1 Logic programming applications: What are the abstractions and implementations?

Yanhong A. Liu
Computer Science Department, Stony Brook University
liu@cs.stonybrook.edu

This chapter presents an overview of applications of logic programming, classifying them based on the abstractions and implementations of logic languages that support the applications. The three key abstractions are join, recursion, and constraint. Their essential implementations are for-loops, fixed points, and backtracking, respectively. The corresponding kinds of applications are database queries, inductive analysis, and combinatorial search, respectively. We also discuss language extensions and programming paradigms, summarize example application problems by application areas, and touch on example systems that support variants of the abstractions with different implementations.

1.1 Introduction

Common reasoning with logic is the root of logic programming, which allows logic rules and facts to be expressed formally and used precisely for inference, querying, and analysis in general. Logic formalisms, or languages, allow complex application problems to be expressed declaratively with high-level abstractions and allow desired solutions to be found automatically with potentially efficient low-level implementations.

The biggest challenge in logic programming has been the need for efficient implementations. Much progress has been made, with efficient implementations in some cases beating manually written low-level code. However, inadequate performance in many cases has led to the introduction of non-declarative features in logic languages and resulted in the writing of obscure logic programs.

Despite the challenges, the most exciting aspect of logic programming is its vast areas of applications. They range from database queries to program analysis, from text processing to

decision making, from security to knowledge engineering, and more. These vast, complex, and interrelated areas make it challenging but necessary to provide a deeper understanding of the various kinds of applications in order to help advance the state of the art of logic programming and realize its benefits.

This chapter presents an overview of applications of logic programming based on a study of the abstractions and implementations of logic languages. The rationale is that abstractions and implementations are the enabling technologies of the applications. The abstractions are essential for determining what kinds of application problems can be expressed and how they can be expressed, for ease of understanding, reuse, and maintenance. The underlying implementations are essential for high-level declarative languages to be sufficiently efficient for substantial applications.

We discuss the following essential abstractions, where data abstractions are for expressing the data, and control abstractions are for expressing computations over the data:

1. data abstractions: objects and relationships;
2. control abstractions: (1) join, (2) recursion, and (3) constraint, which capture bounded, cyclic, and general computations, respectively.

In logic languages, the data abstractions as objects and relationships are essential for all three control abstractions.

The essential techniques for implementing the three control abstractions listed are (1) for-loops, (2) fixed points, and (3) backtracking, respectively. The corresponding kinds of applications are

(1) database-style queries, e.g., for ontology management, business intelligence, and access control;

(2) inductive analysis, e.g., for text processing, program analysis, network traversal, and trust management;

(3) combinatorial search, e.g., for decision making, resource allocation, games and puzzles, and administrative policy analysis.

We categorize application problems using these three control abstractions because they capture conceptually different kinds of problems, with inherently different implementation techniques, and at the same time correspond to very different classes of applications.

Note that the same application domain may use different abstractions and implementations for different problems. For example, enterprise software may use all three of traditional database queries, inductive analysis, and combinatorial search, for business intelligence and decision making; and security policy analysis and enforcement may use database-style queries

for access control, inductive analysis for trust management, and combinatorial search for administrative policy analysis.

We also discuss additional extensions, especially regular-expression paths for higher-level queries and updates for modeling actions; additional applications; and abstractions used in main programming paradigms. We also touch on several well-known systems while discussing the applications.

There is a large body of prior work, including surveys of logic programming in general and applications in particular, as discussed in Section 1.7. This chapter distinguishes itself from past work by analyzing classes of applications based on the language abstractions and implementations used.

The rest of the chapter is organized as follows. Section 1.2 presents essential abstractions in logic languages. Sections 1.3, 1.4, and 1.5 describe abstractions, implementations, and applications centered around join, recursion, and constraint. Section 1.6 discusses additional language extensions, applications, and programming paradigms. Section 1.7 discusses related literature and future directions.

1.2 Logic language abstractions

Logic languages provide very high-level data and control abstractions, using mostly very simple language constructs. We describe these abstractions and their meanings intuitively.

1.2.1 Data abstractions

All data in logic languages are abstracted, essentially, as objects and relationships.

Objects. Objects are primitive values, such as numbers and strings, or structured values whose components are objects.

Examples of primitive values are integer number `3` and string `'Amy'`. We enclose a string value in single quotes; if a string starts with a lower-case letter, such as `'amy'`, the quotes can be omitted, as has been conventional in logic languages.

Examples of structured values are `succ(3)`, `father(amy)`, and `cert('Amy',birth('2000-02-28','Rome'))`, denoting the successor integer of `3`, the father of `amy`, and the certificate that `'Amy'` was born on `'2000-02-28'` in `'Rome'`, respectively.

The names of structures, such as `succ`, `father`, `cert`, and `birth` above, are called function symbols. They correspond to object constructors in object-oriented languages.

Relationships. Relationships are predicates, or properties, that hold among objects. In particular, $p(o_1, \ldots, o_k)$, i.e., predicate p over objects $o_1, \ldots, o_k$ being true, is equiva-

lent to `(`o_1`,...,`o_k`) in` p, i.e., tuple `(`o_1`,...,`o_k`)` belonging to relation p—a table that holds the set of tuples of objects over which p is true.

Examples of relationships are `male(bob)`, `is_parent(bob,amy)`, and `issue(mario,'Amy',birth('2000-02-28','Rome'))`, denoting that `bob` is male, `bob` is a parent of `amy`, and `mario` issued a certificate that `'Amy'` was born on `'2000-02-28'` in `'Rome'`, respectively.

Structured values can be easily captured using relationships, but not vice versa. For example, `f` being the structured value `father(c)` can be captured using relationship `is_father(f,c)`, but relationship `is_parent(p,c)` cannot simply be captured as `p` being the structured value `parent(c)` when `c` has two parents.

Such high-level data abstraction allows real-world objects or their lower-level representations, from bits and characters to lists to sets, to be captured easily without low-level implementation details. For example,

- bits and characters are special cases of integers and strings, respectively.
- lists are a special case of linearly nested structured values, and
- sets are a special case of relations consisting of tuples of one component.

Objects and relationships can be implemented using well-known data structures, including linked list, array, hash table, B-tree, and trie, usually taking $O(1)$ or $O(\log n)$ time to access an object, where n is the size of the data.

1.2.2 Control abstractions

Control in logic languages is abstracted at a high level, as logical inference or logic queries over asserted relationships among objects:

- asserted relationships can be connected by logical connectives: conjunction (read "and"), disjunction (read "or"), negation (read "not"), implication (read "then"), backward implication (read "if"), and equivalence (read "if and only if");
- variables can be used in place of objects and be quantified over with universal quantifier (read "all") and existential quantifier (read "some"); and
- one can either infer all relationships that hold or query about certain relationships, among all objects or among certain objects.

Rules and facts are the most commonly supported forms in existing logic languages:

Rules. A rule is of the following form, where $assertion_0$ is called the conclusion, and other assertions are called the hypotheses. Each assertion is a predicate over certain objects,

where variables may be used in place of objects. Intuitively, left arrow (`<-`) indicates backward implication, comma (`,`) denotes conjunction, and all variables in a rule are implicitly universally quantified, i.e., the rule holds for all values of the variables.

assertion$_0$ `<-` *assertion*$_1$`, ...,` *assertion*$_k$`.`

For example, the second rule below says: `X` is a grandfather of `Y` if `X` is the father of `Z` and `Z` is a parent of `Y`, and this holds for all values of variables `X`, `Y`, and `Z`; the other rules can be read similarly. Following logic language conventions, names starting with an upper-case letter are variables.

```
is_parent(X,Y) <- is_father(X,Y).
is_grandfather(X,Y) <- is_father(X,Z), is_parent(Z,Y).
is_ancestor(X,Y) <- is_parent(X,Z), is_ancestor(Z,Y).
is_positive(succ(N)) <- is_positive(N).
```

The second rule is a join query—its two hypotheses have a shared variable, and it concludes a new predicate.

The third and fourth rules are recursive—the predicate in the conclusion depends on itself in a hypothesis, or in general possibly indirectly through another predicate.

Note that disjunction of a set of hypotheses can be expressed using a set of rules with the same conclusion.

Facts. A fact is a rule that has no hypotheses and is denoted simply as `assertion0.` For example, `is_father(bob,amy).` says that `bob` is the father of `amy`, and `is_positive(1).` says that `1` is positive.

The meaning of a set of rules and facts is the least set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the rules. This set can be computed by starting with the given facts and repeatedly applying the rules to conclude new facts—i.e., matching hypotheses of rules against facts, instantiating variables in rules with values in matched facts, and adding instantiated conclusions of rules as new facts. However,

- repeated application of rules might not terminate if function symbols are used in the rules, because facts about infinitely many new objects may be concluded, e.g., the fourth example rule above may infer `is_positive(succ(1))`, `is_positive(succ(succ(1)))`, and so on.
- when only certain relationships about certain objects are queried, application of rules may stop as soon as the query can be answered, e.g., if only `is_positive(succ(1))` is queried, application of rules can stop after one use of the given rule and the given fact.

Rules that do not contain function symbols are called Datalog rules. For example, the first three example rules given earlier in this section are Datalog rules.

