Advanced System Software
（先端システムソフトウェア）

#9 (2018/11/12)

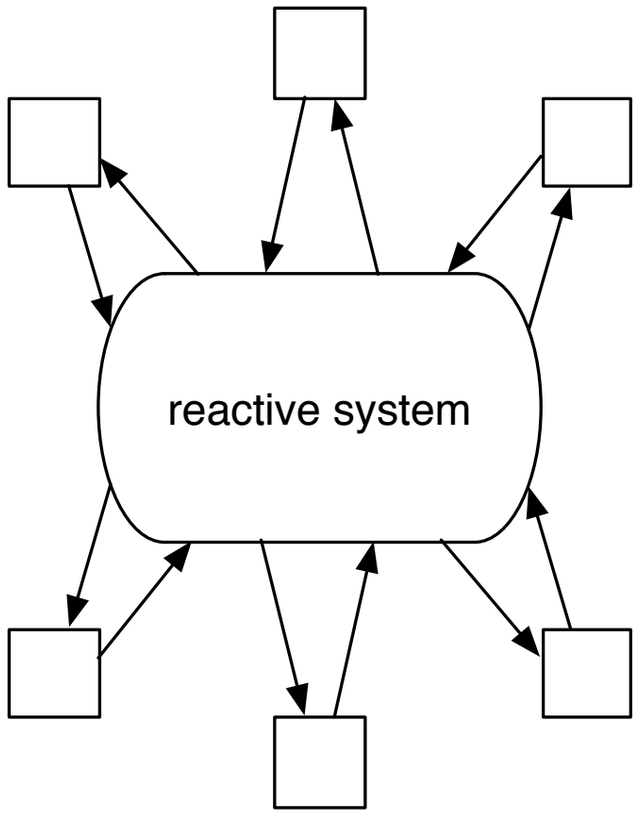
CSC.T431, 2018-3Q
Mon/Thu 9:00-10:30, W832
Instructor: Takuo Watanabe（渡部卓雄）
Department of Computer Science

e-mail: takuo∅c.titech.ac.jp
http://www.psg.c.titech.ac.jp/~takuo/
ext: 3690, office: W8E-805

Agenda

- Advanced Topics
  - Functional Reactive Programming for Embedded Systems

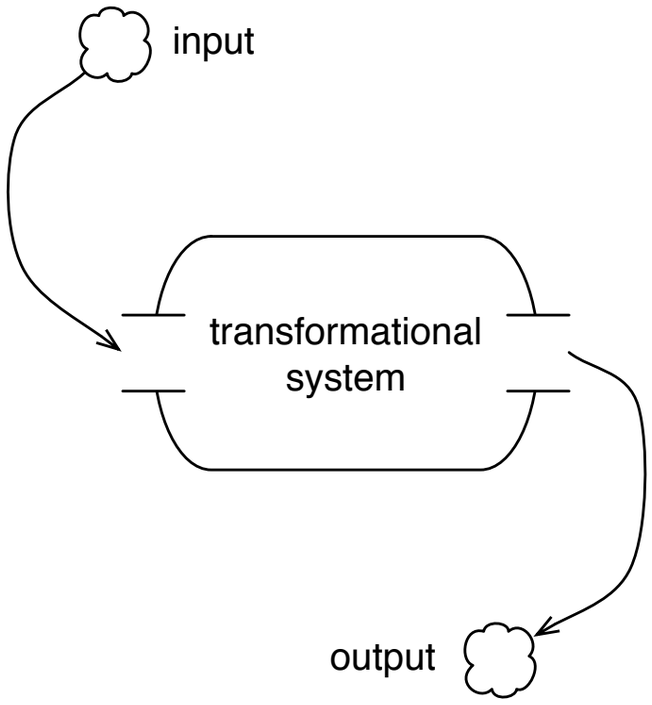
Reactive System



- A computing system that interacts with its environment
  - A reactive system is not supposed to stop but should be continuously ready for interactions.
  - A real-time system is often a reactive system.

- Cf. transformational system

Transformational System



- A system that accepts an input and then produces an output and stops
- Classical model of computation (function)

Computational Model

- Transformational Systems
  - computational system as a function
  - λ-calculus, Recursive Functions, Turing Machine
- Reactive Systems
  - computational system as a process
  - π-calculus, CCS, CSP, Chemical Abstract Machine, Actor Model

Programming Embedded Systems

- Embedded System ⊆ Reactive System
  - Embedded systems can be categorazed into reactive systems because they typically interact with their environment. They are not supposed to stop but should be continuously ready for incoming events.
    - ex) GUI, Web Apps., Embedded Systems
  - The order and timing of events are not predictable

- Typical Patterns for Programming Embedded Systems
  - Polling
  - Callbacks (e.g., interrupt handers, event handlers)

Example: Simple Fan Controller

- Two environmental sensors
  - Temperature
  - Humidity

- Controlling policy
  - Turns the fan ON when $DI \geq 75.0$
  - OFF otherwise

  - DI : Discomfort Index
    - $0.81T + 0.01H\,(0.99T - 14.3) + 46.3$
      - T : temperature (Celsius)
      - H : relative humidity (%)
    - Empirically, 50% of people feel uncomfortable if $DI = 75$.

Typical Pattern (1): Polling (a)

using an integrated sensor

```
#define FAN_PIN 21
DHT12 dht12;

void setup() {
    Wire.begin();
    pingMode(FAN_PIN, OUTPUT);
}
// Main polling loop
void loop () {
    // read sensor values via I2C
    dht12.update();
    float tmp = dht12.temperature();
    float hmd = dht12.humidity();
    // calculate current discomfort index
    float di = 0.81 * tmp + 0.01 * hmd
                * (0.99 * tmp - 14.3) + 46.3;
    // turns the fan on if di >= 75, off otherwise
    digitalWrite(FAN_PIN, di >= 75.0);
}
```

Typical Pattern (1): Polling (b)

using two separate sensors

```
#define FAN_PIN 21
TMP11 tmp11;  // temperature sensor
HMD22 hmd22;  // humidity sensor

void setup() {
    Wire.begin();
    pingMode(FAN_PIN, OUTPUT);
}
// Main polling loop
void loop () {
    // read sensor values via I2C
    tmp11.update();    // 3ms
    hmd22.update();    // 15ms
    float tmp = tmp11.read();
    float hmd = hmd22.read();
    // calculate current discomfort index
    float di = 0.81 * tmp + 0.01 * hmd
               * (0.99 * tmp - 14.3) + 46.3;
    // turns the fan on if di >= 75, off otherwise
    digitalWrite(FAN_PIN, di >= 75.0);
}
```

