RHEA

A Reactive, Heterogeneous, Extensible, and Abstract Framework for Dataflow Programming

Orestis Melkonian, Angelos Charalambidis

October 29, 2018

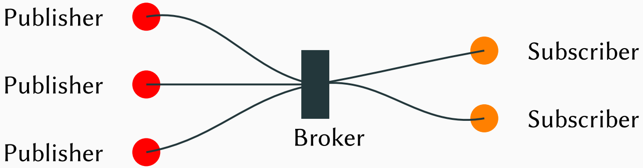
Utrecht University, National Center for Scientific Research ”Demokritos”

Utrechts Universiteitsfonds

Universiteit Utrecht

Robot Operating System (ROS)

- Most popular middleware for robotic applications
- Provides a Publish-Subscribe messaging platform

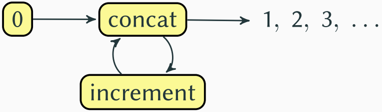


ROS Code

```
bool scanReceived = FALSE, imageReceived = TRUE;
LaserScan scan; Image image;
subscribe<LaserScan>("/scan", scanCallback);
subscribe<Image>("/camera/rgb", imageCallback);
// Main ROS loop
while (ros::ok()) {
    if (scanReceived && imageReceived) {
      window.show(embedLaser(scan, image));
      scanReceived = FALSE;  imageReceived = FALSE;
    }
    ros::spinOnce();
}
// Callback for topic "/scan"
void scanCallback(LaserScan newScan) {
  if (!scanReceived) {
    scan = newScan;
    scanReceived = TRUE;
  }
}
// Callback for topic "/camera/rgb"
void imageCallback (Image newImage) {
  if (!imageReceived) {
    image = new Image(newImage);
    imageReceived = TRUE;
  }
}
// OpenCV stuff...
Mat embedLaser(LaserScan scan, Image image) { ... }
```

Dataflow computational model

- Completely decentralized
    - Independent nodes communicating with each other
- No control-flow
- Implicit concurrency

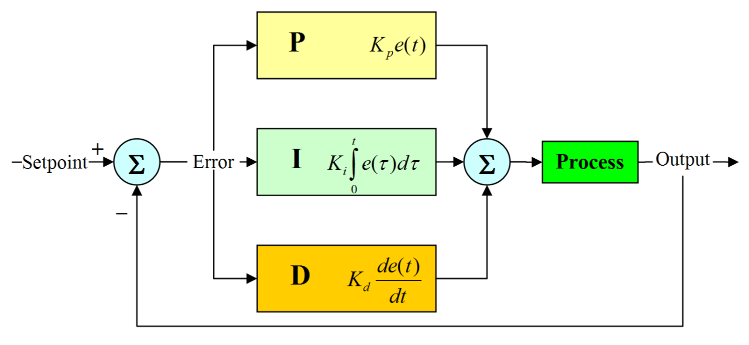


Motivation: Other Domains

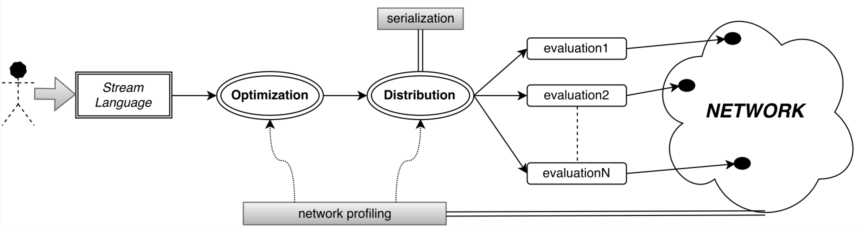
- Big Data
    - Apache Flink
    - Map-Reduce
    - RX framework
- Interactive Systems
    - UIs (ReactJS)
    - Games (Yampa)
- Neural Networks (TensorFlow)

Motivation: Robotics

- Robot Perception Architecture (RPA)
- Many dataflow examples in control theory



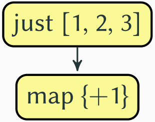
RHEA



- Dataflow framework for the JVM
- Current frontends only in Java & Scala
- Set of libraries in github.com/rhea-flow

