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

jonathan.thaler@nottingham.ac.uk

September 15, 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

The most important property of a general purpose programming language is being *Turing Complete*. This term says roughly that a programming language which has this property can compute anything which is *computable*. We will not go into the precise meaning of computability but it suffices to say that a problem is computable if we can formulate a program which, after a finite number of steps, produces an output for a given input to the problem. This also implies that there are non-computable problems e.g. comparing whether two infinite sets are extensionally equal is uncomputable. Turing completeness also says something about the theoretical limits of a programming language, termed the *halting-problem*: it is not possible for a program written in such a programming language to decide *in general* whether another program of a turing complete language terminates or not. From this follows, that all general purpose programming languages which are turing complete are equal in power: none can compute something another can't and all suffer the same limitations of the halting-problem. These fundamental insights come from the foundations and theory of computation and research on computability, which originated in the 1930s by the pioneering work of Turing, Church and Gödel. All of them have developed different models of computation but all were shown to be equal in power: they are all turing complete which means each of them can simulate the others [1]. Today's general purpose programming languages can be generally categorized into two categories: they either build on the model of the Turing Machine as introduced by Alan Turing or on the Lambda Calculus as introduced by Alonzo Church. Thus assuming that all general purpose programming languages are turing complete, one could ask the following questions:

1. Why don't we implement our problems directly in a Turing Machine or the Lambda Calculus?
2. Why are all these programming languages necessary in the first place if all are equal in power?

[1] This implies that a language which guarantees that its programs always halt, is not powerful enough to interpret itself.

3. What are the implications for a programming language when choosing the one or the other model of computation as their foundation?

The reason why we do not program directly in these models of computation is because the raw power quickly becomes unmanageable and too complex which is also due to the lack of a type system. The interesting thing though is that compared to the Turing Machine, the Lambda Calculus is much more manageable in terms of complexity. When one programs in a TM, the solution gets extremely complicated very quickly whereas one can program quite a while in LC because of its fundamental different nature [2].

Implementing simple arithmetic operations on natural numbers can be already quite challenging with surprisingly substantial amount of complexity [3] In the subsequent chapters on functional programming and object-oriented programming we will give implementations of such simple arithmetic operations in the Lambda Calculus and the Turing Machine respectively - this will also show by example that the raw power of the Lambda Calculus is more manageable than the one of the Turing Machine. Still we can conclude that the TM and LC are basically only useful as theoretical foundations and to study fundamental and foundational problems but not to solve real problems. This is because we think problems different than a TM or LC works. So both do not allow us directly to express in the way we think, we need to build more mechanisms on top of this raw complexity. To control complexity, the best solution has always been to add layers of abstraction. Thus we build up more and more levels of abstractions where each depends on preceding ones. Some programming languages stop at some point, where other languages go further in abstractions - the point here is that not all programming languages are equal in their abstractions of complexity.

In the end a programming language is a tool to solve a problem by expressing the solution in this language. Depending on the underlying computational model some problems may be easier to solve in different languages *because the language supports expressing a given solution more natural for the given problem.* When looking closer at the nature of the Turing Machine and Lambda Calculus we can attribute the Turing Machine to be an operational model and the Lambda Calculus to be a denotational one. Roughly speaking, an operational model describes *how* to compute something whereas a denotational model describes *what* to compute. This fundamental difference of operational vs. denotational is directly reflected in programming languages which build on these two different models of computation: imperative languages follow the operational model of the TM and functional languages basically the denotational model of the LC. Although today's computers are basically highly efficient Turing Machines and

[2] Actually when programming in Haskell (or ML / similar class of families) one programs basically in a slightly 'sugarized' version of the Lambda Calculus.

[3] Note that these seemingly 'trivial' problems already require substantial amount of work, namely the encoding of natural numbers in either the Turing Machine or the Lambda Calculus, which is not trivial. This for a good reason: operations on natural numbers were used by Gödel to study the basics of computation.

follow thus the operational model [4] this means that also declarative, functional languages are ultimately translated into an operational model, which is possible because both models are of the same power. This but does not make functional languages operational, what matters here is that they allow to approach a problem from a very different perspective - how it is ultimately executed does not matter and is not visible to the programmer, nor should he or she think about it. The conclusion is that because of the different approach of operational vs. denotational, one thinks problems very different in either models which in turn also implies that different paradigms and languages are differently well suited to formulate solutions to problems.

