Chapter 1

Introduction

Functional programming is an implementation mathematical models for computations. Therefore functional programming languages appear quite elegant, powerful, predictable, and analyzable. Every programmer, functional or not, should be concerned with the quality of the code he or she produces. What properties hold about it? Is it safe to execute? What is the code supposed to do? Does it do it? Always? Or only sometimes? How fast is it? A few test cases worked already, does this mean that the code is ok?

The reader should have noticed in his or her career as computer scientist so far that the answer to these questions depend to a good deal on which language is used. In C, for example, it is easy to write a functions that adds up all the elements in a list and returns the desired result. Clearly, that the function always returns the same result depends crucially on the fact that no subroutines changes the content of the array. In Java, programmers do not have to be concerned about memory, because the allocations and deallocations of memory are done automatically by the Java virtual machine. But this does not mean that all problems are solved, on the contrary, because aliasing is permitted objects may be referred to by two or more different names. And again, it is very difficult to quantify locally over the properties of a procedure, because their behavior can only be described globally.

Those features, that render local reasoning largely impossible are called imperative features of a programming language. The name comes from the early days of computer science, where programs were understood as a sequence of commands: “load memory cell x into register i”, “Add register i to register j” “Jump to line number x ”, “Update memory cell x with the value in register i”. In summary, the property of a procedure in an imperative language always depends on the entire and global state of the system, because every procedure can have side effects.

One way to localize properties and behaviors of procedures is to treat them mathematically as functions. The function is side effect free, which means it expects input arguments and maps them to results, without having any effects of any future evaluation of any future function call. Under such an interpretation, programs simply compute mathematical functions. Therefore, we never have to worry how functions influence one another from a global point of view. On the contrary, the interpreter guarantees that it will always compute the same result of a function when called with the same arguments, time after time the function is evoked. This actually makes it possible to reason mathematically about properties of a programs, very often by induction, but also by coinduction, or logical relations.

The objective of this course is to convey the fundamental ideas and design principles underlying functional programming languages. After completing the course, students will be able to write programs in various functional languages, and be able to think critically about their features, and underlying design choices. We approach functional programming languages from a semantical point of view, and reason about their properties formally. Students will gain hands-on experience with eager and lazy programming languages, such as Standard ML and Haskell, respectively. Eventually, we will discuss possible future trends in the design of functional programming languages, including dependent datatypes, and programming with open objects.

1.1 Deductive Systems

The way we think semantically about functional programming languages is in form of deductive systems. Surely, there are many different kind of semantics which we could build on, including, denotational semantics, where the meaning of a program is identified with its denotation in a mathematical domain. We however will concentrate these lectures on the operational semantics on programs, that is we identify the meaning of a program with the result of executing it. The main advantage of doing so is that we will actually learn how functional programming languages are implemented.

The mathematics we use to describe operational semantics are deductive systems which we discuss in this section. Deductive systems, are quite common in the area of programming language technology, that it is worth explaining them in detail here. Another example of inductively defined structure are abstract syntax trees that corresponds to programs to which we will refer often as *expressions*. Expressions are different from concrete syntax in that they abstract away from details that are present in concrete implementations of programming languages such as SML or Haskell. Other examples whose meaning can be nicely expressed using deductive systems are typing derivations, and operational semantics. For the philosophical underpinnings of deductive systems I recommend Per Martin-Löf's paper "on the meanings of the Logical Constants and Justifications of the Logical Laws" [**?**].

Deductive systems are described by judgments and inference rules. Judgments stand for something which we hope to hold true, s.t. "e is an expression" or "v is a value", or $\cdot \vdash e : \tau$ which relates expressions e to their respective type τ for concrete instances of e, v or τ. The truth of these judgments is expressed by evidence — and evidence are trees whose nodes are justifications. Justifications are expressed as inference rules are always of the form

$$\frac{J_1 \quad \dots \quad J_n}{J}\,\mathsf{name}$$

where we refer each J_i as premiss, J the conclusion, and name the name of the rule. If an inference rule does not define any premisses, we call it an axiom. This might be only a side remark, but note, that in the formulation of the inference rules the judgments may contain variables, such as e, v, or τ. These variables are called *schematic variables*.

We use calligraphic letters $\mathcal{D}$, $\mathcal{E}$, or $\mathcal{P}$ to denote derivations of judgment J. To express that $\mathcal{D}$ is a valid derivation, we would write

$$\begin{array}{c}\mathcal{D}\\ J\end{array}$$

or simply $\mathcal{D} :: J$.

Example 1.1.1 (Well formed expressions) Here are two rules that define well-formed integers $0, 0+1, 0+1+1, \ldots$.

$$\frac{}{0 \text{ is an expression}}\,\mathsf{wff\_zero} \qquad \frac{e \text{ is an expression}}{e+1 \text{ is an expression}}\,\mathsf{wff\_succ}$$

The rule wff_zero provides evidence that the number 0 is an expression, and the other rule wff_succ turns evidence that e is an expression into evidence that $e+1$ is also an expression. The e in the second rule is a schematic variable. The following derivation provides evidence that $0+1+1$ is an expression.

$$\frac{\dfrac{\dfrac{}{0 \text{ is an expression}}\,\mathsf{wff\_zero}}{0+1 \text{ is an expression}}\,\mathsf{wff\_succ}}{0+1+1 \text{ is an expression}}\,\mathsf{wff\_succ}$$

Judgments and evidence in form of inference rules are our main vehicle of mathematics to describe programming languages. When we prove things about are systems we either do this by induction, coinduction, or logical relations. In the next few weeks we will see plenty of applications of those proof techniques.

