Chapter 1

Declarative programming in AnsProlog*: introduction and preliminaries

Among other characteristics, an intelligent entity – whether an intelligent autonomous agent, or an intelligent assistant – must have the ability to go beyond just following direct instructions while in pursuit of a goal. This is necessary to be able to behave intelligently when the assumptions surrounding the direct instructions are not valid, or there are no direct instructions at all. For example even a seemingly direct instruction of 'bring me coffee' to an assistant requires the assistant to figure out what to do if the coffee pot is out of water, or if the coffee machine is broken. The assistant will definitely be referred to as lacking intelligence if he or she were to report to the boss that there is no water in the coffee pot and ask the boss what to do next. On the other hand, an assistant will be considered intelligent if he or she can take a high level request of 'make travel arrangements for my trip to International AI conference 20XX' and figure out the lecture times of the boss; take into account airline, hotel and car rental preferences; take into account the budget limitations, etc.; overcome hurdles such as the preferred flight being sold out; and make satisfactory arrangements. This example illustrates *one benchmark of intelligence – the level of request an entity can handle*. At one end of the spectrum the request is a detailed algorithm that spells out *how* to satisfy the request, which no matter how detailed it is may not be sufficient in cases where the assumptions inherent in the algorithm are violated. At the other end of the spectrum the request spells out *what* needs to be done, and the entity has the knowledge – again in the *what* form rather than the *how* form – and the knowledge processing ability to figure out the exact steps (that will satisfy the request) and execute them, and when it does not have the necessary knowledge it either knows where to obtain the necessary knowledge, or is able to gracefully get around its ignorance through its ability to reason in the presence of incomplete knowledge.

The languages for spelling out *how* are often referred to as *procedural* while the languages for spelling out *what* are referred to as *declarative.* Thus our initial thesis that intelligent entities must be able to comprehend and process descriptions of *what*

leads to the necessity of inventing suitable declarative languages and developing support structures around those languages to facilitate their use. We consider the development of such languages to be fundamental to knowledge based intelligence, perhaps similar to the role of the language of calculus in mathematics and physics. *This book is about such a declarative language – the language of* **AnsProlog***. We now give a brief history behind the quest for a suitable declarative language for knowledge representation, reasoning, and declarative problem solving.

Classical logic which has been used as a specification language for procedural programming languages was an obvious initial choice to represent declarative knowledge. But it was quickly realized that classical logic embodies the monotonicity property according to which the conclusion entailed by a body of knowledge stubbornly remains valid no matter what additional knowledge is added. This disallowed human like reasoning where conclusions are made with the available (often incomplete) knowledge and may be withdrawn in the presence of additional knowledge. This led to the development of the field of *nonmonotonic logic*, and several nonmonotonic logics such as circumscription, default logic, auto-epistemic logic, and nonmonotonic modal logics were proposed. The AI journal special issue of 1980 (volume 13, numbers 1 and 2) contained initial articles on some of these logics. In the last twenty years there have been several studies on these languages on issues such as representation of small common-sense reasoning examples, alternative semantics of these languages, and the relationship between the languages. But the dearth of efficient implementations, use in large applications – say of more than ten pages, and studies on building block support structures has for the time being diminished their applicability. Perhaps the above is due to some fundamental deficiency, such as: all of these languages which build on top of the classical logic syntax and allow nesting are quite complex, and all except default logic lack structure, thus making it harder to use them, analyze them, and develop interpreters for them.

An alternative nonmonotonic language paradigm with a different origin whose initial focus was to consider a subset of classical logic (rather than extending it) is the programming language PROLOG and the class of languages clubbed together as 'logic programming'. PROLOG and logic programming grew out of work on automated theorem proving and Robinson's resolution rule. One important landmark in this was the realization by Kowalski and Colmerauer that logic can be used as a programming language, and the term PROLOG was developed as an acronym from PROgramming in LOGic. A subset of first-order logic referred to as Horn clauses that allowed faster and simpler inferencing through resolution was chosen as the starting point. The notion of closed world assumption (CWA) in databases was then imported to PROLOG and logic programming and the negation as failure operator **not** was used to refer to negative information. The evolution of PROLOG was guided by concerns that it be made a full fledged programming language with

efficient implementations, often at the cost of sacrificing the declarativeness of logic. Nevertheless, research also continued on logic programming languages with declarative semantics. In the late 1980s and early 1990s the focus was on finding the right semantics for agreed syntactic sub-classes. One of the two most popular semantics proposed during that time is the *answer set semantics*, also referred to as the *stable model semantics*.

