CAMBRIDGE

Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

Introduction

This book is about the nature and benefits of logic programming in the setting of a higher-order logic. We provide in this Introduction a perspective on the different issues that are relevant to a discussion of these topics. Logic programming is but one way in which logic has been used in recent decades to understand, specify, and effect computations. In Section I.1, we categorize the different approaches that have been employed in connecting logic with computation, and we use this context to explain the particular focus we will adopt. The emphasis in this book will be on interpreting logic programming in an expressive way. A key to doing so is to allow for the use of an enriched set of logical primitives while preserving the essential characteristics of this style of specification and programming. In Section I.2, we discuss a notion of expressivity that supports our later claims that some of the logic programming languages that we present are more expressive than others. The adjective "higher order" has been applied to logic in the past in a few different ways, one of which might even raise concern about our plan to use such a logic to perform computations. In Section I.3, we sort these uses out and make clear the kind of higher-order logic that will interest us in subsequent chapters. Section I.4 explains the style of presentation that we follow in this book: Broadly, our goal is to show how higher-order logic can influence programming without letting the discussion devolve into a formal presentation of logic or a description of a particular programming language. The last two sections discuss the prerequisites expected of the reader and the organization of the book.

I.1 Connections between logic and computation

The various roles that logic has played in analyzing and performing computations can be understood as falling under two broad categories that we call the *computation-as-model* and the *computation-as-deduction* approaches. We describe these below.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

In the computation-as-model approach, computations are understood abstractly via mathematical structures that are based on notions such as nodes, transitions, and states. Logic is employed in an external sense in this context to make statements *about* such structures. That is, computations are treated as *models* for logical expressions. Intensional operators, such as the triples of Hoare logic or the modals of temporal and dynamic logics, are often employed to express propositions about change in state. This use of logic to describe and reason about computations probably represents the oldest and most broadly successful interactions between the two areas.

In contrast, the computation-as-deduction approach uses logical expressions such as formulas, terms, types, and proofs directly as elements of the specified computation. In this more rarefied setting, two rather different methods have been employed in describing computations. The *proof normalization* approach views the state of a computation as a proof term and the process of computing as reducing such a term to normal form via, say, β-reduction. This view of computation provides a theoretical basis for the *functional programming paradigm*. In the proof normalization approach, one uses the fact that a given program (proof) has at most one normalized value, and one focuses on producing this value. If types are used, they generally denote "abstract domains" of values, such as the integers and function spaces. In the alternative *proof search* approach, the state of a computation is viewed as a *sequent* that comprises a formula that is to be proved and a collection of assumptions from which the formula is to be established. The process of computing is identified with the search for a derivation of a sequent: The changes that take place in sequents during proof search capture the dynamics of computation. This view of computation can be used to provide a proof-theoretic basis for the *logic programming paradigm*.

Of course, proof search is a rather general activity. For example, mathematicians can be said to be searching for proofs when they try to determine the validity of a proposition. However, it is not sensible to identify the steps that mathematicians take in building proofs with the low-level steps that are used to propel computations associated with a logic program. A particularly important difference between proofs used to realize computations and unrestricted proofs is the fact that in general reasoning, lemmas are discovered and used routinely. In the general setting, the attempt to prove one proposition, say, B, often results in the enunciation of a lemma, say, C, and subsequent attempts to find proofs of C and $C \supset B$. This process may be repeated—another lemma D may be helpful in proving C, and so on—and the result could be a large number of lemmas whose proofs are all used to support the proof of B. In the *sequent calculi*, i.e., the calculi that have been proposed for proving sequents, the *cut rule* provides the mechanism for introducing lemmas in the course of proof search. As such,

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

this rule is a frequent and critical component in any attempt to model genuine mathematical reasoning using such calculi.

