# Introduction: Logic and Logic Programming Languages

**Michael J. O'Donnell**

---

## Contents



This text is a preprint of Chapter 1 in the *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5 on *Logic Programming*, Dov Gabbay editor, Oxford University Press. José Meseguer, Dale Miller, and Edward L. Wimmers served as official readers, and provided useful feedback on drafts of the chapter.

# 1 Introduction

## 1.1 Motivation

*Logic*, according to Webster's dictionary [Web87], is *'a science that deals with the principles and criteria of validity of inference and demonstration: the science of the formal principles of reasoning.'* Such *'principles and criteria'* are always described in terms of a *language* in which inference, demonstration, and reasoning may be expressed. One of the most useful accomplishments of logic for mathematics is the design of a particular formal language, the *First Order Predicate Calculus* (FOPC). FOPC is so successful at expressing the assertions arising in mathematical discourse that mathematicians and computer scientists often identify logic with classical logic expressed in FOPC. In order to explore a range of possible uses of logic in the design of programming languages, we discard the conventional identification of logic with FOPC, and formalize a general schema for a variety of logical systems, based on the dictionary meaning of the word. Then, we show how logic programming languages may be designed systematically for any sufficiently effective logic, and explain how to view Prolog, Datalog, $\lambda$Prolog, Equational Logic Programming, and similar programming languages, as instances of the general schema of logic programming. Other generalizations of logic programming have been proposed independently by Meseguer [Mes89], Miller, Nadathur, Pfenning and Scedrov [MNPS91], Goguen and Burstall [GB92].

The purpose of this chapter is to introduce a set of basic concepts for understanding logic programming, not in terms of its historical development, but in a systematic way based on retrospective insights. In order to achieve a systematic treatment, we need to review a number of elementary definitions from logic and theoretical computer science and adapt them to the needs of logic programming. The result is a slightly modified logical notation, which should be recognizable to those who know the traditional notation. Conventional logical notation is also extended to new and analogous concepts, designed to make the similarities and differences between logical relations and computational relations as transparent as possible. Computational notation is revised radically to make it look similar to logical notation. The chapter is self-contained, but it includes references to the logic and theoretical computer science literature for those who wish to explore connections.

There are a number of possible motivations for developing, studying, and using logic programming languages. Many people are attracted to Prolog, the best known logic programming language, simply for the special programming tools based on unification and backtracking search that it provides. This chapter is not concerned with the utility of particular logic programming languages as programming tools, but with the value of concepts from logic, particularly semantic concepts, in the design, implementation, and use of programming languages. In particular, while denotational and algebraic semantics provide excellent tools to *describe* important aspects of programming systems, and often to prove correctness of implementations, we will see that logical semantics can exploit the strong traditional consensus about the meanings of certain logical notations to *prescribe* the behavior of programming systems. Logical semantics also provides a natural approach, through proof systems, to verifiably correct implementations, that is sometimes simpler than the denotational and algebraic approaches. A comparison of the three styles of semantics will show that denotational and algebraic semantics provide *descriptive* tools, logical semantics provides *prescriptive* tools, and the methods of algebraic semantics may be used to translate logical semantics into denotational/algebraic semantics.

In this chapter, a relation is called *computable* if and only if its characteristic function is total recursive, and a relation is *semicomputable* if and only if the set of ordered pairs in the relation is recursively enumerable. Recursion theorists and theoretical computer scientists often refer to computable sets as *decidable* sets, but logicians sometimes call a theory *decidable* when every formula is either provable or refutable in the theory. The two meanings of 'decidable' are closely connected, but not identical, and we avoid confusion by choosing a different word. When some component of a relation is a finite set, the set is assumed to be represented by a list of its members for the purpose of discussing computability and semicomputability.

## 1.2 A Notational Apology

In order to understand logic programming rigorously in terms of formal concepts from mathematical logic, and at the same time intuitively, we need to look closely at the details of several formal relations from logic and from theory of computation. We must come to understand the formal similarities and differences between these relations, and how those formal properties arise from the intuitive similarities and differences in our intended applications of these relations. Unfortunately, the conventional notations for logic and computation look radically different, and take advantage of different simplifying assumptions, which obscures those connections that are essential to intuitive applications of the corresponding concepts. So, we will make visually small variations on conventional logical notation, extending it to deal with questions and their answers as well as the traditional assertions and their semantic interpretations. Then, we will radically redesign conventional recursion-theoretic notation in order to display visually the connections between computational relations and logical relations. In order to be prepared for the strange look of the notations, we need to review them all briefly in advance, although the precise definitions for the concepts that they denote will be introduced gradually through sections 2–3.

The important domains of conventional logical objects for our study are the sets of

- logical assertions, or formulae $\mathbf{F}$
- sets of formulae, or theories $\mathbf{T} \in \mathbf{2}^{\mathbf{F}}$
- semantic interpretations, or models $\mathbf{M}$
- sets of models, representing knowledge $\mathbf{K} \in \mathbf{2}^{\mathbf{M}}$
- proofs, or derivations $\mathbf{D}$

We add the unconventional domain of

- questions $\mathbf{Q}$

Answers to questions are particular formulae, so no additional domain is required for them. The domains of conventional computational objects are the sets of

- programs $\mathbf{P}$
- inputs $\mathbf{I}$
- computations $\mathbf{C}$
- outputs $\mathbf{O}$

In recursion-theoretic treatments of computation, programs, inputs, and outputs are all integers, but our analysis is more convenient when they are allowed to be different domains. We will find strong intuitive analogies and formal connections between

- programs and sets of formulae
- inputs and questions
- computations and proofs
- outputs and formulae (intended as answers to questions)

In order to understand the analogies and formal connections thoroughly, we must investigate a number of relations between domains with varying arities from two to four. In all cases, we will use *multiple infix* notation. That is, each $n$-ary relation will be denoted by $n-1$ symbols separating its arguments. With some care in the geometrical design of the separator symbols, we get a reasonably mnemonic notation.

There are two quaternary relational notations from which all the other notations may be derived. Let $Q$ be a question, $\mathbf{T}$ a set of formulae, $D$ a proof, and $F$ a formula. The notation

$$Q \mathrel{?\!-} \mathbf{T} \mid D - F$$

means that in response to the question $Q$, given the postulates in $\mathbf{T}$, we may discover the proof $D$ of the answer $F$. Similarly, let $I$ be an input, $P$ a program, $C$ a computation, and $O$ an output. The notation

$$I \rhd P \mathbin{[\!]} C \rightarrow O$$

means that in response to the input $I$, the program $P$ may perform the computation $C$, yielding output $O$. The correspondence between the arguments $Q$ and $I$, $\mathbf{T}$ and $P$, $D$ and $C$, $F$ and $O$ displays the crucial correspondence between logic and computation that is at the heart of logic programming.

There are two closely related trinary notations.

$$Q \mathrel{?\!-} \mathbf{T} \vdash F$$

means that there exists a proof $D$ such that $Q \mathrel{?\!-} \mathbf{T} \mid D - F$, and

$$I \rhd P \mathrel{[\!]\!\!\rightarrow} O$$

means that there exists a computation $C$ such that $I \rhd P \mathbin{[\!]} C \rightarrow O$. The symbol $\vdash$ in $Q \mathrel{?\!-} \mathbf{F} \vdash F$ is the conventional symbol for proof in mathematical logic; we take the liberty of decomposing it into the two symbols $\mid$ and $-$ for the quaternary notation. The conventional recursion-theoretic notation for our $I \rhd P \mathrel{[\!]\!\!\rightarrow} O$ is $\phi_P(I) = O$. The computational symbol $\mathrel{[\!]\!\!\rightarrow}$ and its components $[\!]$ and $\rightarrow$ are designed to have similar shapes to $\vdash$, $\mid$, and $-$.

Other relations from logic do *not* correspond directly to computational relations, but can be understood by their connections to the quaternary form, in which the logic/computation correspondence is direct and transparent. In Section 3.2 I define $Q \mathrel{?\!-} \mathbf{T} \mid D - F$ to hold exactly when both

$$Q \mathrel{?\!-} F \qquad\qquad \text{and} \qquad\qquad \mathbf{T} \mid D - F$$

where $Q \mathrel{?\!-} F$ means that $F$ is an answer (not necessarily a correct one) to the question $Q$, and $\mathbf{T} \mid D - F$ means that $D$ is a proof of $F$, using postulates in the set $\mathbf{T}$. $\mathbf{T} \mid D - F$ is a conventional concept from mathematical logic (often written $\mathbf{T}, D \vdash F$ or $\mathbf{T} \vdash_D F$). The question-answer relation $?\!-$ is not conventional. Notice that each separating symbol in the quaternary notation $Q \mathrel{?\!-} \mathbf{T} \mid D - F$ is used exactly once in the binary and trinary forms from which it is defined, so the notational conjunction of symbols suggests the logical conjunction of the denoted relations. Unfortunately, while the symbol $?\!-$ appears between the question $Q$ and the answer formula $F$ in the binary notation $Q \mathrel{?\!-} F$, it is not adjacent to $F$ in the quaternary notation $Q \mathrel{?\!-} \mathbf{T} \mid D - F$. The dash component $-$ of the symbol $?\!-$ mimics the $-$ symbol at the end of the quaternary notation, and the similar component of the $\vdash$ symbol from the trinary notation above, as a reminder that the $?\!-$ symbol is expressing a relation to the final answer formula $F$, rather than to the set $\mathbf{T}$ of postulated formulae.

The quaternary computational relation is also defined as the conjunction of a binary and a trinary relation, but the arguments involved in these relations do *not* correspond to the arguments of the binary and trinary relations from logic. In Section 3.3 I define $I \rhd P \Box C \rightharpoonup O$ to hold exactly when both

$$I \rhd P \Box C \qquad \text{and} \qquad C \rightharpoonup O$$

where $I \rhd P \Box C$ means that the program $P$ on input $I$ may perform the computation $C$, and $C \rightharpoonup O$ means that the computation $C$ yields output $O$. In this case, the mnemonic suggestion of the conjunction of the trinary and binary relations in the quaternary notation works out perfectly, as all argument positions are adjacent to the appropriate separator symbols.

A few other derived notations are useful for denoting relations from logic. These all agree with conventional notation in mathematical logic.

$$\mathbf{T} \vdash F$$

means that there exists a proof $D$ such that $\mathbf{T} \mid D - F$—that is, $F$ is formally derivable from $\mathbf{T}$. Corresponding to the relation $\vdash$ of *formal derivability* is the relation $\models$ of *semantic entailment*.

$$\mathbf{T} \models F$$

means that $F$ is semantically entailed by $\mathbf{T}$. Similarly,

$$Q \mathrel{?\!-} \mathbf{T} \models F$$

means that $F$ is an answer to $Q$ semantically entailed by $\mathbf{T}$ ($Q \mathrel{?\!-} F$ and $\mathbf{T} \models F$) in analogy to $Q \mathrel{?\!-} \mathbf{T} \vdash F$. The mathematical definition of semantic entailment involves one more semantic relation. Let $M$ be a model, and $F$ a formula.

$$M \models F$$

means that $F$ is true in $M$.

Table 1 displays all of the special notations for semantic, proof-theoretic, and computational relations. The precise meanings and applications of these notations are developed at length in subsequent sections. The notation described above is subscripted when necessary to distinguish the logical and computational relations of different systems.

| **Logic** | | **Computation** |
|---|---|---|
| **Semantics** | **Proof** | |
| | $Q \mathrel{?\!\!-} \mathbf{T} \mid D - F$ | $I \triangleright\ P \mathrel{\Box} C \rightarrow O$ |
| $Q \mathrel{?\!\!-} \mathbf{T} \models F$ | $Q \mathrel{?\!\!-} \mathbf{T} \vdash F$ | $I \triangleright\ P \mathrel{\Box\!\!\mapsto} O$ |
| | | $I \triangleright\ P \mathrel{\Box} C$ |
| $Q \mathrel{?\!\!-} F$ | | |
| | | $C \rightarrow O$ |
| | $\mathbf{T} \mid D - F$ | |
| $\mathbf{T} \models F$ | $\mathbf{T} \vdash F$ | |
| $M \models F$ | | |

**Table 1. Special Notations for Logical and Computational Relations**

# 2 Specifying Logic Programming Languages

Logic typically develops its *'principles and criteria of validity of inference'* by studying the relations between *notations* for assertions, the *meanings* of those assertions, and *derivations* of notations expressing true assertions. Mathematical logic formalizes these concepts using *logical formulae* as notations, sets of *models* to analyze meanings and characterize truth, and *demonstrations* or *proofs* as derivations of true formulae and inferences. The structure of formulae alone is *syntax*, their relation to models is *semantics*, and their relation to proofs is *proof theory*. Syntax is not relevant to the present discussion. We must examine formal systems of semantics, and augment them with formal concepts of questions and their answers, in order to understand the specification of a logic programming language. In Section 3 we see how formal systems of proof provide a natural approach to the implementation of computations for logic programming.

## 2.1 Semantic Systems and Semantic Consequences

A *semantic system* relates a set $\mathbf{F}$ of logical formulae to a set $\mathbf{M}$ of formal models, each representing a conceivable state of the world in enough detail to determine when a given formula represents a true assertion in that state of the world.

**Definition 2.1.1.** A *semantic system* is a system $\mathcal{S} = \langle \mathbf{F}, \mathbf{M}, \models \rangle$, where

1. $\mathbf{F}$ is a set of *logical formulae*
2. $\mathbf{M}$ is a set of *models*
3. $\models$ is a relation on $\mathbf{M} \times \mathbf{F}$

Let $\mathbf{K} \subseteq \mathbf{M}$. $\mathbf{Theory}(\mathbf{K}) = \{F \in \mathbf{F} : M \models F \textit{ for all } M \in \mathbf{K}\}$.
Let $\mathbf{T} \subseteq \mathbf{F}$. $\mathbf{Models}(\mathbf{T}) = \{M \in \mathbf{M} : M \models F \textit{ for all } F \in \mathbf{T}\}$.

Intuitively, $M \models F$ is intended to mean that formula $F$ *holds in*, or *is valid in*, or *is satisfied by*, model $M$. $\mathbf{Theory}(\mathbf{K})$ is the fullest possible description of $\mathbf{K}$ using a set of formulae in the language of the system. $\mathbf{Models}(\mathbf{T})$ represents the state of knowledge given implicitly by the formulae in $\mathbf{T}$—knowing $\mathbf{T}$ we know that reality corresponds to one of the models in $\mathbf{Models}(\mathbf{T})$, but we do not know which one. Notice the anti-monotone relation between $\mathbf{F}$ and $\mathbf{M}$:

$$\begin{array}{c}
\mathbf{T}_1 \subseteq \mathbf{T}_2 \text{ if and only if } \mathbf{Models}(\mathbf{T}_1) \supseteq \mathbf{Models}(\mathbf{T}_2) \\
\mathbf{K}_1 \subseteq \mathbf{K}_2 \text{ if and only if } \mathbf{Theory}(\mathbf{K}_1) \supseteq \mathbf{Theory}(\mathbf{K}_2)
\end{array}$$

$$\begin{array}{rcl}
\mathbf{Models}(\mathbf{T}_1 \cup \mathbf{T}_2) & = & \mathbf{Models}(\mathbf{T}_1) \cap \mathbf{Models}(\mathbf{T}_2) \\
\mathbf{Models}(\mathbf{T}_1 \cap \mathbf{T}_2) & = & \mathbf{Models}(\mathbf{T}_1) \cup \mathbf{Models}(\mathbf{T}_2) \\
\mathbf{Theory}(\mathbf{K}_1 \cup \mathbf{K}_2) & = & \mathbf{Theory}(\mathbf{K}_1) \cap \mathbf{Theory}(\mathbf{K}_2) \\
\mathbf{Theory}(\mathbf{K}_1 \cap \mathbf{K}_2) & = & \mathbf{Theory}(\mathbf{K}_1) \cup \mathbf{Theory}(\mathbf{K}_2)
\end{array}$$

In order to provide satisfactory intuitive insight, a semantic system must relate the syntactic structure of formulae to the determination of truth. For example, well-known sets of formulae often come with a syntactic operator to construct, from two formulae $A$ and $B$, their *logical conjunction* $A \wedge B$. The semantics for conjunctions is defined structurally, by the rule $\boldsymbol{M} \models A \wedge B$ *if and only if* $\boldsymbol{M} \models A$ *and* $\boldsymbol{M} \models B$. The formal analysis of this chapter deals only with the abstract relation of a model to a formula that holds in the state of the world represented by that model, not the internal structure of that relation, because we are interested here in the *use* of semantics for understanding logic programming, rather than the deeper structure of semantics itself. Goguen's and Burstall's *institututions* [GB92] are similar to semantic systems, but they capture in addition the structural connection between syntax and semantics through category theory, and show that the functions **Models** and **Theory** form a *Galois connection*.

Notice that the sets **F** of formulae and **M** of models are not required to be given effectively. In well-known semantic systems, the set of formulae is normally computable, since formulae are normally finite syntactic objects, and it is easy to determine mechanically whether a given object is a formula or not. Infinite formulae, however, have important uses, and they can be given practical computational interpretations, so we do *not* add any formal requirement of computability. The set of models, on the other hand, is typically quite complex, because models represent conceivable states of an external world, rather than finite constructions of our own minds. In fact, for many semantic systems there are technical set-theoretic problems even in regarding the collection of models in the system as a set, but those problems do not affect any of the results of this chapter.

In this chapter, basic concepts are illustrated through a running example based on the *Shallow Implicational Calculus* (SIC), designed to be almost trivial, but just complex enough to make an interesting example. More realistic examples are treated toward the end of the chapter.

**Example 2.1.2.** Let **At** be a set of *atomic propositional formulae*. The set $\mathbf{F}_{Sh}$ of formulae in the *Shallow Implicational Calculus* is the smallest set such that:

1. $\mathbf{At} \subseteq \mathbf{F}_{Sh}$
2. If $a, b \in \mathbf{At}$, then $(a \Rightarrow b) \in \mathbf{F}_{Sh}$

The set $\mathbf{M}_{Sh}$ of models in SIC is defined by

$$\mathbf{M}_{Sh} = \mathbf{2}^{\mathbf{At}}$$

The semantic relation $\models_{Sh} \subseteq \mathbf{M}_{Sh} \times \mathbf{F}_{Sh}$ is defined by:

1. For $a \in \mathbf{At}$, $M \models_{Sh} a$ if and only if $a \in M$
2. $M \models_{Sh} (a \Rightarrow b)$ if and only if either $b \in M$ or $a \notin M$

Now $\langle \mathbf{F}_{Sh}, \mathbf{M}_{Sh}, \models_{Sh} \rangle$ is a semantic system, representing the classical concept of meaning for the implicational formulae of SIC.

SIC is just the restriction of the Classical Propositional Calculus [And86, Kle52, Gal86] to atomic propositional formulae, and implications between atomic propositional formulae. It is called 'shallow' because no nesting of implications is allowed. Since the truth of formulae in SIC (as in the Propositional Calculus) is determined entirely by the truth of atomic formulae, a model merely specifies the set of atomic formulae that are true in a given conceivable state of the world. Following the tradition of material implication in classical logic, an implication is true precisely when its conclusion is true, or its hypothesis is false.

For the formal definition of a logic programming language, the important thing about a semantic system is the *semantic-consequence* relation that it defines, determining when the truth of a set of formulae justifies inferring the truth of an additional formula.

**Definition 2.1.3 ([And86, Gal86] ).** Let $\mathcal{S} = \langle \mathbf{F}, \mathbf{M}, \models \rangle$ be a semantic system. The *semantic-consequence* relation defined by $\mathcal{S}$ is $\models \subseteq \mathbf{2}^{\mathbf{F}} \times \mathbf{F}$, where $\mathbf{T} \models F$ if and only if $M \models F$ for all $M \in \mathbf{Models}(\mathbf{T})$.

The semantic-consequence relation $\models$ is *compact* if and only if, for all $\mathbf{T} \subseteq \mathbf{F}$ and $F \in \mathbf{F}$, whenever $\mathbf{T} \models F$ then there exists a finite subset $\mathbf{T}^f \subseteq \mathbf{T}$ such that $\mathbf{T}^f \models F$.

Intuitively, $\mathbf{T} \models F$ means that $F$ is a *semantic consequence* of $\mathbf{T}$, since $F$ must be true whenever all formulae in $\mathbf{T}$ are true. Semantic consequences are often called *logical consequences*; our terminology is chosen to highlight the contrast between semantic consequences and the *provable consequences* of Definition 3.1.4. Notice that $\mathbf{Theory}(\mathbf{Models}(\mathbf{T}))$ is the set of semantic consequences of $\mathbf{T}$. It is easy to show that an arbitrary relation $\models$ on $\mathbf{2}^{\mathbf{F}} \times \mathbf{F}$ is the semantic-consequence relation of some semantic system if and only if it is

1. *reflexive*: $F \in \mathbf{T}$ implies that $\mathbf{T} \models F$
2. *monotone*: $\mathbf{T} \models F$ and $\mathbf{T} \subseteq \mathbf{U}$ imply that $\mathbf{U} \models F$
3. *transitive*: $\mathbf{T} \models F$ and $\mathbf{T} \cup \{F\} \models G$ imply that $\mathbf{T} \models G$

In order for a semantic-consequence relation to be useful for logic programming, or for rigorous formal reasoning, it must be sufficiently effective. Well-known semantic systems normally define semantic-consequence relations that are compact—their behavior on arbitrary sets is determined by their behavior on finite sets. Normal semantic-consequence relations are semicomputable, but not necessarily computable, when restricted to finite sets of formulae in the first component. Fortunately, semicomputability is enough for logic programming.