This book is about the language of logic programming with respect to the answer set semantics. We refer to this language as AnsProlog*, as a short form of '**Pro**gramming in **log**ic with **Ans**wer sets'[1]. In the following section we give an overview of how AnsProlog* is different from PROLOG and also the other nonmonotonic languages, and present the case for AnsProlog* to be the most suitable declarative language for knowledge representation, reasoning, and declarative problem solving.

1.1 Motivation: Why AnsProlog*?

In this section[2], for the purpose of giving a quick overview without getting into a lot of terminology, we consider an AnsProlog* program to be a collection of rules of the form:

$$L_0 \ or \cdots or \ L_k \leftarrow L_{k+1}, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n.$$

where each of the L_is is a literal in the sense of classical logic. Intuitively, the above rule means that if $L_{k+1}, \ldots, L_m$ are to be true and if $L_{m+1}, \ldots, L_n$ can be safely assumed to be false then at least one of $L_0, \ldots, L_k$ must be true.

This simple language has a lot going for it to be the leading language for knowledge representation, reasoning, and declarative problem solving. To start with, the nonclassical symbols $\leftarrow$, and **not** in AnsProlog* give it a structure and allow us to easily define syntactic sub-classes and study their properties. It so happens that these various sub-classes have a range of complexity and expressiveness thus allowing us to choose the appropriate sub-classes for particular applications. Moreover, there exists a more tractable approximate characterization which can be used – at the possible cost of completeness – when time is a concern. Unlike the other nonmonotonic logics, AnsProlog* now has efficient implementations which have been used to program large applications. In addition, the expressiveness studies show AnsProlog* to be as expressive as some of these logics, while syntactically it seems less intimidating as it does not allow arbitrary formulas. Finally, the most important reason to study and use AnsProlog* is that there is now a large body (much larger than for any other knowledge representation language) of support structure around AnsProlog*

[1] In the recent literature it has also been referred to as A-Prolog [BGN00, Gel01].
[2] In Section 1.2 we introduce more specific terminologies and use those in the rest of the book.

that includes the above mentioned implementations and theoretical building block results that allow systematic construction of AnsProlog* programs, and assimilation of new information. We now expand on these points in greater detail.

1.1.1 AnsProlog* vs PROLOG

Although, PROLOG grew out of programming with Horn clauses – a subset of first-order logic, several nondeclarative features were included in PROLOG to make it programmer friendly. We propose AnsProlog* as a declarative alternative to PROLOG. Besides the fact that AnsProlog* allows disjunction in the head of rules, the following are the main differences between AnsProlog* and Prolog.

- The ordering of literals in the body of a rule matters in PROLOG as it processes them from left to right. Similarly, the positioning of a rule in the program matters in PROLOG as it processes them from start to end. The ordering of rules and positioning of literals in the body of a rule do not matter in AnsProlog*. From the perspective of AnsProlog*, a program is a *set* of AnsProlog* rules, and in each AnsProlog* rule, the body is a *set* of literals and literals preceded by **not**.
- Query processing in PROLOG is top-down from query to facts. In AnsProlog* query-processing methodology is not part of the semantics. Most sound and complete interpreters with respect to AnsProlog* do bottom-up query processing from facts to conclusions or queries.
- Because of the top-down query processing, and start to end, and left to right processing of rules and literals in the body of a rule respectively, a PROLOG program may get into an infinite loop for even simple programs without negation as failure.
- The *cut* operator in PROLOG is extra-logical, although there have been some recent attempts at characterizing it. This operator is not part of AnsProlog*.
- There are certain problems, such as floundering and getting stuck in a loop, in the way PROLOG deals with negation as failure. In general, PROLOG has trouble with programs that have recursions through the negation as failure operator. AnsProlog* does not have these problems, and as its name indicates it uses the *answer set* semantics to characterize negation as failure.

In this book, besides viewing AnsProlog* as a declarative alternative to PROLOG, we also view PROLOG systems as top-down query answering systems that are correct with respect to a sub-class of AnsProlog* under certain conditions. In Section 8.4 we present these conditions and give examples that satisfy these conditions.

1.1.2 AnsProlog* vs Logic programming

AnsProlog* is a particular kind of logic programming. In AnsProlog* we fix the semantics to *answer set semantics*, and only focus on that. On the other hand logic programming refers to a broader agenda where different semantics are considered

as alternatives. We now compare AnsProlog (a sub-class of AnsProlog* with only one atom in the head, and without classical negation in the body) with the alternative semantics of programs with AnsProlog syntax.