General logic forms have also been increasingly supported, typically by extending the rule form above:

Negation in the hypotheses. A hypothesis in a rule may be prefixed with `not`, denoting negation of the asserted relationship.

For example, the following rule says: for all values of `X` and `Y`, `X` is the mother of `Y` if `X` is a parent of `Y` and `X` is not male.

```
is_mother(X,Y) <- is_parent(X,Y), not male(X).
```

Difficulties arise when negation is used with recursion. For example, what can be inferred from the following rule? Is `good(zak)` true or false?

```
good(zak) <- not good(zak).
```

More general forms. More general forms include disjunction and negation in the conclusion and, most generally, quantifiers `all` and `some` in any scope, not only the outermost scope. For example, the first rule below says: `X` is male or female if `X` is a person. The second rule says: `X` is not a winning position if, for all `Y`, there is no move from `X` to `Y` or else `Y` is a winning position.

```
male(X) or female(X) <- person(X).
not win(X) <- all Y: not move(X,Y) or win(Y).
```

The meaning of recursive rules with negation is not universally agreed upon. The two dominant semantics are well-founded semantics (WFS) [Van Gelder 1993, Van Gelder et al. 1991] and stable model semantics (SMS) [Gelfond and Lifschitz 1988]. Both WFS and SMS use the closed-world assumption, i.e., they assume that what cannot be inferred to be true from the given facts and rules, is false.

- WFS gives a single 3-valued model, with the additional truth value `undefined` besides `true` and `false`.
- SMS gives zero or more 2-valued models, using only `true` and `false`.

Other formalisms and semantics include partial stable models, also called stationary models [Przymusinski 1994]; first-order logic with inductive and fixed-point definitions, called FO(ID) and FO(FD) [Denecker and Ternovska 2008, Hou et al. 2010]; and recently proposed founded semantics and constraint semantics [Liu and Stoller 2018]. The first two are both aimed at unifying WFS and SMS. The last unifies and cleanly relates WFS, SMS, and other

major semantics by allowing assumptions about the predicates and rules to be declared explicitly.

For practical applications, logic languages often also support predefined relationships among objects, including equality, inequality, and general comparisons. Cardinality and other aggregates over relationships are often also supported.

1.2.3 Combinations of control abstractions

There are many possible combinations of the language constructs. We focus on the following three combinations of constructs as essential control abstractions. We identify them by join, recursion, and constraint. They capture bounded, cyclic, and general computations, respectively.

(1) **Join**—with join queries, no recursive rules, and restricted negation and other constructs; the restriction is that, for each rule, each variable in the conclusion must also appear in a hypothesis that is a predicate over arguments. Implementing this requires that common objects for the shared variables be found for the two hypotheses of a join query to be true at the same time; the number of objects considered are bounded, by the predicates in the hypotheses, following a bounded number of dependencies.

(2) **Recursion**—with join queries, recursive rules, and restricted negation and other constructs; the restriction is as for join above plus that a predicate in the conclusion of a rule does not depend on the negation of the predicate itself in a hypothesis. Implementing this requires repeatedly applying the recursive rules following cyclic dependencies, potentially an unbounded number of times if new objects are in some conclusions.

(3) **Constraint**—with join queries, recursive rules, and unrestricted negation and other constructs; unrestricted negation and other constructs can be viewed as constraints to be satisfied. Implementing this could require, in general, trying different combinations of variable values, as in general constraint solving.

Table 1.1 summarizes these three essential control abstractions and the corresponding kinds of computations and applications.

Table 1.1 Essential control abstractions of logic languages.

	Essential	Has join queries	Has rec. rules	Has neg. and others	Computations	Application kinds
(1)	Join	yes	no	restricted	bounded	database-style queries
(2)	Recursion	yes	yes	restricted	cyclic	inductive analysis
(3)	Constraint	yes	yes	unrestricted	general	combinatorial search

1.3 Join and database-style queries

Join queries are the most basic and most commonly used queries in relating different objects. They underlie essentially all nontrivial queries in database applications and many other applications.

1.3.1 Join queries

A join query is a conjunction of two hypotheses that have shared variables, concluding possible values of variables that satisfy both hypotheses. A conjunction of two hypotheses that have no shared variables, i.e., a Cartesian product, or a single hypothesis can be considered a trivial join query. A join query corresponds to a rule whose predicate in the conclusion is different from predicates in the hypothesis, so the rule is not recursive. A non-recursive rule with more than two hypotheses corresponds to multiple join queries, as a nesting or chain of join queries starting with joining any two hypotheses first.

For example, the first rule below, as seen before, is a join query. So is the second rule; it defines `sibling` over `X` and `Y` if `X` and `Y` have a same parent. The third rule defines a chain of red, green, and blue links from `X` to `Y` through `U` and `V`; it can be viewed as two join queries—join any two hypotheses first, and then join the result with the third hypothesis.

```
is_grandfather(X,Y) <- is_father(X,Z), is_parent(Z,Y).
sibling(X,Y) <- is_parent(Z,X), is_parent(Z,Y).
chain(X,Y) <- link(X,U,red), link(U,V,green), link(V,Y,blue).
```

In general, the asserted predicates can be about relationships among any kinds of objects—whether people, things, events, or anything else, e.g., students, employees, patients, doctors, products, courses, hospitals, flights, interviews, and hangouts; and the join queries can be among any kinds of relationships—whether family, friend, owning, participating, thinking, or any other relation in the real world or conceptual world.

Join queries expressed using rules correspond to set queries. For example, in a language that supports set comprehensions with tuple patterns [Liu et al. 2016, Rothamel and Liu 2007] the `is_grandfather` query corresponds to

```
is_grandfather = {(X,Y): (X,Z) in is_father, (Z,Y) in is_parent}
```

Without recursion, join queries can be easily supported together with the following extensions, with the restriction that, for each rule, each variable in the conclusion must also appear in a hypothesis that is a predicate over arguments, so the domain of the variable is bounded by the predicate; queries using these extensions can be arbitrarily nested:

- unrestricted negation, other connectives, and predefined relationships in additional conditions,
- aggregates, such as count and max, about the relationships, and
- general universal and existential quantifiers in any scope.

These subsume all constructs in the `select` statement for SQL queries. Essentially, join queries, with no recursion, relate objects in different relationships within a bounded number of steps.

1.3.2 Implementation of join queries

A join query can be implemented straightforwardly using nested for-loops and if-statements, where shared variables in different hypotheses correspond to equality tests between the corresponding variables. For example, the `is_grandfather` query earlier in this section can be implemented as

```
is_grandfather = {}
for (X,Z1) in is_father:       -- time factor: # is_father pairs
  for (Z2,Y) in is_parent:     -- time factor: # is_parent pairs
    if Z1 == Z2:
      is_grandfather.add(X,Y)
```

In a language that supports set comprehensions, such as Python, the above implementation can be expressed as

```
is_grandfather = {(X,Y) for (X,Z1) in is_father
                        for (Z2,Y) in is_parent if Z1 == Z2}
```

For efficient implementations, several key implementation and optimization techniques are needed, described below; additional optimizations are also needed, e.g., for handling streaming data or distributed data.

Indexing. This creates an index for fast lookup based on values of the indexed arguments of a relation; the index is on the shared arguments of the two hypotheses. For example, for any fact `is_father(X,Z)`, to find the matching `is_parent(Z,Y)`, an index called, say, `children{Z}`—mapping the value of `Z`, the first argument of `is_parent`, to the set of corresponding values of second argument of `is_parent`—significantly speeds up the lookup, improving the time factor for the inner loop to the number of children of `Z`:

```
is_grandfather = {(X,Y) for (X,Z) in is_father
                        for Y in children{Z}}
```

Join ordering. This optimizes the order of joins when there are multiple joins, e.g., in a rule with more than two hypotheses. For example, for the rule for `chain`, starting by joining the first and third hypotheses is never more efficient than starting by joining either of these hypotheses with the second hypothesis, because the former yields all pairs of red and blue links, even if there are no green links in the middle.

Tabling. This stores the result of common sub-joins so they are not repeatedly computed. Common sub-joins may arise when there are nested or chained join queries. For example, for the rule for `chain` earlier in this section, consider joining the first two hypotheses first: if there are many red and green link pairs from a value of `X` to a value of `V`, then storing the result of this sub-join avoids recomputing it when joining with blue links to find each target `Y`.

Demand-driven computation. This computes only those parts of relationships that affect a particular query. For example, a query may only check whether `is_father(dan,bob)` holds, or find all values of `X` for `is_father(dan, X)`, or find all `is_father` pairs, as opposed to finding all relationships that can be inferred.

Basic ideas for implementing the extensions negation, aggregates, etc. are as follows, where nested queries using these extensions are computed following their order of dependencies:

- negation, etc. in additional conditions: test them after the variables in them become bound by the joins.
- aggregates: apply the aggregate operation while collecting the query result of its argument.
- quantifiers: transform them into for-loops, or into aggregates, e.g., an existential quantification is equivalent to a count being positive.

Efficient implementation techniques for join queries and extensions have been studied in a large literature, e.g., [Ioannidis 1996]. Some methods also provide precise complexity guarantees, e.g., [Liu et al. 2016, Willard 2002].

