Functional Automatic Differentiation with Dirac Impulses

Henrik Nilsson
Department of Computer Science
Yale University
Henrik.Nilsson@yale.edu

ABSTRACT

Functional Reactive Programming (FRP) is a framework for reactive programming in a functional setting. FRP has been applied to a number of domains, such as graphical animation, graphical user interfaces, robotics, and computer vision. Recently, we have been interested in applying FRP-like principles to hybrid modeling and simulation of physical systems. As a step in that direction, we have extended an existing FRP implementation, *Yampa*, in two new ways that make it possible to express certain models in a very natural way, and reduces the amount of work needed to put modeling equations into a suitable form for simulation. First, we have added *Dirac impulses* that allow certain types of discontinuities to be handled in an easy yet rigorous manner. Second, we have adapted *automatic differentiation* to the setting of Yampa, and *generalized* it to work correctly with Dirac impulses. This allows derivatives of piecewise continuous signals to be well-defined at all points. This paper reviews the basic ideas behind automatic differentiation, in particular Jerzy Karczmarczuk's elegant version for a lazy functional language with overloading, and then considers the integration with Yampa and the addition of Dirac impulses.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*functional languages, data-flow languages*; I.6.2 [**Simulation And Modeling**]: Simulation Languages; I.6.8 [**Simulation And Modeling**]: Types of Simulation—*continuous, discrete event*

General Terms

Languages, Algorithms

Keywords

FRP, Haskell, functional programming, synchronous dataflow languages, modeling languages, hybrid modeling, automatic differentiation, distribution theory, Dirac impulses

ICFP'03 August 25–27, 2003, Uppsala, Sweden.

1. INTRODUCTION

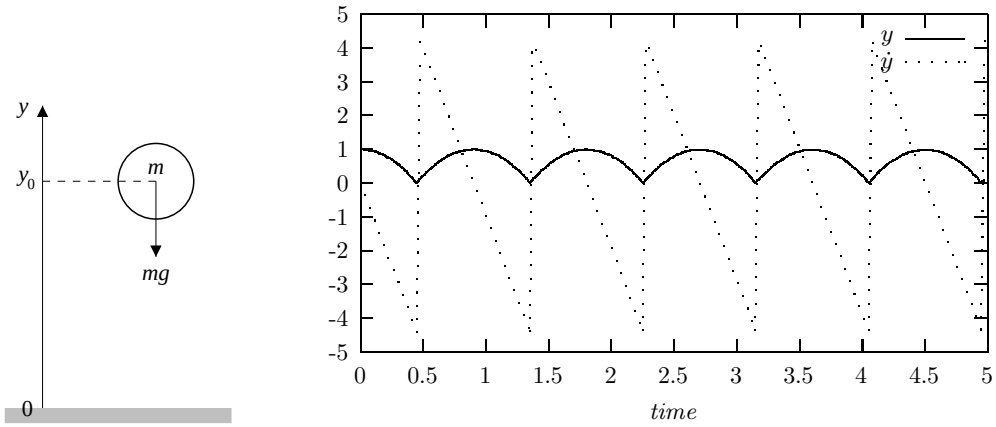
Consider a highly simplified model of a bouncing ball as shown in figure 1(a). It is dropped from an initial height y_0 and falls toward the floor at $y = 0$ under the influence of the gravitational force $-mg$, where m is the mass of the ball. For simplicity's sake, we want to model the ball as if it were a point mass. Moreover, we do not want to model the physics of the interaction when the ball hits the floor in any great detail. Instead we assume that the impact is fully elastic and that it occurs instantaneously. Under these assumptions, the velocity of the ball will simply change sign whenever it bounces off the floor. Figure 1(b) shows the position y and velocity $\dot{y}$ as functions of time when a ball modeled in this fashion is released from $y_0 = 1$ m.

This is an example of a *hybrid model*; that is, a model having both *continuous* and *discrete* aspects. The behavior of the falling ball is described continuously through Newton's laws of motion, whereas the instantaneous interaction between the ball and the floor are isolated events occurring at discrete points in time. While the present model is trivial, the kind of *modeling simplifications* it exemplifies are very common in practice, and a major reason for why models of physical systems often become hybrid models.

Hybrid models are described by a *set* of systems of differential equations. Each individual system of equations describes the continuous behavior of the modeled system in a particular *mode* of operation by relating time-varying, continuous variables and their time derivatives. Discrete aspects of the behavior are expressed by *switching* among the modes of operation, governed by predicates specifying switching events and suitable initial conditions for the continuous state variables of the mode being activated. Hybrid automata [7] are one way to formalize these notions. These models can then be used in different ways, but evaluation of the behavior over time through simulation based on numerical integration is the most common application.

Explicit mode switching is not always the most intuitive and clear way to describe a hybrid model. In the example of the bouncing ball, most modeling languages require a mode switch when the ball bounces off the floor. The initial condition of the new mode will have the same position for the ball (on the floor) but negate the velocity of the ball so that it is moving up. In systems like this one where the sole purpose of mode switching is an abrupt change in one of the state variables, explicit switching leads to an unnecessarily complicated description of the system.

In cases like these, *Dirac impulses* [19, 6] offer an alternative way to account for abrupt changes in continuous vari-



(a) Ball over floor

(b) Position and velocity of the ball

Figure 1: Idealized bouncing ball

ables. Informally, a Dirac impulse can be understood as an infinitely high and narrow impulse whose *area* is finite. This is exactly what is needed to model the force of interaction between the ball and the floor in this case in which the interaction is assumed to take place instantaneously. This idea has obviously been considered in the modeling community, but does not yet seem to be available in any mainstream systems for hybrid modeling and simulation [12].

As will become clear, mode switching also has serious implications for another aspect of many simulation systems: differentiation. It is often necessary compute derivatives of various functions in a model. One reason is the use of sophisticated numerical integration methods that solve non-linear equations at each time step. The need also arises when one time-varying quantity in a model is the *time derivative* of another, and the *causality*, i.e., the direction of data flow, in a particular mode of operation is such that the former needs to be computed from the latter through differentiation, rather than the latter from the former through numerical integration.

Either way, *automatic differentiation* [2] is a key technique for computing derivatives for this kind of applications. It is an *algebraic* method based on computing the value of a function along with the derivative of that function with respect to some specific variable at the same point. Thus, unlike symbolic differentiation, one does not obtain a closed, separate expression for the derivative. In return, the method is quite general and capable of differentiating functions expressed using arbitrary programming language constructs, such as conditionals and iteration.

Naturally automatic differentiation only yields correct results if the function being differentiated is *continuously differentiable* at the point of differentiation. If we consider time derivatives of variables in the continuous part of a model, it should be clear that we have a problem in a hybrid setting since each mode switch can introduce arbitrary discontinuities in a variable and its derivatives. That is, the variables are only *piecewise* continuous, and we cannot trust the results of automatic differentiation at all points in time.

However, the theory of Dirac impulses was developed precisely to deal with problems associated with discontinuities. Specifically, the unit impulse can be seen as the derivative of the unit step function. Thus, could we not again turn to Dirac impulses to remedy the situation?

