Comparing the Pure Functional and Object Oriented Paradigm for implementing ABS

jonathan.thaler@nottingham.ac.uk

August 10, 2017

Abstract

This study we compares the object oriented and pure functional programming paradigms to implement Agent-Based Simulation. Due to fundamentally different concepts both propagate fundamental different approaches in implementing ABS. In this document we seek to precicesly identify these fundamental differences, compare them and also look into general benefits and drawbacks of each approach.

Contents

1 Introduction **3**
1.1 Programming Paradigms 4
1.2 Haskell 4

2 Challenges **5**
2.1 Agent Representation 6
2.2 Agent Updating 7

Chapter 1

Introduction

TODO: line of argumentation and structure of the report computation to language paradigms to concrete languages to libraries: turing machine and lambda to functional and imperative / operational to haskell and java to frabs and repast. on all levels can we identify the parallels to ABS or do they only show up in the very end? i think we can trace them to the paradigms

[ ] start with the question: all these programming languages are turing complete, why then not implement directly in turing machine or lambda calculus and why bother about different paradigms? the power is there isnt it? [ ] then look into the very foundations of conputation: turing model vs. lambda calculu denotational [ ] then make it clear that we dont program in a turing machine or lambda calculus (actually haskell is much much closer to lambda calculus than e.g. java or even is to a turing machine) because the raw power becomes unmanagable, we loose control. why? because we think problems which are more complex than operations on natural numbers very different and these lowlevel computational languages dont allow us to express this - they are not very expressive: too abstract. [ ] thus we arrive at a first conclusion: TM in theory yes but its not practical because we think about problems different and TM does not allow us directly to express in the way we think, we need to build more mechanisms on top of this concept. so we have introduced the concept of expressivity. how can we express e.g. an if statement or a loop in a TM? [ ] for the lambda calculus it is about the same with the difference that it is much more expressive than a TM [ ] so then the argumentation continues: we build up more and more levels of abstractions where each depends on preceeding ones. the point is that some languages stop at some level of abstraction and others continue. [ ] also there are different types of abstraction depending if we come either from lambda or turing direction [ ] the question is then: which level of abstraction is necessary for ABS? how provide FP and OOP these?

1.1 Programming Paradigms

define what is functional programming define what is imperative programming define what is object-oriented programming

make it clear that just because java has now lambdas does not make it functional. distinguish between functional style and functional programming. when using a specific style one abuses language features to emulate a different paradigm than the one of the host language - so it is also possible to emulate oop in C or Haskell but this does not make them oop languages, one just emulate a programming-style (with potentially disastrous consequences as the code will probably become quite unreadable)

[ ] java lambdas are syntactic sugar for anonymous classes to resemble a more functional style of programming. sideeffects still possible [ ] look into the aggregate functions of java. also support functional style of programming. [ ] same for method references as above: it is impressive how much bulk was added to the language to introduce these concepts which work out of the box in Haskell with higher order functions, currying and lambdas

TODO: investigate lambdas in java

TODO: there is already a low-level haskell/java comparison there: in the code of the update-strategies paper. Also make direct use of this paper as it discusses some of the fundamental challenges implementing ABS in an language- and paradigm-agnostic way

1.2 Haskell

haskells real power: side-effect polymorph. enabled through monadic progranming which becomes possible through type parameters, typeclasses, higher-order functions, lambdas and pattern matching

Chapter 2

Challenges

The challenges one faces when implementing an Agent-Based Simulation (ABS) plain, without support from a library (e.g. Repast) are manifold. In the paper on update-strategies (TODO: cite) we've discussed already in a very general, programming language agnostic way, the fundamental things to consider. Here we will look at the problem in a much more technical way by precisely defining what problems need to be solved and what approaches are from a programming paradigm view-point - where we focus on the pure functional (FP) and imperative object-oriented (OO) paradigms.

Generally one faces the following challenges:

1. Agent Representation - How is an Agent represented in the paradigm?
2. Agent Updating - How is the set of Agents organized and how are all of them updated?
3. Agent-Agent Interactions
4. Environment Representation
5. Environment Updating
6. Agent-Environment Interactions
7. Replications

It is important to note that we are facing a non-trivial software-engineering problem which implies that there are no binary correct wrong approaches - whatever works good enough is OK. This implies that the challenges as discussed below, can be also approached in different ways but we tried to stick as close as possible to the *best practices* of the respective paradigm.

