National University
of Singapore

Friot: Functional Reactive Abstraction for IoT Programming

Computer Science Research Week @ Soc

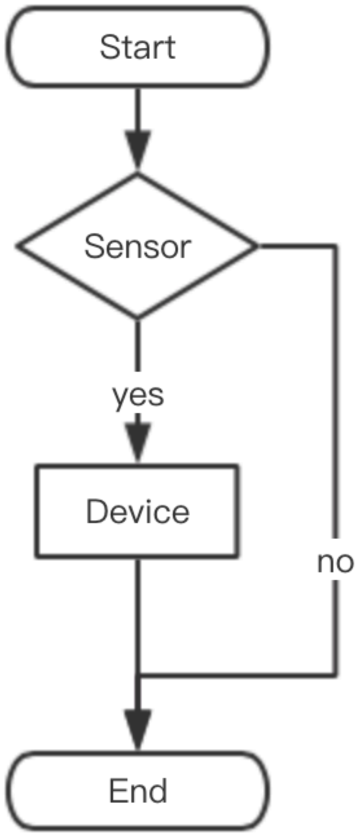
07/08/2019

Yahui Song

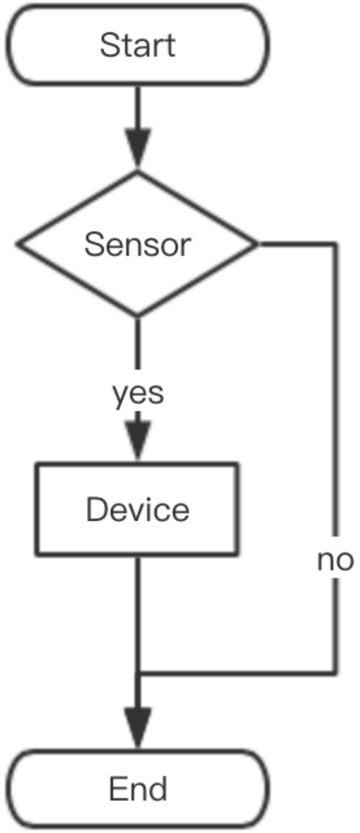
Advisor: A/P Wei-Ngan Chin

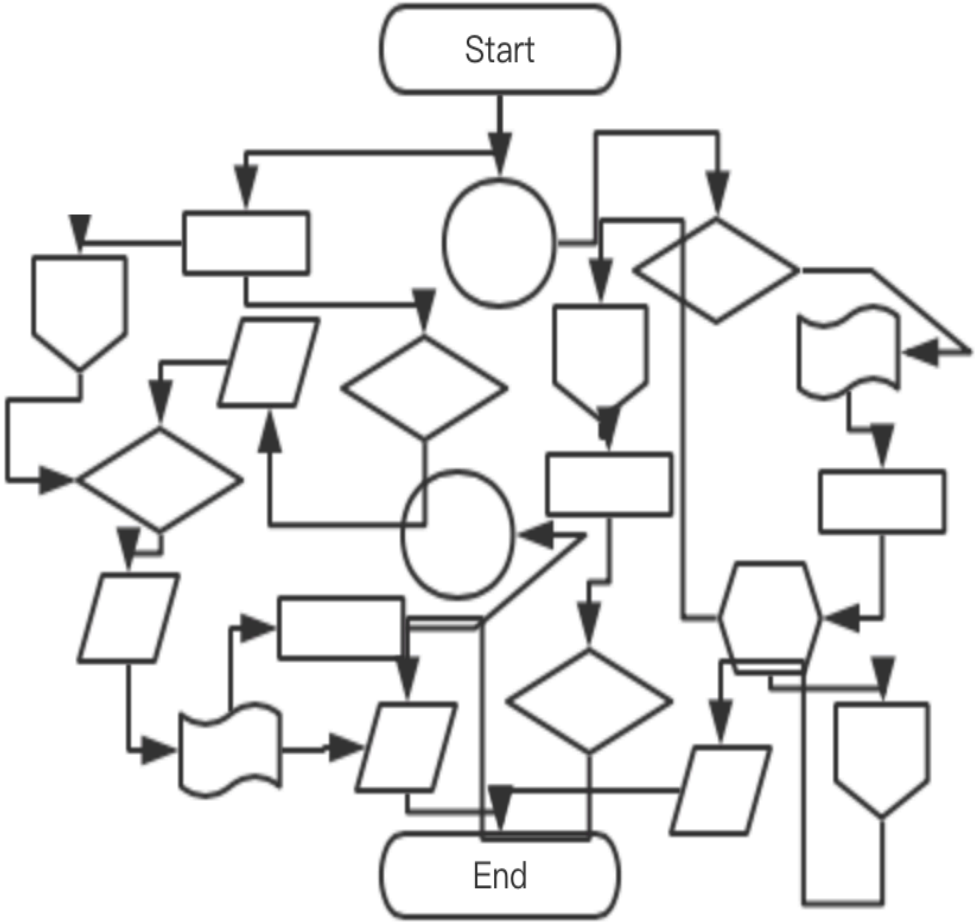
Internet of Things (IoT)

Internet of Things (IoT)