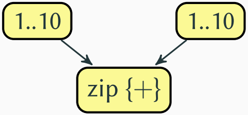
Stream Language: Sources and single-input nodes

```
Stream.just(1, 2, 3)
      .map(x -> x + 1);
```



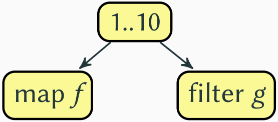
Stream Language: Multiple-input nodes

```
Stream.zip(
  Stream.range(1, 10),
  Stream.range(1, 10),
  (x, y) -> x + y
);
```



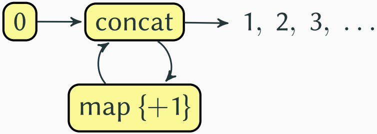
Stream Language: Split

```
Stream<Int> st =
  Stream.range(1, 10);
st.map(f)
st.filter(g)
```



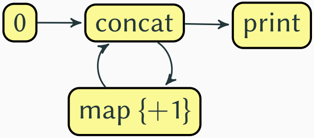
Stream Language: Cycle

```
Stream
.just(0)
.loop(s -> s.map(i -> i + 1));
```

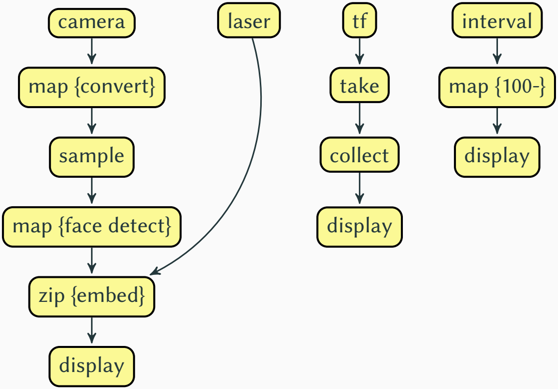


Stream Language: Actions

```
Stream
.just(0)
.loop(s -> s.map(i -> i + 1))
.subscribe(System::println);
```



Application: Robot Panel

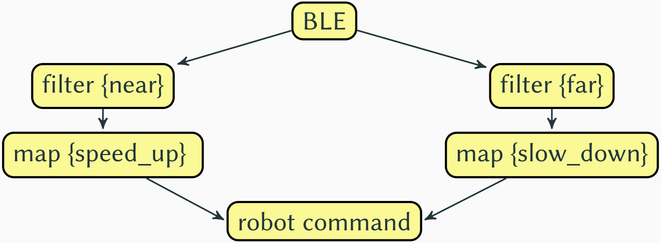


Robot panel demo

Application: Robot Hospital Guide

1. Robot guides patients to parts of the hospital
2. Patient holds a smartphone that broadcasts bluetooth signals
3. Robot adjusts its speed, according to distance

Application: Robot Hospital Guide



Application: Robot Hospital Guide

```
// RHEA setup
Stream.configure(new HazelcastDistributionStrategy(
           RxjavaEvaluationStrategy::new,
           RosEvaluationStrategy::new,
           MqttEvaluationStrategy::new));
// Topics
Topic<RobotCommand> vel = new RosTopic<>("/robot/cmd");
Topic<Proximity> ble = new MqttTopic<>("/ble");
// Running on smartphone
Stream.from(ReactiveBeacons.observe())
      .map(Beacon::getProximity)
      .subscribe(ble);
// Running on robot
Stream<Proximity> prox = Stream.from(ble);
prox.filter(Proximity::isNear)
    .map(d -> Commands.SPEED_UP)
    .subscribe(vel);
prox.filter(Proximity::isFar)
    .map(d -> Commands.SLOW_DOWN)
    .subscribe(vel);
```

RHEA as a coordination language

- Declarative glue code
- Multiple heterogeneous devices/streams
- Dataflow in the large, whatever in the small

