Semantics

Johan Montelius

Spring Term 2020

Introduction

This is an attempt to clarify how our miniature programming language is given an interpretation by its relation to the lambda calculus and being defined by its operational semantics. You should have read both about the lambda calculus and the operational semantics, this text is meant to tie the knots together.

1 why the lambda calculus

The lambda calculus is the foundation of any functional programming language. By translating programming constructs to lambda expressions we give a precise meaning to the constructs i our language. A pure functional programming language does not have any constructs that can not be described as lambda expressions. Many languages do however introduce extensions that fall outside the scope of lambda calculus. Extending a functional programming language beyond the lambda calculus is fine but we should know which constructs that violates the rules and have a good reason to include them.

If we have the expression below, we could argue what the result should be.

```
y = 3; f = fn x -> x + y; y = 2; f.(y)
```

If we can give it an interpretation as a lambda expression we will remove any ambiguities (or be confused on a higher level).

$$(\lambda y \rightarrow (f \rightarrow (\lambda y \rightarrow f \quad y) \quad 2)(\lambda x \rightarrow x + y)) \quad 3$$

In this course our focus is not on the lambda calculus and we will not dig deeper into how we can define arithmetic nor data structures such as atoms or tuples. We do not explain how pattern matching can be described but you should have seen the lambda calculus and understand its relation to our functional programming language.

2 the operational semantics

Even if we could give a complete description of all our language constructs in terms of lambda expressions we would not have a precise definition of our language. The problem is that even though the lambda calculus clearly defines what the result should be if we had infinite amount of time, it does not define in what order things are done.

Since we are interested in not only the final result of a computation but also the time and memory needed to perform the execution, we need something more. This is where our operational semantics comes in.

The rules of our operational semantics leaves less doubt about what will actually happen during execution. The lambda calculus simply describe what is allowed but not i what order things should be done. The beauty of the lambda calculus is that if one evaluation strategy results in an answer so will any evaluation strategy ... if it terminates.

The operational semantics should be precise enough for a programmer to control the execution in order to avoid inifinte computations and estimate time and memory complexity. We could still allow alternative execution orders but it should be clear to the programmer when these alternatives exist.