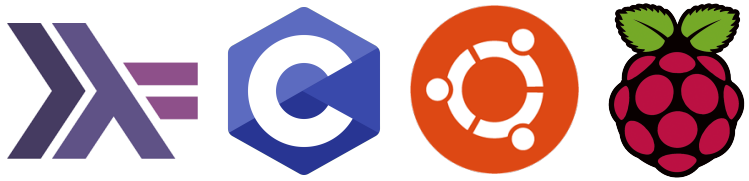

# Friot: Functional Reactive Abstraction for IoT Programming

The 31st symposium on Implementation and Application of Functional Languages

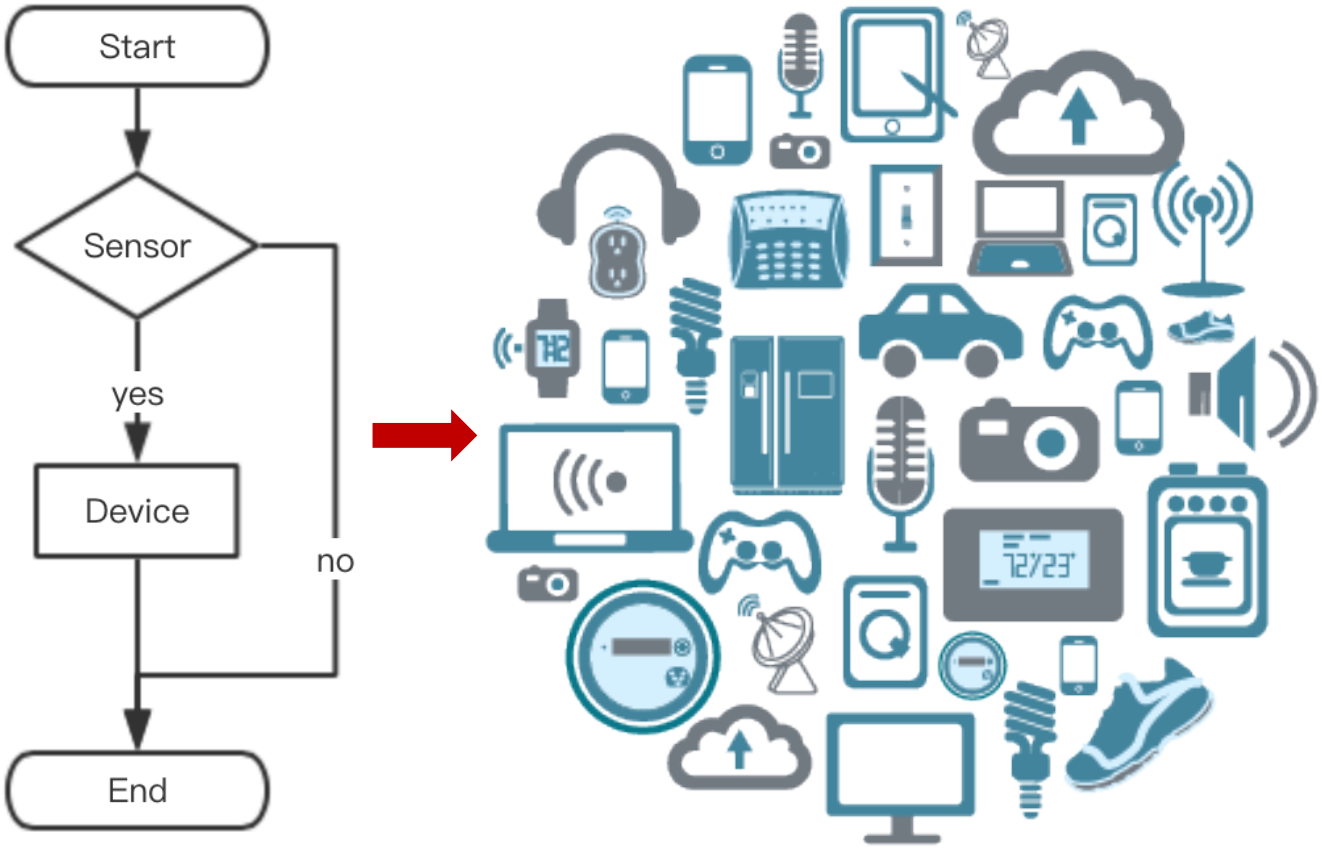
@ NUS Soc, Singapore

**Yahui Song** Andreea Costea Wei-Ngan Chin

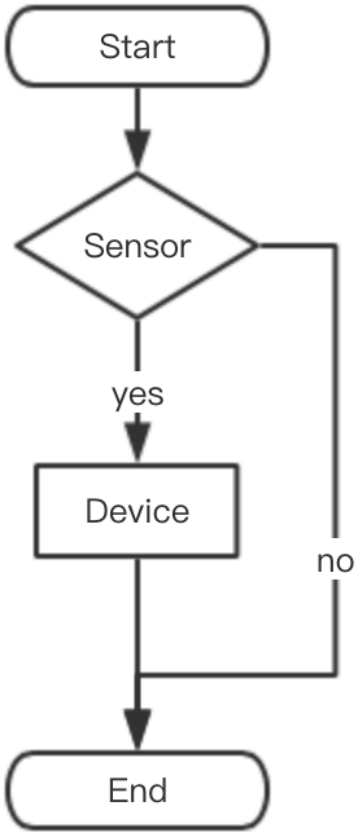


