Declarative Real-World Abstractions

Paul Hudak
Department of Computer Science
Yale University

P.O. Box 208285
New Haven, CT 06520-8285
paul.hudak@yale.edu

November 2, 2001

Abstract

Abstraction is often described as the key to effective software design, and *declarative* approaches to software design are often promoted as the most clear and succinct. Unfortunately, language and systems designers often get caught up in the their own little worlds, inventing abstractions and declarative techniques that are justified based on intrinsic values known only to language and systems design specialists. What the designers have forgotten about is the *real world*. What we should be striving for are declarative abstractions of real world phenomena. In t his paper several examples of such abstractions are described in brief.

Introduction

Modern programming languages are full of promising abstraction techniques: functions and procedures, objects and classes, modules and namespaces, functors and monads, higher-order functions and types, abstract and algebraic data types, and so on. With them the task of writing complex programs is simplified, modularity is enhanced, and code reuse is facilitated. Indeed, abstraction techniques are the closest thing language designers have to a "silver bullet" for solving modern software development problems.

Unfortunately, most of these abstraction techniques are motivated from an inward-looking perspective, or from mathematical foundations that are allegedly self-evident. Inward-looking abstractions are improvements to previous abstraction techniques, and are designed with the assumption that the previous techniques were on the right track. Ones based on mathematical principles, or that model mathematical structures, are designed with the assumption that mathematics is a suitable basis for programming. In fact both of these assumptions may be true, at least to some extent, but are these the only foundations on which to design new abstractions?

In this paper I argue that we should be looking to the *real world* for guidance in designing new abstractions. Indeed, one could argue that the success of object-oriented programming stems from the fact that objects are motivated, to a great extent, by the real-world notion of an object. I argue that there are also other natural real-world phenomena that are not only intuitive, but are also powerful, useful, and quite effective in designing modern software systems.

Time

Perhaps one of the simplest and most obvious aspects of the real world is that it is continuously changing over *time*. It is natural for us to speak not only about what is happening at the present, but also in terms of the future and the past. Yet continuous time as a truly first-class concept is conspicuously absent from most programming languages, which instead tend to focus on discrete concepts that at best depend implicitly on time. Many applications have critical components that are best described as time-varying behaviors. Examples include animation and multimedia, signal processing, robotics, control systems, computer vision, and graphical user interfaces. These are some of the most challenging software development domains, and could benefit greatly from a suitable abstraction of continuous time.

To see this concretely, suppose that **time** is a value that represents the passage of time in seconds. The value **sin(pi*time)** then denotes a continuous sinusoidal signal with a period of two seconds. In this way complex time-varying *behaviors* could be defined easily and succinctly. Furthermore, with first-class time-varying behaviors, one could perform various time-based computations, such as

integration and differentiation. For example, the position of a body with mass **m** being accelerated by a force **f** could be defined as:

x = x0 + integral v
v = v0 + integral (f/m)

where **x0** and **v0** are the initial position and velocity, respectively. These are exactly the equations that one would write to describe the physical system. It is easy to imagine how similar concepts could be used to describe animations, to control robots, or in general, to realize control systems typically described as recursive integral / differential equations.

Furthermore, time transformations could be performed with simple expressions such as **timetransform(f,b)**, which transforms **b** in time according to the function **f**. Formally, if **b(t)** is **b**'s behavior at time **t**, then **b(f(t))** is the behavior of **timetransform(f,b)** at time **t**.

Interactivity

Interaction in the real world is pervasive. People interact with all sorts of objects, including computers, and objects interact with each other. This *interactivity* (a word that I prefer over *reactivity*, which conjures an image of one-directional interaction) is qualitatively different from continuous behavior, and is characterized by the occurrence of discrete *events*. These events can be viewed as the discontinuities that link together otherwise continuous behaviors. A ball falls gracefully under the influence of gravity, only to be abruptly changed through interaction with the floor. A heating system behaves smoothly when on, heat dissipates smoothly when the system is off, and the hysteresis of a thermostat is what links them.

