Functional Automatic Differentiation with Dirac Impulses

Henrik Nilsson

Yale University

New Haven, CT, USA

Big picture

Functional Reactive Programming (FRP) as a starting point for a language for modeling and simulation of physical systems.

Functional languages can offer quite a lot, e.g:

- Powerful abstraction facilities
- Higher order features
- Advanced type systems

FRP itself is a flexible modeling language in some ways.

Big picture (2)

What kind of modeling?

- Differential equations.
- Equations solved numerically (integration).
- Often *hybrid* continuous and discrete systems and/or models: solutions may have “jumps”.

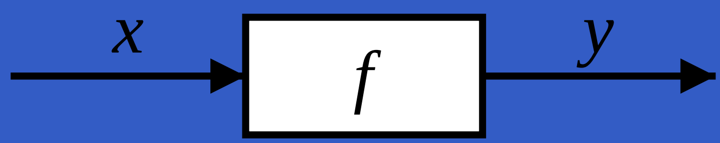
Typical systems:

- electrical circuits
- gear boxes
- chemical plants

Yampa (1)

Our current FRP implementation is called ***Yampa***.

Key concept 1: first class ***signal functions***.



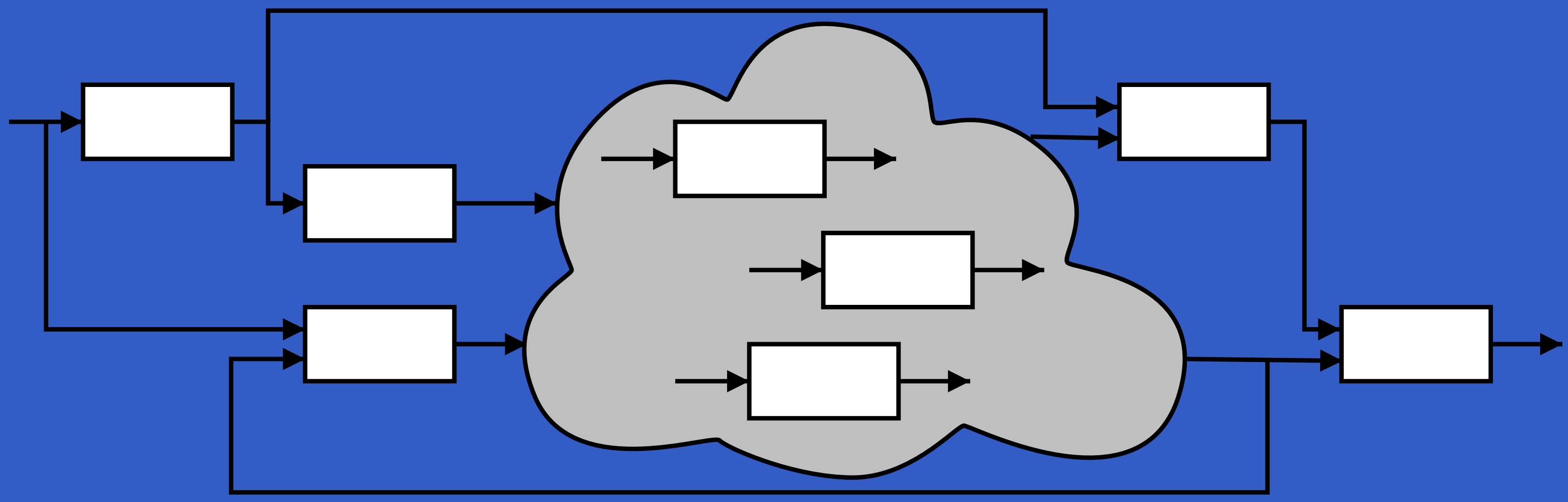
Intuition:

$$\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$$
$$\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$$
$$f\ \texttt{::}\ \texttt{SF T1 T2}$$

Signals are ***not*** first class!

