Lecture Notes 1
Introduction

Carlo Angiuli

B522: PL Foundations
January 12, 2026

This course is an introduction to the *theoretical foundations of programming languages*. In our first lecture, besides going through the basic mechanics of this course (as outlined in the syllabus), I want to spend a significant amount of time explaining what I mean by *the theoretical foundations of programming languages*, by describing some basic problems that motivate the definitions and theorems that we will encounter in this course.

Remark 1.1. This lecture includes some snippets of Racket code, but in general I do not assume familiarity with any specific programming language(s).

1 Central Problem 1: Defining PLs

By this point in your CS career, you have used at least a few different programming languages. In C311/B521, you've played around with a small Racket-like (or perhaps Haskell-like) programming language, writing multiple interpreters for it and adding features to it one by one. If you've taken P423/P523, you've even written a compiler that takes Racket-like programs as input and turns them into executables that you can run directly on your computer.

But what exactly *is* the Racket programming language? What *is* a Racket program? And what is the *meaning* of a Racket program? What about Java, or JavaScript? If you think about this question, you might say that a Racket program is *a text file that DrRacket agrees to run when you hit the Run button*, and its meaning is *whatever DrRacket prints out.* I find these answers rather intellectually unsatisfying, and on further inspection, they actually raise more questions than they answer:

> What version? (What about `racket` on the command line?) On whose computer? (Do I need to understand my operating system and hardware, too?) Does this mean it's impossible for DrRacket to have bugs,

because its behavior is by definition correct? Wait, what language is DrRacket written in, again?

Some programming languages, particularly older languages with multiple implementations, have standards committees that get together and write a document in English carefully explaining what the language means. The C23 standard (ISO/IEC 9899:2024) is a 758-page PDF that costs 227 Swiss francs to download. In theory this allows us to compare implementations to an agreed-upon reference, but is such a standards document really better than defining a language in terms of a freely-available implementation? (Maybe?)

C311/B521-style interpreters are more concise than the whole implementation of Racket and more precise than a text document, but they still reduce the meaning of the interpreted programming language to the meaning of a particular Racket program and hence to the meaning of Racket itself. And this isn't just an idle worry: the parameter-passing convention implemented by the standard `value-of` function actually relies on the parameter-passing convention used by Racket!

In this class We will learn how to view programming languages as mathematical objects. Throughout the semester we will precisely define and analyze a series of simple but representative programming languages, considering in each case the abstract *syntax* of its programs, as well as those programs' *semantics*.[1]

Abstract syntax is a mathematical representation of the structure of programs, divorced from implementation details such as whitespace, Unicode, precedence, and other practical issues related to inputting programs into a computer. (To distinguish abstract syntax from "actual syntax," we will refer to the latter as *concrete syntax*.) We will define abstract syntax in terms of *abstract syntax trees* (ASTs), *abstract binding trees* (ABTs), and *inference rules*.

We will study the semantics, or meaning, of programs primarily via a family of techniques known as *operational semantics*, including what are called *structural operational semantics*, *context-sensitive reduction semantics*, *natural semantics*, and *abstract machines*. All of these abstractly capture some notion of "program execution." We may also briefly discuss two other families of techniques for assigning meaning to programs: *denotational semantics*, which capture the meaning of programs via mathematical functions, and *axiomatic semantics*, which capture the meaning of programs via logical formulas. In each case we strive for a *compositional* understanding of program syntax and semantics, in which the meaning of a program is determined by the meanings of its constituent parts.

[1] Harper [Har16] refers to these as *statics* and *dynamics* respectively.

2 Central Problem 2: Going right and going wrong

Once we have a precise mathematical definition of a programming language, we can start asking precise mathematical questions about it. One of the first questions we might ask when trying to get a handle on a programming language is, "What are all the possible results that programs can produce?"

In Racket, `2` is both a valid program and a valid program result, or *value*. `(+ 1 1)` is a program but not a value: it is not fully simplified.[2] `+` is both a program and a value in Racket, but in many other languages it is neither. In some languages, `2` is not even a program (e.g., because programs must contain a `main` function) but it is still a value.

But `2` and `"hello"` and `+` are all "normal" values that a program might return when it succeeds. What about programs where something goes wrong?

```
... contract violation ...

... segmentation fault (core dumped) ...

... java.lang.ArrayIndexOutOfBoundsException ...
```

Just as different programming languages have very different values, they also have very different errors. Some errors are recoverable; others are not. Some errors, like dereferencing a null pointer, are unrecoverable in some languages and recoverable in others. Some errors are not even *possible* in certain languages: running `(+ 1 "hello")` produces a `contract violation` in Racket, but Typed Racket rejects it as a `type mismatch` before even attempting to run it.

In this class One of the major themes in programming language theory is to understand exactly what program behaviors, both "good" and "bad," are possible in a given programming language. Throughout the semester, we will define a series of languages with increasingly complex *type systems*. In each case, we will analyze their syntax and semantics in tandem to mathematically prove that certain behaviors (e.g., attempting to add non-numbers) are ruled out by the type system.

We will learn how to perform rule induction on the syntax and semantics of programming languages to establish *type safety*, a central concept in programming language theory often summarized as "Well-typed programs cannot 'go wrong.'" The standard technique for establishing type safety decomposes it into several smaller lemmas known as *progress*, *preservation*, and the *canonical forms lemma*.