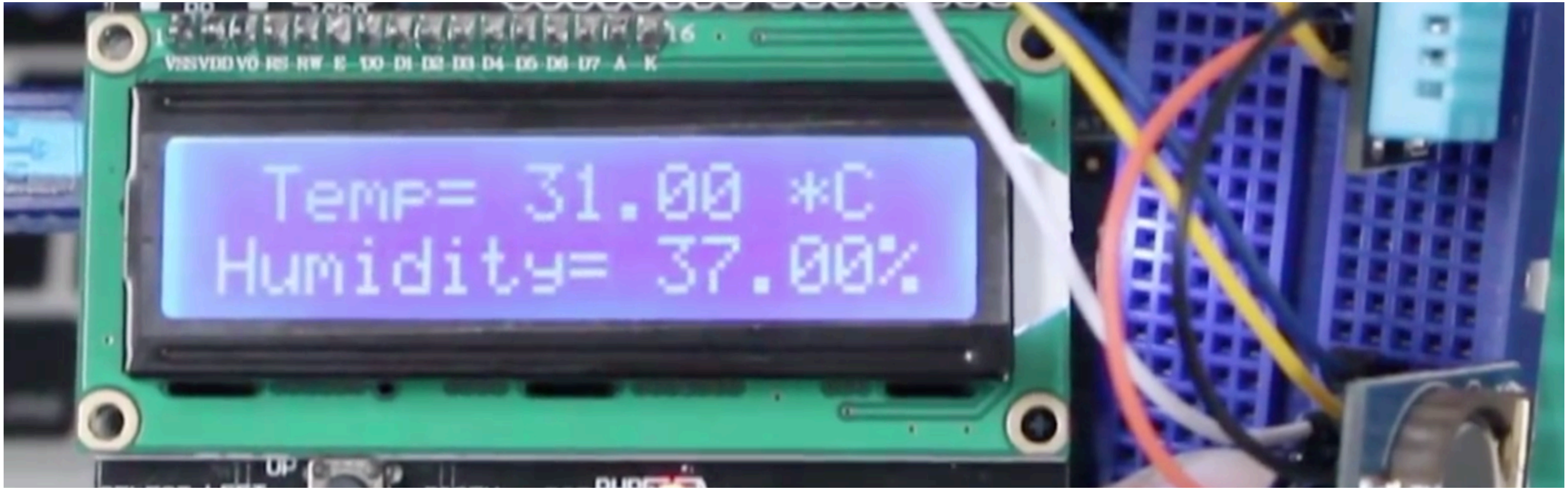


Internet of Things (IoT)





Motivation Example — LCD



Requirement: to show the current temperature and a clock.

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;
}
```

GPIO library
(General-purpose input/output)

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;
}
```

```
#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);
}
```

To initialize GPIO

1. Read temperature

2. Get current time

3. Show on the 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;
}
```

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

```
#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.

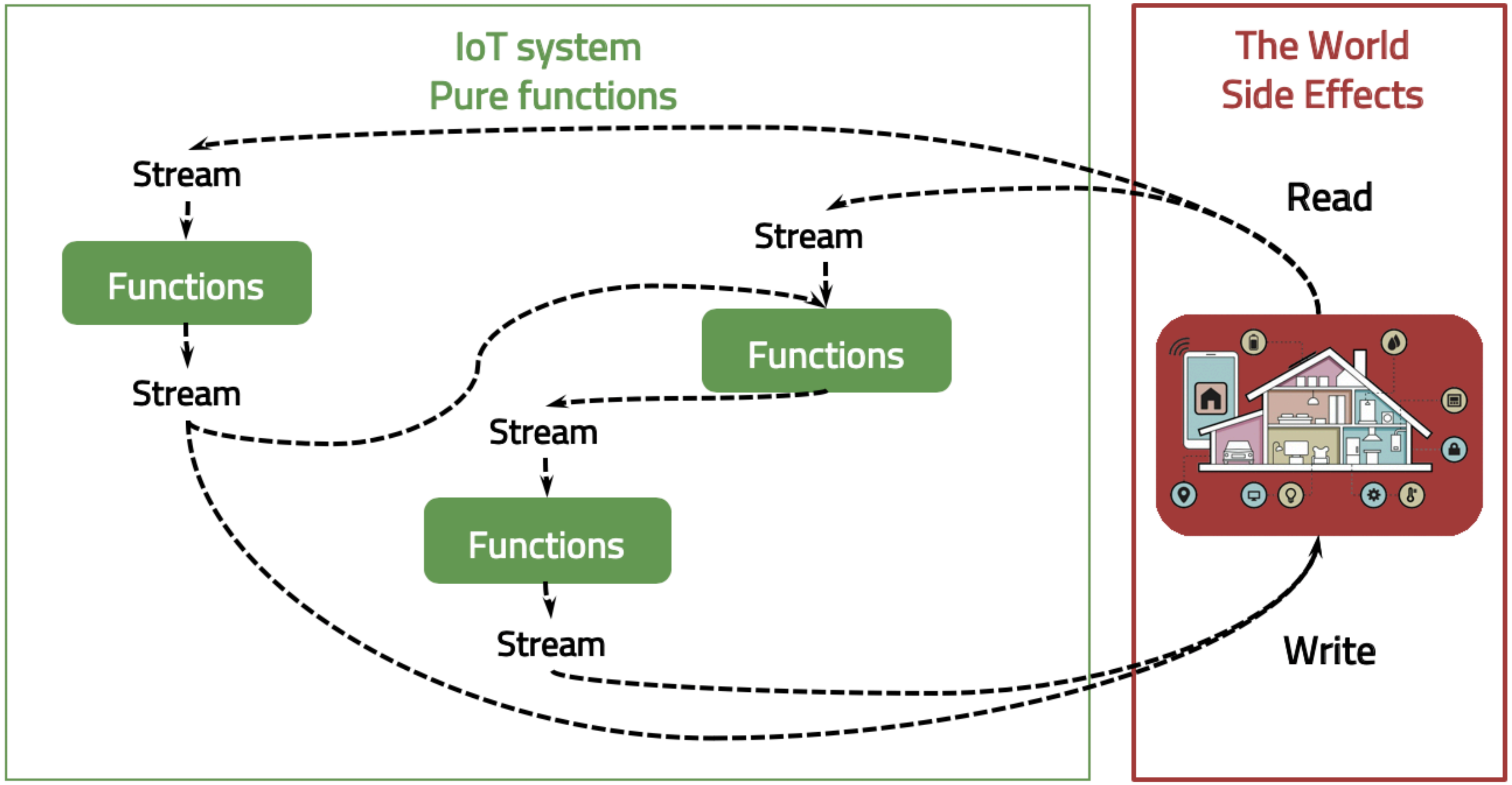
```c
#include <wiringPi.h>
#include <time.h>
#include <string.h>

int main() {
        if (wiringPiSetup() == -1) {

        }
        set
        while(1) loop();
        return 0;
}
```

```c
#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);
}
```