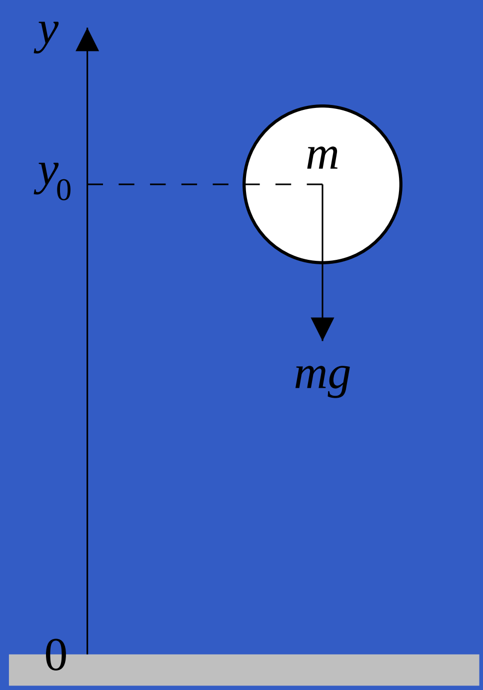
Yampa (2)

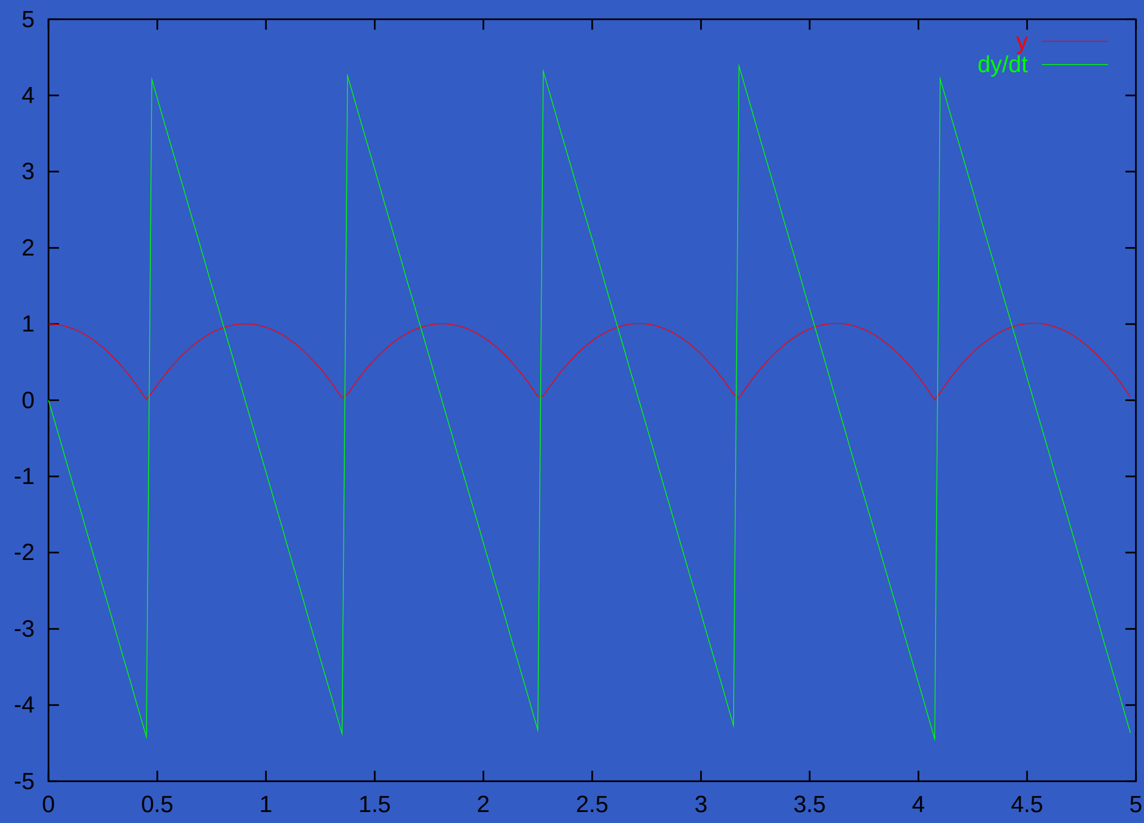
Key concept 2: ***Switch constructs*** for describing systems with varying structure:



Switching introduces discontinuities!

