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

#10 (2018/11/15)

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

Time-Varying Values

- TVV α = Time → α
  - Function from time to data of type α
  - First class entities representing (continuously) changing data (of type α) over time
  - ex) tmp :: TVV float (temperature sensor value)
  - aka. Signal α, Behavior α
- Event α = [(Time, α)]
  - List of pairs of time and data of type α
  - First class entities representing discrete events of type a
  - ex) key :: Event Char (keyboard event)

Time/Space-Leaks

- Unrestricted access to a time varying values leads to time/space-leaks

- x :: TVV a
- The value of x at time t (i.e., x t) depends on the values of x in the interval [0, t).
- e.g., If x t is allowed to be accessed and
  - If x t is always calculated using { x t' | t' ∈ [0, t) }, it takes unexpectedly long time (time-leak).
  - If { x t' | t' ∈ [0, t) } is stored to calculate x t, it requires unexpectedly large memory (space-leak)

To Avoid Time/Space-Leaks

- Restricted representation of TVV α
  - TVV α as a (primitive) type other than Time → α
  - ex) Emfrp
    - Nodes (TVVs in Emfrp) are not first class values
    - Only @last can be used to access the past value of a node
- Arrowization
  - Abandon time-varying values themselves and use "functions" on time-varying values
  - Arrow is used to construct the "functions"
  - ex) Yampa (FRP library for Haskell)

Signal Functions (Signal Transformers)

- `SF a b` ≃ `Signal a -> Signal b`
  - Note: Here, `Signal a` is the type of time-varying values (corresponds to TVV a)
- `SF a b` is an abstract type
  - Operations on it provide a *disciplined* way to compose signals
  - The notion of *arrow* provides the discipline.

Arrow Operations (1)

- Lifting
  - `arr :: (a -> b) -> SF a b`
- Composition
  - `(>>>) :: SF a b -> SF b c -> SF a c`
    - Note: `f >>> g = (>>>) f g`

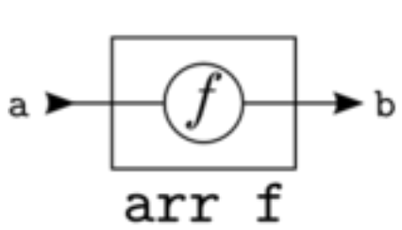
- The following equations hold
  - `arr f >>> arr g = arr (g . f)`
    - Note: `(g . f) x = g (f x)`
  - `(f >> g) >> h = f >> (g >> h)`

Arrow Operations (2)

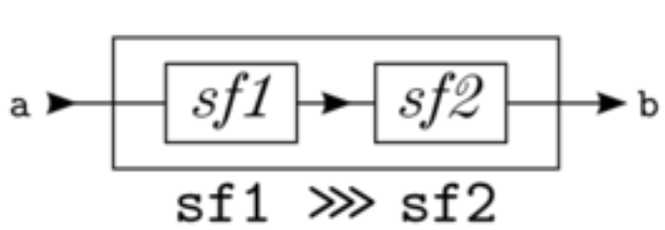
- `(&&&) :: SF a b -> SF a c -> SF a (b, c)`
  - `arr f &&& arr g = arr (f & g)`
    - where `(f & g) x = (f x, g x)`
- `(***) :: SF a b -> SF c d -> SF (a, c) (b, d)`
  - `f *** g = (arr fst >>> f) &&& (arr snd >>> g)`
    - where `fst (x, y) = x, snd (x, y) = y`
- `first :: SF a b -> SF (a, c) (b, c)`
  - `first f = f *** arr id`
    - where `id x = x`
- `second :: SF a b -> SF (c, a) (c, b)`
  - `second f = arr id *** f`