Typical Pattern (2): Callbacks (a)

```
TMP11 tmp11;
HMD22 hmd22;

void do_action() {
    float di = 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3;
    digitalWrite(FAN_PIN, di >= 75.0);
}

void tmp_handler () {
    tmp = tmp11.read();
    do_action();
}

void hmd_handler () {
    hmd = hmd22.read();
    do_action();
}

void setup() {
    Wire.begin();
    tmp11.attach(tmp_handler);
    hmd22.attach(hmd_handler):
}

void loop () { vTaskDelay(portMAX_DELAY); }
```

Typical Pattern (2): Callbacks (b) If ISR cannot do I/O

```
TMP11 tmp11;
HMD22 hmd22;
float tmp, hmd;
SemaphoreHandle_t sem;
TaskHandle_t tmp_reader_task, hmd_reader_task;

void tmp_reader_task_fun() {
    for (;;) {
        xTaskNotifyWait(0, 0, NULL, portMAX_DELAY);
        float tmp0 = tmp11.read();
        xSemaphoreTake(sem, portMAX_DELAY);
        tmp = tmp0;
        xSemaphoreGive(sem);
    }
}

void hmd_reader_task_fun() {
    for (;;) {
        xTaskNotifyWait(0, 0, NULL, portMAX_DELAY);
        float hmd0 = hmd22.read();
        xSemaphoreTake(sem, portMAX_DELAY);
        hmd = hmd0;
        xSemaphoreGive(sem);
    }
}
```

Typical Pattern (2): Callbacks (b)

```
void tmp_handler () { xTaskNotifyFromISR(tmp_reader_task); }

void hmd_handler () { xTaskNotifyFromISR(hmd_reader_task); }

void setup() {
    Wire.begin();
    sem = xSemaphoreCreateBinary();
    xTaskCreatePinnedToCore(tmp_reader_task_fun, "tmp_reader_task",
                            4096, NULL, 1, &tmp_reader_task, 1);
    xTaskCreatePinnedToCore(hmd_reader_task_fun, "hmd_reader_task",
                            4096, NULL, 1, &hmd_reader_task, 1);
    tmp11.attach(tmp_handler);
    hmd22.attach(hmd_handler):
}

void loop () {
    float di;
    xSemaphoreTake(sem, portMAX_DELAY);
    di = 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3;
    xSemaphoreGive(sem);
    digitalWrite(FAN_PIN, di >= 75.0);
    delay(100);
}
```

Polling and Callbacks

- Polling
  - OK for simple system
  - The slowest input govern the input timing
- Callbacks
  - Can handle different timing inputs
  - Split the code into small pieces
  - Callback functions (handlers) may have constraints

- In general, the above patterns are used with threads and/or state machines. This complicates the code and lowers the readability, maintainability, etc.

Functional Reactive Programming (FRP)

- A programming paradigm for reactive systems based on functional (declarative) abstractions to express time-varying values and events.

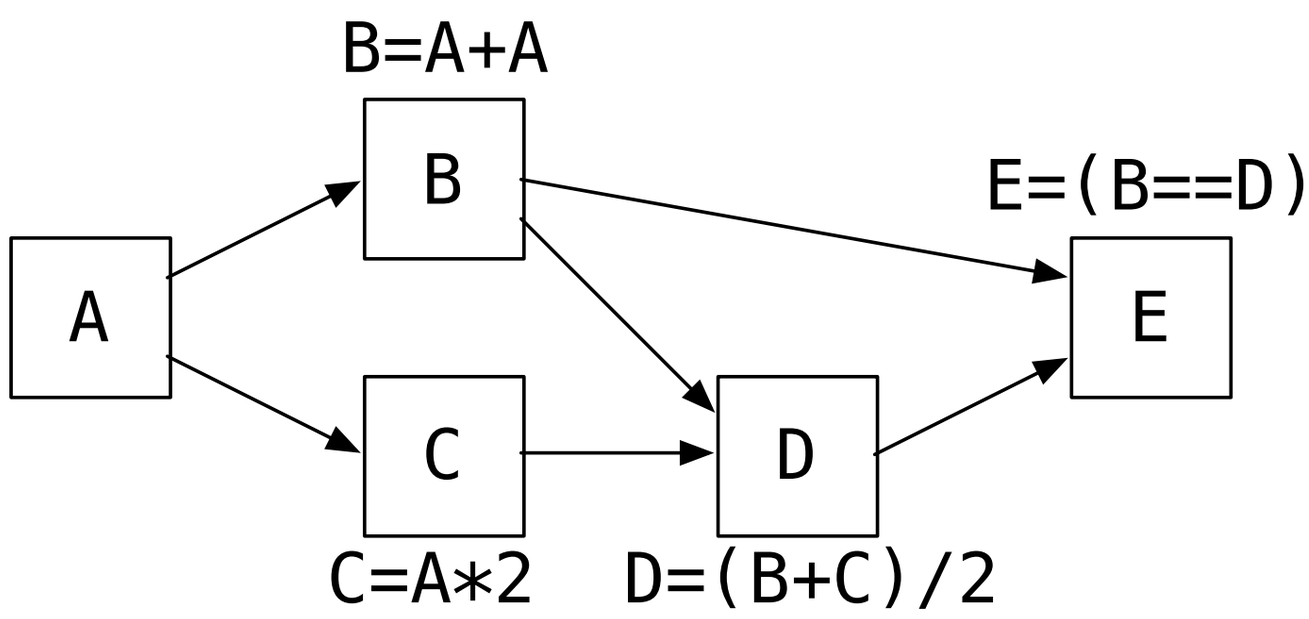
- Time-Varying Values: TVV α = Time → α
  - First class entities representing (continuously) changing data (of type α) over time
  - ex) tmp :: TVV float
  - aka. Signal α, Behavior α
- Events: Event α = [(Time, α)]
  - List of discrete events of type α
  - ex) button :: Event bool