In this paper, we will show that this indeed is possible. Moreover, we will show that a lazy functional language like Haskell allows a rather elegant and general solution. The development is carried out in the context of *Yampa* [13, 8], a recent member in the family of languages based on the ideas of *Functional Reactive Programming* (FRP). FRP, in various incarnations, has been applied to a number of domains, such as graphical animation [4], graphical user interfaces [3], robotics [17, 16], and computer vision [18]. Recently, we in the Yale Haskell group have been interested in applying FRP-like principles to hybrid modeling and simulation of physical systems, where our ultimate goal is a system in which models can be expressed through non-directed equations, so called *non-causal* (or *object oriented*[1]) modeling [5, 1, 14]. The work presented here is a step in that direction, even if Yampa only allows causal modeling.

The rest of the paper is organized as follows. Sections 2 and 3 review the basic ideas behind automatic differentiation, in particular Jerzy Karczmarczuk's elegant version for a lazy functional language with overloading [10], and the fundamental concepts of Yampa. Piecewise continuous, time-varying variables are known as *signals* in Yampa, and as a first step in the actual development, section 4 specializes Karczmarczuk's method to computing time derivatives of signals and considers the interplay between this and Yampa's facilities for numerical integration. Sections 5 and 6 then pave the way for a correct treatment of discontinuities in signals by moving to *generalized signals*, where a signal no longer is regarded as a function of time, but as a *generalized function* or *distribution*, such as a Dirac impulse.

[1]Not to be confused with object-oriented *programming* languages. Concepts like classes and inheritance may be part of an object-oriented modeling language, but methods and imperative variables are not.

The automatic differentiation machinery is then adapted to work on generalized signals. The final step in the development is taken in section 7, which considers the integration of generalized signals into Yampa. The end result is a *unified* framework in which impulses are available for the benefit of the modeler, and in which *generalized* automatic differentiation guarantees correct results for time derivatives of signals at all points.

2. AUTOMATIC DIFFERENTIATION

Automatic differentiation [2] is the technique of choice for computing derivatives in many application areas. Its advantages include the ability to differentiate arbitrary program code (as long as the code implements differentiable functions), exact results within the limitations of floating point arithmetic, and good performance. In contrast, symbolic differentiation has much more limited applicability, and when applicable often yields unwieldy results and thus bad performance. Numeric differentiation is fraught with problems stemming from the fact that it does not yield exact results. Automatic differentiation is a purely *algebraic* method, and the key idea is to augment every computation in a code fragment so that derivatives with respect to a chosen variable are computed along with the main result. This is also the main drawback of the method: unlike symbolic differentiation, the end result is not a separate, self-contained expression for the derivative that can be used to compute the value of the derivative at arbitrary points. Instead, the value of the derivative is obtained in conjunction with the value of the function at whichever point the latter is evaluated.

The following example illustrates the basic idea. Consider the code fragment

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

If we assume that the code fragment that assigned some values to `x` and `y` has been augmented so that the derivatives of these variables with respect to some common chosen variable of differentiation is available in the variables `x'` and `y'` respectively, then the above code fragment can be augmented to compute the derivatives of `z1` and `z2` with respect to that same variable of differentiation as follows:

```
z1  = x + y
z1' = x' + y'
z2  = x * z1
z2' = x' * z1 + x * z1'
```

Methods for automatic differentiation can be quite a bit more involved, especially for multi-variate cases, but the basic approach outlined above is enough for our purposes.

The next question is how to actually go about augmenting a program. One possibility is to employ source-to-source transformations. Another is to overload arithmetic operators and functions so that they compute derivatives along with the main result. Jerzy Karczmarczuk has described a particularly elegant formulation for a lazy functional language with overloading, where lazy evaluation is exploited to, conceptually, compute derivatives of *all* orders, not just the first one [10]. That is the approach we are going to use.

First, we need a numeric domain for our differential algebra. Elements in this domain represent values of continuous functions at some point along with *derivatives of all orders at that same point*:

```
data C = C Double C

valC (C a _)  = a
derC (C _ x') = x'
```

Note that the name `C` (suggesting *continuous*) is used both for the type itself and for the value constructor. Elements of type `C` are pairs, where the first field, here of type `Double`[2], represents the value of some function, and the second, recursively of type `C`, represents the first derivative. But that means the derivative can be differentiated yielding the second derivative, again of type `C`, and so on. Thus the representation includes all derivatives, to be computed lazily.

The value of the constant zero is 0 everywhere, and so is its derivative. The representation of zero in `C` is thus:

```
zeroC :: C
zeroC = C 0.0 zeroC
```

`zeroC` can be used to define a function that computes the representation of arbitrary constants, which in turn defines the meaning in `C` of Haskell's overloaded numerical literals:

```
constC :: Double -> C
constC a = C a zeroC
```

The derivative of the variable of differentiation with respect to itself is always 1, and the representation of its value in `C` at some arbitrary point is thus given by:

```
dVarC :: Double -> C
dVarC a = C a (constC 1.0)
```

The definitions of arithmetic operators are straightforward:

```
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')
```

Note that these definitions are *recursive* since the overloaded operators `+` and `*` are used at type `C` for computing the derivative of the result. Further operators and mathematical functions like `sin` and `cos` can be defined with equal ease.

As an illustration, suppose we have $y = t^2 + k$ and that we want to compute y, $\dot{y}$, and $\ddot{y}$ for $t = 2$ and $k = 1$. This can be done by simply transliterating these equations into Haskell:

```
k = constC 1
t = dVarC 2
y = t * t + k
```

We now have:

```
                    valC y  =  5
               valC (derC y)  =  4
          valC (derC (derC y))  =  2
```

3. YAMPA

This section gives a short introduction to Yampa, a language embedded in Haskell for describing reactive, hybrid systems [13, 8]. A fundamental notion in Yampa is that of *signals*. A signal is, conceptually, a function of time, or,

[2]In a more thorough implementation, `C` would be parameterized w.r.t the numerical carrier type.

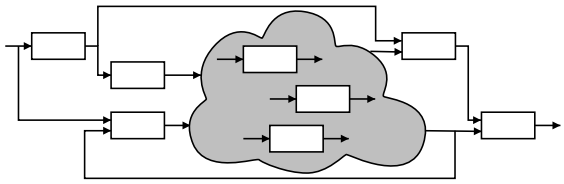


Figure 2: System of interconnected signal functions with varying structure

equivalently, a time-varying value (sometimes called *fluent*). Thus, intuitively, for some suitable type `Time` representing continuous time:

$$\texttt{Signal}\,\alpha \approx \texttt{Time} \to \alpha$$

However, signals are *not* first class entities in Yampa: they only exist indirectly through the notion of *signal functions* introduced below. Moreover, an executable Yampa implementation can only approximate this conceptual signal model since continuous-time signals necessarily are evaluated for only a discrete set of sample points.

Conceptually, the domain of a signal can either be continuous or discrete. In the former case, the signal is defined at every point in time. In the latter case, the signal is a partial function, only defined at discrete points in time associated with the occurrence of some *event*. In Yampa, we have chosen to blur this distinction. The notion of discrete-time signals is captured by lifting the *range* of of continuous-time signals using an option type called `Event`. This type has two constructors: `NoEvent`, representing the absence of a value; and `Event`, representing the presence of a value:

```
data Event a = NoEvent | Event a
```

A discrete-time signal carrying elements of type A can thus be thought of as a function of type $\texttt{Signal}\,(\texttt{Event}\,A)$.