Arrow Operations Visualized

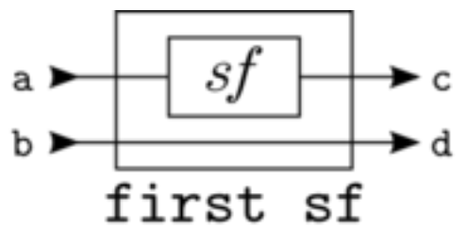
https://wiki.haskell.org/Yampa



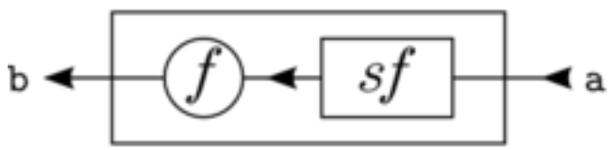
arr f



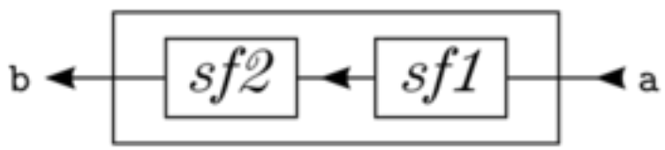
sf1 >>> sf2



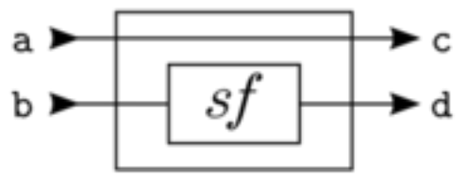
first sf



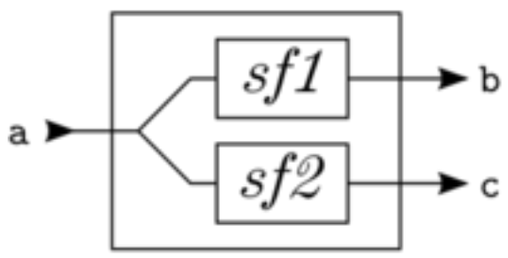
f ^<< sf



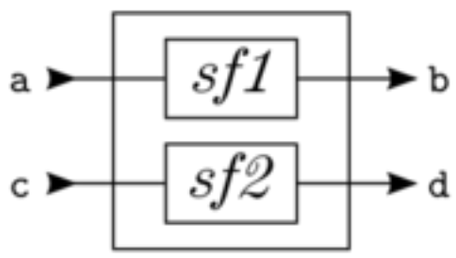
sf2 <<< sf1



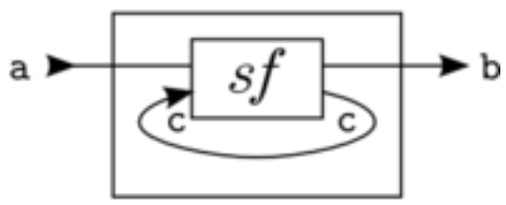
second sf



sf1 &&& sf2



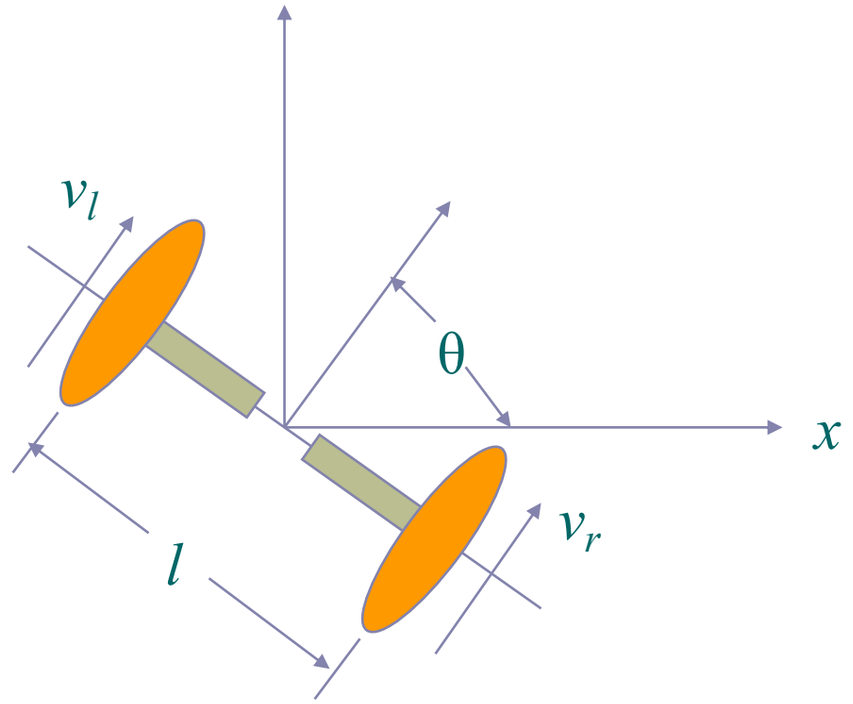
sf1 *** sf2



loop sf

Example

- Calculate the x position of a two-wheel robot



https://wiki.haskell.org/Yampa

$$x(t) = \int_0^t \frac{vr(t) + vl(t)}{2} \cos\theta(t) dt$$

```
vrSF, vlSF, thetaSF :: SF Input Float

x :: SF Input Float
x = let v = (vrSF &&& vlSF) >>> arr2 (+)
        t = thetaSF >>> arr cos
    in (v &&& t) >>> arr2 (*) >>> integral >>> arr (/ 2)
```

```
arr2 :: (a -> b -> c) -> SF (a, b) c
arr2 = arr . uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

Using Arrow Syntax

```
vrSF, vlSF, thetaSF :: SF Input Float

x :: SF Input Float
x = proc input -> do
      vr <- vrSF -< input
      vl <- vlSF -< input
      theta <- thetaSF -< input
      i <- integral -< (vr + vl) * cos theta
      returnA -< (i / 2)