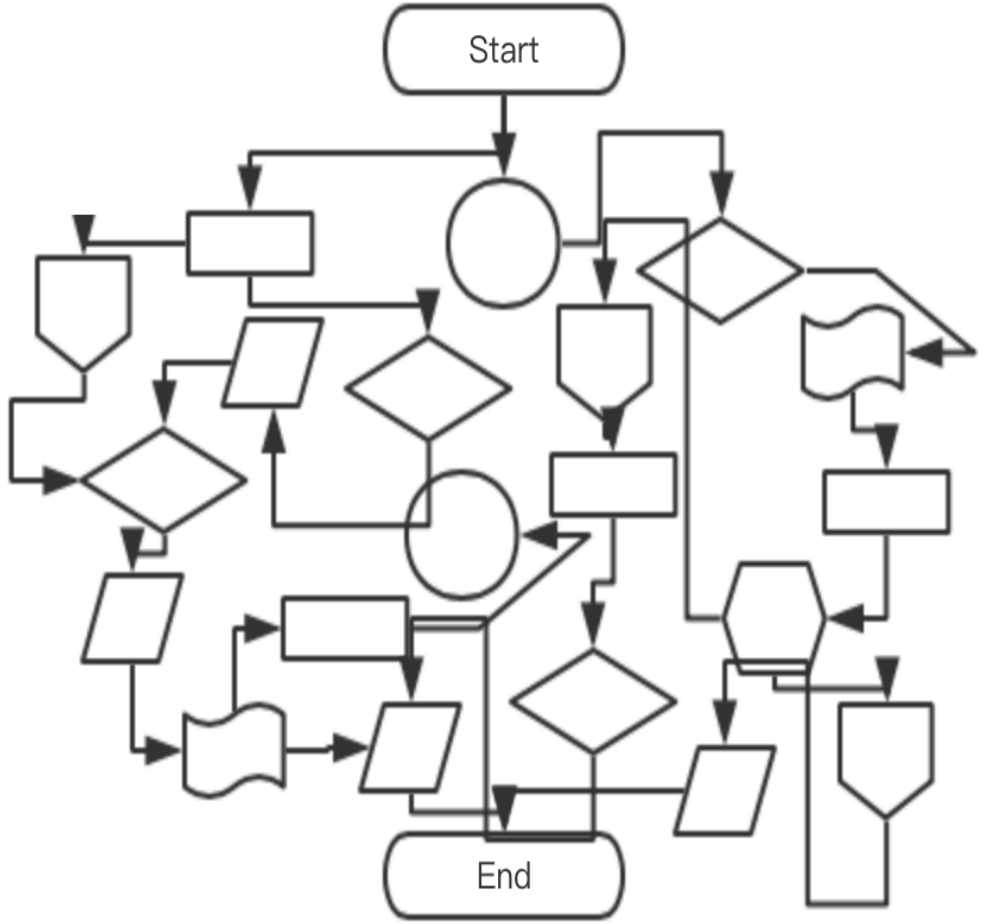
# Internet of Things (IoT)

# Internet of Things (IoT)

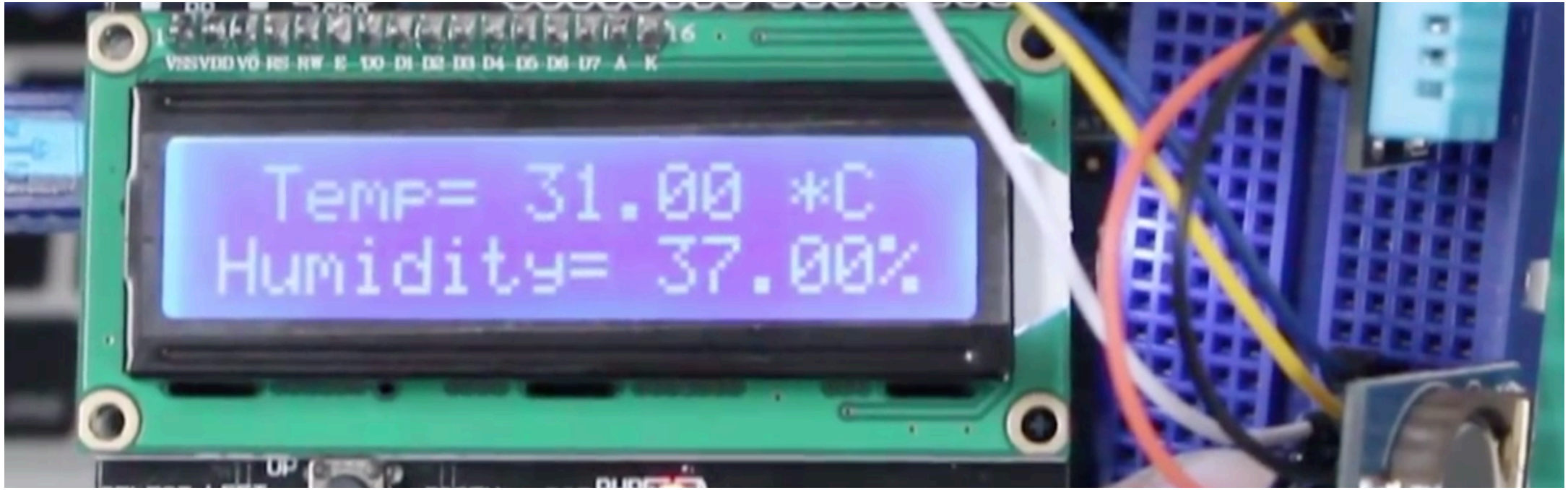


# Internet of Things (IoT)



# Motivation Example — LCD




Requirement: to show the current temperature and a clock.