**Example 2.1.4.** The semantic-consequence relation $\models_{Sh}$ of the Shallow Implicational Calculus of Example 2.1.2 is compact, and has a particularly simple behavior:

1. for atomic formulae $a \in \mathbf{At}$, $\mathbf{T} \models_{Sh} a$ if and only if there is a finite sequence $\langle a_0, \ldots, a_m \rangle$ of atomic formulae such that $a_0 \in \mathbf{T}$, and $(a_i \Rightarrow a_{i+1}) \in \mathbf{T}$ for all $i < m$, and $a_m = a$
2. $\mathbf{T} \models_{Sh} (a \Rightarrow b)$ if and only if there is a finite sequence $\langle a_0, \ldots, a_m \rangle$ of atomic formulae such that $a_0 \in \mathbf{T} \cup \{a\}$, and $(a_i \Rightarrow a_{i+1}) \in \mathbf{T}$ for all $i < m$, and $a_m = b$

We may think of the implications in $\mathbf{T}$ as directed edges in a graph whose vertices are atomic formulae. Atomic formulae in $\mathbf{T}$ are marked *true*. An atomic formula $a$ is a semantic consequence of $\mathbf{T}$ precisely if there is a directed path from some atomic formula in $\mathbf{T}$ to $a$. Similarly, an implication $a \Rightarrow b$ is a semantic consequence of $\mathbf{T}$ precisely if there is a directed path from $a$, or from an atomic formula in $\mathbf{T}$, to $b$. Notice that SIC satisfies the *deduction property*: [And86, Kle52, Gal86]

$$\mathbf{T} \models_{Sh} (a \Rightarrow b) \text{ if and only if } \mathbf{T} \cup \{a\} \models_{Sh} b$$

A semantic system provides a conceptual tool for analyzing a primitive sort of communication in a monologue. A state of implicit knowledge is naturally represented by the set of models corresponding to conceivable states of the world that are consistent with that knowledge. Notice that larger sets of models represent smaller amounts of knowledge. For a general discussion of knowledge as sets of models, the shortcomings of such representations, and problems and paradoxes that arise when subtle sorts of knowledge are considered, see [FHV84]. The knowledge involved in formal analysis of the examples of logic programming in this chapter is simple enough to be represented by sets of models without presenting the problems that arise in a more general setting. Explicit knowledge is naturally represented by a set of formulae. $\mathbf{Models}(\mathbf{T})$ is the implicit knowledge given explicitly by $\mathbf{T}$. Similarly, $\mathbf{Theory}(\mathbf{K})$ is the strongest explicit representation of the implicit knowledge $\mathbf{K}$ that is expressible in a given language, but there is no guarantee that an agent with implicit knowledge $\mathbf{K}$ can effectively produce all of the explicit knowledge $\mathbf{Theory}(\mathbf{K})$.

Consider a *speaker*, whose state of knowledge is represented by $\mathbf{K}_s$, and an *auditor* with initial knowledge $\mathbf{K}_a^0$. The speaker wishes to communicate some of her knowledge to the auditor, so she utters a set of formulae $\mathbf{T} \subseteq \mathbf{Theory}(\mathbf{K}_s)$. The impact of the speaker's utterance on the auditor's state of knowledge is to remove from the auditor's set of models those that do not satisfy $\mathbf{T}$. That is, $\mathbf{K}_a^0$ is replaced by $\mathbf{K}_a^1 = \mathbf{K}_a^0 \cap \mathbf{Models}(\mathbf{T})$. Notice that, if the auditor's initial knowledge is minimal, that is if $\mathbf{K}_a^0$ is the set of all models in the semantic system, then $\mathbf{K}_a^1 = \mathbf{Models}(\mathbf{T})$, so the formulae implied by the new knowledge, $\mathbf{Theory}(\mathbf{K}_a^1)$, are exactly the semantic consequences of $\mathbf{T}$. In logic programming systems, the programmer plays the part of the speaker above, the processor plays the part of the auditor, the program is the utterance, and the logical meaning of the program is the resulting state of knowledge produced in the auditor/processor. Inputs to, computations of, and outputs from logic programs are treated later.

Notice that this style of semantic analysis of communication does not give either speaker or auditor direct access to inspect or modify the models constituting the other's state of implicit knowledge. Rather, all such access is mediated by the utterance of explicit logical formulae. Also, notice that there is no attempt to construct a *unique* model to represent a state of knowledge, or the information communicated by an utterance. Rather, an increase in implicit knowledge is represented by a reduction in the variability of members of a set of models, any one of which might represent the real state of the world. Unless the semantic-consequence relation of a semantic system is very easy to compute—which it seldom is—the difference between implicit knowledge and effectively utterable explicit knowledge can be quite significant. The *proof systems* of Section 3.1 help describe a way in which implicit knowledge is made explicit, and yield a rough description of the computations of logic programs.

The preceding scheme for representing communication of knowledge deals naturally with a sequence of utterances, by iterating the process of shrinking the auditor's set of models. There is no provision, however, for analyzing any sort of interactive *dialogue*, other than as a pair of formally unrelated monologues. The *query systems* of the next section introduce a primitive sort of interactive question-answering dialogue.

## 2.2 Query Systems, Questions and Answers

Semantic systems and semantic-consequence relations are conventional subjects for logical investigation. They suffice for discussions of the truth of a formula and the validity of the inference of a new formula from a given set of formulae. In order to analyze the relation between input to a logic program and the corresponding output, we need a formal basis for discussing questions and their answers. Mathematical logicians have given very little attention to this branch of logic—one exception is the formal treatment by Belnap and Steel [BS76]. *Query systems* are an abstraction of the common formal schema from a number of instances of question-answer domains defined by Belnap and Steel.

**Definition 2.2.1.** A *query system* is a system $\mathcal{Q} = \langle \mathbf{F}, \mathbf{Q}, ?\!\!- \rangle$, where

1. $\mathbf{F}$ is a set of *logical formulae*
2. $\mathbf{Q}$ is a set of *questions*
3. $?\!\!-$ is a relation on $\mathbf{Q} \times \mathbf{F}$

Questions, like formulae, are normally finite syntactic objects, and the set of all questions is normally computable, but we allow exceptions to the normal case.

$Q \mathrel{?\!\!-} F$ is intended to mean that $F$ is an *answer* to $Q$. $?\!\!-$ is intended only to determine the acceptable form for an answer to a question, *not* to carry any information about *correctness* of an answer. For example, it is reasonable to say that '$2 + 2 = 5$' is an incorrect answer to 'what is $2+2$?', while '$2+2=2^2$' is correct, but not an answer. The correctness or incorrectness of an answer is evaluated semantically with respect to explicit knowledge.

**Definition 2.2.2.** Let $\mathcal{Q} = \langle \mathbf{F}_Q, \mathbf{Q}, ?\!\!- \rangle$ be a query system, and let $\mathcal{S} = \langle \mathbf{F}_S, \mathbf{M}, \models \rangle$ be a semantic system with $\mathbf{F}_Q \subseteq \mathbf{F}_S$.

$Q \mathrel{?\!\!-} \mathbf{T} \models F$ means that $F \in \mathbf{F}_Q$ is a *semantically correct* answer to $Q \in \mathbf{Q}$ for explicit knowledge $\mathbf{T} \subseteq \mathbf{F}_S$, defined by

$$Q \mathrel{?\!\!-} \mathbf{T} \models F \text{ if and only if } Q \mathrel{?\!\!-} F \text{ and } \mathbf{T} \models F$$

A question $Q \in \mathbf{Q}$ is *semantically answerable* for explicit knowledge $\mathbf{T} \subseteq \mathbf{F}_S$ if and only if there exists a formula $F \in \mathbf{F}_Q$ such that $F$ is a semantically correct answer to $Q$ in $\mathbf{T}$.

Meseguer [Mes89, Mes92] proposes a different notion of question answering, in which a question is a formula, and an answer is a proof (in an abstract notation omitting many details) of the formula. (This is an interesting twist on the *formulae as types* concept [How80, Tai67], which is more usually applied by letting a program specification be a formula, and a program be a proof of the formula [CAB$^+$86].)

Several interesting query systems may be defined for the Shallow Implicational Calculus.

**Example 2.2.3.** Let **imp** be a new formal symbol, and let $\mathbf{F}_{Sh}$ be the set of formulae in SIC defined in Example 2.1.2. Let

$$\mathbf{Q}_{S1} = \{\mathbf{imp}(F) : F \in \mathbf{At}\}$$

Define the relation $?\!\!-_{S1} \subseteq \mathbf{Q}_{S1} \times \mathbf{F}_{Sh}$ by

$$\mathbf{imp}(c) \mathrel{?\!\!-_{S1}} (a \Rightarrow b) \text{ if and only if } a = c$$

Now $\langle \mathbf{F}_{Sh}, \mathbf{Q}_{S1}, ?\!\!-_{S1} \rangle$ is a query system representing the conceivable answers to questions of the form 'what atomic formula does $a$ imply?'

The query system of Example 2.2.3 above is susceptible to two sorts of answers that may be intuitively unsatisfying. First, in a state of knowledge in which an atomic formula $b$ is known to be true, $(a \Rightarrow b)$ is a correct answer to questions $\mathbf{imp}(a)$ for all atomic formulae $a$. This problem may be avoided by considering states of knowledge in which only implications are known, or it may be addressed by changing the underlying semantic system to one with a *relevant* interpretation of implication [AB75]. Second, $(a \Rightarrow a)$ is a correct answer to the question $\mathbf{imp}(a)$. $(a \Rightarrow a)$ is a *tautology*, that is, it holds in all models, so it cannot give any information about a state of knowledge. We could define a new query system, in which only nontautologies are considered to be answers. Since, for most useful logics, the detection of tautologies ranges from intractable to impossible, such a technique is generally unsatisfying. A better approach is to let a question present a set of atomic formulae that must not be used in an answer, since the questioner considers them to be insufficiently informative. We may find later that certain nontautological formulae are uninformative for various reasons, and this technique reserves the flexibility to handle those cases.

**Example 2.2.4.** Let **rest-imp** be a new formal symbol, and let

$$\mathbf{Q}_{S2} = \{\mathbf{rest\text{-}imp}(a, \mathbf{A}) : a \in \mathbf{At} \text{ and } \mathbf{A} \subseteq \mathbf{At}\}$$

Define the relation $?\!\!-_{S2} \subseteq \mathbf{Q}_{S2} \times \mathbf{F}_{Sh}$ by

$$\mathbf{rest\text{-}imp}(c, \mathbf{C}) \ ?\!\!-_{S1} (a \Rightarrow b) \text{ if and only if } a = c \text{ and } b \notin \mathbf{C}$$

Now $\langle \mathbf{F}_{Sh}, \mathbf{Q}_{S1}, ?\!\!-_{S1} \rangle$ is a query system representing the conceivable answers to questions of the form 'what atomic formula not in $\mathbf{A}$ does $a$ imply?'

The new query system of Example 2.2.4 may be used very flexibly to guide answers toward the most informative implications of an atomic formula $a$. If the explicit knowledge available to the auditor to answer questions is finite, then there are only a finite number of atomic formulae that can appear in an answer, so the sets of prohibited formulae may simply be listed. In more sophisticated languages than SIC, we need some sort of finite notation for describing large and even infinite sets of prohibited answers.

Query systems allow a further enrichment of the analysis of communication. Once a speaker has communicated some implicit knowledge $\mathbf{K}$ to an auditor by uttering formulae, a *questioner* (sometimes, but not always, identical with the speaker) may ask a question $Q$, which the auditor tries to answer by discovering a formula $F$ such that $Q \ ?\!\!- F$ ($F$ is an answer to the question $Q$), and $F \in \mathbf{Theory}(\mathbf{K})$ ($Q$ is correct according to the implicit knowledge $\mathbf{K}$).

So, given a set $\mathbf{T}$ of formulae expressing the knowledge $\mathbf{Models}(\mathbf{T})$, a question $Q$ provides an additional constraint on the search for a formula $F$ such that $\mathbf{T} \models F$, to ensure that $Q \ ?\!\!- F$ as well. In many cases, there is more than one correct answer $F$ such that $Q \mathrel{?\!\!-} \mathbf{T} \models F$. Depending on the context, the questioner may want a single answer chosen nondeterministically from the set of correct answers, or a best answer under some criterion. The case where the questioner wants a list of all answers may be modelled by representing that list by a single formula giving the conjunction of all the list elements. A particularly useful criterion for best answer uses the logical consequence relation.

**Definition 2.2.5.** Let $\mathcal{Q} = \langle \mathbf{F}_Q, \mathbf{Q}, \mathrel{?\!\!-} \rangle$ be a query system, and let $\mathcal{S} = \langle \mathbf{F}_S, \mathbf{W}, \models \rangle$ be a semantic system with $\mathbf{F}_Q \subseteq \mathbf{F}_S$. $F$ is a *consequentially strongest* correct answer to the question $Q$ for explicit knowledge $\mathbf{T}$ if and only if

1. $Q \mathrel{?\!\!-} \mathbf{T} \models F$
2. for all $G \in \mathbf{F}_Q$, whenever $Q \mathrel{?\!\!-} \mathbf{T} \models G$, then $\{F\} \models G$

Consequentially strongest answers are not necessarily unique, but all consequentially strongest answers are semantically equivalent. Notice that the comparison of strength for two answers $F$ and $G$ is done without taking into account the knowledge $\mathbf{T}$. That is, we require $\{F\} \models G$, rather than $\mathbf{T} \cup \{F\} \models G$. This makes sense because $\mathbf{T}$ is known to the auditor, but not necessarily to the questioner. Even if the questioner knows $\mathbf{T}$, he may not be able to derive its consequences. The whole purpose of the communication between questioner and auditor is to give the questioner the benefit of the auditor's knowledge and inferential power. So, the value of an answer to the questioner must be determined independently of the knowledge used by the auditor in its construction (the alternative form $\mathbf{T} \cup \{F\} \models G$ holds trivially by monotonicity, so it carries no information anyway).

In order to illustrate the use of consequentially strongest answers, we extend SIC to deal with conjunctions of implications.

**Example 2.2.6.** Expand the formulae of SIC to the set

$$\mathbf{F}_{Sc} = \mathbf{F}_{Sh} \cup \{F_1 \wedge \cdots \wedge F_m : F, G \in \mathbf{F}_{Sh}\}$$

of formulae in the *Shallow Implicational-Conjunctive Calculus* (SICC). The semantic systems and proof systems of Examples 2.1.2, 3.1.3, 3.1.2 extend in the natural way to deal with conjunctive formulae. Let **conj-imp** be a new formal symbol, and let

$$\mathbf{Q}_{Sc} = \{\textbf{conj-imp}(a) : a \in \mathbf{At}\}$$

Define the relation $\mathrel{?\!\!-}_{Sc} \subseteq \mathbf{Q}_{Sc} \times \mathbf{F}_{Sc}$ by

$\textbf{conj-imp}(c) \mathrel{?\!\!-}_{Sc} (a_1 \Rightarrow b_1) \wedge \cdots \wedge (a_m \Rightarrow b_m)$ if and only if $a_i = c$ for all $i \leq m$

Now $\langle \mathbf{F}_{Sc}, \mathbf{Q}_{Sc}, \mathrel{?\!\!-}_{Sc} \rangle$ is a query system representing the conceivable answers to questions of the form 'what are some atomic formulae implied by

$a$?' A consequentially strongest answer to **conj-imp**($a$) is a conjunction of *all* of the implications with hypothesis $a$ that hold in a given state of knowledge.

The concept of consequentially strongest answers is particularly helpful in systems where a potentially infinite answer is produced incrementally. The entire infinite answer may often be read as an infinite conjunction of finite formulae, and the requirement of consequentially strongest answers guarantees that the incremental production of the answer does not stop prematurely, before all available information is expressed.

In logic programming systems, the *user* of a program plays the part of the questioner. The input is the question, and the output is the answer, if any, discovered and proved by the processor/auditor. This scenario allows the knowledge resources of a programmer/speaker to be combined with the deductive powers of a processor/auditor, in order to answer questions from the user/questioner.

## 2.3 Examples of Logic Programming Languages

Now we can design a wide variety of logic programming languages, by defining appropriate semantic systems and query systems.

### *2.3.1 Programming in First-Order Predicate Calculus*

Several logic programming systems, particularly Prolog and Relational Databases, are essentially sublanguages of a general language for logic programming in FOPC.

**Definition 2.3.1 ([And86, Kle52, Gal86] ).** Let $\mathbf{V}$ be a countably infinite set. Members of $\mathbf{V}$ are called *variables*, and are written $u, v, w, x, y, z$, sometimes with subscripts.

Let $\mathbf{Fun}_i$ be a countably infinite set for each $i \geq 0$, with $\mathbf{Fun}_i$ and $\mathbf{Fun}_j$ disjoint when $i \neq j$. Members of $\mathbf{Fun}_i$ are called *function symbols of arity i*, and are written $f, g, h$, sometimes with subscripts. A function symbol of arity 0 in $\mathbf{Fun}_0$ is called a *constant*, and may be written $a, b, c, d, e$.

Let $\mathbf{Pred}_i$ be a countably infinite set for each $i \geq 0$, with $\mathbf{Pred}_i$ and $\mathbf{Pred}_j$ disjoint when $i \neq j$, $\mathbf{Pred}_i$ and $\mathbf{Fun}_j$ disjoint for all $i$ and $j$. Members of $\mathbf{Pred}_i$ are called *predicate* or *relation symbols of arity i*, and are written $P, Q, R$, sometimes with subscripts. A predicate symbol of arity 0 in $\mathbf{Pred}_0$ is called a *propositional symbol*, and is closely analogous to an atomic propositional formula in $\mathbf{At}$ as used in Example 2.1.2.

The set $\mathbf{T}_P$ of *terms* in FOPC is defined inductively as the least set such that:

1. if $x \in \mathbf{V}$ then $x \in \mathbf{T}_P$

2. if $a \in \mathbf{Fun}_0$ then $a \in \mathbf{T}_P$

3. if $f \in \mathbf{Fun}_i$ for some $i > 0$ and $t_1, \ldots, t_i \in \mathbf{T}_P$, then $f(t_1, \ldots, t_i) \in \mathbf{T}_P$

Terms are intended to represent objects in some universe of discourse. $f(t_1, \ldots, t_i)$ is intended to represent the result of applying the function denoted by $f$ to the objects represented by $t_1, \ldots, t_i$. We often take the liberty of writing binary function application in infix notation. For example, if $+ \in \mathbf{Fun}_2$ we may write $(t_1 + t_2)$ for $+(t_1, t_2)$. A *ground term* is a term containing no variables.

The set $\mathbf{F}_P$ of *formulae* in FOPC, is defined inductively as the least set such that:

1. True, False $\in \mathbf{F}_P$

2. if $P \in \mathbf{Pred}_0$, then $P \in \mathbf{F}_P$

3. if $P \in \mathbf{Pred}_i$ for some $i > 0$ and $t_1, \ldots, t_i \in \mathbf{T}_P$, then $P(t_1, \ldots, t_i) \in \mathbf{F}_P$

4. if $A, B \in \mathbf{F}_P$, then $(A \wedge B), (A \vee B), (A \Rightarrow B), (\neg A) \in \mathbf{F}_P$

5. if $A \in \mathbf{F}_P$ and $x \in \mathbf{V}$, then $(\exists x : A), (\forall x : A) \in \mathbf{F}_P$

Formulae in FOPC are intended to represent assertions about the objects represented by their component terms. True and False are the trivially true and false assertions, respectively. $P(t_1, \ldots, t_i)$ represents the assertion that the relation denoted by $P$ holds between $t_1, \ldots, t_i$. $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(\neg A)$ represent the usual conjunction, disjunction, implication, and negation. $(\exists x : A)$ represents 'there exists $x$ such that $A$,' and $(\forall x : A)$ represents 'for all $x$ $A$.' Parentheses may dropped when they can be inferred from normal precedence rules.

In a more general setting, it is best to understand $\mathbf{Fun}_i$ and $\mathbf{Pred}_i$ as parameters giving a *signature* for first-order logic, and let them vary to produce an infinite class of predicate calculi. For this chapter, we may take $\mathbf{Fun}_i$ and $\mathbf{Pred}_i$ to be arbitrary but fixed. In many texts on mathematical logic, the language of FOPC includes a special binary predicate symbol '$\doteq$' for equality. We follow the Prolog tradition in using the *Pure First-Order Predicate Calculus*, without equality, and referring to it simply as FOPC.

The intended meanings of FOPC formulae sketched above are formalized by the traditional semantic system defined below. First, we need a set of models for FOPC.

**Definition 2.3.2 ([And86, Kle52, Gal86] ).** Let $\mathbf{U}$ be an infinite set, called the *universe*. Let $U \subseteq \mathbf{U}$.
A *variable assignment* over $U$ is a function $\nu : \mathbf{V} \rightarrow U$.
A *predicate assignment* over $U$ is a function $\rho : \bigcup\{\mathbf{Pred}_i : i \geq 0\} \rightarrow \bigcup\{\mathbf{2}^{(U^i)} : i \geq 0\}$

such that $P \in \mathbf{Pred}_i$ implies $\rho(P) \subseteq U^i$ for all $i \geq 0$.
A *function assignment* over $U$ is a function $\tau : \bigcup\{\mathbf{Fun}_i : i \geq 0\} \rightarrow \bigcup\{U^{(U^i)} : i \geq 0\}$ such that $f \in \mathbf{Fun}_i$ implies $\tau(f) : U^i \rightarrow U$ for all $i \geq 0$.

If $U \subseteq \mathbf{U}$, $\tau$ is a function assignment over $U$, and $\rho$ is a predicate assignment over $U$, then $\langle U, \tau, \rho\rangle$ is a *model of FOPC*. $\mathbf{M}_P$ is the set of all models of FOPC.

Notice that FOPC models assign values to function and predicate symbols, but not to variables. The exclusion of variable assignments from models is the only technically significant distinction between variables and constant symbols. A function assignment and a variable assignment together determine a valuation of terms. With a predicate assignment, they also define a valuation of formulae.