Since the early days of logic programming there have been several proposals for semantics of programs with AnsProlog syntax. We discuss some of the popular ones in greater detail in Chapter 9. Among them, the most popular ones are the *stable model semantics* and the *well-founded semantics*. The stable models are same as the answer sets of AnsProlog programs, the main focus of this book. The well-founded semantics differs from the stable model semantics in that:

- Well-founded models are three-valued, while stable models are two valued.
- Each AnsProlog program has a unique well-founded model, while some AnsProlog programs have multiple stable models and some do not have any.

  For example, the program $\{p \leftarrow \textbf{not } p.\}$ has no stable models while it has the unique well-founded model where p is assigned the truth value *unknown.*

  The program $\{b \leftarrow \textbf{not } a., a \leftarrow \textbf{not } b., p \leftarrow a., p \leftarrow b.\}$ has two stable models $\{p, a\}$ and $\{p, b\}$ while its unique well-founded model assigns the truth value *unknown* to p, a, and b.

- Computing the well-founded model or entailment with respect to it is more tractable than computing the entailment with respect to stable models. On the other hand the latter increases the expressive power of the language.

As will be clear from many of the applications that will be discussed in Chapters 4 and 5, the nondeterminism that can be expressed through multiple stable models plays an important role. In particular, they are important for enumerating choices that are used in planning and also in formalizing aggregation. On the other hand, the absence of stable models of certain programs, which was initially thought of as a drawback of the stable model semantics, is useful in formulating integrity constraints whose violation forces elimination of models.

1.1.3 AnsProlog* vs Default logic

The sub-class AnsProlog can be considered as a particular subclass of default logic that leads to a more efficient implementation. Recall that a default logic is a pair (W, D), where W is a first-order theory and D is a collection of defaults of the type $\frac{\alpha:\beta_1,\ldots,\beta_n}{\gamma}$, where α, β, and γ are well-founded formulas. AnsProlog can be considered as a special case of a default theory where $W = \emptyset$, γ is an atom, α is a conjunction of atoms, and β_is are literals. Moreover, it has been shown that AnsProlog* and default logic have the same expressiveness. In summary, AnsProlog* is syntactically simpler than default logic and yet has the same expressiveness, thus making it more usable.

1.1.4 AnsProlog* vs Circumscription and classical logic

The connective '$\leftarrow$' and the negation as failure operator '**not**' in AnsProlog* add structure to an AnsProlog* program. The AnsProlog* rule $a \leftarrow b.$ is different from the classical logic formula $b \supset a$, and the connective '$\leftarrow$' divides the rule of an AnsProlog* program into two parts: the head and the body.

This structure allows us to define several syntactic and semi-syntactic notions such as: *splitting, stratification, signing*, etc. Using these notions we can define several subclasses of AnsProlog* programs, and study their properties such as: *consistency, coherence, complexity, expressiveness, filter-abducibility*, and *compilability to classical logic*.

The sub-classes and their specific properties have led to several building block results and realization theorems that help in developing large AnsProlog* programs in a systematic manner. For example, suppose we have a set of rules with the predicates $p_1, \ldots, p_n$ in them. Now if we add additional rules to the program such that $p_1, \ldots, p_n$ only appear in the body of the new rules, then if the overall program is consistent the addition of the new rules does not change the meaning of the original predicates $p_1, \ldots, p_n$. Additional realization theorems deal with issues such as: When can closed world assumption (CWA) about certain predicates be explicitly stated without changing the meaning of the modified program? How to modify an AnsProlog* program which assumes CWA so that it reasons appropriately when CWA is removed for certain predicates and we have incomplete information about these predicates?

The non-classical operator $\leftarrow$ encodes a form of directionality that makes it easier to encode causality, which can not be expressed in classical logic in a straightforward way. AnsProlog* is more expressive than propositional and first-order logic and can express transitive closure and aggregation that are not expressible in them.

1.1.5 AnsProlog* as a knowledge representation language

There has been extensive study about the suitability of AnsProlog* as a knowledge representation language. Some of the properties that have been studied are:

- When an AnsProlog* program exhibits *restricted monotonicity*. That is, it behaves monotonically with respect to addition of literals about certain predicates. This is important when developing an AnsProlog* program where we do not want future information to change the meaning of a definition.
- When is an AnsProlog* program *language independent*? When is it *language tolerant*? When is it *sort-ignorable*; i.e., when can sorts be ignored?
- When can new knowledge be added through filtering?