All this ultimately leads to the question of how well the pure functional and object-oriented paradigms are suited to formulate ABS in comparison to each other. Also we are interested in the strengths of both paradigms in general and ask if they apply directly when formulating ABS. In this text we want to give an extensive answer to these questions by first looking closer into the abstractions used in pure FP and OOP to formulate an ABS and what the challenges one encounters in doing so.

1.1 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

[4] TODO: why this is so is probably due to history reasons and because probably also a philosophical problem: when facing the real world we need simply operational models as this is the way to manifest something in the real-world, declarative model resorts more to magic which is not as reliable.

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 [5] defines Functional Programming as a methodology and identifies it with the following properties (amongst others):

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. Suitable for Parallel Programming.
6. Algebraic reasoning.

[2] defines Functional Programming as "a computer programming paradigm that relies on functions modelled on mathematical functions." Further they explicate that it is

- in Functional programming programs are combinations of expressions
- Functions are *first-class* which means the can be treated like values, passed as arguments and returned from functions.

[5] makes the subtle distinction between *applicative* and *functional* programming. Applicative programming can be understood as applying values to functions where one deals with 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.

Note that applicative programming is not necessarily unique to the functional programming paradigm but can be emulated in an imperative language e.g. C as well. Functional programming is then defined by [5] 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. 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 [1].

The equivalence in functional programming to to the *;* operator of imperative programming which allows to compose imperative statements is function composition. 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 - 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.

2.1 Theoretical Foundation

The theoretical foundation of Functional Programming is the Lambda Calculus, which was introduced by Alonzo Church in the 1930s. After some revision

[1] Object-Oriented languages like Java let you to partially work around this limitation but are still far from *pure* functional programming.

due to logical inconsistencies which were shown by Kleene and Rosser, Church published the untyped Lambda Calculus in 1936 which, together with a type-system (e.g. Hindler-Milner like in Haskell) on top is taken as the foundation of functional programming today.

[5] defines a calculus to be "... a notation that can be manipulated mechanically to achieve some end;...". The Lambda Calculus can thus be understood to be a notation for expressing computation based on the concepts of *function abstraction*, *function application*, *variable binding* and *variable substitution*. It is fundamentally different from the notation of a Turing Machine in the way it is applicative whereas the Turing Machine is imperative / operative. To give a complete definition is out of the scope of this text, thus we will only give a basic overview of the concepts and how the Lambda Calculus works. For an exhaustive discussion of the Lambda Calculus we refer to [5] and [3].

Function Abstraction Function abstraction allows to define functions in the Lambda Calculus. If we take for example the function $f(x) = x^2 - 3x + a$ we can translate this into the Lambda Calculus where it denotes: $\lambda x.x^2 - 3x + a$. The λ symbol denotes an expression of a function which takes exactly one argument which is used in the body-expression of the function to calculate something which is then the result. Functions with more than one argument are defined by using nested λ expressions. The function $f(x, y) = x^2 + y^2$ is written in the Lambda Calculus as $\lambda x.\lambda y.x^2 + y^2$.

Function Application When wants to get the result of a function then one applies arguments to the function e.g. applying $x = 3, y = 4$ to $f(x, y) = x^2 + y^2$ results in $f(3, 4) = 25$. Function application works the same in Lambda Calculus: $((\lambda x.\lambda y.x^2 + y^2)3)4 = 25$ - the question is how the result is actually computed - this brings us to the next step of variable binding and substitution.

Variable Binding In the function $f(x) = x^2 - 3x + a$ the variable x is *bound* in the body of the function whereas a is said to be *free*. The same applies to the lambda expression of $\lambda x.x^2 - 3x + a$. An important property is that bound variables can be renamed within their scope without changing the meaning of the expression: $\lambda y.y^2 - 3y + a$ has the same meaning as the expression $\lambda x.x^2 - 3x + a$. Note that free variable *must not be renamed* as this would change the meaning of the expression. This process is called α-conversion and it becomes sometimes necessary to avoid name-conflicts in variable substitution.

