Book reviews

Practical Foundations for Programming Languages, by Robert Harper, Cambridge University Press, New York, December 2012, English, ISBN-10: 1107029570, Hardcover, 487 pp.
doi: 10.1017/S095679681400015X

In this book, Robert Harper comprehensively surveys the concepts that underlie the design of programming languages. The book's organising principle is type theory, and it convincingly demonstrates the central role of types in language design.

The book consists of 50 (short!) chapters, grouped into 18 broad topics such as function types, finite data types (products and sums), infinite data types, variable types (including abstract types), subtyping, classes and methods, exceptions and continuations, state, laziness, parallelism, concurrency, and modularity. A typical chapter explains a particular concept in the context of a little language; presents the language's formal syntax, static, and dynamic semantics; and often proceeds to prove fundamental language properties such as type safety.

This approach illuminates deep issues in language design. One such issue is the expressiveness of polymorphic types ("theorems for free"). Another is the importance of distinguishing between mobile types (whose values can safely be passed out of scope) and immobile types (such as the type of scoped references).

The book's title is little misleading. The book is certainly about the foundations of programming languages – their syntax, their semantics, and properties derivable from the semantics. But it offers few insights into the pragmatics of these languages – how they might be implemented, and how they would be used by programmers in practice.

The index is patchy, and there is no glossary. The book inevitably introduces a lot of notation, but it is hard to find where a particular notation is introduced. There is a bibliography, of course, but it is hard to find where a particular researcher's work is cited.

Harper airily dismisses the notion that dynamically typed languages are in any way distinctive. He points out that a dynamically typed language can be seen as a degenerate statically typed language with a single type, namely the sum of a number of "classes," with each value tagged to indicate its class; dynamic type-checks are therefore just class-checks. But this is sophistry: If the summands of the single sum type are not themselves types, what are they? This reductionist view of a dynamically typed language is theoretically convenient, but does not reflect the way that programmers think of their language (pragmatics).

The most disappointing part of the book is its inadequate treatment of object-oriented language concepts. The book explains how a dispatch matrix can be extended incrementally by adding new classes and new methods, but conspicuously fails to illuminate the key concept of inheritance. Harper downplays its importance by pointing out that the behaviour of the methods of a subclass could be completely different from the behaviour of the methods of a superclass, but that is true only for methods that are *not* inherited! (Here again the language's pragmatics are ignored.)

The book appears to be aimed at experienced researchers in the theory of programming languages, who will find it invaluable for consolidating their understanding of this fascinating field. Research students in this field will find it challenging but ultimately rewarding.

DAVID WATT
School of Computing Science, University of Glasgow