A. Review of Mathematical Preliminaries

- Sets\(^1\) - explicit representation, construction, empty set, power set, set difference \((S \setminus T)\)
- Relation - n-ary relation
- Predicate - relation on \(S \times \{true, false\}\)
- Domain, range of a relation \(R - \text{dom}(R), \text{range}(R)\)
- Partial function - \(R \subseteq S \times T\) is a partial function if, whenever \(s \in S, t_1, t_2 \in T, (s, t_1) \in R \land (s, t_2) \in R\), we have \(t_1 = t_2\)
- Total function - \(R \subseteq S \times T\) is total function if, \(\text{dom}(R) = S\)
- Function \(f\) undefined for \(x - f(x) \uparrow\)
- Function \(f\) defined for \(x - f(x) \downarrow\)
- Divergence vs. explicit failure
- For \(R \subseteq S \times S\) and \(P \subseteq S \times \{true, false\}\), \(P\) is preserved by \(R\) if whenever \(sRs' \land P(s)\) for \(s, s' \in S, P(s')\).
- Reflexive Relations: we say a relation \(R \subseteq S \times S\) is reflexive, if \(\forall s \in S, sR_s\).
- Symmetric Relations: we say a relation \(R \subseteq S \times S\) is symmetric, if \(\forall s, t \in S, sRt\) implies \(tRs\).
- Transitive Relations: we say a relation \(R \subseteq S \times S\) is transitive, if \(\forall p, s, t \in S, pRs\) and \(sRt\) implies \(pRt\).
- Antisymmetric Relations: we say a relation \(R \subseteq S \times S\) is transitive, if \(\forall s, t \in S, sRt\) and \(tRs\) implies \(s = t\).
- Set Ordering
  - A set \(S\) is preordered w.r.t. \(R\), if \(R\) is reflexive and transitive
  - \(R\) is also called a preorder on \(S\), also written as \(\leq\) or \(\sqsubseteq\)
  - \(R\) is partial order, if \(R\) is preorder and antisymmetric
  - \(R\) is total order, if \(R\) is partial order and \(\forall s, t \in S, s \leq t \lor t \leq s\)
  - Least upper bound (\textit{join})
  - Greatest lower bound (\textit{meet})
  - Equivalence - reflexive, symmetric, and transitive
  - Reflexive and transitive closures (written \(R^+\))

\(^1\)What is Russell’s paradox and how does axiomatic set theory works around this paradox?
B. Lambda Calculus

Lambda calculus was originally invented by Alonzo Church in 1920s. Peter Landin’s work in 1960s first used this calculus for formalizing semantics of programming languages. Since then it has formed the basis for the design and implementation of many programming languages such as Lisp, Scheme, ML, Haskell, etc. The key advantage of lambda calculus is that it is small and simple, but powerful, which makes it a suitable target language for expressing language features.

Core Calculus. A small language design and associated formal system that is necessary and sufficient to capture the essential mechanisms of a language design for the sake of studying them rigorously. Lambda calculus is an example of core calculus. Other examples are Pi calculus [4] for message-based concurrent languages, Abadi and Cardelli’s object calculus [1] for object-oriented languages, Abadi and Gordon’s Spi calculus [2] for cryptographic features in programming languages and communication protocols, etc.

Exercise B.1 Why is it beneficial to work with a core calculus instead of a full-fledged language design?

Syntactic Sugar. Convenient derived forms whose behavior can be understood by translating them into the core calculus.

Exercise B.2 Assume that we have rigorously analyzed the behavior of ‘+’ (addition over integers) and ‘=’ (assignment) in a programming language, can we understand the behavior of a convenient syntactic form ‘++’ in terms of these?

Exercise B.3 Why do we add syntactic sugars to a programming language?

Exercise B.4 Does adding syntactic sugars to a programming language makes it more expressive (informally)?