2.1 Agent Representation

2.1.1 OO

In the OO paradigm an Agent will (almost) always be represented as an object which encapsulates the state of the Agent and implements the behaviour of the Agent into private and public methods. Care must be taken to not confuse the concept of an Agent with the one of an object: an Agent is pro-active and always in full control over its state and the messages sent to it. We have discussed pro-activity in the update-strategies paper already: what is needed is a method to update the Agent which transports some time-delta to allow the Agent perceive time, ultimately allowing it to become pro-active. Other options would be to spawn a thread within the object which then makes the object an *active* object but then one needs to deal with synchronization issues in case of Agent-Agent interactions. In OO it is tempting to generate getter and setter for all properties of the Agent state but this would make the Agent vulnerable to changes out of its control - state-changes should always come from the Agent within. Of course when generating text- or visual output then getter are required for the properties which need to be observed. Agent-creation in OO is then in the end an instantiation of an Agent-Class resulting in an Agent-Object. When one strictly avoids setter-methods then the only way of instantiating the Agent into a consistent state is the constructor. This could lead to a very bloated constructor in the case of a complex Agent with many properties. Still we think this is better than having setter-methods as setters are always tempting to be used outside of the construction phase, especially when multiple persons are working on the implementation or when the original implementer is not available any more. If an Agent construction is really complicated with many constructor-parameters one can resort to the Builder-Pattern [1]. Another approach to creation is dependency injection [1] but then the application would need to run in an IoC container e.g. Spring. We haven't tried this approach but we think it would over-complicate things and is an overkill in the domain of ABS.

2.1.2 FP

Although there exist object-oriented approaches to functional programming (e.g. F#, OCaml) we assume that there are no classes and inheritance in FP. By a class we understand a collection of functions (called methods in OO) and data (members or properties in OO) where the functions can access this data without the need to explicitly pass it in through arguments. So we need functions which represent the Agent's behaviour and data which represents the Agents state. The functions need to access this state somehow and be able to change the state. This may seem to be an attempt to emulate OO in FP but this is not the case: functions operating on data are not an OO-exclusive concept - it becomes

[1] See `https://martinfowler.com/articles/injection.html`

OO when the data is implicitly bound in the function [2]. FP in general has no notion of a compound data-type but tuples can be used to emulate such. Because it is quite cumbersome to work on tuples or to emulate compound data-types using tuples, FP languages (e.g. Haskell) have built-in features for compound data-types. So we assume that without loss of generality (because compound data-types are in the end tuples with different names for projection-functions) Agent state in FP is represented using a compound data-type. The relevant function in FP for Agent-Behaviour is the update-function. We have two options: Either the function arguments are the compound agent-state, time-delta and incoming messages and must return the (changed) compound agent-state and outgoing messages. Or we use continuation-style programming in which the compound agent-state is updated internally and the only input are the time-delta and incoming messages and the output are outgoing messages, the observable agent-state which can be represented by a different compound data-type AND a continuation function. In the first approach the full agent-state is available outside and could be changed any time - there is no such thing as data-hiding in this case. In the continuation case the state is bound in a closure which is the newly constructed function which will be returned as continuation. This is only possible in a real functional language which allows the construction of functions through lambdas AND return them as a return value of a function.

2.2 Agent Updating

2.2.1 OO

After creating the Agents one ends up with a collection of Agents, represented either as a List, a Vector or a Map. In OO updating is pretty trivial: one iterates over the collection and calls some update-method of the Agent objects. This implies that if one uses inheritance and has a general Agent-Class, this class needs to provide an update-method which feeds a time-delta. When implementing the parallel-strategy things become complicated in OO though. Changes must only be visible in the next iteration. This can only be achieved by either messaging instead of method-calls or creating new Agent-objects after every iteration.

2.2.2 FP

In FP after the construction phase one also ends up with a collection of Agents either a list or a Map. Updating in FP is more subtle because it lacks references and mutable data. In case of the sequential strategy more work needs to be done and we can see the problem in general as a fold over the list of agents. In the case of the parallel strategy we can directly make use of FPs immutability.

[2] We are aware that OO is characterized by many more features e.g. inheritance, but we don't go into those details here as they are not relevant anyway - we simply want to show the subtle differences in Agent-representation of FP and OO where it suffices to emphasise the concept of implicitly / explicitly bound data

References

[1] Bloch, J. *Effective Java (2nd Edition)*. Createspace Independent Pub, Oct. 2014. Google-Books-ID: 5jXGoQEACAAJ.