Advanced Programming Languages CS6110
Spring 2012
Lecture 1

Robert L. Constable
Cornell University

Goals of the Course This course is designed to teach the elements of a mathematically rigorous semantics for programming languages.[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 [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.[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

[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*.

[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

[1] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] Kenneth P. Birman. *Guide to Reliable Distributed Systems.* Springer-Verlag, 2012.

[4] Bruno Buchberger. Theorema: Extending mathematica by automated proving (invited talk). In D. Bosanac and D. Ungar, editors, *Proceedings of the Programming System Mathematica in Science, Technology, and Education (PrimMath 2001)*, pages 10–11, Electrotechnical and Computer Science Faculty, University of Zagreb, September 2001.

[5] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, NJ, 1986.

[6] Robert L. Constable and Michael J. O'Donnell. *A Programming Logic.* Winthrop, Mass., 1978.

[7] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the sel4 microkernel. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 99–114, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674. North-Holland, 1990.

[9] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[10] S. Igarashi, R. London, and D. Luckham. Automatic program verification I: a logical basis and its implementation. *Acta Informatica*, 4:145–82, 1975.

[11] Dexter C. Kozen and Jerzy Tiuryn. Logics of programs. In van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 789–840. North Holland, Amsterdam, 1990.

[12] Christoph Kreitz. The Nuprl Proof Development System, version 5, Reference Manual and User's Guide. Cornell University, Ithaca, NY, 2002. `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`.

[13] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122. ACM Press, 1994.

[14] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In David Kotz and John Wilkes, editors, *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[15] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[16] Per Martin-Löf. *Intuitionistic Type Theory.* Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[17] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, chapter 8, pages 366–458. Elsevier Science Publishers, North-Holland, 1990.

[18] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In J. F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[19] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[20] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations.* Electronic, 2011.

[21] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Journal of Theoretical Computer Science*, pages 125–59, 1975.

[22] Gordon D. Plotkin. LCF considered as a programming language. *Journal of Theoretical Computer Science*, 5:223–255, 1977.

[23] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Department, Denmark, 1981.

[24] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[25] R. Turner. *Constructive Foundations for Functional Languages.* McGraw Hill, New York, 1991.

[26] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburg, and David Karr. Building adaptive systems using Ensemble. *Software: Practice and Experience*, 28(9):963–979, July 1998.

[27] G. Winskel. *Formal Semantics of Programming Languages.* MIT Press, Cambridge, 1993.