Book review

Programming with Higher-Order Logic by Dale Miller and Gopalan Nadathur, xiv + 306 pages, Cambridge University Press, 2012.

Some of the most beautiful discoveries in programming languages are their tight connections to logic and proof theory. These connections take two basic forms. In functional programming, we interpret complete proofs as programs, where computation arises from proof reduction according to a fixed strategy. In logic programming, we interpret logical theories as programs, where computation arises from proof construction according to a fixed strategy. This book is concerned with the second form when the underlying logic is higher-order logic. It covers syntax, semantics, and pragmatics of higher-order logic programming in a systematic and easy-to-read manner that will be of great value as introduction and reference for students and researchers in programming languages.

The progression of the chapters is logical, incrementally introducing more advanced concepts as generalizations of simpler ones. Chapter 1 introduces typed first-order terms, unification, and gives first examples of data representation. Chapter 2 presents logic programming with Horn clauses, already employing a proof-theoretic perspective which pays dividends in later chapters. It also provides a first glimpse of λProlog, which is used throughout the book for example programs. In books developing concepts of programming languages, there is a natural tension between abstract ideas and concrete realizations and examples. The authors resolve this by using just two notations: customary logical formulas and inference rules on one hand, and the closely matching λProlog language on the other hand. While not a manual in the traditional sense, the book can easily be read as an introduction to λProlog and its underlying theory. This extends the scope and also the appeal of the book: writing, executing, and testing programs in a mature implementation is an excellent way for students and researchers to plumb their understanding and gain deeper insights into the pragmatics of programming with logic.

Chapter 3 marks the first significant departure from the traditional semantic view of logic programming. Rather than Herbrand semantics, which tracks atoms that are true in a Tarskian sense, it embraces a notion truth based on constructive proof. This richer notion allows for hypothetical reasoning as well as schematic reasoning and quantifier alternations. Computationally, it permits properly scoped dynamic extensions of programs and the available set of constants. The full impact of these generalizations is developed in Chapters 5–11. Interposed is Chapter 4 that introduces the typed λ-calculus which is also needed for the subsequent chapters.

Chapters 5–8 develop basic higher-order logic programming techniques. Since this is a book precisely on this topic, this is the core of the presented material. The reader benefits most with some prior exposure to functional programming and (traditional,

first-order) logic programming, since the ideas here can be seen as a synthesis of the techniques in these paradigms. Chapter 5 is mostly about programming with predicate variables, for example, composing relations or mapping predicates over data structures. This resembles functional programming idioms but affords some additional expressive power. Chapter 6 introduces λProlog's module layer, which can actually be given an interpretation in terms of implication and quantification. This contribution of λProlog to the theory of logic programming seems to have been largely overlooked—a consolidated presentation in this book is therefore an especially valuable resource.

Chapter 7 is all about computing over data with binding structure, sometimes called *abstract binding trees* or *higher-order abstract syntax*. This is one of the unique application areas of higher-order logic programming, where programs can be significantly more elegant than first-order programs for the same tasks. This elegance pays off especially for reasoning about the programs correctness, which is treated informally in this book but has been subsequently been formalized in other systems. The examples in this chapter are already compelling, with larger ones following in later chapters. Chapter 8 on unification discusses not only traditional higher-order unification but also *higher-order pattern unification*. Unlike higher-order unification, pattern unification is decidable and has most general unifiers. It therefore forms a much more practical basis for higher-order logic programming with its pervasive use of unification for parameter passing and data construction. Unfortunately, pattern unification is given somewhat short shrift, even though experience with λProlog and closely related logic programming languages such as Elf and proof assistants such as Isabelle have demonstrated the significance of this subset.

Chapters 9–11 are extended examples on implementing proof systems (Chapter 9), a functional programming language (Chapter 10) and a process calculus (Chapter 11). These are all applications where λProlog is used as a metalanguage for the mechanization of another logic or, more generally, deductive systems. This is where λProlog shines and the main raison-d'être for the language. Other uses of higher-order features can be accomplished in first-order languages such as Prolog by exploiting a form of reflection on syntax, but at the great cost of being extra-logical and therefore difficult to reason about. The presence of variable binding in an object language necessitates corresponding scoping mechanisms in the meta-language, which is precisely what λ-abstraction and associated algorithms provide (capture-avoiding substitution, β-reduction, β_0-reduction, η-expansion, and pattern unification). The book ends with a brief appendix on the Teyjus implementation of λProlog and an extensive bibliography.

Upon reaching the end of the last chapter, one cannot help but feel some regret that the book does not go further. Much has happened since the initial development of higher-order logic programming. Andreoli discovered the general theory of focusing as a broad basis for logic programming. Proof objects have gained a more prominent status through higher-order logic programming in dependent type theory. Two-level meta-reasoning systems such as Twelf and Abella have been designed and implemented to exploit the representation techniques underlying λProlog and related systems. Even more expressive languages based on focusing, linear logic,

and type theory such as Lolli, Forum, Linear LF, or Concurrent LF have been developed and implemented. Upon reflection, however, one feels a sense of consistency and closure in the line of research starting from the proof theory of higher-order logic and ending in λProlog and its implementation in Teyjus. Going significantly further would require an entirely new approach to logic (e.g., linear logic and focusing), or address a whole new set of questions beyond programming (e.g., formalized metareasoning). The only remaining regret is that the book is so intent on programming that it discusses the *theory* of higher-order logic programming hardly at all and mostly in the bibliographic notes. Still, as a practical guide and higher-order counterpart to books such as *The Art of Prolog* (Leon Sterling and Ehud Shapiro) or *The Craft of Prolog* (Richard O'Keefe) it serves an important purpose for students and researchers alike.

Frank Pfenning
Carnegie Mellon University, Pittsburgh