The next important Yampa notion is that of *signal functions*. A signal function is a *pure* function that maps an input signal onto a output signal. By changing the perspective slightly, a signal function can also be seen as a time-indexed instantaneous mapping from signal values to signal values. If the mapping in fact is time-invariant, the signal function is said to be *stateless*, otherwise it is said to be *stateful*. Unlike signals, signal functions *are* first class entities in Yampa. The type of a signal function mapping a signal of type α onto a signal of type β is written $\texttt{SF}\,\alpha\,\beta$. Intuitively, we have

$$\texttt{SF}\,\alpha\,\beta \approx \texttt{Signal}\,\alpha \to \texttt{Signal}\,\beta$$

If more than one input or output signal are needed, tuples are used for α or β since a continuous signal of tuples is isomorphic to a tuple of continuous signals.

A Yampa system consists of a number of interconnected signal functions, operating on the system input and producing the system output. The signal functions operate in parallel, sensing a common *rate* of time flow.

The structure of a Yampa system may evolve over time. For example, signal functions can be added or deleted; see figure 2. These structural changes are known as *mode switches*. The first class status of signal functions in combination with powerful switching constructs make Yampa an unusually flexible language for describing hybrid systems [13].

Yampa's signal functions are an instance of the arrow framework proposed by Hughes [9]. Two central combinators from that framework are `arr`, which lifts an ordinary function to a stateless signal function, and `<<<`, which composes two signal functions:

```
arr   :: (a -> b) -> SF a b
(<<<) :: SF b c -> SF a b -> SF a c
```

Yampa also provides a combination of the two, the arrow-compose combinator:

```
(^<<) :: (b -> c) -> SF a b -> SF a c
```

Through the use of these and related plumbing combinators, arbitrary signal function networks can be expressed.

Paterson's syntactic sugar for arrows [15] effectively allows signals to be named, despite signals not being first class values. This eliminates a substantial amount of plumbing, resulting in much more legible code. In fact, the plumbing combinators will rarely be used in the examples in this paper. In this syntax, an expression denoting a signal function has the form:

```
proc pat -> do [ rec ]
    pat_1 <- sfexp_1 -< exp_1
    pat_2 <- sfexp_2 -< exp_2
    ...
    pat_n <- sfexp_n -< exp_n
    returnA -< exp
```

The keyword `proc` is analogous to the λ in λ-expressions, *pat* and pat_i are patterns binding signal variables pointwise by matching on instantaneous signal values, *exp* and exp_i are expressions defining instantaneous signal values, and $sfexp_i$ are expressions denoting signal functions. The idea is that the signal being defined pointwise by each exp_i is fed into the corresponding signal function $sfexp_i$, whose output is bound pointwise in pat_i. The overall input to the signal function denoted by the `proc`-expression is bound by *pat*, and its output signal is defined by the expression *exp*. The signal variables bound in the patterns may occur in the signal value expressions, but *not* in the signal function expressions ($sfexp_i$). If the optional keyword `rec` is used, then signal variables may occur in expressions that textually precedes the definition of the variable, allowing recursive definitions (feedback loops). Finally,

```
let pat = exp
```

is shorthand for

```
pat <- arr id -< exp
```

allowing easy binding of instantaneous values.

To illustrate Yampa and the arrow notation, we provide a bouncing ball model according to the specification given in the introduction. The following are the most important Yampa primitives used in the model:

```
integral :: SF Double Double
edge :: SF Bool (Event ())
switch :: SF a (b, Event c) -> (c -> SF a b)
          -> SF a b
```

The signal function `integral` integrates its input. `edge` is a signal function that generates an event whenever the input signal goes from `False` to `True`. The signal function

`switch` switches from one subordinate signal function into another when a switching event occurs. Its first argument is the signal function that initially is active. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, `switch` applies its second argument to the value tagged to the event and switches into the resulting signal function.

The code for the model is as follows. It does not use any of the new facilities described in this paper.

```
bouncing ::
  Position -> SF () (Position, Velocity)
bouncing y0 = bouncing' y0 0.0
  where
    bouncing' y0 yd0 =
      switch (bouncing0 y0 yd0) $ \(y, yd) ->
      bouncing' y (-yd)

    bouncing0 y0 yd0 = proc () -> do
      yd <- (yd0 +) ^<< integral -< -9.81
      y  <- (y0 +)  ^<< integral -< yd
      hit <- edge                -< y <= 0
      returnA -< ((y, yd), hit `tag` (y, yd))
```

Here, `bouncing0` realizes the physics of a falling ball and detects the collision event. The event is tagged with the values of the state variables, i.e. the height above the floor and the velocity, at the point of impact, enabling the switch to pass the state on to the subsequent mode. In this case, we switch back into the same mode, using the same height (to ensure continuity in position) but negating the velocity to obtain the initial value for the state variables in the new mode.

4. INTEGRATING AUTOMATIC DIFFERENTIATION INTO YAMPA

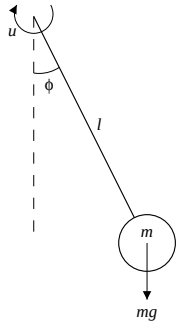
We now turn to computing time derivatives of signals in Yampa through automatic differentiation. In Yampa, signals are only evaluated for the *current time*, and as signals are not first class entities, this is done implicitly. Since we are only interested in time derivatives of signals, there is no thus need for the function `dVarC` from section 2: as long as signal functions that are sources of time-varying signals construct their output correctly in the domain `C`, signals can be added, multiplied, or transformed in other ways using operations on `C`, and differentiation will just work.

Since the signal function `integral` is the main source of continuous time-varying signals, we will focus on that. The output signal $y(t)$ obtained by applying `integral` to an input signal $x(t)$ is defined by:[3]

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau \tag{1}$$

According to the fundamental theorem of calculus, the derivative of $y(t)$ is simply $x(t)$. Thus, all we need to do to achieve our goal, is to define a version `integralC` of `integral` that works on signals of type `C` and that ensures that the derivative of the output signal is equal to the input signal.

[3]Time is *local* time, measured from the time at which a signal function is switched in, i.e., applied to its input signal.



Non-broken mode:

$$\begin{aligned} x &= l\sin(\phi) \\ y &= -l\cos(\phi) \\ ml^2\ddot{\phi} + mgl\sin(\phi) &= u \end{aligned}$$

Initial conditions: $\phi_0 = \pi/4$, $\dot{\phi}_0 = 0$

Broken mode:

$$\begin{aligned} m\ddot{x} &= 0 \\ m\ddot{y} &= -mg \end{aligned}$$

Initial cond.: x, y, $\dot{x}$, $\dot{y}$ continuous.