Since choosing lemmas involves creativity, the cut rule poses a problem for the mechanization of reasoning. A result that has obvious connotations in this context is Gentzen's famous *cut-elimination* theorem (for classical and intuitionistic logic) that says that if a formula can be proved using the cut rule, then it also can be proved without the cut rule. The proof of this theorem is based intuitively on the observation that lemmas always can be *in-lined* or *re-proved* each time they are needed. The derivations that result from the elimination of uses of the cut rule are often huge and of little value to a mathematician. The fact that they can be constructed, however, is quite interesting from the perspective of computation. The in-lining of proofs, via cut elimination or the closely related operation of β-reduction, is the process that underlies computation in the functional programming paradigm. In the logic programming paradigm as we describe it here, the cut rule is excluded from the execution of logic programs, and computation is based on the search for *cut-free* proofs. The cut rule and the cut-elimination theorem, however, can be used to reason *about* logic programs; i.e., they are part of the metatheory of the paradigm.

I.2 Logical primitives and programming expressivity

In the logic programming setting, one generally partitions formulas into two classes. A formula can be a member of a *logic program*, and as such, it provides part of the computational meaning of the nonlogical constants that appear in it. A formula also can be a *goal* or *query*, and in this role, it represents something to be derived from a given logic program. We shall often idealize the state of the search for a proof by a collection of sequents. A *sequent* in this context will be an expression written as $\Sigma; \mathcal{P} \longrightarrow G$, comprising three parts: a *signature* Σ that is a set of typed, nonlogical constants; a logic program $\mathcal{P}$; and a goal G that is to be proved. The signature Σ denotes the set of constants and predicates that are available for building the terms and formulas in G and $\mathcal{P}$.

An important aspect of logic programming is that a complete proof strategy, in principle, can be structured in the following goal-directed fashion. If the goal formula is not an atom, that is, if its top-level symbol is a logical constant or quantifier, then the search for a proof is completely committed to dealing with that top-level logical constant. Thus the "search semantics" of the logical connectives is fixed and independent of the logic program. On the other hand, if the goal formula is atomic, then the logic program $\mathcal{P}$ is consulted to discover how that atom might be proved. Typically, this involves using *backchaining*, which is the process of finding in the logic program an implicational formula whose consequent matches the atom and then trying to prove its antecedent. Logic

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

programming can be seen abstractly as a logical framework in which a strategy that alternates between goal reduction and backchaining is *complete*, i.e., is capable of finding a proof whenever one exists. This viewpoint is developed in more detail in Section 2.2.

The computational dynamics in logic programming arises from the way the signature, the program, and the query change during the search for a proof. We therefore can understand this dynamics qualitatively by considering the following question.

> Assume that during an attempt to prove the sequent $\Sigma; \mathcal{P} \longrightarrow A$, the search yields the attempt to prove the sequent $\Sigma'; \mathcal{P}' \longrightarrow A'$. What differences can occur when moving from the first to the second sequent?

In this book, we shall consider logic programs based on *Horn clauses* and on a more general class of formulas called *hereditary Harrop formulas*. If $\mathcal{P}$ is a Horn clause program (either first order or higher order), then Σ' and $\mathcal{P}'$ must be identical to Σ and $\mathcal{P}$, respectively. Thus the signature and logic program are global and immutable and have a flat structure during computation; in particular, Horn clauses do not support the capability of using some data structures and some clauses locally and only for auxiliary calculations. The differences between the atoms A and A' are determined, on the other hand, by the logic program $\mathcal{P}$, and these can be rich enough to capture arbitrary computations. Notice, however, that the dynamics of such computations has a largely *nonlogical* character; that is, it is dependent on the meaning associated with predicate symbols through the assumptions in the logic program. If programs are allowed to involve more logical primitives, more of the character of the dynamics of computation may depend on the *logical* structure, and as a result, the metatheory of the logic can be of more value in proving properties of those programs.

Using hereditary Harrop formulas improves the dynamics of proof search: In particular, both the signature Σ' and the program $\mathcal{P}'$ can be larger than Σ and $\mathcal{P}$, respectively. As a particular consequence, it is possible for a program to grow by the addition of clauses that can be used only in a local proof search attempt. Similarly, it is possible to introduce data constructors that are available only for part of the computation. In this way, the logical framework is capable of supporting the use of modular programming and data abstraction techniques.

