Parallel Functional Reactive Programming

John Peterson, Valery Trifonov, and Andrei Serjantov

Yale University
peterson-john@cs.yale.edu, trifonov-valery@cs.yale.edu
andrei.serjantov@yale.edu

Abstract. In this paper, we demonstrate how Functional Reactive Programming (FRP), a framework for the description of interactive systems, can be extended to encompass parallel systems. FRP is based on Haskell, a purely functional programming language, and incorporates the concepts of time variation and reactivity.

Parallel FRP serves as a declarative system model that may be transformed into a parallel implementation using the standard program transformation techniques of functional programming. The semantics of parallel FRP include non-determinism, enhancing opportunities to introduce parallelism. We demonstrate a variety of program transformations based on parallel FRP and show how a FRP model may be transformed into explicitly parallel code. Parallel FRP is implemented using the Linda programming system to handle the underlying parallelism. As an example of parallel FRP, we show how a specification for a web-based online auctioning system can be transformed into a parallel implementation.

1 Introduction

A common approach to developing parallel programs is to express a sequential specification of the system in a declarative way and to then transform this model into a parallel implementation of it while preserving its semantics. In this work, we develop a framework for expressing models of interactive systems such as web servers and databases.

Our work is based on Functional Reactive Programming [PESL98] (FRP), a library of functions and types that extend Haskell [PJ99], a purely functional language, with means for describing interactive systems containing values that vary in time.

At the core of FRP are the notions of events and behaviors. An event of type Event a denotes a discrete series of occurrences in time, each having a timestamp and a value of type a, while a behavior of type Behavior b may be sampled at any time to yield a value of type b. FRP defines a rich set of functions operating on these datatypes, and is designed to retain the “look and feel” of pure functional Haskell without resorting to constructs such as monads to handle interaction.

To enable the introduction of parallelism into FRP programs, we have extended the basic framework with constructs which enrich the semantics of the models with non-determinism, representing the fact that the order in which two
computations on separate processors started does not determine the order in which they will finish.

To formalize the process of parallelizing FRP models, we introduce a number of equations which define valid transformations of sequential FRP constructs into parallel ones. Thus, transformations can be performed by means of a well understood meaning preserving process — equational reasoning. This is currently done by hand but possibly could be automated in the future.

2 Basic Concepts

We begin with a simple FRP model of a web server. This system receives requests for web pages (URLs) and reacts by generating events to post the resulting web pages.

```
server :: Event URL -> Event WebPage
server urls = urls ==> lookupWebPage
```

This server is a simple event transformer. That is, the URL in each incoming event is transformed into a WebPage. Event transformation is a primitive in FRP: the function

```
(==> :: Event a -> (a -> b) -> Event b
```

implements this directly. The actual web page generation is performed by the function `lookupWebPage`, a Haskell function that maps a URL onto a WebPage. We assume (for now) that the web pages are unchanging: thus, `lookupWebPage` is a pure function that does not perform IO. The semantics of FRP dictate that the resulting events of `==>` logically occur at the same time as the stimulus; that is, the clock (event times) associated with the web pages exactly matches the clock in the input event stream. For example, if the incoming event stream is

```
[(1, "f.com/a"), (3, "f.com/b"), (6, "f.com/c")]
```

then the output stream might be

```
[(1, "Page a"), (3, "Page b"), (6, "Page c")]
```

We represent event streams as lists of tuples, each tuple containing an occurrence time and a value. This is not necessarily how events are implemented, but it serves to illustrate the operation of the system. We are simplifying the problem somewhat: requests to a real server would carry for example the IP address and port identifying the client.

This model serves well for a single processor web server, but what about a parallel version of this problem? First, we observe that this system is stateless: that is, the generated web pages depend only on the incoming URL, not on previous transactions. We can infer this property directly from the specification: `==>` is stateless (by its definition in FRP) and the `lookupWebPage` function has no interactions with the outside world. Given this property, we can use two processors to serve the requests, dividing the incoming requests among the processors arbitrarily. Rewriting, the new server becomes: