Rulelog: Syntax and Semantics

Benjamin Grosof
Benjamin Grosof & Associates

Michael Kifer
Stony Brook University

February 26, 2014

Abstract

Rulelog is the logic underlying knowledge representation languages such as FLORA-2 and SILK. It combines much of the latest work in logic programming, non-monotonic reasoning, business rules, and the Semantic Web. It is designed to be appropriately expressive for supporting knowledge representation in complex domains, such as sciences and law, and yet to be efficiently implementable. This document provides a formal account of the syntax and semantics of Rulelog.

Contents

1 Introduction

Rulelog is the logic underlying knowledge representation languages such as FLORA-2 and SILK. It combines much of the latest work in logic programming, non-monotonic reasoning, business rules, and the Semantic Web. It is designed to be appropriately expressive for supporting knowledge representation in complex domains, such as sciences and law, and yet to be efficiently implementable. This document provides a formal account of the syntax and semantics of Rulelog.

Rulelog includes a novel combination of features that hitherto have not been present in a single system. Many of these features are drawn from earlier systems, including FLORA-2 [KYWZ13], SweetRules [GDG+], XSB [SW11], and others, and the design of Rulelog incorporates extensive feedback collected from the users of these systems.

1.1 Acknowledgments

An earlier attempt to integrate some of the key features of these systems was made as part of the Semantic Web Services Initiative and resulted in the *Semantic Web Services Language* [BBB+05], which was never finalized or implemented, however. The authors thank Mike Dean for his very useful comments on the earlier draft of this document. Members of the HalAR team provided very helpful comments and critique. This work was sponsored by Vulcan Inc. as part of the SILK project.

1.2 Typography

This document distinguishes several types of sections that may be of varying interest to different audiences:

- **Example:** Contains a Rulelog fragment.
- **Editor's Note:** Identifies an outstanding editorial issue.
- **Issue:** Identifies an outstanding design issue.
- **Important Change:** Identifies an important change with respect to the earlier specifications. May require reimplementation.
- **Rationale:** Discussions on why certain design decisions were made.

2 Organization of the Document

The syntax of Rulelog is divided in three parts: ***base language***, ***syntactic extensions***, and ***extra-logical markup***.

The base language includes those features of Rulelog to which we provide *direct* model-theoretic semantics. Like the base language, syntactic extensions

are used for knowledge representation in Rulelog. However, they are specified indirectly, via a translation into the base syntax, and no direct model theory is given to these features. In contrast to the base language and the extensions, the extra-logical markup is *not* used for knowledge representation per se. This markup includes various macros, like the CURIE macros, pragmas, compilation directives, etc.

Presentation of the semantics of Rulelog is split in two parts. The ***base semantics*** refers only to the base language of Rulelog; it provides direct meaning to the constructs of the base Rulelog language. The rest of the semantics is specified *axiomatically*, with the help of the ***background axioms***.

The base language includes Rulelog's most important features such as as frames, path expressions, higher-order predicates and functions, and defeasible rules. Defining the semantics of these high-level knowledge representation constructs via complex translations would have caused their meaning to be obscured and, literally, lost in translation, making it virtually impossible to use these constructs correctly.

A good example of extra-logical markup is the ***compact URI notation*** (or ***CURIEs***) [BM08], which is used throughout this document. A CURIE has the form *prefix*:*local-name*. It is a macro that evaluates into a concatenation of the value of the *prefix* part and the *local-name* part, and the resulting string is enclosed in single quotes. A *prefix* is an alphanumeric symbol whose value is defined by a special Rulelog statement (also part of the extra-logical markup); this value is a character sequence that is expected to have the form of an IRI. The *local-name* part is a character sequence that must be acceptable as a path component in a URI. A CURIE macro is expected to expand into a full IRI.

In this document we will use the following prefixes, which are defined in Rulelog by default and do not require explicit definitions:

```
rlg:    http://......./2013/Rulelog#
rdf:    http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:   http://www.w3.org/2000/01/rdf-schema#
owl:    http://www.w3.org/2002/07/owl#
xsd:    http://www.w3.org/2001/XMLSchema#
rif:    http://www.w3.org/2007/rif#
silk:   http://vulcan.com/2008/silk#
```

Although these prefixes can be redefined by the user, this is strongly discouraged.

