1 Introduction

What is a program? Is it just something that tells the computer what to do? Yes, but there is much more to it than that. The basic expressions in a program must be interpreted somehow, and a program's behavior depends on how they are interpreted. We must have a good understanding of this interpretation, otherwise it would be impossible to write programs that do what is intended.

It may seem like a straightforward task to specify what a program is supposed to do when it executes. After all, basic instructions are pretty simple. But in fact this task is often quite subtle and difficult. Programming language features often interact in ways that are unexpected or hard to predict. Ideally it would seem desirable to be able to determine the meaning of a program completely by the program text, but that is not always true, as you well know if you have ever tried to port a C program from one platform to another. Even for languages that are nominally platform-independent, meaning is not necessarily determined by program text. For example, consider the following Java fragment.

```
class A { static int a = B.b + 1; }
class B { static int b = A.a + 1; }
```

First of all, is this even legal Java? Yes, although no sane programmer would ever write it. So what happens when the classes are initialized? A reasonable educated guess might be that the program goes into an infinite loop trying to initialize `A.a` and `B.b` from each other. But no, the initialization terminates with initial values for `A.a` and `B.b`. So what are the initial values? Try it and find out, you may be surprised. Can you explain what is going on? This simple bit of pathology illustrates the difficulties that can arise in describing the meaning of programs. Luckily, these are the exception, not the rule.

Programs describe computation, but they are more than just lists of instructions. They are mathematical objects as well. A programming language is a logical formalism, just like first-order logic. Such formalisms typically consist of

- *Syntax*, a strict set of rules telling how to distinguish well-formed expressions from arbitrary sequences of symbols; and
- *Semantics*, a way of interpreting the well-formed expressions. The word "semantics" is a synonym for "meaning" or "interpretation". Although ostensibly plural, it customarily takes a singular verb. Semantics may include a notion of *deduction* or *computation*, which determines how the system performs work.

In this course we will see some of the formal tools that have been developed for describing these notions precisely. The course consists of three major components:

- *Dynamic semantics*—methods for describing and reasoning about what happens when a program runs.
- *Static semantics*—methods for reasoning about programs *before* they run. Such methods include type checking, type inference, and static analysis. We would like to find errors in programs as early as possible. By doing so, we can often detect errors that would otherwise show up only at runtime, perhaps after significant damage has already been done.
- *Language features*—applying the tools of dynamic and static semantics to study actual language features of interest, including some that you may not have encountered previously.

We want to be as precise as possible about these notions. Ideally, the more precisely we can describe the semantics of a programming language, the better equipped we will be to understand its power and limitations and to predict its behavior. Such understanding is essential not only for writing correct programs, but also for building tools like compilers, optimizers, and interpreters. Understanding the meaning of programs allows us to ascertain whether these tools are implemented correctly. But the very notion of *correctness* is subject to semantic interpretation.

It should be clear that the task before us is inherently mathematical. Initially, we will characterize the semantics of a program as a function that produces an output value based on some input value. Thus, we will start by presenting some mathematical tools for constructing and reasoning about functions.

1.1 Binary Relations and Functions

Denote by $A \times B$ the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$. A *binary relation* on $A \times B$ is just a subset $R \subseteq A \times B$. The sets A and B can be the same, but they do not have to be. The set A is called the *domain* and B the *codomain* (or *range*) of R. The smallest binary relation on $A \times B$ is the null relation $\varnothing$ consisting of no pairs, and the largest binary relation on $A \times B$ is $A \times B$ itself. The *identity relation* on A is $\{(a, a) \mid a \in A\} \subseteq A \times A$.

An important operation on binary relations is *relational composition*

$$R \,;\, S = \{(a, c) \mid \exists b \; (a, b) \in R \wedge (b, c) \in S\},$$

where the codomain of R is the same as the domain of S.

A (*total*) *function* (or *map*) is a binary relation $f \subseteq A \times B$ in which each element of A is associated with exactly one element of B. If f is such a function, we write:

$$f : A \to B$$

In other words, a function $f : A \to B$ is a binary relation $f \subseteq A \times B$ such that for each element $a \in A$, there is exactly one pair $(a, b) \in f$ with first component a. There can be more than one element of A associated with the same element of B, and not all elements of B need be associated with an element of A.

The set A is the *domain* and B is the *codomain* or *range* of f. The *image* of f is the set of elements in B that come from at least one element in A under f:

$$\begin{aligned} f(A) &\triangleq \{x \in B \mid x = f(a) \text{ for some } a \in A\} \\ &= \{f(a) \mid a \in A\}. \end{aligned}$$

The notation $f(A)$ is standard, albeit somewhat of an abuse.

The operation of *functional composition* is: if $f : A \to B$ and $g : B \to C$, then $g \circ f : A \to C$ is the function

$$(g \circ f)(x) = g(f(x)).$$

Viewing functions as a special case of binary relations, functional composition is the same as relational composition, but the order is reversed in the notation: $g \circ f = f \,;\, g$.