Functional Reactive IoT Programming !

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

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

port number

- Evn.temperature :: Int -> Signal Float

- clock :: Signal (Int, Int, Int)

Day : Hour : Second

- lcd :: Int -> Signal String -> IO ()

Higher-order functions !

High-order functions — Transformation

- lift :: (a -> b) -> Signal a -> Signal b

```
data :: Signal String
data = lift  floatToStr  (Evn.temperature 0)
```

port number

High-order functions — Transformation

- lift :: (a -> b) -> Signal a -> Signal b

Merge

- lift_2 :: (a -> b -> c) -> Signal a -> Signal b -> Signal c
- lift_n …

port number

```
data :: Signal String
data = lift  floatToStr  (Evn.temperature 0)
```

High-order functions — State

Flod from the past

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

port number

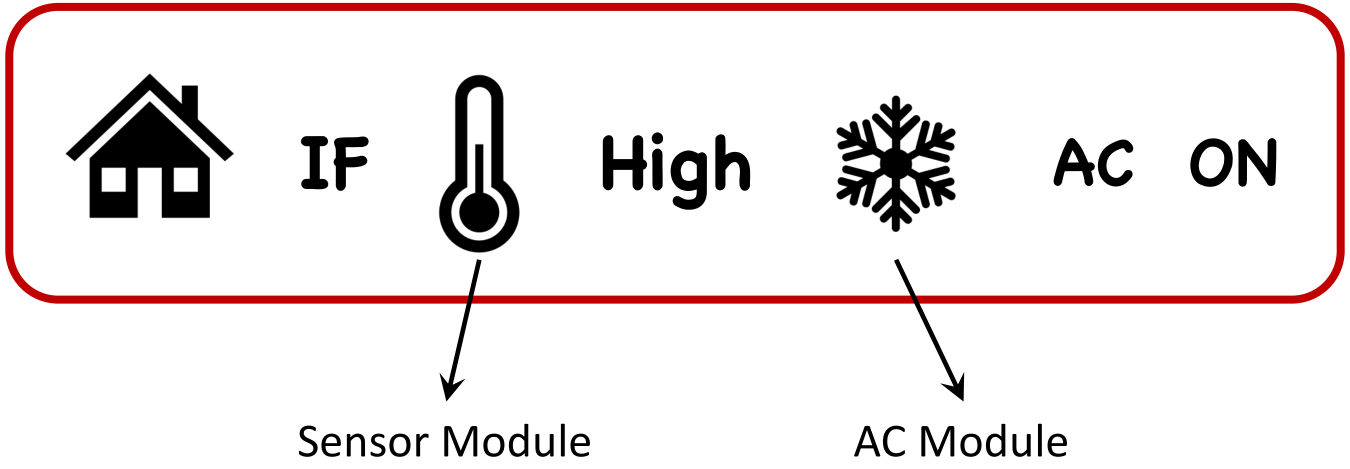
```
totalStep :: Siganl Int
totalStep = foldP  (\step count -> count + 1) 0 (Evn.motion 0)
```

Initialization of the accumulator

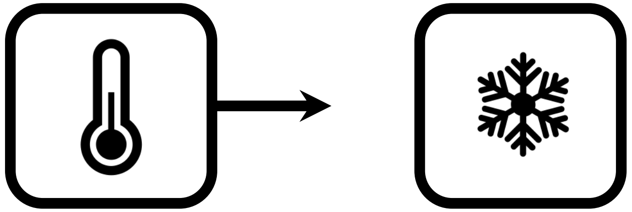
Functional Reactive IoT Programming

If the temperature rose too high,

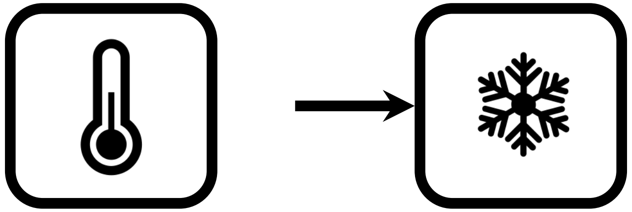
the air conditioner (AC) would be turned on automatically.



