Frabjous Language Reference

April 8th, 2014

Ivan Vendrov
April 8th, 2014

Contents

1 Overview

Frabjous is a domain-specific language for agent-based modeling that aims to make it easier to develop, analyze, and share agent-based models by using state-of-the-art techniques from functional programming.

Frabjous consists of a set of macros or syntactic extensions to the programming language Haskell. Haskell is a popular functional programming language closely related to the *lambda calculus*, the pre-eminent mathematical model of computation. We describe the relationship between Frabjous and Haskell in more detail in sections 4 and ???.

This document provides

1. the necessary background information on *functional reactive programming*, the paradigm on which Frabjous is based (section 2)

2. a complete syntactic and semantic description of the Frabjous language (all other sections)

2 Background

2.1 Functional Reactive Programming

Functional reactive programming (FRP) is a paradigm for describing dynamic systems in a purely functional way by introducing time-varying values or *signals* as first-class language entities. FRP has been applied to domains such as robotics [**?**], games[1], and websites [**?**], yielding remarkably concise specifications of complex systems.

Frabjous is based on a variant of FRP called Arrowized Functional Reactive Programming (AFRP), which prohibits the direct manipulation of signals, replacing them with *signal functions*, operators that take and produce functions of time.

A familiar signal function is $\frac{d}{dt}$, which takes any function and produces its derivative with respect to time. To describe agent behaviour as a system of differential equations, this signal function is all we need. But by adding only a few more signal functions, we can describe a much richer variety of functions - including piecewise, impulse, and partially defined functions of time - which allows us to freely mix discrete and continuous behaviours in the same formalism.

2.2 Some Basic Operators

The simplest AFRP operator is `constant`, which creates a signal function that ignores its input signal and produces the same value at all times. For example,

the signal function `constant 3` produces `3` forever.

A slightly more interesting operator is `function`, which takes a regular function and applies it to each value of the input signal to produce the output signal. Using the composition operator '`.`', which connects the output signal of its right argument to the input signal of its left argument, similar to function composition, we can write

```
function (+1) . constant 3
```

to get a signal function that produces `4` forever. Signal functions created using `function` are *stateless*, since their output at any time t depends only on their input at time t, and not on the input history.

A familiar *stateful* signal function is `integrate`, the Riemann integration operator. For example,

```
integrate 0 . constant 3
```

ignores its input signal and outputs the signal $s(t) = 3t$. `integrate 0` can be though of as having an internal state which keeps track of the value of the integral up to the current time.

We can also directly specify a signal as a function of time using the operator `timeFunction`, so the signal function `timeFunction (3*)` has the same meaning as the preceding example.

2.3 Continuous Time Semantics

One of the key insights behind FRP is that specifying systems in continuous time vastly simplifies language semantics by providing a clean separation between the core system logic and execution concerns like the time-stepping and discretization mechanism. For agent-based modeling, continuous-time semantics have the added advantage that the time step becomes just another parameter, so models can be trivially analyzed for sensitivity to discretization effects.

To take advantage of this insight, Frabjous models are specified using operators with continuous-time semantics. At execution time, these semantics are approximated with a given time step parameter.

For example, the signal function `integrate i . f` conceptually outputs the signal

$$s(t) = i + \int_0^t f(u)du$$

but it is computationally infeasible, in general, to compute this. A reasonable implementation of `integrate`, using Euler integration, would actually output the signal

$$s'(t) = i + \sum_{k=0}^{n} f(k\Delta t)\Delta t$$

where Δt is the desired time step and $t \approx n\Delta t$. For a more accurate approximation, we could use Runge-Kutta or any other numerical integration method, but the only constraint on the implementation of `integrate` is that it matches the continuous-time semantics *in the limit as* Δt *approaches* 0.

3 The Frabjous Modelling Language

3.1 Overview

At the very highest level, a Frabjous model consists of a set of populations evolving in time. Each population is a dynamic collection of agents; agents can be added to and removed from populations by processes such as birth, death, and immigration. Furthermore, any pair of populations can be linked together by a network, which represents relationships between agents such as "parent of" or "neighbour of".