A *partial function* $f : A \rightharpoonup B$ (note the shape of the arrow) is a function $f : A' \to B$ defined on some subset $A' \subseteq A$. The notation $\mathsf{dom}\, f$ refers to A', the domain of f. If $f : A \to B$ is total, then $\mathsf{dom}\, f = A$.

A function $f : A \to B$ is said to be *one-to-one* (or *injective*) if $a \neq b$ implies $f(a) \neq f(b)$ and *onto* (or *surjective*) if every $b \in B$ is $f(a)$ for some $a \in A$.

1.2 Representation of Functions

Mathematically, a function is equal to its *extension*, which is the set of all its (input, output) pairs. One way to describe a function is to describe its extension directly, usually by specifying some mathematical relationship between the inputs and outputs. This is called an *extensional* representation. Another way is to give an *intensional*[1] representation, which is essentially a program or evaluation procedure to compute the output corresponding to a given input. The main differences are

- there can be more than one intensional representation of the same function, but there is only one extension;
- intensional representations typically give a method for computing the output from a given input, whereas extensional representations need not concern themselves with computation (and seldom do).

A central issue in semantics—and a good part of this course—is concerned with how to go from an intensional representation to a corresponding extensional representation.

2 The λ-Calculus

The λ-calculus (λ = "lambda", the Greek letter l)[2] was introduced by Alonzo Church (1903–1995) and Stephen Cole Kleene (1909–1994) in the 1930s to study the interaction of *functional abstraction* and *functional application*. The λ-calculus provides a succinct and unambiguous notation for the *intensional* representation of functions, as well as a general mechanism based on substitution for evaluating them.

The λ-calculus forms the theoretical foundation of all modern functional programming languages, including Lisp, Scheme, Haskell, OCaml, and Standard ML. One cannot understand the semantics of these languages without a thorough understanding of the λ-calculus.

It is common to use λ-notation in conjunction with other operators and values in some domain (e.g. $\lambda x.\, x{+}2$), but the *pure* λ-calculus has only λ-terms and only the operators of functional abstraction and functional application, nothing else. In the pure λ-calculus, λ-terms act as functions that take other λ-terms as input and produce λ-terms as output. Nevertheless, it is possible to code common data structures such as Booleans, integers, lists, and trees as λ-terms. The λ-calculus is computationally powerful enough to represent and compute any computable function over these data structures. It is thus equivalent to Turing machines in computational power.

2.1 Syntax

The following is the syntax of the *pure untyped* λ-calculus. Here *pure* means there are no constructs other than λ-terms, and *untyped* means that there are no restrictions on how λ-terms can be combined to form other λ-terms; every well-formed λ-term is considered meaningful.

A λ-term is defined inductively as follows. Let *Var* be a countably infinite set of variables $x, y, \ldots$.

- Any variable $x \in \mathit{Var}$ is a λ-term.

[1] Note the spelling: *intensional* and *intentional* are not the same!

[2] Why λ? To distinguish the bound variables from the unbound (free) variables, Church placed a caret on top of the bound variables, thus $\lambda x.\, x + yx^2$ was represented as $\widehat{x}\,.\, x + yx^2$. Apparently, the printers could not handle the caret, so it moved to the front and became a λ.

- If e is a λ-term, then so is $\lambda x.\, e$ (functional abstraction).
- If e_1 and e_2 are λ-terms, then so is $e_1 \cdot e_2$ (functional application).

We usually omit the application operator $\cdot$, writing $e_1\,(e_2)$, $(e_1\,e_2)$, or even $e_1\,e_2$ for $e_1 \cdot e_2$. Intuitively, this term represents the result of applying of e_1 as a function to e_2 as its input. The term $\lambda x.\, e$ represents a function with input parameter x and body e.

2.2 Examples

A term representing the identity function is $\mathsf{id} = \lambda x.\, x$. The term $\lambda x.\, \lambda a.\, a$ represents a function that ignores its argument and return the identity function. This is the same as $\lambda x.\, \mathsf{id}$.

The term $\lambda f.\, fa$ represents a function that takes another function f as an argument and applies it to a. Thus we can define functions that can take other functions as arguments and return functions as results; that is, functions are *first-class values*. The term $\lambda v.\, \lambda f.\, fv$ represents a function that takes an argument v and returns a function $\lambda f.\, fv$ that calls its argument—some function f—on v. A function that takes a pair of functions f and g and returns their composition $g \circ f$ is represented by $\lambda f.\, \lambda g.\, \lambda x.\, g(fx)$. We could *define* the composition operator this way.

In the pure λ-calculus, every λ-term represents a function, since any λ-term can appear on the left-hand side of an application operator.

2.3 BNF Notation

Backus–Naur form (BNF) is a kind of grammar used to specify the syntax of programming languages. It is named for John Backus (1924–2007), the inventor of Fortran, and Peter Naur (1928–), the inventor of Algol 60.

