Friot: Functional Reactive Abstraction for IoT Programming

The 31st symposium on Implementation and Application of Functional Languages
@ NUS Soc, Singapore

Yahui Song      Andreea Costea      Wei-Ngan Chin
Internet of Things (IoT)
Internet of Things (IoT)
Internet of Things (IoT)
Motivation Example — LCD

Requirement: to show the current temperature and a clock.
#include <wiringPi.h>
#include <time.h>
#include <string.h>

int main() {
    if (wiringPiSetup() == -1) {
        return -1;
    }
    setup();
    while(1) loop();
    return 0;
}

Requirement: to show the current temperature and a clock.

GPIO library
(General-purpose input/output)
```c
#include <wiringPi.h>
#include <time.h>
#include <string.h>

int main() {
    if (wiringPiSetup() == -1) {
        return -1;
    }
    setup();
    while(1) loop();
    return 0;
}
void setup() {
    pinMode(TempSensor, INPUT);
    pinMode(LCD, OUTPUT); // actually needs more parameters
}
void loop() {
    int temp = digitalRead(TempSensor);
    // ... initialize the timer ...
    tm_info = localtime(&timer);
    strftime(buffer_time, 26, "Time: %H:%M:%S", tm_info);
    string data = strcat(to_string(temp), buffer_time);
    lcdPuts(LCD, data);
}
```

**Requirement:** to show the current temperature and a clock.

1. Read temperature
2. Get current time
3. Show on the lcd

To initialize GPIO
#include <wiringPi.h>
#include <time.h>
#include <string.h>

int main() {
    if (wiringPiSetup() == -1) {
        return -1;
    }
    setup();
    while(1) loop();
    return 0;
}

#define TempSensor 0
#define LCD 1

void setup() {
    pinMode(TempSensor, INPUT);
    pinMode(LCD, OUTPUT); // actually needs more parameters
}

void loop() {
    int temp = digitalRead(TempSensor);
    // ... initialize the timer ...
    tm_info = localtime(&timer);
    strftime(buffer_time, 26, "Time: %H:%M:%S", tm_info);
    string data = strcat(to_string(temp), buffer_time);
    lcdPuts(LCD, data);
}
Requirement: to show the current temperature and a clock.

```c
#include <wiringPi.h>
#include <time.h>
#include <string.h>

int main() {
    if (wiringPiSetup() == -1) {
        return -1;
    }

    setup();

    while(1) {
        loop();
    }
    return 0;
}

#define TempSensor 0
#define LCD 1

void setup() {
    pinMode(TempSensor, INPUT);
    pinMode(LCD, OUTPUT); // actually needs more parameters
    int temp = digitalRead(TempSensor);

    // ... initialize the timer ...
    tm_info = localtime(&timer);
    strftime(buffer_time, 26, "Time: %H:%M:%S", tm_info);

    string data = strcat(to_string(temp), buffer_time);
    lcdPuts(LCD, data);
}
```

1. High reliance on mutable values
2. Global delay
3. Long running data flow (callbacks)

Functional Reactive IoT Programming!
Functional Reactive IoT Programming

If the temperature rose too high, the air conditioner (AC) would be turned on automatically.

IF High AC ON

Sensor Module AC Module
Functional **Reactive** IoT Programming

- **Sensor** owns the Update method
- Remote setters and updates
- **AC** has no awareness on the dependence

- **AC** owns the Update method
- Observers and self-updates
- Easy to track/add dependencies on **AC** module
Functional **Reactive** IoT Programming

- **Sensor** owns the Update method
- Remote setters and updates
- **AC** has no awareness on the dependence

- **AC** owns the Update method
- Observers and self-updates
- Easy to track/add dependencies on **AC** module

"How does this module work?"
Signal Stream Graphs — Core Design

Signals:
- Continuously changing
- Connect to the world
- Infinite

Signals graphs:
- Static
- Multi-threaded
- Asynchronous by default

Pushing the side effect to the edge of the system!
Input / Output are Signals

• Env.temperature :: Int -> Signal Float

• clock :: Signal (Int, Int, Int) -> Day : Hour : Second

• lcd :: Int -> Signal String -> IO ()
Input / Output are Signals

- Env.temperature :: Int → Signal Float

- clock :: Signal (Int, Int, Int) → Day : Hour : Second

- lcd :: Int → Signal String → IO ()

Higher-order functions!
High-order functions — Transformation

- \( \text{lift} :: (a \rightarrow b) \rightarrow \text{Signal } a \rightarrow \text{Signal } b \)

```haskell
data :: Signal String
data = lift floatToStr (Env.temperature 0)
```
High-order functions — Transformation

- \( \text{lift} :: (a \rightarrow b) \rightarrow \text{Signal} \, a \rightarrow \text{Signal} \, b \)
- \( \text{lift}_2 :: (a \rightarrow b \rightarrow c) \rightarrow \text{Signal} \, a \rightarrow \text{Signal} \, b \rightarrow \text{Signal} \, c \)
- \( \text{lift}_n \) ...