# 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 mutable 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.

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

int main() {
        if (wiringPiSetup() == -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);
}
```

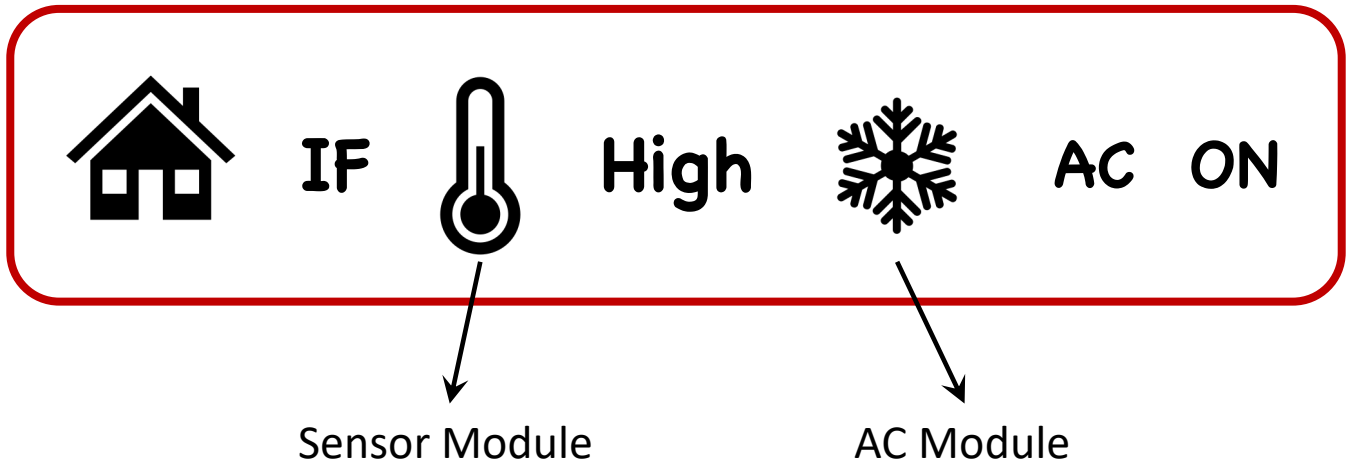
**Functional Reactive IoT Programming !**

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

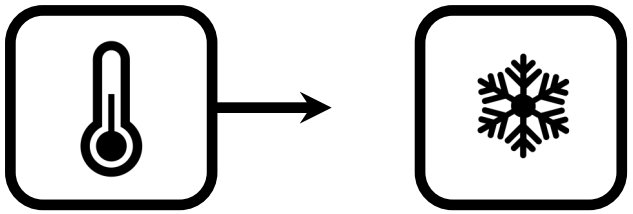
# Functional Reactive IoT Programming

If the temperature rose too high,

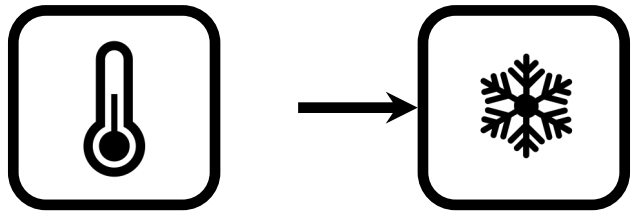
the air conditioner (AC) would be turned on automatically.



# Functional Reactive IoT Programming



**Passive**

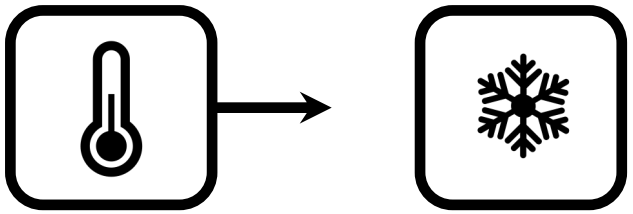


**Reactive**

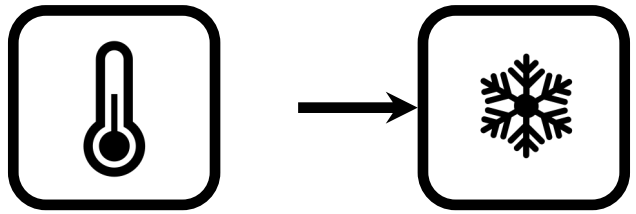
- **Sensor** owns the Update method
- Remote setters and updates
- **AC** has no awareness on the dependence

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

# Functional Reactive IoT Programming



**Passive**



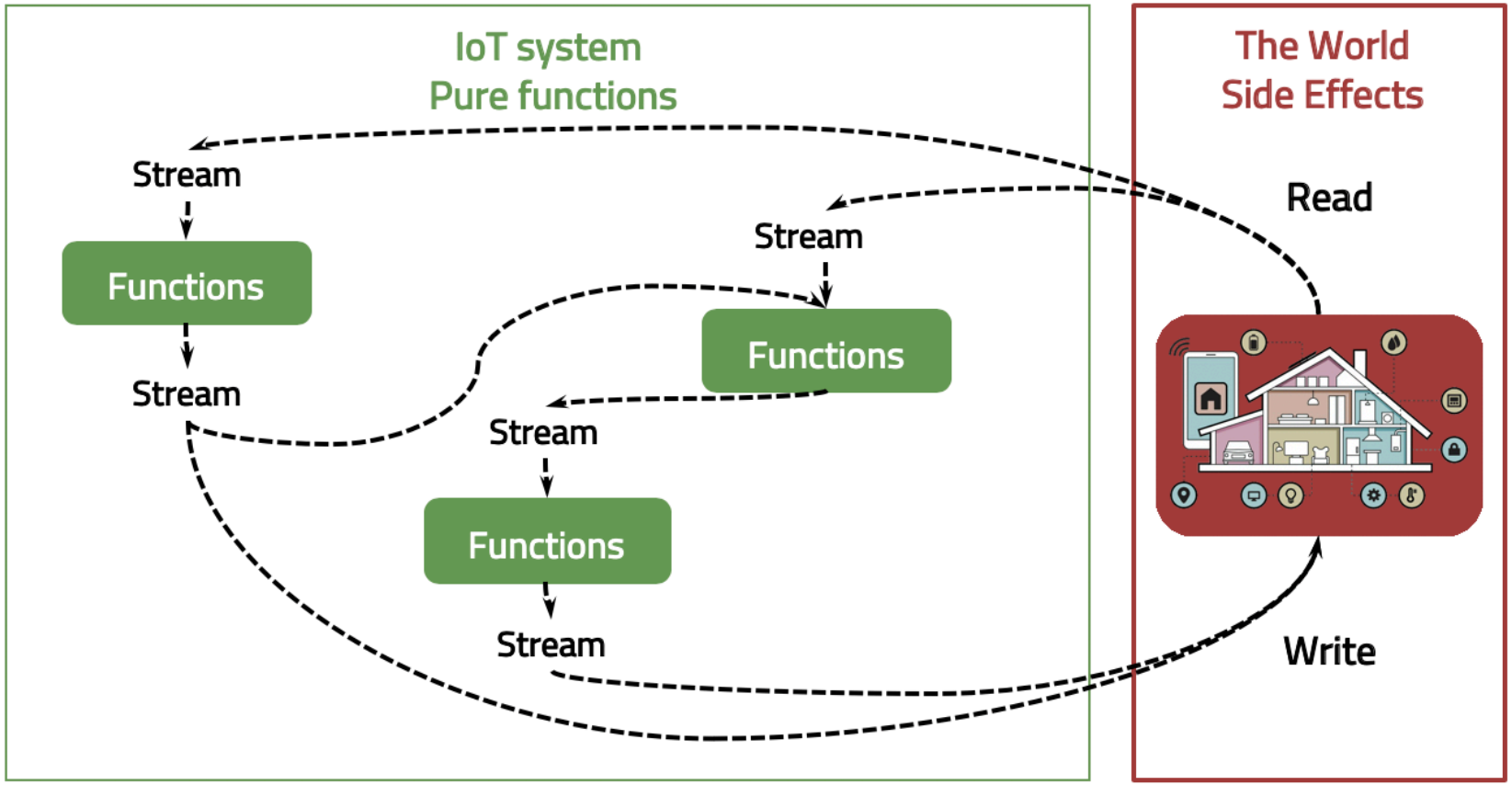
**Reactive**

- **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:**

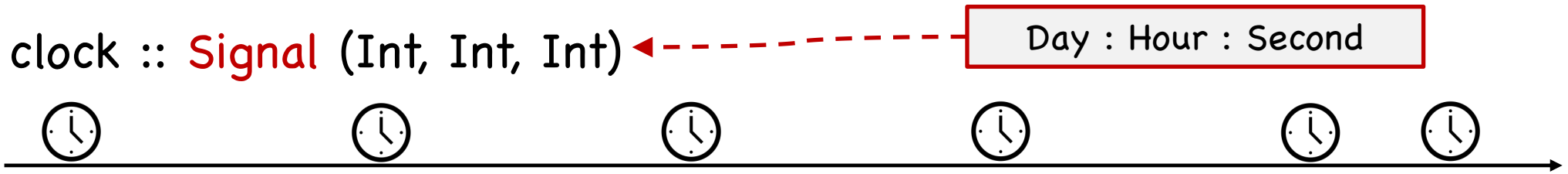
