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.
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
Our current FRP implementation is called \textit{Yampa}.

Key concept 1: first class \textit{signal functions}.

\[ f : : \text{SF} \ T1 \ T2 \]

Intuition:

\[
\begin{align*}
\text{Signal } \alpha & \approx \text{Time } \rightarrow \alpha \\
\text{SF } \alpha \beta & \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta \\
f & : : \text{SF } T1 \ T2
\end{align*}
\]

Signals are \textit{not} first class!
Key concept 2: **Switch constructs** for describing systems with varying structure:

Switching introduces discontinuities!
Simple system: a bouncing ball

\[ y = y_0 - mg \]

Diagram showing the motion of a bouncing ball with variables and labels.
A hybrid model of the bouncing ball

Yampa model of bouncing ball (arrow notation):

```haskell
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?
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.
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.*
Outline

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

One interpretation:

- The work began at **YAle**
- it ended with **Arrows**
- and there was **Much Programming** in between.
Yampa?

Or maybe it means

Yet
Another
Mostly
Pointless
Acronym
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 (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 is a purely algebraic method that is exact (within the limits of FP arithmetic) and 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.
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:

\[
\begin{align*}
z_1 &= x + y \\
z_2 &= x \cdot z_1
\end{align*}
\]

\[
\begin{align*}
z_1' &= x' + y' \\
z_2' &= x' \cdot z_1 + x \cdot z_1'
\end{align*}
\]

How? Jerzy Karczmarczuk’s method:

- Use Haskell’s overloading
- Lazy evaluation to compute all derivatives
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')
Consider $y = t^2 + k$ and wanting to compute $y$, $\dot{y}$, and $\ddot{y}$ for $t = 2$ and $k = 1$:

$$
\begin{align*}
k &= \text{const}1.0 \\
t &= \text{dVar}2.0 \\
y &= t \ast t + k
\end{align*}
$$

Now we have:

$$
\begin{align*}
\text{valC} y &= 5 \\
\text{valC} (\text{derC} y) &= 4 \\
\text{valC} (\text{derC} (\text{derC} y)) &= 2
\end{align*}
$$
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.

```haskell
data SF a b = SF (DTime -> a -> (SF a b, b))
```
A main source of continuous time varying signals in Yampa is the signal function
\[ \text{integral} :: \text{SF} \; \text{Double} \; \text{Double}. \]

All that is needed is to define a version using \( \mathbb{C} \):
\[ \text{integralC} :: \text{SF} \; \mathbb{C} \; \mathbb{C}. \]

Most interesting: computation of the output value. \( a \) and \( a_{\text{prev}} \) are current and previous input:
\[ \mathbb{C} \; \text{igrl}' \; a \]
where
\[ \text{igrl}' = \text{igrl} + \text{dt} \ast \text{valC} \; a_{\text{prev}} \]
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.

$$\int_a^b \delta(t) \, dt = \begin{cases} 1 & \text{if } 0 \in (a, b) \\ 0 & \text{if } 0 \not\in [a, b] \end{cases}$$
The Dirac delta function (2)

\[ \delta(t-\tau) \]

\[ \delta'(t-\tau) \]
Differentiating piecewise continuous signals
Differentiating piecewise cont. signals

\[ f(t) = \begin{cases} 
  t^2 & \text{if } t < 1 \\
  -(2 - t)^2 & \text{if } t \geq 1
\end{cases} \]

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

\[ f''(t) = \begin{cases} 
  2 & \text{if } t < 1 \\
  -2 & \text{if } t \geq 1
\end{cases} - 2\delta'(t - 1) \]

\[ f'''(t) = -4\delta(t - 1) - 2\delta''(t - 1) \]
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:

```haskell
data G = G C I
data C = C Double C
data I = NI | I [Double] I
```
Operations on $G$ (1)

\[
der :: G \rightarrow G 
\der (G \times i) = G (\derC x) (\derI i)
\]

\[
\text{leftLimit} :: G \rightarrow C
\text{leftLimit} (G \times _) = x
\]

\[
\text{rightLimit} :: G \rightarrow C
\text{rightLimit} (G \times NI) = x
\text{rightLimit} (G (C a x') (I _ i')) =
\quad C (a + \text{impStrength} i') (\text{rightLimit} (G x' i'))
\]
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 $C^\infty$ function. But quite complicated, e.g.:

$$\int_{-\infty}^{\infty} f(x) \delta'(t - a) \, dt = -f'(a)$$
Operations on $G(3)$

Product of a $C^\infty$ 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 \( x_{\text{prev}} \) are non-impulse parts of current and previous input, \( i \) is impulse part of current input. Current output is then

\[
G (C \text{igrl'} \ x) \ (\text{integrateImp} \ i)
\]

where

\[
\text{igrl'} = \text{igrl} + dt \times \text{valC} \ x_{\text{prev}}
\]

Accumulated state: \( \text{igrl'} + \text{strengthI} \ i \)

Next previous input: \textit{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:

```plaintext
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:

\[
\text{switchG} :: \text{SF} \ a \ (G, \ \text{Event} \ b) \rightarrow (b \rightarrow \text{SF} \ a \ G) \\
\rightarrow \text{SF} \ a \ G
\]

We also need the ability to introduce impulses explicitly:

\[
\text{impulse} :: \text{Event} \ C \rightarrow \ G
\]
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.