The Pure Functional and Object Oriented Paradigm in Agent-Based Simulation

jonathan.thaler@nottingham.ac.uk

September 5, 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 precisely identify these fundamental differences, compare them and also look into general benefits and drawbacks of each approach.

Contents

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

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

1.2 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.

Chapter 2

Functional Programming

MacLennan [4] defines Functional Programming as a methodology and identifies it with the following properties:

1. It is programming without the assignment-operator.
2. It allows for higher levels of abstraction.
3. It allows to develop executable specifications and prototype implementations.
4. It is connected to computer science theory.
5. Parallel Programming.
6. Suitable for AI.

The last two points don't weight as heavy today as back in 1990 as other languages came up with features for better parallel programming but they all do it by introducing functional features.

MacLennan [4] defines properties of pure expressions

- Value is independent of the evaluation order.
- Expressions can be evaluated in parallel.
- Referential transparency.
- No side effects.
- Inputs to an operation are obvious from the written form.
- Effects to an operation are obvious from the written form.

Thus functional programming is identified as programming without the assignment operator and with pure expressions instead. Further characteristics are the missing of orderings as in imperative programming, caused by assignments: in functional programming the style is applicative which means we apply values to functions. The fundamental theoretical root is in the lambda calculus.

- cite critics

2.1 Applicative Programming

TODO: The question is then if we could implement in a functional style in an imperative object-oriented programming language? Or put otherwise: are these properties unique to functional programming or can we program functional in an imperative language (be it OO or not)? We can say that the functional style can be described as applicative programming: applying values to functions as described above. This functional style or applicative style applies to imperative languages as well and is not restricted to functional languages alone

2.2 Functional Programming

[4] defines functional programming as applicative programming with *higher-order* functions. These are functions which operate themselves on functions: they can take functions as arguments, construct new functions and return them as values. This is in stark contrast to the *first-order* functions as used in applicative or imperative programming which just operate on data alone. Higher-order functions allow to capture frequently recurring patterns in functional programming in the same way like imperative languages captured patterns like GOTO, while-do, if-then-else, for. Common patterns in functional programming are the map, fold, zip, operators.

TODO: discuss function-composition, which is the equivalent of the ; operator in imperative languages. Function composition has no side-effects as opposed to the imperative ; operator which simply composes destructive assignment statements which are executed after another resulting in side-effects. At the heart of modern functional programming is monadic programming which is polymorphic function composition: one can implement a user-defined function composition by allowing to run some code in-between function composition (TODO: explain using some code) - this code of course depends on the type of the Monad one runs in. This allows to emulate all kind of effectful programming in an imperative style within a pure functional language. Although it might seem strange wanting to have imperative style in a pure functional language, some problems are inherently imperative in the way that computations need to be executed in a given sequence with some effects. Also a pure functional language needs to have some way to deal with effects otherwise it would never be able to interact with the outside-world and would be practically useless. The real benefit of monadic programming is that it is explicit about

side-effects and allows only effects which are fixed by the type of the monad - the side-effects which are possible are determined statically during compile-time by the type-system. Some general patterns can be extracted e.g. a map, zip, fold over monads which results in polymorphic behaviour - this is the meaning when one says that a language is polymorphic in its side-effects.

TODO: so functional programming is not really possible in this way in classic imperative languages e.g. C as you cannot construct new functions and return them as results from functions. The question is if it is possible in OO by using some OO features to work around the limitations of procedural languages.

TODO: algebraic reasoning. [4] page 233: "Much of the power of functional programming derives from its ability to manipulate programs algebraically by means of identities such as Eq. (6.6)"

[2] defines Functional Programming as

2.3 Haskell

The language of choice for discussing real implementations of ABS in the pure functional programming paradigm we select Haskell. The reason is that it is a mature language with lots of useful and stable libraries and because it has been proved to be useful in Real-World applications as well (TODO: take from 1st year report). Also the reason why selecting Haskell over e.g. Scala, Clojure is its purity, strong static type-system, non-strictness.

Chapter 3

Object-Oriented Programming

- OO has not a generally accepted theory behind it although there have been attempts to establish one [1] - OO is thus a bunch of concepts and terms and methods that make up this methodology - this methodology was inspired from AI and human-computer interaction (alan kay, sutherland) and has grown and matured in the 90s. Its development was driven mainly by the software-industry which painfully has learned how to best use OO over the time of a decade. - here we focus on imperative OOP (there are mixed-paradigm oo / functional with oo languages: F#, OCAML, Scala?) because imperative OOP is primarily used in implementing ABS - there are concepts which show up both in OO and functional languages e.g. type-classes and inheritance of type-classes in haskell, but that does not make haskell an OO language - rather it shows that there are type-theoretic concepts which are not unique to OO. - we also focus on 'modern' OOP as it is implemented and used in the languages Java, C# and C++. There are fundamental differences between these implementations of OO and the original ideas alan kay had (no references, messages no mutable state). Some languages are much closer to the original version (e.g. Smalltalk) but are not widely used anymore. - cite critics of OO - study [1] to derive the concepts of OOP from a theoretical point-of-view - important terms / concepts of OOP -¿ Liskov substitution principle -¿ Dynamic dispatch -¿ Encapsulation -¿ Subtype polymorphism -¿ object inheritance -¿ Open recursion

3.1 Java

TODO: discuss applicative- and functional programming constructs provided in java

As the language of choice for discussing the object-oriented paradigm in implementing ABS is Java. The reason for this is that it is a very popular language widely in use implementing ABS and the basis for ABS libraries and

frameworks as Repast Simphony and AnyLogic. Other options would have been C++ which we abandoned due to its high inherent complexity and C# which can be seen as roughly equivalent to Java.