3 The Syntax of Rulelog

This section describes the syntax of Rulelog. First we introduce the alphabet of the language followed by the syntax of the base language, the ***base syntax***. Then we define syntactic extensions followed by the extra-logical markup.

Throughout this section, we will be using a convention in which the symbols written in the monospace font will denote concrete elements of the language

while italicized symbols will denote non-specific examples. Specific examples will also use monospace. For instance, in "*string*"^^xsd#string the symbol *string* denotes *some* arbitrary string but the double quotes, ^^, and xsd#string are part of the language. Likewise, in "ab12"^^xsd#string, ab12 is a concrete sequence of the characters a, b, 1, and 2.

3.1 The Alphabet of Rulelog

The ***alphabet*** of Rulelog consists of *disjoint* subsets of symbols described below. Some additional symbols, including don't-care variables "?", Skolem constants, and local constants, are not part of the logic and are therefore defined as part of syntactic extensions in Section 3.4.

- A countably infinite set of ***constant symbols*** $\mathcal{C}onsts$. This set is specified in more details below.
- A countably infinite set of ***variable symbols*** $\mathcal{V}ars$.
  Variables are written as alphanumeric symbols prefixed by the character "?" (e.g., ?x, ?ABC). The alphanumeric part of a variable must begin with a letter or an underscore "_". (Jumping ahead, the symbol "?" by itself is also a variable, but it is an extension and *not* part of the base syntax.)
- ***Negation*** symbols: naf (*default negation*) and neg (*explicit negation*).
- ***Frame construction*** symbols: -> (*has value*), => (*has type*).
- ***Unification*** and ***equality*** symbols: = (unifies), :=:(logically equal), != (does not unify), !=!(not equal).
- ***Class membership*** and ***subclass relationship*** symbols: :, ::.
- ***Connectives***: and, or, :-, <~~, ~~>, <~~>, <==, ==>, <==>.
- ***Quantifiers***: exists, forall.
- ***Aggregation*** operators: min, max, count, sum, avg, prod.
- ***Comprehension*** operators: setof, bagof.

  Spec Change 3.1: The symbols collectset and collectbag are deprecated: too long. □

- ***Auxiliary symbols*** (,), [,], , , <, >, |, ?, @, ^^.

Editor's Note 3.1: Functions and predicates with named arguments are *not* included in the syntax at this point. □

The set of constants, $\mathcal{C}onsts$, can be broadly divided into two *disjoint* categories: ***uninterpreted symbols*** and ***interpreted symbols***. The main kinds

of uninterpreted symbols are ***Rulelog abstract symbols*** and ***IRIs***. These two types of symbols are disjoint. The main kinds of interpreted symbols are data types. Other kinds of interpreted and uninterpreted symbols may be introduced in the future.

The syntax for Rulelog abstract symbols and IRIs is given below. In addition, a Rulelog constant can have the form "*some string*"^^*symbol_space_name*. In this case it is called a constant with an ***explicit symbol space***. Typically such constants belong to one of the supported ***datatypes*** (see below) but other, non-datatype symbol spaces may be introduced later.

- ***Uninterpreted symbols.***
  - ***Rulelog abstract symbols***. A Rulelog abstract symbol is a sequence of characters enclosed between a pair of single quotes. For instance, `'abc#$%'`. Single quotes that are part of a symbol are escaped with a backslash.

    For example, the symbol `a'bc''d` should be written as `'a\'bc\'\'d'`.

    A backslash is escaped with another backslash. Symbols that consist exclusively of alphanumeric characters and the underscore (_) and begin with a letter or an underscore do not need to be quoted.

    For example: `abc1` or `_abc`. Compare this to `'12abc'` and `'abc#de'`, which must be quoted.
  - ***IRIs*** or ***international resource identifiers***. An IRI is a sequence of characters enclosed between the angle brackets. All printable ASCII characters are allowed except for `<` and `>`.

    For example: `<mailto:silk-announce@semwebcentral.org>`, `<http://w3.org/>`. The intent is that the enclosed sequence of characters forms a syntactically valid IRI as specified in [DG05]. However, Rulelog allows malformed IRIs as well.

    Remark 3.1: This syntax for IRIs is hard to parse because in `?X=<0,1>?Y` one has to do unbounded look-ahead in order to decide that this is a conjunction of `?X=<0` and `1>?Y` rather than `?X=<0,1>` followed by an illegal token `?Y`. Due to this difficulty, parsers might opt for the second possibility and require a space after `?X=<0`.

