Implementing pure functional ACE in Haskell

Jonathan Thaler

November 15, 2016

Abstract

So far the literature on agent-based modelling & simulation (ABM/S) hasn't focused much on models for functional agents and is lacking a proper treatment of implementing agents in pure-functional languages like Haskell. This paper looks into how agents can be specified functionally and then be implemented properly in the pure functional language Haskell. The functional agent-model is inspired by wooldridge 2.6. The programming paradigm used to implement the agents in Haskell is functional reactive programming (FRP) where the Yampa framework will be used. The paper will show that specifying and implementing agents in a pure functional language like Haskell has many advantages over classical object-oriented, concurrent ones but needs also more careful considerations to work properly.

1 Introduction

In the end it all boils down to 1. the agent-model and 2. how the agent-model is implemented in the according language. Of course both points influence each other: functional languages will come up with a different agent-model (e.g. hybrid like yampa) than object-oriented ones (e.g. actors).

1.1 Functional programming

Functional languages can best be characterized by their way computation works: instead of *how* something is computed, *what* is computed is described. Thus functional programming follows a declarative instead of an imperative style of programming. The key points are:

- No assignment statements - variables values can never change once given a value.
- Function calls have no side-effect and will only compute the results - this makes order of execution irrelevant, as due to the lack of side-effects the logical point in *time* when the function is calculated within the program-execution does not matter.

- higher-order functions
- lazy evaluation
- Looping is achieved using recursion, mostly through the use of the general fold or the more specific map.
- Pattern-matching

This alone does not really explain the *real* advantages of functional programming and one must look for better motivations using functional programming languages. One motivation is given in Hughes (1989) 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 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. Hughes (1989) 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 Hughes (1989) argues that in this ability functional languages are superior to structured programming.
TODO: comparison of functional and object-oriented programming. My points are:

- The way state can be changed and treated - distributed over multiple objects - is often very difficult to understand.
- Inheritance is a dangerous thing if not used with care because inheritance introduces very strong dependencies which cannot be changed during runtime anymore.
- Objects don't compose very well: `http://zeroturnaround.com/rebellabs/why-the-debate-on-object-oriented-vs-functional-programming-is-all-about-composition/`
- (Nearly) impossible to reason about programs

In conclusion the upsides of functional programming as opposed to OO are:

- Much more explicit flow of data & control
- Much better compose-able
- Much better parallelism

2 Related Research

Bezirgiannis (2013) constructs two frameworks: an agent-modelling framework and a DES framework, both written in Haskell. They put special emphasis on

parallel and concurrency in their work. The author develops two programs: HLogo which is a clone of the NetLogo agent-modelling framework and HDES, a framework for discrete event simulation - where in both implementations is the very strong emphasis on parallelism. Here only the HLogo implementation is of interest as it is directly related to agent-based simulation. In this implementation the author claims to have implemented an EDSL which tries to be close to the language used for modelling in NetLogo (Logo) "which lifts certain restrictions of the original NetLogo implementation". Also the aim was to be "faster in most circumstances than NetLogo" and "utilizes many processor cores to speedup the execution of Agent Based Models". The author implements a primitive model of concurrent agents which implements a non-blocking concurrent execution of agents which report their results back to the calling agent in a non-blocking manner. The author mentions that a big issue of the implementation is that repeated runs with same inputs could lead to different results due to random event-orderings happening because of synchronization. The problem is that the author does not give a remedy for that and just accepts it as a fact. Of course it would be very difficult, if not impossible, to introduce determinism in an inherently concurrent execution model of agents which may be the reason the author does not even try. Unfortunately the example implementation the author uses for benchmarking is a very simplistic model: the basic pattern is that agent A sends to agent B and thats it - no complex interactions. Of course this lends itself very good to parallel/concurrent execution and does not need a sophisticated communication protocol. The work lacks a proper treatment of the agent-model presented with its advantages and disadvantages and is too sketchy although the author admits that is is just a proof of concept.

