Behaviours are Dual to Events

Philip Wadler
University of Edinburgh
Hudak Symposium, 29 April 2016
Functional Reactive Animation

Conal Elliott
Microsoft Research Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of behaviors and events. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform event detection using interval analysis is also described, which relies on the partial information structure on the domain of event times. Fran has been implemented in Hugs, yielding surprisingly good performance for an interpreter-based system. Several examples are given, including the ability to describe physical phenomena involving gravity, springs, velocity, acceleration, etc. using ordinary differential equations.

1 Introduction

The construction of richly interactive multimedia animations (involving audio, pictures, video, 2D and 3D graphics) has long been a complex and tedious job. Much of the difficulty, we believe, stems from the lack of sufficiently high-level abstractions, and in particular from the failure to clearly distinguish between modeling and presentation, or in other words, between what an animation is and how it should be presented. Consequently, the resulting programs must explicitly manage common implementation chores that have nothing to do with the content of an animation, but rather its presentation through low-level display libraries running on a sequential digital computer. These implementation chores include:

- stepping forward discretely in time for simulation and for frame generation, even though animation is conceptually continuous;
- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to date each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the “what” of an interactive animation, one can hope to then automate the “how” of its presentation. With this point of view, it should not be surprising that a set of richly expressive recursive data types, combined with a declarative programming language, serve comfortably for modeling animations, in contrast with the common practice of using imperative languages to program in the conventional hybrid modeling/presentation style. Moreover, we have found that non-strict semantics, higher-order functions, strong polymorphic typing, and systematic overloading are valuable language properties for supporting modeled animations. For these reasons, Fran provides these data types in the programming language Haskell [9].

Advantages of Modeling over Presentation

The benefits of a modeling approach to animation are similar to those in favor of a functional (or other declarative) programming paradigm, and include clarity, ease of construction, composability, and clean semantics. But in addition there are application-specific advantages that are in some ways more compelling, painting the picture from a software engineering and end-user perspective. These advantages include the following:

- Authoring. Content creation systems naturally construct models, because the end users of such systems think in terms of models and typically have neither the expertise nor interest in programming presentation details.

- Optimizability. Model-based systems contain a presentation sub-system able to render any model that can be constructed within the system. Because higher-level information is available to the presentation sub-system than with presentation programs, there are many more opportunities for optimization.

- Reusability. The presentation sub-system can also more easily determine level-of-detail management, as well as sampling rates required for interactive animations, based on scene complexity, machine speed and load, etc.
patOrbitsCharlotte =
  stretch wiggle charlotte `over`
  moveXY wiggle waggle pat
A Simple Example

First two bars expressed in Euterpea:

\[
\begin{align*}
&\text{(c 4 qu :+ d 4 qu :+ e 4 qu :+ f 4 qu :+)} \\
&\quad \text{(g 4 qu :+ f 4 qu :+ e 4 qu :+ d 4 qu)} :== \\
&\text{(e 4 qu :+ f 4 qu :+ g 4 qu :+ a 4 qu :+)} \\
&\quad \text{(b 4 qu :+ a 4 qu :+ g 4 qu :+ f 4 qu)}
\end{align*}
\]

- Verbose, but accurate.
- Can we do better?
- Yes, using various forms of \textit{abstraction}.
The Haskell School of Expression
LEARNING FUNCTIONAL PROGRAMMING THROUGH MULTIMEDIA
PAUL HUDAK
RIDICULOUSLY BRIEF HISTORY

Conal Elliott & Paul Hudak write Fran
A bunch of libraries are made (Reactive Banana, Yampa, etc)
Conal writes Push/Pull paper to address issues.
Eric Meijer & Friends make it more accessible starting with C#, then js
Lots of other frameworks made (Knockout, RxJs, Bacon, Flapjax)
I’m talking to you now
PROPOSALS FOR A MONUMENT TO
GIROLAMO CASSAR AND FRANCESCO LAPARELLI
PROPOSALS FOR A MONUMENT TO GIROLAMO CASSAR AND FRANCESCO LAPARELLI
LTL types FRP
Linear-time Temporal Logic Propositions as Types
Proofs as Functional Reactive Programs

