Towards a proof-theoretic understanding of reactive and interactive programming

Chris Martens

November 14, 2012

1 Introduction

With cutting-edge programming language technology, we can write complex specifications of real systems, execute those specifications, and even (for a large class of systems) state and prove theorems about them. These reasoning tools are based variously on dependent types (LF), substructural logic (linear, ordered, and affine, for the most part), logic programming, and focusing (and its role in forward- and backward-chaining proof search, corresponding to bottom-up and top-down logic programming).

The systems we are capable of expressing with this technology are, however, limited to those where we expect either a single execution or a universal characterization of a class of executions. The specifications represent a self-contained simulation that runs, computes an answer or set of answers, and finishes (or silently fails to terminate). A stack machine to evaluate a programming language is a common example of such a thing: it expects a closed program, containing all the information it needs to execute, and it derives (through the rules encoded in logic of the specification language) an evaluation of that program.

I'm interested in extending these techniques to reason about *reactive* and *interactive* programs, or (to frame it another way) *partial* specifications that may be written and reasoned about separately and compositionally.

2 Problem domain: reactive and interactive systems

A *reactive system* is one that accepts external inputs and behaves differently upon receipt of those inputs; an *interactive system* may respond more rapidly or proactively. Examples are everywhere. In computing, they manifest as the operating system, shell, a programming language's REPL, web browser, a game, or any application that accepts user input. Examples (ostensibly) outside computing include biological systems (living organisms), thermostats, other home appliances, and nuclear plants. In fact, I might guess that the *vast majority* of systems we want to reason about are reactive or interactive.

The "hello world" example of a reactive system is a toggle or lightswitch. An external signal can flip it from on to off and vice versa. Already, we can dissect such a problem into three parts: its input channel (the switch), its internal state (whether it is on or off), and its output channel (the bulb, or a printout of the state). This ontology is related to the common "model, view, controller" (MVC) way of conceptualizing interactive programs.

While the world has plenty of tools for writing such systems, very few have been developed for describing them in such a way that we can formally reason about their properties (for example, Hennessy-Milner sorts of properties, like that as long as the switch keeps receiving signals, there will always be a future in which the light is off).

As soon as several input or output channels are allowed, complexity skyrockets. We must worry about race conditions, shared state, and every other worry ever to plague concurrency. In these settings, reasoning tools are even more dire.

3 Starting point: linear logic programming

Focussing; substructuralness; fwd/backward chaining

Encoded examples

4 Proposed work

5 Goals and evaluation criteria