- Static
- Multi-threaded
- Asynchronous by default

Pushing the side effect to the edge of the system !

# Input / Output are Signals

- Env.temperature :: Int -> Signal Float


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




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

# Input / Output are Signals

port number

- Env.temperature :: Int -> Signal Float

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

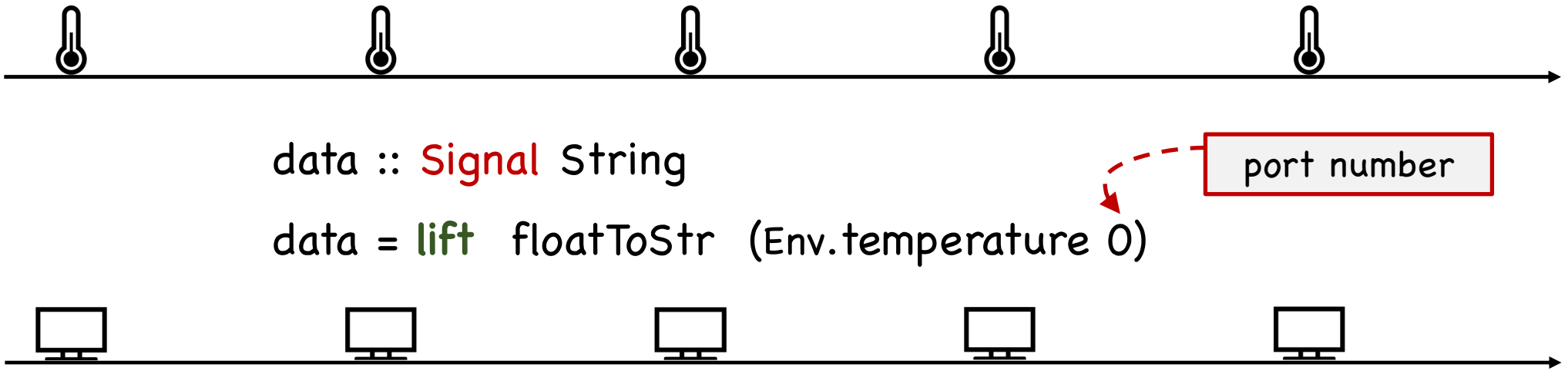
Day : Hour : Second

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

Higher-order functions !

# High-order functions — Transformation

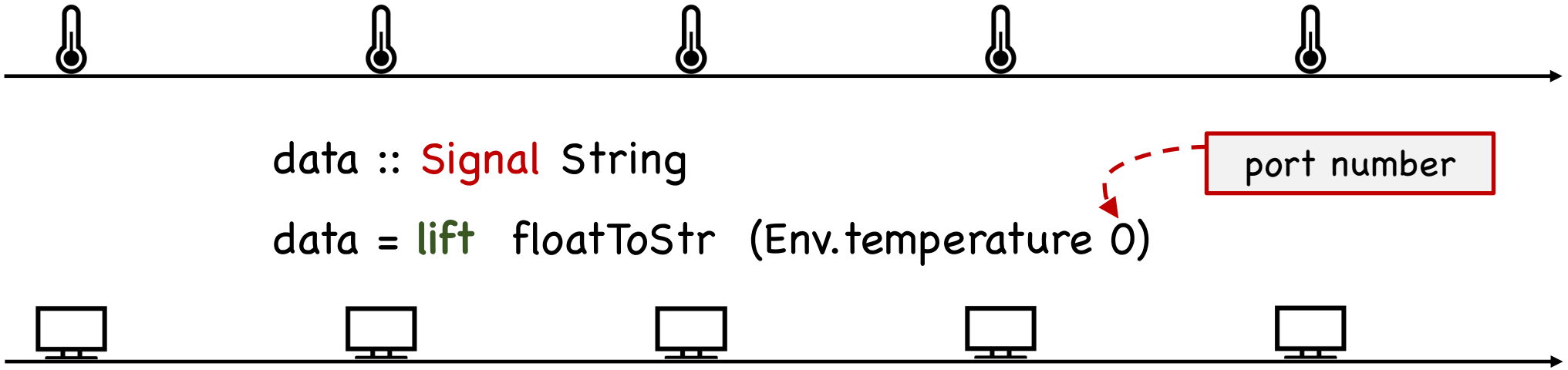
- `lift :: (a -> b) -> Signal a -> Signal b`



# High-order functions — Transformation

- `lift :: (a -> b) -> Signal a -> Signal b`
- `lift_2 :: (a -> b -> c) -> Signal a -> Signal b -> Signal c`
- `lift_n ...`

Merge



# High-order functions — State

Flod from the past

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

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

port number

Initialization of the count accumulator

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