**Definition 2.3.3 ([And86, Kle52, Gal86] ).** Let $\tau$ be a function assignment over $U$, $\nu$ a variable assignment over $U$. The *term valuation* $\tau_\nu : \mathbf{T}_P \rightarrow U$ is defined inductively by

1. if $x \in \mathbf{V}$, then $\tau_\nu(x) = \nu(x)$
2. if $f \in \mathbf{Fun}_i$, and $t_1, \ldots, t_i \in \mathbf{T}_P$, then $\tau_\nu(f(t_1, \ldots, t_i)) = \tau(f)(\tau_\nu(t_1), \ldots, \tau_\nu(t_i))$

In addition, let $\rho$ be a predicate assignment over $U$. The *formula valuation* $\rho_{\tau,\nu} : \mathbf{F}_P \rightarrow \{0, 1\}$ is defined inductively by

1. $\rho_{\tau,\nu}(\mathrm{False}) = 0$
2. $\rho_{\tau,\nu}(\mathrm{True}) = 1$
3. if $P \in \mathbf{Pred}_i$, and $t_1, \ldots, t_i \in \mathbf{T}_P$, then $\rho_{\tau,\nu}(P(t_1, \ldots, t_i)) = 1$ if and only if $\langle \tau_{\tau,\nu}(t_1), \ldots, \tau_{\tau,\nu}(t_i)\rangle \in \rho(P)$
4. $\rho_{\tau,\nu}(A \wedge B) = 1$ if and only if $\rho_{\tau,\nu}(A) = 1$ and $\rho_{\tau,\nu}(B) = 1$
5. $\rho_{\tau,\nu}(A \vee B) = 1$ if and only if $\rho_{\tau,\nu}(A) = 1$ or $\rho_{\tau,\nu}(B) = 1$
6. $\rho_{\tau,\nu}(A \Rightarrow B) = 1$ if and only if $\rho_{\tau,\nu}(A) = 0$ or $\rho_{\tau,\nu}(B) = 1$
7. $\rho_{\tau,\nu}(\neg A) = 1$ if and only if $\rho_{\tau,\nu}(A) = 0$
8. $\rho_{\tau,\nu}(\exists x : A) = 1$ if and only if $\rho_{\tau,\nu'}(A) = 1$ for some $\nu'$ such that $y \neq x$ implies $\nu(y) = \nu'(y)$
9. $\rho_{\tau,\nu}(\forall x : A) = 1$ if and only if $\rho_{\tau,\nu'}(A) = 1$ for all $\nu'$ such that $y \neq x$ implies $\nu(y) = \nu'(y)$

Now, we may define an appropriate semantic system for FOPC.

**Definition 2.3.4.** The classical semantic system for FOPC is $\langle \mathbf{F}_P, \mathbf{M}_P, \models_P \rangle$, where $\langle U, \tau, \rho \rangle \models_P F$ if and only if $\rho_{\tau,\nu}(F) = 1$ for all variable assignments $\nu$ over $U$.

FOPC is particularly well suited to defining relations, letting variables stand for the parameters in the relations. For such purposes, it is important to distinguish between uses of variables that are *bound* by the quantifiers $\exists$ and $\forall$, and those that are *free* to be used as relational parameters.

**Definition 2.3.5 ([And86, Kle52, Gal86] ).** An occurrence of a variable $x$ in a formula is *bound* if and only if it is located within a subformula of the form $(\exists x : F)$ or the form $(\forall x : F)$. An occurrence of a variable $x$ in a formula is *free* if and only if it is not bound.

A *sentence* is a formula with no free occurrences of variables. If $F \in \mathbf{F}_P$ is a formula, and all free occurrences of variables in $F$ are among $x_1, \ldots, x_i$, then the sentence $(\forall x_1 : \cdots \forall x_i : F)$ is a *closure* of $F$. It is easy to see that $F$ is semantically equivalent to each of its closures.

Let $x_1, \ldots, x_i$ be a list of variables with no repetitions, and let $t_1, \ldots, t_i \in \mathbf{T}_P$. $F[t_1, \ldots, t_i/x_1, \ldots, x_i]$ is the formula that results from substituting the term $t_j$ for every free occurrence of the variable $x_j$ in the formula $F$, for each $j$, $1 \leq j \leq i$, renaming bound variables of $F$ when necessary so that the variables of $t_1, \ldots, t_i$ are free in the result [And86, Kle52, Gal86] . When $G = F[t_1, \ldots, t_i/x_1, \ldots, x_i]$, we say that $G$ is an *instance* of $F$, and that $F$ is *more general* than $G$. These relations apply naturally to terms as well as formulae.

$G$ is a *variable renaming* of $F$ if and only if $G = F[y_1, \ldots, y_i/x_1, \ldots, x_i]$, for some list of variables $y_1, \ldots, y_i$ with no repetitions (equivalently, $G$ is an instance of $F$ and $F$ is an instance of $G$, since we get $F = G[x_1, \ldots, x_i/y_1, \ldots, y_i]$).

It is very natural to think of a query system in which we may ask what substitutions for the free variables of a formula make it true.

**Definition 2.3.6 ([BS76]).** Let **what** be a new formal symbol. Let

$$\mathbf{Q}_P = \{(\textbf{what } x_1, \ldots, x_i : F) : F \in \mathbf{F}_P\}$$

Define the relation $?\!\!-_P \subseteq \mathbf{Q}_P \times \mathbf{F}_P$ by

$$(\textbf{what } x_1, \ldots, x_i : F) ?\!\!-_P G$$

if and only if

$$G = F[t_1, \ldots, t_i/x_1, \ldots, x_i] \text{ for some } t_1, \ldots, t_i \in \mathbf{T}_P$$

Now $\langle \mathbf{F}_P, \mathbf{Q}_P, ?\!\!-_P \rangle$ is a query system representing the conceivable single answers to questions of the form 'for what terms $t_1, \ldots, t_i$ does $F[t_1, \ldots, t_i/x_1, \ldots, x_i]$ hold?'

The query system above has a crucial role in the profound theoretical connections between logic and computation. For each finite (or even semicomputable) set $\mathbf{T} \subseteq \mathbf{F}_P$ of formulae, and each question $Q = (\textbf{what}\ x_1, \ldots, x_i : F)$, the set

$$\{\langle t_1, \ldots, t_i \rangle : Q \mathrel{?\!\!-}_P \mathbf{T} \models_P F[t_1, \ldots, t_i/x_1, \ldots, x_i]\}$$

is semicomputable. If we use some constant symbol $c \in \mathbf{Fun}_0$ to represent the number 0, some unary function symbol $f \in \mathbf{Fun}_1$ to represent the successor function, and some binary predicate symbol $E$ to represent the equality relation, then we may define all semicomputable sets of integers by formulae in a simple and natural way. We let $\mathbf{R} \subseteq \mathbf{F}_P$ be a finite set of postulates for *Robinson's Arithmetic* (system $Q$ of [MRT53])—a primitive theory sufficient for deriving answers to all addition and multiplication problems. Then *every* semicomputable set is of the form

$$\{j : (\textbf{what}\ x : F) \mathrel{?\!\!-}_P \mathbf{R} \models_P F[f^j(c)/x]\}$$

for some formula $F \in \mathbf{F}_P$. Similarly, every partial computable function $\phi$ may be defined by choosing an appropriate formula $F \in \mathbf{F}_P$, and letting $\phi(i)$ be the unique number $j$ such that

$$(\textbf{what}\ y : F[f^i(c)/x]) \mathrel{?\!\!-}_P \mathbf{R} \models_P F[f^i(c), f^j(c)/x, y]$$

Notice that the FOPC questions $(\textbf{what}\ x_1, \ldots, x_i : F)$ do not allow trivial tautological answers, such as the correct answer $(a \Rightarrow a)$ to the question $\textbf{imp}(a)$ ('what atomic formula does $a$ imply?', Example 2.2.3, Section 2.2). In fact, $(\textbf{what}\ x_1, \ldots, x_i : F)$ has a tautological answer if and only if $F$ is a tautology. FOPC questions avoid this problem through the distinction between predicate symbols and function symbols. When we try to find an answer $F[t_1, \ldots, t_i/x_1, \ldots, x_i]$ to the question $(\textbf{what}\ x_1, \ldots, x_i : F)$, the information in the question $(\textbf{what}\ x_1, \ldots, x_i : F)$ is carried largely through predicate symbols, while the information in the answer $F[t_1, \ldots, t_i/x_1, \ldots, x_i]$ is carried entirely by the function symbols in $t_1, \ldots, t_i$, since the predicate symbols in the formula $F$ are already fixed by the question. It is the syntactic incompatibility between the formula given in the question and the terms substituted in by the answer that prevents tautological answers. Suppose that FOPC were extended with a symbol **choose**, where $(\textbf{choose}\ x : F)$ is a term such that $(\exists x : F)$ implies $F[(\textbf{choose}\ x : F)/x]$. Then $F[(\textbf{choose}\ x : F)/x]$ would be a trivially (but not quite tautologically) correct answer to $(\textbf{what}\ x : F)$ except when no correct answer exists.

The absence of trivial tautological answers does *not* mean that all answers to FOPC questions are equally useful. In some cases a question has a most general semantically correct answer. This provides a nice syntactic way to recognize certain consequentially strongest answers, and a useful characterization of all answers even when there is no consequentially strongest answer.

**Proposition 2.3.7.** *If $G'$ is an instance of $G$, then $G'$ is a semantic consequence of $G$ ($G \models_P G'$). It follows immediately that if $G$ is a semantically correct answer to $Q \in \mathbf{Q}_P$ ($Q \mathrel{?\!\!-}_P \mathbf{T} \models_P G$), then $G'$ is also a semantically correct answer ($Q \mathrel{?\!\!-}_P \mathbf{T} \models_P G'$).*

*If $G'$ is a variable renaming of $G$, then they are semantically equivalent.*

**Definition 2.3.8.** Let $\mathbf{T} \subseteq \mathbf{F}_P$ be a set of formulae, and let $Q = (\mathbf{what}\ x_1, \ldots, x_i : F) \in \mathbf{Q}_P$ be a question. $G$ is a *most general* answer to $Q$ for explicit knowledge $\mathbf{T}$ if and only if

1. $Q \mathrel{?\!\!-}_P \mathbf{T} \models_P G$
2. for all $G_0 \in \mathbf{F}_P$, if $G$ is an instance of $G_0$ and $Q \mathrel{?\!\!-}_P \mathbf{T} \models_P G_0$, then $G_0$ is a variable renaming of $G$.

A set $\mathbf{A} \subseteq \mathbf{F}_P$ of formulae is a *most general set* of correct answers to $Q$ for $\mathbf{T}$ if and only if

1. each formula $F \in \mathbf{A}$ is a most general answer to $Q$ for $\mathbf{T}$
2. for all formulae $G \in \mathbf{F}_P$, if $Q \mathrel{?\!\!-}_P \mathbf{T} \models_P G$, then $G$ is an instance of some $F \in \mathbf{A}$
3. for all formulae $F_1, F_2 \in \mathbf{A}$, if $F_2$ is an instance of $F_1$, then $F_2 = F_1$

It is easy to see that for each question $Q \in \mathbf{Q}_P$ and set of formulae $\mathbf{T} \subseteq \mathbf{F}_P$ there is a most general set of correct answers (possibly empty or infinite). Furthermore, the most general set of correct answers is unique up to variable renaming of its members.

Notice that it is very easy to generate all correct answers to a question in $\mathbf{Q}_P$ from the most general set—they are precisely the instances of its members. If $Q$ has a consequentially strongest answer, then it has a consequentially strongest answer that is also most general. If the most general set of correct answers is the singleton set $\{F\}$, then $F$ is a consequentially strongest answer.

**Example 2.3.9.** Let $G \in \mathbf{Pred}_2$ be a binary predicate symbol, where $G(t_1, t_2)$ is intended to assert that $t_1$ is strictly larger than $t_2$. Suppose that objects in the universe have left and right portions, and that $l, r \in \mathbf{Fun}_1$

denote the operations that produce those portions. A minimal natural state of knowledge about such a situation is

$$\mathbf{T}_0 = \{\forall x : G(x, l(x)) \wedge G(x, r(x))\}$$

A natural question is $Q = (\mathbf{what}\ x, y : G(x, y))$. The most general set of answers is

$$\mathbf{A}_0 = \{G(x, l(x)),\ G(x, r(x))\}$$

Other answers include $G(l(x), r(l(x)))$, $G(r(x), l(r(x)))$, etc.

$\mathbf{T}_0$ has only the specific knowledge that a whole is larger than its portions, but not the general knowledge that the relation $G$ is a strict ordering relation. Let

$$\mathbf{T}_1 = \mathbf{T}_0 \cup \{\forall x, y, z : (G(x, y) \wedge G(y, z)) \Rightarrow G(x, z),\ \forall x, y : \neg(G(x, y) \wedge G(y, x))\}$$

For this extended knowledge, the most general set of answers to $Q$ is the infinite set

$$\mathbf{A}_1 = \mathbf{A}_0 \cup \{G(x, l(l(x))),\ G(x, l(r(x))),\ G(x, r(r(x))),\ G(x, l(l(l(x)))), \ldots\}$$

The first formula added to $\mathbf{T}_1$ above, which expresses the transitivity of ordering, leads to the additional answers.

In some cases, it is convenient to allow conjunctions of answers to questions in $\mathbf{Q}_P$ to be conjoined into a single answer.

**Definition 2.3.10.** Let **conj-what** be a new formal symbol. Let

$$\mathbf{Q}_{P\wedge} = \{(\mathbf{conj\text{-}what}\ x_1, \ldots, x_i : F) : F \in \mathbf{F}_P\}$$

Define the relation $?\!\!-_{P\wedge} \subseteq \mathbf{Q}_{P\wedge} \times \mathbf{F}_P$ by

$$(\mathbf{conj\text{-}what}\ x_1, \ldots, x_i : F)\ ?\!\!-_{P\wedge}\ G$$

if and only if

$$G = F[t_1^1, \ldots, t_i^1/x_1, \ldots, x_i] \wedge \cdots \wedge F[t_1^n, \ldots, t_i^n/x_1, \ldots, x_i] \text{ for some } t_1^1, \ldots, t_i^n \in \mathbf{T}_P$$

Now $\langle \mathbf{F}_{P\wedge}, \mathbf{Q}_{P\wedge}, ?\!\!-_{P\wedge}\rangle$ is a query system representing the conceivable conjunctive answers to questions of the form 'for what terms $t_1, \ldots, t_i$ does $F[t_1, \ldots, t_i/x_1, \ldots, x_i]$ hold?'

Answers to $(\mathbf{conj\text{-}what}\ x_1, \ldots, x_i : F)$ are precisely conjunctions of answers to $(\mathbf{what}\ x_1, \ldots, x_i : F)$. $(\mathbf{conj\text{-}what}\ x_1, \ldots, x_i : F)$ may have a consequentially strongest answer even though $(\mathbf{what}\ x_1, \ldots, x_i : F)$ does not. In particular, whenever the most general set of answers to $(\mathbf{what}\ x_1, \ldots, x_i : F)$ is finite, the conjunction of those answers is a consequentially strongest answer to $(\mathbf{conj\text{-}what}\ x_1, \ldots, x_i : F)$.

### 2.3.2 Prolog

The most famous programming language associated with logic programming, and the one that instigated scientists to study logic programming as a specialty within computer science, is Prolog [Kow74, vK76], the creation of Kowalski and Colmerauer. The name 'Prolog' is usually associated with a group of very similar programming languages based on logic programming in Horn clauses—a particular sublanguage of the First-Order Predicate Calculus. Prolog as it is actually used deviates from pure logic programming in several ways: it fails to produce some logically entailed answers; in rare cases it produces logically incorrect answers; and it contains constructs that are not definable naturally in FOPC, and which are normally understood in conventional imperative ways. Furthermore, the criteria by which Prolog chooses one of several possible answers cannot be explained naturally in terms of the semantics of FOPC. The discrepancy between Prolog and Horn-clause logic programming is closely comparable to the discrepancy between Lisp and the purely functional language based on the lambda calculus that is sometimes called 'pure lazy Lisp.' In spite of the discrepancies, the best way to understand Prolog is to conceive of it as an approximation to, and extension of, a Horn-clause logic programming language.

The essential idea behind Prolog is to find correct answers to predicate calculus questions in $\mathbf{Q}_P$ of Section 2.3.1 above. In principle, all such answers are computable. Currently known implementation techniques require fairly stringent conditions on the sets of formulae that may be used as explicit knowledge for question answering, and on the questions that may be asked, in order to search for and generate proofs in a relatively simple and disciplined way. Prolog is based on the restriction of programs to sets of *Horn clauses*, and questions to simple conjunctions of positive atomic formulae.

**Definition 2.3.11.** A formula $F \in \mathbf{F}_P$ is a *Horn clause* if and only if it is in one of the forms

1. $F = (R_1(t_{1,1}, \ldots, t_{1,i_1}) \wedge \cdots \wedge R_m(t_{m,1}, \ldots, t_{m,i_m}) \Rightarrow P(u_1, \ldots, u_j))$
2. $F = (R_1(t_{1,1}, \ldots, t_{1,i_1}) \wedge \cdots \wedge R_m(t_{m,1}, \ldots, t_{m,i_m}) \Rightarrow \mathrm{False})$
3. $F = (\mathrm{True} \Rightarrow P(u_1, \ldots, u_j))$
4. $F = (\mathrm{True} \Rightarrow \mathrm{False})$

where $R_1, \ldots, R_m, P$ are predicate symbols, and $t_{1,1}, \ldots, t_{m,i_m}, u_1, \ldots, u_j$ are terms.

A *pure Prolog program* is a finite set of Horn clauses.

A *pure Prolog input* is a question of the form

$$(\textbf{what } x_1, \ldots, x_i : P_1(t_{1,1}, \ldots, t_{1,i_1}) \wedge \cdots \wedge P_m(t_{m,1}, \ldots, t_{m,i_m}))$$

where $x_1, \ldots, x_i$ are precisely the free variables of $P_1(t_{1,1}, \ldots, t_{1,i_1}) \wedge \cdots \wedge P_m(t_{m,1}, \ldots, t_{m,i_m})$.

In order to promote readability of programs, within the limitations of older keyboards and printers, typical Prolog implementations replace the conventional logical symbol '$\wedge$' by ';', they write the arguments to implications in the reverse of the usual order and replace the symbol '$\Leftarrow$' by ':–'. Also, they denote the symbol 'False' in the conclusion of an implication, and 'True' in the hypothesis, by the empty string. Predicate symbols are written in lower case, and variables are written in upper case. So, the four forms of Horn clause in actual Prolog programs look like

1. $p(u_1, \ldots, u_j) :- r_1(t_{1,1}, \ldots, t_{1,i_1}); \cdots ; r_m(t_{m,1}, \ldots, t_{m,i_m})$
2. $:- r_1(t_{1,1}, \ldots, t_{1,i_1}); \cdots ; r_m(t_{m,1}, \ldots, t_{m,i_m})$
3. $p(u_1, \ldots, u_j) :-$
4. $:-$

Since a question always requests substitutions for all free variables, the header '**what** $x_1, \ldots, x_i$' is omitted and the question is abbreviated in the form

$$r_1(t_{1,1}, \ldots, t_{1,i_1}); \cdots ; r_m(t_{m,1}, \ldots, t_{m,i_m})$$

Since the substitution in an answer of the form

$$(R_1(t_{1,1}, \ldots, t_{1,i_1}) \wedge \cdots \wedge R_m(t_{m,1}, \ldots, t_{m,i_m}))[s_1, \ldots, s_i/x_1, \ldots, x_i]$$

completely determines the answer, actual Prolog output presents only the substitution, in the form

$$x_1 = s_1; \cdots ; x_i = s_i$$

These notational variations and abbreviations have a substantial impact on the way Prolog programs, inputs, and outputs *look*, but they are completely transparent to the logical meaning.

When a pure Prolog input is presented to a pure Prolog program, all possible answers may be derived systematically by treating each clause of form 1 as a recursive part of a definition of the procedure $P$ in terms of calls to the procedures $R_1, \ldots, R_m$. Because the same predicate symbol $P$ may appear in the conclusion of more than one clause, each clause normally provides only a part of the definition of $P$. Clauses of form 3 are nonrecursive parts of the definition of a procedure $P$. Clauses of form 2 are somewhat peculiar: they act like extra hardcoded parts of the input. Clauses of form 4 are useless in programs, but allowed by the formal definition. Different implementations may or may not prohibit the degenerate forms. Prolog tries to choose more general answers instead of their instances, but it does not guarantee that all answers produced are most general. An understanding of the precise correspondence of Prolog to the answering of pure Prolog input questions using pure Prolog programs requires a lot of detail that must wait until the chapter 'Horn Clause Logic Programming.'

A notable variation on Prolog is $\lambda$Prolog [NM90, NM88]. This language extends Predicate Calculus logic programming into the *Omega-Order Predicate Calculus,* also called *Type Theory* [And86]. Higher-order predicate calculi add variables ranging over predicates, quantification over such variables, and predicates that apply to other predicates. $\lambda$Prolog generalizes the Horn clauses of FOPC to the hereditary Harrop formulae of the Omega-Order Predicate calculus [MNPS91].

### *2.3.3 Relational Databases and Datalog*

Relational databases [Dat86, Cod70] were invented by Codd, completely independently of the development of Prolog and logic programming in the programming language community. Nonetheless, relational databases and their queries may be understood very naturally in terms of logic programming concepts. This point has been noted by the Prolog community, leading to the definition of Datalog [MW88], a variant of relational databases in the style of Prolog. Gallaire, Minker, and Nicolas have developed the concept of *deductive databases* [GMN84] to capture the logical content of relational databases and their variants. Reiter [Rei84] has shown how a logical view of databases has advantages of robustness under several useful generalizations of database functionality. My proposed approach to logic programming applies logic to programming languages in essentially the same way that Reiter applies logic to databases.