Figure 3: Breaking pendulum

Like in other simulation systems or synchronous data flow languages, signals in Yampa are represented by streams of instantaneous signal values and signal functions are stateful stream processors. A signal function is implemented as a function that maps the *time difference* since the previous sample time and the current instantaneous input value to a pair of a new signal function (a kind of continuation, carrying any state) and an instantaneous output value:[4]

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

The signal function `integralC` is defined as follows:

```
integralC :: SF C C
integralC = SF tf0
  where
    q0       = 0.0
    tf0 _ x0 = (igrlAuxC q0 x0, C q0 x0)
    igrlAuxC q_prev x_prev = SF tf
      where
        tf dt x = (igrlAuxC q x, C q x)
          where
            q = q_prev + dt * valC x_prev
```

Note that the output value, e.g. `C q x` *after* the first step, is defined so that the instantaneous value of the derivative `x` is the instantaneous input value, as required by the fundamental theorem of calculus. Also note that the value part `q` of the output does *not* depend on the current input value, only on the previous one. This is crucial to make recursive equations involving integrals well-defined. For illustrative purposes, the employed integration method is simple Euler integration. A more sophisticated approach such as a Runge-Kutta method could be used. Since we have access to derivatives of arbitrary order, another possibility would be to use Taylor methods [11]. However, it is unclear if variable step-size methods could be used in the Yampa setting.

To illustrate automatic differentiation in the context of Yampa, consider modeling the pendulum in figure 3. It consists of a mass m attached to the end of a stiff, massless rod l, fixed to some supporting structure at the other end, where an external torque u also can be applied. The angle ϕ gives the deviation of the rod from the plumb line. At some time the rod could break causing the mass to fall freely. Thus the system has two modes, described by the equations given in the figure along with initial conditions.

[4]This is a simplified account; see [13] for details.

The most interesting aspect of the model for our purposes is that the state of the system is best described by *different* variables in the two modes: in the non-broken case by the angle of deviation ϕ and its derivative $\dot{\phi}$; in the broken mode by the position and velocity of the mass in Cartesian co-ordinates. However, in a simulation, we may be interested in the latter information regardless of the mode, e.g. to ensure continuity when the pendulum breaks. Computing the Cartesian position given the angle of deviation is easy enough using basic trigonometry, whereas computing the velocity is most naturally achieved by simply differentiating the position. The automatic differentiation machinery allows the user to do the latter directly. Without it, the user would either have had to calculate the derivatives symbolically by hand, or he would have had to reformulate the model to use Cartesian state also in the non-broken mode.

To develop a Yampa model for the non-broken mode, we first have to determine the state variables and the causality, or data flow direction, and rework the model equations into causal (directed) form, where the state variables are computed by integration. As was noted above, ϕ and $\dot{\phi}$ are our state variables in this mode. If we consider the dot notation more as a way to name variables rather than a differentiation operator, we obtain:

$$\begin{aligned} \phi &= \frac{\pi}{4} + \int \dot{\phi}\,\mathrm{d}t & \qquad x &= l\sin(\phi) \\ \dot{\phi} &= \int \ddot{\phi}\,\mathrm{d}t & \qquad y &= -l\cos(\phi) \\ \ddot{\phi} &= \frac{u - mgl\sin(\phi)}{ml^2} & \qquad \dot{x} &= \frac{\mathrm{d}}{\mathrm{d}t}x \\ & & \qquad \dot{y} &= \frac{\mathrm{d}}{\mathrm{d}t}y \end{aligned}$$

We can now obtain a Yampa model for the non-broken mode by transliterating these equations into Haskell, employing the syntactic sugar for arrows. Despite some syntactic noise, the one-to-one correspondence should be obvious.

```
nonBroken :: SF Torque (Position2, Velocity2)
nonBroken = proc u -> do
  rec
    phi  <- (pi/4 +) ^<< integralC -< phid
    phid <- integralC              -< phidd
    let
        phidd = (u - m*g*l*sin phi) / m*l*l
        x     = l * sin phi
        y     = -l * cos phi
        xd    = derC x
        yd    = derC y
  returnA -< ((x, y), (xd, yd))
```

Figure 4 shows the simulation results with no externally applied torque and the pendulum breaking after 9 s.

5. DISTRIBUTION THEORY

There are many functions that do not have derivatives in the classical sense. For example, consider the unit step function (or Heaviside step function) $H(t)$:

$$H(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 & \text{if } t \geq 0 \end{cases} \tag{2}$$

The derivative of $H(t)$ in the usual sense simply is not well-defined at $t = 0$.

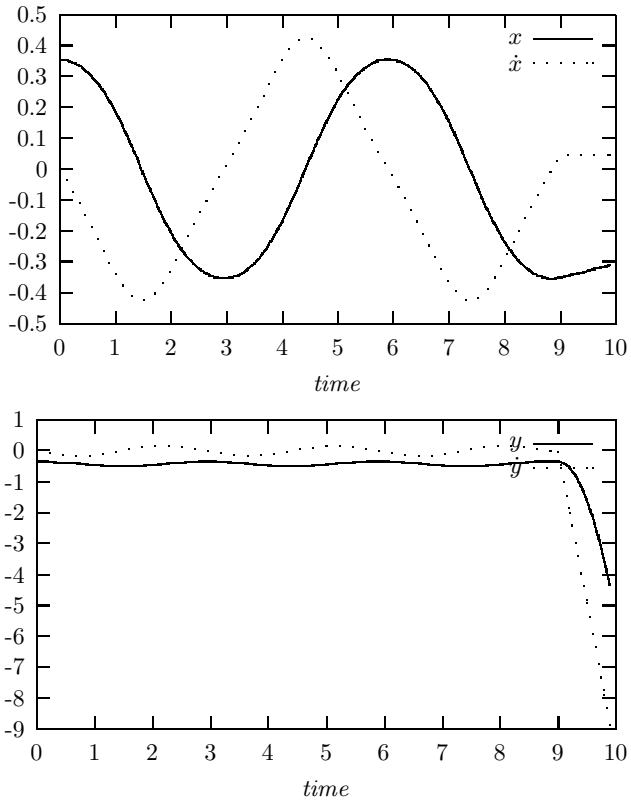


Figure 4: Simulation of the breaking pendulum

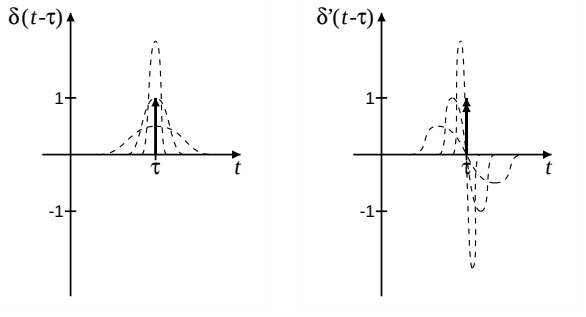
Nevertheless, if we consider a sequence $f_n(t)$ of ever narrower and taller impulses whose area is exactly 1, for example like in figure 5(a) with $\tau = 0$, it should be clear that the sequence of functions $H_n(t)$ defined by

$$H_n(t) = \int_{-\infty}^{t} f_n(\tau)\mathrm{d}\tau \tag{3}$$

will become better and better approximations of $H(t)$ as $n \to \infty$. Thus it is, according to the fundamental theorem of calculus, intuitively appealing to think of the limit of a function sequence like $f_n(t)$ as the derivative of the unit step function. This limit is called the *(Dirac) delta "function"* (despite not being a function in the usual sense) or *unit impulse*, and is denoted by $\delta(t)$. In a similar way, the limit of a sequence of functions like those shown in figure 5(b) is an intuitive way to understand the derivative $\delta'(t)$ of the unit impulse.

