FRP

Marek Materzok

16 listopada 2010

Znacie Państwo taki kod?

```
void onChange1(boolean val) {
    label.setText(val ? "x" : "y");
    frame.setVisible(val && checkbox2.isChecked());
}
void onChange2(boolean val) {
    button.setEnabled(!val);
    frame.setVisible(val && checkbox1.isChecked());
}
checkbox1.setChangeHandler(onChange1);
checkbox2.setChangeHandler(onChange2);
```

A taki?

```
double time = getTime();
double x = 10, xv = 2;
while (looping) {
    double oldTime = time;
    time = getTime();
    drawTriangle(40, 10*sin(time/20.)+20, 4);
    drawSquare(getMouseX(), getMouseY(), 6);
    if (mouseClicked()) xv *= -1;
    x += xv * (time-oldTime);
    drawCircle(x, 20, 3);
    refresh();
}
```

Reagowanie na zdarzenia

- „Callbacki” – inwersja sterowania (Hollywood Principle)
- Imperatywne z natury (callbacki wpływaja na wykonanie programu przez zmianę globalnego stanu)
- Czułe na kolejność wykonania – im większa liczba zdarzeń, tym łatwiej zapomnieć o jakiejś kolejności i coś zepsuć

A może lepiej tak?

```
labelText = cond "x" "y" <$> checkbox1
buttonEnabled = not <$> checkbox2
frameVisible = (&&) <$> checkbox1 <*> checkbox2
```

```
triangle (pure 40) ((\t -> 10*sin(t/20)+20) <$> time)
    (pure 4)
square mouseX mouseY (pure 6)
xv = accumB 2 (const negate <$> mouseClick)
x = (+10) <$> integral timer xv
circle x (pure 20) (pure 3)
```

A może nawet tak?

```
labelText = cond "x" "y" checkbox1
buttonEnabled = not checkbox2
frameVisible = checkbox1 && checkbox2
```

```
triangle 40 (10*sin(time/20)+20) 4
square mouseX mouseY 6
xv = accumB 2 (const negate mouseClick)
x = 10 + integral timer xv
circle x 20 3
```

(gdyby mieć zgrabny autolifting)

Abstrakcja 1 – zachowania

Semantycznie – wartość zmieniająca się w czasie:

```
data Behavior t a = Beh { at :: t -> a }
```

Porządna monada:

```
fmap f r `at` t  == f (r `at` t)
return a `at` t  == a
join rr `at` t   == (rr `at t) `at` t
(s <*> r) `at` t == (s `at` t) (r `at` t)
```

Abstrakcja 2 – wydarzenia

Semantycznie – lista wystąpień z przypisanymi czasami uporządkowana według czasów:

```
data Event t a = Evt { inst :: [(t, a)] }
```

Monoid:

```
mempty          == [(infinity, undefined)]
e `mappend` e'  == ... -- uporzadkowane scalanie
```

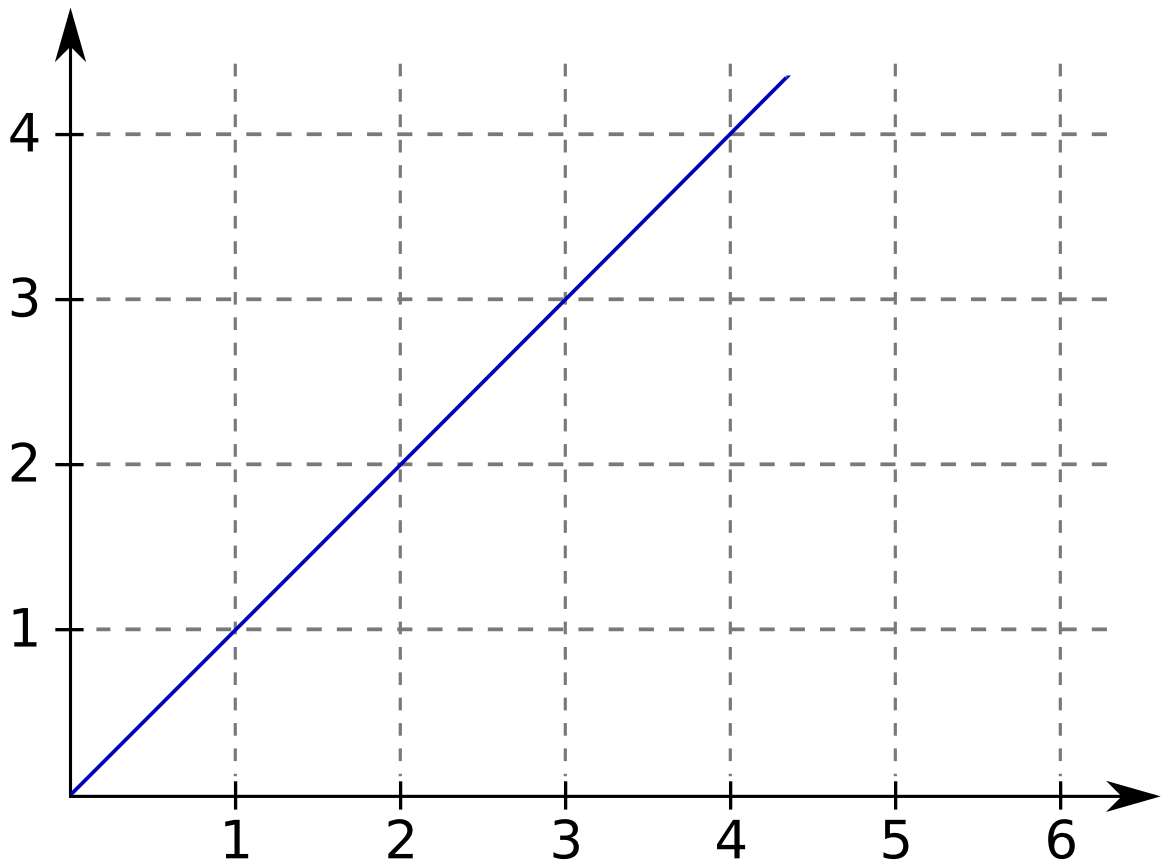
Też monada:

```
inst (fmap f e) == map (second f) e
inst (return a) == [(-infinity, a)]
inst (join ee)  == ... -- scalenie wewnetrznych zdarzen
```

Kombinatory – time

Aktualna wartość czasu:

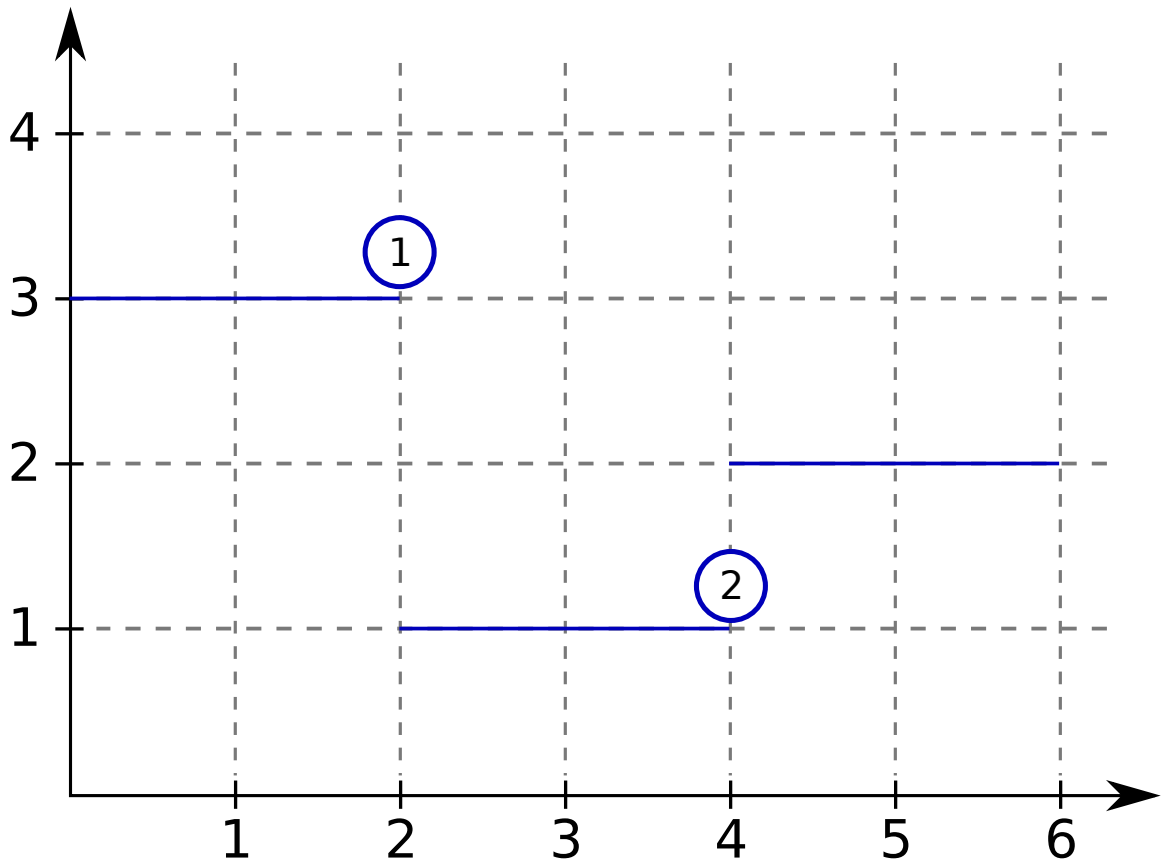
```
time :: Behavior t t
time `at` t = t
```