Chapter 4

Agent Representation

4.1 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 [3]. 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. TODO: add

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

some illustrating code

4.2 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 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. TODO: add some illustrating code

[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

Chapter 5

Agent Updating

5.1 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.

5.2 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.

Chapter 6

Agent-Agent Interactions

6.1 OO

In OO we have basically two possibilities to implement Agent-Agent Interactions: either by direct method calls to the other Agent which requires the calling Agent to hold a reference *with the subtype of the callee if using an abstract Agent-Class* OR by adding messages to a message-box (e.g. a HashMap or List) of the receiving Agent. Both have fundamental differences: a direct method call is like transferring the action to the callee: the Agent suddenly becomes active where before the calling Agent was active - as soon as the callee has finished the method, action returns to the callee. Note that the callee can again call other Agents methods which could, if not guarded explicitly against, lead to a cycle if the model-semantics permit it. When adding a message to a message-box no action is transferred to another Agent. Still a reference to the Agent with type of the abstract Agent-Class must be held if one implements it as a direct access to the mailbox. This has the advantage that it is fast but the disadvantage that we deal again with references and need synchronization in case of parallelism/-concurrency. Another option would be to have a outgoing-box into which the Agent adds messages it wants to send and the ABS system handles then the delivery after each step. This has the advantage that no direct references to other Agents are required, only their (numerical) IDs but the disadvantage that this delivery process has O(n^2) complexity (TODO: prove that or back it up with evidence. could we also reduce it to O (n log n) when using a map? or is it even possible to somehow use O (n)?)

6.2 FP

In FP when staying completely pure the only option we have is an outgoing-box into which messages are queued and the ABS system then distributes them into the ingoing-boxes of the receivers. This is because we have no method calls - of course we could simulate method calls by dragging the complete state around

which would allow to execute some message-handler by the receiving Agent within the calling Agent but then the Agents themselves need to be aware of implementation-details and have full access to all other agents - reasoning becomes difficult and robustness will inevitably suffer, so we won't go there. If we allow explicit side-effects in the form of Monadic programming then we can implement direct access to other Agents mailboxes as in the OO version: either we use references and run in the IO monad or we use STM channels and run in the STM monad. As IO would ruin very much we can reason about the most reasonable approach would be to make use of STM. But as long as there is no real need for concurrency as in the concurrent- and actor-strategies STM is an overkill and we will stick to outgoing boxes. This buys us the reasoning abilities and robustness but hits us with a penalty in performance.

Chapter 7

Environment Representation

7.1 OO

An Environment can be represented quite arbitrarily in OO and shared between the Agents using references. The downside is that it must be protected in case of parallel or concurrent access.

7.2 FP

Here we can go again two ways: stay pure or use explicit side-effects using IO or STM references. When staying pure the environment must be passed in through the behaviour function - and returned if it has changed. This allows us to easily reason which functions only read the environment, change it or don't need it at all: its visible from the type of the function and we can guarantee it statically at compile-time. Things are not so when using references (either STM or IO) as reading or writing both requires to run in the Monad thus making it not possible any more to tell at compile-time and rather from the type if its a read or write operation - we can only tell that the environment is touched or not. Also the behaviour-function must run in the respective monad although the environment is never accessed, thus removing further abilities to reason.

Chapter 8

Environment Updating

8.1 OO

The ABS system holds a reference to the Environment which will be accessed and changed by the Agents through references. The Environment class can implement some update-function which can then be called by the ABS system in every step, allowing the environment to update itself (e.g. regrowing some resource)

8.2 FP

In the FP version the ABS holds also an instance of an environment data-structure and if the environment should be able to update itself (e.g. regrowing some resource) then simply a environment-behaviour function must be provided which just maps the environment-type to the environment-type and has some time-input: (Double -¿ e -¿ e)

Chapter 9

Agent-Environment Interactions

9.1 OO

Agents simply call methods on the Environment to which they hold a reference thus changing the environment. The problem is much more subtle if we want true parallel or concurrent / actor strategies. In the case of parallel strategy the other agents must not see the changes to the environment. In the case of the concurrent / actor strategies the access to the environment must be synchronized.

9.2 FP

Chapter 10

Replications

10.1 OO

Replications need careful considerations in OO especially when using global references and data *when running them in parallel* [1]. All objects which have mutable state need to be accessed only by one replication thus one might to change the implementation to allow parallel replication.

10.2 FP

In FP running Replications in parallel or not makes to difference from a programming perspective: because of the nature of FP we can execute multiple simulations in parallel. Of course copying the initial agents and environment is necessary but that is very easily done in FP.

[1] If not then one does not need to care further

References

[1] Abadi, M., and Cardelli, L. *A Theory of Objects*, 1st ed. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] Allen, C., and Moronuki, J. *Haskell Programming from First Principles*. Allen and Moronuki Publishing, July 2016. Google-Books-ID: 5FaX-DAEACAAJ.

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

[4] MacLennan, B. J. *Functional Programming: Practice and Theory*. Addison-Wesley, Jan. 1990. Google-Books-ID: JqhQAAAAMAAJ.