When drawing diagrams, it is conventional to represent impulses with an arrow, as in figure 5(a). As impulses can be scaled, the height of the arrow represents the area or *strength* of the impulse. Similarly, as shown in figure 5(b), we will draw the impulse derivative using a double-headed arrow, whose height corresponds to the strength of the differentiated impulse. The second impulse derivative will be drawn as a triple-headed arrow.

By analyzing the limits of operations on sequences of functions that tend to the unit impulse, properties that should



(a) Dirac impulse (b) Impulse derivative

Figure 5: Dirac impulse $\delta(t-\tau)$ and its derivative $\delta'(t-\tau)$ as the limits of function sequences

hold for reasons of *consistency* emerge. For example:

$$\int_a^b \delta(t)\,\mathrm{d}t = \begin{cases} 1 & \text{if } 0 \in (a,b) \\ 0 & \text{if } 0 \notin [a,b] \end{cases} \tag{4}$$

More generally, given any function $\phi(t)$ continuous at the origin, a central property is that

$$\int_{-\infty}^{\infty} \delta(t)\phi(t)\,\mathrm{d}t = \phi(0) \tag{5}$$

The theory of *distributions* or *generalized functions* [19, 6] is a rigorous formalization of the ideas outlined above. The key step is to consider functions that map a certain class of *functions* to numbers (i.e., a kind of higher-order functions), instead of functions that map numbers to numbers. This allows a consistent theory to be developed in which every distribution has a derivative that is also a distribution.

Formally, a distribution is a linear and continuous *functional*. A functional is a function that maps so called *test functions*, infinitely differentiable functions that vanish outside a finite interval, to numbers. The application of a functional T to a test function ϕ is conventionally written $\langle T, \phi\rangle$ or sometimes $\langle T(t), \phi(t)\rangle$. Two functionals S and T are *defined* to be equal if

$$\langle S, \phi\rangle = \langle T, \phi\rangle \tag{6}$$

for every test function ϕ. A functional T is linear if

$$\langle T, c_1\phi_1 + c_2\phi_2\rangle = c_1\langle T, \phi_1\rangle + c_2\langle T, \phi_2\rangle \tag{7}$$

Continuity is analogous to the usual notion of continuity, but in terms of limits of sequences of test functions instead of limits of sequences of numbers.

Every locally integrable function f induces a distribution T_f through the following definition:

$$\langle T_f, \phi\rangle = \int_{-\infty}^{\infty} f(t)\phi(t)\,\mathrm{d}t \tag{8}$$

Such distributions are said to be *regular*. Now, the unit impulse is not a function in the usual sense, so (8) is not applicable. But (5) shows what to expect in the limit for a sequence of functions tending to the impulse. Thus, for reasons of consistency, the delta distribution is *defined* as

$$\langle \delta, \phi\rangle = \phi(0) \tag{9}$$

Differentiation is *defined* through

$$\langle T', \phi\rangle = -\langle T, \phi'\rangle \tag{10}$$

Thus, in particular, we have that the impulse derivative is

$$\langle \delta', \phi\rangle = -\langle \delta, \phi'\rangle = -\phi'(0) \tag{11}$$

or in general

$$\langle \delta^{(k)}, \phi\rangle = (-1)^k\langle \delta, \phi^{(k)}\rangle = (-1)^k\phi^{(k)}(0) \tag{12}$$

Shifting is defined by

$$\langle T(t-\tau), \phi(t)\rangle = \langle T(t), \phi(t+\tau)\rangle \tag{13}$$

Unfortunately, the theory of distributions does not generalize the theory of classical functions in every respect. For example, it is not possible to define the product of two distributions in general. However, it is possible to define multiplication with a C^∞ (infinitely differentiable) function:

$$\langle gT, \phi\rangle = \langle T, g\phi\rangle \tag{14}$$

provided $g \in C^\infty$.

6. GENERALIZED SIGNALS

In this section, we will see how the theory of distributions allows us to apply automatic differentiation to *signals* in a hybrid simulation setting. Note that we are not trying to develop a generalized differential algebra independently of that setting. The present development focus on the easier problem where discontinuities ultimately stem from mode switching or explicit impulses.[5]

Let us first see how the theory of distributions helps us differentiate piecewise continuous functions.

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

Figure 6 shows these (generalized) functions graphically. Note how the derivatives are described by the sum of, on the one hand, a classical function, and, on the other, impulses (and impulse derivatives) to account for discontinuities (and impulses) in the differentiated function.

Turning to our original problem, differentiation of piecewise continuous signals in Yampa (and, in turn, their derivatives), it should now be clear that we can address this if we conceptually regard signals not as functions of time, but as *generalized* functions of time of the following form:

$$S(t) = s(t) + \sum_{i=0}^{m}\sum_{j=1}^{n} a_{ij}\delta^{(i)}(t-\tau_j) \tag{15}$$

[5]As suggested by one of the anonymous reviewers, it may be possible to develop such a generalized differential algebra. That would allow functions like `abs` that introduce discontinuities to be handled properly. Currently functions like `abs` should not be used. Substitutes, defined in terms of switching, are provided, but those substitutes are consequently stateful signal functions as opposed to pure functions in a differential algebra.

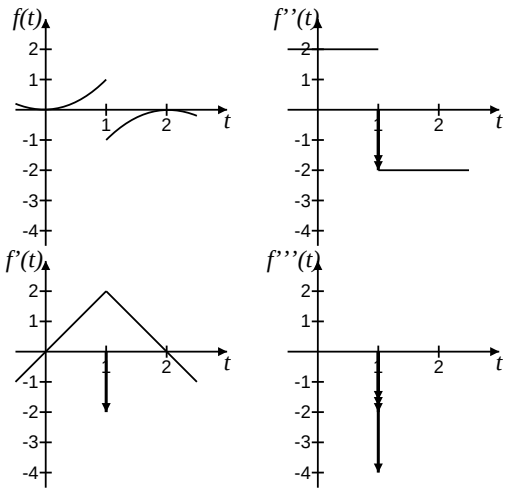


Figure 6: Differentiation of generalized functions

where $s(t)$ is a regular signal (piecewise continuous function) and m is the order of the highest impulse derivative. We call this a *generalized signal*. τ_j, $j \in [1, n]$ are the points in time at which the generalized signal or its derivatives have discontinuities.

At the implementation level, what we need to represent is samples of generalized signals. Let Δ be the set of points of discontinuity:

$$\Delta = \{\, \tau_j \mid j \in [1, n] \,\} \tag{16}$$

Let S_t be the representation of a sample of a generalized signal S as defined by (15) at time t. For sample times $t \notin \Delta$ we have

$$S_t = s(t) \tag{17}$$

For $t = \tau_j \in \Delta$ we choose to represent the sample by a pair of the *left limit* of the regular part s at t, together with a list of the *strengths* of those impulses and impulse derivatives that are non-zero at t, where the nth list element represents the strength of the impulse derivative of order n:

$$S_t = (s(t-),\, [a_{0j}, a_{1j}, \ldots, a_{mj}]) \tag{18}$$

What about the *right limit* and representing discontinuities? We will return to that question shortly.

Underlying this representation is an assumption that Δ includes a point for every *theoretical* point of discontinuity, and that this point is sufficiently close to its theoretical value. Ensuring this is a hard but separate problem that has received widespread attention in the hybrid modeling and simulation literature.