Alan Jeffrey
Alcatel-Lucent Bell Labs
ajeffrey@bell-labs.com

Abstract
Functional Reactive Programming (FRP) is a form of reactive programming whose model is pure functions over signals. FRP is often expressed in terms of arrows with loops, which is the type class for a Freyd category (that is a premonoidal category with a cartesian centre) equipped with a premonoidal trace. This type system suffices to define the dataflow structure of a reactive program, but does not express its temporal properties. In this paper, we show that Linear-time Temporal Logic (LTL) is a natural extension of the type system for FRP, which constrains the temporal behaviour of reactive programs. We show that a constructive LTL can be defined in a dependently typed functional language, and that reactive programs form proofs of constructive LTL properties. In particular, implication in LTL gives rise to stateless functions on streams, and the “constrains” modality gives rise to causal functions. We show that reactive programs form a partially traced monoidal category, and hence can be given as a form of arrows with loops, where the type system enforces that only decoupled functions can be looped.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms Languages, Verification

Keywords Linear-time Temporal Logic, Functional Reactive Programming, Dependent Types

1. Introduction
Functional Reactive Programming (FRP) is a form of reactive programming whose model is pure functions over signals, which are time-dependent values. FRP was first introduced by Elliott in the context of functional animation [7, 9], and has since formed the basis of a number of reactive Domain Specific Embedded Languages, notably Fruit [5], Grapefruit [13] and Yampa [14].

Since the development of Yampa, FRP is often expressed in terms of arrows [15] with loops [26], which is the type class for a Freyd category [28] (that is a premonoidal category [29] with a cartesian centre) equipped with a premonoidal trace [3]. This type system suffices to define the dataflow structure of a reactive program, but does not express its temporal properties.

Sattar and Nilsson [32] have shown that FRP can be implemented in a dependently typed language, but their goals were rather different. Although their programs are reactive, their types are not, in contrast to ours. Moreover, they are interested in run-time safety, not logical soundness, and so they disable the termination checker.

In this paper, we show that Linear-time Temporal Logic (LTL), introduced by Pousel[27], is a natural extension of the type system for FRP, which constrains the temporal behaviour of reactive programs. In particular, the notions of stateless function, causal function and decoupled function which occur in FRP have natural expressions as LTL operators.

We show how LTL can be embedded in a dependently typed programming language such as Agda, and that LTL formulae can be used as reactive types for FRP programs, such that any well-typed program constitutes a proof of an LTL formula. Paraphrasing Curry–Howard, we consider LTL propositions as types, and proofs as FRP programs. This correspondence between LTL propositions and types for FRP was discovered simultaneously by Hetich [19].

We show that many FRP combinators can be given natural LTL types, and that the LTL types express the temporal behaviours of programs, for example allowing us to distinguish between a program which must be used immediately, and one which may be used at some point in the future. The most interesting typing is that of the loop combinator, which allows the construction of cyclic dataflow graphs. Its type is that of a partial trace [11], but it is derivable from the known result [23, 25] that LTL can be used to provide compositional proofs for cyclic in/it/guarantee systems.

As well as a model for FRP, we provide a sketch of an implementation, based on internal types, and makes use of an embedded process language describing the IO/behaviour of causal functions.

2. Model
In this section, we discuss the signals model of FRP, and show that LTL can be used as a type system for FRP programs, such that well-typed FRP programs constitute proofs of LTL formulae.

We give definitions in pseudo-Agda, making use of dependent functions, dependent products, and inductive and coinductive data types. Most of Agda’s notation is hopefully familiar, but we call attention to Agda’s notation for in/it/guaranteed arguments. The types:

\[ \forall [z:] A \quad \exists [z:] A \]

are universal (resp. existential) quantification over \( z \), which may occur free in \( A \). When instantiated, the witness for \( z \) may be provided explicitly:

\[ \alpha [w] \quad (\{ w \}, \alpha) \]
\( \bigcirc, \Diamond, \square : \text{RSet} \rightarrow \text{RSet} \)

\[(\bigcirc A)(t) = A(t + 1)\]
\[(\Diamond A)(t) = \exists \{u\} (t \leq u) \times A(u)\]
\[(\square A)(t) = \forall \{u\} (t \leq u) \rightarrow A(u)\]
An Abstract Categorical Semantics for Functional Reactive Programming with Processes

Wolfgang Jeltsch
TTU Kõhvero Instituut
Akadeemia tee 21, 12618 Tallinn, Estonia
http://www.wolfgang.jeltsch.info/

Abstract
Linear-time temporal logic and functional reactive programming (FRP) are related via a Curry–Howard correspondence. To prove proofs of “always,” “eventually,” and “until” propositions correspond to behaviors, events, and processes, respectively. Processes in the FRP sense combine continuous and discrete aspects and generalize behaviors and events. In this paper, we develop a class of axiomatically defined categorical models of FRP with processes. We call these models abstract process categories (APCs). We relate APCs to other categorical models of FRP, namely temporal categories and concrete process categories.