Like Prolog, relational database systems find correct answers to predicate calculus questions in $\mathbf{Q}_P$ of Section 2.3.1. The different natural assumptions and performance requirements of the database world lead to far more stringent restrictions on the sets of formulae that may be used for question answering. The questions, which correspond to database queries, are essentially unrestricted. Because of the stringent restrictions on knowledge formulae, which limit answers to a predetermined finite set of possibilities, actual relational database implementations do not deviate from pure logic programming, except by offering constructs that go beyond the logical definitions. On purely relational queries, they produce precisely all of the semantically correct answers.

**Definition 2.3.12.** A *pure relational database* is a finite set of formulae of the form $P(c_1, \ldots, c_i)$, where $P \in \mathbf{Pred}_i$ is a predicate symbol and

$c_1, \ldots, c_i \in \mathbf{Fun}_0$ are constant symbols.

A pure relational database as defined above is the natural logical view of the contents of a relational database system at any given time. The constant symbols are the objects that may appear in fields of relations in the database. Each predicate symbol represents one of the relations in the database. Each formula $P(c_1, \ldots, c_i)$ represents the presence of the tuple $\langle c_1, \ldots, c_i \rangle$ in the relation $P$.

Pure relational database query languages are equivalent to $\mathbf{Q}_P$—all queries of the form $(\mathbf{what}\ x_1, \ldots, x_i : F)$ are allowed. Because of the simplicity of the formulae in the database, restrictions on the queries are not required for tractable implementation. Notice the complementarity of Prolog and relational database query languages. Prolog allows relatively powerful Horn clauses as knowledge formulae, but restricts queries to conjunctions of atomic formulae. Relational database query languages restrict knowledge formulae to the very simple form of predicate symbols applied to constant symbols, but allow unrestricted FOPC **what** queries.

The simplicity of the formulae in a pure relational database guarantees that the set of answers to $(\mathbf{what}\ x_1, \ldots, x_i : F)$ is finite, and relational database query systems actually produce *all* the semantically correct answers. Equivalently, we may think of a relational database query system as producing the consequentially strongest answer (unique up to the order of the conjuncts) to $(\mathbf{conj\text{-}what}\ x_1, \ldots, x_i : F)$. The consequentially strongest answer to the **conj-what** form of the question is simply the conjunction of all the answers to the **what** form.

Datalog restricts queries to the Prolog form, but allows Horn clauses with no function symbols (no 'functors' in Prolog jargon) to be added to the formulae of the database to provide additional knowledge for the purpose of answering a given query. The Horn clauses are thought of as defining new relations to be used in the query, rather than as adding information to the database.

A variety of notations have been used for the expression of FOPC queries in relational database systems. These variant notations may look very different from FOPC at first glance, but in fact they are equivalent in querying power. The complete translations between notations are clumsy to define formally, so we consider the general principles behind the notation, and illustrate the translation with examples. I use Codd's language DSL ALPHA [Dat86, Cod71], often called *relational calculus*, for the database notation.

Instead of referring directly to objects in the universe $\mathbf{U}$, relational database languages generally refer to tuples in relations, because they correspond to records in a conventional file. Instead of the positional notation $P(t_1, \ldots, t_i)$, they give *domain names* $D_1, \ldots, D_i$ to the parameter positions. The name of a relation (predicate symbol) is treated as a variable ranging over the set of tuples in that relation. The value of the domain $D$ in an arbitrary tuple of the relation $P$ is denoted by the record notation $P.D$. $P.D = c$ means that the value in the domain $D$ of an unknown tuple in the relation $P$ is $c$, and $P.D = Q.E$ means that the value in the domain $D$ of some tuple in $P$ is the same as that in domain $E$ of some tuple in $Q$. Because of the absence of function symbols with arity greater than 0, this use of equality does not introduce the general capabilities of FOPC with equality, it merely captures the limited sort of equality information that is represented in pure FOPC by multiple occurrences of constants and variables. For example, if $P$ and $Q$ are binary predicates and $D, E, F, G$ are appropriate domain names, then the DSL ALPHA expression

$$P.D = c \wedge P.E = Q.F \wedge Q.G = d$$

is equivalent to the FOPC formula

$$P(c, x) \wedge Q(x, d)$$

Additional variables may be declared in DSL ALPHA to range over the tuples of a relation in the database, and these variables may be quantified with $\forall$ and $\exists$. So, in the presence of the declarations

$$\underline{\text{RANGE}}\, X\ P$$
$$\underline{\text{RANGE}}\, Y\ Q$$

which declare the variable $X$ to range over tuples of the relation $P$ and $Y$ to range over tuples of $Q$, the expression

$$\forall X(X.D = c \vee \exists Y(X.E = Y.F \wedge Y.G = d))$$

is equivalent to the FOPC formula

$$\forall x : (P(c, x) \vee Q(x, d))$$

Notice that the existential quantifier in the DSL ALPHA expression is left out of the FOPC formula, because one of its components is bound to the constant $d$, and the other to the $E$ component of the variable $X$, whose quantification occurs before that of $Y$. In general, a FOPC quantifier is required for each domain position of a quantified tuple in a DSL ALPHA expression that is not bound by equality either to a constant or to a component of a variable that has already been quantified. With the same RANGE declarations above, the DSL ALPHA expression

$$\forall X \exists Y(X.D = Y.G)$$

is equivalent to the FOPC formula

$$\forall x_1 : \forall x_2 : P(x_1, x_2) \Rightarrow \exists y : Q(y, x_1)$$

So, there are some syntactic subtleties in translating quantification from DSL ALPHA into FOPC, but they are all solvable with sufficient care.

There is one semantic problem that prevents DSL ALPHA from expressing everything that can be expressed by FOPC formulae with only 0-ary function symbols. That is the limitation of the range of quantified variables in DSL ALPHA to the set of tuples actually occurring in a relation in the database, while FOPC may quantify over an abstract universe $\mathbf{U}$. There is no expression in DSL ALPHA semantically equivalent to $\forall x : P(x)$, since DSL ALPHA can only express the fact that every object in a tuple of the database satisfies the predicate $P$. Because these objects are exactly those mentioned by the knowledge formulae, however, the restricted quantification of DSL ALPHA yields the same answers to queries as the unrestricted quantification of FOPC, so in terms of the behavior of query-answering systems the notations are equivalent. Another way of viewing this semantic difference is to suppose that each database implicitly contains a *closed world assumption* [Rei78] expressing the fact that only the objects mentioned in the database exist. In FOPC with equality, the closed world assumption may be expressed by the formula $\forall x : x \doteq c_0 \vee \cdots \vee x \doteq c_n$, where $c_0, \ldots, c_n$ are all of the constant symbols appearing in the database. Without equality, we can only express the fact that every object acts just like one of $c_0, \ldots, c_n$ (i.e., it satisfies exactly the same formulae), and even that requires an infinite number of formulae.

Given the translation of DSL ALPHA expressions to FOPC formulae suggested above, we may translate DSL ALPHA queries into FOPC questions. DSL ALPHA queries have the general form

$$\underline{\text{GET}}\, Q(P_1.D_1, \ldots, P_i.D_i) : E$$

where $P_1, \ldots, P_i$ are relations in the database, $Q$ is a new relational symbol not occurring in the database, $D_1, \ldots, D_i$ are domain names, and $E$ is an expression. Appropriate declarations of the ranges of variables in $E$ must be given before the query. Let $F$ be a FOPC formula equivalent to $E$, using the variables $x_1, \ldots, x_i$ for the values $P_1.D_1, \ldots, P_i.D_i$. Then, the effect of the DSL ALPHA query above is to assign to $Q$ the relation (set of tuples) answering the question

$$\mathbf{what}\ x_1, \ldots, x_i : F$$

That is,

$$Q = \{\langle c_1, \ldots, c_i\rangle : (\mathbf{what}\ x_1, \ldots, x_i : F) \mathrel{?\!\!-}_P \mathbf{D} \models_P F[c_1, \ldots, c_i/x_1, \ldots, x_i]\}$$

where $\mathbf{D}$ is the set of formulae in the database. Equivalently, the value of $Q$ may be thought of as an abbreviation for the consequentially strongest answer to **conj-what** $x_1, \ldots, x_i : F$ for $\mathbf{D}$, which is just the conjunction of all the answers to **what** $x_1, \ldots, x_i : F$.

Another type of notation for relational database queries avoids explicit quantification entirely, and uses the relational operations of *projection, join,* etc. to define new relations from those in the database. Notation in the style of DSL ALPHA above is called *relational calculus,* because of the similarity to Predicate Calculus in the use of explicit quantification. The alternate approach through operations on relations is called *relational algebra,* because the equations that express the properties of the relational operations resemble well-known definitions of algebras. In fact, each notation has its own algebra and its own calculus, and the difference is just the way in which relations are denoted. Most recent work on relational databases refers to the relational algebra notation, which looks even more distant from FOPC than the relational calculus notation, but is still easily translatable into FOPC. See [Dat86, Cod72] for a description of relational algebra notation for database queries, and a translation to the relational calculus notation.

### 2.3.4 Programming in Equational Logic

Another natural logical system in which to program is Equational Logic. A large number of programming languages may be viewed essentially as different restrictions of programming in Equational Logic.

**Definition 2.3.13.** Let the set $\mathbf{V}$ of variables, the sets $\mathbf{Fun}_0, \mathbf{Fun}_1, \ldots$ of constant and function symbols, and the set $\mathbf{T}_P$ of terms be the same as in FOPC (see Definition 2.3.1). Let $\doteq$ be a new formal symbol (we add the dot to distinguish between the formal symbol for equality in our language and the meaningful equality symbol $=$ used in discussing the language). The set $\mathbf{F}_{\doteq}$ of *equational formulae* (or simply *equations*) is

$$\mathbf{F}_{\doteq} = \{t_1 \doteq t_2 : t_1, t_2 \in \mathbf{T}_P\}$$

Models for Equational Logic are the same as models for FOPC, omitting the predicate assignments. Although $\doteq$ behaves like a special binary predicate symbol, it is given a fixed meaning (as are $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$), so it need not be specified in each model. An equational formula $t_1 \doteq t_2$ holds in a model precisely when $t_1$ denotes the same object as $t_2$ for every variable assignment.

**Definition 2.3.14.** Let the infinite universe $\mathbf{U}$ and the set of function assignments be the same as in FOPC (Definition 2.3.4). If $U \subseteq \mathbf{U}$ and $\tau$ is a function assignment over $U$, then $\langle U, \tau\rangle$ is a *model of Equational Logic.* $\mathbf{M}_{\doteq}$ is the set of all models of Equational Logic.

Let the set of variable assignments be the same as in FOPC (Definition 2.3.2), as well as the definition of a term valuation $\tau_\nu$ from a function assignment $\tau$ and variable assignment $\nu$ (Definition 2.3.3). The classical semantic system for Equational Logic is $\langle \mathbf{F}_{\doteq}, \mathbf{M}_{\doteq}, \models_{\doteq} \rangle$, where $\langle U, \tau \rangle \models_{\doteq} t_1 \doteq t_2$ if and only if $\tau_\nu(t_1) = \tau_\nu(t_2)$ for all variable assignments $\nu$ over $U$.

Models of Equational Logic are essentially *algebras* [Coh65, Grä68, MB67] . The only difference is that algebras are restricted to *signatures*—subsets, usually finite, of the set of constant and function symbols. Such restriction does not affect any of the properties discussed in this chapter. If $\mathbf{T}$ is a finite subset of $\mathbf{T}_{\doteq}$, then the set of algebras $\mathbf{Models}(\mathbf{T})$ (restricted, of course, to an appropriate signature) is called a *variety*. For example, the *monoids* are the models of $\{m(x, m(y, z)) \doteq m(m(x, y), z),\ m(x, e) \doteq x,\ m(e, x) \doteq e\}$ restricted to the signature with one constant symbol $e$ and one binary function symbol $m$.

Perhaps the simplest sort of question that is naturally answered by an equation is 'what is $t_0$?' for a term $t_0$. For each term $t_1$, the equation $t_0 \doteq t_1$ is an answer to this question. The trouble with such a primitive question is that it admits too many answers. For example, the tautology $t_0 \doteq t_0$ is always a correct answer to 'what is $t_0$?'. This problem is closely analogous to the problem of the tautological answer $(a \Rightarrow a)$ to the question $\mathbf{imp}(a)$ ('what atomic formula does $a$ imply?', Example 2.2.3, Section 2.2). For the Shallow Implicational Calculus, we avoided undesired answers simply by listing a set $\mathbf{A}$ of them, in the form $\mathbf{rest\text{-}imp}(a, \mathbf{A})$ ('what atomic formula not in $\mathbf{A}$ does $a$ imply?', Example 2.2.4). Since the number of terms $t_1$ making $t_0 \doteq t_1$ true is generally infinite, we need a finite notation for describing the prohibited answers. A particularly useful way is to give a finite set of terms with variables, and prohibit all instances of those terms from appearing as subterms in an answer term.

**Definition 2.3.15.** Let $x_1, \ldots, x_i$ be a list of variables with no repetitions, and let $t, t_1, \ldots, t_i \in \mathbf{T}_P$. $t[t_1, \ldots, t_i/x_1, \ldots, x_i]$ is the formula that results from substituting the term $t_j$ for every occurrence of the variable $x_j$ in the term $t$, for each $j$, $1 \leq j \leq i$. When $s = t[t_1, \ldots, t_i/x_1, \ldots, x_i]$, we say that $s$ is an *instance* of $t$, and that $t$ is *more general* than $s$.

The concepts of substitution, instance, and generality for terms are analogous to the corresponding concepts defined for formulae in Definition 2.3.5, simplified because all occurrences of variables in terms are free.

**Definition 2.3.16.** Let $t_1, \ldots, t_i \in \mathbf{T}_P$ be terms. A term $s$ is a *normal form* for $\{t_1, \ldots, t_i\}$ if and only if no subterm of $s$ is an instance of a term in $\{t_1, \ldots, t_i\}$.

Let $\mathbf{norm}$ be a new formal symbol. Let

$$\mathbf{Q}_{\doteq} = \{(\mathbf{norm}\, t_1, \ldots, t_i : t) : t, t_1, \ldots, t_i \in \mathbf{T}_P\}$$

Define the relation $?\!\!-_{\doteq} \subseteq \mathbf{Q}_{\doteq} \times \mathbf{F}_{\doteq}$ by

$$(\mathbf{norm}\, t_1, \ldots, t_i : t) \; ?\!\!-_{\doteq} \; (s_1 \doteq s_2)$$

if and only if $s_1 = t$ and $s_2$ is a normal form for $\{t_1, \ldots, t_i\}$.

Now $\langle \mathbf{F}_{\doteq}, \mathbf{Q}_{\doteq}, ?\!\!-_{\doteq} \rangle$ is a query system representing the answers to questions of the form 'what normal form for $t_1, \ldots, t_i$ is equal to $t$?'

For every semicomputable set $\mathbf{T} \subseteq \mathbf{F}_{\doteq}$, the set of equations $(t \doteq s)$ such that $(\mathbf{norm}\, t_1, \ldots, t_i : t) \; ?\!\!-_{\doteq} \; \mathbf{T} \models_{\doteq} (t \doteq s)$ is semicomputable. It is easy to define a query system with conjunctive equational answers, similar to the conjunctive FOPC answers of Definition 2.3.10. Such a step is most useful when *infinite* conjunctions are allowed, so it is reserved for Section 7.1 of the chapter 'Equational Logic Programming.'

**Example 2.3.17.** Let $a \in Fun_0$ be a constant symbol representing *zero*, let $s \in Fun_1$ be a unary function symbol representing the *successor* function, and let $p \in Fun_2$ be a binary function symbol representing *addition*. A natural state of knowledge about these symbols is

$$\mathbf{T} = \{p(a, x) \doteq x,\; p(s(x), y) \doteq s(p(x, y))\}$$

A natural question is $(\mathbf{norm}\, p(x, y) : p(s(s(a)), s(s(s(a)))))$. The unique correct answer is $(p(s(s(a)), s(s(s(a)))) \doteq s(s(s(s(s(a))))))$. That is, the answer to the question 'what is a term for 2 plus 3, not using the plus operator?' is '2 plus 3 equals 5.'

Another question, peculiar but formally legitimate, is $(\mathbf{norm}\; s(s(x)), s(p(x, y)) : s(s(a)))$. Correct answers include $(s(s(a)) \doteq p(s(a), s(a)))$, $(s(s(a)) \doteq p(s(a), p(s(a), a)))$, $(s(s(a)) \doteq p(p(s(a), a), s(a)))$, etc. The answers to this question express 2 as sums of 1 and 0. The simplest form is 1 plus 1; all other forms simply add extraneous 0s.

Every partial computable function $\phi$ may be defined similarly by letting $\phi(i)$ be the unique $j$ such that

$$(\mathbf{norm}\, t_1, \ldots, t_i : f(s^i(a))) \; ?\!\!-_{\doteq} \; \mathbf{T} \models_{\doteq} (f(s^i(a)) \doteq s^j(a))$$

for appropriately chosen $t_1, \ldots, t_i$ and finite set $\mathbf{T}$ of equations defining $f$. In principle, we might ask for most general or consequentially strongest answers to questions in $\mathbf{Q}_{\doteq}$. In practice, multiple answers to such questions are usually incomparable.

A more powerful form of equational question answering involves the solution of equations.

**Definition 2.3.18.** Let **solve** be a new formal symbol. Let

$$\mathbf{Q}_{s\doteq} = \{(\mathbf{solve}\, x_1, \ldots, x_i : t_1 \doteq t_2) : t_1, t_2 \in \mathbf{T}_P\}$$

Define the relation $?\!\!-_{s\doteq}$ by

$$(\mathbf{solve}\, x_1, \ldots, x_i : t_1 \doteq t_2) \mathrel{?\!\!-_{s\doteq}} (s_1 \doteq s_2)$$

if and only if there are terms $u_1, \ldots, u_i \in \mathbf{T}_P$ such that $s_1 = t_1[u_1, \ldots, u_i/x_1, \ldots, x_i]$ and $s_2 = t_2[u_1, \ldots, u_i/x_1, \ldots, x_i]$.

Now $\langle \mathbf{Q}_{s=}, \mathbf{F}_{\doteq}, ?\!\!-_{s=} \rangle$ is a query system representing the answer to questions of the form 'what values of $x_1, \ldots, x_i$ solve the equation $t_1 \doteq t_2$?'

Notice the close analogy between the question $(\mathbf{solve}\, x_1, \ldots, x_i : t_1 \doteq t_2)$ above, and the FOPC question $(\mathbf{what}\, x_1, \ldots, x_i : F)$ of Definition 2.3.6, Section 2.3.1. Syntactically, the equational question is merely the special case of the FOPC question where the formula $F$ is restricted to be an equation. The semantics of Equational Logic lead, however, to very different typical uses for, and implementations of, equation solving.

### 2.3.5 Functional and Equational Programming Languages

A wide variety of nonprocedural programming languages have been inspired by Backus' proposal of *functional programming languages* [Bac74, Bac78] defined by equations. The previous development of Lisp by McCarthy [McC60], although not originally conceived in terms of functional programming, fits in retrospect into the functional approach, and its success has boosted the interest in functional languages substantially. Languages for the algebraic description of abstract data types [GH78, Wan76, FGJM85] use equations in individual programs, rather than in the language design, and one experimental language is defined explicitly in terms of equational logic programming [HO82, HOS85, O'D85]. The essential idea behind functional, algebraic, and equational, programming languages is to find correct answers to normalization questions in $\mathbf{Q}_{\doteq}$ of Section 2.3.4 above. A number of different styles are used to specify these languages, often obscuring the logic programming content. In this section, 'functional programming languages' include all programming languages that *can be* described naturally as answering normalization questions, and we view them as a form of equational logic programming, whether or not they are conventionally thought of in that way. Actual functional languages differ widely on a number of dimensions:

- the notation in which terms are written;
- the way in which the knowledge formulae are determined by the language design and the program;

- the way in which questions are determined by the language design, the program, and the input;
- deviations from pure equational logic programming.

Because of the complexity of these variations, the discussion in this section is organized around the various decisions involved in designing a functional programming language, rather than around a survey of the most important languages.

The style in which many functional programming languages are specified creates an impression that there are fundamental logical differences between functional programming and equational logic programming. This impression is false—functional programming and equational logic programming are two different ways of describing the same behaviors. The different styles of description may encourage different choices in language design, but they do *not* introduce any fundamental logical deviations. In particular, 'higher order' functional programming languages are *not* higher order in any fundamental logical sense, and may be described very naturally by first order equational logic [Gog90]. The chapter 'Equational Logic Programming,' Section 1.2, provides more discussion of the connection between functional and equational programming ideologies.

**Determining Equational Knowledge from Language Design and Program.** The equational formulae used as explicit knowledge for answering normalization questions in functional programming languages are derived from the language design itself and from the program that is being executed. In principle, they could also be given in the input, but this possibility has not been exploited explicitly. Many functional languages are processed by interactive interpreters, which blur the distinction between program and input, and many functional languages have mechanisms, such as *lambda abstraction* [McC60] or the *let* construct, that simulate the introduction of certain equations within an input term. Interactive interpretive processing, and the simulation of additional equations within terms, provide implicitly a lot of the power of explicit equations in the input.

Most functional languages are designed around substantial sets of primitive operations, defined by equations. For example, the primitives *cons* (construct ordered pair), *car* (first component of pair), and *cdr* (second component of pair) in Lisp are defined by the two equations $(car(cons(x, y)) \doteq x)$ and $(cdr(cons(x, y)) \doteq y)$ [McC60]. Primitive operators that manipulate term structure, or provide program control structures, are usually defined by explicitly given small finite sets of equations. Primitive operators for basic mathematical operations are defined by large or infinite sets of equations that must be described rather than listed. For example, the conventional arithmetic operation of addition is defined by the equations $add(0,0) \doteq 0$, $add(0,1) \doteq 1, \ldots$, $add(1,0) \doteq 1$, $add(1,1) \doteq 2, \ldots$, $add(2,0) \doteq 2$, $add(2,1) \doteq 3$, $add(2,2) \doteq 4, \ldots$

Most functional programming languages have rich enough primitive sets of operators defined by equations in the language design that it is not necessary to introduce new equations in a program—the goals of the program may be accomplished by appropriate combinations of primitive operations. In particular, many functional languages have operators that simulate the introduction of additional local equations within a term. Even in languages with powerful primitives, it is often convenient to introduce explicit equations defining new operators in a given program. A few functional languages have weak primitives, and depend upon equations in programs for their expressive power.

