

Position Paper: Practical Foundations for Programming Languages

Robert Harper

Draft of May 28, 2008

1 Introduction

A program is a form of expression that conveys an idea, much as does a proof in mathematics. But whereas proofs have only a *static* significance, as a form of communication, programs also have a *dynamic* significance, as commands to be executed by a machine. This places very strong demands on the programmer to explain herself so precisely and rigorously that a program can be run on a computer, while at the same time so crisply and elegantly that a human would be able and willing to understand her.

Programming, therefore, is intrinsically about the *language* we use to write programs and to express the reasoning that gives rise to them. As with natural languages, a programming language is a social construct created by and for people, not just for computers. Unlike natural languages, programming languages must, to be meaningful both to people and machines, must meet a very high standard of rigor unmatched in other fields, even mathematics.

Such high standards can only be met by through a thorough understanding of the rules of the game, the possible moves in the space of computation. What we can *do* is limited by what we can *say*: in programming language truly does constrain action. The study of programming languages is therefore a study of what is computable in the broadest sense, namely what can be programmed? The study of programming languages is the science of computation itself, and as such lies at the heart of the discipline of Computer Science.

The purpose of a course in Programming Languages (PL) in an undergraduate curriculum is to introduce the scientific theory of the computable. Undergraduate PL is not taxonomy, the mere description of languages as they are found in the wild (the “wildebeests” and the “walruses”), nor is it cladistics, the classification of languages based on their heritage or morphology (“paradigms” or “orientations”). The purpose of PL is to *analyze* and *codify* core computational concepts that arise in all languages, and to *synthesize* these concepts in the construction of tools and systems for program development. Through analysis we discover *what computation consists of* and through synthesis we learn *how to compute useful things*. To draw a rough analogy with undergraduate

Chemistry, the role of PL is to study the periodic table of elements, the fundamental ingredients, and to discuss how these may be combined to form useful substances.

A skeptic may wonder whether there are any core concepts in computing. Is Computer Science a science, or a craft? Is there a scientific theory of programming? Or is programming a collection of methods and practices that are passed on as a kind of folk art? Less contentiously, can we separate core concepts from the context in which they are utilized? Are there shared principles that can be isolated and codified, or are all software systems constructions unto themselves? There are two forms of answer to these questions, one empirical, the other conceptual.

Experience has shown that the same concepts arise repeatedly, and the same errors are repeated, in many different concepts. The fundamental concept of a variable is a good case in point. Few languages have managed to get variables right, and even the latest fad languages continue to get it wrong. For example, the behavior of identifiers in languages such as `TeX` or the Unix shell (in its various incarnations) is appalling, and the errors are repeated in popular languages such as Perl or TCL. To take another example, the by-now well-established concept of pipes and filters in the Unix shell is simply a form of composition of functions over streams of bytes, concepts that transcend their use in the shell, yet which are absent in many languages. Modularity concepts, such as linking and loading, are found in many disguises, as DLL's in operating systems, or applets in web browsers, or reflection in Java. Dozens of “features” found in many software systems amount to a composition of a few core concepts. More broadly, programming concepts such as communicating processes or stochastic computation are being employed to model phenomena in the natural world, notably in systems biology, but also in user interface design, business processes, or securities markets. The core concepts of computation, which are codified in programming languages, are becoming increasingly important outside of the software industry.

Given their ubiquity, is there a conceptual framework in which one may present the core concepts of programming languages in a manner that avoids tying them too closely to their realizations in specific languages, admits rigorous analysis, informs implementation, and supports a pedagogical methodology for teaching this material? Over the last couple of decades there has emerged a standard framework for defining and analyzing programming languages that achieves all of these goals. This approach is developed in the author's forthcoming text, *Practical Foundations for Programming Languages*, which is described briefly in the next section.¹

2 Practical Foundations

Drawing on the author's experience with logical frameworks and programming language definition, the whole of *Practical Foundations* is based on the simple

¹The manuscript is available at <http://www.cs.cmu.edu/~rwh/plbook/book.pdf>.

concept of an *inductively defined judgement*. A judgement is just an assertion (predicate or relation) about one or more objects. An inductive definition of a judgement consists of a collection of *rules* that specify sufficient conditions for the judgement to hold. In one direction the rules tell us how to demonstrate that the judgement holds (simply compose the rules to construct a derivation), and in the other tell us how to reason from the fact that it holds (by an inductive analysis of the rules that define it).