1.3.3 Applications of join queries

Join queries are fundamental in querying complex relationships among objects. They are the core of database applications [Kifer et al. 2006], from enterprise management to ontology management, from accounting systems to airline reservation systems, and from electronic health records management to social media management. Database and logic programming are so closely related that one of the most important computer science bibliographies is called DBLP, and it was named after Database and Logic Programming [Ley 2002]. Join queries

also underlie applications that do not fit in traditional database applications, such as complex access control policy frameworks [ANSI INCITS 2004].

We describe three example applications below, in the domains of ontology management, enterprise management, and security policy frameworks. They all heavily rely on the use of join queries and optimizations, especially indexing. We give specific examples of facts, rules, and indexing for the first application.

Ontology management—Coherent definition framework (CDF). CDF is a system for ontology management that has been used in numerous commercial projects [Gomes et al. 2010], for organizing information about, e.g., aircraft parts, medical supplies, commercial processes, and materials. It was originally developed by XSB, Inc. Significant portions have been released in the XSB packages [Swift et al. 2014].

The data in CDF are classes and objects. For example, XSB, Inc. has a part taxonomy, combining UNSPSC (United Nations Standard Products and Services Code) and Federal INC (Item Name Code) taxonomies, with a total of over 87,000 classes of parts. The main relationships are variants of `isa`, `hasAttr`, and `allAttr`. Joins are used extensively to answer queries about closely related classes, objects, and attributes. Indexing and tabling are heavily used for efficiency. Appropriate join order and demand-driven computation are also important.

An example fact is as follows, indicating that specification `'A-A-1035'` in ontology `specs` has attribute `'MATERIAL'` whose value is `'ALUMINUM ALLOY UNS A91035'` in `material_taxonomy`. Terms `cid(Identifier, Namespace)` represent primitive classes in CDF.

```
hasAttr_MATERIAL(cid('A-A-1035',specs),
                 cid('ALUMINUM ALLOY UNS A91035',
                     material_taxonomy)).
```

An example rule is as follows, meaning that a part `PartNode` has attribute `'PART-PROCESS-MATERIAL'` whose value is process-material pair `(Process,Material)` in `'ODE Ontology'` if `PartNode` has attribute `'PROCESS'` whose value is `Process`, and `Process` has attribute `'PROCESS-MATERIAL'` whose value is `Material`.

```
hasAttr_PART-PROCESS-MATERIAL(PartNode,
        cid('process-material'(Process,Material),
            'ODE Ontology')) <-
  hasAttr_PROCESS(PartNode, Process),
  hasAttr_PROCESS-MATERIAL(Process, Material).
```

An example of indexing is for `hasAttr_ATTR`, for any `ATTR`, shown below, in XSB notation, meaning: use as index all symbols of the first argument if it is bound, or else do so for the second argument.

```
[*(1), *(2)]
```

XSB, Inc. has five major ontologies represented in CDF, for parts, materials, etc., with a total of over one million facts and five meta rules. The rules are represented using a Description Logic form—an ontology representation language. The example rule above is an instance of such a rule when interpreted. The indexing used supports different appropriate indices for different join queries.

CDF is used in XSB, Inc.'s ontology-directed classifier (ODC) and extractor (ODE) [Swift and Warren 2012]. ODC uses a modified Bayes classifier to classify item descriptions. For example, it is used quarterly by the U.S. Department of Defense to classify over 80 million part descriptions. ODE extracts attribute-value pairs from classified descriptions to build structured knowledge about items. ODC uses aggregates extensively, and ODE uses string pattern rules.

Enterprise management—Business intelligence (BI). BI is a central component of enterprise software. It tracks the performance of an enterprise over time by storing and analyzing historical information recorded through online transaction processing (OLTP), and is then used to help plan future actions of the enterprise. LogicBlox simplifies the hairball of enterprise software technologies by using a Datalog-based language [Aref et al. 2015, Green et al. 2012].

All data are captured as logic relations. This includes not only data as in conventional databases, e.g., sale items, price, and so on for a retail application, but also data not in conventional databases, e.g., sale forms, display texts, and submit buttons in a user interface. Joins are used for easily querying interrelated data, as well as for generating user interfaces. Many extensions such as aggregates are also used. For efficiency, exploiting the rich literature of automatic optimizations, especially join processing strategies and incremental maintenance, is of paramount importance.

Using the same Datalog-based language, LogicBlox supports not only BI but also OLTP and prescriptive and predictive analytics. "Today, the LogicBlox platform has matured to the point that it is being used daily in dozens of mission-critical applications in some of the largest enterprises in the world, whose aggregate revenues exceed $300B" [Aref et al. 2015].

Security policy frameworks—Core role-based access control (RBAC). RBAC is a framework for controlling user access to resources based on roles. It became an ANSI standard [ANSI INCITS 2004] building on much research during the preceding decade and

earlier, e.g., [Ferraiolo and Kuhn 1992, Ferraiolo et al. 2001, Gavrila and Barkley 1998, Landwehr et al. 1984].

Core RBAC defines users, roles, objects, operations, permissions, sessions and a number of relations among these sets; the rest of RBAC adds a hierarchical relation over roles, in hierarchical RBAC, and restricts the number of roles of a user and of a session, in constrained RBAC. Join queries are used for all main system functions, especially the `CheckAccess` function, review functions, and advanced review functions on the sets and relations. They are easily expressed using logic rules [Barker and Fernández 2006, Barker et al. 2004].

Efficient implementations rely on all main optimizations discussed, especially auxiliary maps for indexing and tabling [Liu et al. 2006]. Although the queries are like relational database queries, existing database implementations would be too slow for functions like `CheckAccess`. Unexpectedly, uniform use of relations and join queries also led to a simplified specification, with unnecessary mappings removed, undesired omissions fixed, and constrained RBAC drastically simplified [Liu and Stoller 2007].

1.4 Recursion and inductive analysis

Recursive rules are most basic and essential in relating objects that are an unknown number of relationships apart. They are especially important for problems that may require performing the inference or queries a non-predetermined number of steps, depending on the data.

1.4.1 Recursive rules and queries

Given a set of rules, a predicate p depends on a predicate q if p is in the conclusion of a rule, and either q is in a hypothesis of the rule or some predicate r is in a hypothesis of the rule and r depends on q. A given set of rules is recursive if a predicate p in the conclusion of a rule depends on p itself.

For example, the second rule below, as seen in Section 1.2.2, is recursive; the first rule is not recursive; the set of these two rules is recursive, where the first rule is the base case, and the second rule is the recursive case.

```
is_ancestor(X,Y) <- is_parent(X,Y).
is_ancestor(X,Y) <- is_parent(X,Z), is_ancestor(Z,Y).
```

In general, recursively asserted relationships can be between objects of any kind, e.g., relatives and friends that are an unknown number of connections apart in social networks, direct and indirect prerequisites of courses in universities, routing paths in computer networks, nesting of parts in products, supply chains in supply and demand networks, transitive role hierarchy relation in RBAC, and repeated delegations in trust management systems.

Recursive queries with restricted negation correspond to least fixed-point computations. For example, in a language that supports least fixed points, the `is_ancestor` query corresponds to the minimum `is_ancestor` set below, where, for any sets `S` and `T`, `S subset T` holds iff every element of `S` is an element of `T`:

```
min is_ancestor: is_parent subset is_ancestor,
                 {(X,Y): (X,Z) in is_parent,
                         (Z,Y) in is_ancestor} subset is_ancestor
```

With cyclic predicate dependencies, recursion allows the following restricted extensions to be supported while still providing a unique semantics; there is also the restriction that, for each rule, each variable in the conclusion must also appear in a hypothesis that is a predicate over arguments, as in extensions to join queries:

- stratified negation, where negation and recursion are separable, i.e., there is no predicate that depends on the negation of itself, and
- other connectives and predefined relationships in additional conditions, aggregates, and general quantifiers, as in extensions for join queries, when they do not affect the stratification.

Essentially, recursive rules capture an unbounded number of joins, and allow inference and queries by repeatedly applying the rules.

1.4.2 Implementation of recursive rules and queries

Inference and queries using recursive rules can be implemented using while-loops; for-loops with predetermined number of iterations do not suffice, because the number of iterations depends on the rules and facts. Each iteration applies the rules in one step, so to speak, until no more relevant facts can be concluded. For example, the `is_ancestor` query earlier in this section can be implemented as

```
is_ancestor = is_parent
while exists (X,Y): (X,Z) in is_parent, (Z,Y) in is_ancestor,
                    (X,Y) not in is_ancestor:
  is_ancestor.add((X,Y))
```

Each iteration computes the existential quantification in the condition of the while-loop, and picks any witness `(X,Y)` to add to the result set. It can be extremely inefficient to recompute the condition in each iteration after a new pair is added.

For efficient implementations, all techniques for joins are needed but are also more critical and more complex. In particular, to ensure termination,

- tabling is critical if relationships form cycles, and
- demand-driven computation is critical if new objects are created in the cycles.