Simple system: a bouncing ball





A hybrid model of the bouncing ball

Yampa model of bouncing ball (arrow notation):

```
bouncing0 :: Double -> SF () (Double, Double)
bouncing0 init_pos = bouncing init_pos 0.0
    where
        bouncing init_pos init_vel =
            switch (bouncing' init_pos init_vel) $ \(pos, vel) ->
            bouncing pos (-vel)

        bouncing' init_pos init_vel = proc () -> do
            vel <- (init_vel +) ^<< integral -< -9.81
            pos <- (init_pos +) ^<< integral -< vel
            hit <- edge                      -< pos <= 0
            returnA -< ((pos, vel), hit `tag` (pos, vel))
```

Problems

Bouncing ball example exemplifies two problems we would like to address to make a better modeling language:

- Unsatisfying model: a physical force modeled by switching and recursion. Not as ***declarative*** as we would like.
- It is desirable to be able to compute derivatives of signals. But how in a hybrid setting where signals may be discontinuous?

This talk (1)

Possible solutions:

- ***Automatic differentiation*** to compute derivatives of signals.
- ***Dirac Impulses*** to
  - allow modeling of e.g. impulsive forces;
  - allow differentiation of discontinuous signals.

This talk (1)

Possible solutions:

- ***Automatic differentiation*** to compute derivatives of signals.
- ***Dirac Impulses*** to
  - allow modeling of e.g. impulsive forces;
  - allow differentiation of discontinuous signals.

Is it possible to combine Automatic Differentiation with Dirac Impulses into a ***unified*** framework?

This talk (1)

Possible solutions:

- ***Automatic differentiation*** to compute derivatives of signals.
- ***Dirac Impulses*** to
  - allow modeling of e.g. impulsive forces;
  - allow differentiation of discontinuous signals.

Is it possible to combine Automatic Differentiation with Dirac Impulses into a ***unified*** framework?

Answer: Yes, at least to some extent. This talk shows how ***in the context of Yampa***.

This talk (2)

Outline

- Automatic Differentiation
- Adding Automatic Differentiation to Yampa
- Dirac Impulses and Generalized Signals
- Differentiation of Generalized Signals

Yampa?

One interpretation:

- The work began at ***YA***le
- it ended with ***A***rrows
- and there was ***M***uch ***P***rogramming in between.

Yampa?

Or maybe it means

Y et
A nother
M ostly
P ointless
A cronym

Yampa?

Yampa is a river . . .

Yampa?

. . . with long calmly flowing sections . . .

Yampa?

. . . and abrupt whitewater transitions in between.

A good metaphor for hybrid systems!

Automatic Differentiation (1)

Automatic (or Computational) Differentiation is

- a purely ***algebraic*** method
- exact (within the limits of FP arithmetic)
- capable of finding the derivative of arbitrary computations.

Automatic Differentiation (1)

Automatic (or Computational) Differentiation is

- a purely ***algebraic*** method
- exact (within the limits of FP arithmetic)
- capable of finding the derivative of arbitrary computations.

Adding automatic differentiation is easy thanks to prior work by Jerzy Karczmarczuk, . . .

Automatic Differentiation (1)

Automatic (or Computational) Differentiation is

- a purely ***algebraic*** method
- exact (within the limits of FP arithmetic)
- capable of finding the derivative of arbitrary computations.

Adding automatic differentiation is easy thanks to prior work by Jerzy Karczmarczuk, . . .
. . . as long as signals are differentiable in the usual sense.