We shall limit our attention to classical and intuitionistic logic as they are applied to Horn clauses and to hereditary Harrop formulas. If one were to consider proof search in the more general setting of linear logic, the alternation between goal reduction and backchaining still would yield a complete proof procedure (for a suitable presentation of linear logic), and the dynamics of proof search would improve beyond what we have observed for the two fragments of logic just discussed. Although this is an interesting direction to pursue,

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

logic programming based on linear logic is beyond the scope of the topics we consider here.

I.3 The meaning of *higher-order logic*

The term *higher-order logic* has been used ambiguously in the literature. We identify three common interpretations below and then explain the sense in which we will be using the form in this book.

Philosophers of mathematics often distinguish between first-order logic and second-order logic. The latter logic, which is used as a formal basis for all of mathematics, involves quantification over the domain of all possible functions. A consequence of Kurt Gödel's celebrated first incompleteness theorem is that truth in this logic cannot be recursively axiomatized. Thus higher-order logic interpreted in this sense consists largely of a model-theoretic study, typically of the *standard model of arithmetic*.

Proof-theoreticians take logic to be synonymous with a formal system that provides a recursive enumeration of the notion of theoremhood. A higher-order logic is understood no differently. The distinctive characteristic of such a logic, instead, is the presence of predicate quantification and of comprehension, i.e., the ability to form abstractions over formula expressions. These features, especially the ability to quantify over predicates, profoundly influence the proof-theoretic structure of the logic. One important consequence is that the simpler induction arguments of cut elimination that are used for first-order logic do not carry over to the higher-order setting, and more sophisticated techniques, such as the "*candidats de réductibilité*" due to Jean-Yves Girard, must be used. Semantical methods also can be employed, but the collection of models now must include *nonstandard models* that use restricted function spaces in addition to the standard models used for second-order logic.

Implementers of deduction usually interpret higher-order logic as any computational logic that employ λ-terms and quantification at higher-order types, although not necessarily at predicate types. Notice that if quantification is extended only to non–predicate function variables, then the logic is similar to a first-order one in that the cut-elimination process can be defined using an induction involving the sizes of (cut) formulas. However, such a logic may incorporate a notion of equality based on the rules of λ-conversion, and the implementation of theorem proving in it must use (some form of) higher-order unification.

Clearly, it is not sensible to base a programming language on a higher-order logic in the first sense. Our use of this term therefore is restricted to the second

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

and third senses. Notice that these two views are distinct. As we have already commented, a logic that is higher order in the third sense may well not permit quantification over predicates and thus may not be higher order in the second sense. Conversely, a logic can be higher order in the second sense but not in the third: There have been proposals for adding forms of predicate quantification to computational logics that do not use λ-terms and in which the equality of expressions continues to be based on the identity relation.

The actual higher-order logic that we shall use in this book is a simplified form of an intuitionistic version of the *Simple Theory of Types* that was developed by Alonzo Church. Our simplification leaves out axioms concerning extensionality, infinity, and choice that are needed for formalizing mathematics but that do not play a role in and indeed interfere with use of the logic in describing computations. The resulting logic extends first-order logic by permitting quantification at all types and replaces both first-order terms and first-order formulas by simply typed λ-terms complemented by a notion of equality based on β- and η-conversion. This logic does permit predicate quantification, which makes theorem proving in it particularly challenging. In first-order logic, substitution into an expression does not change its logical structure, and all the needed instantiations in a proof can be produced simply through the unification of atomic formulas. With the inclusion of predicate quantification, instantiations can introduce new occurrences of logical connectives and quantifiers in formulas, and as a result, unification is not rich enough to find all substitutions needed for proofs. However, we shall, restrict the uses of predicate variables in the logic programming languages we consider in such a way that unification becomes sufficient once again for finding all the necessary instantiations.