Categories and Subject Descriptors:
E3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; E4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Temporal logic

Keywords: Functional Reactive Programming, Curry–Howard Correspondence, Category Theory, Categorical Semantics

1. Introduction
Functional reactive programming (FRP) allows programs to deal with temporal aspects in a declarative way. FRP is based on behaviors and events, which denote time-varying values and values attached to times, respectively.

There is a Curry–Howard correspondence between FRP and intuitionistic temporal logic (Jeltsch 2012; Jeffrey 2012). By the type constructors for behaviors and events correspond to the “always” and “eventually” modalities. We have shown that the proofs of “until” propositions in temporal logic correspond to constructs that combine aspects of behaviors and events (Jeltsch 2013, Section 2). We call these constructs processes.

We have developed two kinds of FRP models, which can also be used to model temporal logic:

Temporal categories (Jeltsch 2012) model FRP with behaviors and events, but without processes. They are based on categorical models of intuitionistic S4 developed by Kobayashi (1997) and

Biermania and de Paiva (2000). Temporal categories are defined in a purely axiomatic way.

Concrete process categories (Jeltsch 2013) model FRP with processes. They are not purely axiomatic defined, but use concrete constructions to express time-dependence of type inhabitation and causality of FRP operations.

In this paper, we develop an axiomatic definition of categorical semantics for FRP with processes. We make the following contributions:

• In Section 2, we give a revised definition of temporal categories that fixes some issues of the original definition. We discuss the differences between the old and the new definition in Appendix A.

• In Section 3, we define abstract process categories (APCs), which are models of FRP with processes. Like temporal categories, APCs are axiomatic defined, but they contain more structure than temporal categories.

• In Section 4, we present several results about APCs. In particular, we show that every APC gives rise to a temporal category and that APCs generalize concrete process categories.

We discuss related work in Section 5 and give conclusions and an outlook on further work in Section 6.

2. Temporal Categories
Temporal categories model a variant of FRP that supports behaviors and events, but not processes. This FRP variant is based on a linear notion of time. However, the time scale does not need to be discrete; so the time domain can be any totally ordered set. Type inhabitation is time-dependent, which means that every FRP type corresponds to a time-indexed family of sets of inhabitants.

Besides the ordinary type constructors 1, 0, *, +, and →, there are two unary type constructors ⊕ and ⊕′ for behaviors and events. A value that inhabits a type ⊕ x at a time t corresponds to a function that maps each time t′ > t to a value that inhabits ⊕ at t′. Likewise a value that inhabits a type ⊕′ x at a time t corresponds to a pair of a time t′ > t and a value that inhabits ⊕ at t′.

The FRP variant described above corresponds to an intuitionistic temporal logic via a Curry–Howard isomorphism. Thereby time-dependent type inhabitation is related to time-dependent truthness of temporal propositions. The type constructors ⊕ and ⊕′ correspond to an “always” operator ⊕ and an “eventually” operator ⊕′, respectively. A proposition ⊕ϕ states that ϕ will hold at some future time, and a proposition ⊕′ϕ states that ϕ will hold at some future time.

In the remainder of this section, we describe how type constructors and operations of FRP can be modeled by categorical constructs, which ultimately leads to the definition of temporal categories.
The Essence of Event-Driven Programming

Jennifer Paykin¹, Neelakantan R. Krishnaswami², and Steve Zdancewic³
1 University of Pennsylvania, Philadelphia, USA
   jpaykin@as.upenn.edu
2 University of Birmingham, Birmingham, United Kingdom
   N.Krishnaswami@cs.bham.ac.uk
3 University of Pennsylvania, Philadelphia, USA
   steven@cis.upenn.edu

Abstract
Event-driven programming is based on a natural abstraction: an event is a computation that can eventually return a value. This paper exploits the intuition relating events and time by drawing a Curry-Howard correspondence between a functional event-driven programming language and a linear-time temporal logic. In this logic, the eventually proposition $\Diamond A$ describes the type of events, and Girard's linear logic describes the effective and concurrent nature of the programs. The correspondence reveals many interesting insights into the nature of event-driven programming, including a generalization of selective choice for synchronizing events, and an implementation in terms of callbacks where $\Diamond A$ is just $\neg\neg A$.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction
Event-driven programming is a popular approach to functional concurrency in which events, also known as "futures," "deferred values," or "lightweight threads," execute concurrently with each other and eventually produce a result. The abstraction of the event has been extremely successful in producing lightweight, extensible, and efficient concurrent programs. As a result, a wide range of programming languages and libraries use the event-driven paradigm to describe everything from message-passing concurrency [24] and lightweight threads [26] to graphical user interfaces [11] and I/O [13, 21, 25].

