Towards a proof-theoretic understanding of reactive and interactive programming

Chris Martens

November 15, 2012

1 Introduction

This project seeks to establish a proof-theoretic, unifying approach to encoding and reasoning about reactive and interactive systems.

With cutting-edge programming language technology, we can write complex specifications of real systems, execute those specifications, and for certain classes of systems, state and prove theorems about them. These reasoning tools are based variously on dependent types (LF), substructural logic (linear, ordered, affine, modal), logic programming, and focusing.

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 continues uninterrupted). 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.

CLF (cite) was given that name as the "concurrent logical framework", and its proof theory is capable of expressing a wide range of concurrent systems, so long as we think of *concurrent* as a synonym for *nondeterministic*. What it fails to do is let us encapsulate a set of rules as a distinct process, reason about that process with respect to the interfaces of others, prove properties about its own interface, or otherwise give meaning to the partial program.

I'm interested in using the proof-theoretic techniques present in CLF as a starting point, extending and modifying them to reason about *reactive* and *interactive* programs, i.e. *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 real systems in the worlds that 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, the need for reasoning tools is even more dire.

3 Starting point: linear logic programming

Here I show how far one can get with encoding several examples in CLF and explain their limitations that I wish to resolve.

3.1 Separability

Here is an encoding of the toggle example in CLF:

```
state : type.
on : state.
off : state.

switch : state -> type.

sigtoggle : type.
sigtoggled : type.
```

```
t1 : switch on * sigtoggle -o {switch off * sigtoggled}.
t2 : switch off * sigtoggle -o {switch on * sigtoggled}.
```

By convention, names preceded by `sig` can be interpreted as “signals”, either inputs or outputs. `sigtoggle` represents some signal that comes from the outside. Note that no rule in this program generates a `sigtoggle`, they only consume it.

A CLF `#trace` of this program is capable of seeding it with any initial state for the switch and a finite number n of toggle signals. The execution of such a trace will carry out those n toggle flips and terminate. This is fine for writing tests of individual simulations, but doesn’t simulate the *dynamic* adding of signals to the context.

Instead, I’d like to write a separate process

```
siginit : type.
ext : sigtoggled -o {sigtoggle}.
ext-first : siginit -o {sigtoggle}.
```

That takes an initial entry point and sends a toggle, waits for the toggle to fire, and sends another toggle, repeatedly.

Then seeding the trace with

```
#trace * {siginit * switch off}.
```

creates an infinite (but coinductively characterizeable) whole program that runs infinitely.

Indeed, I can put all of these definitions in the same file and execute the whole program in CLF. But what I can’t do is encapsulate the pieces in any meaningful way. So far in the code, I can identify for each rule which “process” it belongs to, on the basis of which signals are on the left of a lolli and which are on the right. But I could of course add a rule like

```
sigtoggled -o {sigtoggled}
```

or

```
switch on -o {switch off}
```

and it wouldn’t be clear to which process they belong. Further, supposing we extend the program such that the toggle (which would no longer simply be a toggle) does a lot more internal computation, we would like the behavior of the program to avoid considering rules that manipulate that logic while we are “inside a different process”, i.e. when a `sigtoggled` is generated, we should know that we need not (at this time) consider any rules in the toggle’s program.

Thus I claim this leads to the first desired property of a framework for interactivity: *separability.*

3.2 Contingent transitions

In a separate document in this directory, `roar.pdf`, I demonstrate that linear logic does not adequately capture a notion of *contingent transition* or *read-only access to resources*. The idea is that you'd like to only fire a rule $A \multimap B$ under the condition C, where C is a set of linear resources that you demand of your current state (though of course the initial confusion to resolve is what we mean by "current state"). You might initially think to encode such a rule as $A \otimes C \multimap B \otimes C$, but this doesn't actually preserve the predondition as we might expect: $A \otimes (C_1 \oplus C_2) \multimap B \otimes (C_1 \oplus C_2)$ in the context with C_1, A can eventually transition to $(C_1 \oplus C_2), B$, but the information about which disjunct we started with is lost.

3.3 Broadcast

In the past several weeks I have begun to investigate a wider range of communication patterns that users of existing concurrent languages expect to express. One such pattern is that a process may receive or send the same signal from or to multiple sources.

In a blog post `http://lambdamaphone.blogspot.com/2012/11/interactivity-week-3.html` I present a notion of broadcast that CLF is incapable of expressing. The idea is that we can of course always produce a finite, tensored-together group of signals, but we can't describe that set generally – adding a new "neighbor" to the signal graph amounts to editing every rule that sends the signal.

We can write

```
handle : sigout N -o {Pi x. nbr x N -o sigin x}.
```

...but the `Pi` in that rule behaves more like an infinitary &, allowing us to instantiate it at just one node, rather than what we really want, an infinitary $\otimes$.

This in some sense just a special case of the Separability criterion, but one that particularly suggests some interesting proof theory.

4 Related work

Puredata (Pd) (cite) is a language for declaratively describing signal interactions as a graph: nodes of various data types and edges between their output and input ports. It is commonly used for programming generative sound, music, and visualizations. Being able to encode, execute and reason about the idioms present in such a language has been and I believe will continue to be fruitful in presenting challenges and examples.

Functional Reactive Programming (cite) is probably the most developed attempt at declarative programming for interactive settings. It is implemented mainly as libraries in Haskell and as Elm (cite), a new FRP language that's attractive because of the ability to write and run it in a browser. The fundamental notion in FRP is *time-varying values*; functions over such values behave like stream transformers.

A group at CMU (cite) recently explored the connection between the proof theory of linear logic (context transitions) and the reasoning methodologies used in process calculi a la Milner. An intermediate semantics they give uses a relation

$$\Delta \longrightarrow^{\sigma} \Delta'$$

Where σ is an "external" signal; there is a special symbol to represent internal "silent" transitions. This system is a proof device, but as a stepping stone between linear logic and process calculi it seems compelling, particularly as I want to frame certain signals as external.

Separately, also at CMU, Umut Acar and his student Stefan Muller submitted an NSF GRFP proposal to explore interactivity in the context of self-adjusting computation (cite). Their idea is that since SAC is a framework for re-running pieces of computation on new partial inputs, a programmatically-repeated SAC could efficiently process interactively-changing inputs. I plan to pay close attention to their work along these lines to understand how it lines up with the proof-theoretical viewpoint.

5 Proposed work

First, I intend to do further reading and investigation of FRP, SAC, puredata and other communication graph languages, while simultaneously encoding examples in CLF and imagined proto-syntax for a distinct language.

Then I plan to investigate what kind of metatheory or program-specific properties we desire about these programs, starting with standard process calculus metatheoretic techninques, including bisimulation, coinduction, and Henessy-Milner logic. I will sketch how I plan to approach this sort of reasoning within my framework.

I propose to build a prototype implementation of this framework and encode many examples.

6 Goals and evaluation criteria

To summarize the criteria I have in mind for an interactivity framework, it should be capable of expressing

- Separability/composability: distinct processes can be encoded, run, and reasoned about in isolation, even if they need external signals to make progress
- Contingent transitions, i.e. combine deductive, reflective reasoning on current states with transitions between those states

- Broadcast, and other multi-channel communication patterns.