Spec equals Code
Functional reactive ABS

Jonathan THALER

February 28, 2017

Abstract

The pure functional paradigm has so far got not very much attention in ABS where the dominant programming paradigm is object-orientation, realized by Java. In preliminary research of ours, we came to the insight that pure functional programming using Haskell without any additional libraries is very well suited to implement ABS but that in order to build more complex, reald-world models it would be necessary to leverage the basic concepts with functional reactive programming (FRP). In this paper we show how ABS can be fused with FRP to design an embedded domain specific language (EDSL) which allows to write specifications which (nearly) match pure functional code - the specification becomes Haskell code. For a proof of concept, we chose to implement the famous and well known *Sugarscape* model from the social sciences and the *ACE Trading World* from agent-based computational economics (ACE).

1 Introduction

TODO: talk about preliminary research and its drawbacks: need something like FRP / Yampa to leverage
TODO: need a formal specification

2 Background

2.1 Agent Based Simulation

We understand ABS as a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge. Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. So the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [13]. It is important to note that we focus our understanding of ABS on a very specific kind of agents where the focus is on communicating entities with individual, localized behaviour from out of which the global behaviour of the system emerges. We informally assume the following about our agents:

- They are uniquely addressable entities with some internal state.

- They can initiate actions on their own e.g. change their internal state, send messages, create new agents, kill themselves.

- They can react to messages they receive with actions as above.

- They can interact with an environment they are situated in.

We will give a formal model of agents in a subsequent section (TODO: reference)

1

2.2 Pure functional paradigm

TODO: short description what pure functional paradigm is Object-oriented (OO) programming is the current state-of-the-art method used in implementing ABM/S due to the natural way of mapping concepts and models of ABM/S to an OO-language. Although this dominance in the field we claim that OO has also its serious drawbacks:

- Mutable State is distributed over multiple objects which is often very difficult to understand, track and control.

- Inheritance is a dangerous thing if not used properly and with care because it introduces very strong dependencies which cannot be changed during runtime any-more.

- Objects don't compose very well due to their internal (mutable) state (note that we are aware that there is the concept of immutable objects which are becoming more and more popular but that does not solve the fundamental problem.

- It is (nearly) impossible to reason about programs.

We claim that these drawbacks are non-existent in pure functional programming like Haskell due to the nature of the functional approach. To give an introduction into functional programming is out of scope of this paper but we refer to the classical paper of [4] which is a great paper explaining to non-functional programmers what the significance of functional programming is and helping functional programmers putting functional languages to maximum use by showing the real power and advantages of functional languages. The main conclusion of this classical paper is that *modularity*, which is the key to successful programming, can be achieved best using higher-order functions and lazy evaluation provided in functional languages like Haskell. [4] argues that the ability to divide problems into sub-problems depends on the ability to glue the sub-problems together which depends strongly on the programming-language and [4] argues that in this ability functional languages are superior to structured programming.

2.3 Naive implementation

TODO: shortcomings Of course the basic pure functional primitives alone do not make a well structured functional program by themselves as the usage of classes, interfaces, objects and inheritance alone does not make a well structured object-oriented program. What is needed are *patterns* how to use the primitives available in pure functional programs to arrive at well structure programs. In object-orientation much work has been done in the 90s by the highly influential book [3] whereas in functional programming the major inventions were also done in the 90s by the invention of Monads through [?], [?] and [?] and beginning of the 2000s by the invention of Arrows through [?].

2.3.1 Higher Order Functions & Monads

map & fmap, foldl, applicatives [5] gives a great overview and motivation for using fmap, applicatives and Monads. TODO: explain Monads

2.3.2 Arrows

[?] is a great tutorial about *Arrows* which are very well suited for structuring functional programs with effects.

> Just like monads, arrow types are useful for the additional operations they support, over and above those that every arrow provides.

The main difference between Monads and Arrows are that where monadic computations are parameterized only over their output-type, Arrows computations are parameterised both over their input- and output-type thus making Arrows more general.

> In real applications an arrow often represents some kind of a process, with an input channel of type a, and an output channel of type b.

In the work [?] an example for the usage for Arrows is given in the field of circuit simulation. They use previously introduced streams to advance the simulation in discrete steps to calculate values of circuits thus the implementation is a form of *discrete event*

simulation - which is in the direction we are heading already with ABM/S. Also the paper mentions Yampa which is introduced in the section (TODO: reference) on functional reactive programming.