Kombinatory – stepper

Schodkowo zmieniająca się wartość, wystąpienie zdarzenia zmienia wartość zachowania:

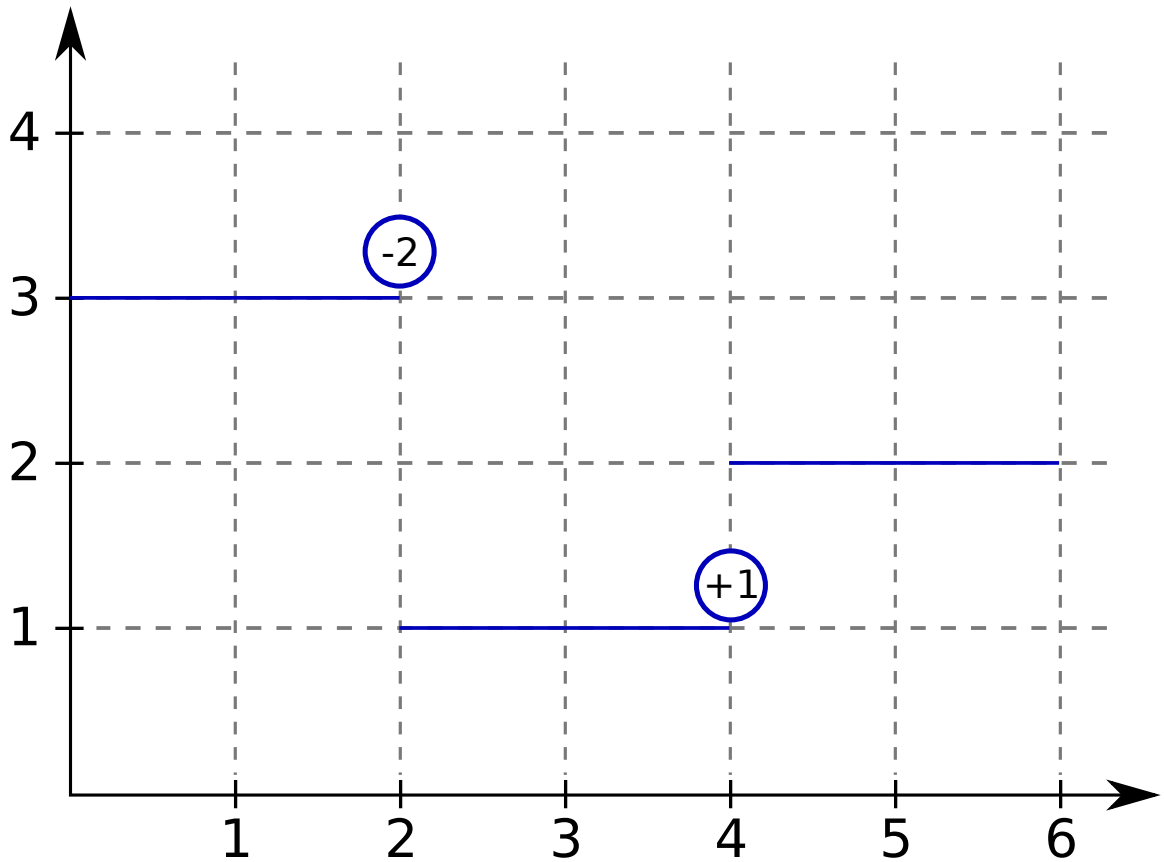
```
stepper :: a -> Event a -> Behavior a
```