Reactive Systems

- A reactive system is a function of type TVV α → TVV β
  - α : input type
  - β : output type
- Causality requirement
  - The value of the output at time t must be determined by input on interval [0, t].
  - pure and stateless
    - if output at time t only depends on input at time t
  - impure and stateful
    - if output at time t depends on input over the interval [0, t]

DAG Representation

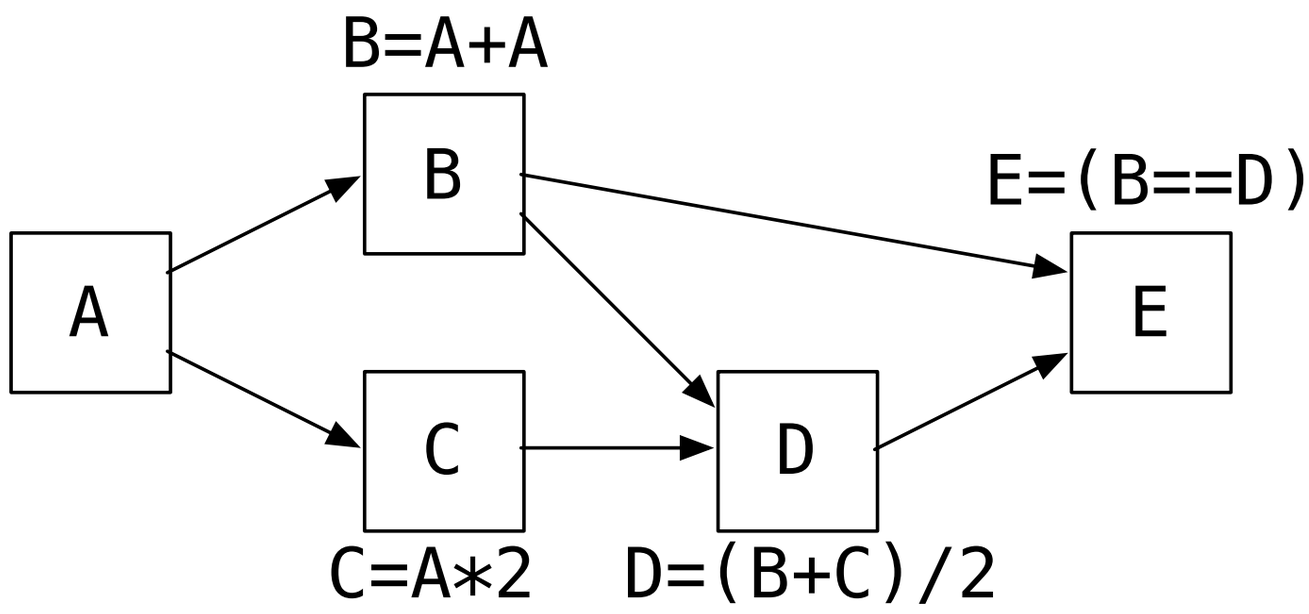
- In FRP, a program can be represented as a directed graph without cycles (= DAG)



- Each box corresponds to a TVV
  - A : input, E : output
- In this example, the value of E is always true

Glitch-Freedom

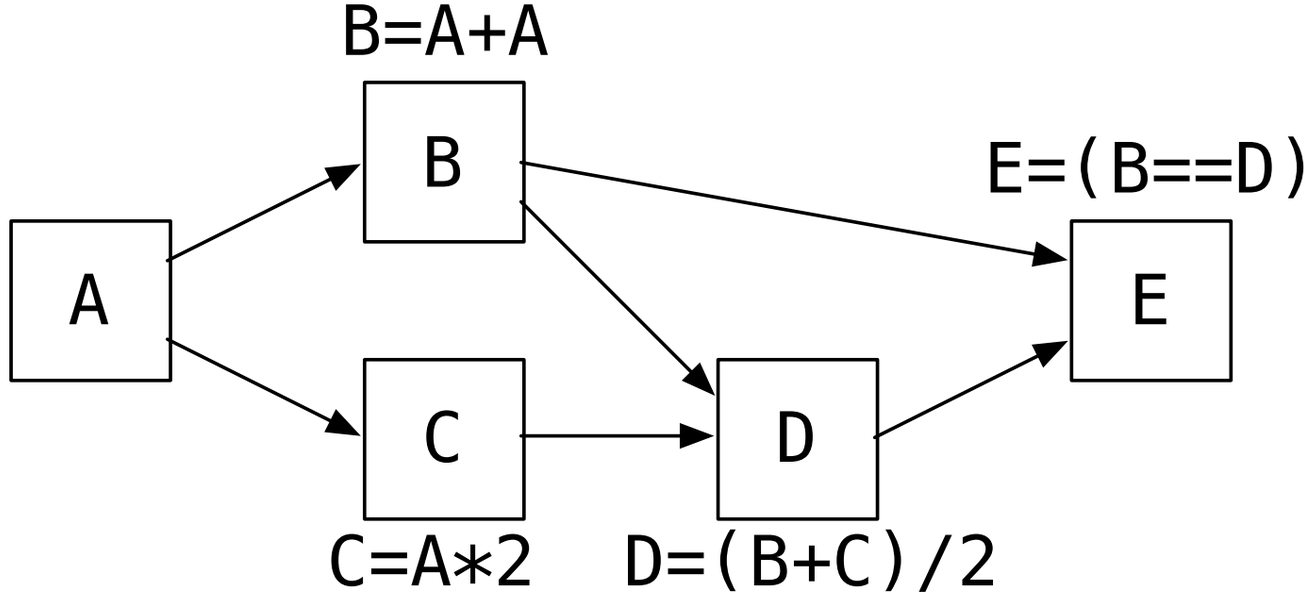
- In this example, a glitch is the phenomenon that there is an observable difference between B and D due to the delays in updating B, C and D.



- In a glitch-free FRP language, the value of E should always be true regardless of the current value of A.