2.4   FRP

FRP is a paradigm for programming hybrid systems which combine continuous and discrete components. Time is explicitly modelled: there is a continuous and synchronous time flow.

there have been many attempts to implement FRP in frameworks which each has its own pro and contra. all started with fran, a domain specific language for graphics and animation and at yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa which is the reason why it was chosen as the FRP framework. Also, compared to other frameworks it does not distinguish between discrete and synchronous time but leaves that to the user of the framework how the time flow should be sampled (e.g. if the sampling is discrete or continuous - of course sampling always happens at discrete times but when we speak about discrete sampling we mean that time advances in natural numbers: 1,2,3,4,... and when speaking of continuous sampling then time advances in fractions of the natural numbers where the difference between each step is a real number in the range of [0..1])

time- and space-leak: when a time-dependent computation falls behind the current time. TODO: give reason why and how this is solved through Yampa. Yampa solves this by not allowing signals as first-class values but only allowing signal functions which are signal transformers which can be viewed as a function that maps signals to signals. A signal function is of type SF which is abstract, thus it is not possible to build arbitrary signal functions. Yampa provides primitive signal functions to define more complex ones and utilizes arrows [?] to structure them where Yampa itself is built upon the arrows: SF is an instance of the Arrow class.

Fran, Frob and FAL made a significant distinction between continuous values and discrete signals. Yampas distinction between them is not as great. Yampas signal-functions can return an Event which makes them then to a signal-stream - the event is then similar to the Maybe type of Haskell: if the event does not signal then it is NoEvent but if it Signals it is Event with the given data. Thus the signal function always outputs something and thus care must be taken that the frequency of events should not exceed the sampling rate of the system (sampling the continuous time-flow). TODO: why? what happens if events occur more often than the sampling interval? will they disappear or will the show up every time?

switches allow to change behaviour of signal functions when an event occurs. there are multiple types of switches: immediate or delayed, once-only and recurring - all of them can be combined thus making 4 types. It is important to note that time starts with 0 and does not continue the global time when a switch occurs. TODO: why was this decided?

[?] give a good overview of Yampa and FRP. Quote: "The essential abstraction that our system captures is time flow". Two *semantic* domains for progress of time: continuous and discrete.

The first implementations of FRP (Fran) implemented FRP with synchronized stream processors which was also followed by [?]. Yampa is but using continuations inspired by Fudgets. In the stream processors approach "signals are represented as time-stamped streams, and signal functions are just functions from streams to streams", where "the Stream type can be implemented directly as (lazy) list in Haskell...":

```
type Time = Double
type SP a b = Stream a -> Stream b
newtype SF a b = SF (SP (Time, a) b)
```

Continuations on the other hand allow to freeze program-state e.g. through closures and partial applications in functions which can be continued later. This requires an indirection in the Signal-Functions

3

which is introduced in Yampa in the following manner.

```
type  DTime = Double

data SF a b =
       SF { sfTF  ::  DTime -> a -> (SF a b, b)
```

The implementer of Yampa call a signal function in this implementation a *transition function*. It takes the amount of time which has passed since the previous time step and the durrent input signal (a). It returns a *continuation* of type SF a b determining the behaviour of the signal function on the next step (note that exactly this is the place where how one can introduce stateful functions like integral: one just returns a new function which encloses inputs from the previous time-step) and an *output sample* of the current time-step.

When visualizing a simulation one has in fact two flows of time: the one of the user-interface which always follows real-time flow, and the one of the simulation which could be sped up or slowed down. Thus it is important to note that if I/O of the user-interface (rendering, user-input) occurs within the simulations time-frame then the user-interfaces real-time flow becomes the limiting factor. Yampa provides the function embedSync which allows to embed a signal function within another one which is then run at a given ratio of the outer SF. This allows to give the simulation its own time-flow which is independent of the user-interface.

One may be initially want to reject Yampa as being suitable for ABM/S because one is tempted to believe that due to its focus on continuous, time-changing signals, Yampa is only suitable for physical simulations modelled explicitly using mathematical formulas (integrals, differential equations,...) but that is not the case. Yampa has been used in multiple agent-based applications: [**?**] uses Yampa for implementing a robot-simulation, [**?**] implement the classical Space Invaders game using Yampa, the thesis of [**?**] shows how Yampa can be used for implementing a Game-Engine, [**?**] implemented a 3D first-person shooter game with the style of Quake 3 in Yampa. Note that although all these applications don't focus explicitly on agents and agent-based modelling / simulation all of them inherently deal with kinds of agents which share properties of classical agents: game-entities, robots,... Other fields in which Yampa was successfully used were programming of synthesizers (TODO: cite), Network Routers, Computer Music Development and various other computer-games. This leads to the conclusion that Yampa is mature, stable and suitable to be used in functional ABM/S. Jason Gregory (Game Engine Architecture) defines Computer-Games as "soft real-time interactive agent-based computer simulations".