Exercise 1.1.1 Read Per Martin-Löfs paper. Summarize and explain all different kinds of judgments described in the paper.

Exercise 1.1.2 Imagine you have to devise an elevator system in a house with five stories where each floor is equipped with one call button for the elevator. The owner of the building is not interested in how you will implement the controls, what he's interested in is that you know how to model a valid elevator ride. He suggests that all requests are recorded in lists in the order they were received.

$$\begin{array}{llll} \text{Stories:} & s & ::= & 1 \mid 2 \mid 3 \mid 4 \mid 5 \\ \text{Requests:} & r & ::= & \cdot \mid s; r \end{array}$$

With t we denote the the time, where we assume that every movement of the element happens in unit time. Assume that the elevator is on the first floor at time 0.

1. Define the meaning of judgment $r \vdash s/t$ by specifying a set of inference rules which reads as follows. It is possible to satisfy all requests r in t movements after the elevator arrives in story s.

2. Prove formally that for each list of requests, a valid elevator ride exists.

3. In your formal system, give a formal proof that there exists a derivation

$$\begin{array}{c} \mathcal{D} \\ 4; 2; \cdot \vdash s/t \end{array}$$

1.2 Types

Functional programming languages come in different flavors. Some have a strong typing discipline (such as ML, Haskell), others rely on dynamic typing (such as Scheme) or do not have any types at all (such as LISP). This course is based on functional programming languages with strong type systems. The reason why prefer strong type systems, is because types approximate the invariant that a function satisfies. Consider for example the standard factorial function written in ML.

```
fun fac 0 = 1
  | fac n = n * (fac (n - 1))
```

The invariant that this function satisfies is the following formula

$$\forall n : \texttt{int}.\, \exists m : \texttt{int}.\, n \geq 0 \wedge \texttt{fac}\; n \hookrightarrow m \wedge m = n! \tag{1.1}$$

which can be easily verified by induction on n. Clearly, we know that n and m range over integer numbers, but n cannot be negative. Let's look for a moment at the following invariant where we write $\top$ for true.

$$\forall n : \texttt{int}.\, \exists m : \texttt{int}.\, \top \tag{1.2}$$

It is clear that (1.2) is also satisfies this invariant for `fac` above. And it is also clear that (1.1) implies (1.2). If we look at this implication from a counter-positive point of view, we see that $\neg$ (1.2) implies $\neg$ (1.1). Consequently, although (1.2) is not the invariant we might be interested in, if it fails to hold the stronger invariant cannot hold either. It is the weak invariants that common type system try to capture.

The set of types which may occur in functional programming languages include base types, such as integers, float, maybe in32, and int64, Booleans, of course, and strings; composite types, such as function types, pair types, sometimes sum types; often type variables which are defined by some kind of polymorphism. In addition, in ML we have reference types, which introduce an imperative feature to ML. Haskell on the other hand provides monadic types, such as the state monad, or the IO monad.

What happens if we omit types from a functional languages? In most cases: nothing! Most languages are designed in such a way that if types are erased, programs can still be executed. In other implementations however type information is exposed to the program during runtime. When printing an object, for example, we might want to select a printing subroutine depending on the type of the object that is to be printed. If types were removed, the printer would never know what to do. Those languages do not have the type erasure property.

It is fair to say, that a strong typing discipline prevents programmers from writing unexecutable code. If we engineer the right operational semantics for functional programming languages, we can actually show that well-typed code is guaranteed to never crash. The execution might get stuck (and terminates with an uncaught match exception), or run on forever. The first property is due to the *type preservation property.* Well-typed programs always evaluate to a value of the same type. If progress, and termination hold, we say that the programming language guarantees *type safety.*

Programming languages for which we can derive type information ahead of time are called statically typed languages. In most languages, not all types need be given by the programmer, which leads to the type inference problem. It is always possible to solve the type inference problem by endowing a program with more types. The `fac` program from above, for example, did not state the type of the argument n, but the ML type inference algorithm could successfully infer its type (from uses of n as an argument to "`-`", and "`*`").

One last world about dynamically typed languages. Those languages are designed in such a way, that type information is maintained during runtime. Before executing instructions, the dynamic types of the entities involved are checked. In the case of a type error, a runtime error is generated, otherwise the computation continues as planned. Therefore, a dynamically typed program can never crash either. Although quite effective, this technique does not capture the interpretation of types being approximations to the strongest invariant. We will not pursue the resulting questions further but refer the interested reader instead to [**?**].

1.3 Implementations

Working with deductive systems can be pretty tedious and difficult. Most systems that we will encounter consist of many inference rules. Proving properties can be pretty painful if many cases have to be checked. And in this enterprise it is really the case that every case counts. Many proofs go by structural induction on the defining rules of a deductive systems, but as we will there are also others.

In the last years, many tools have been developed whose goal it is to support the designer of programming languages, or other endeavors. Just to mention a few, there is Isabelle, Coq, PVS, or Twelf. The use of a proof assistant in this class is optional, but it may be useful to tackle some of the homeworks. If you want to use a proof assistant to encode and reason about deductive systems, I recommend Twelf.

Exercise 1.3.1 Implement your solution of Exercise 1.1.2 in Twelf.

b