Functional Reactive IoT Programming

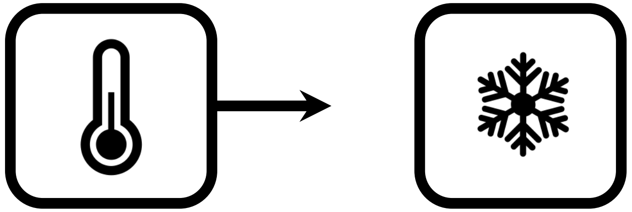


Passive

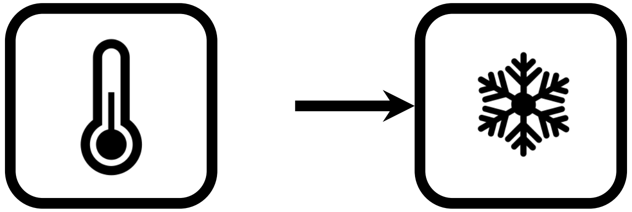


Reactive

Functional Reactive IoT Programming



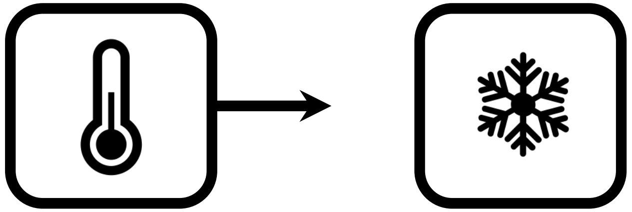
Passive



Reactive

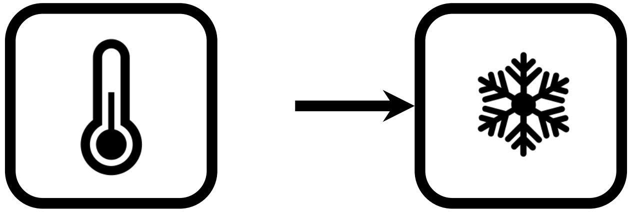
- Update method is defined in the **Sensor** module
- Remote setters and updates
- **AC** has no awareness on the dependence

Functional Reactive IoT Programming



Passive

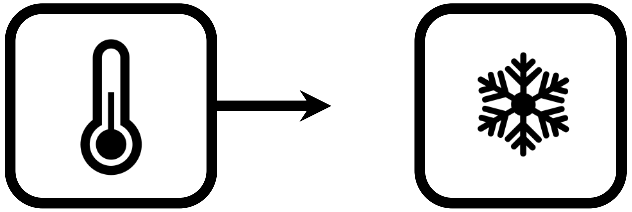
- Update method is defined in the **Sensor** module
- Remote setters and updates
- **AC** has no awareness on the dependence



Reactive

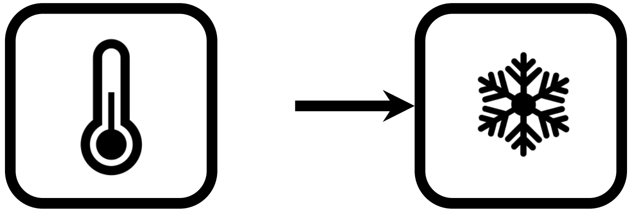
- Update method is defined in the **AC** module
- Events, observations and self-updates
- Easy to track/add dependencies on **AC** module

Functional Reactive IoT Programming



Passive

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



Reactive

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

"How does this module work?"

Revisit: Requirement:

to show the current temperature and a clock. **Use Friot!**

```
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)

                  ]
```

More concise code (LOC: 40 VS 9)

easy to read and easy to write

Revisit: Requirement:

to show the current temperature and a clock. **Use Friot!**

```
import Rpi
import Env
import Time

show :: Signal String
show = lift_2 (,) (Env.temprature 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)
                 ]
```