For the `is_ancestor` query, each iteration computes the following set, which is a join, plus the last test to ensure that only a new fact is added:

```
{(X,Y): (X,Z) in is_parent, (Z,Y) in is_ancestor,
        (X,Y) not in is_ancestor}
```

Two general principles underlying the optimizations for efficient implementations are:

1. incremental computation for expensive relational join operations, with respect to facts that are added in each iteration.
2. data structure design for the relations, for efficient retrievals and tests of relevant facts.

For the restricted extensions, iterative computation follows the order of dependencies determined by stratification; additional aggregates, etc. that do not affect the stratification can be handled as described in Section 1.3.2 for computing the join in each iteration.

Efficient implementation techniques for recursive queries and extensions have been studied extensively, e.g., [Abiteboul et al. 1995]. Some methods also provide precise complexity guarantees, e.g., [Ganzinger and McAllester 2001, Liu and Stoller 2009, McAllester 1999, Tekle and Liu 2011].

1.4.3 Applications of recursive rules and queries

Recursive rules and queries can capture any complex reachability problem in recursive structures, graphs, and hyper-graphs. Examples are social network analysis based on all kinds of social graphs; program analysis over many kinds of flow and dependence graphs about program control and data values; model checking over labeled transition systems and state machines; routing in electronic data networks, telephone networks, or transportation networks; and security policy analysis and enforcement over trust or delegation relationships.

We describe three example applications below, in the domains of text and natural language processing, program analysis, and distributed security policy frameworks. They all critically depend on the use of recursive rules and efficient implementation techniques, especially tabling and indexing.

Text processing—Super-tokenizer. Super-tokenizer is an infrastructure tool for text processing that has been used by XSB, Inc.'s ontology-directed classifier (ODC) and extractor (ODE) for complex commercial applications [Swift and Warren 2012]. It was also developed originally at XSB, Inc.

Super-tokenizer supports the declaration of complex rewriting rules for token lists. For example, over 65,000 of these rules implement abbreviations and token corrections in ODC and complex pattern-matching rules in ODE for classification and extraction based on combined UNSPSC and Federal INC taxonomies at XSB, Inc. Recursion is used extensively in the super-tokenizer, for text parsing and processing. The implementation uses tabled grammars and trie-based indexing in fundamental ways.

Super-tokenizer is just one particular application that relies on recursive rules for text processing and, more generally, language processing. Indeed, the original application of Prolog, the first and main logic programming language, was natural language processing (NLP) [Pereira and Shieber 2002], and a more recent application in NLP helped the IBM Watson question answering system win the Jeopardy Man vs. Machine Challenge by defeating two former grand champions in 2011 [Lally and Fodor 2011, Lally et al. 2012].

Program analysis—Pointer analysis. Pointer analysis statically determines the set of objects that a pointer variable or expression in a program can refer to. It is a fundamental program analysis with wide applications and has been studied extensively, e.g., [Hind 2001, Sridharan et al. 2013]. The studies especially include significantly simplified specifications using Datalog in more recent years, e.g., [Smaragdakis and Balatsouras 2015], and powerful systems such as bddbddb [Whaley et al. 2005] and Doop [Bravenboer and Smaragdakis 2009], the latter built using LogicBlox [Aref et al. 2015, Green et al. 2012].

Different kinds of program constructs and analysis results relevant to pointers are relations. Datalog rules capture the analysis directly as recursively defined relations. For example, the well-known Andersen's pointer analysis for C programs defines a points-to relation based on four kinds of assignment statements [Andersen 1994], leading directly to four Datalog rules [Saha and Ramakrishnan 2005]. Efficient implementation critically depends on tabling, indexing, and demand-driven computation [Saha and Ramakrishnan 2005, Tekle and Liu 2011]. Such techniques were in fact followed by hand to arrive at the first ultra fast analysis [Heintze and Tardieu 2001a,b].

Indeed, efficient implementations can be generated from Datalog rules giving much better, more precise complexity guarantees [Liu and Stoller 2009, Tekle and Liu 2011, 2016] than the worst-case complexities, e.g., the well-known cubic time for Andersen's analysis. Such efficient implementation with complexity guarantees can be obtained for program analysis in general [McAllester 1999]. Commercial tools for general program analysis based on Datalog have also been built, e.g., by Semmle based on CodeQuest [Hajiyev et al. 2006].

Security policy frameworks—Trust management (TM). TM is a unified approach to specifying and enforcing security policies in distributed systems [Blaze et al. 1996, Grandi-

son and Sloman 2000, Ruohomaa and Kutvonen 2005]. It has become increasingly important as systems become increasingly interconnected, and logic-based languages have been used increasingly for expressing TM policies [Bonatti 2010], e.g., SD3 [Jim 2001], RT [Li et al. 2002], Binder [DeTreville 2002], Cassandra [Becker and Sewell 2004], and many extensions, e.g., [Becker et al. 2012, Sultana et al. 2013].

Certification, delegation, authorization, etc. among users, roles, permissions, etc. are relations. Policy rules correspond directly to logic rules. The relations can be transitively defined, yielding recursive rules. For example, one of the earliest TM frameworks, SPKI/SDSI [Ellison et al. 1999], for which various sophisticated methods have been studied, corresponds directly to a few recursive rules [Hristova et al. 2007], and efficient implementations with necessary indexing and tabling were generated automatically.

TM studies have used many variants of Datalog with restricted constraints [Li and Mitchell 2003], not unrestricted negation. A unified framework with efficient implementations is still lacking. For example, based on the requirements of the U.K. National Health Service, a formal electronic health records (EHR) policy was written, as 375 rules in Cassandra [Becker 2005a], heavily recursive. As the largest case study in the TM literature, its implementation was inefficient and incomplete—techniques like indexing were deemed needed but missing [Becker 2005b].

1.5 Constraint and combinatorial search

Constraints are the most general form of logic specifications, which easily captures the most challenging problem-solving activities such as planning and resource allocation.

1.5.1 Constraint satisfaction

A constraint is, in general, a relationship among objects but especially refers to cases when it can be satisfied with different choices of objects and the right choice is not obvious.

For example, the rule below says that `X` is a winning position if there is a move from `X` to `Y` and `Y` is not a winning position. It states a relationship among objects, but its meaning is not obvious, because the concluding predicates are recursively defined using a negation of the predicate itself.

```
win(X) <- move(X,Y), not win(Y).
```

In general, constraints can capture any real-world or conceptual-world problems, e.g., rules for moves in any game—whether recreational, educational, or otherwise; actions with conditions and effects for any planning activities; participants and resource constraints in scheduling—whether for university courses or manufacturer goods production or hospital surgeries; real-

world constraints in engineering design; as well as knowledge and rules for puzzles and brain teasers.

Given constraints may have implications that are not completely explicit. For example, the `win` rule implies not just the first constraint below, but also the second, by negating the conclusion and hypotheses in the given rule, following the closed-world assumption; the second constraint makes the constraint about `not win` explicit:

```
win(X) if some Y: move(X,Y) and not win(Y)
not win(X) if all Y: not move(X,Y) or win(Y)
```

Indeed, with general constraints, objects can be related in all ways using all constructs together with join and recursion: unrestricted negation, other connectives, predefined relationships, aggregates, and general quantifiers in any scope.

However, due to negation in dependency cycles, the meaning of the rules and constraints is not universally agreed on anymore.

- Well-founded semantics (WFS) gives a single, 3-valued model, where relationships that are true or false are intended to be supported from given facts, i.e., well-founded, and the remaining ones are `undefined`.
- Stable model semantics (SMS) gives zero or more 2-valued models, where each model stays the same, i.e., is stable, when it is used to instantiate all the rules; in other words, applying the rules to each model yields the same model.

For example, for the `win` example,

- if there is only one move, `move(a,b)`, not forming a cycle, then

  WFS and SMS both give that `win(b)` is false and `win(a)` is true;
- if there is only one move, `move(a,a)`, forming a self cycle, then

  WFS gives that `win(a)` is undefined, and

  SMS gives that there is no model;
- if there are only two moves, `move(a,b)` and `move(b,a)`, forming a two-move cycle, then

  WFS gives that `win(a)` and `win(b)` are both undefined, and

  SMS gives two models: one with `win(a)` true and `win(b)` false, and one with the opposite results.

Despite the differences, WFS and SMS can be computed using some shared techniques.

1.5.2 Implementation of constraint satisfaction

Constraint solving could in general use straightforward generate-and-test—generate each possible combination of objects for solutions and test whether they satisfy the constraints—but backtracking is generally used, as it is much more efficient.

Backtracking. Backtracking incrementally builds variable assignments for the solutions, and abandons each partial assignment as soon as it determines that the partial assignment cannot be completed to a satisfying solution, going back to try a different value for the last variable assigned; this avoids trying all possible ways of completing those partial assignments or naively enumerating all complete assignments.

For example, the `win(X)` query can basically try a move at each next choice of moves and backtrack to try a different move as soon as the current move fails. Expressed using recursive functions, this corresponds basically to the following:

```
def win(X): return (some Y: move(X,Y) and not_win(Y))
def not_win(X): return (all Y: not move(X,Y) or win(Y))
```

This backtracking answers the query correctly when the moves do not form a cycle. However, it might not terminate when the moves form a cycle, and the implementation depends on the semantics used. Both WFS and SMS can be computed by using and extending the basic backtracking:

- WFS computation could track cycles, where executing a call requires recursively making the same call, and infer undefined for those queries that have no execution paths to infer the query result to be true or false.
- SMS computation could generate possible partial or complete variable assignments, called grounding, and check them, possibly with the help of an external solver like Boolean satisfiability (SAT) solvers or satisfiability modulo theories (SMT) solvers.