Synchronous Execution Model

- To avoid glitches, the nodes in the DAG should be updated synchronously along the sequence obtained by topological sorting of the DAG



- In this example, the order of update can be
  - A, B, C, D, E, or
  - A, C, B, D, E

Time/Space-Leak

- A naive representation of TVV a leads to phenomena called Time/Space-Leaks, which refers to unnecessary computation / storage required to calculate the current value of a TVV.
- Example:
  - Recompute along the history of the TVV
  - Keep the history of the TVV in the list

- To avoid time/space-leak
  - Signal functions
  - TVV as a (restricted) primitive

Lifting

```
pure :: a -> Signal a
lift2 :: (a -> b -> c) ->
         (Signal a -> Signal b -> Signal c)
```

- In an FRP language in which the type of TVVs is defined separately from its value type, functions that convert value types to TVV types are required.

```
di :: Float -> Float -> Float
di t h = 0.81 * t * 0.01 * h * (0.99 * t - 14.3) + 46.3

fan :: Signal Bool
fan = lift2 (>=) (lift2 di tmp hmd) th

th :: Signal Float
th = pure 75.0
```

- Explicit lifting as in the above example may complicate programs in FRP.

FRP Languages and Libraries

- Fran [Elliot97]
  - Haskell FRP library for animation
- Yampa [Hudak03][Coutney03]
  - Haskell FRP library used for robot control
- FrTime [Cooper04]
  - Extension of Scheme for GUI, simulation
- Flapjax [Meyerovich09], Elm [Czaplicki13]
  - Library/language for client-side web programming

- The majority of the FRP systems proposed so far are based on Haskell or other languages that require rich runtime resource.

FRP for Small-Scale Systems

- Observation
  - FRP can be beneficial for developing general embedded systems
- Goal
  - To show that FRP can also be advantageous for developing resource-constrained embedded systems, and
  - To propose an advanced development method based on FRP that can replace traditional C/C++ based methods
- Challenge
  - small memory size, not so powerful processors
  - expressiveness w/o losing the advantage of pure FP
- Our Approach
  - Language Design, Type System

Emfrp

Sawada & Watanabe, Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, CROW 2016.

- An FRP language for small-scale embedded systems
  - Strongly-typed, Purely functional
    - parametric polymorphism & type inference
    - algebraic data types & pattern matching
  - Simple abstraction for time-varying values (aka signals)
    - node: named, non-first-class representation
    - lifting-free
  - Bounded & predictable amount of runtime memory
    - syntactic restrictions & type system

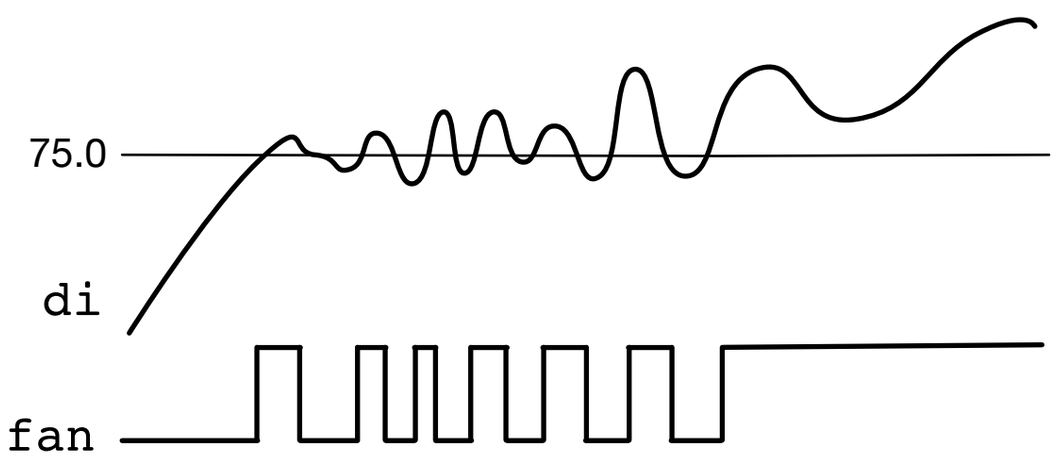
- Implementation
  - http://github.com/psg-titech/emfrp
    - The compiler generates platform-independent C code runnable several microcontrollers (AVR, ARM Cortex-M)

Ex1: Fan Controller

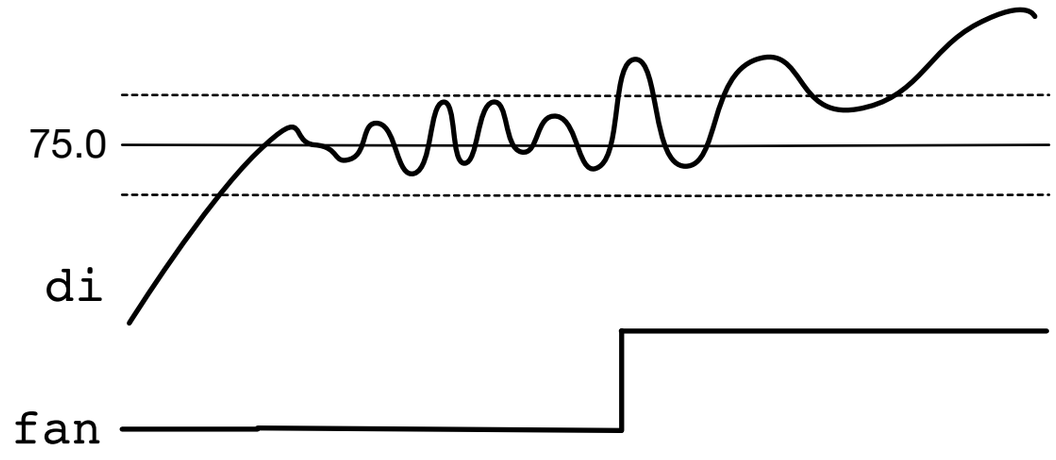
```
module FanController # module name
in  tmp : Float,     # temperature sensor
    hmd : Float      # humidity sensor
out fan : Bool       # fan switch
use Std              # standard library

# discomfort (temperature-humidity) index
node di = 0.81 * tmp + 0.01 * hmd
          * (0.99 * tmp - 14.3) + 46.3

# fan switch
node init[False] fan = di >= 75 + ho

# hysteresis offset
node ho = if fan@last then -0.5 else 0.5
```

History-Sensitive Behavior with `@last`

Emfrp syntax for referring to the *previous* value of an *arbitrary* node (*cf*. `foldp` in Elm / loop in Yampa)




```
# fan switch
node fan = di >= 75
```




```
# fan switch
node init[False] fan =
           di >= 75 + ho

# hysteresis offset
node ho = if fan@last
           then -0.5 else 0.5
```

Emfrp: Language Design

- Pure functional language with:
  - Algebraic datatypes, Parametric polymorphism, Type inference
  - No higher-order functions, No recursion (datatypes and functions)
- Node(τ) : Datatype for TVVs (signals)
  - The syntax and type system of Emfrp disallow that nodes are passed around as function arguments.
  - Functions applied to nodes are automatically lifted.
  - @last notation provides access to the value of arbitrary nodes at the *previous moment*
- The above restrictions and properties guarantee that runtime memory size can be statically determined.