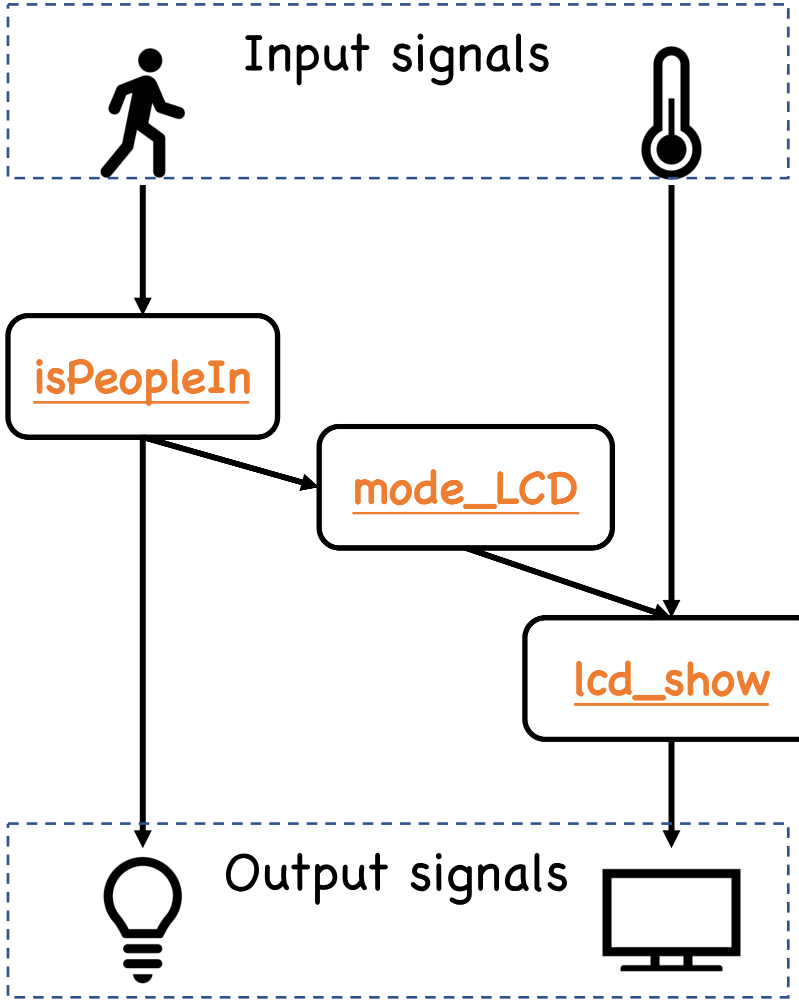
Adding a new device:
A blinking LED

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

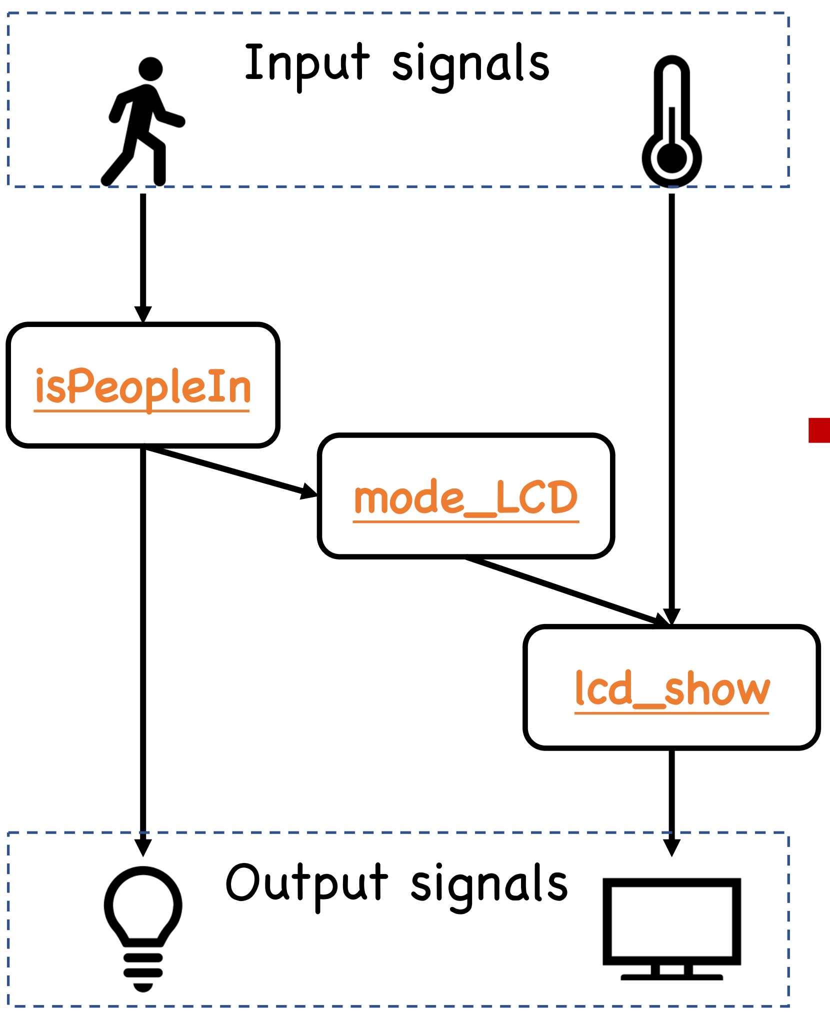
easy to read and easy to write

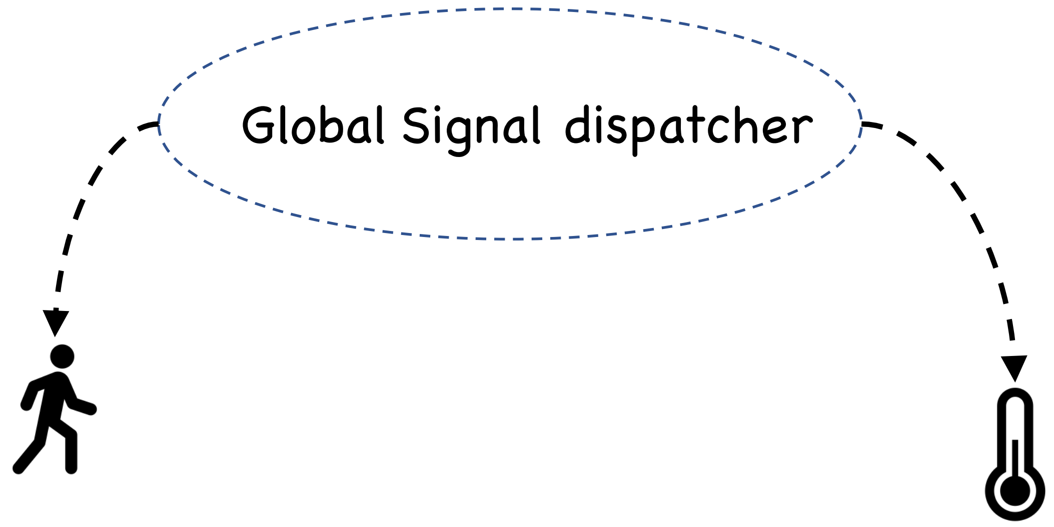
Signal Graph Transformation — multithreads

```
import Rpi
import Env

helper :: Signal 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) ]
```