Kombinatory – accumB

Wydarzenia zmieniają wartość zachowania:

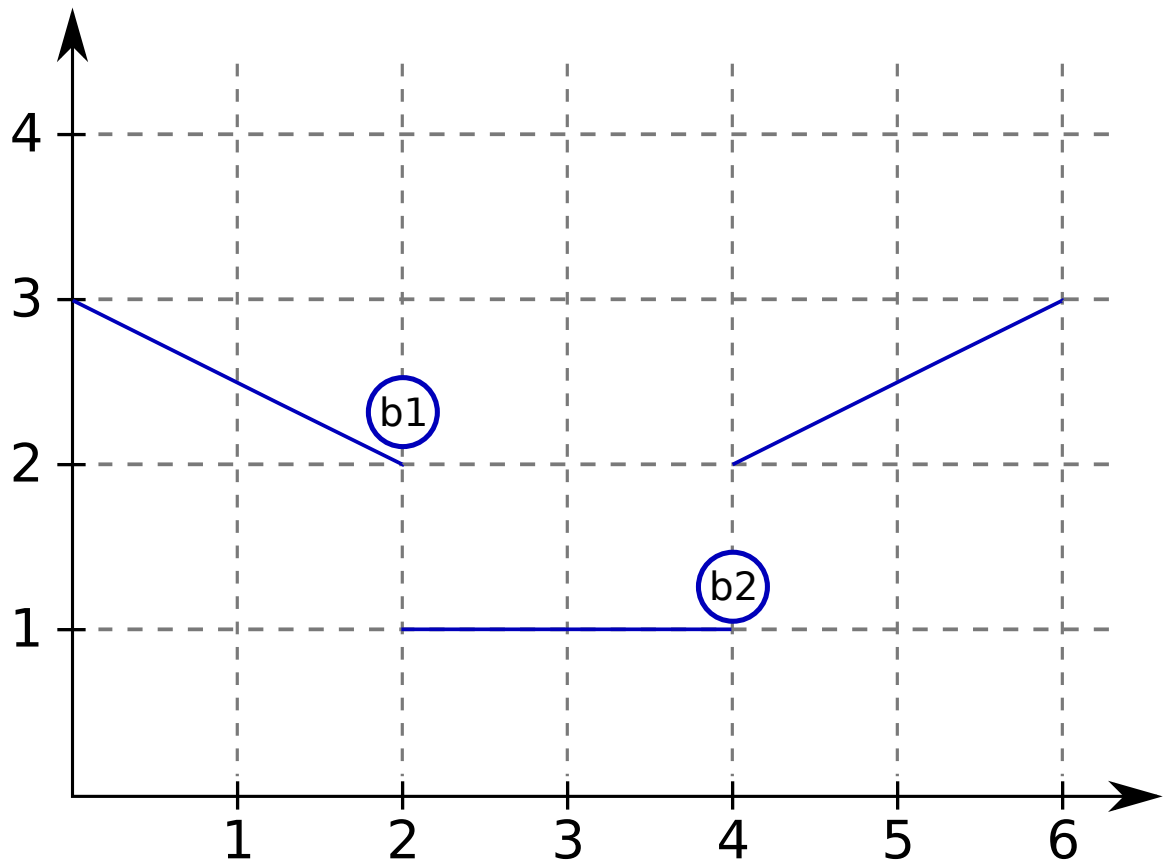
```
accumB :: a -> Event (a -> a) -> Behavior a
```