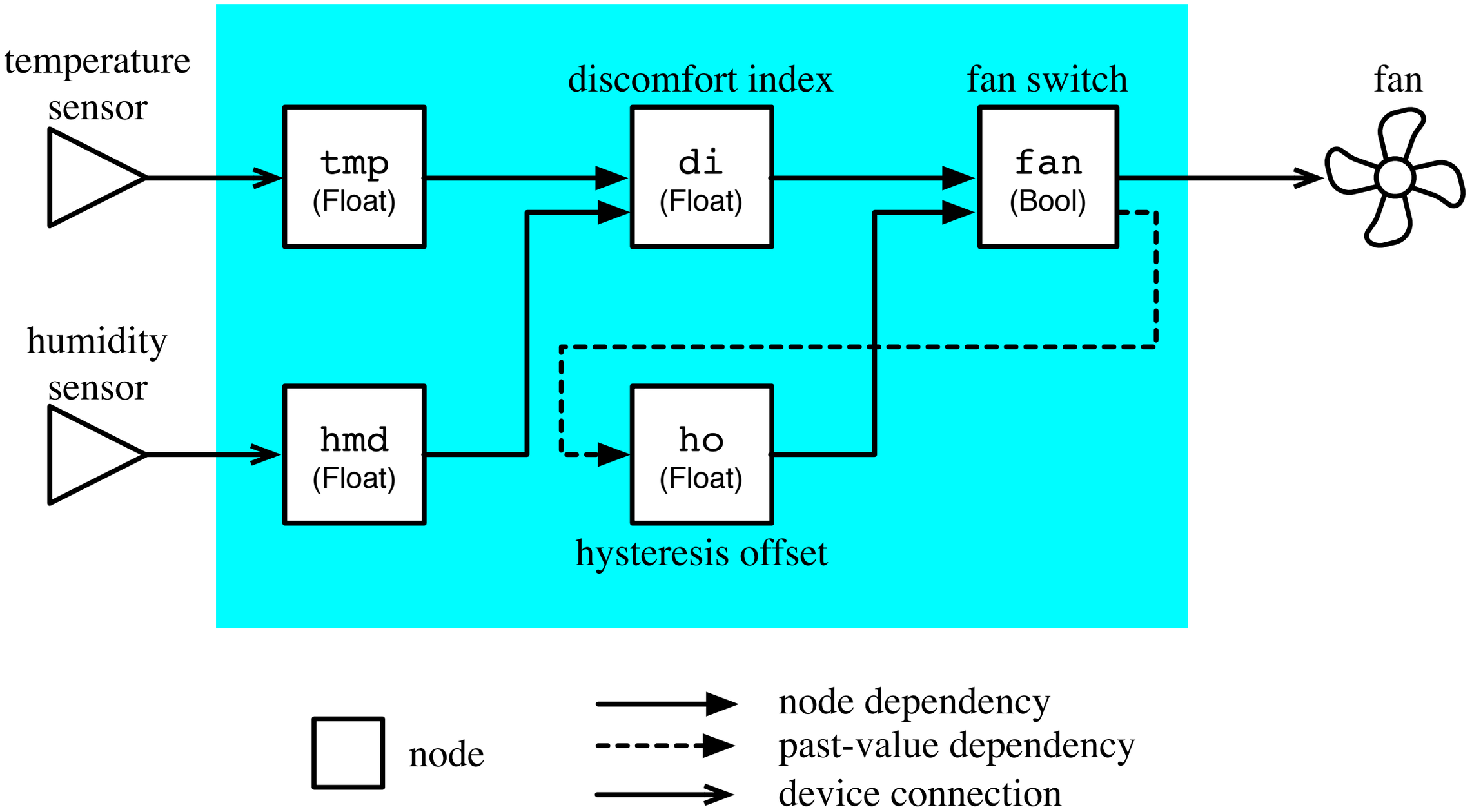
No Lifting Required

- Majority of FRP languages provide separate datatypes for TVVs
  - e.g. Signals, Signal Functions
- They require 'lifting'
  - lift :: (a → b) → Signal a → Signal b
  - s :: Signal a, f :: a → b ⇒ lift f a :: Signal b

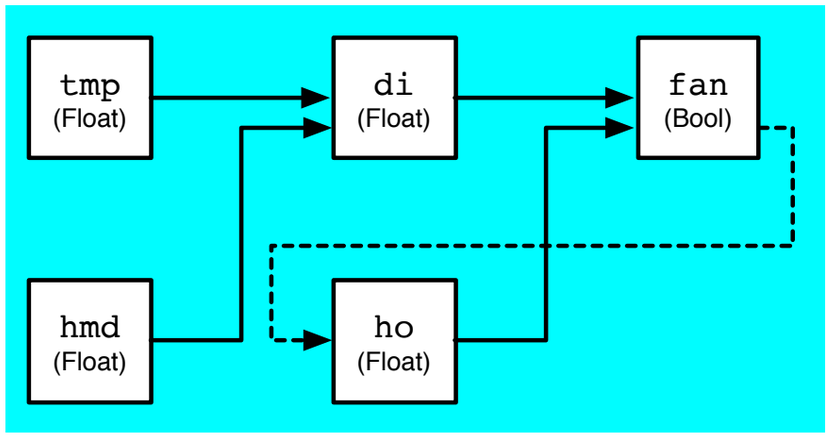
- Emfrp provides "auto lifting"