[2] Not to be confused, of course, with `'(+ 1 1)`, which is both a program and a value.

Later on in the semester, we will learn a much more powerful proof technique known as the technique of *logical relations*, which will allow us to precisely characterize the program behaviors that are possible at every given type. For example, we will consider several typed λ-calculi for which logical relations enable us to prove that all programs *terminate*.

3 Central Problem 3: Program fragment equivalence

The third problem is the one I find most interesting, in part because it requires the deepest understanding of our language yet: "When are two pieces of code equal?"

Every time we refactor or optimize a program, we have some intuitive notion of which maneuvers are meaning-preserving: for example, we might simplify `(if x true false)` to `x`. A disciplined programmer might even have a test suite to help catch regressions as they refactor. But how can we be sure that our program has not changed meaning? How many tests are enough? And what about compiler implementers who dream up a new optimization to apply to *every* program?

We say that two pieces of code are *observationally equivalent* if replacing one with the other never changes the result of a surrounding program. Observational equivalence is a very strong property and is highly sensitive to details of the programming language in question. To pick an extreme example, Python code can access the names of functions as strings via `.__name__`, so even two differently-named functions with identical definitions are inequivalent in Python.[3]

What about Racket? Is `(* 0 e)` always equivalent to `0`? (Think about it for a moment.) No, because the former produces an error when e = `"hello"`. But even if we rule out errors the answer is still "no"—think about it again for a moment—in cases where e is an infinite loop, or prints to the console, or `set!`s something. Suppose for example that e = `(begin (set! x 2) 0)`. Then

```
(let ([x 1]) (+ (* 0 e) x))          = 2
(let ([x 1]) (+ 0 x))                = 1
```

Exercise 1.2. Is `(`f `e e)` always equivalent to `(let ([x e]) (`f `x x))`?

Remark 1.3. Disproving observational equivalence is as simple as finding a single surrounding program that distinguishes two pieces of code, but *proving* observational equivalence naïvely requires checking every possible surrounding program, an impossible task.

[3] In fairness, most "real" programming languages have at least one mechanism—reflection, timers, etc.—capable of distinguishing virtually all non-identical programs.

Type systems often have a profound impact on observational equivalence: by restricting what surrounding programs are valid, they limit the number of ways in which two pieces of code can be distinguished. For example, because Typed Racket rejects `(* 0 "hello")` as a `type mismatch`, it has fewer situations than Racket in which `(* 0 e)` and `0` behave differently.

The most important example of this phenomenon is that of type-based mechanisms for *abstraction*. There are many ways to concretely represent ordered collections of numbers, including linked lists, trees, `ArrayLists`, etc. These collections and their operations are clearly observationally inequivalent—one is a list and the other isn't—but if these implementation details are hidden behind an abstract type of "`Lists`" or "`Sequences`," we can in fact prove that no two programs can distinguish these data representations through the `List` interface. This property is known as *representation independence*.

> "Type structure is a syntactic discipline for enforcing levels of abstraction." —John C. Reynolds

In this class We will precisely define *observational equivalence* (or *contextual equivalence*) of program fragments for a number of programming languages. Using logical relations, we will prove that observational equivalence coincides with a more tractable, type-directed relation known as *logical equivalence*, and use this to establish some interesting observational equivalences.

As an important application of this technique, we will study the core calculus known as System F whose *polymorphic types* and *existential types* encode abstract interfaces. Using logical relations, we will prove a *parametricity theorem* characterizing observational equivalence in System F, and derive some important consequences known as *free theorems* and *representation independence*.

4 Course objectives

By the end of the course, students will be able to:

- formally define a variety of programming languages by their statics (type systems) and dynamics (operational semantics),
- prove theorems relating a language's statics and dynamics,
- formally define and reason about equivalence of program fragments, and
- appreciate the importance of compositionality and abstraction in the design of programming languages.

My goal in this course is to provide students with a basic foundation in the theory of programming languages. For some of you this may be your only foray into this area; others may pursue programming languages research at Indiana University or elsewhere. In either case, I hope that this class deepens your appreciation of the subtleties of programming language design.

4.1 Regarding actual programming languages

In this lecture we have mentioned some "real" programming languages[4] as motivation. However, this course will not focus on such languages but instead on what are called *core calculi*: idealized programming languages designed to exhibit a small number of features rather than to serve as a realistic general-purpose language.

In the long run, researchers aim to understand existing "real" languages by developing core calculi embodying their most important features. Conversely, understanding core calculi and their properties can help us ensure that the next generation of "real" languages are well-behaved in a variety of precise senses.

> "Programming language semanticists should be the obstetricians of programming languages, not their coroners." —John C. Reynolds

In particular, it is not the goal of this course to taxonomize or evaluate "real" programming languages, although I (and, I daresay, my colleagues) believe that the theoretical properties we will study in this class make for well-behaved programming languages in practice. That said, there are certainly many aspects of programming languages that are very important in practice but quite orthogonal to this class, including good editor support, large libraries, friendly online communities, well-named keywords, fast compilers, etc.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[4] That is, a general-purpose language that one might reasonably choose for practical tasks.