Most functional languages impose restrictions on the equations that may be introduced by programs, in order to allow simple and efficient proof searches in the implementation. Typical restrictions are surveyed in the chapter 'Equational Logic Programming,' Section 2.3.2. In most languages, the restrictions on equations in programs allow each equation $f(t_1, \ldots, t_i) \doteq t$ in a program to be treated as a part of the definition of a procedure to compute $f$. Note the similarity to the treatment of clauses in Prolog programs (Section 2.3.2).

**Determining a Question from Language Design, Program, and Input.** Recall that the form of a normalization question is $(\mathbf{norm}\, t_1, \ldots, t_i : t)$. An answer to such a question gives a term equal to $t$ that does not contain a subterm of any of the forms $t_1, \ldots, t_i$. The determination of a question divides naturally into the determination of the prohibited forms $t_1, \ldots, t_i$ and the term $t$ to normalize.

In most functional programming languages, the prohibited forms $t_1, \ldots, t_i$ are determined by partitioning the set of symbols into *constructors, primitive functions,* and *defined functions.* Constructors are intended to be computationally inert—they are treated as elements of a data structure. The use of constructors in equations is highly restricted to ensure this inertness. The binary symbol $cons$ in Lisp is the archetypal constructor. In languages that distinguish constructors, the prohibited forms in normalization questions are precisely the terms $f(x_1, \ldots, x_i)$, where $f$ is a primitive function or a defined function of arity $i$. That is, the normal forms for questions in constructor systems are precisely the *constructor expressions*—terms composed entirely of variables and constructors. The set of constructors may be fixed by the language design, or a program may be allowed to introduce new constructors, in explicit declarations of the symbols or in recursive type definitions.

In a functional language without constructors, the prohibited forms may be implicit in the equations of a program. Most functional languages infer from the presentation of the equation $t_1 \doteq t_2$ that the left-hand side $t_1$ is to be transformed into the right-hand side $t_2$. Such an inference is not justified directly by the semantics of equational logic, but it is often justified by restrictions on the form of equations imposed by the language. So, the prohibited forms in questions are often taken to be precisely the terms on the left-hand sides of equations in the program.

The term to be normalized is typically in the form

$$(t^{\mathrm{la}}[t_1^{\mathrm{pr}}, \ldots, t_{i_{\mathrm{pr}}}^{\mathrm{pr}}/x_1, \ldots, x_{i_{\mathrm{pr}}}])[t_1^{\mathrm{in}}, \ldots t_{i_{\mathrm{in}}}^{\mathrm{in}}/y_1, \ldots, y_{i_{\mathrm{in}}}]$$

where $t^{\mathrm{la}}$ is fixed by the language design, $t_1^{\mathrm{pr}}, \ldots, t_{i_{\mathrm{pr}}}^{\mathrm{pr}}$ are determined by the program, and $t_1^{\mathrm{in}}, \ldots, t_{i_{\mathrm{in}}}^{\mathrm{in}}$ are determined by the input. In principle, other forms are possible, but it seems most natural to view the language design as providing an operation to be applied to the program, producing an operation to be applied to the input. For example, in a pure Lisp *eval* interpreter, $t^{\mathrm{la}} = eval(x_1, nil)$. The program to be evaluated is $t_1^{\mathrm{pr}}$ (the second argument *nil* to the *eval* function indicates an empty list of definitions under which to evaluate the program). Pure Lisp has no input—conceptual input may be encoded into the program, or extralogical features may be used to accomplish input. A natural extension of pure Lisp to allow definitions of symbols in a program yields $t^{\mathrm{la}} = eval(x_1, x_2)$, $t_1^{\mathrm{pr}}$ is the expression given by the program to be interpreted, and $t_2^{\mathrm{pr}}$ is a list of bindings representing the definitions given in the program. In a language, unlike Lisp, where a program designates a particular defined function $f$ as the *main* procedure to be executed on the input, we can have $t^{\mathrm{la}} = x_1$ (the language design does not contribute to the term to be normalized), $t_1^{\mathrm{pr}} = f(y_1)$ where $f$ is the designated main function, and $t_1^{\mathrm{in}}$ is the input. In yet other languages, the term to be normalized is given entirely by the input.

**Notational Variations and Abbreviations.** Functional programming languages vary widely in the notations used for terms, using all sorts of prefix, infix, postfix, and mixfix forms. A survey of the possibilities is pointless. All current functional languages determine the prohibited forms in questions implicitly, either from the language design or from the left-hand sides of equations in a program. Questions are presented by specifying some or all of the term to be normalized—other parts of the term may be implicit as described above. Outputs always present only the normal form $s$ rather than the equation $t \doteq s$, since the question term $t$ is already given explicitly or implicitly.

**Deviations From Pure Equational Logic Programming.** Implementations of functional programming languages have generally come closer to the ideal of pure equational logic programming than Prolog systems have to pure Horn-clause logic programming, largely because of a simpler correspondence between logical and procedural concepts in functional languages than in Prolog. Many functional languages extend pure logic programming with side-effect-producing operations, similar to those in Prolog. These extensions are usually used to deal with input and output. Some functional languages avoid such extensions by modelling the entire input and output as an expression, called a *stream*, that lists the atomic elements of the input and output [Kar81, Tho90, HS88, Hud92, Gor92, Dwe88]; others use a functional representation of input and output based on *continuations* [Per91, HS88, Hud92]. Also see [WW88] for an implicit approach to functional I/O. A protocol that decouples the temporal order of input and output from its representation in a functional program has been proposed as well [Reb93, Reb92] .

The purely functional subsets of functional programming languages usually avoid giving incorrect answers. Implementations of Lisp before the 1980s are arguably incorrect in that their use of *dynamic scope* [Sta90, Mos70] for parameter bindings gives answers that are incorrect according to the conventional logical equations for substitution of terms for parameters, taken from the lambda calculus [Chu41, Ste72, Bar84] . Since the equations for manipulating bindings were never formalized precisely in the early days of Lisp, implementors may argue that their work is correct with respect to an unconventional definition of substitution. Early Lispers seem to have been unaware of the logical literature on variable substitution, and referred to the dynamic binding problem as the 'funarg' problem.

Essentially all functional programming languages before the 1980s fail to find certain semantically correct answers, due to infinite evaluation of irrelevant portions of a term. In conventional Lisp implementations, for example, the defining equation $car(cons(x, y)) \doteq x$ is not applied to a term $car(cons(t_1, t_2))$ until both $t_1$ and $t_2$ have been converted to normal forms. If the attempt to normalize $t_2$ fails due to infinite computation, then the computation as a whole fails, even though a semantically correct answer might have been derived using only $t_1$. Systems that fail to find a normal form for $car(cons(t_1, t_2))$ unless *both* of $t_1$ and $t_2$ have normal forms are said to have *strict cons* functions. The discovery of *lazy evaluation* [FW76, HM76, O'D77] showed how to avoid imposing unnecessary strictness on *cons* and other functions, and many recent implementations of functional programming languages are guaranteed to find all semantically correct answers. Of course, it is always possible to modify defining equations so that the strict interpretation of a function is semantically complete.

**Example 2.3.19.** Consider Lisp, with the standard equations

$$car(cons(x, y)) \doteq x \text{ and } cdr(cons(x, y)) \doteq y$$

To enforce strict evaluation of lists, even in a lazily evaluated implementation of equational logic programming, add new function symbols *test*, *strict*, *and*, and a new constant *true*, with the equations

$$test(true, x) \doteq x, \qquad and(true, true) \doteq true,$$
$$strict(cons(x, y)) \doteq and(strict(x), strict(y)), \qquad strict(a) \doteq true$$

for each atomic symbol $a$. Then, redefine *car* and *cdr* by

$$car(cons(x, y)) \doteq test(strict(y), x) \text{ and } cdr(cons(x, y)) \doteq test(strict(x), y)$$

Lazy evaluation with the new set of equations has the same effect as strict evaluation with the old set.

Some definitions of functional programming language specify strictness explicitly. One might argue that the strict version of *cons* was intended in the original definition of Lisp [McC60], but strictness was never stated explicitly there.

### 2.3.6 Equation Solving and Predicate Calculus with Equality

Given successful applications of logic programming in the pure First-Order Predicate Calculus without equality, and in pure Equational Logic, it is very tempting to develop languages for logic programming in the First-Order Predicate Calculus with Equality (FOPC$_{\doteq}$). Unfortunately, there is a mismatch between the style of questions used in the two sorts of logic programming. FOPC Logic Programming uses questions of the form $(\mathbf{what}\, x_1, \ldots, x_i : F)$ (Definition 2.3.6), whose answers supply substitutions for the variables $x_1, \ldots, x_i$ satisfying $F$. Equational Logic Programming uses questions of the form $(\mathbf{norm}\, t_1, \ldots, t_i : t)$ (Definition 2.3.16), whose answers supply normal forms equal to $t$, not containing the forms $t_1, \ldots, t_i$. The implementation techniques for answering these two sorts of questions do not appear to combine well.

The most natural idea for achieving logic programming in FOPC$_{\doteq}$ is to use the FOPC style of question, and extend it to deal with equations. In principle that is feasible, because the formal equality symbol $\doteq$ may be treated as another binary predicate symbol in $\mathbf{Pred}_2$, and Horn clauses expressing its crucial properties are easy to find. Unfortunately, the natural Horn clauses for the equality predicate, when treated by current implementation techniques for FOPC Logic Programming, yield unacceptably inefficient results. In practice, a satisfactory realization of logic programming in FOPC$_{\doteq}$ will require new techniques for solving equations—that is, for answering questions of the form $(\mathbf{solve}\, x_1, \ldots, x_i : t_1 \doteq t_2)$ (Definition 2.3.18). An interesting experimental language incorporating significant steps toward logic programming in FOPC$_{\doteq}$ is EqL [Jay85], but this language only finds solutions consisting of constructor expressions. The problem of finding nonconstructor solutions is much more difficult.

It is also possible to define FOPC Logic Programming in terms of a normalization-style query. Consider a question 'what acceptable formula implies $F_0$', which presents a goal formula $F_0$, and some description of which answer formulae are acceptable and which are prohibited (the problem of describing such prohibited formulae is more complex in FOPC than in Equational Logic because of the structural properties of the logical connectives $\wedge, \vee, \ldots$). An answer to such a question is an implication of the form $F_1 \Rightarrow F_0$, where $F_1$ is an acceptable formula. The conventional FOPC questions $(\mathbf{what}\ x_1, \ldots, x_i : F_0)$ may be understood as a variant of the 'what implies $F_0$' questions, where the acceptable formulae are precisely those of the form $x_1 \doteq t_1 \wedge \cdots \wedge x_i \doteq t_i$. The proposed new style of FOPC question may be seen as presenting a set of constraints expressed by $F_0$, and requesting a normalized expression of constraints $F_1$ such that every solution to the constraints expressed by $F_1$ is also a solution to the constraints of $F_0$. *Constraint Logic Programming* [JL87, Las91] has taken some steps in this direction, although Constraint Logic Programming generally views the normalized constraints expressed by $F_1$ not as a final result to be output as an answer, but as something to be solved by techniques outside of FOPC, such as numerical techniques for solving linear systems of equations.

# 3 Implementing Logic Programming Languages

Semantic systems and query systems are convenient tools for specifying logic programming languages: we require a language to provide semantically correct answers to questions. In order to implement logic programming languages, we first develop sound (correct) and complete (powerful) *proof systems* to provide effective certificates of the semantic correctness of inferences. Then, we convert proof systems to *programming systems* that process inputs and programs efficiently to produce outputs, by introducing *strategies* for choosing incrementally which proofs to construct. In this chapter we consider only the abstract forms of implementations, far above the level of actual code for real machines. Truly practical implementations do, however, follow these abstract forms quite closely.

## 3.1 Proof Systems and Provable Consequences

A semantic-consequence relation determines in principle whether it is correct to infer one logical formula from others. In order to give a formal justification for an inference, we need a notation for *proofs*. We reserve the initial $P$ for programs, so $D$ is used to stand for proofs, which might also be called *demonstrations* or *derivations*.

**Definition 3.1.1.** A *proof system* is a system $\mathcal{D} = \langle \mathbf{F}, \mathbf{D}, \mid - \rangle$, where

1. $\mathbf{F}$ is a set of *formulae*
2. $\mathbf{D}$ is a set of *proofs*
3. $\mid -$ is a relation on $\mathbf{2}^{\mathbf{F}} \times \mathbf{D} \times \mathbf{F}$ (when $\langle \mathbf{T}, D, F \rangle$ are in the relation $\mid -$, we write $\mathbf{T} \mid D - F$)
4. $\mid -$ is monotone (if $\mathbf{T} \mid D - F$ and $\mathbf{T} \subseteq \mathbf{U}$, then $\mathbf{U} \mid D - F$)

The proof system $\mathcal{D}$ is *compact* if and only if, for all $\mathbf{T} \subseteq \mathbf{F}$, $P \in \mathbf{D}$, and $F \in \mathbf{F}$, whenever $\mathbf{T} \mid D - F$ then there exists a finite subset $\mathbf{T}^f \subseteq \mathbf{T}$ such that $\mathbf{T}^f \mid D - F$.

$\mathbf{T} \mid D - F$ is intended to mean that $D$ is a *proof* of $F$ which is allowed to use hypotheses in $\mathbf{T}$. The fact that $D$ is not *required* to use all hypotheses leads to monotonicity (4). There are a number of proposals for systems of 'nonmonotonic logic,' but they may be regarded as studies of different relations between proofs and formulae than the notion of derivability represented by $\mid -$ above, rather than as arguments about the properties of $\mid -$. The controversy about nonmonotonic logic is not relevant to the discussion in this chapter, and fans of nonmonotonic relations between proofs and formulae may translate into their own notation if they like.

In well-known proof systems, proofs as well as formulae are finite syntactic objects, and the set $\mathbf{D}$ of all proofs is computable. There are important uses, however, for infinite proofs of infinite formulae, so we do not add a formal requirement of computability. Typically, a proof $D$ determines uniquely the conclusion formula $F$ and minimum hypothesis set $\mathbf{T}$ such that $\mathbf{T} \mid D - F$, but there is no need to require such a property. Meseguer [Mes89] proposed a similar general notion of *proof calculus*.

It is straightforward to design a proof system for SIC. The following proof system follows the conventional style of textbooks in logic. Proofs are sequences of formulae, each one either a hypothesis, a postulate, or a consequence of earlier formulae in the proof.

**Example 3.1.2.** Let $\mathbf{F}_{Sh}$ be the set of formulae in SIC. The set of *linear proofs* in SIC is $\mathbf{P}_{Sl} = \mathbf{F}_{Sh}^{+}$, the set of nonempty finite sequences of formulae.
The proof relation $\mid -_{Sl}$ is defined by

$$\mathbf{T} \mid \langle F_0, \ldots, F_m \rangle -_{Sl} F \text{ if and only if } F_m = F \text{ and,}$$
$$\text{for all } i \leq m, \text{ one of the following cases holds:}$$

1. $F_i \in \mathbf{T}$
2. $F_i = (a \Rightarrow a)$ for some atomic formula $a \in \mathbf{At}$
3. $F_i = b$, and there exist $j, k < i$ such that $F_j = a$ and $F_k = (a \Rightarrow b)$ for some atomic formulae $a, b \in \mathbf{At}$
4. $F_i = (a \Rightarrow c)$, and there exist $j, k < i$ such that $F_j = (a \Rightarrow b)$ and $F_k = (b \Rightarrow c)$ for some atomic formulae $a, b, c \in \mathbf{At}$

Now $\langle \mathbf{F}_{Sh}, \mathbf{P}_{Sl}, \mid-_{Sl} \rangle$ is a compact proof system, representing the *Hilbert*, or *linear*, style of proof for implications.

Intuitively, a linear or Hilbert-style proof is a finite sequence of formulae, each one being either a *hypothesis* (case 1 above), a *postulate* (case 2 above, expressing the postulate scheme of *reflexivity* of implication), or the consequence of previous formulae by a *rule of inference* (case 3 above, expressing the rule of *modus ponens*, and case 4, expressing the rule of *transitivity* of implication). The conclusion of a linear proof is the last formula in the list.

An alternative proof system for SIC, less conventional but more convenient for some purposes, follows the *natural deduction* style of proof [Pra65]. Natural deduction proofs are trees, rather than sequences, to display the actual logical dependencies of formulae. They also allow the introduction and discharging of temporary assumptions in proofs, to mimic the informal style in which we prove $a \Rightarrow b$ by assuming $a$ and proving $b$.

**Example 3.1.3.** Let $\mathbf{F}_{Sh}$ be the set of formulae in the Shallow Implicational Calculus defined in Example 2.1.2. Let **assume**, **modus-ponens**, and **deduction-rule** be new formal symbols.
The set $\mathbf{P}_{Sn}$ of *natural deduction proofs* in SIC and the proof relation $\mid-_{Sn} \subseteq \mathbf{2}^{\mathbf{F}_{Sh}} \times \mathbf{P}_{Sn} \times \mathbf{F}_{Sh}$ are defined by simultaneous inductive definition to be the least set and relation such that:

1. for each atomic formula $a \in \mathbf{At}$, and each set of formulae $\mathbf{T} \subseteq \mathbf{F}_{Sh}$,

$$\mathbf{assume}(a) \in \mathbf{P}_{Sn}$$
and
$$\mathbf{T} \cup \{a\} \mid \mathbf{assume}(a) -_{Sn} a$$

2. if $\alpha, \beta \in \mathbf{P}_{Sn}$, and $\mathbf{T} \mid \alpha -_{Sn} a$ for some atomic formula $a \in \mathbf{At}$, and $\mathbf{U} \mid \beta -_{Sn} (a \Rightarrow b)$, then

$$\mathbf{modus\text{-}ponens}(\alpha, \beta) \in \mathbf{P}_{Sn} \text{ and } \mathbf{T} \cup \mathbf{U} \mid \mathbf{modus\text{-}ponens}(\alpha, \beta) -_{Sn} b$$

3. if $\beta \in \mathbf{P}_{Sn}$ and $\mathbf{T} \cup \{a\} \mid \beta -_{Sn} b$ for some atomic formula $a \in \mathbf{At}$, then

   $$\textbf{deduction-rule}(a, \beta) \in \mathbf{P}_{Sn} \text{ and } \mathbf{T} \mid \textbf{deduction-rule}(a, \beta) -_{Sn} (a \Rightarrow b)$$

Now $\langle \mathbf{F}_{Sh}, \mathbf{P}_{Sn}, \mid -_{Sn} \rangle$ is a compact proof system, representing the *natural deduction* style of proof for implications.

Intuitively, $\textbf{assume}(a)$ is a trivial proof of $a$ from hypothesis $a$. $\textbf{modus-ponens}(\alpha, \beta)$ is the result of using a proof $\alpha$ of some atomic formula $a$, a proof $\beta$ of an implication $(a \Rightarrow b)$, and combining the results along with the rule of *modus ponens* to conclude $b$. The set of hypotheses for the resulting proof includes all hypotheses of $\alpha$ and all hypotheses of $\beta$. $\textbf{deduction-rule}(a, \beta)$ is the result of taking a proof $\beta$ of $b$ from hypotheses including $a$, and discharging some (possibly 0) of the assumptions of $a$ from the proof, to get a proof of $(a \Rightarrow b)$ by the *deduction rule*. In clause 3 of the inductive definition above, notice that the hypothesis set $\mathbf{T}$ may contain $a$, in which case $\mathbf{T} \cup \{a\} = \mathbf{T}$. It is this case that allows for the possibility that one or more assumptions of $a$ remain undischarged in an application of the deduction rule.

The style of proof formalized in Example 3.1.3 is called *natural deduction*, since it mimics one popular informal style of proof in which an implication is proved by assuming its hypothesis and deriving its conclusion. Natural deduction style[Pra65], and the similar style of sequent derivation [Gen35, Kle52], both due to Gentzen, are the styles of proof most commonly treated by research in *proof theory* [Ste72, Pra65, Tak75, Sch77, GLT89]. In proof theory, natural deduction rules are expressed very naturally as terms in a typed lambda calculus, where the type of a lambda term is the formula that it proves [How80, Tai67].

In many cases, we are interested only in the provability of an inference, and not the proof itself. So, we let each proof system define a *provable-consequence* relation, analogous to the semantic-consequence relation associated with a semantic system.

**Definition 3.1.4.** Let $\mathcal{D} = \langle \mathbf{F}, \mathbf{D}, \mid - \rangle$ be a proof system. The *provable-consequence* relation defined by $\mathcal{D}$ is $\vdash \subseteq \mathbf{2}^{\mathbf{F}} \times \mathbf{F}$, where

$$\mathbf{T} \vdash F \text{ if and only if there exists a proof } D \in \mathbf{D} \text{ such that } \mathbf{T} \mid D - F$$

The provable-consequence relation $\vdash$ is *compact* if and only if, for all $\mathbf{T} \subseteq \mathbf{F}$, and $F \in \mathbf{F}$, whenever $\mathbf{T} \vdash F$ then there exists a finite subset $\mathbf{T}^f \subseteq \mathbf{T}$ such that $\mathbf{T}^f \vdash F$.