We can express the syntax of the pure untyped λ-calculus very concisely using BNF notation:

$$e \quad ::= \quad x \;\mid\; e_1\,e_2 \;\mid\; \lambda x.\, e$$

Here the e is a *metavariable* representing a *syntactic class* (in this case λ-terms) in the language. It is not a variable at the level of the programming language. We use subscripts to differentiate metavariables of the same syntactic class. In this definition, e_0, e_1 and e all represent λ-terms.

The pure untyped λ-calculus has only two syntactic classes, variables and λ-terms, but we shall soon see other more complicated BNF definitions.

2.4 Other Domains

The λ-calculus can be used in conjunction with other domains of primitive values and operations on them. Some popular choices are the *natural numbers* $\mathbb{N} = \{0, 1, 2, \ldots\}$ and *integers* $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ along with the basic arithmetic operations $+, \cdot$ and tests $=, \leq, <$; and the two-element Boolean algebra $\mathbb{2} = \{\mathit{false}, \mathit{true}\}$ along with the basic Boolean operations $\wedge$ (and), $\vee$ (or), $\neg$ (not), and $\Rightarrow$ (implication).[3]

[3]The German mathematician Leopold Kronecker (1823–1891) was fond of saying, "God created the natural numbers; all else is the work of Man." Actually, there is not much evidence that God created $\mathbb{N}$. But for $\mathbb{2}$, there is no question:

> *And the earth was without form, and void... And God said, Let there be light... And God divided the light from the darkness...* —Genesis 1 : 2–4

The λ-calculus gives a convenient notation for describing functions built from these objects. We can incorporate them in the language by assuming there is a constant for each primitive value and each distinguished operation, and extending the definition of *term* accordingly. This allows us to write expressions like $\lambda x.\, x^2$ for the squaring function on integers.

In mathematics, it is common to define a function f by describing its value $f(x)$ on a typical input x. For example, one might specify the squaring function on integers by writing $f(x) = x^2$, or anonymously by $x \mapsto x^2$. Using λ-notation, we would write $f = \lambda x.\, x^2$ or $\lambda x.\, x^2$, respectively.

2.5 Abstract Syntax and Parsing Conventions

The BNF definition above actually defines the *abstract syntax* of the λ-calculus; that is, we consider a term to be already parsed into its *abstract syntax tree*. In the text, however, we are constrained to use sequences of symbols to represent terms, so we need some conventions to be sure that they are read unambiguously.

We use parentheses to show explicitly how to parse expressions, but we also assign a precedence to the operators in order to save parentheses. Conventionally, function application is higher precedence (binds tighter) than λ-abstraction; thus $\lambda x.\, x\, \lambda y.\, y$ should be read as $\lambda x.\, (x\, \lambda y.\, y)$, not $(\lambda x.\, x)\, (\lambda y.\, y)$. If you want the latter, you must use explicit parentheses.

Another way to view this is that the body of a λ-abstraction $\lambda x.\ \ldots$ extends as far to the right as it can—it is *greedy*. Thus the body is delimited on the right only by a right parenthesis whose matching left parenthesis is to the left of the λx, or by the end of the entire expression.

Another convention is that application is *left-associative*, which means that $e_1\, e_2\, e_3$ should be read as $(e_1\, e_2)\, e_3$. If you want $e_1\, (e_2\, e_3)$, you must use parentheses.

It never hurts to include parentheses if you are not sure.

2.6 Terms and Types

Typically, programming languages have two different kinds of expressions: *terms* and *types*. We have not talked about types yet, but we will soon. A *term* is an expression representing a value; a *type* is an expression representing a class of similar values.

The value represented by a term is determined at runtime by evaluating the term; its value at compile time may not be known. Indeed, it may not even have a value if the evaluation does not terminate. On the other hand, types can be determined at compile time and are used by the compiler to rule out ill-formed terms. When we say a given term has a given type (for example, $\lambda x.x^2$ has type $\mathbb{Z} \to \mathbb{Z}$), we are saying that the value of the term after evaluation at runtime, if it exists, will be a member of the class of similar values represented by the type.

In the pure untyped λ-calculus, there are no types, and all terms are meaningful.

2.7 Multi-Argument Functions and Currying

We would like to allow multiple arguments to a function, as for example in $(\lambda(x, y).\, x + y)\, (5, 2)$. However, we do not need to introduce a primitive pairing operator to do this. Instead, we can write $(\lambda x.\, \lambda y.\, x + y)\, 5\, 2$. That is, instead of the function taking two arguments and adding them, the function takes only the first argument and returns a function that takes the second argument and then adds the two arguments. The

notation $\lambda x_1 \ldots x_n . e$ is considered an abbreviation for $\lambda x_1 . \lambda x_2 . \lambda x_3 . \ldots \lambda x_n . e$. Thus we consider the multi-argument version of the λ-calculus as just syntactic sugar. The "desugaring" transformation

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

for this particular form of sugar is called *currying* after Haskell B. Curry (1900–1982).

3 Preview

Next time we will discuss capture-avoiding (safe) substitution and the computational rules of the λ-calculus, namely α-, β-, and η-reduction. This is the *calculus* part of the λ-calculus.

References

[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 2nd edition, 1984.