Tim Sweeney, CTO of Epic Games gave an invited talk about how "future programming languages could help us write better code" by "supplying stronger typing, reduce run-time failures; and the need for pervasive concurrency support, both implicit and explicit, to effectively exploit the several forms of parallelism present in games and graphics." Sweeney (2006). Although the fields of games and agent-based simulations seem to be very different in the end, they have also very important similarities: both are simulations which perform numerical computations and update objects - in games they are called "game-objects" and in abm they are called agents but they are in fact the same thing - in a loop either concurrently or sequential. His key-points were:

- Dependent types as the remedy of most of the run-time failures.
- Parallelism for numerical computation: these are pure functional algorithms, operate locally on mutable state. Haskell ST, STRef solution enables encapsulating local heaps and mutability within referentially transparent code.
- Updating game-objects (agents) concurrently using STM: update all objects concurrently in arbitrary order, with each update wrapped in atomic block - depends on collisions if performance goes up.

3 ACE

“[...] computational modeling of economic processes (including whole economies) as open-ended dynamic systems of interacting agents.” Leigh Tesfatsion
The book Kirman (2010) is a critique of classic economics with the triple of rational agents, ”average” inidividuum, equilibrium theory. Although it does not mention ACE it can be seen as an important introduction to the approach of ACE as it introduces many important concepts and views dominant in ACE. Also ACE can be seen as an approach of tackling the problems introduced in this book:

page 6: ”the view of economy is much closer to that of social insects than to the traditional view of how economies function.”
page 7: ”... main argument that it is the interaction between individuals that is at the heart of the explanation of many macroeconomic phenomena...”
page 15: ”problem of equilibrium is information”
page 21: ”the theme of this book will be that the very fact individuals interact with each other causes aggregate behaviour to be different from that of individuals”

4 Structuring Functional Programs

4.1 Higher Order Functions

4.2 Monads

4.3 Functional reactive programming (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 Hughes (2005) 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 signalfunctions 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?

Nilsson et al. (2002) 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 Wan and Hudak (2000). 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 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: Hudak et al. (2003) uses Yampa for implementing a robot-simulation, Courtney et al. (2003) implement the classical Space Invaders game using Yampa, the thesis of Meisinger (2010) shows how Yampa can be used for implementing a Game-Engine, Cheong (2005) 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" Wan and Hudak (2000). This quotation describes exactly one of the strengths using FRP in ACE

5 Reasoning

Give example by showing reasoning in ACE: convergence, correctness,...

5.1 Time and Semantics

Wan and Hudak (2000) discuss the semantic framework of FRP. Very difficult to understand and full of corollaries and theorems and proofs, have to study in depth at another time.

6 Determinism

no concurrent execution: deterministic

deterministic: can use random-numbers but to be reproducible/deterministic one has to specify the same seed or even provide an own RNG-implementation (which is easily possible using the RNG in haskell)

it is of most importance in simulations to be reproducible under given conditions: two runs with the same input (e.g. time, agent-count, parameters, RNG seeds) should result in the exact same results otherwise the simulation-software is of very little benefit.

7 Functional Model

Start from wooldridge 2.6 (look at the original papers which inspired the 2.6 chapter) and weiss book and the original papers those chapter is based upon. Look into denotational semantics of actor model. Look also in functional models of czesar ionescu. why: this is the major contribution of my thesis and is new knowledge. Must find intuitive, original and creative approach.
From functional agent models (e.g. Wooldridge) to implementation of ACE in Haskell (this should go into research-proposal introduction)

8 Example: TODO some market model

8.1 EDSL

In this paper I present an EDSL which allows to formulate models for agent-based market-simulations which can be directly run in a Haskell framework implemented for this. Thus the distinction between model specification and programming vanishes - the model specification becomes the actual code. The major novelty of this EDSL is that it allows to model the system in a qualitative way: relations among formulas are expressed which can be understood as a kind of non-causal modelling. TODO: better understand what qualitative modelling/simulation in ACE is.