Intuitively, $\mathbf{T} \vdash F$ means that $F$ is *provable* from hypotheses in $\mathbf{T}$. It is easy to show that an arbitrary relation $\vdash$ on $\mathbf{2}^{\mathbf{F}} \times \mathbf{F}$ is the provable-consequence relation of some proof system if and only if it is *monotone*

($\mathbf{T} \vdash F$ and $\mathbf{T} \subseteq \mathbf{U}$ imply that $\mathbf{U} \vdash F$). See [Mes89] for another abstract definition of provable consequence relations, with more stringent requirements.

Most well-known semantic/provable-consequence relations are compact, and semicomputable on the finite sets of formulae. The trinary proof relations of proof systems (restricted to finite sets of hypotheses) are normally computable. That is, in a reasonable proof system we can determine definitely and mechanically whether or not a supposed proof is in fact a proof of a given conclusion from a given finite set of hypotheses. It is easy to see that a proof system with semicomputable set $\mathbf{D}$ of proofs and semicomputable trinary proof relation $\mid\!-$ also has a semicomputable provable-consequence relation, and that compactness of the proof system implies compactness of the provable-consequence relation. In fact, every semicomputable provable-consequence relation is defined by some proof system with computable $\mathbf{D}$ and trinary proof relation $\mid\!-$. In this respect the trinary proof relation acts as a *Gödel T-predicate* [Kle52] to the binary provable consequence relation.

## 3.2 Soundness and Completeness of Proof Systems

The behavior of a proof system may be evaluated in a natural way with respect to a semantic system with the same or larger set of formulae. We say that the proof system is *sound* for the semantic system when every provable consequence is a semantic consequence, and that the proof system is *complete* when every semantic consequence is provable. Roughly, soundness means correctness, and completeness means maximal power within the constraints imposed by the set of formulae available in the proof system.

**Definition 3.2.1.** Let $\mathcal{D} = \langle \mathbf{F}_D, \mathbf{D}, \mid\!- \rangle$ be a proof system, and let $\mathcal{S} = \langle \mathbf{F}_S, \mathbf{M}, \models \rangle$ be a semantic system, with $\mathbf{F}_D \subseteq \mathbf{F}_S$.

$\mathcal{D}$ is *sound* for $\mathcal{S}$ if and only if, for all $\mathbf{T} \subseteq \mathbf{F}$ and $F \in \mathbf{F}_D$

$$\mathbf{T} \vdash F \text{ implies } \mathbf{T} \models F$$

$\mathcal{D}$ is *complete* for $\mathcal{S}$ if and only if, for all $\mathbf{T} \subseteq \mathbf{F}$ and $F \in \mathbf{F}_D$,

$$\mathbf{T} \models F \text{ implies } \mathbf{T} \vdash F$$

Each of the proposed proof systems for SIC is sound and complete for the semantic system of SIC. The following proofs of completeness for SIC are similar in form to completeness proofs in general, but unusually simple. Given a set of hypotheses $\mathbf{T}$, and a formula $F$ that is not provable from $\mathbf{T}$, we construct a model $M$ satisfying exactly the set of provable consequences of $\mathbf{T}$ within some sublanguage $\mathbf{F}_F$ containing $F$

($\mathbf{Theory}(M) \cap \mathbf{F}_F = \mathbf{Theory}(\mathbf{Models}(\mathbf{T})) \cap \mathbf{F}_F$). In our example below, $\mathbf{F}_F$ is just the set of all shallow implicational formulae ($\mathbf{F}_{Sh}$), and the model construction is particularly simple.

**Example 3.2.2.** Each of the proof systems $\langle \mathbf{F}_{Sh}, \mathbf{P}_{Sn}, |-_{Sn} \rangle$ of Example 3.1.3 and $\langle \mathbf{F}_{Sh}, \mathbf{P}_{Sl}, |-_{Sl} \rangle$ of Example 3.1.2 is sound and complete for the semantic system $\langle \mathbf{F}_{Sh}, \mathbf{M}_{Sh}, \models_{Sh} \rangle$ of Example 2.1.2.

The proofs of soundness involve elementary inductions on the size of proofs. For the natural deduction system, the semantic correctness of a proof follows from the correctness of its components; for the linear system correctness of a proof $\langle F_0, \ldots, F_{m+1} \rangle$ follows from the correctness of the prefix $\langle F_0, \ldots, F_m \rangle$.

The proofs of completeness require construction, for each set $\mathbf{T}$ of formulae, of a model $M = \{a \in \mathbf{At} : \mathbf{T} \vdash_{Sn} a\}$ (or $\mathbf{T} \vdash_{Sl} a$). So $M \models_{Sh} a$ for all atomic formulae $a \in \mathbf{T} \cap \mathbf{At}$ trivially. It is easy to show that $M \models_{Sh} (a \Rightarrow b)$ for all implications $(a \Rightarrow b) \in \mathbf{T}$ as well, since either $a \notin M$, or $b$ follows by modus ponens from $a$ and $a \Rightarrow b$, so $b \in M$. Finally, it is easy to show that $M \models_{Sh} F$ if and only if $\mathbf{T} \vdash_{Sn} F$ (or $\mathbf{T} \vdash_{Sl} F$).

In richer languages than SIC, containing for example disjunctions or negations, things may be more complicated. For example, if the disjunction $(A \vee B)$ is in the set of hypotheses $\mathbf{T}$, each model satisfying $\mathbf{T}$ must satisfy one of $A$ or $B$, yet neither may be a logical consequence of $\mathbf{T}$, so one of them must be omitted from $\mathbf{F}_F$. Similarly, in a language allowing negation, we often require that every model satisfy either $A$ or $\neg A$. Such considerations complicate the construction of a model substantially.

Notice that the formal definitions above do *not* restrict the nature of semantic systems and proof systems significantly. All sorts of nonsensical formal systems fit the definitions. Rather, the relationships of soundness and completeness provide us with conceptual tools for evaluating the behavior of a proof system, with respect to a semantic system that we have already accepted as appropriate. Logic is distinguished from other technical sciences, not by the formal definition of the systems that it studies, but rather by the use of logical concepts to evaluate these systems. The distinction between logic programming and other forms of programming is similarly based on the approach to evaluating the systems, rather than the formal qualities of the systems.

While formal studies may reveal pleasing or disturbing properties of semantic systems, there is also an unavoidable intuitive component in the evaluation of a semantic system. A semantic system is reasonable only if it accurately captures enough of the structure of the mental meaning that we want to associate with a formula to allow a sensible determination of the correctness or incorrectness of steps of reasoning. Since different people have different intuitive notions about the proper meanings of formulae, and the same person may find different intuitions useful for different purposes, we should be open minded about considering a broad variety of semantic systems. But, the mere satisfaction of a given form of definition, whether the form in Definition 2.1.1 above, or one of the popular 'denotational' forms using lattices or chain-complete partial orderings and fixpoints, does not make a 'semantics' meaningful. A number of additional mathematical and philosophical dissertations are needed to give practical aid in the evaluation and selection of semantic proposals. The best one-sentence advice that I can offer is to always ask of a proposed semantic system, 'for a given formula $F$, what does the semantic system tell about the information regarding the world that is asserted by $F$.' For this chapter, I use only systems of semantics based on first-order classical forms of logic that have been shaken down very thoroughly over the years. In these systems, the individual models clearly present enough alleged facts about a possible state of the world to determine the truth or falsehood of each formula. So, the information asserted by a formula is that the world under discussion is one of the ones satisfying the formula in the sense of $\models$. There are many reasons to prefer nonclassical logics for programming and for other purposes. But, we must never rest satisfied with 'semantic' treatments of these logics until they have been connected convincingly to an intuitive notion of *meaning*.

A semantic system and a sound proof system may be used to analyze the process by which implicit knowledge is made explicit—we are particularly interested in the derivation of explicit knowledge in order to answer a question. Consider an agent with *implicit* knowledge given by the set $\mathbf{K}$ of models consistent with that knowledge, and represent the agent's *explicit* knowledge by a set $\mathbf{T}$ of formulae that he can utter effectively. The correctness of the explicit knowledge requires that $\mathbf{T} \subseteq \mathbf{Theory}(\mathbf{K})$. Suppose that the agent knows that the proof system is sound, and suppose that he can recognize at least some cases when the relation $\mathbf{T} \mid D - F$ holds—often this capability results from computing the appropriate decision procedure for a computable proof system (or enumeration procedure for a semicomputable proof system), with an appropriate finite subset of $\mathbf{T}$. Then, whenever he finds a formula $F$ and a proof $D$ such that $\mathbf{T} \mid D - F$, the agent may add the formula $F$ to his explicit knowledge. The soundness of the proof system guarantees that $F \in \mathbf{Theory}(\mathbf{K})$. Notice that sound proofs can never extend explicit knowledge beyond the bound determined by implicit knowledge, which is $\mathbf{Theory}(\mathbf{K})$.

**Definition 3.2.3.** Let $\mathcal{Q} = \langle \mathbf{F}_Q, \mathbf{Q}, ?\!- \rangle$ be a query system, and let $\mathcal{D} = \langle \mathbf{F}_D, \mathbf{D}, \mid - \rangle$ be a proof system with $\mathbf{F}_Q \subseteq \mathbf{F}_D$.

$Q \mathrel{?\!-} \mathbf{T} \vdash F$ means that $F \in \mathbf{F}_Q$ is a *provably correct* answer to $Q \in \mathbf{Q}$ for explicit knowledge $\mathbf{T} \subseteq \mathbf{F}_S$, defined by

$$Q \mathrel{?\!-} \mathbf{T} \vdash F \text{ if and only if } Q \mathrel{?\!-} F \text{ and } \mathbf{T} \vdash F$$

Similarly,

$$Q \mathrel{?\!\!-} \mathbf{T} \mid D - F \text{ if and only if } Q \mathrel{?\!\!-} F \text{ and } \mathbf{T} \mid D - F$$

If $\mathcal{P}$ is sound, then provable correctness implies semantic correctness (Definition 2.2.2). If $\mathcal{P}$ is complete, then semantic correctness implies provable correctness.

Going back to the communication analysis of previous sections, let $\mathbf{K}_s$ be the speaker's implicit knowledge, let $\mathbf{K}_a^0$ be the auditor's initial implicit knowledge, and let $\mathbf{T}_a^0$ be the auditor's initial explicit knowledge. When the speaker utters a set of formulae $\mathbf{T}_u$, consistent with her implicit knowledge, the auditor's implicit knowledge improves as before to $\mathbf{K}_a^1 = \mathbf{K}_a^0 \cap \mathbf{Models}(\mathbf{T}_u)$, and the auditor's explicit knowledge improves to $\mathbf{T}_a^1 = \mathbf{T}_a^0 \cup \mathbf{T}_u$. Let a questioner ask a question $Q$ of the auditor. Without further communication from the speaker, the auditor may improve his explicit knowledge by proving new formulae from hypotheses in $\mathbf{T}_a^1$ in order to answer the question $Q$. If the auditor's initial explicit knowledge is empty, then $\mathbf{T}_a^1 = \mathbf{T}_u$, so the formulae derivable in this way are exactly the provable consequences of $\mathbf{T}_u$, and the answers that may be found are exactly the provably correct answers to $Q$ for $\mathbf{T}_u$. If the proof system used by the auditor is sound, then all such answers are semantically correct; if the proof system is complete then all semantically correct answers are provable. Now let the speaker be a programmer who utters a set of formulae constituting a logic program, let the auditor be a processor, and let the questioner be a user whose question is given as input to the program. Then, computations performed by the processor take the form of search for and construction of proofs deriving the explicit knowledge needed to produce an output answer, from the explicit knowledge given in the program.

## 3.3 Programming Systems

A *programming system* represents the computational behavior of a processor. In order to understand logic programming, we consider an arbitrary, possibly ineffective and nondeterministic, programming system, and then show how to evaluate its behavior logically with respect to a given semantic system and query system. We choose an unconventional formal notation for programming systems in order to expose the close analogy of sets of formulae to programs, questions to inputs, proofs to computations, answers to outputs.

**Definition 3.3.1.** A *programming system* is a system $\mathcal{P} = \langle \mathbf{P}, \mathbf{I}, \mathbf{C}, \mathbf{O}, \rhd\, [\!], \rightarrow \rangle$, where

1. $\mathbf{P}$ is a set of *programs*

2. $\mathbf{I}$ is a set of *inputs*
3. $\mathbf{C}$ is a set of *computations*
4. $\mathbf{O}$ is a set of *outputs*
5. $\triangleright\ \square$ is a relation on $\mathbf{I} \times \mathbf{P} \times \mathbf{C}$ (when $\langle I, P, C\rangle$ are in the relation $\triangleright\ \square$, we write $I \triangleright\ P \,\square\, C$)
6. For each $I \in \mathbf{I}$ and $P \in \mathbf{P}$, there is at least one $C \in \mathbf{C}$ with $I \triangleright\ P \,\square\, C$
7. $\rightharpoondown$ is a relation on $\mathbf{C} \times \mathbf{O}$
8. For each $C \in \mathbf{C}$, there is at most one $O \in \mathbf{O}$ with $C \rightharpoondown O$ (that is, $\rightharpoondown$ is a partial function from $\mathbf{C}$ to $\mathbf{O}$)

We define the relation $\triangleright\ \square \rightharpoondown \subseteq \mathbf{I} \times \mathbf{P} \times \mathbf{C} \times \mathbf{O}$ by

$$I \triangleright\ P \,\square\, C \rightharpoondown O \text{ if and only if } I \triangleright\ P \,\square\, C \text{ and } C \rightharpoondown O$$

$I \triangleright\ P \,\square\, C$ is intended to mean that, when input $I$ is presented to program $P$, one possible resulting computation is $C$. $C \rightharpoondown O$ is intended to mean that the computation $C$ produces output $O$. Multiple computations for a given $P$ and $I$ are allowed, indicating nondeterminism, but each computation produces at most one output. The intended meaning of a nondeterministic computation relation is that we do not know which of the several possible computations will occur for a given input and output [Dij76]. The choice may be determined by unknown and time-dependent factors, or it may be random. In order to guarantee some property of the result of computation, we must ensure that it holds for all possible nondeterministic choices.

In well-known programming systems from theory textbooks, programs, inputs, and outputs (like formulae, proofs, and questions) are finite syntactic objects, and the sets $\mathbf{P}$, $\mathbf{I}$, and $\mathbf{O}$ are computable. Infinite computations, in the form of infinite sequences of finite memory-state descriptions, are allowed, but in theory textbooks the infinite computations normally have no output. On the other hand, the straightforward abstractions of well-known programming systems from real life (abstracted only by ignoring all bounds on time and space resources in the computation) allow infinite computations to consume infinite inputs and produce infinite outputs.

In many cases, we are interested only in the input-output behavior of a program, and not in the computations themselves. So, we let each programming system define a trinary relation determining the possible outputs for a given program and input.

**Definition 3.3.2.** Let $\mathcal{P} = \langle \mathbf{P}, \mathbf{I}, \mathbf{C}, \mathbf{O}, \triangleright\ \square, \rightharpoondown\rangle$ be a programming system. The *computed-output* relation defined by $\mathcal{P}$ is $\triangleright\ \square\!\!\rightharpoondown \subseteq \mathbf{P} \times \mathbf{I} \times \mathbf{O}$, where

$$I \triangleright\ P \;\square\!\!\mapsto O \text{ if and only if there exists a computation } C \in \mathbf{C}$$
$$\text{such that } I \triangleright\ P \;\square\, C \rightharpoondown O.$$

For a programming system to be useful, the computed-output relation $\triangleright\ \square\!\!\mapsto$ must be sufficiently effective to allow a mechanical implementation. Computed-output relations in theory textbooks, like provable-consequence relations, are normally semicomputable; computation and output relations $\triangleright\ \square$ and $\rightharpoondown$, like trinary proof relations $\mid -$, are normally computable (and even primitive recursive). If $C, \triangleright\ \square$, and $\rightharpoondown$ are all semicomputable, then so is $\triangleright\ \square\!\!\mapsto$. In fact, every semicomputable computed-output relation $\triangleright\ \square\!\!\mapsto$ may be defined from computable $C, \triangleright\ \square$, and $\rightharpoondown$. Systems with infinite inputs and/or infinite outputs require more liberal, and less conventional, notions of effectiveness.

The programming systems of Definition 3.3.1 are not required to be deterministic, or effective. They are a simple generalization of the programming systems, also called *indexings* and *Gödel numberings*, of recursion theory [MY78, Kle52]. Our $I \triangleright\ P \;\square\!\!\mapsto O$ corresponds to the recursion-theoretic notation $\phi_P(I) = O$. Recursion theory normally considers only *determinate* programming systems.

**Definition 3.3.3.** A programming system $\mathcal{P} = \langle \mathbf{P}, \mathbf{I}, \mathbf{C}, \mathbf{O}, \triangleright\ \square, \rightharpoondown \rangle$ is *determinate* if and only if, for each program $P$ and input $I$, there is at most one output $O$ such that $I \triangleright\ P \;\square\!\!\mapsto O$. That is, $\triangleright\ \square\!\!\mapsto$ is a partial function from $\mathbf{P} \times \mathbf{I}$ to $\mathbf{O}$.

A programming system $\mathcal{P} = \langle \mathbf{P}, \mathbf{I}, \mathbf{C}, \mathbf{O}, \triangleright\ \square, \rightharpoondown \rangle$ is *deterministic* if and only if, for each program $P$ and input $I$, there is a unique computation $C$ such that $I \triangleright\ P \;\square\, C$. That is, $\triangleright\ \square$ is a partial function from $\mathbf{P} \times \mathbf{I}$ to $\mathbf{C}$.

Determinism implies determinacy, but not the converse—a nondeterministic programming system may provide many computations that yield the same determinate output.

A number of different programming systems may be defined to answer questions in the shallow implicational calculus, depending on the type of question and the stringency of requirements for the answer.

**Example 3.3.4.** Let $\mathbf{F}_{\Rightarrow} = \{(a \Rightarrow b) : a, b \in \mathbf{At}\}$ be the set of implicational SIC formulae ($\mathbf{F}_{Sh} - \mathbf{At}$, with $\mathbf{F}_{Sh}$ and $\mathbf{At}$ defined in Example 2.1.2). The set of *implicational logic programs* ($\mathbf{P}_{\Rightarrow}$) is the set of finite subsets of $\mathbf{F}_{\Rightarrow}$.

Let the set of inputs to implicational logic programs be $\mathbf{Q}_{S1}$—the set of questions of the form $\mathbf{imp}(a)$ ('what atomic formula does $a$ imply?') defined in Example 2.2.3.

The set of *naive implicational computations* ($\mathbf{C}_{S1}$) is the set of nonempty finite and infinite sequences of atomic formulae.

The computation relation $\triangleright\ \square_{S1}$ is defined by $\mathbf{imp}(a) \triangleright\ \mathbf{T} \;\square_{S1} \langle c_0, c_1, \ldots \rangle$

if and only if

1. $c_0 = a$
2. $(c_{i-1} \Rightarrow c_i) \in \mathbf{T}$ for all $i > 0$ in the (finite or infinite) range of the sequence $\langle c_0, \ldots \rangle$

The output relation $\rightharpoondown_{S1}$ is defined by

$$\langle c_0, \ldots, c_m \rangle \rightharpoondown_{S1} (c_0 \Rightarrow c_m)$$

Infinite computations have no output.
Now $\langle \mathbf{P}_{\Rightarrow}, \mathbf{Q}_{S1}, \mathbf{C}_{S1}, \mathbf{F}_{\Rightarrow}, \rhd\, [\!]_{S1}, \rightharpoondown_{S1} \rangle$ is a programming system, computing answers to questions of the form 'what atomic formula does $a$ imply?'

The programming system above behaves nondeterministically and indeterminately in proving some implication of $a$. Its computations may halt at any point and output the latest atomic conclusion found. Loops in the graph of implications lead to infinite computations with no output. Notice that each finite computation $\langle c_0, \ldots c_i, \ldots, c_m \rangle$, with output $(c_0 \Rightarrow c_m)$, translates very easily into the linear proof

$$\langle (c_0 \Rightarrow c_0), \ldots, (c_{i-1} \Rightarrow c_i), (c_0 \Rightarrow c_i), \ldots, (c_{m-1} \Rightarrow c_m), (c_0 \Rightarrow c_m) \rangle$$

of $(c_0 \Rightarrow c_m)$ in the proof system of Example 3.1.2. The first line is an instance of the reflexive rule, and subsequent lines alternate between implications in $\mathbf{T}$, and applications of transitivity.

In order to avoid uninformative outputs, such as $(a \Rightarrow a)$, we need a programming system with a slightly more sophisticated notion of when to stop a computation.

**Example 3.3.5.** Let **fail** be a new formal symbol, and let the set of *naive implicational computations with failure* be

$$\mathbf{C}_{S2} = \mathbf{C}_{S1} \cup \{ \langle c_0, \ldots, c_m, \mathbf{fail} \rangle : c_0, \ldots, c_m \in \mathbf{At} \}$$

(the set of finite and infinite sequences of atomic formulae, possibly ending in the special object **fail**).
Let the set of inputs to implicational logic programs be $\mathbf{Q}_{S2}$—the set of questions of the form $\mathbf{rest\text{-}imp}(a, \mathbf{A})$ ('what atomic formula not in $\mathbf{A}$ does $a$ imply?').
The computation relation $\rhd\, [\!]_{S2}$ is defined by $\mathbf{rest\text{-}imp}(a, \mathbf{A}) \rhd\, \mathbf{T}\, [\!]_{S2} \langle c_0, \ldots \rangle$ if and only if

1. $c_0 = a$
2. if $c_i \in \mathbf{A}$, and there is a $d \in \mathbf{At}$ such that $(c_i \Rightarrow d) \in \mathbf{T}$, then the sequence has an $i + 1$st element $c_{i+1}$, and $(c_i \Rightarrow c_{i+1}) \in \mathbf{T}$

3. if $c_i \in \mathbf{A}$, and there is no $d \in \mathbf{At}$ such that $(c_i \Rightarrow d) \in \mathbf{T}$, then the sequence has an $i+1$st element, and $c_{i+1} = \mathbf{fail}$
4. if $c_i \in \mathbf{At} - \mathbf{A}$, then either there is no $i$+1st element, or $(c_i \Rightarrow c_{i+1}) \in \mathbf{T}$
5. if $c_i = \mathbf{fail}$, then there is no $i+1$st element

So, $\triangleright\,\square_{S2}$ allows computations to terminate only when an atomic formula outside of $\mathbf{A}$ has been found, or a dead end in the implication graph has been reached.
The output relation $\rightharpoonup_{S2}$ is defined by

$$\langle c_0, \ldots, c_m \rangle \rightharpoonup_{S2} (c_0 \Rightarrow c_m) \text{ for } c_m \in \mathbf{At}$$

Infinite computations and finite computations ending in **fail** have no output.
Now $\langle \mathbf{P}_{\Rightarrow}, \mathbf{Q}_{S2}, \mathbf{C}_{S2}, \mathbf{F}_{\Rightarrow}, \triangleright\,\square_{S2}, \rightharpoonup_{S2} \rangle$ is a programming system, computing answers to questions of the form 'what atomic formula not in $\mathbf{A}$ does $a$ imply?'