Variable Substitution To compute the result of a Lambda Expression - also called evaluating the expression - it is necessary to substitute the bound variable by the argument to the function. This process is called β-reduction and works as follows. When we want to evaluate the expression $((\lambda x.\lambda y.x^2 + y^2)3)4$ we first substitute 4 for x, rendering $(\lambda y.4^2 + y^2)3$ and then 3 for y, resulting in $(4^2 + 3^2)$ which then ultimately evaluates to 25. Sometimes α-conversion becomes necessary e.g. in the case of the expression $((\lambda x.\lambda y.x^2 + y^2)3)y$ we must

not substitute y directly for x. The result would be $(\lambda y.y^2+y^2)3 = 3^2+3^2 = 18$ - clearly a different meaning than intended (the first y value is simply thrown away). Here we have to perform α-conversion before substituting y for x. $((\lambda x.\lambda y.x^2 + y^2)3)y = ((\lambda x.\lambda z.x^2 + z^2)3)y$ and now we can substitute safely without risking a name-clash: $((\lambda x.\lambda z.x^2 + z^2)3)y = (\lambda z.y^2 + z^2)3) = (y^2 + 3^2)3) = y^2 + 9$ where y occurs free.

Examples

$(\lambda x.x)$ denotes the identity function - it simply evaluates to the argument.

$(\lambda x.y)$ denotes the constant function - it throws away the argument and evaluates to the free variable y.

$(\lambda x.xx)(\lambda x.xx)$ applies the function to itself (note that functions can be passed as arguments to functions - they are *first class* in the Lambda Calculus) - this results in the same expression again and is thus a non-terminating expression.

We can formulate simple arithmetic operations like addition of natural numbers using the Lambda Calculus. For this we need to find a way how to express natural numbers [2]. This problem was already solved by Alonzo Church by introducing the Church numerals: a natural number is a function of an n-fold composition of an arbitrary function f. The number 0 would be encoded as $0 = \lambda f.\lambda x.x$, 1 would be encoded as $1 = \lambda f.\lambda x.fx$ and so on. This is a way of *unary notation*: the natural number n is represented by n function compositions - n things denote the natural number of n. When we want to add two such encoded numbers we make use of the identity $f^{(m+n)}(x) = f^m(f^n(x))$. Adding 2 to 3 gives us the following lambda expressions (note that we are using a sugared version allowing multiple arguments to a function abstraction) and reduces after 7 steps to the final result:

$2 = \lambda fx.f(fx)$
$3 = \lambda fx.f(f(fx))$
$ADD = \lambda mnfx.mf(nfx)$

ADD 2 3
$1 : (\lambda mnfx.mf(nfx))(\lambda fx.f(f(fx)))(\lambda fx.f(fx))$
$2 : (\lambda nfx.(\lambda fx.f(f(fx)))f(nfx))(\lambda fx.f(fx))$
$3 : (\lambda fx.(\lambda fx.f(f(fx)))f((\lambda fx.f(fx))fx))$
$4 : (\lambda fx.(\lambda x.f(f(fx)))((\lambda fx.f(fx))fx))$
$5 : (\lambda fx.f(f(f(\lambda fx.f(fx))fx)))$
$6 : (\lambda fx.f(f(f(\lambda x.f(fx))x)))$
$7 : (\lambda fx.f(f(f(f(fx)))))$

[2] In the short introduction for sake of simplicity we assumed the existence of natural numbers and the operations on them but in a pure lambda calculus they are not available. In programming languages which build on the Lambda Calculus e.g. Haskell, (natural) numbers and operations on them are built into the language and map to machine-instructions, primarily for performance reasons.

