Goals of the Course  This course is designed to teach the elements of a mathematically rigorous semantics for programming languages.\textsuperscript{1} Based on this semantics it is possible to prove that certain programming languages behave according to precise mathematical rules if they are implemented properly. It is also possible to prove that compilers implementing these languages are correct \textsuperscript{[13]} according to their mathematical semantics. On this basis, computer scientists can now prove that programs have predictable behavior and precise mathematical properties. The level of these languages ranges from C to Haskell and ML, but not all programming languages have a mathematical semantics; in this course we focus on those that do.

Mathematical Semantics  One of the main reasons that the topics covered in this course are considered valuable in graduate computer science education is that the material is a model for how we create software artifacts such as programming languages that are mathematical objects as well as useful tools.\textsuperscript{2} Computer science shares with physics this property that some elements of the subject are both useful tools for thought, such as physical theories, and mathematical objects as well. Computer science is also in the remarkable position that computer programs are important in exploring physical theories. This is especially true for programs that have precise meaning. These aspects of computer science explain why it is one of the most mathematical of all the sciences.

Proof Technology  Over the past few decades, programming languages and other software systems have become so essential in modern life and in life-critical systems that tremendous effort has gone into making them more trustworthy and reliable. Part of that effort has focused on proving properties of languages and programs as rigorously as possible. This

\textsuperscript{1}This semantic theory is covered in Volume B of the *Handbook of Theoretical Computer Science* [11, 8, 17], and is thus sometimes referred to as Theory B.

\textsuperscript{2}What kind of mathematical object is a programming language? The closest approximation is probably a formal mathematical theory fragment; that is a theory which does not make explicit all of its axioms, just a fragment of them. When the axioms are given, we have a programming logic. Early examples of these were the Stanford Pascal Verifier [10], Gypsy, and Cornell’s PLCV [6]. Conversely, a formal mathematical theory with computational semantics is a programming language.
effort has resulted in a new technology that I call proof technology; it allows an unprecedented level of precision and rigor in establishing mathematical facts, both about pure mathematical objects and about software systems and the hardware on which they execute.

**Scope of the Course** In the Programming Logic Laboratory, CS6116, will examine aspects of proof technology as it relates to programming languages. Moreover, because this technology is now moving into the construction of operating systems, such as seL4 [7], and distributed systems, such as Ensemble [26, 14] and Birman’s Isis2 [3], this course will include the semantics of asynchronous message passing computation, i.e. distributed systems. In this relatively new subarea we will be able to point out open research questions that show how rigorous concepts are migrating from programming languages to concurrent systems. Eventually proof technology might move into applications such as symbolic algebra systems and other domain specific languages as illustrated by Theorema project [4].

**Formal Semantics** Proof technology has been so successful in reasoning about programming languages that it is now possible to formally prove many of the theorems normally taught in advanced programming language graduate courses like this one. A formal proof is one that can be checked in every detail. They achieve the highest levels of certainty known using technology. They are produced by software tools called interactive theorem provers or proof assistants, or provers for short. Using these provers it is possible to create a formal semantics for some programming languages. Interestingly the main tools for accomplishing this are based on constructive mathematics formalized in type theory. We will see natural reasons for this approach as the course progresses, and we will use constructive methods from the start so that we can more quickly reach the formal semantics.

**Programming Logic Laboratory** A pioneering enterprize to engage students in reading and creating formal proofs about programming languages is described in the book *Software Foundations* by Benjamin Pierce and his collaborators at the University of Pennsylvania [20] in which they use the Coq proof assistant to formally prove several of the results taught in this course and which appear in the basic books on programming languages recommended for this course [19, 17, 27, 24, 25] and in some of the fundamental papers such as those of Gordon Plotkin [21, 22, 23]. This creates a formal semantics for the concepts in a formal theory called constructive type theory. What is especially gratifying to me and my Cornell collaborators is that the formal theory, *Calculus of Inductive Constructions* (CIC) [18], used by the Coq prover [2] to produce these results is based to a large extent on results from the PRL group who designed Computational Type Theory (CTT) [5, 1] and implemented it in the Nuprl [5, 12] and MetaPRL [9] theorem provers. These are constructive type theories closely related to the ground breaking *Intuitionistic Type Theory* (ITT) of Per Martin-Löf [15, 16]. CTT and Nuprl are used to produce correct by construction software, and they could be brought to bear in this course to prove many of the theorems about semantics. You will learn a bit about these methods in the Programming Logic Lab which meets on Fridays.
References


