Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

1 Introduction

What is a program? Is it just a list of instructions for the computer to do? If a program were just a simple list of instructions we wouldnt be able to do much useful with computers. Programs are much more interesting than that. If we think about a program as just a description of what to do, then part of what a program means is baked into the computer that interprets it. So you need to describe that computer too, and *thats* clearly more interesting than just a list of instructions.

This is a course about the *semantics* of programs and programming languages. "Semantics" is simply a fancy way of saying "meaning"; we want to give a precise meaning to programs. This is useful for many reasons. It means that we can understand what they are going to do, which is clearly helpful in writing programs that work. It also helps us when building tools like compilers, optimizers, and interpreters because it means we can judge whether these tools are implemented correctly.

Programs describe computation, but they're really a kind of information as well. Suppose we step back from my initial question and ask a truly fundamental question: "what is information?". For most of the information we normally think about, familiar mathematical tools like sets and sequences are adequate to describe that information. Programs describe computation, but programs are information too—a particularly interesting kind of information, especially for computer scientists. In this course well see some of the mathematical, formal tools that are needed to precisely describe what programs are, what programming languages are, and maybe better answers to the two questions above.

There are three major components to this course.

- Dynamic Semantics: We will study methods for describing what a program does or computes when it runs.
- Static Semantics: We will also study methods for reasoning about programs before they run. We would like to consider illegal progams that might go wrong in in various ways, before they run. We may consider some programs to be *ill-formed*, for example if they fail type checking and therefore might contain run-time (dynamic) type errors.
- Language Features: We will apply methods for dynamic and static semantics to study actual language features of interest, probably including some interesting features that many students have not seen before.

We would like to characterize the semantics of a program as a function produces the output based on inputs. More generally, real programs are reactive and interact with their inputs arriving from the environment. Describing this kind of program is more challenging, though we can view the reactive behavior of the program again as a function of the inputs it receives from the environment as it runs. Thus, to describe program semantics, we will build up some mathematical tools for constructing and reasoning about functions.

1.1 Review of mathematical functions

A binary relation on sets A and B is a subset $R \subseteq A \times B$. We write aRb to mean that the pair (a, b) is an element of R, meaning that R associates a with b. A *(total) function* is a relation, in which each element of a set A is associated with a *unique* element of another (possibly the same) set B. If f is such a function, we write:

$$f : A \to B$$

Here, A is the *domain* of the function and B the *codomain.* The *image* of the function is the set of elements in B that at least one element in A is mapped to.

$$\mathit{image}(f) = \{x \in B \mid x = f(a) \text{ for some } a \in A\} = \{f(a) \mid a \in A\}$$

We avoid the the word *range* because it is ambiguous: sometimes it is used to refer to the codomain, sometimes to the image. A *partial function* $f : A \rightharpoonup B$ is a relation that is a function on some subset of the domain $\mathrm{dom}(f) \subseteq A$; it associates each element of A with at most one element of B. We typically think of a function as a mathematical object that allows you to apply it to elements of its domain. Alternatively, we can view it as the set of all pairs (a, b) it relates. This set of pairs is called the functions *extension.*

1.2 Abstract syntax trees

One way of describing a function is via its extension. Alternatively, we can give an *intensional* representation—an expression that describes what you have to do in order to evaluate the function. This class is partly about how to get from intensional representations of computations to extensional representation.

In this class, the intensional representation of a program expression is the *abstract syntax tree* (AST) for that program. We are not concerned with *concrete syntax*, the actual characters that are used to express the program. Parsing is an interesting topic, but it is covered elsewhere. So when we write an expression like $2 + 3 \times 4$, we mean *exactly* the same expression as $2 + (3 \times 4)$. The parentheses are there just to make it clear what the parse tree is.

2 Lambda Calculus

The lambda calculus was originally developed in the 1930s before computers. It was developed by mathematicians, Alonzo Church and Stephen Cole Kleene, to describe functions in an unambiguous and compact manner. As discussed above, one way of describing functions mathematically is via their *extension*, which can be a list of pairs of (input,output) values, or a graph mapping one domain to the other. The lambda calculus gives an *intensional* representation—that is, a program, or what you have to do in order to evaluate the function. (As an aside, this class is quite a bit about how to get from an intensional representation, an algorithm, to the extension, meaning, or effect of a function.) The λ-calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to a *Turing machine.*

Real programming languages such as Lisp, Scheme, Haskell and ML are very much based on lambda calculus, although there are differences as well.