An agent consists of a set of attributes such as income, age, or level of education. Attributes are signal functions which together comprise the evolving agent state, and are the main mechanism through which dynamic behaviour is specified.

3.2 History

The problem of maintaining dynamic sets of interacting entities using FRP was recognized and addressed in *The Yampa Arcade* by Courtney et al. Their approach consisted of

1. encapsulating the inputs and outputs of each agent in records.
2. handling dynamic sets of signal functions by means of *parallel switches*, very general operators which allowed a group of a signal functions to be treated as a signal function of groups.

Since Frabjous is intended to be usable by non-programmers, we specialize their approach to the agent-based modelling domain by providing special syntax for specifying agents, populations, and networks.

3.3 Agents

3.3.1 Agent Declarations

An agent declaration consists of an agent name and a list of attribute bindings with local type annotations:

```
agent AgentName { attr1 :: Type1 = exp1 ; ... ; attrN :: TypeN = expN }
```

Here `expJ` is a Haskell expression of type `SF input TypeJ`, where `input` represents the environment the agent depends on. In each expression, a signal function `attr :: SF input Type` is in scope for each attribute `attr :: Type` (including network attributes—see 3.5).

The provided signal functions are the only way of accessing the environment, ensuring that any agent's attribute can only depend on

1. Attributes of the same agent
2. Attributes of adjacent agents in a network (see 3.5)

This restriction rules out many grossly inefficient dependencies, and makes the resulting models much easier to parallelize.

The agent declaration also defines, at global scope, a new record type

```
data AgentName = AgentName { attr1 :: Type1, ... , attrN :: TypeN }
```

Inside the agent declaration, the names of these accessors are overridden by signal functions, so `getAttr1`, ... , `getAttrN` can be used instead. Also, the initial value of each agent attribute (see 5.1) can be accessed using `initAttr1`, ... , `initAttrN`.

3.3.2 An example

The following code declares a new agent type `Person` with two attributes. `age` keeps track of the person's age starting from an initial value, while `income` depends directly on `age`.

```
agent Person {
        age :: Real = integrate initAge . const 1 ;
        income :: Int = function (ageToIncome) . age
                where ageToIncome a = if (a < 18) then 0 else 20000
}
```

We can now define auxiliary functions on `Person` using pure Haskell:

```
hasIncome :: Person → Bool
hasIncome p = income p > 0
```

3.4 Populations

3.5 Networks

4 Frabjous and Haskell

5 The Model as a Whole

5.1 Initial State

dsafd

A The Frabjous Standard Library

B Some Example Models

C Background

C.1 Functional Reactive Programming

An apparent weakness of Haskell is the difficulty of representing systems that vary with time, since there is no mechanism for changing state. As pioneered by Elliott and Hudak, functional reactive programming (FRP) is a paradigm that augments functional programming with "behaviours" - values that change over time - as well as a set of primitive operations on these values [2]. Arrowized functional reactive programming (AFRP) is a version of FRP that shifts the focus onto functions between time-varying values, called signal functions[3]. AFRP allows complex, time-varying, locally stateful systems to be written in a declarative fashion, as demonstrated by Courtney et al [1].

The simplest AFRP operator is `constant`, which defines a constant function with a given value. So the output of the signal function `constant 1` is `1` at all times, and for all inputs. Integration can also be viewed as a signal function, since it operates on a function of time and produces a function of time. So

```
integral . constant 1
```

is a signal function that ignores its input and outputs the current time (`'.'` is the Haskell function composition operator).

Following the Netwire version of AFRP[6], we allow signal functions to sometimes not produce values. For example, `rate` is a signal function that takes a time-varying number and produces a value at a rate specified by that number, so `rate . constant 2` produces twice (on average) in a given time unit. This allows us to model, for example, the Poisson process:

```
poisson lambda = count (rate . constant lambda)
```

where `count` is an operator that counts the number of instants that its argument produces a value, and `lambda` is the rate parameter of the Poisson process.