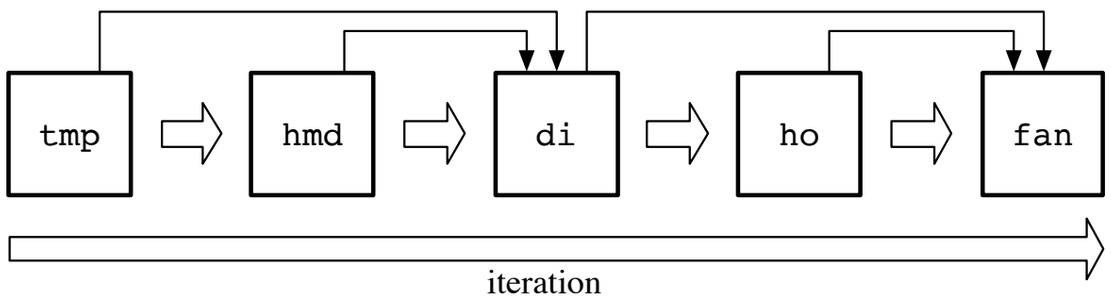
$$\frac{f : \tau_1 \times \tau_2 \to \tau \quad a_1 : \tau_1 \quad a_2 : \tau_2}{f(a_1, a_2) : \tau} \qquad \frac{f : \tau_1 \times \tau_2 \to \tau \quad a_1 : \tau_1 \quad a_2 : \mathsf{Node}(\tau_2)}{f(a_1, a_2) : \mathsf{Node}(\tau)}$$

DAG Representation of Ex1



Synchronous Execution Model





- The runtime system updates the values of the nodes by repeatedly evaluating them in a fixed order compatible to the node graph (DAG).
- Pros
  - Simple and suitable for small devices
- Cons
  - wasting CPU
  - Not flexible/adaptable

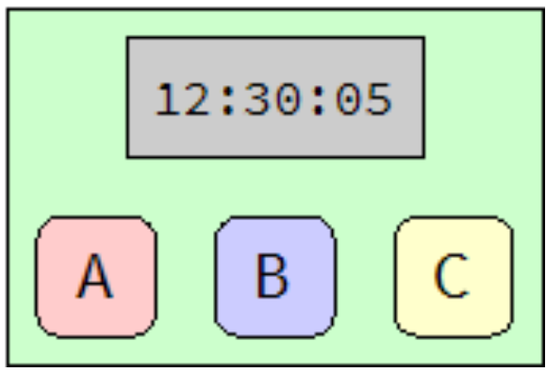
```
// current node values
float node_tmp, node_hmd, node_di, node_ho;
bool node_fan;
// previous node values
float last_tmp, last_hmd, last_di, last_ho;
bool last_fan;

// performs a single iteration
void update(void) {
    // reads input node values
    Input(&node_tmp, &node_hmd);
    // updates internal/output nodes
    update_di(&node_di, node_tmp, node_hmd);
    update_ho(&node_ho, last_fan);
    update_fan(&node_fan, node_di, node_ho);
    // performs output
    Output(&node_fan);
    // records node values for next iteration
    last_tmp = node_tmp;
    last_hmd = node_hmd;
    last_ho = node_ho;
    last_di = node_di;
    last_fan = node_fan;
}
```

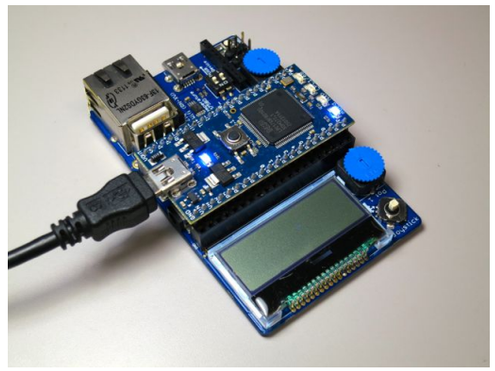
```
// updates the internal/output nodes
static inline void
update_di(float *di, float tmp, float hmd) {
    *di = 0.81 * tmp + 0.01 * hmd
          * (0.99 * tmp - 14.3) + 46.3;
}

static inline void
update_ho(float *ho, bool fan) {
    *ho = fan ? -0.5 : 0.5;
}

static inline void
update_fan(bool *fan, float di, float ho) {
    *fan = di >= 75.0 + ho;
}

int main() {
    // specifies the initial value of fan
    last_fan = false;
    while (true) update();
}
```

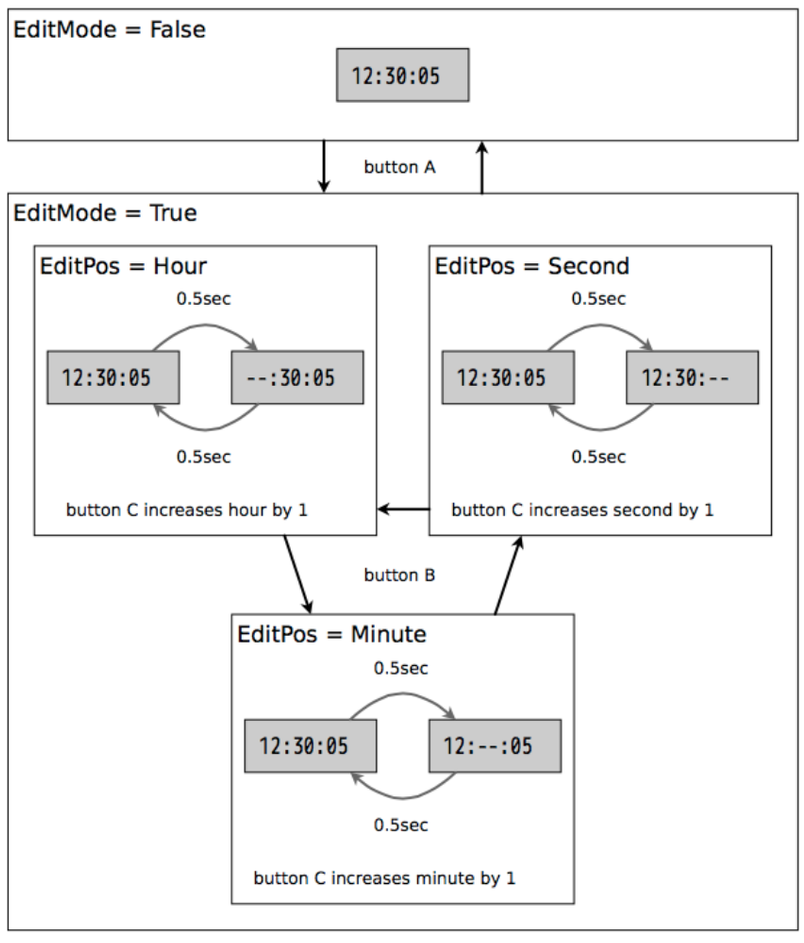
```
// reads the sensor values
void Input(float *tmp, float *hmd)  {
    *tmp = (float)i2c_read_word(ADDR_TMP);
    *hmd = (float)i2c_read_word(ADDR_HMD);
}

// turns the fan ON/OFF
void Output(bool *fan) {
    if (*fan)
        gpio_set_bit(REG_GPIO_A, PIN_FAN_SW);
    else
        gpio_clear_bit(REG_GPIO_A, PIN_FAN_SW);
}
```