In addition it has been shown that AnsProlog* provides *compact representation* in certain knowledge representation problems; i.e., an equivalent representation in a tractable language would lead to an exponential blow-up in space. Similarly, it has been shown that certain representations in AnsProlog* can not be *modularly* translated into propositional logic. On the other hand problems such as constraint satisfaction problems, dynamic constraint satisfaction problems, etc. can be modularly represented in AnsProlog*. In a similar manner to its relationship with default logic, subclasses of other nonmonotonic formalisms such as auto-epistemic logic have also been shown to be equivalent to AnsProlog*.

Finally, the popular sub-class AnsProlog has a sound approximate characterization, called the well-founded semantics, which has nice properties and which is computationally more tractable.

1.1.6 AnsProlog* implementations: Both a specification and a programming language

Since AnsProlog* is fully declarative, representation (or programming) in AnsProlog* can be considered both as a specification and a program. Thus AnsProlog* representations eliminate the ubiquitous gap between specification and programming.

There are now some efficient implementations of AnsProlog* sub-classes, and many applications are built on top of these implementations. Although there are also some implementations of other nonmonotonic logics such as default logic (DeReS at the University of Kentucky) and circumscription (at the Linköping University), these implementations are very slow and very few applications have been developed based on them.

1.1.7 Applications of AnsProlog*

The following is a list of applications of AnsProlog* to database query languages, knowledge representation, reasoning, and planning.

- AnsProlog* has a greater ability than Datalog in expressing database query features. In particular, AnsProlog* can be used to give a declarative characterization of the standard *aggregate operators*, and recently it has been used to define new aggregate operators, and even data mining operators. It can also be used for querying in the presence of different kinds of incomplete information, including *null values*.
- AnsProlog* has been used in planning and allows easy expression of different kinds of (procedural, temporal, and hierarchical) domain control knowledge, ramification and qualification constraints, conditional effects, and other advanced constructs, and can be used for approximate planning in the presence of incompleteness. Unlike propositional logic, AnsProlog* can be used for conformant planning, and there are attempts to use AnsProlog* for planning with sensing and diagnostic reasoning. It has also been used for

assimilating observation of an agent and planning from the current situation by an agent in a dynamic world.
- AnsProlog* has been used in product configuration, representing constraint satisfaction problems (CSPs) and dynamic constraint satisfaction problems (DCSPs).
- AnsProlog* has been used for scheduling, supply chain planning, and in solving combinatorial auctions.
- AnsProlog* has been used in formalizing deadlock and reachability in Petri nets, in characterizing monitors, and in cryptography.
- AnsProlog* has been used in verification of contingency plans for shuttles, and also has been used in verifying correctness of circuits in the presence of delays.
- AnsProlog* has been used in benchmark knowledge representation problems such as reasoning about actions, plan verification, and the frame problem therein, in reasoning with inheritance hierarchies, and in reasoning with prioritized defaults. It has been used to formulate normative statements, exceptions, weak exceptions, and limited reasoning about what is known and what is not.
- AnsProlog* is most appropriate for reasoning with incomplete information. It allows various degrees of trade-off between computing efficiency and completeness when reasoning with incomplete information.

1.2 Answer set frameworks and programs

In this section we define the syntax of an AnsProlog* program (and its extensions and sub-classes), and the various notations that will be used in defining the syntax and semantics of these programs and in their analysis in the rest of the book.

An *answer set framework*[3] consists of two alphabets (an axiom alphabet and a query alphabet), two languages (an axiom language, and a query language) defined over the two alphabets, a set of axioms, and an entailment relation between sets of axioms and queries. The query alphabet will be closely associated with the axiom alphabet and the query language will be fairly simple and will be discussed later in Section 1.3.5. We will now focus on the axiom language.