Kombinatory – switcher

Wydarzenie przełącza aktywne zachowanie:

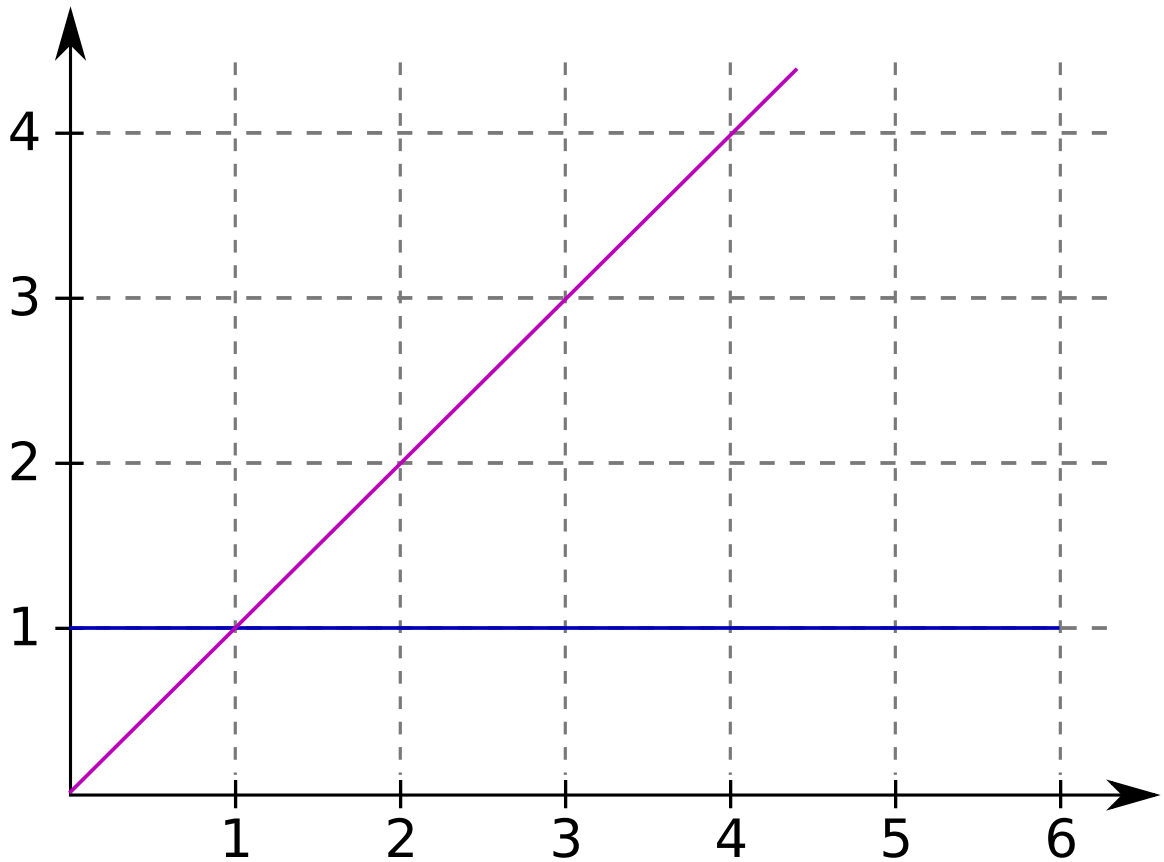
```
switcher :: Behavior t a -> Event (Behavior t a)
    -> Behavior t a
```



Kombinatory – integral

Całka (z próbkowaniem w ustalonych punktach):