    Since IRIs are typically very long, a CURIE macro notation is a typical way of coping with this problem, as described in Section 2. Rulelog reserves IRI constants that have the following CURI prefixes: `silk`, `rlg`, `rdf`, `rdfs`, `owl`, `xsd`, `rif`. For instance, `rlg#defeated`.

    Issue 3.1: Decide what kind of restrictions are to be imposed on the use of these things. For instance, can such predicates be defined by the user? □

- ***Interpreted symbols: datatypes.*** Rulelog datatypes are modeled after XML Schema datatypes [BM04]. A datatype constant is a constant with an explicit symbol space, i.e., it has the form `"`*string representation*`"^^`*typename* where *string representation* is a sequence of characters and *typename* is the name of the data type. The character sequence *string* must belong to the *lexical space* of the data type (see Section 3.2). To include a double-quote "`"`" inside *string* one must escape it with a backslash. For instance, `"ab\"cd\"gf"^^\string`.

  The sets of constants that belong to different datatypes are disjoint from each other. The following is a list of supported datatypes in Rulelog.

  - ***Strings*** have the form `"`*string*`"^^\string`. To simplify the use of strings, Rulelog provides a shortcut notation in which a string of the form `"`*string*`"^^\string` can be written simply as `"`*string*`"`. For instance, `"abcde"` instead of `"abcde"^^\string`.
  - ***Integers*** have the form `"`*integer*`"^^\integer`, where *integer* is an integer value such as `123` or `-25`. A shortcut for integers in Rulelog is simply to write them as above, i.e., `123` and so on.
  - ***Doubles*** have the form `"`*double*`"^^\double`, where *double* is an floating point number such as `123.45`, `34.2e12`, or `-2e-5`. A shortcut for double numbers in Rulelog is simply to write them as above, i.e., `123.45` or `34.2e12`.

Editor's Note 3.2: Add more datatypes. □

Issue 3.2: XSB and Flora do not parse 23e5. They require 23.0e5. Do we want to allow this? □

Issue 3.3: Flora does not treat the full and the shortcut representation of strings and numbers synonymously. One solution is to change Flora to translate the long representation of these types into their short representation.
Also, Flora does not implement decimals because XSB does not. It maps them to doubles (which is contrary to the semantics of XML Schema). □

3.2 Symbol Spaces and Datatypes

A ***symbol space*** consists of

- A *name* that uniquely identifies that symbol space among all symbol spaces.
- A ***lexical space*** — the set of all character sequences that represent valid constants for that symbol space. For a symbol space named α, its lexical space will be denoted $\mathcal{C}onsts_\alpha$.

A constant belongs to a symbol space, *sym*, if and only if it has the form "*charseq*"^^*sym* and *charseq* belongs to the lexical space of *sym*.

Example 3.1: The following are Rulelog constants:

```
"12345"^^\integer
"123.345"^^\double
```

because the sequence of characters `12345` is a syntactically valid integer and `123.345` is a syntactically correct double-precision number. Thus, each belongs to the lexical space of integers and doubles, respectively. In contrast, `"123.45"^^\integer` is *not* a Rulelog constant because `123.45` is not an integer. □

A ***datatype*** is a symbol space that, in addition, has

- A set called that datatype's ***value space***. For a datatype named α, its value space will be denoted $\mathbf{V}_\alpha$.
- A mapping from the datatype's lexical space to its value space, called ***lexical-to-value-space mapping***.

For the datatypes introduced in Section 3.1, these notions are as follows:

- *Strings.* The symbol space name is `\string`. Both the lexical and the value spaces $\mathcal{C}onsts_{\texttt{\string}}$ and $\mathbf{V}_{\texttt{\string}}$ consist of the set of all character sequences, and the lexical-to-value-space mapping is the identity mapping.
- *Integers.* The symbol space name is `\integer`. The lexical space $\mathcal{C}onsts_{\texttt{\integer}}$ consists of character strings that represent syntactically valid positive or negative integers as well as 0. The value space $\mathbf{V}_{\texttt{\integer}}$ is the set of all integer numbers and the lexical-to-value-space map maps any character string that represents an integer into that integer.
- *Doubles.* The symbol space name is `\double`. The lexical space $\mathcal{C}onsts_{\texttt{\double}}$ is the set of all character strings that represent doubles, as specified in the XML Datatypes document [BM04]. The value space $\mathbf{V}_{\texttt{\double}}$ is the set of all 64-bit floating point numbers. The lexical-to-value-space map maps each string in the lexical space into the floating point number it represents.