The programming system of Example 3.3.5 is nondeterministic and indeterminate. It avoids useless answers, but it still may fall into infinite or finite failing computations, even when a legitimate answer exists. It also may find an answer, but fail to output it and proceed instead into a failure or an infinite computation. Successful computations translate into proofs as in Example 3.3.4.

We may further strengthen the behavior of a programming system by letting it back up and try new proofs after finite failures, and by insisting that answers be output as soon as they are found.

**Example 3.3.6.** Let the set of inputs to implicational logic programs again be $\mathbf{Q}_{S2}$—the set of questions of the form $\mathbf{rest\text{-}imp}(a, \mathbf{A})$ ('what atomic formula not in $\mathbf{A}$ does $a$ imply?').
Let **print** be a new formal symbol. The set of *backtracking implicational computations* ($\mathbf{C}_{S3}$) is the set of nonempty finite and infinite sequences of finite sets of atomic formulae, possibly ending in the special form $\mathbf{print}(a)$ where $a$ is an atomic formula (**fail** of Example 3.3.5 is represented now by the empty set).
The computation relation $\triangleright\,\square_{S3}$ is defined by $\mathbf{rest\text{-}imp}(a, \mathbf{A}) \triangleright\, \mathbf{T}\, \square_{S3} \langle \mathbf{C}_0, \ldots \rangle$ if and only if

1. $\mathbf{C}_0 = \{a\}$
2. if $\mathbf{C}_i \subseteq \mathbf{A}$ and $\mathbf{C}_i \neq \emptyset$, then there is an $i+1$st element, and

$$\mathbf{C}_{i+1} = \mathbf{C}_i - \{c\} \cup \{d : (c \Rightarrow d) \in \mathbf{T}\}$$

for some atomic formula $c \in \mathbf{C}_i$

3. if $\mathbf{C}_i \subseteq \mathbf{At}$, and $\mathbf{C}_i - \mathbf{A} \neq \emptyset$, then there is an $i+1$st element, and $\mathbf{C}_{i+1} = \mathbf{print}(c)$ for some $c \in \mathbf{C}_i - \mathbf{A}$
4. if $\mathbf{C}_i = \emptyset$, or $\mathbf{C}_i = \mathbf{print}(c)$, then there is no $i+1$st element

So, $\rhd\ [\!]_{S3}$ allows a computation to replace any atomic formula that has already been proved with the set of atomic formulae that it implies directly in $\mathbf{T}$. A computation halts precisely when it chooses a unique atomic formula not in $\mathbf{A}$ to output, or when it fails by producing the empty set. The output relation $\rightharpoonup_{S3}$ is defined by

$$\langle \{a\}, \ldots, \mathbf{print}(b) \rangle \rightharpoonup_{S3} (a \Rightarrow b)$$

Infinite computations, and finite computations ending in $\emptyset$, have no output. Now $\langle \mathbf{P}_\Rightarrow, \mathbf{Q}_{S2}, \mathbf{C}_{S3}, \mathbf{F}_\Rightarrow, \rhd\ [\!]_{S3}, \rightharpoonup_{S3} \rangle$ is another programming system, computing answers to questions of the form 'what atomic formula not in $\mathbf{A}$ does $a$ imply?'

The programming system of Example 3.3.6 is nondeterministic and indeterminate. It is less susceptible to missing answers than that of Example 3.3.5. The new system does not get stuck with a failure when a single path in the implication graph leads to a dead end: a computation ends in $\emptyset$ only when *all* paths have been followed to a dead end. When there is a finite path to an answer, and also a cycle, the nondeterministic choice of which formula to replace at each step determines which path is followed in the computation, and so determines success or infinite computation. The translation of a computation in the latest system to a proof is not quite as transparent as in Examples 3.3.4 and 3.3.5, but it is still simple. Each successful computation $\langle \{c_0\}, \mathbf{C}_1, \ldots, \mathbf{C}_{m-1}, \mathbf{print}(c_m) \rangle$ must contain a sequence of atomic formulae $\langle c_0, c_1, \ldots, c_{m-1}, c_m \rangle$, where for $i < m$ $c_i \in \mathbf{C}_i$, and for adjacent pairs $c_i, c_{i+1}$, either $c_i = c_{i+1}$, or $(c_i \Rightarrow c_{i+1}) \in \mathbf{T}$. This sequence of atomic formulae transforms to a linear proof as before.

A final example of a programming system illustrates the use of incremental output from possibly infinite computations to produce consequentially strong answers.

**Example 3.3.7.** Let the set of inputs to implicational logic programs be $\mathbf{Q}_{Sc}$—the set of questions of the form $\mathbf{conj\text{-}imp}(a)$ ('what are some atomic formulae that $a$ implies?') from Example 2.2.6.
The set of *conjunctive implicational computations* ($\mathbf{C}_{S4}$) is the set of nonempty finite or infinite sequences of finite sets of atomic formulae (the same as $\mathbf{C}_{S3}$, without the final elements $\mathbf{print}(a)$).
The computation relation $\rhd\ [\!]_{S4}$ is defined by $\mathbf{conj\text{-}imp}(a) \rhd\ \mathbf{T}\ [\!]_{S4}\ \langle \mathbf{C}_0, \ldots \rangle$ if and only if

1. $\mathbf{C}_0 = \{a\}$

2. if $\mathbf{C}_i \neq \emptyset$, then there is an $i+1$st element, and

$$\mathbf{C}_{i+1} = (\mathbf{C}_i - \{c\}) \cup \{d : (c \Rightarrow d) \in \mathbf{T}\}$$

   for some atomic formula $c \in \mathbf{C}_i$

3. if $\mathbf{C}_i = \emptyset$, then there is no $i+1$st element

The computations above are the same as those of Example 3.3.6, except that we never choose a single atomic formula to output. $\emptyset$ is no longer regarded as a failure.
The output relation $\rightharpoonup_{S4}$ is defined by $\langle \mathbf{C}_0, \ldots \rangle \rightharpoonup_{S4} (a \Rightarrow b_1) \wedge \cdots \wedge (a \Rightarrow b_m)$ if and only if

1. $\mathbf{C}_0 = \{a\}$
2. $\{b_1, \ldots b_m\} = \mathbf{C}_0 \cup \mathbf{C}_1 \cup \cdots$, and $b_1, \ldots b_m$ are given in the order of first appearance in the sequence $\mathbf{C}_0, \ldots$, with ties broken by some arbitrary ordering of atomic formulae

Notice that even infinite computations have output.
Now $\langle \mathbf{P}_\Rightarrow, \mathbf{Q}_{Sc}, \mathbf{C}_{S4}, \mathbf{F}_\Rightarrow, \triangleright\, [\!]_{S4}, \rightharpoonup_{S4} \rangle$ is a programming system, computing answers to questions of the form 'what are some atomic formulae that $a$ implies?'

The programming system above should be thought of as producing its output incrementally at each computation step. It is nondeterministic and indeterminate. Even though the computation may be infinite, it never fails to produce an answer, although the output may be the trivial formula $a \Rightarrow a$. The strength of the answer produced depends on the nondeterministic choices of the atomic formula replaced in each computation step.

## 3.4 Soundness and Completeness of Programming Systems

A query system determining what constitutes an answer, and the semantic-consequence relation of a semantic system determining the correctness of an answer, yield criteria for evaluating the behavior of a programming system, similarly to the criteria for evaluating a proof system in Section 3.1. We define *soundness* and *completeness* of programming systems, in analogy to the soundness and completeness of proof systems. Logic programming is distinguished from other sorts of programming by the use of such logical concepts to evaluate programming systems.

There is only one sensible concept of soundness for a programming system: every output is a correct answer to the input program. When a given question has more than one correct answer, completeness criteria vary depending on the way in which we expect an output answer to be chosen.

**Definition 3.4.1.** Let $\mathcal{P} = \langle \mathbf{P}, \mathbf{I}, \mathbf{C}, \mathbf{O}, \triangleright\ \square, \rightharpoonup \rangle$ be a programming system, let $\mathcal{S} = \langle \mathbf{F}_S, \mathbf{M}, \models \rangle$ be a semantic system, and let $\mathcal{Q} = \langle \mathbf{F}_Q, \mathbf{Q}, ?\!- \rangle$ be a query system, with $\mathbf{P} \subseteq \mathbf{2}^{\mathbf{F}_S}$, $\mathbf{I} \subseteq \mathbf{Q}$, and $\mathbf{O} \subseteq \mathbf{F}_S \cap \mathbf{F}_Q$.

$\mathcal{P}$ is *sound* for $\mathcal{S}$ and $\mathcal{Q}$ if and only if, for all $P \in \mathbf{P}$ and $I \in \mathbf{I}$,

$$I \triangleright\ P \ \square\!\!\rightharpoonup O \text{ implies } I\ ?\!-\ P \models O$$

$\mathcal{P}$ is *weakly complete* for $\mathcal{S}$ and $\mathcal{Q}$ if and only if, for all $P \in \mathbf{P}$ and $I \in \mathbf{I}$ such that $I$ is semantically answerable for $P$ (Definition 2.2.2), and for all computations $C \in \mathbf{C}$ such that $I \triangleright\ P \ \square\ C$, there exists $O \in \mathbf{O}$ such that

$$C \rightharpoonup O \text{ and } I\ ?\!-\ P \models O$$

($O$ is unique because $\rightharpoonup$ is a partial function).

$\mathcal{P}$ is *consequentially complete* for $\mathcal{S}$ and $\mathcal{Q}$ if and only if $\mathcal{P}$ is weakly complete and, in addition, $O$ above is a consequentially strongest correct answer ($\{O\} \models N$ for all $N \in \mathbf{F}_Q$ such that $I\ ?\!-\ P \models N$).

So, a programming system is sound if all of its outputs are correct answers to input questions, based on the knowledge represented explicitly by programs. A system is weakly complete if, whenever a correct answer exists, every computation outputs some correct answer. A system is consequentially complete if, whenever a correct answer exists, every computation outputs a consequentially strongest correct answer. Notice that, for consequential completeness, the strength of the output answer is judged against all possible answers in the query system, not just those that are possible outputs in the programming system, so we cannot achieve consequential completeness by the trickery of disallowing the truly strongest answers.

A programming system provides another approach to analyzing a simple form of communication. While semantic systems, proof systems, and query systems yield insight into the meaning of communication and criteria for evaluating the behavior of communicating agents, programming systems merely describe that behavior. A programmer provides a program $P$ to a processor. A user (sometimes, but not always, identical with the programmer) provides an input $I$, and the processor performs a computation $C$ such that $I \triangleright\ P \ \square\ C$ from which the output $O$, if any, such that $C \rightharpoonup O$, may be extracted. We allow the mapping from computation to output to depend on purely conventional rules that are adopted by the three agents. What aspects of a computation are taken to be significant to the output is really a matter of convention, not necessity. Often, only the string of symbols displayed on some printing device is taken to be the output, but in various contexts the temporal order in which they are displayed (which may be different from the printed order if the device can backspace), the temporal or spatial interleaving of input and output, the speed with which output occurs, the color in which the symbols are displayed, which of several devices is used for display, all may be taken as significant. Also, convention determines the treatment of infinite computation as having no output, null output, or some nontrivial and possibly infinite output produced incrementally. The presentation of input to a computation is similarly a matter of accepted convention, rather than formal computation.

In logic programming, where the programmer acts as speaker, the processor as auditor, and the user as questioner, soundness of the programming system guarantees that all outputs constitute correct answers. Various forms of completeness guarantee that answers will always be produced when they exist. In this sense, soundness and completeness mean that a programming system provides a correct and powerful implementation of the auditor in the speaker-auditor-questioner scenario of Section 2.2.

There is a close formal correspondence between programming systems and pairs of proof and query systems: inputs correspond to questions, programs correspond to sets of hypotheses, computations to proofs, and outputs to theorems (for a different correspondence, in which programs in the form of lambda terms correspond to natural deduction proofs, see [How80, Tai67, CAB$^+$86]—compare this to the interpretation of formulae as queries and proofs as answers [Mes89]). Notice that both quaternary relations $Q \mathrel{?\!\!-} \mathbf{T} \mid D - F$ and $I \rhd P \mathbin{[\,]} C \rightarrow O$ are typically computable, while both of the trinary relations $Q \mathrel{?\!\!-} \mathbf{T} \vdash F$ and $I \rhd P \mathrel{[\,]\!\!\mapsto} O$ are typically semicomputable. Furthermore, the definitions of the trinary relations from the corresponding quaternary relations are analogous. In both cases we quantify existentially over the third argument, which is variously a proof or a computation.

There is an important difference, however, between the forms of definition of the provable-answer relation $Q \mathrel{?\!\!-} \mathbf{T} \mid D - F$, and of the computational relation $I \rhd P \mathbin{[\,]} C \rightarrow O$, reflecting the difference in intended uses of these relations. This difference has only a minor impact on the relations definable in each form, but a substantial impact on the efficiency of straightforward implementations based on the definitions. In the query-proof domain, we relate formulae giving explicit knowledge (program) to the proofs (computations) that can be constructed from that knowledge, yielding formulae (outputs) that are provable consequences of the given knowledge in the relation $\mathbf{T} \mid D - F$. We independently relate questions (inputs) to the answers (outputs) in the relation $Q \mathrel{?\!\!-} F$, and then take the conjunction of the two. There is no formal provision for the question (input) to interact with the knowledge formulae (program) to guide the construction of the proof (computation)—the question (input) is only used to select a completed proof. In the computational domain, we relate inputs (questions) directly to programs (knowledge formulae) to determine computations (proofs) that they can produce in the relation $I \rhd P \,\Box\, C$. Then, we extract outputs (answer formulae) from computations (proofs) in the relation $C \rightharpoonup O$. The relation $I \rhd P \,\Box\, C$ provides a formal concept that may be used to represent the interaction of input (question) with program (knowledge) to guide the construction of the computation (proof).

Given a proof system and a query system, we can construct a programming system with essentially the same behavior. This translation is intended as an exercise in understanding the formal correspondence between proofs and computations. Since our requirements for proof systems and programming systems are quite different, this construction does *not* normally lead to useful implementations.

**Proposition 3.4.2.** *Let $\mathcal{D} = \langle \boldsymbol{F}, \boldsymbol{D}, \mid - \rangle$ be a proof system, and let $\mathcal{Q} = \langle \boldsymbol{F}, \boldsymbol{Q}, ?\!- \rangle$ be a query system. Define the programming system*

$$\mathcal{P} = \langle \boldsymbol{2}^{\boldsymbol{F}}, \boldsymbol{Q}, (\boldsymbol{D} \times \boldsymbol{F}) \cup \{\boldsymbol{fail}\}, \boldsymbol{F}, \rhd\, \Box, \rightharpoonup \rangle$$

*where*

1. $Q \rhd \boldsymbol{T} \,\Box\, \langle P, F \rangle$ *if and only if* $Q \mathrel{?\!-} \boldsymbol{T} \mid P - F$
2. $Q \rhd \boldsymbol{T} \,\Box\, \boldsymbol{fail}$ *if and only if there are no $P$ and $F$ such that* $Q \mathrel{?\!-} \boldsymbol{T} \mid P - F$
3. $\langle P, F \rangle \rightharpoonup G$ *if and only if* $F = G$
4. $\boldsymbol{fail} \rightharpoonup F$ *is false for all* $F \in \boldsymbol{F}$

*Then,*

$$Q \mathrel{?\!-} \boldsymbol{T} \mid P - F \text{ if and only if } Q \rhd \boldsymbol{T} \,\Box\, \langle P, F \rangle \rightharpoonup F$$

*Therefore,*

$$Q \mathrel{?\!-} \boldsymbol{T} \vdash F \text{ if and only if } Q \rhd \boldsymbol{T} \,\Box\!\!\rightharpoonup F$$

If $\mathcal{D}$ above is sound for some semantic system $\mathcal{S}$, then $\mathcal{P}$ is sound for $\mathcal{S}$ and $\mathcal{Q}$, but the converse fails because some formulae may never occur as answers to questions. Proposition 3.4.2 shows that a proof may be interpreted as a nondeterministically chosen computation that outputs a theorem.

Because hypotheses to proofs are sets of formulae, rather than single formulae, and the proof relation must be monotone with respect to the subset relation, there are computational relations $\rhd \,\Box\!\!\rightharpoonup$ defined by programming systems that are not the same as the relation $?\!- \vdash$ of any proof and query systems. Intuitively, in order to mimic an arbitrary programming system with a proof system and a query system, we must augment the output $O$

resulting from input $I$ into a formula asserting that *input $I$ produces output $O$*. This augmentation is almost trivial, in the sense that echoed input may just as well be regarded as an implicit part of the output. Informal question answering uses such implicit augmentation: in response to the question 'what is the capital city of Idaho?' the abbreviated answer 'Boise' is generally accepted as equivalent to the full answer 'the capital city of Idaho is Boise.'

**Proposition 3.4.3.** *Let $\mathcal{P} = \langle \mathbf{P}, \mathbf{I}, \mathbf{C}, \mathbf{O} \rangle$ be a programming system. Define the proof system $\mathcal{D} = \langle \mathbf{P} \cup (\mathbf{I} \times \mathbf{O}), \mathbf{C}, \mid\!- \rangle$, where*

1. *$\mathbf{T} \mid C - \langle I, O \rangle$ if and only if $I \rhd\ P \,[\!]\, C \rightarrow O$ for some $P \in \mathbf{T}$*
2. *$\mathbf{T} \mid C - P$ is false for all $P \in \mathbf{P}$*

*Also define the query system $\mathcal{Q} = \langle \mathbf{I} \times \mathbf{O}, \mathbf{I}, ?\!- \rangle$, where*

1. *$I \mathrel{?\!-} \langle J, O \rangle$ if and only if $I = J$.*

*Then,*

$$I \rhd\ P \,[\!]\, C \rightarrow O \text{ if and only if } I \mathrel{?\!-} \{P\} \mid C - \langle I, O \rangle$$

*Therefore,*

$$I \rhd\ P \,[\!]\!\!\mapsto O \text{ if and only if } I \mathrel{?\!-} \{P\} \vdash \langle I, O \rangle$$

As in the construction of Proposition 3.4.2, soundness of $\mathcal{D}$ implies soundness of $\mathcal{P}$, but the converse fails, and completeness does not transfer either way. Proposition 3.4.3 shows that a computation may be interpreted as a proof that a given program and input produce a certain output.

**Example 3.4.4.** The programming systems of Examples 3.3.4, 3.3.5, 3.3.6, and 3.3.7 are all sound for their appropriate semantic and query systems. The proof of soundness is easy—every computation can be transformed easily into a proof in a sound proof system.

The programming system of naive implicational computations in Example 3.3.4 is not weakly complete. Consider the program

$$\{a \Rightarrow b,\ b \Rightarrow a,\ a \Rightarrow c\}$$

Given the input question $\mathbf{imp}(a)$ ('what logical formula does $a$ imply?'), a possible computation is the infinite sequence

$$\langle a, b, a, b, \ldots \rangle$$

which has no output. There are three correct answers,

$$a \Rightarrow a,\ a \Rightarrow b, \text{ and } a \Rightarrow c$$

each of which is found by a short computation.

The programming system of naive implicational computations with failure in Example 3.3.5 is not weakly complete. Consider the program

$$\{a \Rightarrow b,\ a \Rightarrow c,\ c \Rightarrow a,\ a \Rightarrow d\}$$

Given the input question $\mathbf{rest\text{-}imp}(a, \{a, b, c\})$ ('what logical formula not in $\{a, b, c\}$ does $a$ imply?'), two possible computations with no output are

$$\langle a, b, \mathbf{fail}\rangle \text{ and } \langle a, c, a, c, \ldots\rangle$$

There is a correct answer, $a \Rightarrow d$, which is found by the computation $\langle a, d\rangle$.