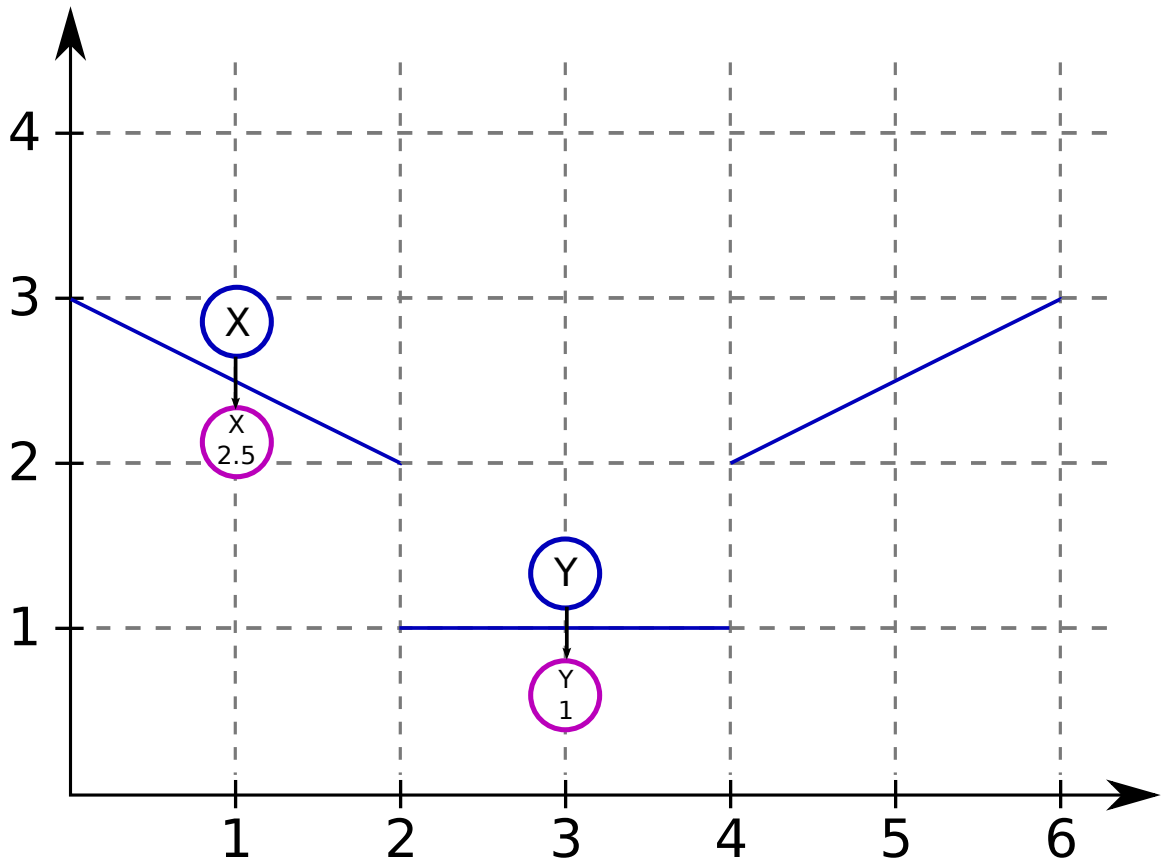
```
integral :: Event t a -> Behavior t v -> Behavior t v
```



Kombinatory – snapshot

Wydarzenie próbkuje wartość zachowania w momencie wystąpienia:

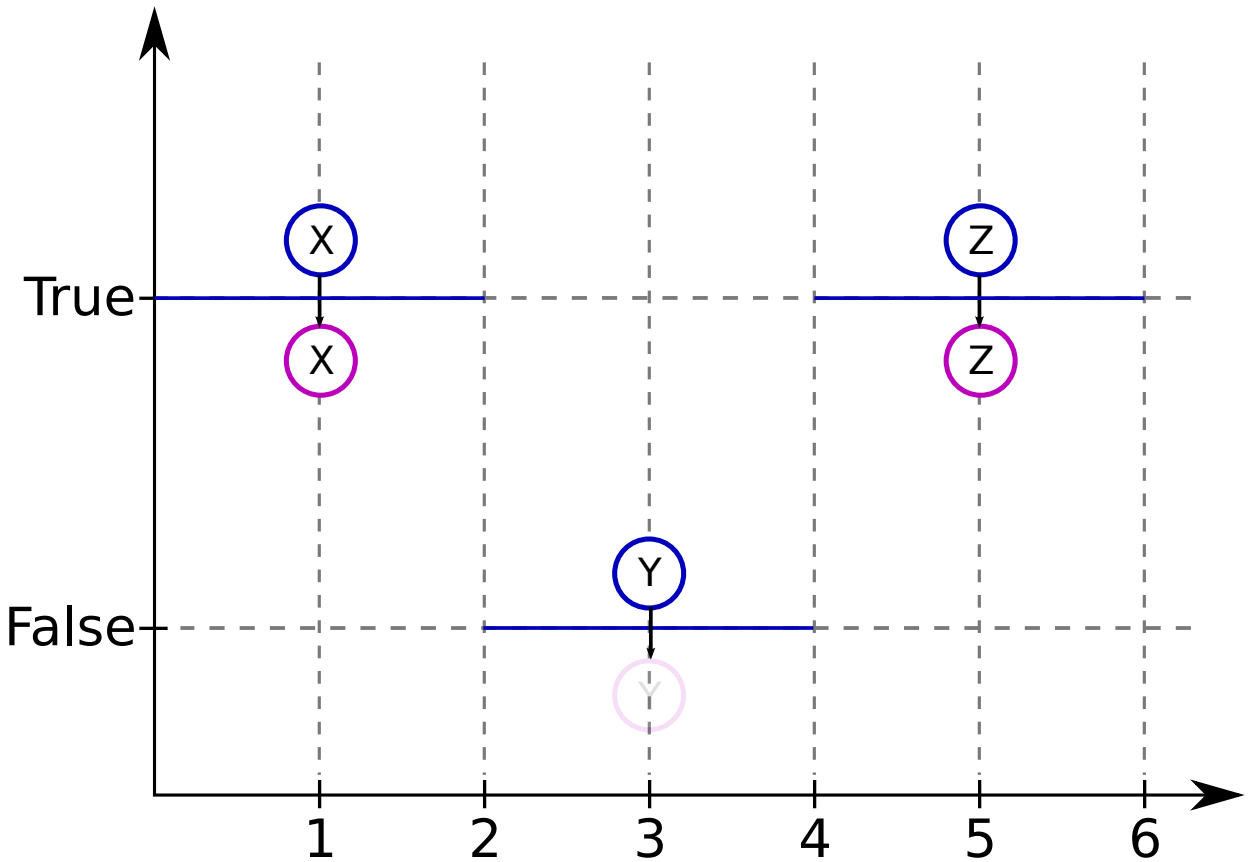
```
snapshot :: Behavior t b -> Event t a -> Event t (a, b)
```



Kombinatory – whenE

Filtrowanie wydarzeń:

```
whenE :: Behavior t Bool -> Event a -> Event a
```



OK – a implementacja?

Implementacja typu *pull* – oparta na regularnym sprawdzaniu wartości zachowań – patrz: Paul Hudak, Haskell School of Expression, rozdział 15. Dany z góry zestaw zdarzeń użytkownika:

```
data Action = KeyPress | MouseMove | MouseClick | ...
```

Parametry – monotonicznie rosnąca lista czasów próbkowania (leniwa) i wystąpień zdarzeń (`Nothing`, jeżeli próbkujemy z innego powodu niż zdarzenie):

```
data Behavior t a = Behavior (([Maybe Action], [t]) -> [a])
```

Zdarzenia traktujemy jak zachowania z `Maybe`:

```
type Event t a = Behavior t (Maybe a)
```

Mało efektywna implementacja. Często większość zachowań zmienia się tylko skokowo, są funkcjami przedziałami stałymi, ponadto zmienia się niewielki odsetek zachowań.

Sprytniejsza implementacja

Conal Elliott, Push-Pull Functional Reactive Programming.
Z grubsza:

```
data Reactive t a = a `Stepper` Event t a
newtype Event t a = Event (Future t (Reactive t a))
newtype Future t a = Future (t, a)
data Fun t a = K a | Fun (t -> a)
newtype Behavior t a = Beh (Reactive t (Fun t a))
```

Wartości reaktywne – funkcje przedziałami stałe, zmieniające wartość w ustalonych punktach czasu.
Zachowania – wartości reaktywne, które definiują na przedziałach funkcję zależną od czasu, być może stałą. (Funkcji stałej nie trzeba próbkować.)