Another common signal operator is `after`, which produces after a given length of time. Two signal functions can be combined in parallel using the <|> operator, which acts like its left hand side when it produces, and like its right hand side otherwise, so

```
constant 1 . after 3 <|> constant 0
```

is a signal function that produces `0` for the first three time units, then `1` forever.

FRP can in fact be viewed as a generalization of differential equations. An FRP program is essentially a set of equations between time-varying values (in other words, functions of time). But where differential equations are limited to the standard mathematical operations such as addition, multiplication, and the differentiation operator, FRP adds a number of operators (such as `rate`

and `count`) that act on and produce a much richer variety of functions. The differentiation operator is only defined on smooth functions in a vector space, but FRP operators can produce piecewise and step functions of arbitrary sets, which allows us to model discrete processes like the Poisson process above.

We use FRP as the basis for our ABM language because its declarative nature provides the transparency, clarity, and concision usually associated with declarative modelling [4, 5], and because FRP has an explicit, formal semantics [2] that simplifies mathematical reasoning.

D The Frabjous Modelling Language

At the very highest level, a Frabjous model consists of a set of populations evolving in time. Each population is a dynamic collection of agents; agents can be added to and removed from populations by processes such as birth, death, and immigration. Furthermore, any pair of populations can be linked together by a network, which represents relationships between agents.

An agent consists of a set of attributes such as income, age, or level of education. Attributes are time-varying values which together comprise the evolving agent state, and are the main mechanism through which dynamic behaviour is specified.

The direct use of time-varying values is the only truly novel language feature of Frabjous, as the handling of populations and networks is mostly an issue of library design. Since time-varying values are the fundamental building blocks of Frabjous models, we proceed to discuss them in depth before presenting the full language.

D.1 Time-Varying Values

Time-varying values are not specified in Frabjous directly, but implicitly by means of signal functions (introduced in). In particular, the dynamics of an agent attribute are specified by a signal function from the entire agent to the attribute; thus the value of an attribute at a particular time depends only on the values of other attributes up to that time.

Normally one defines signal functions by combining simpler functions with one of the provided operators. For example, we might define the attribute age of an agent as follows:

```
age = integral . constant 1
```

so the age of an agent is the amount of time elapsed since the agent was added to the model.

But how would we declare an attribute `isAdult`, which should be `False` during the first 18 years of the agent's life, and `True` from the 18th birthday on? This is a special case of a functional dependency, when the value of an attribute is a function of other attributes. To express functional dependencies in Frabjous, write it as a function, appending `(t)` to the declared signal function and the

signal functions it depends on. This serves to make the dependence on time explicit:

```
isAdult(t) = age(t) ≥ 18
```

As a more elaborate example of this notation, the declaration

```
income(t) = if (isAdult(t))
                  then 1000 ∗ age(t) ∗ (uniform (0.5, 1.5))(t)
                  else 0
```

defines the income of adults at any given time to be 1000 times their age multiplied by a random number drawn from a uniform distribution between 0.5 and 1.5, and the income of non-adults to be 0.

E Proposed Changes

1. Attributes that lack a dynamic binding are considered CONSTANTS, not wires (useful both for syntax and optimizations to have agent-level constants)

References

[1] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM.

[2] Conal Elliott and Paul Hudak. Functional reactive animation. *International Conference on Functional Programming*, pages 263–273, 1997.

[3] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive progamming, continued. *2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[4] Nathaniel Osgood. Systems dynamics and agent-based approaches: Clarifying the terminology and tradeoffs. *First International Congress of Business Dynamics*, 2006.

[5] Nathaniel Osgood. Using traditional and agent based toolsets for system dynamics: Present tradeoffs and future evolution. *2007 International Conference on System Dynamics*, 2007.

[6] Ertugrul Soeylemez. Netwire, 2012. [Online; accessed 29-August-2013].