# Declarative Game Programming

## Distilled Tutorial

Henrik Nilsson
University of Nottingham
nhn@cs.nott.ac.uk

Ivan Perez
University of Nottingham
psxip1@nottingham.ac.uk

## Abstract

Video games are usually not programmed very declaratively. There are a number of reasons for this, from low-level efficiency concerns, via the nature of commonly employed programming languages, libraries, and frameworks, to the conceptual nature of such games, with state and effects being omnipresent. However, by structuring games in terms of time-varying values and transformations on such values, it is possible to design and implement video games in a more declarative way. This tutorial shows how this can be achieved through Functional Reactive Programming (FRP) by implementing the high-level parts of a 2D game akin to the classical game Breakout. The tutorial uses the Haskell-embedded FRP implementation Yampa and bindings to SDL (Simple DirectMedia Layer) to obtain game play and visual standards typical of the 2D genre; for example, as seen in many currently popular games for smartphones and tablets.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (functional) programming; I.6.2 [*Simulation And Modeling*]: Simulation Languages

***Keywords*** Video Game Programming, Functional Reactive Programming, Hybrid Modelling

## 1. Introduction

Programming of video games is not what first springs to mind when successful applications of declarative programming are considered. This is not too surprising. For starters, performance requirements subject to resource constraints have often necessitated a low-level approach to implementation. This is turn has influenced the choice of implementation languages and the design of supporting libraries and frameworks. One might also argue that the nature of video games, being much about state and effects, simply does not lend itself to a declarative approach.

However, by changing the perspective from focusing on the state of a system at a particular point in time to how it evolves *over time*, these conceptual objections are mitigated. In particular, implicit state is replaced by explicit operations over the system history, such as accumulation or integration.



*PPDP'14*, September 08 -10 2014, Canterbury, United Kingdom.

http://dx.doi.org/10.1145/2643135.2643160

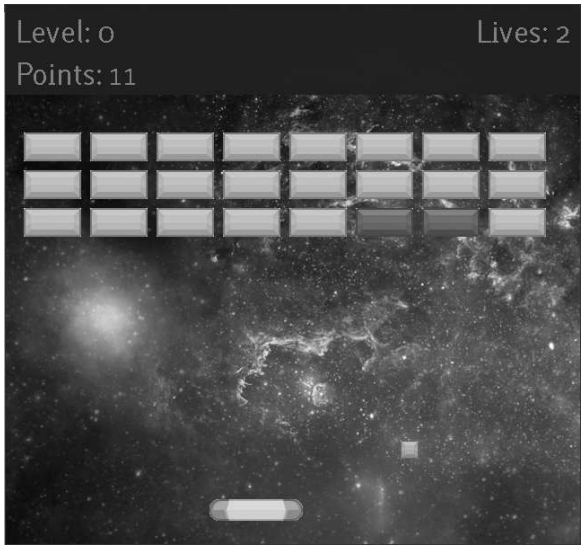



**Figure 1.** A version of the game Breakout

This tutorial demonstrates how to program a video game declaratively in this way using Functional Reactive Programming (FRP) [4], a framework for reactive programming in a functional setting originally developed for graphical animation, but that since has been applied to a diverse set of domains such as graphical user interfaces, robotics, and computer vision. As video games often encompass both continuous-time and discrete-time aspects, FRP's support for hybrid systems is of particular interest.

## 2. The Tutorial

The objective of the tutorial is to implement the high-level aspects (game logic) of a 2D game in the same vein as the classical game Breakout: see figure 1. We use the Haskell-embedded FRP implementation Yampa [6], along with bindings to SDL (Simple DirectMedia Layer) to obtain game play and visual standards typical of the 2D genre; for example, as seen in many currently popular games for smartphones and tablets. However, by choosing a graphical backend such as OpenGL, 3D games can be programmed in much the same way [2].

Yampa is a powerful language for programming reactive systems, combining the conceptual simplicity of synchronous dataflow approach [1] with the flexibility and abstraction power of higher-order functional programming. A prominent feature of Yampa is its very flexible support for programming hybrid systems, including systems with evolving structure, such as games where game entities appear and disappear during the course of play [3, 6].