For automatic differentiation, as described earlier, the representation of a sample must include the value of the signal as well as the values of the derivatives of the signal at the sample point. We choose to keep the regular part and the impulse part of a sample separate, allowing us to reuse the the type `C` introduced in section 2, and giving us an easy way to test for the absence of impulses.The following type `G` represents a sample of a *generalized* signal:

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

`C` is exactly as before; the `C`-part of `G` represents the value or left limit of the regular part s and its derivatives according to (17) or (18). Since we assume that Δ includes all points of discontinuity, these derivatives are well-defined. The `I`-part accounts for the impulses and impulse derivatives. It is either `NI` for "no impulse" in case (17) applies, or it is a *finite* list of impulse strengths as defined in (18) together with a value of type `I` representing the impulse part of the derivative of the *entire* generalized signal at the sample point. It turns out to be useful to maintain the invariant that the list of impulse strengths never end with zero. Unlike `C`, `I` should not be considered to have a meaning separate from `G`. This is because differentiation of the regular part can require further impulses to be added to the impulse part.

As an added benefit of explicitly including the value of the derivative in the representation, we get a way to represent points of discontinuity; i.e. points where the *right limit* is different from the left limit. We exploit the fact that a discontinuity gives rise to an impulse in the *derivative* of a signal, and take such an impulse to be the *representation* of a discontinuity. The right limit can thus be computed from the left limit by adding the strength of any impulse in the derivative.

Two basic selectors for `I` are `strengthI` and `derI`:

```
strengthI :: I -> Double
strengthI NI          = zeroStrength
strengthI (I []    _) = zeroStrength
strengthI (I (a:_) _) = a

derI :: I -> I
derI NI       = NI
derI (I _ i') = i'
```

Note that `derI` yields the impulse part of the derivative of the *generalized* signal sample of which a particular entity of type `I` is part.

`stepI` and `impulseI` computes the impulse parts for a step and an impulse at the point of discontinuity:

```
stepI :: Double -> I
stepI a | isZeroStrength a = NI
        | otherwise        = I [] (impulseI a)

impulseI :: Double -> I
impulseI a | isZeroStrength a = NI
           | otherwise        = impAux [a]
  where
    impAux as = I as (impAux (zeroStrength:as))
```

Addition of impulse parts is straightforward since distributions are linear:

```
addI :: I -> I -> I
addI i  NI = i
addI NI j  = j
addI (I as i') (I bs j') =
  I (addI' as bs) (addI i' j')
  where
    addI' as [] = as
    addI' [] bs = bs
    addI' (a:as) (b:bs) =
      consStrengths (a + b) (addI' as bs)
```

`consStrengths` is a list constructor that maintains the invariant that lists of impulse strengths should not end in zero (recall that the list of impluse strengths is finite).

`discI` computes the impulse part at a point of discontinuity given the left and right limits:

```
discI :: C -> C -> I
discI (C a x') (C b y') =
  stepI (b - a) `addI` (I [] (discI x' y'))
```

The product of a C^∞ function and an impulse or impulse derivative is considerably more interesting. At first, one might think that multiplication just amounts to a pointwise scaling. But a closer study of equations (12) and (14) reveals that that is only true if the C^∞ function is a *constant*; i.e., when its derivatives are identically zero.

Let us study the product of a function $f(t) \in C^\infty$ and $\delta^{(n)}(t-\tau)$. Our goal is to find the strengths of the resulting impulses and impulse derivatives at point τ, enabling us to construct a correct representation of the sample of the product signal at that point. The following derivation is just a straightforward application of equations (7) to (14):

$$\begin{aligned}
&\langle f(t)\delta^{(n)}(t-\tau), \phi(t)\rangle \\
&\quad = (-1)^n [f(\tau)\phi(\tau)]^{(n)} \\
&\quad = (-1)^n \sum_{k=0}^{n} \binom{n}{k} f^{(k)}(\tau)\phi^{(n-k)}(\tau) \\
&\quad = (-1)^n \sum_{k=0}^{n} \binom{n}{k} f^{(k)}(\tau)(-1)^{(k-n)}\langle \delta^{(n-k)}(t-\tau), \phi(t)\rangle \\
&\quad = \sum_{k=0}^{n} (-1)^k \binom{n}{k} f^{(k)}(\tau)\langle \delta^{(n-k)}(t-\tau), \phi(t)\rangle \qquad (19)
\end{aligned}$$

Thus, multiplying a C^∞ function f by an impulse derivative of order n shifted by τ, results in a sum of impulse derivatives of order 0 to n, *all* shifted by the same amount, where the strength for the impulse derivative of order $n-k$ is $(-1)^k \binom{n}{k} f^{(k)}(\tau)$.

There is a technical problem. The derivation above assumes that f is a C^∞ function; i.e., neither f nor its derivatives have any discontinuities. In our implementation, the role of f is played by some signal, and since the implementation is sampled, that implies that we only know whether a signal is C^∞ or not in the immediate neighborhood of the current sample point. Everywhere else, the signal could be arbitrarily ill behaved: there is no recollection of the past, and the future is unknown. However, intuitively, since only the behavior of a C^∞ function at the point of the impulse is relevant for the result, and since an impulse is identically zero everywhere else, the behavior of the signal elsewhere should not matter. But we have not yet proved this formally.

We now know how to obtain the impulse strengths for the product of a C^∞ function and arbitrary impulse derivatives. Thanks to linearity, we can thus multiply the impulse part of a generalized signal with a C^∞ function. But in our setting, we also need to construct the impulse part of the *derivative* of the product, and we would like to do that using *only* the impulse part of the generalized signal.

Let us denote the regular part of a generalized signal S by $\sim S$ and the impulse part by $\uparrow S$. Thus

$$S = \sim S + \uparrow S \qquad (20)$$

It is easy to show that the derivative of a product of a function $f \in C^\infty$ and a distribution S can be rewritten in the expected way:

$$[f(t)S(t)]' = f(t)S'(t) + f'(t)S(t) \qquad (21)$$

The impulse part of the derivative of the product can then be obtained as follows:

$$\begin{aligned}
&\uparrow [f(t)S(t)]' \\
&\quad = \uparrow [f(t)S'(t) + f'(t)S(t)] \\
&\quad = \uparrow [f(t)S'(t)] + \uparrow [f'(t)S(t)] \\
&\quad = \uparrow [f(t)(\sim S'(t) + \uparrow S'(t))] + \uparrow [f'(t)(\sim S(t) + \uparrow S(t))] \\
&\quad = \uparrow [\underbrace{f(t) \sim S'(t)}_{\text{no impulses}}] + \uparrow [f(t) \uparrow S'(t)] \\
&\qquad + \uparrow [\underbrace{f'(t) \sim S(t)}_{\text{no impulses}}] + \uparrow [f'(t) \uparrow S(t)] \\
&\quad = \uparrow [\underbrace{f(t) \uparrow S'(t)}_{\text{only impulses}}] + \uparrow [\underbrace{f'(t) \uparrow S(t)}_{\text{only impulses}}] \\
&\quad = f(t) \uparrow S'(t) + f'(t) \uparrow S(t) \qquad (22)
\end{aligned}$$