\[
data :: \text{Signal} \, \text{String} \\
data = \text{lift} \; \text{floatToStr} \; (\text{Env}.\text{temperature} \, 0)
\]
High-order functions — State

- **foldP** :: \((a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ b\)

```haskell
foldP :: (a -> b -> b) -> b -> Signal a -> Signal b
```

- `totalStep :: Signal Int`

  ```haskell
  totalStep = foldP (\step count -> count + 1) 0 (Env.motion 0)
  ```

  **Port number**

  **Initialization of the count accumulator**
import Rpi
import Env
import Time

show :: Signal String
show = lift_2 (,) (Env.temprature 0) Time.everySec

main :: IO ()
main = Rpi.bPlus [(lcd 2 show)]
Revisit: Requirement:
to show the current temperature and a clock. Use Friot!

```haskell
import Rpi
import Env
import Time

show :: Signal String
show = lift_2 (,) (Env.temperature 0) Time.everySec

blink :: Signal Bool
blink = foldP (\a state -> not state) False Time.everySec

main :: IO ()
main = Rpi.bPlus [(lcd 2 show),
                 (led 3 blink)]
```

More concise code (LOC: 40 (+12) VS 9 (+3))

easy to read and easy to write
ML-like syntax for Friot

(Basic Types) \( \tau ::= \text{Unit} | \text{Int} | \text{Bool} | \text{String} | \tau_1 \rightarrow \tau_2 \)

(Signal Types) \( \sigma ::= \text{Signal} \; \tau | \tau \rightarrow \sigma | \sigma_1 \rightarrow \sigma_2 \)

(Types) \( t ::= \tau | \sigma \)

(Expressions) \( e ::= c | x | F | \lambda x.e | e_1 \oplus e_2 | e_1 e_2 \)

| \text{if} \; (e_1; e_2; e_3) | \text{let} \; (e_1; x.e_2) | \text{fold} \; e_1 \; e_2 \; e_3 \)

| \text{lift}_n \; e \; e_1 \ldots e_n | \text{sync} \; e | \text{prior} \; n \; e \)

(Program) \( \mathcal{P} ::= \mathcal{P} \cup \{F \; \bar{x} = e\} | \emptyset \)

\( n ::= \mathbb{Z} \quad b ::= \text{Bool} \quad \text{string} ::= \text{String} \quad x, F ::= \text{Var} \)
import Rpi
import Env

helper :: Bool -> Bool
helper a = if a then True else False

isPeopleIn :: Signal Bool
isPeopleIn = lift helper (Env.motion 0)

mode_LCD :: Signal Bool
mode_LCD = lift helper isPeopleIn

lcd_show :: Signal String
lcd_show = lift_2 (\a b -> if a then toStr b else "null")
  mode_LCD (Env.temprature 1)

main :: IO ()
main = Rpi.bPlus [ (lcd 2 lcd_show)
  ,(led 3 isPeopleIn) ]
Signal Graph Transformation — multithreads

Input signals

- isPeopleIn
- mode_LCD
- lcd_show

Output signals

Global Model
Signal Graph Transformation — multithreads

Input signals

isPeopleIn

mode_LCD

lcd_show

Output signals

Global Model

isPeopleIn
Signal Graph Transformation — multithreads

Input signals

isPeopleIn → mode_LCD → lcd_show

Output signals

Global Model

isPeopleIn → mode_LCD

Input signals

isPeopleIn → mode_LCD

Output signals
Signal Graph Transformation — multithreads

Input signals:
- isPeopleIn
- mode_LCD
- lcd_show

Output signals:
- lcd_show

Global Model:
- isPeopleIn
- mode_LCD
- lcd_show
Signal Graph Transformation — multithreads

Input signals

isPeopleIn

mode_LCD

lcd_show

Output signals

Single-writer multi-reader lock

Global Model
Signal Stream Graphs — Benefits

1. High reliance on mutable values
2. Global delay
3. Long running data flow (callbacks), passive

1. Immutability of functional language
2. Efficient (Asynchronous and multithreads)
3. Reactive programming style
Conclusion

• Friot is a new FRP language designed for IoT control systems
• Has many functional programming features
• Embedded in Haskell Compiles to C (multithreads)
• Shows clear benefits for logic re-use; time dependent behaviors, being able to be formal verified.
• Efficient (explicit synchronous, Asynchronous by default)
Conclusion

• Friot is a new FRP language designed for IoT control systems
• Has many functional programming features
• Embedded in Haskell Compiles to C (multithreads)
• Shows clear benefits for logic re-use; time dependent behaviors, being able to be formal verified.
• Efficient (explicit synchronous, Asynchronous by default)

https://www.comp.nus.edu.sg/~yahuis/

Thanks a lot for your attention!