9 Implementation

idea: can we implement a message between two agents through events? thus two states: waiting for messages, processing messages. BUT: then sending a message *will take some time*
NOTE: it is important to make a difference about whether the simulation will dynamically *add* or *remove* agents during execution. If this is not the case, a simple par-switch is possible to run ALL agent SF in parallel. If dynamically changes to the agent-population should be part of the simulation, then the dpSwitch or dpSwitchB should be used. Also it should be possible to start/stop agents: if they are inactive then they should have no running SF because would use up resources. Inactive means: doing nothing, also not awaiting something/"doing nothing in the sense that DOING something which is nothing - the best criteria to decide if an agent can be set inactive is when the event which decides if the agents SF should be started comes from outside e.g. if the agent is just statically "living" but not changing and then another agent will "ignite" the "living" agent then this is a clear criterion for being static without a running SF.

NOTE: the route-function will be used to distribute "messages" to the agents when they are communicating with each other

NOTE: Meisinger (2010) argues that in Game-Engines (it is paraphrased in english, as the thesis was written in german): "communication among Game-Objects is always computer-game specific and must be implemented always new but the functionality of Game-objects can be built by combining independent functions and signal-functions which are fully reuse-able". Game-Objects can be understood as agents thus maybe this also holds true for agent-based simulation. Meisinger (2010) thus distinguishes between normal functions e.g. mathematical functions, signal functions which depend on output since its creation in localtime and game-object functions which output depends on inputs AND time (which is but another input).

TODO: need a mechanism to address agents: if agent A wants to send a message to agent B and agent B wants to react by answering with a message to agent A then they must have a mechanism to address each other

TODO: design general input/output data-structures

TODO: design general agent SF

TODO: don't loose STM out of sight!
Wormholes in FRP?

10 Performance

10.1 Active vs inactive Agents

signalfunctions add up, multiple chains of events add up, need to remove inactive agents or exclude them somehow from computation chain: use freezing?

10.2 Parallelism and Concurrency

Bezirgiannis (2013) puts special emphasis on concurrency and parallelism in implementing simulation framework in Haskell. TODO: explain deeper

Due to the pure-functional property of a agents SF (which fully describes the agents behaviour over time) all signal-functions of all agents should be able to run in parallel in each iteration as they are pure functional: they don't touch global/shared state. Also the routing and switching functions could be sped up using par. But Question: how is this possible in Yampa?

Wormholes in FRP?

References

Bezirgiannis, N. (2013). Improving performance of simulation software using haskell's concurrency & parallelism. Master's thesis, Utrecht University - Dept. of Information and Computing Sciences.

Cheong, M. H. (2005). Functional programming and 3d games. Master's thesis, University of New South Wales, Sydney, Australia.

Courtney, A., Nilsson, H., and Peterson, J. (2003). The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA. ACM.

Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. (2003). *Arrows, Robots, and Functional Reactive Programming*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg.

Hughes, J. (1989). Why functional programming matters. *Comput. J.*, 32(2):98–107.

Hughes, J. (2005). Programming with arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming*, AFP'04, pages 73–129, Berlin, Heidelberg. Springer-Verlag.

Kirman, A. (2010). *Complex Economics: Individual and Collective Rationality*. Routledge, London ; New York, NY.

Meisinger, G. (2010). Game-engine-architektur mit funktional-reaktiver programmierung in haskell/yampa. Master's thesis, Fachhochschule Oberösterreich - Fakultät für Informatik, Kommunikation und Medien (Campus Hagenberg).

Nilsson, H., Courtney, A., and Peterson, J. (2002). Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA. ACM.

Sweeney, T. (2006). The next mainstream programming language: A game developer's perspective. *SIGPLAN Not.*, 41(1):269–269.

Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 242–252, New York, NY, USA. ACM.