Arrowized FRP Abstraction for Functional IoT Programs

Yahui Song and Wei-Ngan Chin

National University of Singapore

OVERVIEW

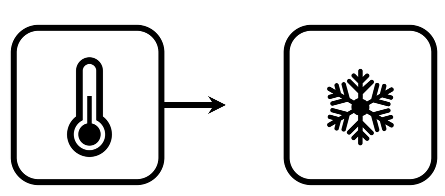
We propose a practical Arrowized Functional Reactive Programming(AFRP) abstraction and a prototype of the embedded domain-specific language (EDSL) in Haskell for Internet of Things (IoT) development which guides IoT developers to write high-order FRP programs directly.

- Immutability and static checking of purely functional programming are good for program reliability and maintainability.
- Continues time-varying values in FRP can be neatly mapped into IoT systems.
- Arrowized FRP further tackles the biggest drawback "space leak" problem of classical FRP.

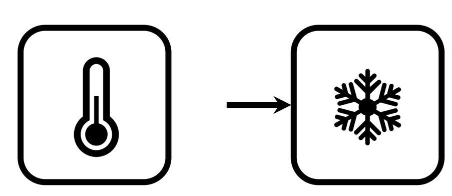
Altogether, providing this abstraction not only simplifies the complex task of building responsive and type-safe IoT systems but also provides the ability to reason about IoT event streams.

REACTIVE PROGRAMMING & INTERNET OF THINGS

Assuming there is a dependency between temperature and the air conditioner (AC) such as if the temperature rose too high, the AC would be turned on automatically. From this functionality, we may abstract two modules, one is **Sensor**, one is **AC**, and the update method is needed to be defined.



Passive Programming



Reactive Programming

Passive Programming:

- Update method is defined in the **Sensor** module
- Remote setters and updates
- **Sensor** module is responsible for changes
- **AC** has no awareness on the dependence

Reactive Programming:

- Update method is defined in the **AC** module
- Events, observations and self-updates
- **AC** module is responsible for changes
- Easy to track/add dependencies on **AC** module

Though there is no absolute goodness or badness between these two styles, in real cases, no matter for debugging or extending the system, we care more *"How does this module work?"*, which is easy to be answered in reactive programming.

FRP & Arrowized FRP

Functional Reactive Programming works with mutable values by recasting them as **time-varying** values, capturing the temporal aspect of mutability. FRP originally composes two particular abstractions: a continuous modelling of **behaviors**, and discrete reactive **events** from users or processes.

$$Behavior\ \alpha = Time \rightarrow \alpha \text{ - - - - - - - - - } continuous$$

$$Event\ \alpha = [(Time,\ \alpha)] \text{ - - - - - - - - - - - - } discrete$$

$$Event\ \alpha \approx Behavior\ (Maybe\ \alpha)$$

Classical FRP

$$Signal\ \alpha = Time \rightarrow \alpha$$

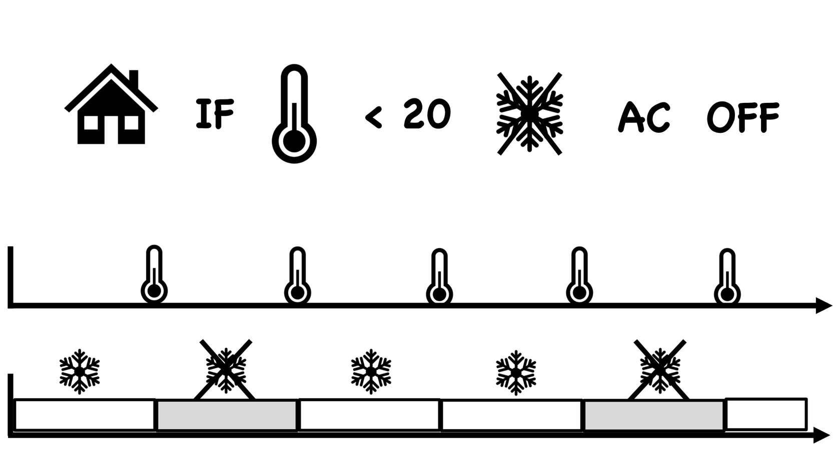
$$SF\ \alpha\ \beta = Signal\ \alpha \rightarrow Signal\ \beta$$

Arrowized FRP

Arrowized FRP rules out the "space leak" of functional programming by rewriting the signal functions as functions from one signal to another signal.

