Full Presentation: First Class Representation of Differential Equations

Gershom Bazerman, Deutsche Bank <gershomb@gmail.com>

Lazy, functional languages excel at the direct representation of recursively defined values. Not only can we map expressions to equations corresponding to their value, but many mathematical equations, including those involving discrete recurrence relations, can be directly mapped to expressions. This is precisely what makes the infinite Fibonacci sequence such a compelling "Hello World" for Haskell. But when one moves from discrete to continuous equations, the picture is less rosy. There is no straightforward way (outside of the purely syntactic) to even represent the differential equations describing the exponential function ($f(x) = f'(x)$, $f(0) = 1$), much less anything more interesting. Not only would we like to make it easier for engineers and scientists to code with differential equations directly, but a mechanism for dealing with first-class continuous functions would also be directly applicable to a number of current fields of interest for computer science, including computer graphics, constraint-based programming, and functional reactive programming.

One possible approach to this problem is through the use of lazy splines—streams composed of lists of polynomials (described through their coefficients) and their durations. These can provide arbitrarily accurate approximations to a wide range of functions. As long as they depend only on prior and not current values, recursively defined functions may be defined directly. This makes it immediately possible to describe delay differential equations, which arise in a range of modeling and simulation problems. When we then introduce a method of forward extrapolation, ordinary differential equations may be handled as well. In fact, it transpires that this method loosely corresponds to a standard method for numerical solution to differential equations—the forward (or explicit) Adams method [1]. The addition of additional combinators to control the flow of information in turn yields a loose analogue to the Runge-Kutta family of solvers. This approach bridges an important gap–at once more symbolic and straightforward to reason about than heavy-duty packages typically implemented in languages such as Fortran, and more numeric and hence general-purpose than approaches relying solely on symbolic manipulation. Although less powerful than Automatic Differentiation (because more approximate), this technique can nonetheless be said to do for ordinary differential equations roughly what AD does for pure differentiation—shifting the terrain from implementing math to representing executable math.

The presentation will provide a brief discussion of differential equations (as Computer Scientists are more accustomed to differentiating data structures than standard mathematical functions), and, using the above equations for the exponential function and the equations for a simple spring-mass system [2] as running examples, demonstrate how the lazy spline approach arises naturally from attempting to represent ordinary differential equations, and from the fundamental theorem of calculus.

The lazy spline approach lends itself to many possibilities for future work. Areas of work include generalizations to multivariate and partial differential equations, formalizing error bounds in approximations, abstracting an approach to implicit methods, and developing mechanisms to encode appropriate invariants about differential equations at the type level.

Haskell code demonstrating the lazy spline approach, developed by myself and Jeff Polakow, may be found on Hackage [3]. Additionally, a presentation on this topic was given to Lisp-NYC in November 2009, for which video and code/slides are also available [4].

[1] http://en.wikipedia.org/wiki/Linear_multistep_method#Multistep_Method_Families
[2] http://en.wikipedia.org/wiki/Harmonic_oscillator#Spring.E2.80.93mass_system
[3] http://hackage.haskell.org/package/lazysplines
[4] http://www.lispnyc.org/meetings