To conclude: when programming systems in Haskell and Yampa one describes the system in terms of signal functions in a declarative manner (functional programming) using the EDSL of Yampa. During execution the top level signal functions will then be evaluated and return new signal functions (transition functions) which act as continuations: "every signal function in the dataflow graph returns a new continuation at every time step".
"A major design goal for FRP is to free the programmer from 'presentation' details by providing the ability to think in terms of 'modeling'. It is common that an FRP program is concise enough to also serve as a specification for the problem it solves" [**?**]. This quotation describes exactly one of the strengths using FRP in ACE

TODO: short description what FRP is, cite the necessary papers TODO: hypothesis that FRP is a remedy for the shortcomings

2.5   The models

2.5.1   Sugarscape

TODO [2]

2.6   ACE Trading World

TODO [11]

2.7 Related Research

The amount of research on using the pure functional paradigm using Haskell in the field of ABS has been moderate so far. Most of the papers look into how agents can be specified using the belief-desire-intention paradigm [1], [10], [6]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [9]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. [6] discuss using functional programming for discrete event simulation (DES) and mention the paradigm of FRP to be very suitable to DES.

[10] present an EDSL for Haskell allowing to specify Agents using the BDI model. TODO: We don't go there, thats not our intention.

[8] and [12] present a domain-specific language for developing functional reactive agent-based simulations. This language called FRABJOUS is very human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell/Yampa but is compiled to Haskell/Yampa code which they claim is also readable. This is the direction we want to head but we don't want this intermediate step but look for how a most simple domain-specific language embedded in Haskell would look like. We also don't touch upon FRP and Yampa yet but leave this to further research for another paper of ours.

[7] TODO

TODO: cite julie greensmith paper on haskell

3 The formal Agent-Model

TODO: define an agent

4 Functional Reactive ABS

5 Conclusion and future research

References

[1] DE JONG, T. Suitability of Haskell for Multi-Agent Systems. Tech. rep., University of Twente, 2014.

[2] EPSTEIN, J. M., AND AXTELL, R. *Growing Artificial Societies: Social Science from the Bottom Up.* The Brookings Institution, Washington, DC, USA, 1996.

[3] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., AND BOOCH, G. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition ed. Addison-Wesley Professional, Oct. 1994.

[4] HUGHES, J. Why Functional Programming Matters. *Comput. J. 32*, 2 (Apr. 1989), 98–107.

[5] HUTTON, G. *Programming in Haskell.* Cambridge University Press, Cambridge, UK ; New York, Jan. 2007.

[6] JANKOVIC, P., AND SUCH, O. Functional Programming and Discrete Simulation. Tech. rep., 2007.

[7] KLÜGL, F., AND DAVIDSSON, P. AMASON: Abstract Meta-model for Agent-Based SimulatiON. In *Multiagent System Technologies: 11th German Conference, MATES 2013, Koblenz, Germany, September 16-20, 2013. Proceedings*, M. Klusch, M. Thimm, and M. Paprzycki, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 101–114. DOI: 10.1007/978-3-642-40776-5_11.

[8] SCHNEIDER, O., DUTCHYN, C., AND OSGOOD, N. Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation. In *Proceedings of the 2Nd ACM*

SIGHIT International Health Informatics Symposium (New York, NY, USA, 2012), IHI '12, ACM, pp. 785–790.

[9] SOROKIN, D. *Aivika 3: Creating a Simulation Library based on Functional Programming*. 2015.

[10] SULZMANN, M., AND LAM, E. Specifying and Controlling Agents in Haskell. Tech. rep., 2007.

[11] TESFATSION, L. Agent-Based Computational Economics: A Constructive Approach to Economic Theory. Handbook of Computational Economics, Elsevier, 2006.

[12] VENDROV, I., DUTCHYN, C., AND OSGOOD, N. D. Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, W. G. Kennedy, N. Agarwal, and S. J. Yang, Eds., no. 8393 in Lecture Notes in Computer Science. Springer International Publishing, Apr. 2014, pp. 385–392.

[13] WOOLDRIDGE, M. *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.