```

Example in Emfrp

```
module Robot
in  vl, vr, theta : Float,
    t : Int
out x : Float
use Std

node intv = (vl + vr) * cos(theta) / 2.0

node init[0] x = x@last +
                 (intv - intv@last) * (t - t@last)
```

Procedural Abstraction

- Synchronous (reactive) programming languages
  - Lustre, Esterel
  - Ceu
- Provides procedural abstractions for reactive event-based behaviors

Ceu

- Prints "Hello World!" every 250ms.

```
loop do
    await 250ms;
    _printf("Hello World!\n");
end
```


- await
  - stops the execution until specified event (in this example, the expiration of 250ms timer) occurs

Input Events

```
input int MY_EVT;

loop do
    var int v = await MY_EVT;
    _printf("MY_EVT=%d\n", v);
    if v == 0 then
        break;  // escapes the loop when v==0
    end
end
escape 0;
```

- MY_EVT
  - External event identifier (should be in uppercase)

Blinking an LED

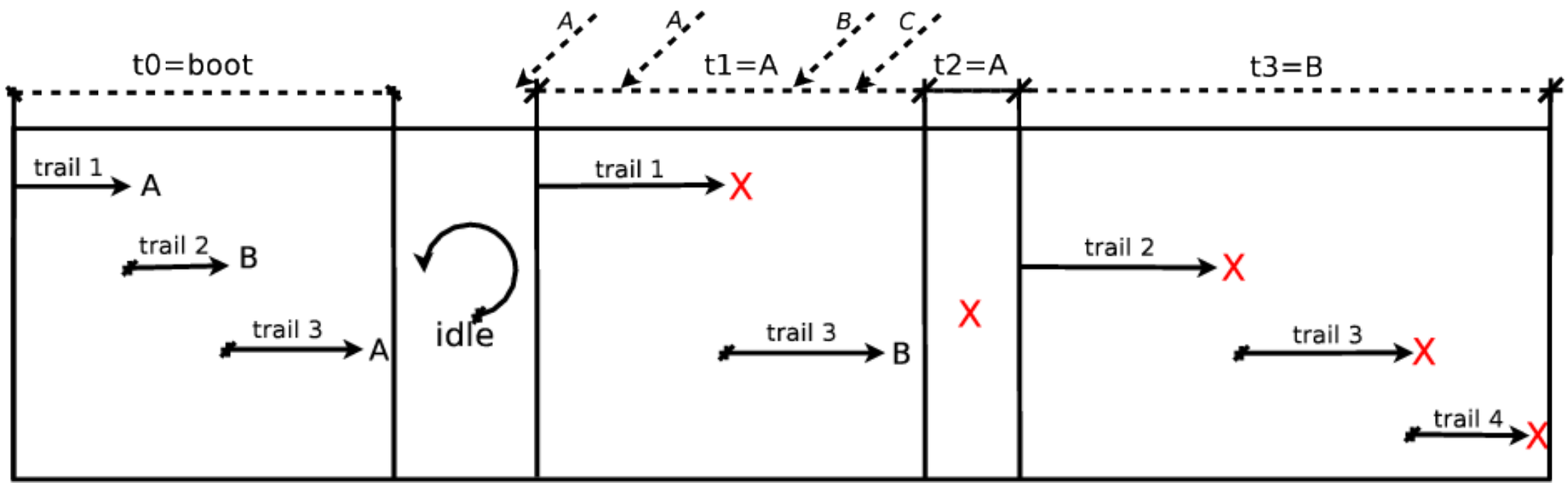
```
input  none   BUTTON;
output on/off LED;
par/or do
    await BUTTON;
with
    loop do
        await 1s;
        emit LED(on);
        await 1s;
        emit LED(off);
    end
end
```

- This program blinks an LED ~~after~~ until pressing a button

Synchronous Execution Model of Céu