Now we are at long last in a position to define the product of a sample of a C^∞ signal and the impulse part of a sample of a generalized signal.

```
mulCI :: C -> I -> I
mulCI _ NI = NI
mulCI x@(C _ x') i@(I as i') =
  I (mciAux x as) (addI (mulCI x i')
                        (mulCI x' i))
  where
    mciAux x as = loop1 pascal as
      where
        loop1 _ [] = []
        loop1 (ps:pss) aas@(a:as) =
          consStrengths (loop2 1 ps x aas)
                        (loop1 pss as)

        loop2 _ _ _ [] = 0
        loop2 sign (p:ps) (C b x') (a:as) =
          sign * fromIntegral p * a * b
          + loop2 (negate sign) ps x' as

pascal :: [[Int]]
pascal = (repeat 1) : map (scanl1 (+)) pascal
```

`mciAux` computes the impulse part of the value of the product according to (19), whereas the impulse part of the derivative of the product is computed by invoking `mulCI` recursively according to (22). The summation ordering in `mciAux` has been optimized for reasons of efficiency. `pascal` is a version of Pascal's triangle that provides binomial coefficients in an order suitable for `mciAux`. The following equality holds:

$$\texttt{pascal!!}\, m \,\texttt{!!}\, n = \binom{m+n}{n} \qquad (23)$$

Let us finally define some operations on `G`. `leftLimit` and `rightLimit` are the left and right limits of a signal at the current point. The left limit is simply a projection by (17) and (18). The right limit is the same as the left limit if there are no impulses at the point in question. Otherwise it is

computed from the left limit by adding the strength of any impulse in the first derivative, as explained before.

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

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

We want to allow the user to explicitly introduce impulses. Since events and impulses share the property that they both occur at a specific point in time, and since Yampa has a rich sub-language for events, it seems natural to do this through a function mapping events to impulses. We chose to let the value tagged to the event specify the strength of the resulting impulse.

```
impulse :: Event C -> G
impulse e = G 0.0 i
  where
    i = case e of
          NoEvent   -> NI
          (Event x) -> impulseI (valC x)
```

Defining arithmetic operators like `+` and `*` on `G` is straightforward. The corresponding operators on `C` are used for the regular part and operations such as `addI` and `mulCI` for the impulse part. For multiplication, four cases are distinguished. The first two cover the cases where at least one of the factors and all its derivatives are free from impulses at the point in question. The third concerns the case where two regular signals that do not have regular derivatives are multiplied. This is defined by multiplying left and right limits separately, and using `discI` to compute an impulse part that accounts for any resulting discontinuities. The last case is an error case. However, it would be possible to generalize multiplication a bit further. For example, it is possible to define the product of a coinciding step and impulse.

```
instance Num G where
  (G x i) + (G y j) = G (x + y) (i `addI` j)

  (G x NI) * (G y j)  = G (x * y) (x `mulCI` j)
  (G x i)  * (G y NI) = G (x * y) (y `mulCI` i)
  u * v | isRegular u && isRegular v =
            G lluv (discI lluv rluv)
        | otherwise = error "Illegal mult."
    where
      lluv = leftLimit u * leftLimit v
      rluv = rightLimit u * rightLimit v
```

7. INTEGRATING GENERALIZED SIGNALS INTO YAMPA

In this section, we will see how to fit generalized signals and the associated generalized automatic differentiation machinery into Yampa. Again, integration is one of the fundamental capabilities that needs to be adapted. The main novelty compared with the development in section 4 is integration of impulses and impulse derivatives. As the antiderivative of an impulse is a step, integration across an impulse should cause a jump in the value of the integral, the size of which is given by the strength of the impulse. The signal function `integralG` integrates generalized signals (compare the implementation of `integralC` in section 4):

```
integralG :: SF G G
integralG = SF tf0
  where
    q0 = 0.0
    tf0 _ ~u@(G x0 i0) =
      (igrlAuxG (q0 + strengthI i0)
                (rightLimit u),
       G (C q0 x0) (integrateI i0))
    igrlAuxG q_prev x_prev = SF tf
      where
        tf dt ~u@(G x i) =
          (igrlAuxG (q + strengthI i)
                    (rightLimit u),
           G (C q x) (integrateI i))
          where
            q = q_prev + dt * valC x_prev

integrateI :: I -> I
integrateI NI             = NI
integrateI i@(I [] _)     = I [] i
integrateI i@(I (_:as) _) = I as i
```

The (recursive) calls to `igrlAuxG` are fairly straightforward. For example, consider the code fragment

```
igrlAuxG (q + strengthI i) (rightLimit u)
```

The strength of any impulse on the input is added to the internal integral state causing the desired jump (`strengthI` yields 0 if there is no impulse). Note that the *right limit* of the input is used for passing on what will be the previous value of the regular part of the input at the next step.

The construction of the output value is more subtle. For example, consider the fragment

```
G (C q x) (integrateI i)
```

If there is no impulse, the auxiliary function `integrateI` yields `NI`, and the value of the output signal and its derivatives at this point is simply given by the regular part `C q x`, where `q` is the value of the integral and `x` is the regular part of the input. This is exactly as for `integralC` in section 4. If there *is* an impulse, then `C q x` denotes the *left limit* of the output, and `integrateI` accounts for the jump by putting the input impulse into the *derivative* of the impulse part of the output. This is how a step is represented, and the right limit will thus have the correct value. Finally, any impulse *derivatives* in the input has to be integrated by putting their anti-derivatives into the output. This is accomplished by simply reducing the order of all impulse derivatives by one, which in our representation amounts to a simple tail operation (in `integrateI`) on the list of impulse strengths.

There is one crucial difference in the behavior of `integralG` compared with `integralC` (and `integral`). Recall that the output of `integralC` at any point in time only depends on inputs at *earlier* points in time. Recursive equations like the following (in arrow notation) are thus well defined:

```
x <- integralC -< x + 1
```

(In this particular case, the result is an exponentially growing signal as one would expect.)

In the same way, the *left limit* of the output of `integralG` has carefully been defined to only depend on earlier input values. In contrast, the *impulse part* of the output, and thus also the *right limit* of the output, by necessity depends on

the impulse part of the input at the *same* point in time. Thus `integralG` cannot be used in a recursive equation like the one above: the result would be bottom.

However, the ability to write recursive definitions is absolutely critical, especially in a modeling context since any interesting system of differential equations gives rise to a set of recursive definitions involving integrals. Our solution is to appeal to modeling knowledge in order to break the bad recursion. Certain variables (signals) in a model are usually required to be continuous or at least free from impulses because of the underlying physical reality. For example, the trajectories of physical bodies moving in space are typically required to be continuous. If such knowledge is stated explicitly, this can be exploited to make a signal independent of its impulse part since it is then *known* that there are no impulses. To catch modeling mistakes, it should be checked that there in fact are no impulses, but if this check is delayed until the *following* time step, everything will work out fine.