Compiled Code of Ex1 (abridged)

Ex2: Digital Clock [CROW2016]



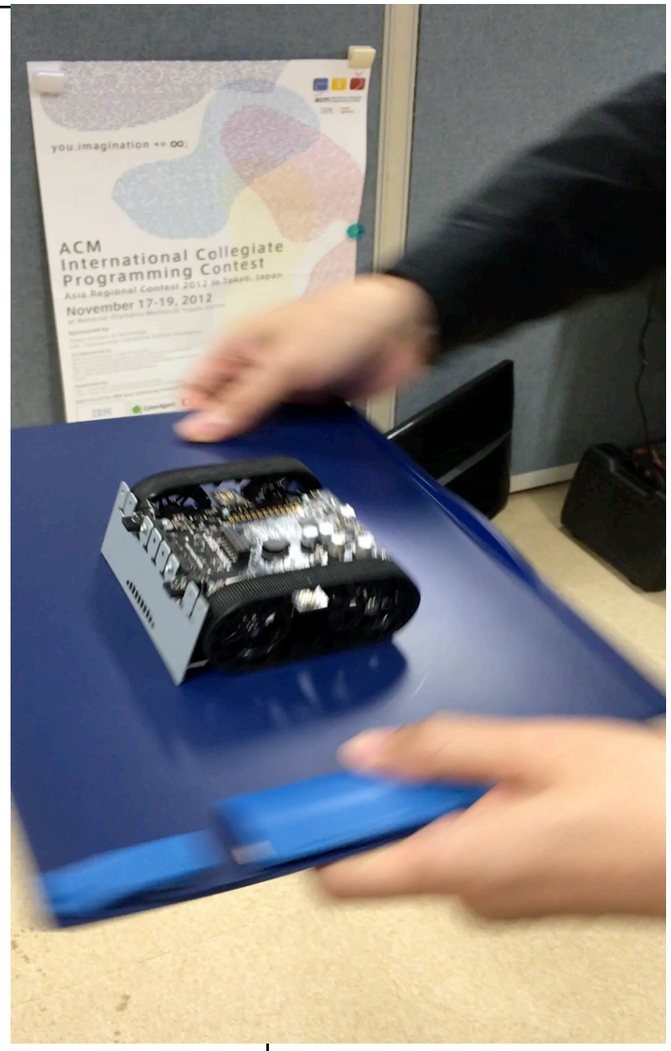




- Target: mbed LPC1768
  - ARM Cortex M3, 96MHz
  - RAM 32K, Flash 512K
  - LCD, 3 Buttons
- Code
  - Emfrp: 80 loc
  - Glue code in C++: 45 loc
  - External Libraries
- Compiled Code
  - Generated C code: 592 loc
  - Binary (ARM RVDS4.1)
    - w/o library : 2.3 KB
    - w/ library: 30.1 KB

Ex3: Resisting Rotation

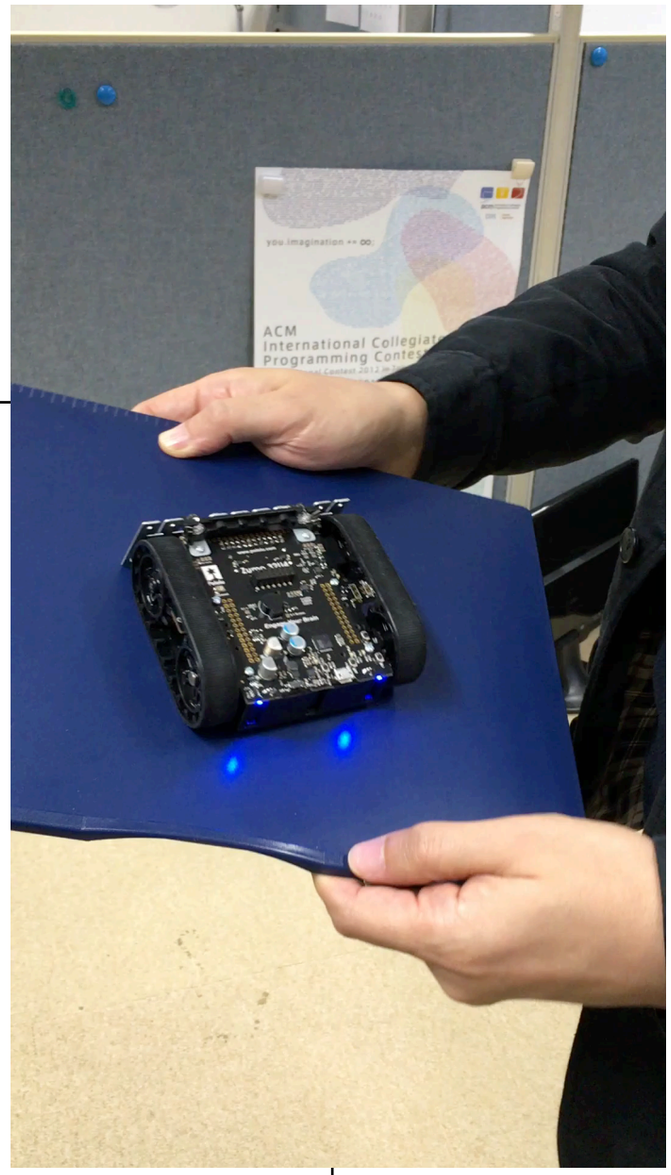
Pololu Zumo 32U4
ATmega 32U4 (32KB Flash, 2.5KB RAM)




```
module RotResist
in  gyroZ : Int,     # gyroscope (z-axis)
    t     : Int      # current time (usec)
out motorL : Int,    # left motor
    motorR : Int     # right motor
use Std

data maxSpeed = 400
func motorSpeed(s) =
     if s < -maxSpeed then -maxSpeed
     else if s > maxSpeed then maxSpeed
     else s

# PD-control parameters
data kp = 11930465 / 1000
data kd = 8

node dt = t - t@last
node init[0] turnAngle =
     turnAngle@last + gyroZ * dt * 14680064 / 17578125
node speed = motorSpeed(-turnAngle / kp - gyroZ / kd)
node (motorL, motorR) = (-speed, speed)
```