EXAMPLE – ENERGY AUTOMATION

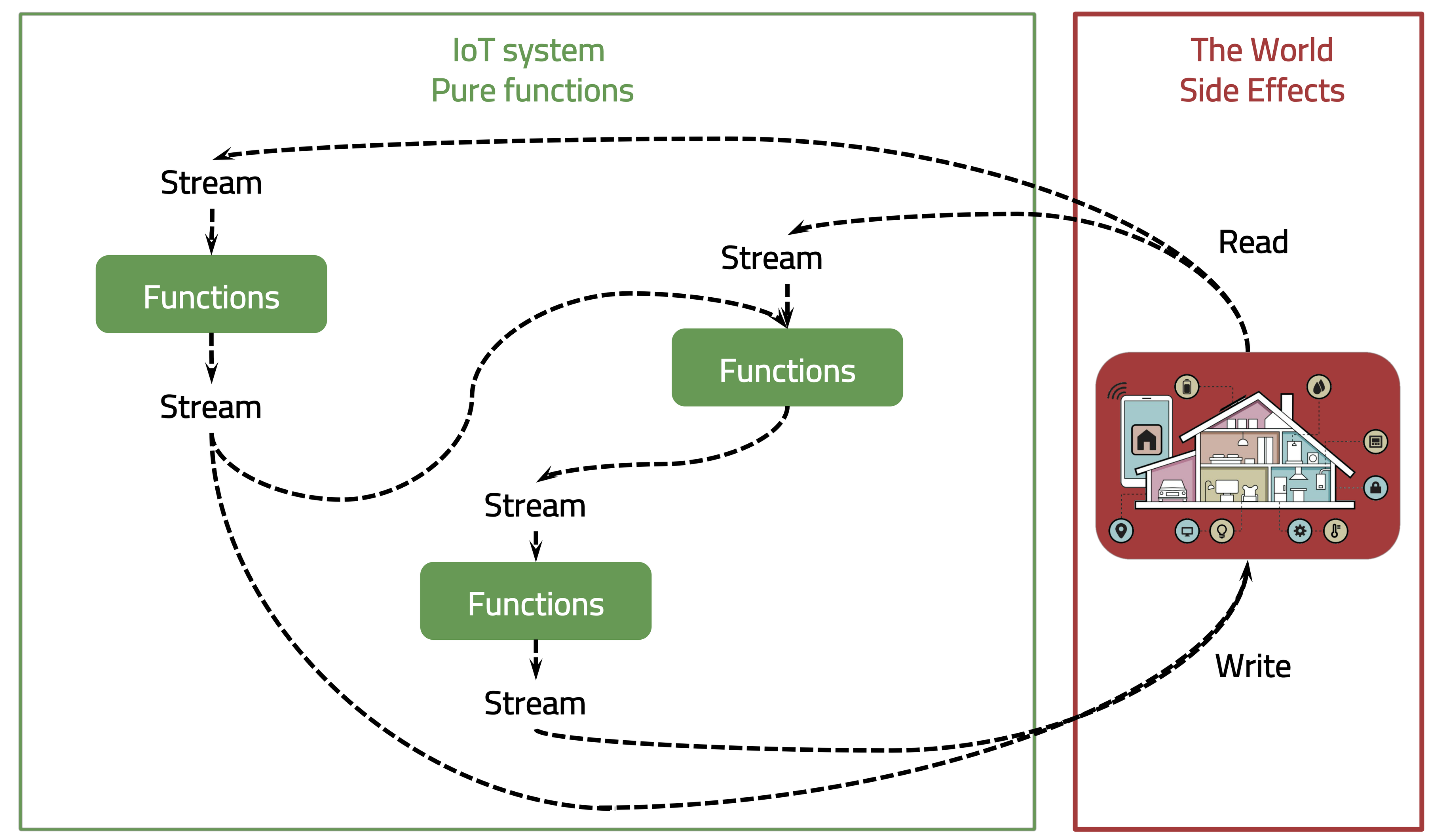
Each temperature is a discrete Event while the state of the AC is a continuous Behavior.




```
data Signal a = Signal {func :: Time -> a}

type Time = Double

type Temperature = Signal Float

type AC = Signal Bool
```