Reprezentacja czasu dla zdarzeń

- Scalanie zdarzeń przez `mappend` – co zrobić, żeby w scalonym zdarzeniu w głowie pojawiło się pierwsze zdarzenie, które wystąpi, od razu po jego wystąpieniu?
- Improving values – leniwa lista kolejnych dolnych ograniczeń, zakończona wartością ostateczną.

```
newtype Improving a = Imp [a]
```

Jakieś rozwiązanie, ale wciąż kłopotliwe – trzeba wrzucać dużo dolnych ograniczeń, aby obliczanie minimów odbywało się szybko.

- Improved improving values:

```
data IImproving a = IImp { exact :: a,
    comparel :: a -> Ordering }
```

Używamy operatora `unamb` dla wyszukania mniejszej wartości:

```
(u `comparel` exact v \= GT)
   `unamb` (v `comparel` exact u \= LT)
```

Operator unamb

- Dwuargumentowy, jego wartością jest wartość któregoś z argumentów nie będących $\bot$ lub $\bot$, jeśli obydwa są:

$$\begin{aligned} a \,\texttt{unamb}\, b &= a && \text{gdy } a \neq \bot \\ a \,\texttt{unamb}\, b &= b && \text{gdy } b \neq \bot \\ a \,\texttt{unamb}\, b &= \bot && \text{gdy } a = \bot \text{ i } b = \bot \end{aligned}$$

  Programista musi zapewnić, że jeśli $a \neq \bot$ i $b \neq \bot$, to $a = b$.

- Typowe zastosowanie – poprawienie symetrii. Przykładowo, operator && zaimplementowany bez `unamb` musi być *strict* na jednym z argumentów: albo $\bot$ && $a = \bot$, albo a && $\bot = \bot$. Ale mając taki operator, możemy napisać tak:

```
a &:& b = a && b `unamb` b && a
```

  I wtedy `False` &:& a = a &:& `False` = `False`, nawet wtedy, gdy $a = \bot$.

Operator `unamb` – implementacja

Najprostsza działająca wersja – uruchamiamy dwa wątki, wynikiem jest wynik działania pierwszego wątku, który się zakończy:

```
a `unamb` b = unsafePerformIO (a `amb` b)
a `amb` b = evaluate a `race` evaluate b
a `race` b = do
    v <- newEmptyMVar
    forkIO (a >>= putMVar v)
    forkIO (b >>= putMVar v)
    takeMVar v
```

Praktyczna implementacja powinna zadbać o zakończenie pracy drugiego z wątków (i wątków stworzonych przez niego w zagnieżdżonych wywołaniach `unamb`) po obliczeniu wartości przez pierwszy.

OK, ale jak ZROBIĆ TYM COKOLWIEK?

Moduł `LegacyAdapters`:

```
makeEvent :: Clock -> IO (a -> IO (), Event a)
adaptE    :: Event (IO ()) -> IO ()
mkUpdater :: Clock -> Behavior (IO ()) -> IO (IO ())
```

- `Clock` – zegar, w zasadzie `IO TimeT`.
- `makeEvent` – tworzy nowe wydarzenie oraz akcję, którą sygnalizujemy wystąpienia. Zastosowanie – na przykład tworzymy wątek, który przy naciśnięciu klawisza sygnalizuje jego kod.
- `adaptE` – tworzy z wydarzenia akcję, która wykonuje akcje przekazywane w wystąpieniach wydarzenia, jak tylko się pojawią.
- `mkUpdater` – tworzy z zachowania akcję, która wykonuje akcje z zachowania. Jeśli zachowanie jest w fazie stałej, akcja jest wykonywana raz przy każdej dyskretnej zmianie, w przeciwnym wypadku następuje odpytywanie.

Dosyć tego Haskella!

- Flapjax – programowanie reaktywne dla WWW, w Javascripcie
- Implementacja czysto Javascriptowa (imperatywna, tylko „push”), do tego translator wykonujący m.in. automatyczny lifting

```
<p>The time: {! Math.floor(timerB(100) / 1000) !}.</p>
```

```
<div id="themouse"
  style={! {color: '#FFFFFF',
            backgroundColor: '#000000',
            position: 'absolute',
            left: mouseLeftB(document),
            top: mouseTopB(document),
            padding: '10px'} !}>
the mouse
</div>
```