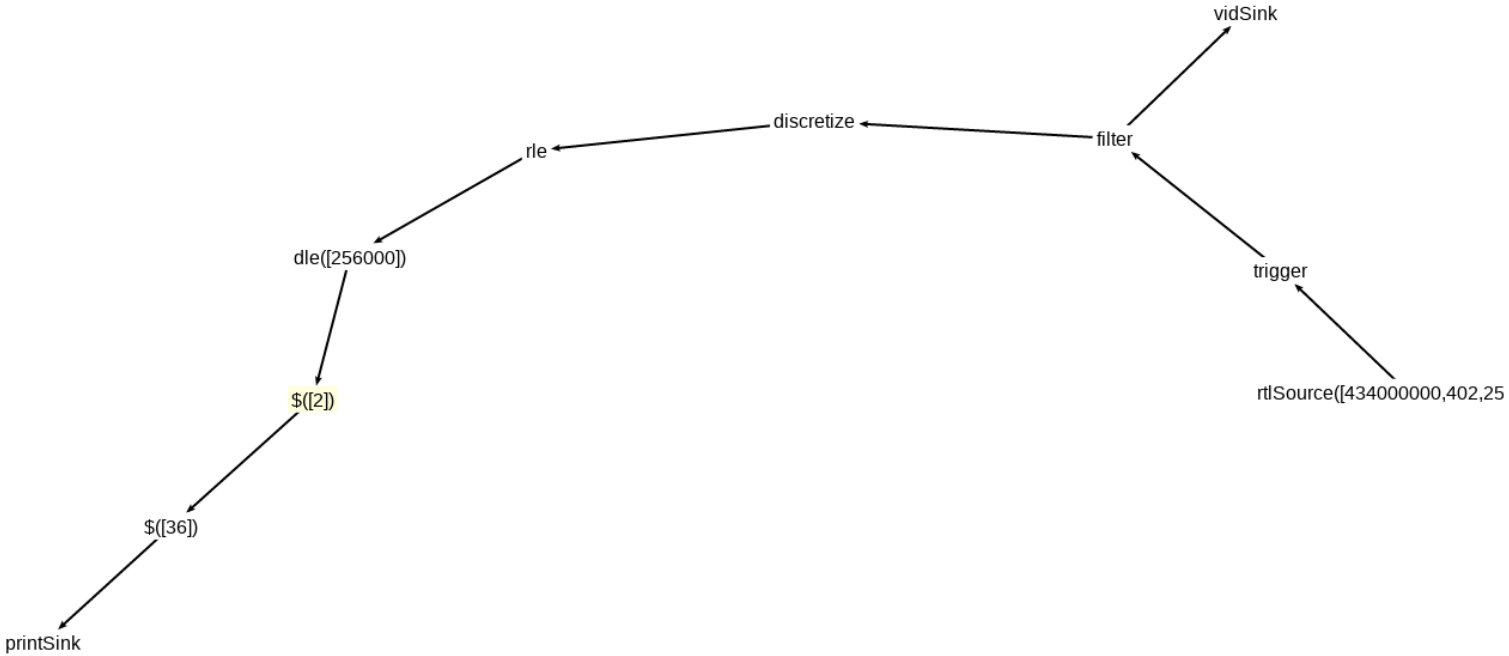
Check it out!

(demo time)

Demo

```
main
     rtlSource (434e6, 402, 256e3)
     ^ (*1) *2
     %
     dle 256e3
     $2
     $36
     printSink
```

Demo


vidSink
discretize
filter
rle
dle([256000])
trigger
$([2])
rtlSource([434000000,402,25
$([36])
printSink

How?

Stage 1: Parser / Lexer / Graph Serialization

Stage 2: Graph manipulation, rust codegen

Stage 3: Rust executable!

Stage 1

Prototyped in CoffeeScript.

Uses pegjs rule set to break into lines, each of one or more expressions.

Parses line set to nodes and edges.

JSON output.