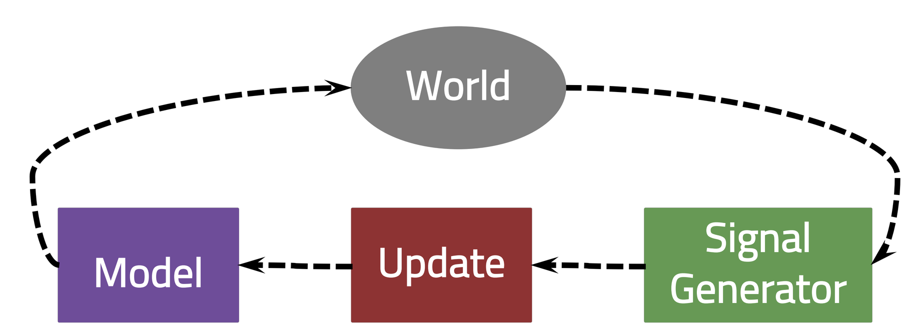
SIGNAL STREAM GRAPHS

We start with reading from the world and end with writing to the world. Inside of this functional reactive IoT system, we only have pure functions and data streams pushing the side-effects to the edge of the system.



EDSL DESIGN

This embedded domain-specific language (EDSL) only expose 3 functions for developers to fill up: Model, Update and Signal Generator.



Syntax

(value) V, W ::= () | c | x | λx^{α}.M | <V, W> | i

(program) M, N ::= V | MN | op($\overrightarrow{M}$)

| **if** M **then** M_1 else M_2

| **let** x = M_1 **in** M_2

| **lift** M_1 M_2 M_3 $(\alpha \rightarrow \beta) \rightarrow Signal\ \alpha \rightarrow Signal\ \beta$

| **foldp** M_1 M_2 M_3 $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow Signal\ \alpha \rightarrow Signal\ \beta$

$$n \in \mathbb{R} \cup Boolean \qquad x \in Var \qquad i \in Input$$

Type System

$$\tau :: unit \mid number \mid \text{bool} \mid \tau \rightarrow \tau'$$

$$\sigma :: \text{signal}\ \tau \mid \tau \rightarrow \sigma \mid \sigma \rightarrow \sigma'$$

$$\eta :: \tau \mid \sigma$$

Type Judgments

UNIT
$$\frac{}{\Gamma \vdash () : \text{unit}}$$

NUMBER
$$\frac{}{\Gamma \vdash n : \text{number}}$$

BOOL
$$\frac{}{\Gamma \vdash n : \text{bool}}$$

LET
$$\frac{\Gamma \vdash e_1 : \eta \qquad \Gamma, x : \eta \vdash e_2 : \eta'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \eta'}$$

INPUT
$$\frac{\Gamma(\texttt{i}) = \tau}{\Gamma \vdash \texttt{i} : \text{signal } \tau}$$

VAR
$$\frac{\Gamma(x) = \eta}{\Gamma \vdash x : \eta}$$

LAMBDA
$$\frac{\Gamma, x : \eta \vdash e : \eta'}{\Gamma \vdash \lambda x : \eta.e : \eta \rightarrow \eta'}$$

LIFT
$$\frac{\Gamma \vdash e : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \qquad \Gamma \vdash e_i : \text{signal } \tau_i\ \forall\, i \in 1 \ldots n}{\Gamma \vdash \text{lift}_n\ e\ e_1 \ldots e_n : \text{signal } \tau}$$

APPLICATION
$$\frac{\Gamma \vdash e_1 : \eta \rightarrow \eta' \qquad \Gamma \vdash e_2 : \eta}{\Gamma \vdash e_1\ e_2 : \eta'}$$

FOLD
$$\frac{\Gamma \vdash e_{fun} : \tau \rightarrow \tau' \rightarrow \tau' \qquad \Gamma \vdash e_{ini} : \tau' \qquad \Gamma \vdash e_{new} : \text{signal } \tau}{\Gamma \vdash \text{foldp}_n\ e_{fun}\ e_{ini}\ e_{new} : \text{signal } \tau'}$$

EXPERIMENTS

- Raspberry Pi 3
- Energy automation examples
- https://youtu.be/p3CbJl8lLls

{yahuis, chinwn}@comp.nus.edu.sg.