# ML-like syntax for Friot

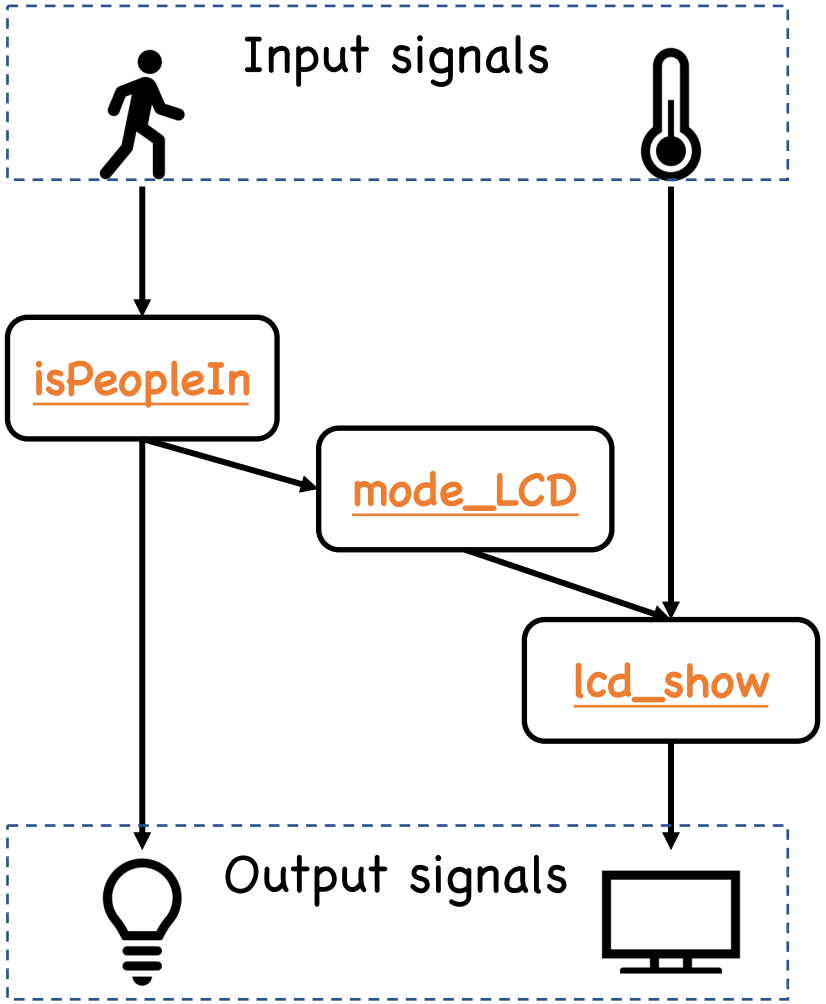
Type guided program transformation

$$
\begin{array}{rrcl}
(\textit{Basic Types}) & \tau & ::= & \textit{Unit} \mid \textit{Int} \mid \textit{Bool} \mid \textit{String} \mid \tau_1 \to \tau_2 \\
(\textit{Signal Types}) & \sigma & ::= & \textit{Signal}\ \tau \mid \tau \to \sigma \mid \sigma_1 \to \sigma_2 \\
(\textit{Types}) & t & ::= & \tau \mid \sigma \\
(\textit{Constants}) & c & ::= & () \mid n \mid b \mid \textit{string} \\
(\textit{Expressions}) & e & ::= & c \mid x \mid F \mid \lambda x.e \mid e_1 \oplus e_2 \mid e\ e_1 \\
 & & & \mid \textbf{if}\ (e_1; e_2; e_3) \mid \textbf{let}\ (e_1; x.e_2) \mid \textbf{fold}\ e_1\ e_2\ e_3 \\
 & & & \mid \textbf{lift\_n}\ e\ e_1\ \ldots\ e_n \mid \textbf{sync}\ e \mid \textbf{prior}\ n\ e \\
(\textit{Program}) & \mathcal{P} & ::= & \mathcal{P} \cup \{F\ \overline{x} = e\} \mid \varnothing
\end{array}
$$

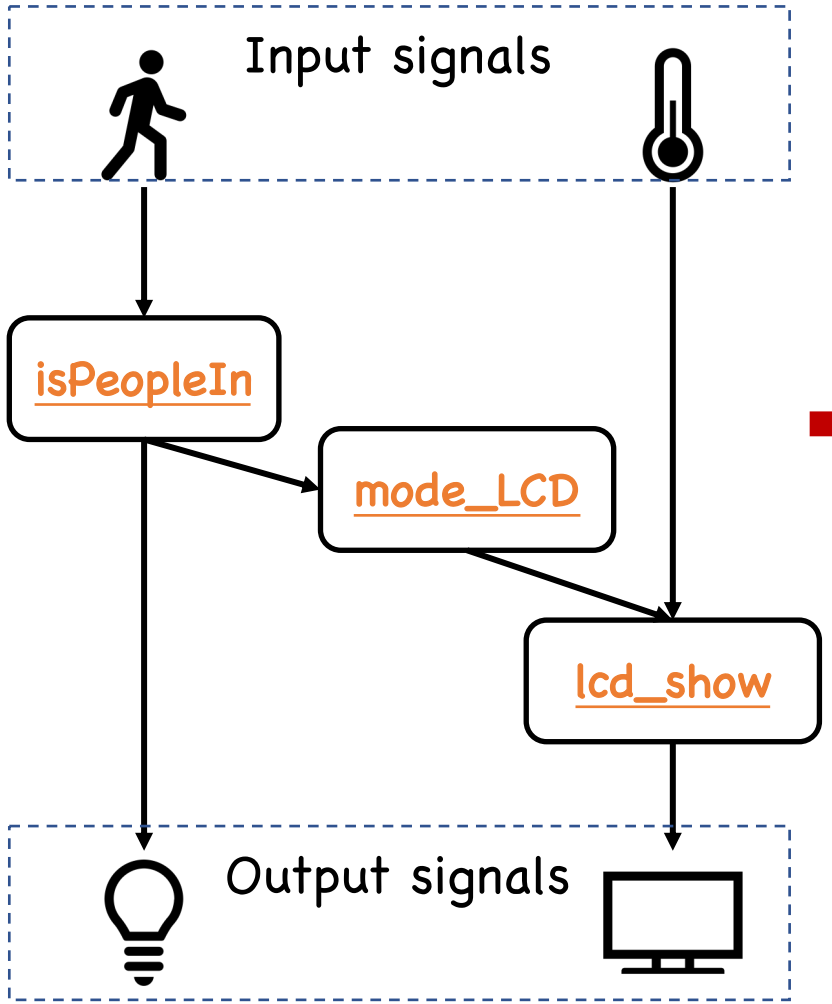
$$
n ::\in \mathbb{Z} \qquad b ::\in \textbf{Bool} \qquad \textit{string} ::\in \textbf{String} \qquad x, F ::\in \textbf{Var}
$$

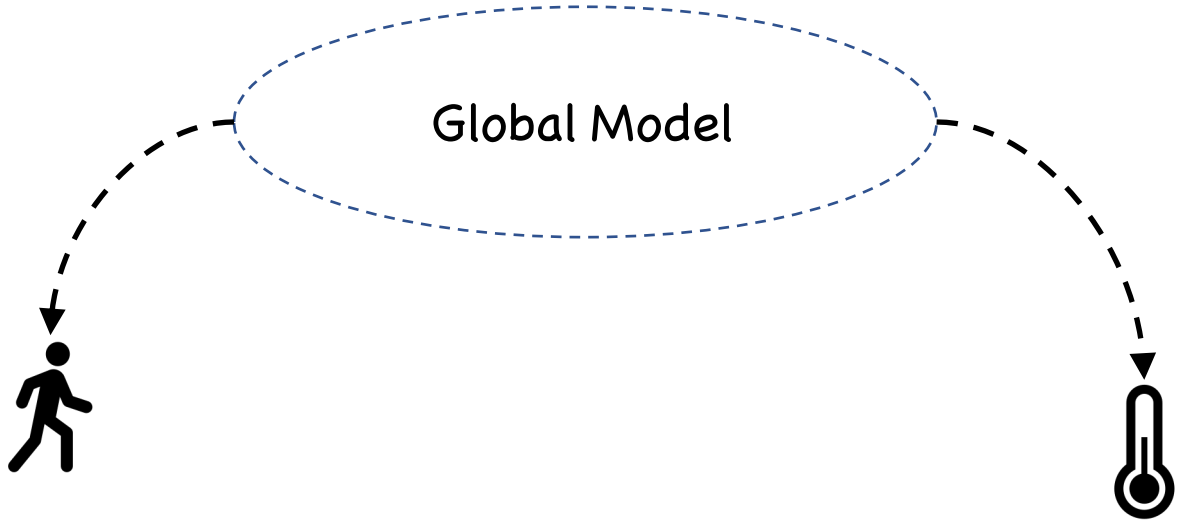
# Signal Graph Transformation — multithreads