For efficient implementations, techniques for join and recursion are critical as before, especially tabling to avoid repeated states in the search space. Additionally, good heuristics for pruning the search space can make drastic performance difference in computing SMS, e.g., as implemented in answer set programming (ASP) solvers.

Backjumping. One particular optimization of backtracking in SMS computation is backjumping. Backtracking always goes back one level in the search tree when all values for a variable have been tested. Backjumping may go back more levels, by realizing that a prefix of the partial assignment can lead to all values for the current variable to fail. This helps prune the search space.

For extensions that include additional constraints, such as integer constraints, as well as aggregates and quantifiers, an efficient solver such as one that supports mixed integer programming (MIP) can be used.

Efficient implementation techniques for constraint solving have been studied extensively, e.g., for ASP solvers [Gebser et al. 2012, Leone et al. 2006].

1.5.3 Applications of constraint satisfaction

The generality and power of constraints allow them to be used for all applications described previously, but constraints are particularly important for applications beyond those and that require combinatorial search. Common kinds of search problems include planning and scheduling, resource allocation, games and puzzles, and well-known NP-complete problems such as graph coloring, k-clique, set cover, Hamiltonian cycle, and SAT.

We describe three example applications, in the domains of decision making, resource allocation, and games and puzzles. They all require substantial use of general constraints and efficient constraint solvers exploiting backtracking, backjumping, and other optimizations.

Enterprise decision making—Prescriptive analysis. Prescriptive analysis suggests decision options that lead to optimized future actions. It is an advanced component of enterprise software. For example, for planning purposes, LogicBlox supports prescriptive analysis using the same Datalog-based language as for BI and OLTP [Aref et al. 2015, Green et al. 2012].

The data are objects and relations, same as used for BI, but may include, in particular, costs and other objective measures. Constraints capture restrictions among the objects and relations. When all data values are provided, constraints can simply be checked. When some data values are not provided, different choices for those values can be explored, and values that lead to certain maximum or minimum objective measures may be prescribed for deciding future actions. Efficient implementations can utilize the best constraint solvers based on the kinds of constraints used.

LogicBlox's integrated solution to decision making based on BI and OLTP has led to significant success. For example, for a Fortune 50 retailer with over $70 billion in revenue and with products available through over 2,000 stores and digital channels, the solution processes 3 terabytes of data on daily, weekly, and monthly cycles, deciding exactly what products to sell in what stores in what time frames; this reduces a multi-year cycle of a challenging task for a large team of merchants and planners to an automatic process and significantly increases profit margins [LogicBlox 2015a].

Resource allocation—Workforce management (WFM) in Port of Gioia Tauro. WFM handles activities needed to maintain a productive workforce. The WFM system for automobile

logistics in the Port of Gioia Tauro, the largest transshipment terminal in the Mediterranean, allocates available personnel of the seaport such that cargo ships mooring in the port are properly handled [Leone and Ricca 2015, Ricca et al. 2012]. It was developed using the DLV system [Leone et al. 2006].

The data include employees of different skills, cargo ships of different sizes and loads, teams and roles to be allocated, and many other objects to be constrained, e.g., workload of employees, heaviness of roles, and contract rules. Constraints include matching of available and required skills, roles, hours, etc., fair distribution of workload, turnover of heavy or dangerous roles, and so on. The constraints are expressed using rules with disjunction in the conclusion, general negation, and aggregates. The DLV system uses backtracking and a suite of efficient implementation techniques.

This WFM system was developed by Exeura s.r.l. and has been adopted by the company ICO BLG operating automobile logistics in the Port of Gioia Tauro [Leone and Ricca 2015], handling every day several ships of different sizes that moor in the port [Ricca et al. 2012].

Games and puzzles—N-queens. We use a small example in a large class of problems. The n-queens puzzle is the problem of placing `n` queens on a chessboard of `n`-by-`n` squares so that no two queens threaten each other, i.e., no two queens share the same row, column, or diagonal. The problem is old, well-studied, and can be computationally quite expensive [Bell and Stevens 2009].

The allowed placements of queens can be specified as logic rules with constraints. Naively enumerating all possible combination of positions and checking the constraints is prohibitively expensive. More efficient solutions use backtracking, and furthermore backjumping, to avoid impossible placement of each next queen as soon as possible. Stronger forms of constraints may also be specified to help prune the search space further [Gebser et al. 2012]. For example, backtracking can solve for one or two scores of queens in an hour, but backjumping and additional constraints help an ASP system like Clingo solve for 5000 queens in 3758.320 seconds of CPU time [Schaub 2014].

Many other games and puzzles can be specified and solved in a similar fashion. Examples are all kinds of crossword puzzles, Sudoku, Knight's tour, nonograms, magic squares, dominos, coin puzzles, graph coloring, palindromes, among many others, e.g., [Demoen et al. 2005, Edmunds 2015, Hett 2015, Kjellerstrand 2015, Malita 2015].

1.6 Further extensions, applications, and discussion

We discuss additional language extensions and applications, summarize applications based on the key abstractions used, touching on example logic programming systems, and finally put the abstractions into the perspective of programming paradigms.

1.6.1 Extensions

Many additional extensions to logic languages have been studied. Most of them can be viewed as abstractions that capture common patterns in classes of applications, to allow applications to be expressed more easily. Important extensions include:

- regular-expression paths, a higher-level abstraction for commonly-used linear recursion;
- updates, for real-world applications that must handle changes;
- time, for expressing changes over time, as an alternative to supporting updates directly;
- probability, to capture uncertainty in many challenging applications; and
- higher-order logic, to support applications that require meta-level reasoning.

We discuss two of the most important extensions below:

Regular-expression paths. A regular-expression path relates two objects using regular expressions and extensions. It allows repeated joins of a binary relation to be expressed more easily and clearly than using recursion; such joins capture reachability and are commonly used. For example, `is_ancestor(X,Y)`, defined in Section 1.4 using two rules including a recursive rule, can now be defined simply as below; it indicates that there are one or more `is_parent` relationships in a path from `X` to `Y`:

```
is_ancestor(X,Y) <- is_parent+(X,Y).
```

This is also higher-level than using recursion, because the recursive rule has to pick one of three possible forms below: with `is_parent` on the left, as seen before; with `is_parent` on the right; and with both conjuncts using `is_ancestor`.

```
is_ancestor(X,Y) <- is_parent(X,Z), is_ancestor(Z,Y).
is_ancestor(X,Y) <- is_ancestor(X,Z), is_parent(Z,Y).
is_ancestor(X,Y) <- is_ancestor(X,Z), is_ancestor(Z,Y).
```

Depending on the data, the performance of these forms can be asymptotically different in most implementations.

Regular-expression paths have many important applications including all those in Section 1.4, especially graph queries, with also parametric extensions for more general relations, not just binary relations [de Moor et al. 2003, Liu and Stoller 2006, Liu et al. 2004, Tekle et al. 2010].

Updates. An update, or action, can be expressed as a predicate that captures the update, e.g., by relating the values before and after the update and the change in value. The effect of the update could be taken immediately after the predicate is evaluated, similar to updates in common imperative languages, but this leads to lower-level control flows

that are harder to reason about. Instead, it is better for the update to take effect as part of a transaction of multiple updates that together satisfy high-level logic constraints. For example, with this approach, the following rule means that `adopted_by_from` holds if the updates `add_child` and `del_child` and the check `adoption_check` happen as a transaction.

```
adopted_by_from(C,X,Y) <- add_parent(X,C), del_parent(Y,C),
                          adoption_check(C,X,Y).
```

It ensures at a high-level that certain bad things won't happen, e.g., no child would end with one fewer parent or one more parent than expected. Transaction logic is an extension of logic rules for reasoning about and executing transactional state changes [Bonner and Kifer 1994]. LPS, a Logic-based approach to Production Systems, captures state changes by associating timestamps with facts and events, and this is shown to correspond to updating facts directly [Kowalski and Sadri 2015].

Logic languages with updates have important applications in enterprise software [Aref et al. 2015, Green et al. 2012]. Transaction logic can also help in planning [Basseda et al. 2014].

Additional implementation support can help enhance applications and enable additional applications. A particular helpful feature is to record justification or provenance information during program execution [Damásio et al. 2013, Roychoudhury et al. 2000], providing explanations for how a result was obtained. The recorded information can be queried to improve understanding and help debugging.

1.6.2 Additional applications

Many additional applications have been developed using logic programming, especially including challenging applications that need recursion and those that furthermore need constraint.

Table 1.2 lists example application areas with example application problems organized based on the main abstractions used. Note that application problems can often be reduced to each other, and many other problems can be reduced to the problems in the table. For example, model checking a property of a system [Clarke et al. 1994, 1999] can be reduced to planning, where the goal state is a state violating the property specified, so a plan found by a planner corresponds to an error trace found by a model checker [Cimatti et al. 2014, Edelkamp et al. 2014]. Administrative policy analysis also has correspondences to planning, by finding a sequence of actions to achieve the effect of a security breach [Stoller et al. 2011].