Although these systems vary considerably in the details of their implementations and APIs, they share a common, basic structure. They provide:

- the abstraction of the event, often given explicitly as a monad;
- the ability to synchronize across events;
- a continuation-passing style implementation in terms of callbacks; and
- a source or sources of primitive events.

In this paper we distill event-driven programming to its essence, demonstrating how to derive these components from first principles. Starting from a logical basis and building up the minimal machinery needed to explain the four points above, we proceed as follows:

The logic of events. In Section 2 we identify a Curry-Howard correspondence between events that eventually return a value and the $\Diamond$ ("eventually") modality from temporal logic. We define a core language of pure events where the monad from the event-driven abstraction is identified as $\Diamond$. We formulate the type system in the setting of Girard's linear logic to
Fair Reactive Programming

Andrew Cave Francisco Ferreira Prakash Panangaden Brigitte Pientka
School of Computer Science, McGill University, Montreal, Canada {acave1, ferrer8, prakash, bpieneka}@cs.mcgill.ca

Abstract

Functional Reactive Programming (FRP) models reactive systems with events and signals, which have previously been observed to correspond to the “eventually” and “always” modalities of linear temporal logic (LTL). In this paper, we define a constructive variant of LTL with least fixed point and greatest fixed point operators in the spirit of the modal mu-calculus, and give it a proofs-as-programs interpretation as a foundational calculus for reactive programs. Previous work emphasized the propositions-as-types part of the correspondence between LTL and FRP; here we emphasize the proofs-as-programs part by employing structural proof theory. We show that the type system is expressive enough to enforce liveness properties such as the fairness of schedulers and the eventual delivery of results. We illustrate programming in this calculus using co/iteration operators. We prove type preservation of our operational semantics, which guarantees that our programs are causal. We give also a proof of strong normalization which provides justification that our programs are productive and that they satisfy liveness properties derived from their types.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming

Keywords temporal logic; propositions-as-types; functional reactive programming; liveness

1. Introduction

Reactive programming seeks to model systems which react and respond to input such as games, print and web servers, or user interfaces. Functional reactive programming (FRP) was introduced by Elliott and Hudak [10] to raise the level of abstraction for writing reactive programs, particularly emphasizing higher-order functions. Today FRP has several implementations [16, 17, 22]. Many allow one to write unimplementable non-causal functions, where the present output depends on future input, and space leaks are all too common.

Recently there has been a lot of interest in type-theoretic foundations for (functional) reactive programming [16, 17, 12, 24] with the intention of overcoming these shortcomings. In particular, Jeffrey [16, 17] and Jeltsch [17] have recently observed that Poincaré’s linear temporal logic (LTL) [31] can act as a type system for FRP.

In this paper, we present a novel logical foundation for discrete time FRP with co/iteration operators which exploits the expressiveness afforded by the proof theory (i.e. the universal properties) for least and greatest fixed points, in the spirit of the modal mu-calculus [17]. The “always”, “eventually” and “until” modalities of LTL arise simply as special cases. We do this while still remaining relatively conservative over LTL.

Moreover, we demonstrate that distinguishing between and interleaving of least and greatest fixed points is key to statically guarantee liveness properties, i.e. something will eventually happen, by type checking. To illustrate the power and elegance of this idea, we describe the type of a fair scheduler — any program of this type is guaranteed to be fair, in the sense that each participant is guaranteed that his requests will eventually be served. Notably, this example requires the expressive power of interleaving least fixed points and greatest fixed points (a construction due to Park [19]) which is unique to our system.