```
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) ]
```

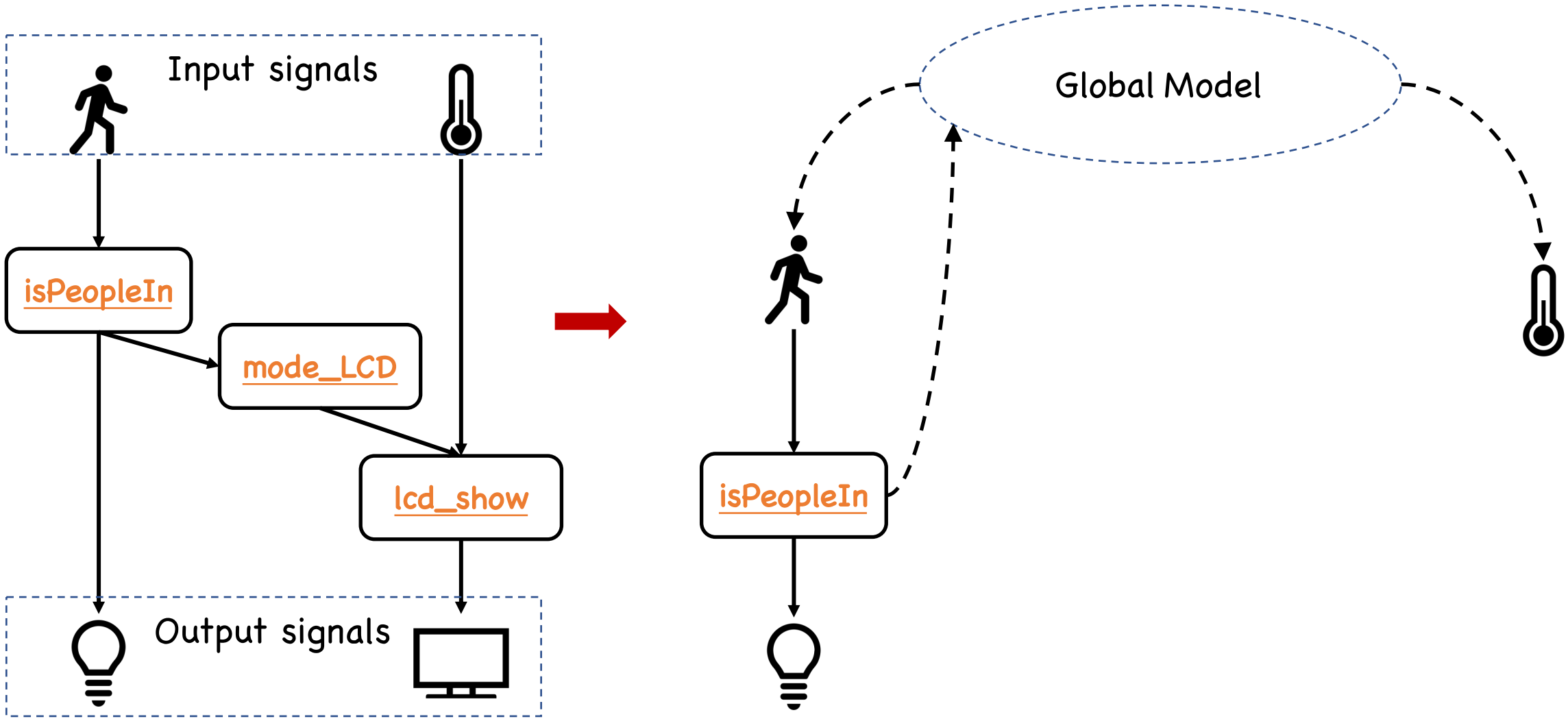


# Signal Graph Transformation — multithreads

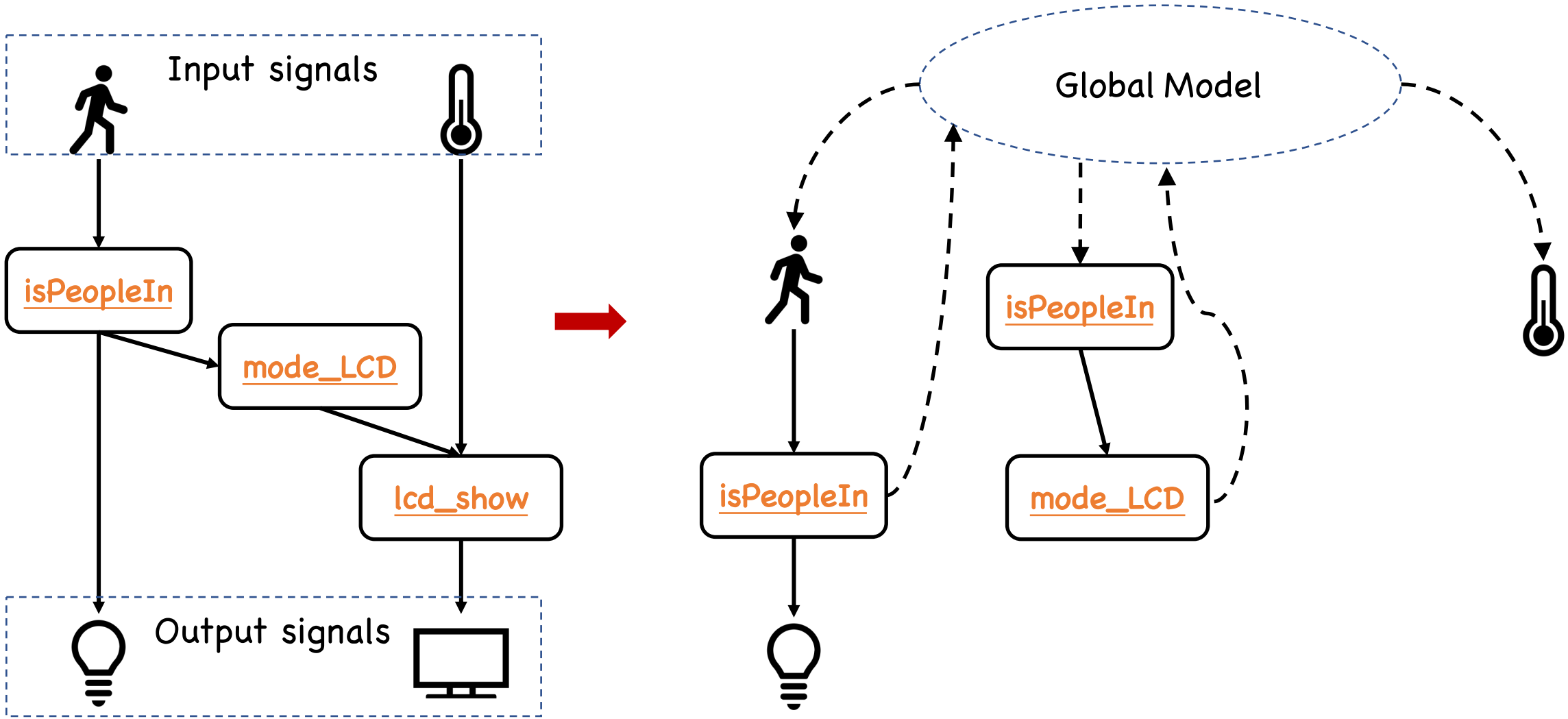