Table 1.2 is only a small sample of the application areas, with example application problems or kinds of application problems in those areas. Many more applications have been

Table 1.2 Example application areas with example application problems organized based on the main abstractions used. Applications discussed in some detail in this chapter are marked with an asterisk.

Area	Using join	Using recursion	Using constraint
Data management	business intelligence,* many database join queries	route queries, many database recursive queries	data cleaning, data repair
Knowledge management	ontology management*	ontology analysis	reasoning with knowledge
Decision support		supply-chain management, market analysis	prescriptive analysis,* planning, scheduling, resource allocation*
Linguistics		text processing,* context-free parsing, semantic analysis	context-sensitive analysis, deep semantics analysis
Program analysis	type checking, many local analyses	pointer analysis,* type analysis, many dependency analyses	type inference, many constraint-based analyses
Security	role-based access control*	trust management,* hierarchical role-based access control	administrative policy analysis, cryptanalysis
Games and puzzles		Hanoi tower, many recursion problems	n-queens,* Sudoku, many constraint puzzles
Teaching	course management	course analysis	question analysis, problem diagnosis, test generation

developed, in many more areas, using systems that support variants of the abstractions with different implementations. Some examples are:

- XSB has also been used to develop applications for immunization survey [Burton et al. 2012], standardizing data, spend analysis, etc. [XSB 2015], and it is discussed in many publications[1].

[1] A Google Scholar search with `+XSB +"logic programming"` returns about 2500 results, February 27, 2018.

- LogicBlox has also been used to create solutions for predicting consumer demand, optimizing supply chain, etc. [LogicBlox 2015b] and more [Green et al. 2012].
- ASP systems have been used in bioinformatics, hardware design, music composition, robot control, tourism, and many other application areas [Group 2005, Schaub 2011], including part of a decision support system for the Space Shuttle flight controller [Balduccini and Gelfond 2005, Nogueira et al. 2001].
- Logic systems have been developed for additional applications, e.g., PRISM [Sato and Kameya 1997] and ProbLog [De Raedt et al. 2007] for probabilistic models; XMC [Ramakrishnan et al. 2000] and ProB [Leuschel and Butler 2008] for verification; and NDlog [Loo et al. 2009], Meld [Ashley-Rollman et al. 2009], Overlog [Alvaro et al. 2010], and Bloom [Berkeley Orders of Magnitude] for network and distributed algorithms.

Languages and systems with more powerful features such as constraints for general applications are often also used in less challenging application areas such as those that need only join queries. For example, DLV has also been used in ontology management [Ricca et al. 2009].

1.6.3 Additional discussion on abstractions

We give an overview of the main abstractions in the larger picture of programming paradigms, to help put the kinds of applications supported into broader perspective.

The three main abstractions—join, recursion, and constraint—correspond generally to more declarative programming paradigms. Each is best known in its corresponding main programming community:

- Join in database programming. Database systems have join at the core but support restricted recursion and constraints in practice.
- Recursion in functional programming. Functional languages have recursion at the core but do not support high-level join or constraints.
- Constraint in logic programming. Logic engines support both join and recursion at the core, and have increasingly supported constraints at the core as well.

The additional extensions help further raise the level of abstraction and broaden the programming paradigms supported:

- Regular-expression paths raise the level of abstraction over lower-level linear recursion.
- Updates, or actions, are the core of imperative programming; they help capture real-world operations even when not used in low-level algorithmic steps.

- Time, probability, higher-order logic, and many other features correspond to additional arguments, attributes, or abstractions about objects and relationships.

One main paradigm not yet discussed is object-oriented programming. Orthogonal to data and control abstractions, objects in common object-oriented languages provide a kind of module abstraction, encapsulating both data structures and control structures in objects and classes. Similar abstractions have indeed been added to logic languages as well. For example, F-logic extends traditional logic programming with objects [Kifer et al. 1995] and is supported in Flora-2 [Kifer et al. 2014]; it was also the basis of a highly scalable commercial system, Ontobroker [Semafora 2012], and a recent industry suite, Ergo [Grosof et al. 2015]. For another example, ASP has been extended with object constructs in OntoDLV [Ricca et al. 2009].

Finally, building practical applications requires powerful libraries and interfaces for many standard functionalities. Many logic programming systems provide various such libraries. For example, SWI-Prolog has libraries for constraint logic programming, multithreading, interface to databases, GUI, a web server, etc, as well as development tools and extensive documentation.

1.7 Related literature and future work

There are many overview books and articles about logic programming in general and applications of logic programming in particular. This chapter differs from prior works by studying the key abstractions and their implementations as the driving force underlying vastly different application problems and application areas.

Kowalski [Kowalski 2014] provides an extensive overview of the development of logic programming. It describes the historical root of logic programming, starting from resolution theorem-proving; the procedural interpretation and semantics of rules with no negated hypotheses, called Horn clause programs; negation as failure, including completion semantics, stratification, well-founded semantics, stable model semantics, and ASP; as well as logic programming involving abduction, constraints, and argumentation. It focuses on three important issues: logic programming as theorem proving vs. model generation, with declarative vs. procedural semantics, and using top-down vs. bottom-up computation. Our description of abstractions and implementations aims to separate declarative semantics from procedural implementations.

Other overviews and surveys about logic programming in general include some that cover a collection of topics together and some that survey different topics separately. Example collections discuss the first 25 years of logic programming from 1974 [Apt et al. 1999] and the first 25 years of the Italian Association of Logic Programming from 1985 [Dovier and Pontelli 2010]. Example topics surveyed separately include logic programming semantics [Fitting

2002], complexity and expressive power [Dantsin et al. 2001], constraints [Jaffar and Maher 1994], ASP and DLV [Grasso et al. 2013], deductive databases [Abiteboul et al. 1995, Ceri et al. 1990, Minker et al. 2014, Ramakrishnan and Ullman 1995], and many more. Our description of abstractions and implementations is only a highly distilled overview of the core topics.

Overviews and surveys about logic programming applications in particular are spread across many forums. Example survey articles include an early article on Prolog applications [Roth 1993], DLV applications [Grasso et al. 2009, 2011, Leone and Ricca 2015], applications in Italy [Dal Palù and Torroni 2010], emerging applications [Huang et al. 2011], and a dedicated workshop AppLP—Applications of Logic Programming [Warren and Liu 2017]. For example, the early article [Roth 1993] describes six striking practical applications of Prolog that replaced and drastically improved over systems written previously using Fortran, C++, and Lisp. Example collections of applications on the Web include one at TU Wien [Group 2005], one by Schaub [Schaub 2011], and some of the problems in various competitions, e.g., as described by Gebser et al. [Gebser et al. 2017]. We try to view the applications by the abstractions and implementations used, so as to not be distracted by specific details of very different applications.

There are also many articles on specific applications or specific classes of applications. Examples of the former include team building [Ricca et al. 2012], program pointer analysis [Smaragdakis and Balatsouras 2015], and others discussed in this chapter. Examples of the latter include applications in software engineering [Ciancarini and Sterling 1995], DLV applications in knowledge management [Grasso et al. 2009], and IDP applications in data mining and machine learning [Bruynooghe et al. 2014]. We used a number of such specific applications as examples and described some of them in slightly more detail to illustrate the common technical core in addition to the applications per se.

Directions for future work. There are several main areas for future study: (1) more high-level abstractions that are completely declarative, (2) more efficient implementations with complexity guarantees, and (3) more unified and standardized languages and frameworks with rich libraries. These will help many more applications to be created in increasingly complex problem domains.

Acknowledgment

I would like to thank David S. Warren for his encouragement over the years at Stony Brook, and his patient and stimulating explanations about logic programming in general and XSB implementation in particular. I am grateful to Molham Aref, Francesco Ricca, and David Warren for helpful suggestions and additional information about applications using LogicBlox, DLV, and XSB, respectively. I thank Molham Aref and others at LogicBlox,

Jon Brandvein, Christopher Kane, Michael Kifer, Bob Kowalski, Bo Lin, Francesco Ricca, Scott Stoller, Tuncay Tekle, David Warren, Neng-Fa Zhou, and anonymous reviewers for helpful comments on drafts of this chapter. This work was supported in part by NSF under grants CCF-0964196, CCF-1248184, CCF-1414078, and IIS-1447549; and ONR under grant N00014-15-1-2208.

Bibliography

S. Abiteboul, R. Hull, and V. Vianu. 1995. *Foundations of Databases: The Logical Level*. Addison-Wesley.

P. Alvaro, T. Condie, N. Conway, J. Hellerstein, and R. Sears. 2010. I do declare: Consensus in a logic language. *ACM SIGOPS Operating Systems Review*, 43(4): 25–30.

L. O. Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen. http://www.diku.dk/forskning/topps/bibliography/1994.html.

ANSI INCITS, Feb. 2004. Role-Based Access Control. ANSI INCITS 359-2004, American National Standards Institute, International Committee for Information Technology Standards.

K. R. Apt, D. S. Warren, and M. Truszczynski, eds. 1999. *The Logic Programming Paradigm: A 25-Year Perspective*. Springer.

M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1371–1382.