Automatic Differentiation (2)

Idea: Augment every computation so that the derivative(s) w.r.t. some variable is computed using the chain rule along with the main result:

```
                        z1  = x+y
z1 = x+y                z1′ = x′+y′
z2 = x*z1      ⇒        z2  = x*z1
                        z2′ = x′*z1 + x*z1′
```

How? Jerzy Karczmarczuk's method:

- Use Haskell's overloading
- Lazy evaluation to compute ***all*** derivatives

Automatic Differentiation (3)

```
data C = C Double C

zeroC     = C 0.0 zeroC
constC a = C a zeroC
dVarC a  = C a (constC 1.0)
valC (C a _)  = a
derC (C _ x') = x'

instance Num C where
    (C a x') + (C b y') = C (a+b) (x'+y')
    x@(C a x') * y@(C b y') =
        C (a*b) (x'*y + x*y')
```

Automatic Differentiation: Example

Consider $y = t^2 + k$ and wanting to compute y, $\dot{y}$, and $\ddot{y}$ for $t = 2$ and $k = 1$:

```
k = constC 1.0
t = dVarC 2.0
y = t * t + k
```

Now we have:

$$\begin{aligned} \texttt{valC y} &= \texttt{5} \\ \texttt{valC (derC y)} &= \texttt{4} \\ \texttt{valC (derC (derC y))} &= \texttt{2} \end{aligned}$$

Implementation of Yampa

Basic Yampa implementation is like other simulation systems or synchronous data flow languages:

- signals are represented by "streams" of instantaneous signal values;
- signal functions are (stateful) processors of such streams.

```
data SF a b = SF (DTime -> a -> (SF a b, b))
```

Automatic Differentiation in Yampa

A main source of continuous time varying signals in Yampa is the signal function `integral :: SF Double Double`.

All that is needed is to define a version using `C`: `integralC :: SF C C`.

Most interesting: computation of the output value. `a` and `a_prev` are current and previous input:

```
C igrl' a
  where
    igrl' = igrl + dt*valC a_prev
```

The Dirac delta function (1)