Procedural Abstraction. Also known as abstraction by parameterization. According to Liskov and Guttag, “abstraction by parameterization allows us, through the introduction of parameters, to represent a potentially infinite set of different computations with a single program text that is an abstraction of all of them [3].” When we say \( f(x, y) = x \times x + y \times y \), we are describing the set of computations that adds the square of \( x \) to the square of \( y \) for all integer values of \( x \) and \( y \). Thus, this is an abstract representation of the set of computations \( \{(1 \times 1 + 1 \times 1), (1 \times 1 + 2 \times 2), \ldots\} \), i.e. an abstraction of all computations where \( x \) and \( y \) ranges over \( \mathbb{N} \).
**Difference between Abstract and Concrete Syntax.** Concrete (or surface) syntax is what programmers write, whereas abstract syntax is a much simpler internal representation of programs as labeled trees (called abstract syntax trees or ASTs). In abstract syntax conventions such as operator precedence, associativity, etc, are not important. For example, an expression like $1 + 2 * 3$ is the same as $1 + (2 * 3)$ in abstract syntax. The parentheses are there just to make it clear what the parse tree is. Precedence, associativity, etc, play an important role in operations on concrete syntax such as parsing. We will not discuss parsing and lexical analysis in detail here. Those are the subject of another course Com S 440/540. In this class we will only be concerned about abstract syntax.

**B.1 Abstract Syntax**

The abstract syntax of the pure lambda calculus is shown below:

\[
\begin{align*}
  e & ::= \text{"Terms"} \\
    & | x \quad \text{"Variable"} \\
    & | \lambda x . e \quad \text{"Abstraction"} \\
    & | e e \quad \text{"Application"}
\end{align*}
\]

The abstraction rule is the most important as it allows us to define anonymous functions that provide procedural abstraction. In this rule $x$ is the argument and $e$ is the body of the function. In the application rule $e_1 e_2$, $e_1$ is the function and $e_2$ is the argument to the function. Computation can only take place using application rule that applies functions to arguments. These arguments can also be functions.

Note that this calculus does not provide any build-in constants (such as $0$, $true$, $false$), primitive operators (such as $+$, $-$), numbers, arithmetic operations, conditionals, records, loops, sequencing, etc.

**B.2 Example Programs**

**Identity function.** The mathematical representation of an identity function would be $f(x) = x$, an identity function in lambda calculus also has a very similar syntax, except that the function declared by the lambda expression is anonymous.

\[ \lambda x . x \]

**Application of identity function.** From the abstract syntax shown above application has the syntactic form $e e$. Here the first term is the function and second term is the argument to the function. If we consider, the function to be the identity function and argument to be some term $y$, application would be:

\[ \lambda x . x \ y \]

We have not defined the domain of $x, y$ yet.

**Function evaluating to identity function.** We mentioned earlier that the arguments can be functions and the terms in lambda calculus can themselves evaluate as a function. Here is an example:

\[ \lambda x . \lambda y . y \]

For any argument, this term evaluates to the identity function.

**Exercise B.5** What does $(\lambda x . \lambda y . y) \ a$ evaluates to?
B.3 Variable Scoping

Bound Variables. An occurrence of the variable $x$ is said to be bound when it occurs in the body $t$ of an abstraction $\lambda x.t$. Here the scope of the binder $\lambda x$ is the body $t$. For example, $x$ is bound in $\lambda x.x$.

Free Variables. An occurrence of the variable $x$ is said to be free when it is not bound by the enclosing abstraction. For example, $x$ is free in $\lambda y.x$.

Closed Terms. A term with no free variable is called a closed term. For example, $\lambda x.x$ is a closed term, whereas $\lambda y.x$ is not a closed term. A closed term is also called a combinator.

B.4 Multi-argument Functions

Core calculus of lambda calculus only provides for single argument functions. How can we build multiple argument functions from this basic definition of abstraction that we have available. Suppose we want to write a function that takes two arguments $x$ and $y$, and applies the first argument $x$ on the second argument. We can define such function as $\lambda x.\lambda y.x y$. Following shows the successive steps of evaluation for two terms $a$ and $b$.

$$(\lambda x.\lambda y.x y) \ a \ b$$
$$> (\lambda y.a \ y) \ b$$
$$> (a \ b)$$

This is an example of higher-order function that yield other functions as result. The transformation of multi-argument functions into higher-order functions is called currying.

References