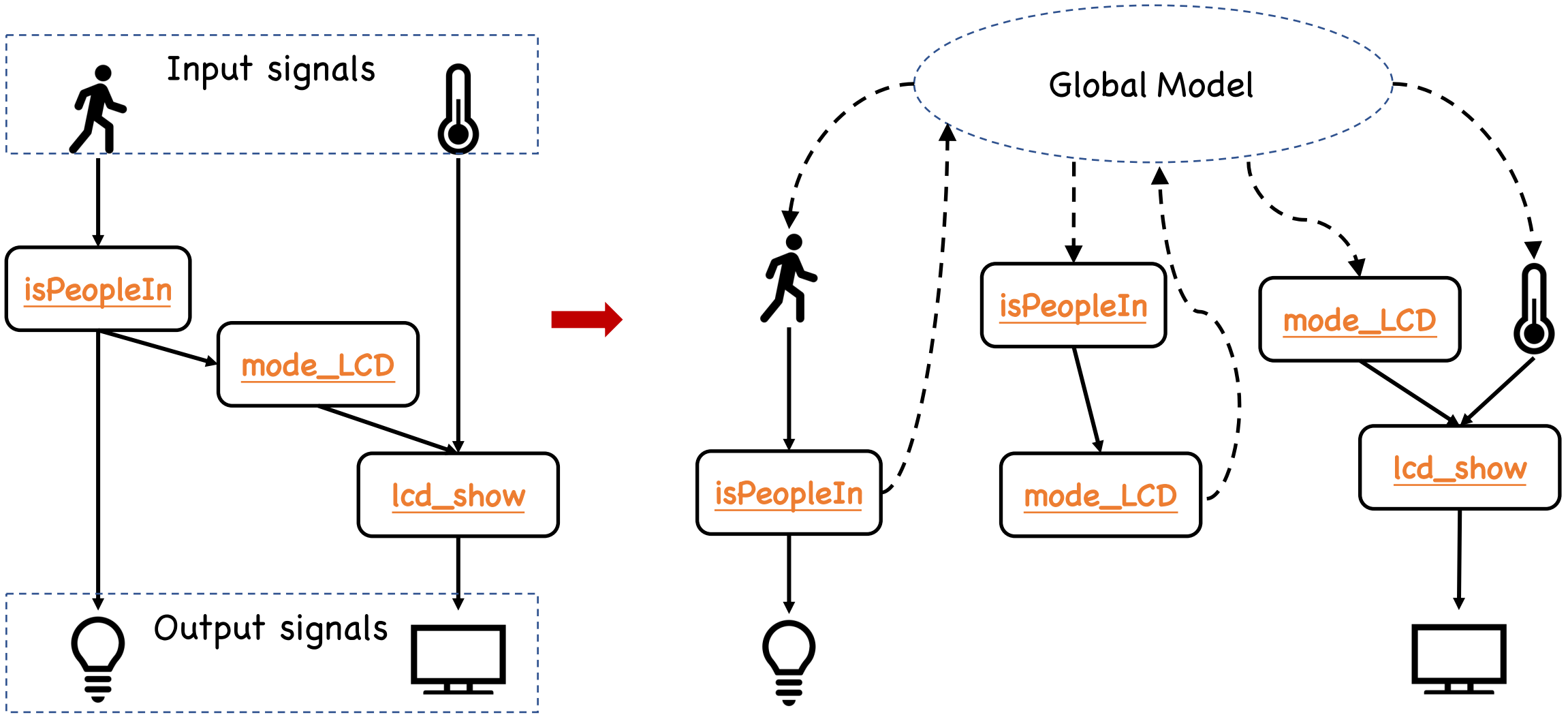
# Signal Graph Transformation — multithreads



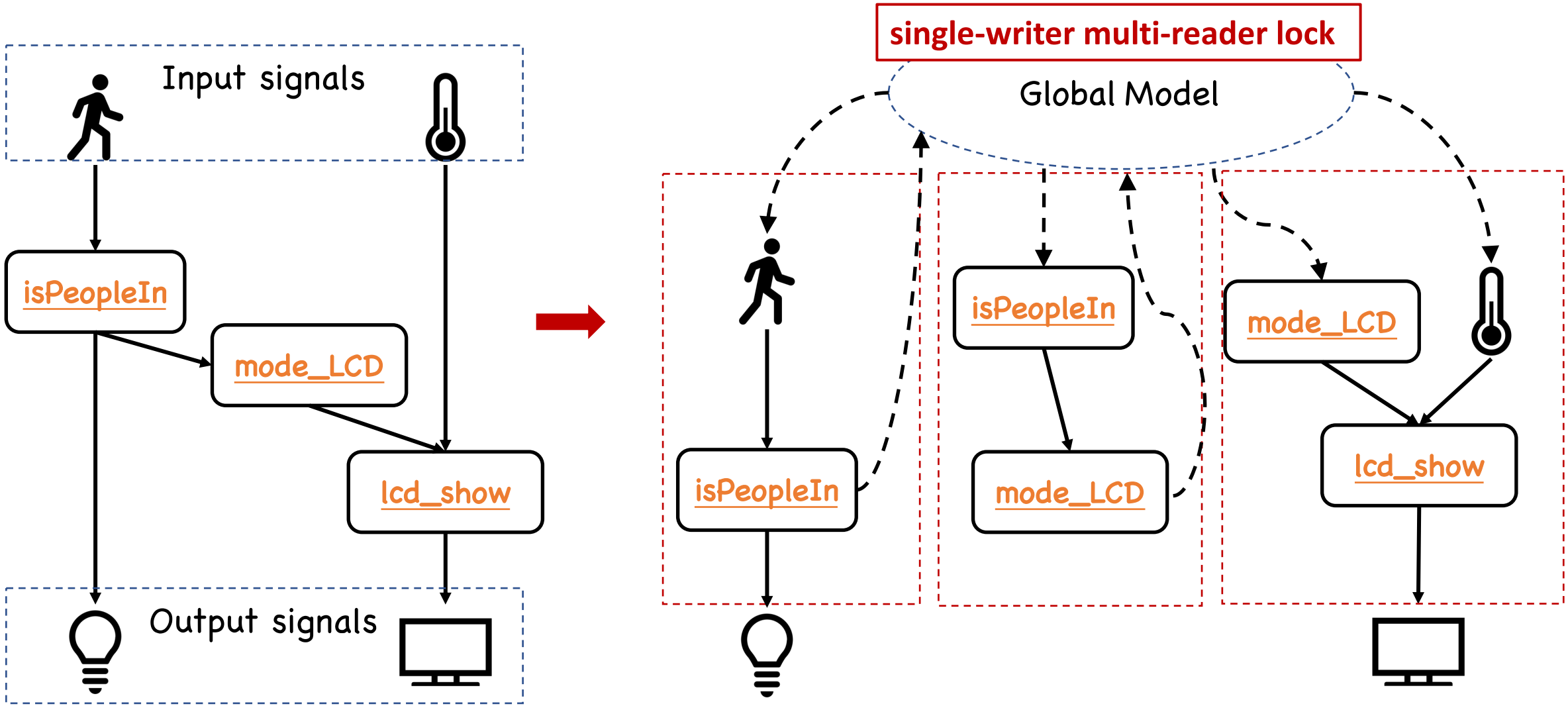
# Signal Graph Transformation — multithreads