```
 1:  input none A;
 2:  input none B;
 3:  input none C;
 4:  par/and do
 5:      // trail 1
 6:      <...>
 7:      await A;
 8:      <...>
 9:  with
10:      // trail 2
11:      <...>
12:      await B;
13:      <...>
```

```
14:  with
15:      // trail 3
16:      <...>
17:      await A;
18:      <...>
19:      await B;
20:      par/and do
21:          // trail 3
22:          <...>
23:      with
24:          // trail 4
25:          <...>
26:      end
27:  end
```



Parallel Composition

- par/and
  - rejoins after all trails in parallel terminate
- par/or
  - rejoins after any trail in parallel terminates
  - aborts all other trails
- par
  - never rejoins

Parallel Compositions: par/and

```
loop do
    par/and do
        await 100ms;
        _printf("Hello ");
    with
        await 250ms;
        _printf("World!\n");
    end
end
```

- Repeatedly prints "Hello World!"

Parallel Compositions: par/or

```
loop do
    par/or do
        await 100ms;
        _printf("Hello ");
    with
        await 250ms;
        _printf("World!\n");        // never reached
    end
end
```

- Repeatedly prints "Hello " only

Parallel Compositions: par/or

```
input none TERM;
par/or do
    loop do      // par/and loop never terminates
        par/and do
            await 100ms;
            _printf("Hello ");
        with
            await 250ms;
            _printf("World!\n");
        end
    end
with
    await TERM;  // but par/or terminates on TERM
                 // and kills the original loop
end
```

- Repeatedly prints "Hello World!" until receives TERM

Waiting for Multiple Events

```
input none HELLO;
input none WORLD;
par do      // par/and, par/or would behave the same
    loop do
        await HELLO;
        _printf("Hello!\n");
    end
with
    loop do
        await WORLD;
        _printf("World!\n");
    end
end
```

- Waits for HELLO or WORLD
  - (cf. select)

Internal Events

```
// internal event begin in lowercase
event int e;
var   int i = 0;
par do
    loop do
        i = await e;    // waits for changes
        _printf("Hello World: %d!\n", i);
    end
with
    loop do
        await 250ms;
        emit e(i+1);    // triggers 'e'
    end
end
```

- Prints "Hello World: 1!", "Hello World: 2!", ...

Internal Events

```
event none e;       // an event carrying no values
var int v = 0;
par do
    loop do         // a simple loop that
        await e;    // when 'e' occurs
        v = v + 1; // increments 'v'
    end
with
    // 1st trail is awaiting 'e'
    emit e;         // resumes after 1st trail halts
                    // v=0+1 => 1

    // 1st trail is awaiting 'e' again
    emit e;         // resumes after 1st trail halts
                    // v=1+1 => 1

    escape v;       // v=2
end
```