The programming system of backtracking implicational computations in Example 3.3.6 avoids the finite failure of the naive computations with failure, but is still not weakly complete because of infinite computations. It succeeds on the program and question above, with the unique computation

$$\langle\{a\}, \{b, c, d\}, \{d\}\rangle$$

but fails on a slightly trickier case. Consider the program

$$\{a \Rightarrow b,\ a \Rightarrow c,\ c \Rightarrow a,\ a \Rightarrow d,\ d \Rightarrow e\}$$

and the question $\mathbf{rest\text{-}imp}(a, \{a, b, c, d\})$. There is no finite failing computation, and the correct answer $a \Rightarrow e$ is output by the computation

$$\langle\{a\}, \{b, c, d\}, \{b, d\}, \{a, d\}, \{a, e\}, \mathbf{print}(e)\rangle$$

But there is still an infinite computation that misses the output:

$$\langle\{a\}, \{b, c, d\}, \{b, a, d\}, \{b, c, d\}, \ldots\rangle$$

The programming system of conjunctive implicational computations in Example 3.3.7 is weakly complete, simply because every computation outputs some correct answer of the form $(a \Rightarrow a) \wedge \ldots$, where in the worst case only the first conjunct is given. This system was clearly aimed, however, toward producing consequentially strong answers. It is not consequentially complete. Consider again the program

$$\{a \Rightarrow b,\ a \Rightarrow c,\ c \Rightarrow a,\ a \Rightarrow d,\ d \Rightarrow e\}$$

and the new question $\mathbf{conj\text{-}imp}(a)$. The computation

$$\langle\{a\}, \{b, c, d\}, \{b, c, e\}, \{b, a, e\}, \{b, c, d, e\}, \{b, a, d, e\}, \{b, c, d, e\}, \ldots\rangle$$

outputs the consequentially strongest answer

$$(a \Rightarrow a) \wedge (a \Rightarrow b) \wedge (a \Rightarrow c) \wedge (a \Rightarrow d) \wedge (a \Rightarrow e)$$

But the computation

$$\langle \{a\}, \{b, c, d\}, \{b, a, d\}, \{b, c, d\}, \ldots \rangle$$

outputs only the weaker answer

$$(a \Rightarrow a) \wedge (a \Rightarrow b) \wedge (a \Rightarrow c) \wedge (a \Rightarrow d)$$

missing the conjunct $a \Rightarrow e$.

In each case above, the failure of completeness results from the possibility of unfortunate choices for the next computational step.

Most practical implementations of logic programming languages are not complete, and many are not even sound. Nonetheless, soundness and completeness are useful standards against which to judge implementations. Most implementations are sound and complete for well-characterized subsets of their possible programs and inputs. The cases where soundness and/or completeness fail are typically considered at least peculiar, and sometimes pathological, and they are the topics of much discussion and debate. The history of programming languages gives some hope for a trend toward stricter adherence at least to soundness criteria. For example, early Lisp processors employed dynamic scoping, which is essentially an unsound implementation of logical substitution. Modern Lisp processors are usually statically scoped, and provide sound implementations of substitution [MP80, BGS82, Ree86]. As compiler technology matured, the logically correct static scoping was found to be more efficient than dynamic scoping, although early work assumed the contrary.

In spite of the close formal correspondence outlined above between proof and computation, our natural requirements for proof systems and programming systems differ significantly. The requirements for correctness, formalized as soundness, are essentially the same—everything that is proved/computed must be a logical consequence of given information. But, the requirements for power, formalized as completeness, vary substantially.

Proofs are thought of as things to search for, using any available tools whether formal, intuitive, or inspirational, and we only demand formal or mechanical verification of the correctness of a proof, not mechanical discovery of the proof. So, proofs are quantified existentially in the definition of completeness of a proof system, and we are satisfied with the mere existence of a proof of a given true formula. Computations, on the other hand, are thought of as being generated on demand by a computing agent in order to satisfy the requirement of a user. So, we require that a complete programming system be guaranteed to produce a sufficiently strong correct answer whenever a correct answer exists. Since we do not know which of several possible computations will be generated nondeterministically by the computing agent, we quantify universally over computations.

Because of the requirement that *all* computations in a complete programming system yield correct answers, merely mimicking the relational behavior of a proof system, as in Proposition 3.4.2, is not sufficient for useful implementation. Practical implementations must use complete *strategies* for choosing proofs in order to provide programming systems with the desired guaranteed results.

## 3.5 Proof-Theoretic Foundations for Logic Programming

Given a suitable proof system, a practical implementation of a logic programming language still must solve the difficult problem of searching the set of proofs for one that provides an answer to a given input question. Methods for choosing and generating proofs are called *proof strategies*. While logical semantics provides the conceptual tools for specifying logic programming languages, *proof theory* [Ste72, Pra65, Tak75, Sch77, GLT89]. provides the tools for developing proof strategies. Once a proof strategy has been defined, the remaining problems in implementation are the invention of appropriate algorithms and data structures for the strategy, and the details of code generation or interpreting. The organization of logic programming into the application of semantics to specification, and the application of proof theory to implementation, does *not* mean, however, that the former precedes the latter in the design of a logic programming language. Design normally requires the simultaneous consideration of specification and implementation, and the designer must search the two spaces of semantic specifications and proof-theoretic strategies in parallel for a compatible pair of ideas. In different circumstances either topic can be the primary driver of design decisions. In writing this chapter, I have not been able to develop the proof theoretic side of logic programming design as thoroughly as the semantic side, merely because I ran out of time, pages, and energy. In this section, I will only outline the issues involved in applying proof theory to logic programming.

The choice of a proof strategy affects both the power and the complexity of an implementation, but not the soundness. Given a sound proof system, a proof strategy can only choose (or fail to find) a correct proof, it cannot expand the class of proofs. But, a given proof strategy may be incapable of discovering certain provably correct answers, so it may yield an incomplete computing system, even when starting with a complete proof system.

So, there is great value in proof theoretic theorems demonstrating that, whenever a formula $F$ is provable, there is a proof in some usefully restricted form. Even when these do not lead to complete implementations, they can improve the power/complexity tradeoff dramatically. Fortunately, proof theorists have concentrated a lot of attention on such results, particularly in the form of *proof normalization* theorems, which show that all proofs may be reduced to a normal form with special structure. Many normalization results are expressed as *cut elimination* theorems, showing that a particular version of *modus ponens* called the *cut* rule may be removed from proofs. Cut elimination theorems are usually associated with the predicate calculus and its fragments, variants, and extensions. The impact of cut elimination on proof strategies has been studied very thoroughly, leading to an excellent characterization of the sequent-style proof systems that are susceptible to generalizations of the simple goal-directed proof strategy used in Prolog [MNPS91]. These proof-theoretic methods have been applied successfully in some novel logics where the model-theoretic semantics are not yet properly understood.

In the term rewriting literature, there are similar results on the normalization of equational proofs. Many of these come from *confluence* (also called *Church-Rosser*) results. A system of equations

$$\{l_1 \doteq r_1, \ldots, l_m \doteq r_m\}$$

presented with each equation oriented in a particular left-right order, is *confluent* precisely if every proof of an equation $s \doteq t$ may be transformed into a rewriting of each of $s$ and $t$ to a common form $u$. *Rewriting* means here that an equational hypothesis $l_i \doteq r_i$ may only be used from left to right, to replace instances of $l_i$ by corresponding instances of $r_i$ but not the reverse. The restriction of equational proofs to rewritings allows complete strategies that are much simpler and more efficient than those that search through all equational proofs. See Sections 2.2 and 2.3 of the chapter 'Equational Logic Programming' in this volume, as well as [Klo91], for more on the application of term rewriting to equational proofs.

# 4 The Uses of Semantics

The development of logic programming systems from logics, given above, provides a particular flavor of semantics, called *logical semantics,* for logic programming languages. Logical semantics, rather than competing directly with other flavors of programming language semantics, provides different insights, and is useful for different purposes. The careful comparison of different styles of semantics is a wide-open area for further research. In this section, I sketch the sort of relations that I believe should be explored between logical semantics, denotational semantics, and algebraic semantics.

Meseguer proposes two sorts of logic programming—'weak' logic programming uses essentially the same notion of logical semantics as mine, while 'strong' logic programming uses the theory of a single model, such as a model derived by algebraic semantics [Mes89].

## 4.1 Logical Semantics vs. Denotational Semantics

Roughly, denotational semantics [Sco70, SS71, Sto77] takes the meaning of a program to be an abstract description of its input/output behavior, where inputs and outputs are uninterpreted tokens. Denotational semantics assigns to each program a unique value carrying that meaning. One problem of denotational semantics is how to deal with observable computational behavior, such as nontermination, that does not produce output tokens in the concrete sense. This problem was solved by expanding the domains of input and output, as well as the domains of program meanings, to partially ordered sets (usually chain-complete partial orderings [Mar76] or lattices [Sco76]) containing objects representing abstract computational behaviors, not all of which produce tokens as output [Rey73, Sco82]. In practice, the definition of appropriate domains is often the most challenging task in creating a denotational semantic description of a programming language, and *domain theory* has become a definite specialty in theoretical computer science [Sch86, Sto77, Zha91, Gun92, Win93].

The denotational approach provides a useful tool for characterizing what a particular type of implementation actually does, but it does not give any intuitive basis for discussing what an implementation *ought* to do. Logical semantics, on the other hand, begins with an interpretation of input and output. It does not directly address techniques for analyzing the behavior of programs—that is left to a metalanguage. But it *does* provide an intuitive basis for distinguishing logically reasonable behaviors from other behaviors.

For example, denotational semantics for functional languages was initially defined using eager evaluation [Bac78] The domains that were used to define eager evaluation are not rich enough to represent lazy evaluation. In fact the definition of domains for lazy evaluation [Win93] posed difficult technical problems, causing resistance to the use of lazy evaluation. Denotational semantics for lazy evaluation matured long after the idea had been implemented, and informal evidence for its utility had been presented. [FW76, HM76] Logical semantics for equational programming, on the other hand, *requires* lazy evaluation for completeness, and the demand for lazy evaluation from this point of view precedes its invention as a programming tool—at latest it goes back to [O'D77] and the essential roots are already there in work on combinatory logic and the lambda calculus [CF58]. Once lazy evaluation was explained denotationally, that explanation became a very useful tool for analysis and for deriving implementations. In general logical semantics predicts and prescribes useful techniques, while denotational semantics explains and analyzes them.

## 4.2 Logical Semantics vs. Initial/Final-Algebra and Herbrand Semantics

Semantics that use initial or final algebras or Herbrand models [GH78, GTW78, MG85] to represent the meanings of programs provide systematic techniques for deriving denotational-like semantics from logical semantics. Logical semantics determines a large class of models consistent with a given program. Algebraic semantic techniques construct a single model, depending on the language in which output is expressed as well as the given program, whose output theory is the same as that of the class of models given by logical semantics. This single model can be used for the same sorts of analysis as denotational semantics (although it is not always based on a lattice or chain-complete partial ordering). Such single-model semantics must be reconsidered whenever the output language expands, since in the larger language the theory of the single model may *not* be the same as the theory of the class of models consistent with the program.

For example, consider a language (based on ideas from Lucid [AW77] ) with symbols *cons*, *first*, *more*, *a*, *b*, satisfying the equations

$$\begin{array}{rcl} \mathit{first}(\mathit{cons}(x,y)) & \doteq & \mathit{first}(x) \\ \mathit{first}(a) & \doteq & a \\ \mathit{first}(b) & \doteq & b \\ \mathit{more}(\mathit{cons}(x,y)) & \doteq & y \\ \mathit{more}(a) & \doteq & a \\ \mathit{more}(b) & \doteq & b \end{array}$$

Assume that only the symbols $a$ and $b$ are allowed as output—that is, we are only interested in deriving equations of the forms $s \doteq a$ and $t \doteq b$, where $s$ and $t$ are arbitrary input terms. Algebraic semantic techniques typically interpret this system over a universe of infinite flat (i.e., not nested) lists with elements from the set $\{a, b\}$, where after some finite prefix, all elements of the list are the same. $\mathit{cons}(s,t)$ is interpreted as the list beginning with the first element of $s$, followed by all the elements of $t$. In this algebraic interpretation, $\mathit{cons}(\mathit{cons}(a,b),b) \doteq \mathit{cons}(a,b)$ and $\mathit{cons}(b,\mathit{cons}(a,a)) \doteq \mathit{cons}(b,a)$ hold, although neither is a semantic consequence of the given equations. If, however, we add the conventional symbols *car* and *cdr*, and define them by the equations

$$\begin{array}{rcl} \mathit{car}(\mathit{cons}(x,y)) & \doteq & x \\ \mathit{cdr}(\mathit{cons}(x,y)) & \doteq & y \end{array}$$

then we must expand the universe of the algebraic interpretation to the universe of binary trees with leaves marked $a$ and $b$. There is no way to define the functions $car$ and $cdr$ in the flat list model so that they satisfy the new equations. If we take the full equational theory of the flat list model, and add the defining equations for $car$ and $cdr$, then the resulting theory trivializes. Every two terms $s$ and $t$ are equal by the derivation

$$\begin{aligned} s &\doteq cdr(car(cons(cons(a,s),b))) \\ &\doteq cdr(car(cons(a,b))) \\ &\doteq cdr(car(cons(cons(a,t),b))) \\ &\doteq t \end{aligned}$$

Of course, nobody would apply algebraic semantics in this way—taking the model for a smaller language and trying to interpret new function symbols in the same universe. But, what the example shows is that an algebraic model of a given system of equations may not preserve all of the relevant information about the behavior of those equations in extended languages. The set of models associated with a system of equations by logical semantics is much more robust, and carries enough information to perform extensions such as the example above.

In general, algebraic semantic techniques, based on initial models, final models, and Herbrand universes, provide useful tools for determining, in a given program, the minimum amount of information that a data structure must carry in order to support the computational needs of that program. They do *not*, and are not intended to, represent the inherent information given by the formulae in the program, independently of a particular definition of the computational inputs and outputs that the program may operate on.

# References

[AB75] Alan Ross Anderson and Nuel D. Belnap Jr. *Entailment—the Logic of Relevance and Necessity*, volume 1. Princeton University Press, Princeton, NJ, 1975.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Computer Science and Applied Mathematics. Academic Press, New York, NY, 1986.

[AW77] E. Ashcroft and W. Wadge. Lucid: A nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[Bac74] John Backus. Programming language semantics and closed applicative languages. In *Proceedings of the 1st ACM Symposium* *on Principles of Programming Languages*, pages 71–86. ACM, 1974.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[Bar84] Hendrik Peter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[BGS82] R. A. Brooks, R. P. Gabriel, and Guy L. Steele. An optimizing compiler for lexically scoped Lisp. In *Proceedings of the 1982 ACM Compiler Construction Conference*, June 1982.

[BS76] Nuel D. Belnap Jr. and T. B. Steel. *The Logic of Questions and Answers*. Yale University Press, New Haven, CT, 1976.

[CAB+86] Robert L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, Todd B. Knoblock, N. P. Mendler, Prakesh Panangaden, J. T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), June 1970.

[Cod71] E. F. Codd. A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*, 1971.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In *Data Base Systems*, volume 6 of *Courant Computer Science Symposia*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Coh65] P. M. Cohn. *Universal Algebra*. Harper and Row, New York, NY, 1965.

[Dat86] C. J. Date. *An Introduction to Database Systems*. Systems Programming. Addison-Wesley, Reading, MA, 4 edition, 1986.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dwe88] Andrew Dwelly. Synchronizing the I/O behavior of functional programs with feedback. *Information Processing Letters*, 28, 1988.

[FGJM85] K. Futatsugi, Joseph A. Goguen, J.-P. Jouannaud, and José Meseguer. Principles of OBJ2. In *12th Annual Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[FHV84] Ronald Fagin, Joseph Y. Halpern, and Moshe Y. Vardi. A model-theoretic analysis of knowledge. In *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 268–278, 1984.

[FW76] Daniel Friedman and David S. Wise. Cons should not evaluate its arguments. In *3rd International Colloquium on Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.

[Gal86] Jean H. Gallier. *Logic for Computer Science—Foundations of Automatic Theorem Proving*. Harper & Row, New York, NY, 1986.

[GB92] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [?].

[GH78] John V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):1–26, 1978.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1989.

[GMN84] Hervé Gallaire, Jack Minker, and J. M. Nicolas. Databases: A deductive approach. *ACM Computing Surveys*, 16(2), June 1984.

[Gog90] Joseph A. Goguen. Higher order functions considered unnecessary for higher order programming. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 309–351. Addison-Wesley, 1990.

[Gor92] Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, 1992.

[Grä68] G. Grätzer. *Universal Algebra.* Van Nostrand, Princeton, NJ, 1968.

[GTW78] Joseph A. Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, Cambridge, MA, 1992.

[HM76] P. Henderson and J. H. Morris. A lazy evaluator. In *3rd Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. SIGPLAN and SIGACT, 1976.

[HO82] C. M. Hoffmann and M. J. O'Donnell. Programming with equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112, 1982.

[HOS85] C. M. Hoffmann, M. J. O'Donnell, and R. I. Strandh. Implementation of an interpreter for abstract equations. *Software — Practice and Experience*, 15(12):1185–1203, 1985.

[How80] William Howard. The formulas-as-types notion of construction. In John P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, New York, NY, 1980.

[HS88] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/RR665, Yale University, New Haven, CT, December 1988.

[Hud92] Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Jay85] Bharat Jayaraman. Equational programming: A unifying approach to functional and logic programming. Technical Report 85-030, The University of North Carolina, 1985.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[Kar81] K. Karlsson. Nebula, a functional operating system. Technical report, Chalmers University, 1981.

[Kle52] Steven Cole Kleene. *Introduction to Metamathematics*, volume 1 of *Biblioteca Mathematica*. North-Holland, Amsterdam, 1952.

[Klo91] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, Oxford, 1991.

[Kow74] R. Kowalski. Predicate logic as a programming language. In *Information Processing 74*, pages 569–574. North-Holland, 1974.

[Las91] Jean-Louis Lassez. From LP to CLP: Programming with constraints. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software: International Conference*, volume 526 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Mar76] G. Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6:53–68, 1976.

[MB67] Saunders Mac Lane and G. Birkhoff. *Algebra*. Macmillan, New York, NY, 1967.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[Mes89] José Meseguer. General logics. In H.-D. Ebbinghaus et. al., editor, *Logic Colloquium '87: Proceedings of the Colloquium held in Granada, Spain July 20–25, 1987*, Amsterdam, 1989. Elsevier North-Holland.

[Mes92] José Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proceedings of the 3rd International Conference on Algebraic and Logic Programming, Volterra, Italy, September 1992*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[MG85] José Meseguer and Joseph A. Goguen. Initiality, induction, and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Mos70] Joel Moses. The function of FUNCTION in LISP, or why the FUNARG problem should be called the environment problem. *ACM SIGSAM Bulletin*, 15, 1970.

[MP80] Steven S. Muchnick and Uwe F. Pleban. A semantic comparison of Lisp and Scheme. In *Proceedings of the 1980 Lisp Conference*, pages 56–64, 1980. Stanford University.

[MRT53] Andrzej Mostowski, Raphael M. Robinson, and Alfred Tarski. *Undecidability and Essential Undecidability in Arithmetic*, chapter II, pages 37–74. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1953. Book author: Alfred Tarski in collaboration with Andrzej Mostowski and Raphael M. Robinson. Series editors: L. E. J. Brouwer, E. W. Beth, A. Heyting.

[MW88] David Maier and David S. Warren. *Computing with Logic—Logic Programming with Prolog*. Benjamin Cummings, Menlo Park, CA, 1988.

[MY78] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Theory of Computation. North-Holland, New York, NY, 1978.

[NM88] Gopalan Nadathur and Dale Miller. An overview of $\lambda$Prolog. In *Proceedings of the 5th International Conference on Logic Programming*, pages 810–827, Cambridge, MA, 1988. MIT Press.

[NM90] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[O'D77] Michael James O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.

[O'D85] Michael James O'Donnell. *Equational Logic as a Programming Language*. Foundations of Computing. MIT Press, Cambridge, MA, 1985.

[Per91] Nigel Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1991.

[Pra65] Dag Prawitz. *Natural Deduction—a Proof-Theoretic Study*. Alqvist and Wiksell, Stockholm, 1965.

[Reb92] Samuel A. Rebelsky. I/O trees and interactive lazy functional programming. In Maurice Bruynooghe and Martin Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 458–472. Springer-Verlag, August 1992.

[Reb93] Samuel A. Rebelsky. *Tours, a System for Lazy Term-Based Communication*. PhD thesis, The University of Chicago, June 1993.

[Ree86] The Revised$^3$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[Rei78] Raymond Reiter. On closed world databases. In Hervé Gallaire and Jack Minker, editors, *Logic and Databases*, pages 149–178. Plenum Press, 1978. also appeared as [?].

[Rei84] Raymond Reiter. Towards a logical reconstruction of relational database theory. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling—Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Topics in Information Systems, pages 191–233. Springer-Verlag, 1984.

[Rey73] John C. Reynolds. On the interpretation of Scott's domains. In *Proceedings of Convegno d'Informatica Teorica*, Rome, Italy, February 1973. Instituto Nazionale di Alta Matematica (Citta Universitaria).

[Sch77] Kurt Schutte. *Proof Theory*. Springer-Verlag, New York, NY, 1977.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sco70] Dana Scott. *Outline of a Mathematical Theory of Computation*, volume PRG-2 of *Oxford Monographs*. Oxford University Press, Oxford, UK, 1970.

[Sco76] Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3), 1976.

[Sco82] Dana Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming—Ninth Colloquium*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, Berlin, 1982.

[SS71] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, pages 19–46, Polytechnic Institute of Brooklyn, 1971.

[Sta90] W. Richard Stark. *LISP, Lore, and Logic—An Algebraic View of LISP Programming, Foundations, and Applications.* Springer-Verlag, New York, NY, 1990.

[Ste72] Sören Stenlund. *Combinators, $\lambda$-Terms, and Proof Theory.* D. Reidel, Dordrecht, Netherlands, 1972.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[Tai67] William W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

[Tak75] Gaisi Takeuti. *Proof Theory.* North-Holland, Amsterdam, 1975.

[Tho90] Simon Thompson. Interactive functional programs, a method and a formal formal semantics. In David A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[vK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[Wan76] Mitchell Wand. First order identities as a defining language. *Acta Informatica*, 14:336–357, 1976.

[Web87] *Webster's Ninth New Collegiate Dictionary.* Merriam-Webster Inc., Springfield, MA, 1987.

[Win93] Glynn Winksel. *The Formal Semantics of Programming Languages—An Introduction.* Foundations of Computing. MIT Press, Cambridge, MA, 1993.

[WW88] John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 169–179. ACM, 1988.

[Zha91] Guo-Qiang Zhang. *Logic of Domains.* Progress in Theoretical Computer Science. Birkhauser, Boston, MA, 1991.

# Index

# Index

# Index