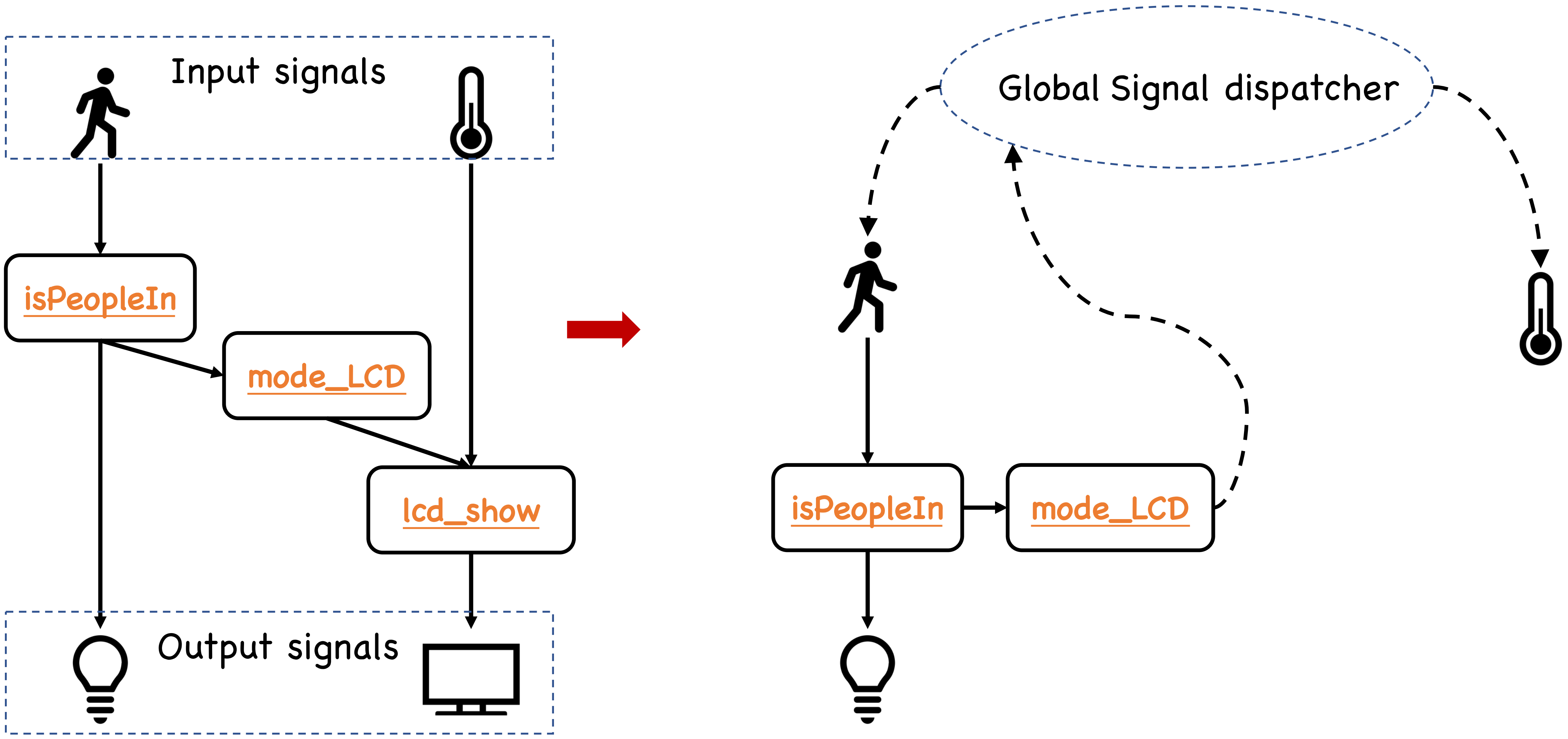


Signal Graph Transformation — multithreads

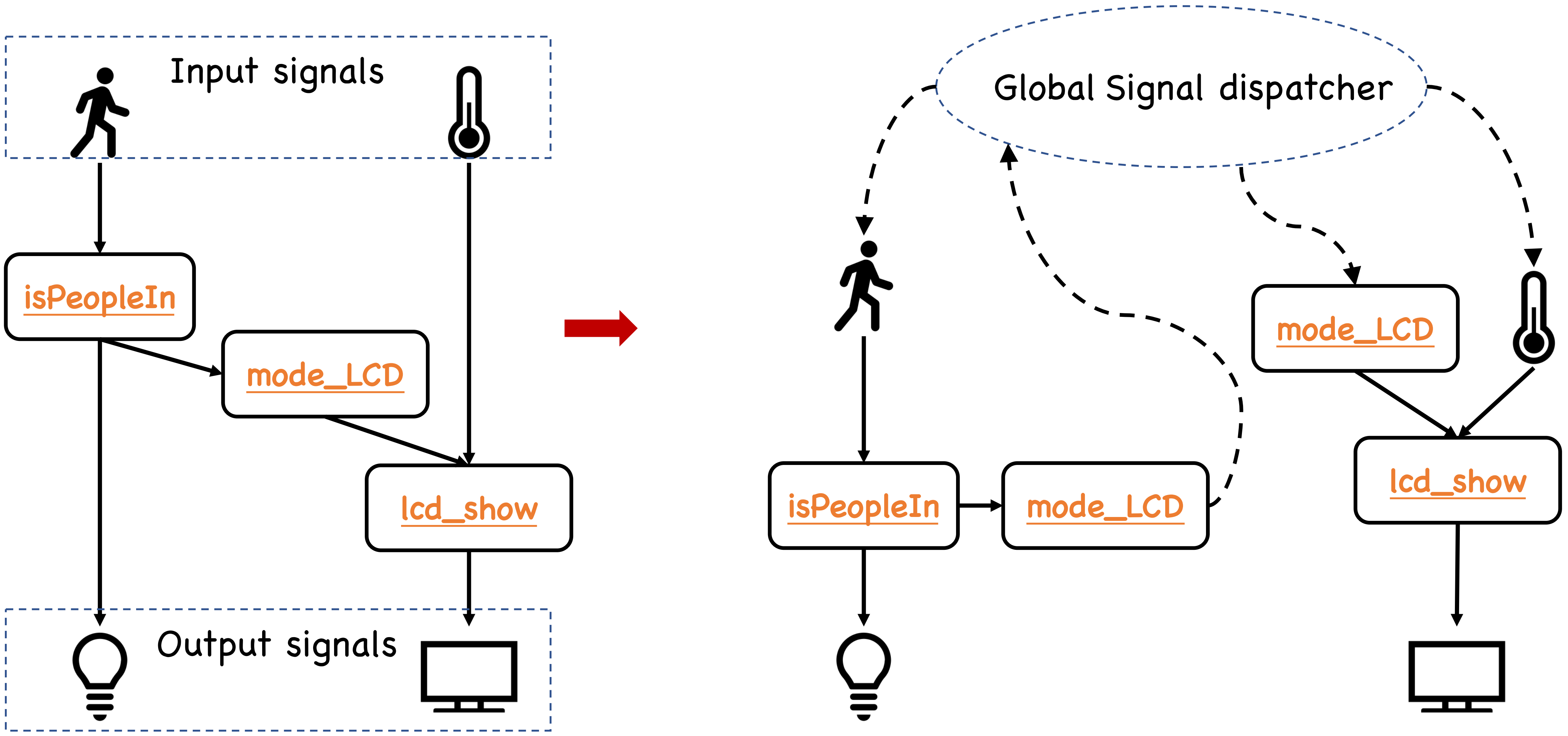