3 Untyped Lambda Calculus

3.1 Syntax

The following is the syntax of the λ-calculus. A *term* (or *expression*) is defined as follows:

$$\begin{array}{rcll} e & ::= & x & x \in \mathbf{Var} \quad \text{where } \mathbf{Var} \text{ is some countable set} \\ & | & e_0\, e_1 & \textit{(application)} \\ & | & \lambda x.\, e & \textit{(abstraction)} \end{array}$$

In an abstraction, x is the argument, e is the body of the function. Notice that this function doesn't have a name. In mathematics it is common to define functions as $f(x) = x^2$. A corresponding anonymous notation that is frequently used is $x \mapsto x^2$; this is really the same thing as $\lambda x.\, x^2$. One nice thing about lambda terms is that they are an *anonymous* representation of functions: the function doesn't have to have a name like f in order to talk about it.

Here are some examples of terms. First, we present (and define) the identity function:

$$ID = \lambda x.\, x$$

The next example is a function that will ignore its argument and return the identity function.

$$\lambda x.\, \lambda a.\, a$$

which is the same as

$$\lambda x.\, ID$$

Parentheses are used to show explicitly how to parse expressions, but they are not strictly necessary. It's a good idea to include them if you aren't sure. Lambda terms are like sums: they extend as far to the right as they can. For example, $(\lambda x.\, x\ \lambda y.\, y)$ is the same thing as $(\lambda x.\, (x\ \lambda y.\, y))$, not $((\lambda x.\, x)\ (\lambda y.\, y))$. Application is *left-associative* - that is, $e_1\ e_2\ e_3$ is the same thing as $(e_1\ e_2)\ e_3$.

3.2 Closed terms

An occurrence of a variable x in a term is said to be *bound* if there is an enclosing $\lambda x.\, e$; otherwise, it is *free*. A *closed term* is one in which all identiers are bound.

Consider the following term:

$$\lambda x.\, (x\ (\lambda y.\, y\ a)\ x)\ y$$

Both occurrences of x are bound, the first occurrence of y is bound, the a is free, and the last y is also free, since it is outside the scope of the λy.

If a *program* has some variables that are free, then you do not have a complete program as you do not know what to do with the free variables. Hence, a well formed *program* in lambda calculus is a closed term.

3.3 Higher-order functions

In lambda calculus, we can define functions that can take functions as arguments and/or return functions as results. In fact, every argument is a function and every result is a function. That is, functions are first-class values.

This example takes a function as an argument and applies it to 5:

$$\lambda f.\, (f\ 5)$$

We can further generalize. This function takes an argument v and returns a function that calls its argument on v:

$$\lambda v.\, \lambda f.\, (f\ v)$$

3.4 Multi-argument functions and currying

A multi-argument lambda calculus (with functions that take multiple arguments) could be defined as:

$$\begin{array}{rcll} e & ::= & \ldots\ \textit{earlier stuff}\ \ldots & \\ & \mid & \lambda x_1, x_2, \ldots, x_n.\, e & \text{(multi-argument abstraction)} \\ & \mid & e_0\ e_1\ e_2\ \ldots\ e_n & \text{(multi-argument application)} \end{array}$$

In the multi-argument application, e_0 is an n-argument function and $e_1, \ldots, e_n$ are the arguments.

It is known that this isn't any more expressive than basic lambda calculus. So, we stick to basic lambda calculus but we may occasionally write down terms that look like the above for convenience. It's just *syntactic sugar*, which means that it can be transformed into something more basic that does not require the extended syntax. A transformation that removes syntactic sugar is called *desugaring*.

Desugaring multi-argument functions:

$$\begin{array}{lcl} \lambda x_1, x_2, \ldots, x_n.\, e & \Rightarrow & \lambda x_1.\, \lambda x_2.\, \ldots \lambda x_n.\, e \\ e_0\ e_1\ e_2\ \ldots\ e_n & \Rightarrow & (\ldots((e_0\ e_1)\ e_2)\ \ldots e_n) \end{array}$$

The above transformation converts multi-argument functions to *curried* functions; the transformation is known as *currying*.