The following signal function allows the user to assert that a signal is free from impulses, and uses this knowledge to make the output independent of the impulse part of the input at the current time step:

```
assertNoImpulseG :: SF G G
assertNoImpulseG = SF tf0
  where
    tf0 _ (G x0 i0) =
      (aniAux i0, G x0 (noImp i0))

    aniAux i_prev = SF tf
      where
        tf _ (G x i) =
          (aniAux i, seq (checkNoImp i_prev)
                         (G x (noImp i)))

    -- A promise that there are no impulses.
    noImp i = (I [] (derI i))

    -- Check that we kept the promise.
    checkNoImp NI = ()
    checkNoImp (I [] _) = ()
    checkNoImp _ = error "assertion failed"
```

Ideally, assertions like this one should be inserted automatically where needed based on declaratively stated continuity assumptions, but that would probably require a more sophisticated language implementation strategy than an embedding in Haskell, as is currently the case for Yampa.

As noted in section 1, switching is what introduces discontinuities in signals in the first place (along with explicitly introduced impulses). The following version of the basic Yampa switching combinator `switch` (see section 3) ensures that any discontinuities resulting from switching is accounted for by introducing impulses in the overall output:

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

The arguments are as for `switch`. The impulse part of the output at the point of switching is computed by applying `discI` to the left and right limit of the output signal at that point; i.e., the last output from the signal function being switched out and the first output from the signal function being switched in. How to handle any impulses in the outputs of the subordinate signal functions *at* the point of switching has not yet been satisfactorily resolved. Currently such impulses are ignored. The implementation details are omitted.

Finally, let us look at an example that makes use of some of the new capabilities introduced above. We return to the bouncing ball example from the introduction. By using impulses we can model the ball very concisely. The variable y represents the height above the floor, and its derivative $\dot{y}$ the velocity. The constant y_0 is the initial height. We allow ourselves to use the notation $\delta(y \leq 0)$ for introducing impulses whenever the predicate $y \leq 0$ becomes true, and the notation $\dot{y}(t-)$ for the left limit of $\dot{y}$.

$$
\begin{aligned}
y &= y_0 + \int \dot{y}\,\mathrm{d}t \\
\dot{y} &= \int -9.81 + (-2)\dot{y}(t-)\delta(y \leq 0)\,\mathrm{d}t
\end{aligned}
$$

Thus, whenever the ball hits the floor, it will be subjected to a force that instantaneously accelerates it by twice its current velocity in the direction opposite to the current velocity. Note that "current velocity" really means the velocity immediately prior to impact. That is why the left limit of $\dot{y}$ has to be used.

Transliterating these equations into Yampa yields:

```
bouncing ::
  Position -> SF () (Position, Velocity)
bouncing y0 = proc () -> do
  rec
    y   <- (y0 +) ^<< integralG -< yd_ni
    hit <- edge                 -< y <= 0
    yd  <- integralG
        -< -9.81 +
           impulse (hit `tag` (-2*leftLimit yd))
    yd_ni <- assertNoImpulseG -< yd
  returnA -< (y, yd)
```

Figure 1(b) is a plot of the simulation result from this model.

Ignoring the distinction between `yd` and `yd_ni`, the correspondence between the equations for y and $\dot{y}$ and the code above is pretty direct. The only major difference is the use of `assertNoImpulseG`. As discussed above, `integralG` cannot be used in recursive definitions without asserting that one of the recursively defined signals is free from impulses. Here we have chosen to make this assertion for the velocity, `yd`, yielding the signal `yd_ni` that safely can be used recursively since it is known that it is free from impulses.

Without impulses, the bouncing ball would have to be modeled using explicit mode switching, for example as shown in section 3. That model has a considerably more complicated structure and is arguably less declarative than the impulse-based one.

8. CONCLUSIONS

This paper showed how to add support for Dirac impulses to a causal hybrid simulation system, and how to integrate this with the technique of automatic differentiation, ensuring everywhere correct derivatives even for signals that are only piecewise continuous. The development required introducing the notion of *generalized* signals, i.e. signals that conceptually are distributions rather than classical functions, and extending arithmetic operations to work pointwise on such signals, computing the value of the result as well as its

derivatives at each point. Multiplication turned out to be a somewhat tricky case and required extra care.

The paper also demonstrated the utility of impulses and automatic differentiation for causal (hybrid) modeling. We expect that these techniques could be even more useful in a *non-causal* setting, which is our ultimate goal [14], where the techniques would be employed transparently behind the scenes as needed depending on the choice of state variables and the resulting causality.

For computationally demanding simulation applications, the present system is really only a proof of concept: lazy evaluation in itself would usually be considered too expensive, and the somewhat elaborate representation of samples of generalized signals does not make it any more efficient. However, in a compilation-based implementation of a simulation language, it ought to be possible to infer statically how many derivatives that needs to be computed for the various variables, and the extent to which it is necessary to keep track of impulses. Perhaps a suitable type system could help. It should then be possible to generate reasonably efficient simulation code.

Nevertheless, functional programming and lazy evaluation made it possible to implement quite sophisticated simulation techniques remarkably concisely. As a consequence, exploring various design alternatives was also easy and did not require too much work, something that was very important during the development.

9. ACKNOWLEDGEMENTS

The author would like to thank the members of the Yale Haskell group, in particular John Peterson and Antony Courtney, as well as Valery Trifonov and Ana Bove, who all in their own way have contributed to this paper. Thanks also to the anonymous reviewers for very detailed and constructive feedback.

10. REFERENCES

[1] François E. Cellier. Object-oriented modelling: Means for dealing with system complexity. In *Proceedings of the 15th Benelux Meeting on Systems and Control, Mierlo, The Netherlands*, pages 53–64, 1996.

[2] George F. Corliss. Automatic differentiation bibliography. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 331–353. SIAM, Philadelphia, Pennsylvania, USA, 1991. Updated on-line version: `http://liinwww.ira.uka.de/bibliography/-Math/auto.diff.html`

[3] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP'97: International Conference on Functional Programming*, pages 163–173, June 1997.

[5] Hilding Elmqvist, François E. Cellier, and Martin Otter. Object-oriented modeling of hybrid systems. In *Proceedings of ESS'93 European Simulation Symposium*, pages xxxi–xli, Delft, The Netherlands, 1993.

[6] G. Friedlander and M. Joshi. *Introduction to the theory of distributions*. Cambridge University Press, 1998.

[7] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logics in Computer Science (LICS 1996)*, pages 278–292, 1996.

[8] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[9] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[10] Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, March 2001.

[11] John H. Mathews. *Numerical methods for mathematics, science, and engineering*. Prentice-Hall, 2nd edition edition, 1992.

[12] Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In Fritz W. Vaadrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control '99*, number 1569 in Lecture Notes in Computer Science, pages 165–177, 1999.

[13] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[14] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

[15] Ross Paterson. A new notation for arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, Firenze, Italy, September 2001.

[16] Izzet Pembeci, Henrik Nilsson, and Greogory Hager. System presentation – functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Principles and Practice of Declarative Programming (PPDP'02)*, pages 168–179, Pittsburgh, Pennsylvania, USA, October 2002.

[17] John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conference on Robotics and Automation*, May 1999.

[18] John Peterson, Paul Hudak, Alastair Reid, and Greg Hager. FVision: A declarative language for visual tracking. In *Proceedings of PADL'01: 3rd International Workshop on Practical Aspects of Declarative Languages*, pages 304–321, January 2001.

[19] Ian Richards and Heekyung Youn. *Theory of distributions: a non-technical introduction*. Cambridge University Press, 1990.