Definition 1 The axiom alphabet (or simply the *alphabet*) of an answer set framework consists of seven classes of symbols:

(1) variables,
(2) object constants (also referred to as constants),
(3) function symbols,
(4) predicate symbols,
(5) connectives,
(6) punctuation symbols, and
(7) the special symbol $\bot$;

[3] In contrast logical theories usually have a single alphabet, a single language, and have inference rules to derive theorems from a given set of axioms. The theorems and axioms are both in the same language.

where the connectives and punctuation symbols are fixed to the set $\{\neg,\ or,\ \leftarrow,$ **not** $,`,'\}$ and $\{$ `(', `)', `.' $\}$ respectively; while the other classes vary from alphabet to alphabet. □

We now present an example to illustrate the role of the above classes of symbols. Consider a world of blocks in a table. In this world, we may have object constants such as $block1$, $block2, \ldots$ corresponding to the particular blocks and the object constant *table* referring to the table. We may have predicates on_table, and *on* that can be used to describe the various properties that hold in a particular instance of the world. For example, $on\_table(block1)$ means that $block1$ is on the table. Similarly, $on(block2, block3)$ may mean that *block2* is on top of *block3*. An example of a function symbol could be on_top, where $on\_top(block3)$ will refer to the block (if any) that is on top of block3.

Unlike the earlier prevalent view of considering logic programs as a subset of first order logic we consider answer set theories to be different from first-order theories, particularly with some different connectives. Hence, to make a clear distinction between the connectives in a first-order theory and the connectives in the axiom alphabet of an answer set framework, we use different symbols than normally used in first-order theories: *or* instead of $\vee$, and `,' instead of $\wedge$.

We use some informal notational conventions. In general, variables are arbitrary strings of English letters and numbers that start with an upper-case letter, while constants, predicate symbols and function symbols are strings that start with a lower-case letter. Sometimes – when dealing with abstractions – we use the additional convention of using letters $p, q, \ldots$ for predicate symbols, $X, Y, Z, \ldots$ for variables, $f, g, h, \ldots$ for function symbols, and $a, b, c, \ldots$ for constants.

Definition 2 A *term* is inductively defined as follows:

(1) A variable is a term.
(2) A constant is a term.
(3) If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term. □

Definition 3 A term is said to be *ground*, if no variable occurs in it. □

Definition 4 Herbrand Universe and Herbrand Base

- The Herbrand Universe of a language $\mathcal{L}$, denoted by $HU_{\mathcal{L}}$, is the set of all ground terms which can be formed with the functions and constants in $\mathcal{L}$.
- An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and each t_i is a term. If each of the t_is is ground then the atom is said to be ground.
- The Herbrand Base of a language $\mathcal{L}$, denoted by $HB_{\mathcal{L}}$, is the set of all ground atoms that can be formed with predicates from $\mathcal{L}$ and terms from $HU_{\mathcal{L}}$.

- A *literal* is either an atom or an atom preceded by the symbol $\neg$. The former is referred to as a positive literal, while the latter is referred to as a negative literal.

  A literal is referred to as ground if the atom in it is ground.
- A *naf-literal* is either an atom or an atom preceded by the symbol **not**.

  The former is referred to as a positive naf-literal, while the latter is referred to as a negative naf-literal.
- A *gen-literal* is either a literal or a literal preceded by the symbol **not**. □

Example 1 Consider an alphabet with variables X and Y, object constants a, b, function symbol f of arity 1, and predicate symbols p of arity 1. Let $\mathcal{L}_1$ be the language defined by this alphabet.

Then $f(X)$ and $f(f(Y))$ are examples of terms, while $f(a)$ is an example of a ground term. Both $p(f(X))$ and $p(Y)$ are examples of atoms, while $p(a)$ and $p(f(a))$ are examples of ground atoms.

The Herbrand Universe of $\mathcal{L}_1$ is the set $\{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$.

The Herbrand Base of $\mathcal{L}_1$ is the set $\{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))), p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}$. □

Definition 5 A *rule* is of the form:

$$L_0 \; or \cdots or \; L_k \leftarrow L_{k+1}, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n. \tag{1.2.1}$$

where L_is are literals or when $k = 0$, L_0 may be the symbol $\bot$, and $k \geq 0$, $m \geq k$, and $n \geq m$.

A rule is said to be ground if all the literals of the rule are ground.

The parts on the left and on the right of '$\leftarrow$' are called the *head* (or *conclusion*) and the *body* (or *premise*) of the rule, respectively.

A rule with an empty body and a single disjunct in the head (i.e., $k = 0$) is called a *fact*, and then if L_0 is a ground literal we refer to it as a ground fact.

A fact can be simply written without the $\leftarrow$ as:

$$L_0. \tag{1.2.2}$$

□

When $k = 0$, and $L_0 = \bot$, we refer to the rule as a *constraint*.

The $\bot$s in the heads of constraints are often eliminated and simply written as rules with empty head, as in

$$\leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n. \tag{1.2.3}$$