I.4 Presentation style

This book is intended to be an exposition of programming techniques based on the use of a higher-order logic. In order to discuss these techniques in detail, we need to be able to present actual logic programs. More specifically, a concrete syntax must be picked for programs and goals, language principles such as modularity and typing must be established, and strategies for dealing effectively with nondeterministic proof search must be chosen. Toward meeting these requirements, we introduce the programming language λProlog, which represents one way of addressing these pragmatic aspects. This language also gives us a setting in which to discuss relevant issues concerning the computational use of higher order logic. Thus goal-directed search for higher-order hereditary Harrop formulas must be translated into an operational semantics and, subsequently, an implementation of λProlog. Similarly, higher-order logic and a rich use of logical primitives raises the issue of solving equations between λ-terms

© in this web service Cambridge University Press www.cambridge.org

Cambridge
Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

modulo β- and η-conversion rules and in the presence of mixed quantifier prefixes. The λProlog language gives us a concrete setting in which to understand the structure of such issues as well as to appreciate practical approaches to solving them.

Although we discuss λProlog explicitly, this is not intended to be a book *about* λProlog. We introduce the syntax of this language and we display several λProlog programs, but we do not provide enough information about the language for this book to serve as a programming manual. Rather, the focus is on painting a broad picture of the interplay between proof search in higher-order logic and computational principles: This focus underlies the discussion of language structure initially and the presentation later of several applications where higher-order logic programming techniques lead to appealing and natural solutions. A reader who is not satisfied with this kind of exposure to the language and wants a more detailed, manual-like presentation should consult the documentation accompanying one of its implementations, such as the Teyjus system that is briefly described in the Appendix.

While our emphasis is on understanding high-level, logic-related aspects of programming, we emphasize that this book is not a formal development of logic in any sense. In particular, we try to build a good intuitive understanding of higher-order logic characteristics, but we do this without providing many formal definitions and theorems. Instead, most formal aspects of this logic are exposed through examples and probed by tracing computational behavior. However, detailed bibliographic references to literature containing such formal presentations are included at the end of many chapters for the interested reader.

I.5 Prerequisites

The ideal reader of this book would have had prior exposure to high-level programming and to the rudiments of logic and logic programming. We specifically assume that the reader knows how to write and execute simple programs in some dialect of Prolog. We use small programming examples in λProlog to bring out the different ideas we present. A reader who has a programming feel for Prolog will find these examples easy to understand because λProlog inherits many features and conventions from Prolog. Conversely, someone not familiar with how computations are organized in logic programming languages may have difficulty in understanding the λProlog examples in detail. Knowledge of "advanced" aspects of Prolog, however, is not necessary. In fact, such knowledge could be confusing: Advanced Prolog features often derive from nonlogical aspects of the language, whereas our focus here will be on finding logical solutions to the problems that have led to the proliferation of nonlogical solutions that are familiar to Prolog programmers.

© in this web service Cambridge University Press www.cambridge.org

Cambridge
Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

I.6 Organization of the book

This book has four conceptual parts that are identified in Figure I.1 together with their dependencies.

The first part introduces a proof-theoretic foundation for logic programming in the setting of first-order logic. Chapter 1 describes how symbolic objects might be represented using simply typed first-order terms that are manipulated using first-order unification. Chapter 2 presents an abstract framework for logic programming and elaborates this framework using first-order Horn clauses. The resulting language then is extended in Chapter 3 by using a richer class of formulas known as first-order hereditary Harrop formulas.

The second part of this book generalizes the structure of logic programming languages discussed in the first part to the higher-order setting. Chapter 4 introduces simply typed λ-terms and exposes some of the properties of the reduction computation and the process of solving equations relative to these terms. Formulas are identified as the specific collection of simply typed λ-terms that have a certain type, and Church's Simple Theory of Types defines a logic over these formulas. Chapter 5 identifies higher-order versions of Horn clauses and hereditary Harrop formulas within this logic. These classes of formulas provide the basis for *higher-order logic programming*, some characteristics of which we also expose in this chapter.