A programming language should have rich abstractions to describe interactivity, but most are limited to very sequential process- or event-based reactivity that lacks clarity and succinctness. What is needed is a more *declarative* approach to interactivity. For example, using the previous examples, a bouncing ball can be described declaratively as:

y = y0 + integral v
v = v0 + integral g `until` (y==0) ➔ (-v)

(Interestingly, this is an area where programming language design could have an impact on methods for describing physical systems. In most physics textbooks, for example, interactions such as these are at best described informally and piecemeal, with no closed-form equation to describe the entire behavior.)

In the above example, **(y==0)** is a *predicate event*. There are many other kinds of events too: interactions with the user (mouse, keyboard, etc.), ethernet messages, robot sensors, and so on. They can all be captured within this same framework.

Non-Causality

It is far too easy when modeling the real world to assume that everything is *causal,* which leads one to treat most objects as black boxes that receive input at one end, and produce output at the other. But in fact, it is sometime more natural to connect objects together in a *non-causal* manner, with bi-directional connections rather than unidirectional. When done in this way, properties of the connections themselves contribute to the overall system behavior. Two black boxes with ports **a** and **b** could be connected with a declaration such as:

connect (box1.a, box2.b)

This style of programming is reminiscent of logic programming, where logical variables play the role of both input and output. But here, the intent is that **connect** itself is a user-defined connection that specifies the overall behavior of the system.

Real-Time

I've talked about time, and I've talked about the real world, but I have not yet talked about *real-time behavior.* Real-time programming is perhaps one of the most difficult domains in which to work, and is also one of the most important. Many life-critical systems depend intimately on proper real-time behavior. As with the concepts discussed above, I believe that we need a more declarative approach to dealing with real-time and, in general, resource-bounded computation. This can be achieved in one of two ways: either *implicitly,* through a restricted language for which resource bounds can be guaranteed, or *explicitly* through a design whereby the user specifies resource constraints and (either manually or automatically) constructs a program guaranteed to meet those constraints.

Conclusions

Not surprisingly in a white paper such as this, my research group has been working on programming language abstractions along the lines of what has been described herein (see some references below). For example, we have been very successful in designing languages that capture time and reactivity, and have successfully applied those languages in the areas of animation, robotics, and general control. However, we don't nearly have all the answers. There are many open problems such as performance issues, interactions between large numbers of objects, and so on. We have just recently made

some progress on real-time behavior, but have only just started thinking about non-causal behavior. I have tried to suggest avenues for exploration that will yield answers different from the ones that we currently have, and to ask questions that we have not previously raised. What ties all of these ideas together is the emphasis on creating abstractions of the real world since, after all, it is within that world that our programs run and with which they interact.

References

P. Hudak, *The Haskell School of Expression – Learning Functional Programming through Multimedia*, Cambridge University Press, New York, 2000.

Z. Wan and P. Hudak, Functional Reactive Programming From First Principles, in *Proceedings of Symposium on Programming Language Design and Implementation*, ACM Press, 2000.

J. Peterson, G. Hager, and P. Hudak, A Language for Declarative Robotic Programming, Int'l Conference on Robotics and Automation, 1999.

J. Peterson, P. Hudak, and C. Elliott, Lambda in Motion: Controlling Robots With Haskell, First International Workshop on Practical Aspects of Declarative Languages, SIGPLAN, January 1999.

A. Reid, J. Peterson, G. Hager, and P. Hudak, Prototyping Real-Time Vision Systems: An Experiment in DSL Design, International Conference on Software Engineering, May 1999.

C. Elliott and P. Hudak, Functional Reactive Animation, *Proceedings of the International Conference on Functional Programming,* 1998.

Z. Wan, W. Taha, and P. Hudak, Real-Time FRP, *Proceedings of the International Conference on Functional Programming,* 2001.