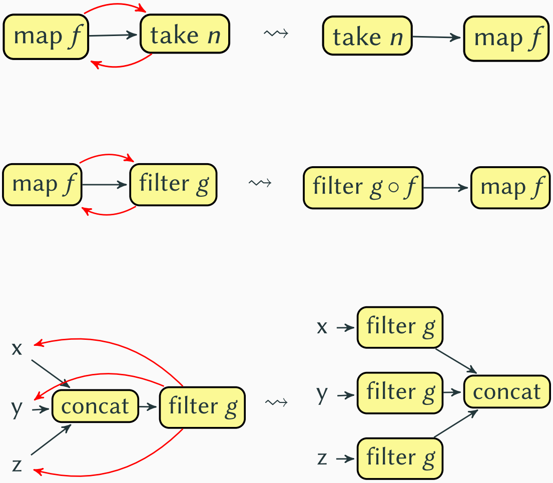
Optimizations

Series of semantics-preserving graph transformations

- Proactive filtering
- Granularity adjustment

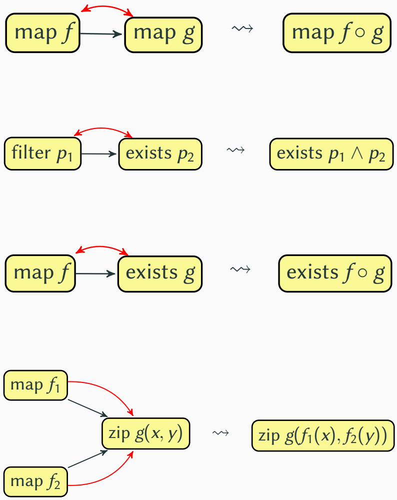
Optimizations: Proactive Filtering

Transfer as few elements as possible



Optimizations: Granularity Adjustment

Merge nodes



map f → map g ⇝ map $f \circ g$

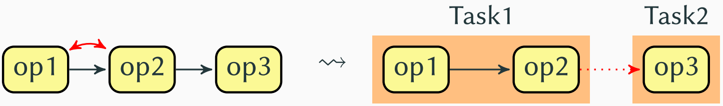
filter p_1 → exists p_2 ⇝ exists $p_1 \land p_2$

map f → exists g ⇝ exists $f \circ g$

map f_1, map f_2 → zip $g(x, y)$ ⇝ zip $g(f_1(x), f_2(y))$

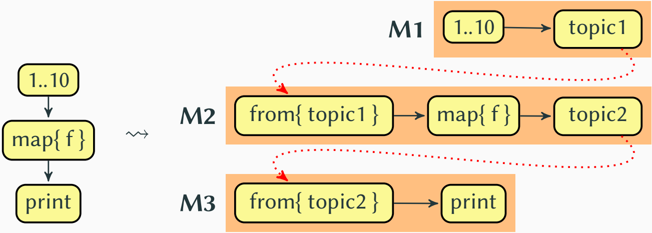
Distribution: Task Fusion

1. If desired granularity not reached, perform task fusion:



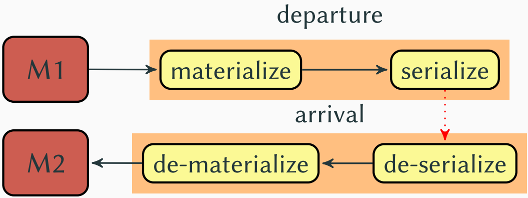
Distribution: Node Placement

2. Place nodes in the available machines, in order to:
   - minimize communication overhead
   - satisfy hard constraints (e.g. ROS not available on raspberry)



Distribution: Serialization

3. Streams can terminate either with `Complete` or `Error`.
    - Necessary to materialize them when transferring



Limitations

- Difficult to extend the available operators
- The surface syntax is not strict enough
    - Only single-assignment in `Stream` variables
    - Specific program structure (configuration $\rightarrow$ dataflow)
    - Only pure functions as arguments to higher-order operators

Future Work

- More sophisticated optimizations
- Reinforcement learning for node placement
- Dynamic reconfiguration (hot-swapping code)
- Erlang-style error handling
- Machine-learning backend
- . . .

Conclusion

- Some domains are still full of low-level techniques
- The FP paradigm can overcome this quite nicely
- Higher, higher, higher!

Questions?