M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. 2009. A language for large ensembles of independently executing nodes. In *Proceedings of the 25th International Conference on Logic Programming*, pp. 265–280. Springer.

M. Balduccini and M. Gelfond. 2005. Model-based reasoning for complex flight systems. In *Proceedings of the 5th AIAA Conference on Aviation, Technology, Integration, and Operations*.

S. Barker and M. Fernández. 2006. Term rewriting for access control. In *Data and applications security XX*, pp. 179–193. Springer.

S. Barker, M. Leuschel, and M. Varea. 2004. Efficient and flexible access control via logic program specialisation. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 190–199.

R. Basseda, M. Kifer, and A. J. Bonner. 2014. Planning with transaction logic. In *Proceedings of the 8th International Conference on Web Reasoning and Rule Systems*, pp. 29–44. Springer.

M. Y. Becker. 2005a. A formal security policy for an NHS electronic health record service. Technical Report UCAM-CL-TR-628, Computer Laboratory, University of Cambridge. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-628.html.

M. Y. Becker. 2005b. Cassandra: Flexible trust management and its application to electronic health records. PhD dissertation, Technical Report UCAM-CL-TR-648, Computer Laboratory, University of Cambridge. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-648.html.

M. Y. Becker and P. Sewell. 2004. Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pp. 139–154. IEEE CS Press. http://www.cl.cam.ac.uk/users/mywyb2/http://www.cl.cam.ac.uk/users/pes20/.

M. Y. Becker, A. Russo, and N. Sultana. 2012. Foundations of logic-based trust management. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pp. 161–175. IEEE CS Press.

J. Bell and B. Stevens. 2009. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1): 1–31.

Berkeley Orders of Magnitude, 2013. Bloom Programming Language. http://www.bloom-lang.net. Lastest release April 23, 2013. Accessed January 14, 2017.

M. Blaze, J. Feigenbaum, and J. Lacy. 1996. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 164–173.

P. A. Bonatti. 2010. Datalog for security, privacy and trust. In *Proceedings of the 1st International Conference on Datalog Reloaded*, pp. 21–36. Springer.

A. J. Bonner and M. Kifer. 1994. An overview of transaction logic. *Theoretical Computer Science*, 133(2): 205–265.

M. Bravenboer and Y. Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pp. 243–262.

M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. 2014. Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3. *Theory and Practice of Logic Programming*, pp. 1–35.

A. Burton, R. Kowalski, M. Gacic-Dobo, R. Karimov, and D. Brown. Oct. 2012. A formal representation of the WHO and UNICEF estimates of national immunization coverage: A computational logic approach. *PLOS ONE*.

S. Ceri, G. Gottlob, and L. Tanca. 1990. *Logic Programming and Databases*. Springer.

P. Ciancarini and L. Sterling. 1995. Report on the Workshop: Applications of Logic Programming in Software Engineering. *The Knowledge Engineering Review*, 10(01): 97–100.

A. Cimatti, S. Edelkamp, M. Fox, and E. Plaku, Nov. 23–28, 2014. Dagstuhl Seminar 14482: Automated Planning and Model Checking. http://www.dagstuhl.de/no_cache/en/program/calendar/semhp/?semnr=14482. Accessed June 6, 2015.

E. M. Clarke, O. Grumberg, and D. E. Long. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5): 1512–1542.

E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. 1999. *Model Checking*. MIT Press.

A. Dal Palù and P. Torroni. 2010. 25 years of applications of logic programming in Italy. In *A 25-Year Perspective on Logic Programming*, pp. 300–328. Springer.

C. V. Damásio, A. Analyti, and G. Antoniou. 2013. Justifications for logic programming. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 530–542. Springer.

E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3): 374–425.

O. de Moor, D. Lacey, and E. V. Wyk. 2003. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1–2): 15–35.

L. De Raedt, A. Kimmig, and H. Toivonen. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, pp. 2468–2473. Morgan Kaufman.

B. Demoen, P.-L. Nguyen, T. Schrijvers, and R. Troncon, 2005. The first 10 Prolog programming contests. http://dtai.cs.kuleuven.be/ppcbook/. Accessed May 20, 2015.

M. Denecker and E. Ternovska. 2008. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic*, 9(2): 14.

J. DeTreville. 2002. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 105–113. IEEE CS Press. ISBN 0-7695-1543-6.

A. Dovier and E. Pontelli, eds. 2010. *A 25-year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, GULP*. Springer.

S. Edelkamp, D. Magazzeni, and E. Plaku, Portsmouth, NH, June 23, 2014. Workshop on Model Checking and Automated Planning (MOCHAP'14). http://icaps14.icaps-conference.org/workshops_tutorials/mochap.html. Accessed June 6, 2015.

D. Edmunds, 2015. Learning constraint logic programming—finite domains with logic puzzles. http://brownbuffalo.sourceforge.net/. Accessed May 20, 2015.

C. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. Thomas, and T. Ylonen, Sept. 1999. RFC 2693: SPKI Certificate Theory. http://www.ietf.org/rfc/rfc2693.txt. Accessed June 4, 2015.

D. Ferraiolo and R. Kuhn. 1992. Role-based access control. In *Proceedings of the 15th NIST-NSA National Computer Security Conference*, pp. 554–563. Blatimore, Maryland.

D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. 2001. Proposed NIST standard for role-based access control. *ACM Transactions on Information and Systems Security*, 4(3): 224–274. http://doi.acm.org/10.1145/501978.501980.

M. Fitting. 2002. Fixpoint semantics for logic programming: A survey. *Theoretical Computer Science*, 278(1): 25–51.

H. Ganzinger and D. A. McAllester. 2001. A new meta-complexity theorem for bottom-up logic programs. In *Proceedings of the 1st International Joint Conference on Automated Reasoning*, pp. 514–528. Springer.

A. Gavrila and J. Barkley. 1998. Formal specification for RBAC user/role and role relationship management. In *Proceedings of the 3rd ACM Workshop on Role Based Access Control*, pp. 81–90.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool.

M. Gebser, M. Maratea, and F. Ricca. 2017. The sixth answer set programming competition. *J. Artif. Intell. Res.*, 60: 41–95.

M. Gelfond and V. Lifschitz. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 1070–1080. MIT Press.

A. S. Gomes, J. J. Alferes, and T. Swift. 2010. Implementing query answering for hybrid MKNF knowledge bases. In *Proceedings of the 12th International Conference on Practical Aspects of Declarative Languages*, pp. 25–39. Springer.

T. Grandison and M. Sloman. 2000. A survey of trust in Internet applications. *IEEE Communications Surveys and Tutorials*, 3(4): 2–16.

G. Grasso, S. Iiritano, N. Leone, and F. Ricca. 2009. Some DLV applications for knowledge management. In E. Erdem, F. Lin, and T. Schaub, eds., *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 591–597. Springer.

G. Grasso, N. Leone, M. Manna, and F. Ricca. 2011. ASP at work: Spin-off and applications of the DLV system. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning—Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, pp. 432–451. Springer.

G. Grasso, N. Leone, and F. Ricca. 2013. Answer set programming: Language, applications and development tools. In *Proceedings of the 7th International Conference on Web Reasoning and Rule Systems*, pp. 19–34. Springer.

T. J. Green, M. Aref, and G. Karvounarakis. 2012. LogicBlox, platform and language: A tutorial. In *Proceedings of the 2nd International Conference on Datalog in Academia and Industry*, Datalog 2.0, pp. 1–8. Springer.

B. Grosof, J. Bloomfield, P. Fodor, M. Kifer, I. Grosof, M. Calejo, and T. Swift. 2015. Automated decision support for financial regulatory/policy compliance, using textual rulelog. In *Proceedings of the RuleML 2015 Challenge, the Special Track on Rule-based Recommender Systems for the Web of Data, the Special Industry Track and the RuleML 2015 Doctoral Consortium*. http://ceur-ws.org/Vol-1417/.

T. W. K.-B. S. Group, Aug. 2005. WP5 report: Model applications and proofs-of-concept. http://www.kr.tuwien.ac.at/research/projects/WASP/report.pdf. Accessed May 20, 2015.

E. Hajiyev, M. Verbaere, and O. D. Moor. 2006. CodeQuest: Scalable source code queries with Datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pp. 2–27. Springer.

N. Heintze and O. Tardieu. 2001a. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp. 24–34. ISBN 1-58113-414-2. http://www.eecs.umich.edu/acal/swerve/docs/55-1.pdf.

N. Heintze and O. Tardieu. 2001b. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp. 254–263. http://www.cs.ucla.edu/~palsberg/course/cs232/papers/HeintzeTardieu-pldi01.pdf.

W. Hett, 2015. Prolog Site—Prolog Problems. http://sites.google.com/site/prologsite/prolog-problems/. Accessed May 28, 2015.

M. Hind. 2001. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pp. 54–61. DOI: http://doi.acm.org/10.1145/379605.379665.

P. Hou, B. De Cat, and M. Denecker. 2010. FO(FD): Extending classical logic with rule-based fixpoint definitions. *Theory and Practice of Logic Programming*, 10(4-6): 581–596.

K. Hristova, K. T. Tekle, and Y. A. Liu. 2007. Efficient trust management policy analysis from rules. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pp. 211–220.