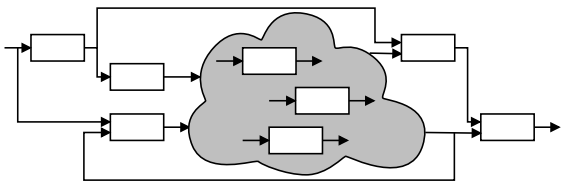
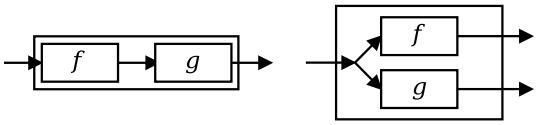


**Figure 2.** A Yampa system is described in terms of interconnected signal functions, possibly with variable structure.




(a) $f \ggg g$ (b) $f \mathbin{\&\&\&} g$

**Figure 3.** Two arrow combinators

The central idea of Yampa is to describe systems in terms of *Signal Functions*: pure functions on time-varying values or *signals*. Only signal functions are first-class entities in Yampa: signals exist only indirectly, through signal functions. A Yampa system consists of a number of interconnected signal functions, operating on the system input and producing the system output: see figure 2, which also illustrates that the system structure can be variable. The signal functions operate in parallel, sensing a common *rate* of time flow. This is why Yampa is a *synchronous* language.

The type of a signal function mapping a signal of type $\alpha$ onto a signal of type $\beta$ is written $\texttt{SF}\,\alpha\,\beta$. Intuitively:

$$\texttt{SF}\,\alpha\,\beta \approx \texttt{Signal}\,\alpha \to \texttt{Signal}\,\beta$$

where

$$\texttt{Signal}\,\alpha \approx \texttt{Time} \to \alpha$$

for some suitable type `Time` representing continuous time. If more than one input or output signal are needed, tuples are used for $\alpha$ or $\beta$ since a signal of tuples is isomorphic to a tuple of signals. If the output of a signal function at time $t$ is determined solely by the input at time $t$, the signal function is said to be *stateless*, otherwise it is said to be *stateful*. A simple example of a stateful signal function is `integral`:

$$integral :: SF\ Double\ Double$$

defined by

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

where $x(t)$ is the input signal and $y(t)$ is the output signal.

Signal functions are *arrows* [5], and Yampa systems are actually constructed by combining signal function using arrow combinators, such as the ones shown in figure 3. However, describing large systems purely in a point-free style is cumbersome. A special arrow notation [7], similar to the monadic `do`-notation, that effectively allows signals to be named and the interconnection structure described in terms of named signals is therefore commonly used.

As a concrete example of the arrow notation and how Yampa allows a declarative formulation of basic game physics, consider the following code from the Breakout game describing the behaviour of the ball when moving freely. It states that the position of the ball is obtained by integrating its velocity and adding this to the initial position:

$$p \leftarrow (p0\ \hat{+}\ )\ \hat{\ll}\ integral \prec v$$

(The operator $\hat{+}$ denotes vector addition, and $\hat{\ll}$ is composition of a lifted pure function and a signal function.) This is not far removed from the mathematical description:

$$\mathbf{p} = \mathbf{p}_0 + \int_0^t \mathbf{v}\,\mathrm{d}\tau$$

In FRP, the domain of a signal can conceptually be either continuous or discrete. In the former case, the signal is defined at every point in time. In the latter case, the signal is a partial function, only defined at discrete points in time. Such a point of definition is called an *event*. In Yampa, this distinction has been deliberately blurred to make it easier to mix and match continuous-time and discrete-time signals. The notion of discrete-time signals is captured by lifting the *range* of continuous-time signals using an option type called `Event`, similar to Haskell's `Maybe` type.

There is a rich set of operations for working with events and mediating between continuous-time and discrete-time signals. One example is $edge$ that generates an event when the Boolean continuous-time input signal changes from false to true. The following code snippet illustrates how to define a discrete-time signal $over$ with an occurrence each time a level has been completed:

$$over \leftarrow edge \prec \neg\,(any\ isBlock\ (map\ objectKind\ (showObjects\ s)))$$

Note again how this is a declarative definition *over* time: a level is completed whenever the last object of type "block" disappears.

In the course of the tutorial, the complete core of the Breakout game is developed step by step through succinct declarative definitions like those illustrated above. The needed features of Yampa are introduced and explained along the way. In particular, the tutorial covers *switching* for temporal composition of behaviours (like making a ball bounce) and evolving system structure (like making bricks disappear once hit), and *feedback* to allow current results to influence future behaviour and separation of mutually interdependent subsystems.

## References


[1] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

[2] M. H. Cheong. Functional programming and 3D games. BEng thesis, University of New South Wales, Sydney, Australia, Nov. 2005.

[3] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, Aug. 2003. ACM Press.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP'97: International Conference on Functional Programming*, pages 163–173, June 1997.

[5] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[6] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct. 2002. ACM Press.

[7] R. Paterson. A new notation for arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, Firenze, Italy, Sept. 2001.