2.1.1 Types

The Lambda Calculus as initially introduced by Church and presented above is *untyped*. This means that the data one passes around and upon one operates has no type: there are no restriction on the operations on the data, one can apply all data to all function abstractions. This allows for example to add a string to a number which behaviour may be undefined thus leading to a non-reducible expression. This led to the introduction of the simply typed Lambda Calculus which can be understood to add tags to a lambda-expression which identifies its type. One can then only perform function application on data which matches the given type thus ensuring that one can only operate in a defined way on data e.g. adding a string to a number is then not possible any-more because it is a semantically wrong expression. The simply typed lambda calculus is but only one type-system and there are much more evolved and more powerful type-system e.g. *System F* and *Hindley-Milner Type System* which is the type-system used in Haskell. It is completely out of the scope of this text to discuss type systems in depth but we give a short overview of the most important properties.

Generally speaking, a type system defines types on data and functions. Raw data can be interpreted in arbitrary ways but a type system associates raw data with a type which tells the compiler (and the programmer) how this raw data is to be interpreted e.g. as a number, a character,... Functions have also types on their arguments and their return values which defines upon which types the function can operate. Thus ultimately the main purpose of a type system is to reduce bugs in a program. Very roughly one can distinguish between static / dynamic and strong / weak typing.

Static and dynamic typing A statically typed language performs all type checking at compile time and no type checking at runtime, thus the data has no type-information attached at all. Dynamic typing on the other hand performs type checking during run-time using type-information attached to values. Some languages use a mix of both e.g. Java performs some static type checking at compile time but also supports dynamic typing during run-time for downcasting, dynamic dispatch, late binding and reflection to implement object-orientation. Haskell on the other hand is strictly statically typed with no type checks at runtime.

Strong and weak typing A strong type system guarantees that one cannot bypass the type system in any way and can thus completely rule out type errors at runtime. Pointers as available in C are considered to be weakly typed because they can be used to completely bypass the type system e.g. by casting to and from a (void*) pointer. Other indications of weak typing are implicit type conversions and untagged unions which allow values of a given typed to be viewed as being a different type. There is not a general accepted definition of strong and weak typing but it is agreed that programming languages vary across the strength of their typing: e.g. Haskell is seen as very strongly typed, C very

weakly, Java more strongly typed than C whereas Assembly is considered to be untyped.

2.2 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.

Being *pure* describes that all Haskells features are directly translatable to the Lambda-Calculus. Other functional languages (e.g. Scala), although also based on the Lambda-Calculus like all functional languages, have features which are not directly translatable into the Lambda-Calculus and are thus considered to be impure. Another meaning of *pure* in the context of Haskell and of functional programming can be understood as *free from side-effects* or more precisely as having the property of *referential transparency*.

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

2.3 Conclusions

- lambda calculus very close to mathematical notation: denotational / declarative way to think about computation: describe WHAT one wants to compute. - referential transparency & evaluation order independence: absence of side-effects and stateful machine (as in TM) removes difficult bugs as well. - static strong type system: can guarantee absence of huge number of bugs already at compile-time. Still non-terminating functions possible (this would need more sophisticated type-system e.g. Agda, Coq,...). - disadvantages: reasoning about space- and time-leaks but that depends also on the programming language and its implementation

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 Theoretical Foundation

There is no direct theoretical foundation for Object-Oriented Programming because it is rather a modelling and engineering approach but it builds heavily on the imperative style in turn builds directly on the Turing Machine. Still there have been attempts on developing a theory for OOP as in [1], which we will discuss briefly below.

3.1.1 Turing Machine

[6]

3.1.2 Types

We already introduced the basics of a type system in Chapter 2 and discussed the terms of static / dynamic and strong / weak typing. Due to the nature of the paradigm oo type systems tend to be dynamically weakly typed. TODO: is this true? e.g. what with Smalltalk? other exotic languages? Java is definitely weak dynamically typed with some amount of static type-checking during compile time.

[1]

3.1.3 An Example: arithmetic operations

TODO: addition and multiplication in the turing machine

3.2 Java

TODO: discuss applicative- and functional programming constructs provided in java, investigate lambdas in java [ ] 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 make it clear that just because java has now lambdas does not make it functional.

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 [4]. 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] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984. Google-Books-ID: KbZFAAAAYAAJ.

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

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

[6] Webber, A. B. *Formal Language: A Practical Introduction*. Franklin, Beedle & Associates Inc., Wilsonville, OR, USA, 2008.