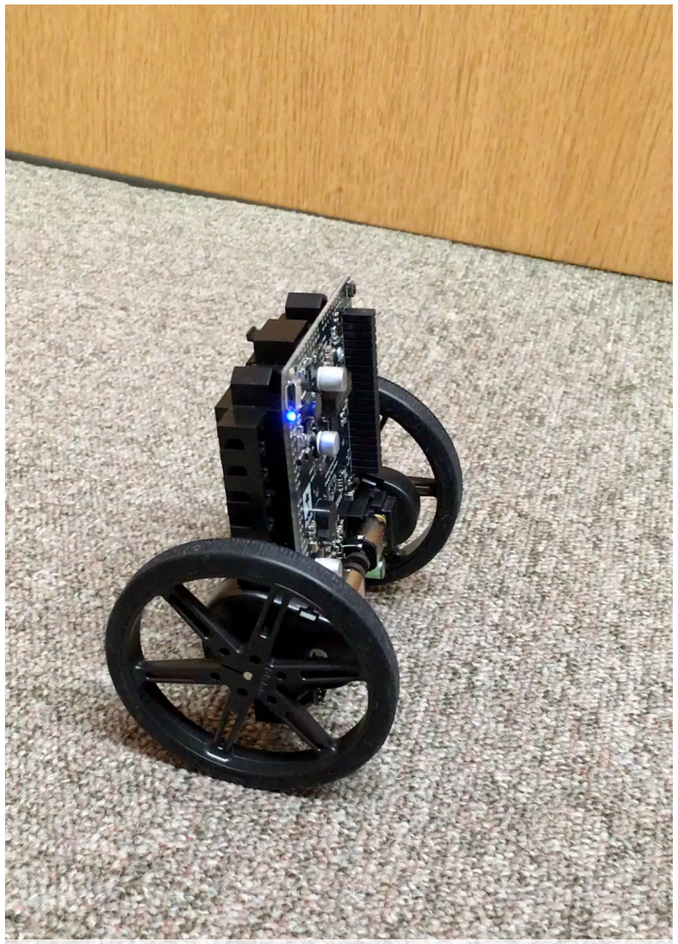
Ex4: Face Uphill

```
module FaceUphill
in  accX   : Int,  # accelerometer (x-axis)
    accY   : Int,  # accelerometer (y-axis)
    encL   : Int,  # left motor encoder
    encR   : Int   # right motor encoder
out motorL : Int,  # left motor
    motorR : Int   # right motor
use Std

data maxSpeed = 150
func motorSpeed(s) =
     if s < -maxSpeed then -maxSpeed
     else if s > maxSpeed then maxSpeed
     else s

node magSq = accX * accX + accY * accY
node turn = if magSq > 1427 * 1427 then accY / 16 else 0
node forward = -(encL + encR)

node motorL = motorSpeed(forward - turn)
node motorR = motorSpeed(forward + turn)
```



Ex5: Balancing

```
module Balancer
in  gyroY  : Int,  # gyroscope (y-axis)
    accX   : Int,  # accelerometer (x-axis)
    encL   : Int,  # left motor encoder
    encR   : Int   # right motor encoder
out motorL : Int,  # left motor
    motorR : Int   # right motor
use Std
...                # definitions of constants

node init[0] angle =
    (angle@last + gyroY * update_time_ms) * 99 / 100

node speedL = encL - encL@last
node speedR = encR - encR@last
node init[0] distL = distL@last + speedL
node init[0] distR = distR@last + speedR

node risingAngleOff  = gyroY * angle_rate_ratio + angle
node init[0] motor =
    motorSpeed(motor@last +
               (angle_resp * risingAngleOff +
                dist_resp * (distL + distR ) +
                speed_resp * (speedL + speedR)) / 100 / gear_ratio)

node diffSpeed = (distL - distR) * dist_diff_resp / 100
node motorL = if accX > 0 then motor + diffSpeed else 0
node motorR = if accX > 0 then motor - diffSpeed else 0
```



Pololu Balboa 32U4 (ATmega 32U4, 32KB Flash, 2.5KB RAM)

Related Work / Systems

	Yampa (FRP)	Elm (FRP)	Simulink (dataflow)	Céu (procedural)	Juniper (FRP)	Emfrp (FRP)
Usable for Small-Scale Embedded Systems	✕	✕	○	○	○	○
Frist-Class Functions	○	○	✕	○	○	✕
Algebraic Data Types & Pattern Matching	○	○	✕	✕	○	○
Declarative Programming & Referential Transparency	○	○	△	✕	○	○
Automated (Unit) Test	✕	✕	○	✕	✕	○
Modularity & Abstractness	○	○	○	△	○	○
Statically Determined Runtime Memory Size	✕	✕	△	○	△	○
Automatic Lifting	✕	✕			✕	○

Current / Future Work

- More efficient execution model
  - Update of unchanged nodes should be eliminated
- Asynchronous execution
- Networking
  - Combining Actor model
- Adaptability, Extensibility
  - Reflection
- Practical Library

- Project Site (source code, papers)
  - http://www.psg.c.titech.ac.jp/frp_embedded.html

Emfrp Compiler

- Source code
  - https://github.com/psg-titech/emfrp
- Sample code
  - https://github.com/psg-titech/emfrp_samples
- Usage, Documentation
  - See Wiki in the above source distribution

Summary

- FRP for small-scale embedded systems
  - Concepts of FRP
  - Design and Implementation of Emfrp