Stage 1

```
57 args =
58 ·   " "? "(" a:args ")" " "? {a}
59 ·   / " "? v:values " "? {if typeof(v) != 'string' then ((if x.length == 1 then x[0] else x) for x in v
60
61 modif =
62 ·   " "? op:"^" " " {op}
63 ·   / op:"/" {op}
64
65 proc =
66 ·   " "? "(" " "? p:proc " "? ")" " "? {p}
67 ·   / n: argless { @row += 1;{"proc": n, "pos":{"y":line() - @start, "x":@row} }}
68 ·   / n: name r: ref+ { @row += 1;{"proc":n, "refs": r, "pos":{"y":line() - @start, "x":@row}} }
69 ·   / n: name a: args { @row += 1;{"proc": n, "args": a, "pos":{"y":line() - @start, "x":@row}} }
70 ·   / n: name { @row += 1;{"proc": n, "pos":{"y":line() - @start, "x":@row} }}
71
72 expr =
73 ·   " "? "(" " "? e:expr " "? ")" " "? {e}
74 ·   / p: proc {p}
75 ·   / d: modif  rp:(proc " "?)+ {
76 ·   ·   if d != "/"
77 ·   ·   ·   @_.union rp[0][0], (@_.extend(p[0], {"modif":d}) for p in rp when (rp.indexOf p) > 0)
78 ·   ·   else
79 ·   ·   ·   (@_.extend(p[0], {"modif":d}) for p in rp)
80 ·   }
```

Stage 2

JSON decoded to Rust Struct.

Rust Struct to Rust AST.

Boilerplate and AST serialization dumped.

```
15 #[deriving(Decodable,Encodable, Clone)]
16 struct Node {
17 ·   pname: ~str,
18 ·   oct: uint,
19 ·   ict: uint
20 }
21
22 #[deriving(Decodable,Encodable)]
23 struct Graph {
24 ·   edges: Vec<Vec<~str>>,
25 ·   nodes: Vec<(~str, Node)>,
26 ·   name: ~str,
27 ·   consts: Option<Vec<~str>>,
28 ·   inrx: bool,
29 ·   outtx: bool,
30 }
```

Stage 2


```
{
  "edges": [
    [
      "001001",
      "002001"
    ],
    [
      "002001",
      "003001"
    ],
    [
      "003001",
      "004001"
    ],
    [
      "003001",
```

```
"nodes": [
    [
      "001001",
      {
        "pname": "rtlSource",
        "label": "rtlSource([434000000,402,256000])",
        "args": [
          434000000,
          402,
          256000
        ],
        "ict": 0,
        "oct": 1
      }
    ],
```

```
"name": "main",
  "inrx": false,
  "outtx": false,
  "consts": null
}
```

Stage 3

Boilerplate implementing primitives added.

Single “fn main () {}” block containing statements defining endpoints and expressions spawning tasks.

Stage 3

Boilerplate defines ratpak primitives.

^ * + $ % ? Z b

```
let n = if nodepname.slice_from(0) == "*" { "mulAcross".to_str() }
·   else if nodepname.slice_from(0) == "+" {"sumAcross".to_str()}
·   else if nodepname.slice_from(0) == "Z" {"delay".to_str()}
·   else if nodepname.slice_from(0) == "%" {"grapes".to_str()}
·   else if nodepname.slice_from(0) == "b" {"binconv".to_str()}
·   else if nodepname.slice_from(0) == "$" {"shaper".to_str()}
·   else if nodepname.slice_from(0) == "?" {"matcher".to_str()}
·   else { nodepname.clone() };
```

Stage 3

mulAcross and sumAcross

```
25 pub fn mulAcross<T: Float+Send>(u: ~[Receiver<T>], v: Sender<T>, c: T) {
26 ·   loop {
27 ·   ·   v.send(u.iter().map(|y| y.recv()).fold(c, |b, a| b*a))
28 ·   }
29 }
30
31 pub fn sumAcross<T: Float+Send>(u: ~[Receiver<T>], v: Sender<T>, c: T) {
32 ·   loop {
33 ·   ·   v.send(u.iter().map(|y| y.recv()).fold(c, |b, a| b+a))
34 ·   }
35 }
```

Stage 3

```
16 pub fn fork<T: Clone+Send>(u: Receiver<T>, v: ~[Sender<T>]) {
17 ·   loop {
18 ·   ·   let x = u.recv();
19 ·   ·   for y in v.iter() {
20 ·   ·   ·   y.send(x.clone());
21 ·   ·   }
22 ·   }
23 }
```

```
54 pub fn Z<T: Send+Clone>(u: Receiver<T>, v: Sender<T>) {
55 ·   let x = u.recv();
56 ·   v.send(x.clone());
57 ·   v.send(x.clone());
58 ·   loop {
59 ·   ·   v.send(u.recv());
60 ·   }
61 }
```

Stage 4?

Rewriting Stage 1 in Rust, using Rust-PEG

Implement more complicated primitives as macros - esp “?” aka “matcher”

Stage 4?

Better IDE / IDExperience

Web UI?

Good format for graph layout & entry?

SPICE Integration

Closed-Loop Control Demos