This simple foundation is sufficient to build up a comprehensive theory of programming languages. We use inductive definitions to define classes of objects, functions on them, and relations between them. For example,

1. Syntactic objects such as strings and abstract syntax trees.
2. Functions over these objects, including parsers and unparsers.
3. An enriched notion of abstract syntax that includes a uniform treatment of the concepts of variable, binding, and scope.
4. Functions and relations on abstract syntax such as capture-avoiding substitution and equivalence up to renaming of bound variables.
5. Hypothetical and parametric judgements to express conditional and schematic assertions and rules.

This provides the building blocks from which the subsequent treatment of programming language features is constructed.

The central organizing principle of *Practical Foundations* is the thesis that *language features are manifestations of type structure*. Programming concepts are classified by types, which are specified by the operations that are used to *introduce*, or create, values of that type, and the mechanisms that are used to *eliminate*, or compute with, values of that type. The *static semantics* of a language is an inductive definition of its type structure, which is given by a *typing judgement*, $e : \tau$, specifying that an expression e has the type τ . The *dynamic semantics* is an inductive definition of an abstract machine for executing programs, which is given by a *transition judgement*, $e \mapsto e'$, specifying the steps of execution of programs. *Type safety* is simply the statement of coherence of the static and dynamic semantics.

This single framework may be used to account for a wealth of programming language concepts, including

1. Finite and infinite persistent data structures such as tuples, tagged values, and linked structures.
2. Functions and procedures acting on and transforming values of such data structures.
3. Dynamic typing and hybrid typing (combining static and dynamic typing).
4. Polymorphism (universals, or generics) and data abstraction (existentials).

5. Exceptions for to permit limited forms of non-local control transfer with associated data.
6. Continuations to permit unlimited forms of non-local control transfer, including re-entrancy and co-routines.
7. Subtyping to codify reuse of a value of one type as a value of another type.
8. Finite and infinite ephemeral data structures through mutation.
9. Isolation of computational effects using monads and co-monads.
10. Eager and lazy evaluation, including call-by-need, memoization, and futures.
11. Speculative and non-speculative deterministic parallelism. Cost semantics and work efficiency.
12. Non-deterministic concurrent composition.
13. Modularity and linking.
14. Logics for reasoning about programs.

This approach to teaching PL has been used at Carnegie Mellon for at least ten years, at both the undergraduate and graduate levels. The text, and substantially similar development, has been in use at Carnegie Mellon's Qatar campus for two years, and at various universities around the world, including Princeton, Yale, Harvard, Cornell, and UMass, at both undergraduate and graduate levels.

3 Thoughts and Recommendations

The biggest problem with PL in the undergraduate curriculum is that research advances have far outstripped conventional wisdom in the area. There are dozens of textbooks on programming languages, very few of which, if any, are based on the modern scientific theory of computation on which contemporary programming language research is based. The most popular texts reflect a 1970's view of the field, and have no serious theoretical component. This has given rise to the impression that PL is largely common sense, a body of folk wisdom built up from experience with a half dozen or so well-known languages.

One goal of *Practical Foundations* is to present a more modern, scientific view of programming languages. The major obstacle to achieving this goal is a classic Catch-22 situation: the reason teaching PL is stuck in the 1970's is because the teachers of PL are stuck in the 1970's. Major research universities have on their faculty professors who are expert in the area, but there are very few universities in the U.S. with faculty in this area. It should be emphasized that the obstacle is not the ability of the students: all that is required is facility with proof by induction, an essential tool for any student of Computer Science.

The real reason to teach PL in the style of *Practical Foundations*, however, is not because it is practical, but because it is beautiful. Students are consistently drawn to the area because of its elegance, its depth, and its scope of applicability. We should not reduce Computer Science to training in skills fashionable in the computer industry, but rather elevate it