The third part of this book deals with pragmatic issues related to programming. Chapter 6 shows how code-structuring possibilities can be realized by exploiting features of higher-order hereditary Harrop formulas. The Appendix

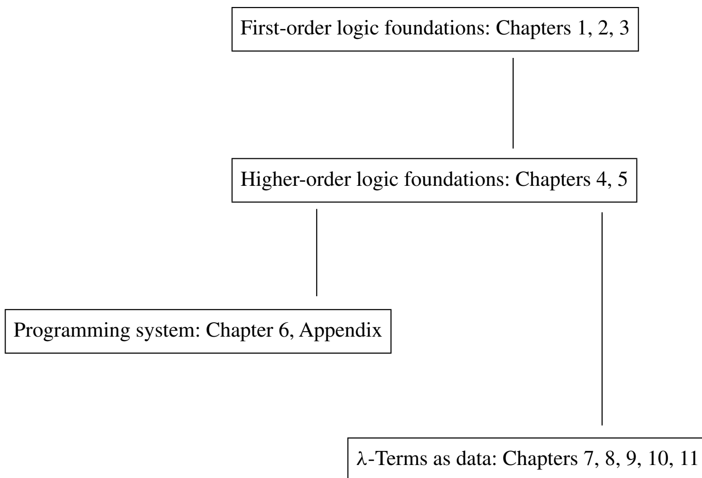

Figure I.1 Dependency and grouping of chapters.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

describes how the logic specifications presented in this book can be written as λProlog programs and executed using the Teyjus implementation of λProlog.

The fourth part of this book is devoted to showing the benefits of the ability to compute directly on λ-terms. One part of this discussion consists of explaining the general structure that supports this approach. Chapter 7 illustrates how computations on λ-terms can be used to encode and manipulate syntactic objects that contain binding operators. Proof search in higher-order logic requires solving variously quantified equalities between λ-terms, and as a result, higher-order unification plays an important role in the implementation of such logic programming languages. Chapter 8 discusses the structure of procedures for higher-order unification and the more limited higher-order pattern unification that underlies computation in an important subset of higher-order hereditary Harrop formulas that is known as L_λ. The remaining chapters in this fourth part, which can be read independently of each other, present different applications that involve computing on symbolic structures encoded using λ-terms. In particular, Chapter 9 considers the problem of implementing natural deduction and sequent calculus proof systems as well as tactic-based provers, Chapter 10 considers several computations in the context of the untyped λ-calculus and a simple functional programming language built on it, and Chapter 11 considers specifications and computations related to the π-calculus.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-87940-8 - Programming with Higher-Order Logic
Dale Miller and Gopalan Nadathur
Excerpt
More information

1

First-Order Terms and Representations of Data

Our initial discussion of logic programming focuses on first-order languages. In this chapter, we limit our attention to the capabilities for representing data that are present in such languages. These capabilities are provided for by first-order terms. The terms that we use in our exposition of data representation here are similar to those in a conventional logic programming language such as Prolog with one difference: We shall be interested in a *typed* version of these terms. In the first two sections that follow, we describe the structure of the types that are employed to classify terms. Section 1.3 then introduces typed first-order terms, and the following section discusses the pragmatics of using such terms to represent structured and recursively constructed data. The last section in this chapter considers the operation of first-order unification, the primary mechanism for analyzing data that are encoded using first-order terms. To ground this discussion—in particular, to show how the type and term languages may be identified in a programming setting—we use the actual syntax of λProlog in our presentation.

1.1 Sorts and type constructors

The starting point for a type system is a set of atomic or unanalyzable types. We shall refer to such types as *sorts*. Most typed programming languages have a set of built-in sorts associated with them. In the case of λProlog, any implementation of the language is expected to support at least the following collection of sorts with the corresponding denotations:

`int`	an implementation dependent range of integers
`real`	an implementation dependent set of real numbers
`string`	sequences of characters
`in_stream`	character streams that can be read from

© in this web service Cambridge University Press www.cambridge.org