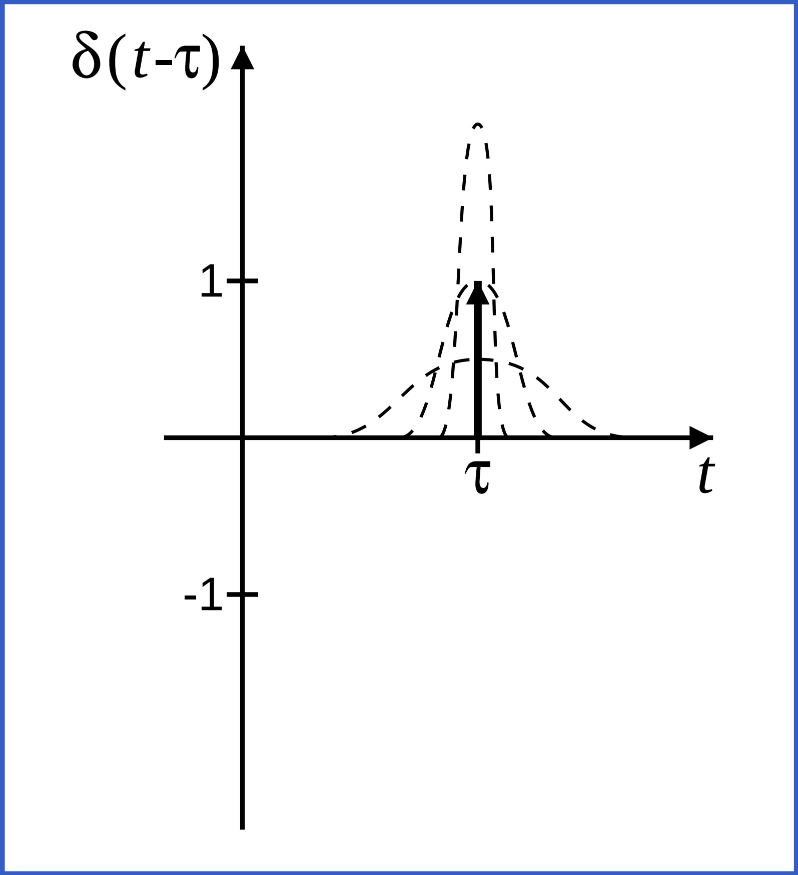
What is

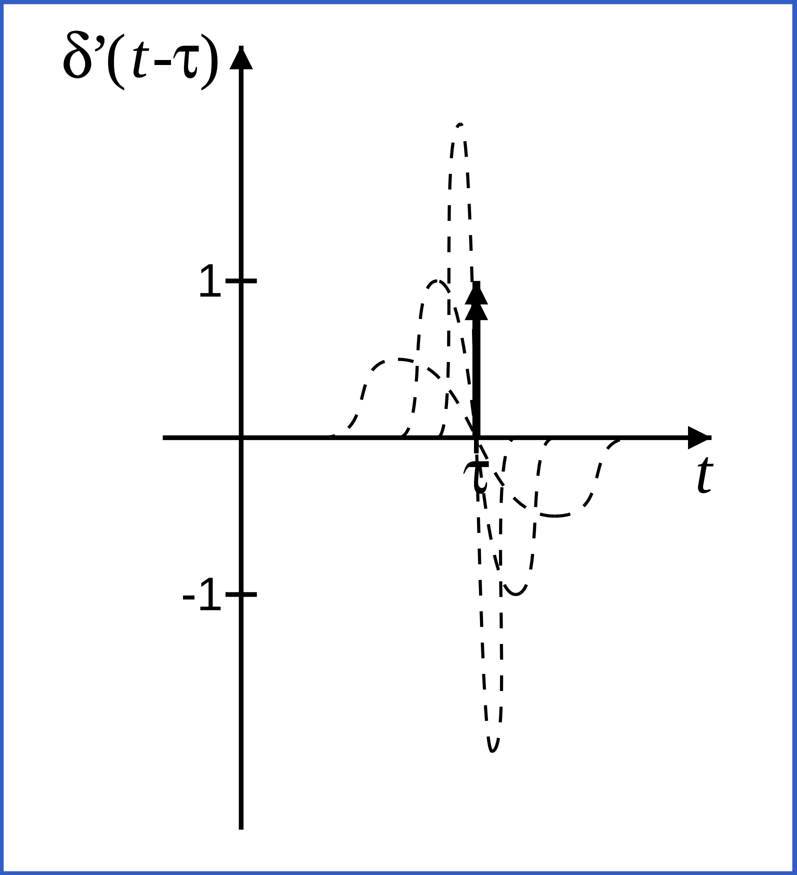
- the derivative of the unit step function?
- the force $F(t)$ associated with an "instantaneous" collision?

Such quantities can be understood through $\delta(t)$, the ***Dirac delta "function"*** or ***unit impulse***.

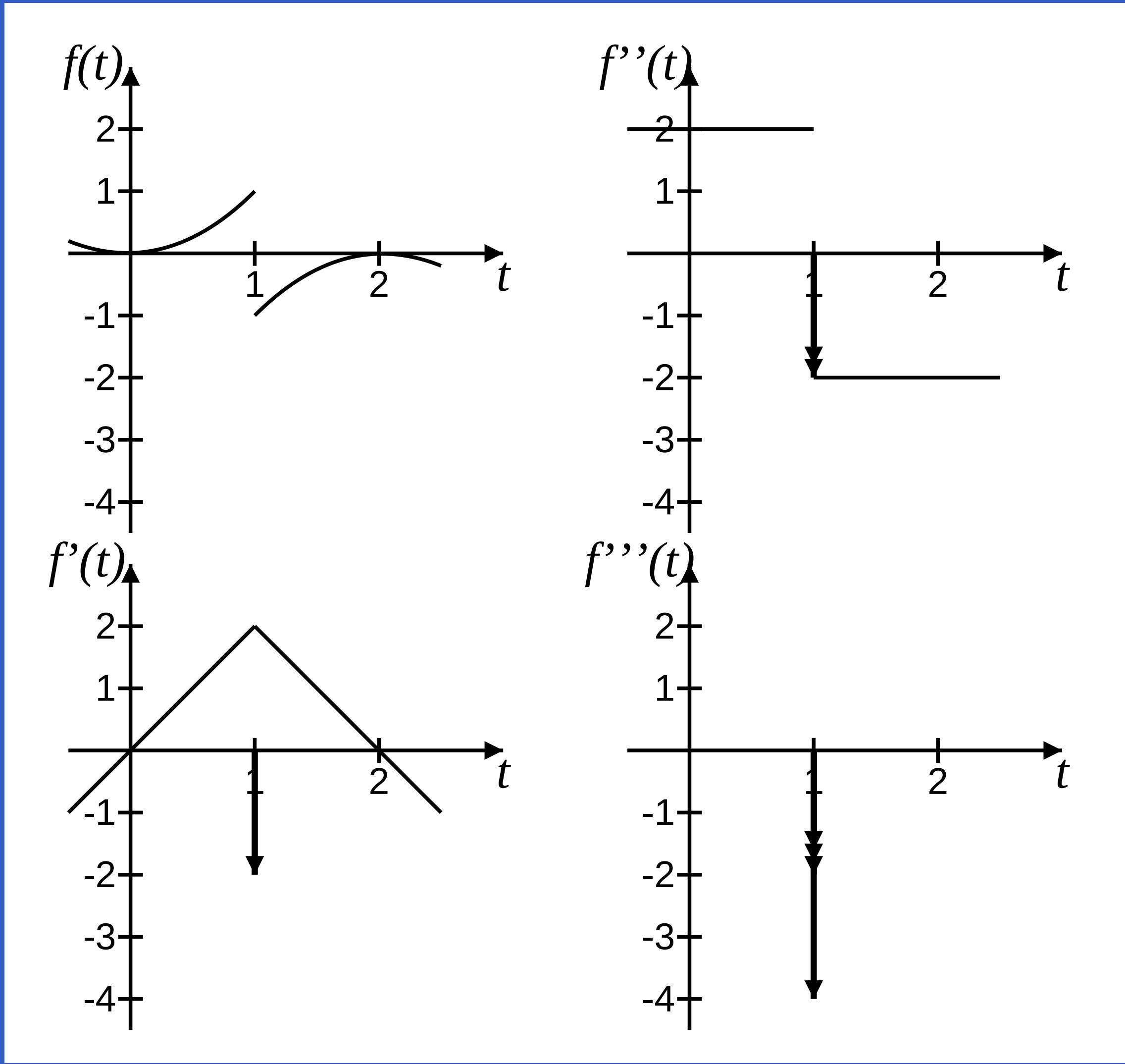
$$\int_a^b \delta(t)\,\mathrm{d}t = \begin{cases} 1 & \text{if } 0 \in (a,b) \\ 0 & \text{if } 0 \notin [a,b] \end{cases}$$

The Dirac delta function (2)





Differentiating piecewise cont. signals



Differentiating piecewise cont. signals

$$
\begin{aligned}
f(t) &= \begin{cases} t^2 & \text{if } t < 1 \\ -(2-t)^2 & \text{if } t \geq 1 \end{cases} \\
f'(t) &= \begin{cases} 2t & \text{if } t < 1 \\ 4-2t & \text{if } t \geq 1 \end{cases} - 2\delta(t-1) \\
f''(t) &= \begin{cases} 2 & \text{if } t < 1 \\ -2 & \text{if } t \geq 1 \end{cases} - 2\delta'(t-1) \\
f'''(t) &= -4\delta(t-1) - 2\delta''(t-1)
\end{aligned}
$$

Representing generalized signals (1)

Conceptually, a piecewise continuous signal can be seen as a ***generalized*** function of time:

$$s(t) = s_0(t) + \sum_{i=0}^{m} \sum_{j=1}^{n} a_{ij}\delta^{(i)}(t - \tau_j)$$

where $s_0(t)$ is an impulse-free signal.