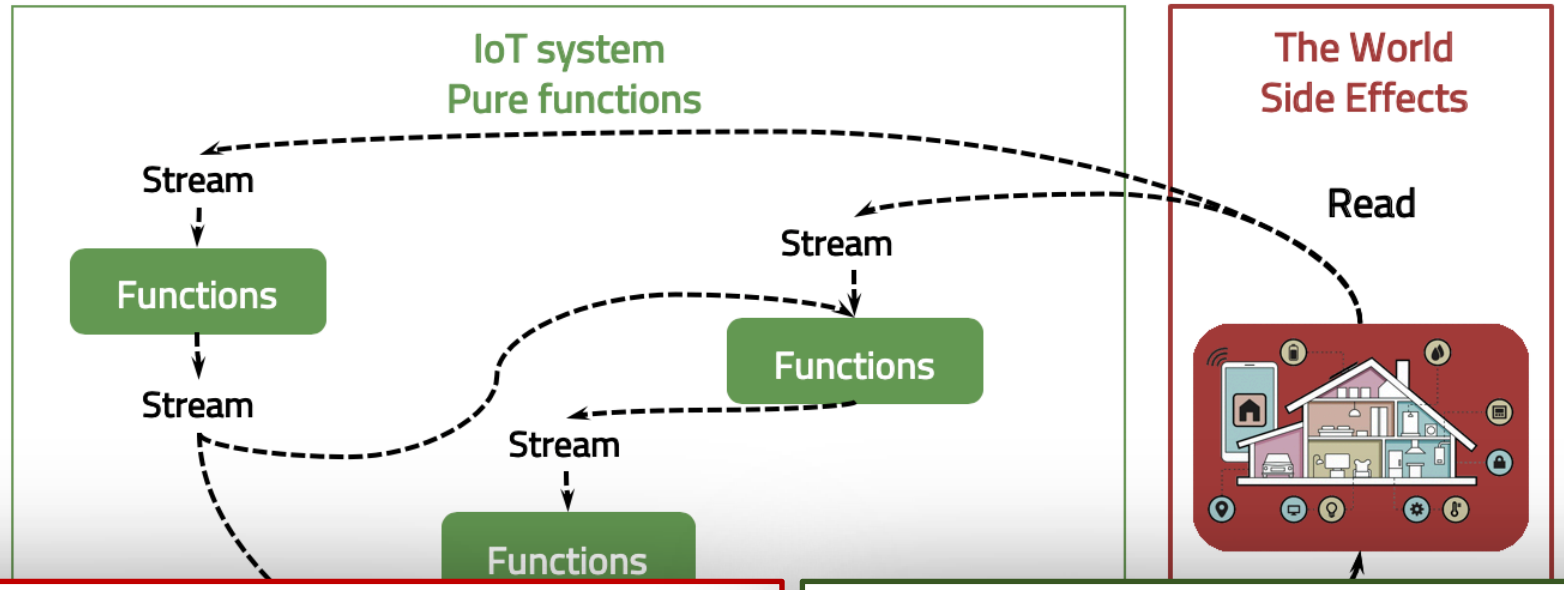
# Signal Graph Transformation — multithreads



# Signal Graph Transformation — multithreads



# 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!**