Comparing the Pure Functional and Object Oriented Paradigm for implementing ABS

jonathan.thaler@nottingham.ac.uk

August 1, 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

0.1 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 / operatiinal 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

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

[ ] 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?

0.2 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

0.3 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

References