Representing a sample of $s(t)$ at $t = \tau_j$, $j \in [1, n]$:

$$s_{\tau_j} = (s_0(\tau_j-), [a_{0j}, a_{1j}, \ldots, a_{mj}])$$

Representing generalized signals (2)

However, to make generalized signals work with automatic differentiation, each sample should include *all* derivatives at that point.

Actual representation:

```
data G = G C I
data C = C Double C
data I = NI | I [Double] I
```

Operations on G (1)

```
der :: G -> G
der (G x i) = G (derC x) (derI i)

leftLimit :: G -> C
leftLimit (G x _) = x

rightLimit :: G -> C
rightLimit (G x NI) = x
rightLimit (G (C a x') (I _ i')) =
    C (a + impStrength i') (rightLimit (G x' i'))
```

Operations on G (2)

What about numeric instances?

- Generalized functions can be added and subtracted without problem.
- In general, ***not*** possible to multiply generalized functions!
- A generalized function can be multiplied with a C^∞ function. But quite complicated, e.g.:

$$\int_{-\infty}^{\infty} f(x)\delta'(t-a)\,\mathrm{d}t = -f'(a)$$

Operations on G (3)

Product of a C^∞ function and arbitrary impulse derivative:

$$f(t)\delta^{(n)}(t-\tau) = \sum_{k=0}^{n} (-1)^k \binom{n}{k} f^{(k)}(\tau)\delta^{(n-k)}(t-\tau)$$

Thus we know the ***strengths*** of all impulse derivatives in the product, allowing us to construct a correct representation of a sample of the result.

Integration of generalized signals (1)

`x` and `x_prev` are non-impulse parts of current and previous input, `i` is impulse part of current input. Current output is then

```
G (C igrl' x) (integrateImp i)
  where
    igrl' = igrl + dt * valC x_prev
```

Accumulated state: `igrl' + strengthI i`

Next previous input: ***right limit*** of current output.

Integration of generalized signals (2)

- The left limit of the basic output value only depends on input at ***earlier*** points in time.
- The impulse part of the output ***does*** depend on the input at the current point in time: bad for recursively defined signals!

Solution: appeal to modeling knowledge and break loop by asserting that a signal is impulse-free:

```
assertNoImpulse :: SF G G
```

Where do impulses come from?

Switching introduces discontinuities. We need a version of switch that account for that by introducing impulses:

```
switchG :: SF a (G, Event b) -> (b -> SF a G)
           -> SF a G
```

We also need the ability to introduce impulses explicitly:

```
impulse :: Event C -> G
```

Bouncing ball with impulses

```
bouncing :: Position -> SF () (Position, Velocity)
bouncing init_pos = proc () -> do
    rec
        pos <- (init_pos +) ^<< integralG -< vel_ni
        hit <- edge                      -< pos <= 0
        vel <- integralG -<
            -9.81 + impulseG (hit `tag` (-2*leftLimit vel))
        vel_ni <- assertNoImpulse -< vel
    returnA -< (pos, vel)
```

Conclusions

- Automatic Differentiation can be neatly integrated with a system like Yampa.
- Dirac impulses can be used to account for discontinuities and can be made to work with the Automatic Differentiation machinery.
- Dirac impulses are also useful for modeling purposes.
- More work needed to implement algebraic operations on generalized signals properly.