S. S. Huang, T. J. Green, and B. T. Loo. 2011. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 1213–1216.

Y. E. Ioannidis. Mar. 1996. Query optimization. *ACM Computing Surveys*, 28(1): 121–123.

J. Jaffar and M. J. Maher. 1994. Constraint logic programming: A survey. *Journal of Logic Programming*, 19: 503–581.

T. Jim. 2001. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 106–115. IEEE CS Press.

M. Kifer, G. Lausen, and J. Wu. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4): 741–843.

M. Kifer, A. Bernstein, and P. M. Lewis. 2006. *Database Systems: An Application Oriented Approach, Complete Version*, 2nd. Addison-Wesley.

M. Kifer, G. Yang, H. Wan, and C. Zhao. July 2014. *Flora-2: User's Manual Version 1.0*. Stony Brook University. http://flora.sourceforge.net/. Accessed June 6, 2015.

H. Kjellerstrand, 2015. My Picat page. http://www.hakank.org/picat/. Accessed May 29, 2015.

R. Kowalski. 2014. Logic programming. In D. M. Gabbay, J. H. Siekmann, and J. Woods, eds., *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pp. 523–569. Elsevier.

R. Kowalski and F. Sadri. 2015. Reactive computing as model generation. *New Generation Computing*, 33(1): 33–67.

A. Lally and P. Fodor. Mar. 31 2011. Natural language processing with Prolog in the IBM Watson system. *Association for Logic Programming (ALP) Issue, Featured Articles*. \url{http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watsonsystem/}. Accessed April 23, 2015.

A. Lally, J. M. Prager, M. C. McCord, B. K. Boguraev, S. Patwardhan, J. Fan, P. Fodor, and J. Chu-Carroll. 2012. Question analysis: How Watson reads a clue. *IBM Journal of Research and Development*, 56(3/4): 2:1–2:13.

C. E. Landwehr, C. L. Heitmeyer, and J. McLean. 1984. A security model for military message systems. *ACM Transactions on Computer Systems*, 2(3): 198–222. ISSN 0734-2071. DOI: http://doi.acm.org/10.1145/989.991.

N. Leone and F. Ricca. 2015. Answer Set Programming: A tour from the basics to advanced development tools and industrial applications. In *Proceedings of the 11th International Summer School on Reasoning Web*, pp. 308–326. Springer.

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. July 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3): 499–562.

M. Leuschel and M. Butler. 2008. ProB: An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2): 185–203.

M. Ley. 2002. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, pp. 1–10. Springer.

N. Li and J. C. Mitchell. 2003. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pp. 58–73. Springer. ISBN 3-540-00389-4.

N. Li, J. C. Mitchell, and W. H. Winsborough. 2002. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pp. 114–130.

Y. A. Liu and S. D. Stoller. 2006. Querying complex graphs. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages*, pp. 199–214. Springer.

Y. A. Liu and S. D. Stoller. 2007. Role-based access control: A corrected and simplified specification. In *Department of Defense Sponsored Information Security Research: New Methods for Protecting Against Cyber Threats*, pp. 425–439. Wiley.

Y. A. Liu and S. D. Stoller. 2009. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6): 1–38.

Y. A. Liu and S. D. Stoller. Jan. 2018. Founded semantics and constraint semantics of logic rules. In *International Symposium on Logical Foundations of Computer Science*, volume 10703 of *Lecture Notes in Computer Science*, pp. 221–241. Springer.

Y. A. Liu, T. Rothamel, F. Yu, S. Stoller, and N. Hu. 2004. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 219–230.

Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. 2006. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation*, pp. 112–120.

Y. A. Liu, J. Brandvein, S. D. Stoller, and B. Lin. 2016. Demand-driven incremental object queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, pp. 228–241. ACM Press.

LogicBlox, 2015a. Assortment planning and management. http://www.logicblox.com/solution-four.html. Accessed May 18, 2015.

LogicBlox, 2015b. Solutions. http://www.logicblox.com/solutions.html. Accessed May 18, 2015.

B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. 2009. Declarative networking. *Communications of the ACM*, 52: 87–95.

M. Malita, 2015. Logic puzzles in Prolog. http://www.anselm.edu/internet/compsci/faculty_staff/mmalita/HOMEPAGE/logic/index.html. Accessed May 28, 2015.

D. A. McAllester. 1999. On the complexity analysis of static analyses. In *Proceedings of the 6th International Static Analysis Symposium*, pp. 312–329. Springer.

J. Minker, D. Seipel, and C. Zaniolo. 2014. Logic and databases: History of deductive databases. In D. Gabbay, J. Siekmann, and J. Woods, eds., *Handbook of Computational Logic*, chapter 17, pp. 571–628. North-Holland.

M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. 2001. An A-Prolog decision support system for the Space Shuttle. In *Practical Aspects of Declarative Languages*, pp. 169–183. Springer.

F. C. Pereira and S. M. Shieber. 2002. *Prolog and Natural-Language Analysis*. Microtome Publishing. \url{http://mtome.com/Publications/PNLA/pnla.html}. Revision of October 5, 2005.

T. C. Przymusinski. 1994. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(3): 141–187.

C. Ramakrishnan, I. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. Venkatakrishnan. 2000. XMC: A logic-programming-based verification toolset. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pp. 576–580. Springer.

R. Ramakrishnan and J. D. Ullman. 1995. A survey of deductive database systems. *Journal of Logic Programming*, 23(2): 125–149.

F. Ricca, L. Gallucci, R. Schindlauer, T. Dell'Armi, G. Grasso, and N. Leone. 2009. OntoDLV: An ASP-based system for enterprise ontologies. *Journal of logic and computation*, 19(4): 643–670.

F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano, and N. Leone. 2012. Team-building with answer set programming in the Gioia-Tauro Seaport. *Theory and Practice of Logic Programming*, 12(3): 361–381.

A. Roth. 1993. The practical application of Prolog. *AI Expert*, 8: 24–24. In Dr. Dobb's, http://www.drdobbs.com/parallel/the-practical-application-of-prolog/184405220, Dec.10, 2002. Accessed June 6, 2015.

T. Rothamel and Y. A. Liu. 2007. Efficient implementation of tuple pattern based retrieval. In *Proceedings of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, pp. 81–90.

A. Roychoudhury, C. Ramakrishnan, and I. Ramakrishnan. 2000. Justifying proofs using memo tables. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pp. 178–189.

S. Ruohomaa and L. Kutvonen. 2005. Trust management survey. In *Proceedings of the Third international conference on Trust Management*, pp. 77–92. Springer.

D. Saha and C. R. Ramakrishnan. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pp. 117–128.

T. Sato and Y. Kameya. 1997. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artifical Intelligence-Volume 2*, pp. 1330–1335. Morgan Kaufman.

T. Schaub, Mar. 2011. Collection on Answer Set Programming (ASP) and more. http://www.cs.uni-potsdam.de/~torsten/asp/. Accessed May 18, 2015.

T. Schaub, Dec. 23, 2014. Answer set solving in practice. http://www.cs.uni-potsdam.de/~torsten/Potassco/Slides/asp.pdf. Accessed May 20, 2015.

Semafora, 2012. Semantic infrastructure: OntoBroker. http://www.semafora-systems.com/en/products/ontobroker/. Accessed May 18, 2015.

Y. Smaragdakis and G. Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1): 1–69. http://yanniss.github.io/points-to-tutorial15.pdf.

M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*, pp. 196–232. Springer.

S. D. Stoller, P. Yang, M. I. Gofman, and C. Ramakrishnan. 2011. Symbolic reachability analysis for parameterized administrative role-based access control. *Computers & Security*, 30(2): 148–164.

N. Sultana, M. Y. Becker, and M. Kohlweiss. 2013. Selective disclosure in Datalog-based trust management. In *Proceedings of the 9th International Workshop on Security and Trust Management*, pp. 160–175. Springer.

T. Swift and D. S. Warren. 2012. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2): 157–187.

T. Swift, D. S. Warren, et al. June 2014. *The XSB System Version 3.5.x*. http://xsb.sourceforge.net. Accessed June 6, 2015.

K. T. Tekle and Y. A. Liu. 2011. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 661–672.

K. T. Tekle and Y. A. Liu. 2016. Precise complexity guarantees for pointer analysis via Datalog with extensions. *Theory and Practice of Logic Programming*, 16(5-6): 916–932.

K. T. Tekle, M. Gorbovitski, and Y. A. Liu. 2010. Graph queries through Datalog optimizations. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pp. 25–34.

A. Van Gelder. 1993. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1): 185–221.

A. Van Gelder, K. Ross, and J. S. Schlipf. 1991. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3): 620–650.

D. S. Warren and Y. A. Liu. Apr. 2017. AppLP: A dialogue on applications of logic programming. *Computing Research Repository*, arXiv:1704.02375 [cs.PL].

J. Whaley, D. Avots, M. Carbin, and M. S. Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems*, pp. 97–118. Springer.

D. E. Willard. 2002. An algorithm for handling many relational calculus queries efficiently. *Journal of Computer and System Sciences*, 65: 295–331.

XSB, 2015. Case Studies. http://www.xsb.com/case-studies. Accessed May 18, 2015.