Our approach of distinguishing between least and greatest fixed points and allowing for iteration and coiteration is in stark contrast to prior work in this area. Jeffrey’s work [16, 17] for example only supports less expressive combinators instead of the primitive co/recursion our system affords, and only particular instances of our recursive types. Krishnaswami et al. employ a more expressive notion of recursion, which entails unique (guarded) fixed points [12, 22, 24]. This implies that their type systems are incapable of expressing liveness properties, in contrast to our work. Jeltsch [17, 18] describes denotational settings for FRP, some of which are capable of expressing liveness guarantees. However, the theory of fixed points is largely undeveloped in his setting. Our technical contributions are as follows:

- A type system which, in addition to enforcing causality (as in previous systems [16, 22]), also enables one to enforce liveness properties; fairness being a particularly intriguing example. Moreover, our type system acts as a sound proof system for LTL. While previous work [16] emphasized the propositions-as-types component of the correspondence between LTL and FRP, the present work additionally emphasizes the proof-as-programs part of the correspondence through the lens of structural proof theory.

- A novel operational semantics which provides a reactive interpretation of our programs. One can evaluate the result of a program for the first $n$ time steps, and in the next time step, resume evaluation for the $(n+1)$st result. It allows one to evaluate programs one time step at a time. Moreover, we prove type preservation of our operational semantics. As a consequence, our foundation is causal: future inputs do not affect present results.

[Copyright notice will appear here once "preprint" option is removed.]
\[ \square A \equiv \nu X.A \times \Box X \]
\[ \Diamond A \equiv \mu X.A + \Box X \]

Rules for \( \Box \) modality and least and greatest fixed points

\[ \vdash \Theta \vdash M : A \quad \Theta ; \Gamma \vdash M : \Box A \quad \Theta, x : A ; \Gamma \vdash N : C \]
\[ \Theta ; \Gamma \vdash \mathbf{let} \ x = M \ \mathbf{in} \ N : C \]
\[ \Theta ; \Gamma \vdash M : [\mu X.F / X]F \quad \vdash x : [C / X]F \vdash M : C \quad \Theta ; \Gamma \vdash N : \mu X.F \]
\[ \Theta ; \Gamma \vdash \mathbf{inj} \ M : \mu X.F \]
\[ \Theta ; \Gamma \vdash \mathbf{iter}_{X.F} \ (x.M) N : C \]
\[ \vdash x : C \vdash M : [C / X]F \quad \Theta ; \Gamma \vdash N : C \]
\[ \Theta ; \Gamma \vdash \mathbf{coit}_{X.F} \ (x.M) N : \nu X.F \]
\[ \Theta ; \Gamma \vdash \mathbf{out} \ M : [\nu X.F / X]F \]
1 Introduction

Functional programmers know where they stand: upon the foundation of $\lambda$-calculus. Its canonicality is confirmed by its double discovery, once as natural deduction by Gentzen and again as $\lambda$-calculus by Church. These two formulations are related by the Curry–Howard correspondence, which takes

propositions as types,
proofs as programs, and
normalisation of proofs as evaluation of programs.

The correspondence arises repeatedly: Hindley’s type inference corresponds to Milner’s Algorithm W; Girard’s System F, corresponds to Reynold’s polymorphic $\lambda$-calculus; Peirce’s law in classical logic corresponds to Landin’s J operator (better known as call/cc).
\[
\frac{\Gamma, A \vdash \Delta}{\Gamma, \Box A \vdash \Delta} \quad \text{\textit{\Box}-L} \\
\frac{\Box \Gamma \vdash A, \Diamond \Delta}{\Box \Gamma \vdash \Box A, \Diamond \Delta} \quad \text{\textit{\Box}-R}
\]
\[
\frac{\Box \Gamma, A \vdash \Diamond \Delta}{\Box \Gamma, \Diamond A \vdash \Diamond \Delta} \quad \text{\textit{\Diamond}-L} \\
\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \Diamond A, \Delta} \quad \text{\textit{\Diamond}-R}
\]
\(\Box A = \text{Time} \rightarrow A\)
\(\Diamond A = \text{Time} \times A\)
\(\neg A = A \rightarrow \bot\)

\(\neg \Box \neg A\)
\(= (\text{Time} \rightarrow (A \rightarrow \bot)) \rightarrow \bot\)
\(= ((\text{Time} \times A) \rightarrow \bot) \rightarrow \bot\)
\(= \neg \neg \Diamond A\)
\(= \Diamond A\)

\(\neg \Diamond \neg A\)
\(= (\text{Time} \times (A \rightarrow \bot)) \rightarrow \bot\)
\(= \text{Time} \rightarrow ((A \rightarrow \bot) \rightarrow \bot)\)
\(= \Box \neg \neg A\)
\(= \Box A\)