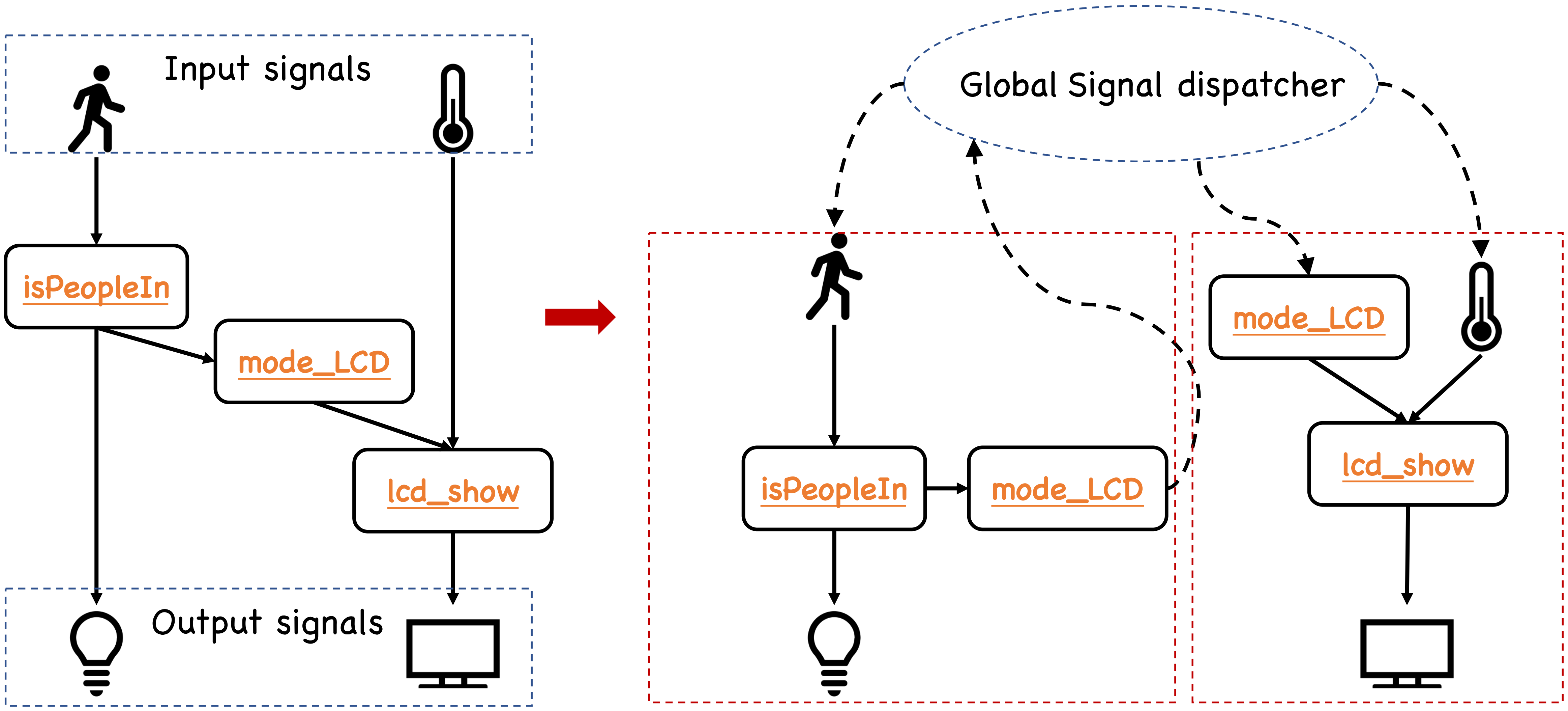
Signal Graph Transformation — multithreads



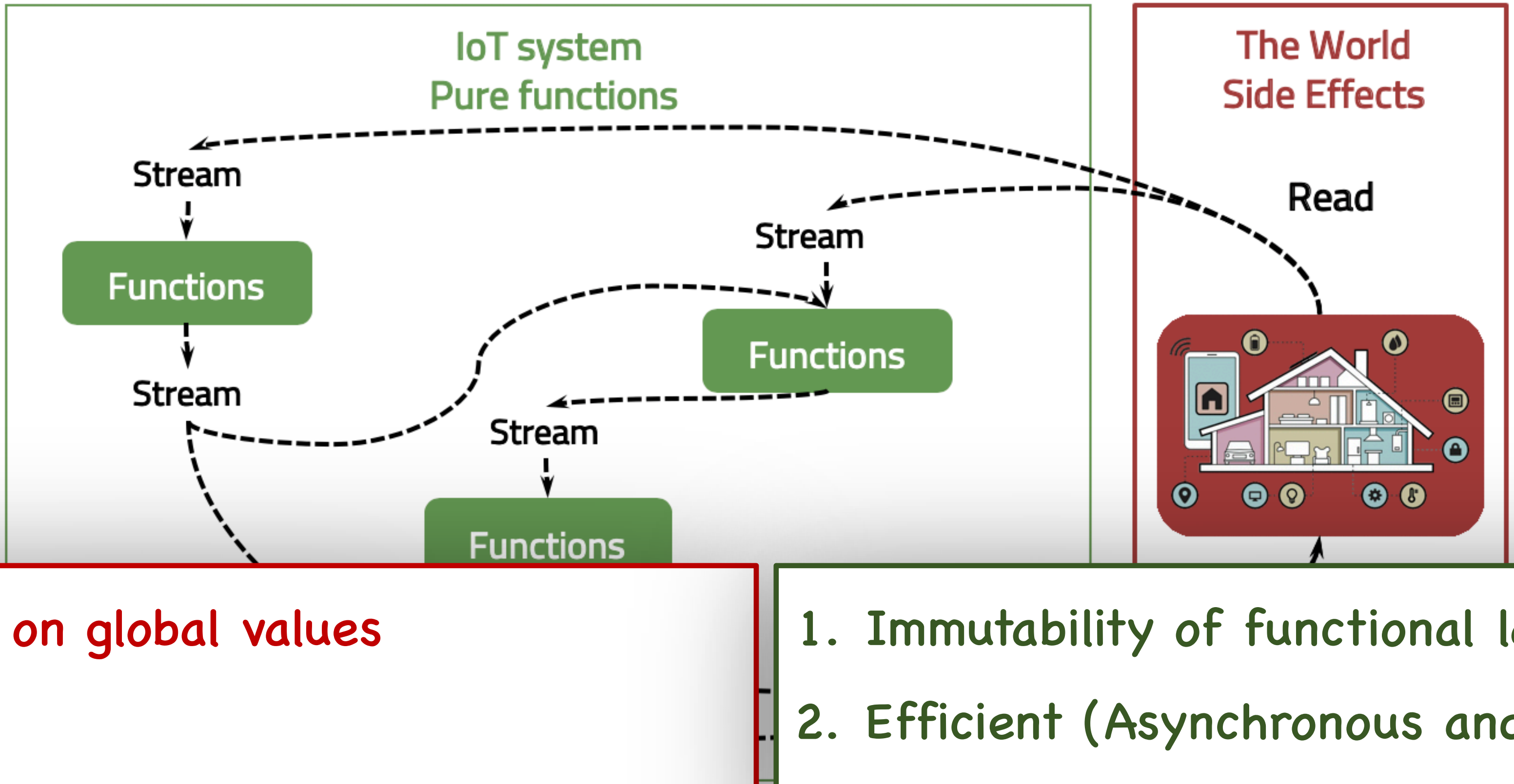
Signal Graph Transformation — multithreads



Signal Graph Transformation — multithreads



Signal Stream Graphs — Benefits



1. High reliance on global 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; specifically with time dependent behaviors, being able to be formal verified.
- Efficient (Asynchronous)

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; specifically with time dependent behaviors, being able to be formal verified.
- Efficient (Asynchronous)

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

Thanks a lot for your attention!

Co

- Fr 

T control systems
- Hc
- En reads)
- Sh fically with time
  de al verified.
- Ef

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

Thanks a lot for your attention!

Formal Verification — Refinement Types

- n :: { v : Int | v ≥ 0 } basic type
- max :: x: Int ⟶ y: Int ⟶ { v : Int | x ≤ v ∧ y ≤ v } function type
- xs :: { v: List Nat } polymorphic datatype

```
Evn.temperature :: Signal Int  -> Signal Float

Evn.temperature :: Signal {v: Int | v ≤ 20 ∧ v ≥ 0}  -> Signal {t: Float | t >10}
```