Remark 3.2: Note that the lexical-to-value-space mapping for integers is not a 1-1 mapping. For example, `"2"^^\integer` and `"+2"^^\integer` are mapped into the same integer number in the value space of `\integer`.
The same is true about the mapping for the doubles. For instance, `"1.2e2"^^\double` and `"12e1"^^\double` are mapped into the same floating point number in the value space $\mathbf{V}_{\texttt{\double}}$.

3.3 The Base Language

Rulelog has several types of *terms* and *formulas*. In this section we first define the notion of a term, then of a basic formula, and then of a compound formula, of a rule, and of a query.

3.3.1 Terms

A (pure) ***HiLog term*** has one of these forms:

- A constant from the set $\mathcal{C}onsts$; or
- A variable in $\mathcal{V}ars$; or
- An expression of the form $t(t_1, ..., t_n)$, where $n \geq 0$ and t, t_1, ..., t_n are HiLog terms. When $n = 0$, the term has the form $t()$.

In Section 3.4.1, we will introduce *generalized* HiLog terms, which incorporate *path expressions* into the above syntax.

For example: `f(?X,abc)`, `?X(a,?Y)`, `?X(a)(?Y,b)`, `b(?X,?Y)`, `abc`, `p()`, `p()q()r()`, `b(?X,c)e(?Z)(u,22)`.

A ***first-order term*** has an even more restricted form of a HiLog term: If is either

- A constant from the set $\mathcal{C}onsts$; or
- A variable in $\mathcal{V}ars$; or
- An expression of the form $t(t_1, ..., t_n)$, where t is a constant in $\mathcal{C}onsts$ and t_1, ..., t_n are first-order terms. When $n = 0$, the term has the form $t()$.

Note that this precludes variables over function symbols.

A ***list term*** has one of these two forms:

- `[`t_1`,...,`t_n`]`
- `[`t_1`,...,`t_n `|`$rest$`]`

where t_1,...,t_n, and $rest$ are terms.

For example: `[a,b,?X(?Z,s),c(p)]`, `[a,?X,c|?Y]`.

Editor's Note 3.3: Not sure if it is worth giving a model-theoretic semantics for lists. If not, we'll move them to Syntactic Extensions. □

A ***reification term*** is an expression of the form ${$\varphi$}, where φ is a Rulelog formula (see Section 3.3.2).

For example: `${p(a,b)}`, `${a[p->b,q->c]}`, ${p(?X) `:-q(?X,?Y)`}.

Editor's Note 3.4: Need to see what kind of formulas can go under reification or instance, quantifiers? □

An ***aggregation term*** is an expression that has one of the following forms:

aggr{ *ResultVar* | *Query* }
aggr{ *ResultVar* [*GroupByVars*] | *Query* }

where

- *aggr* is the name of the aggregate operation, which can be `sum`, `avg`, `min`, `max`, `count`, or `prod`.
- *ResultVar* is a variable.
- *Query* is a query, as described in Section 3.3.4.
- *GroupByVars* is a comma-separated list of group-by variables.

For example: `avg{?X|o[p->?X], ?X[q->cde]}`,
`count{?X[?Y,?Z] |o[p->?X[?Y->cde], q->?Z]}`.

Editor's Note 3.5: The `prod` operator is not implemented. Also rarely used. Keep it? □

An ***comprehension term*** is an expression that has one of the following forms:

cmpr{ *ResultVar* | *Query* }
cmpr{ *ResultVar* [*GroupByVars*] | *Query* }
cmpr{ *ResultVar* (*SortSpec*) | *Query* }
cmpr{ *ResultVar* [*GroupByVars*](*SortSpec,...,SortSpec*) | *Query* }

where

- *cmpr* is the name of the comprehension operation, which can be `setof` or `bagof`.
- *ResultVar* is a variable.
- *Query* is a query, as described in Section 3.3.4.
- *GroupByVars* is a comma-separated list of group-by variables.
- `SortSpec` can be `asc`, `desc`, or a list of the form `[`*spec*,...,*spec*`]`, where each *spec* has the form `asc(`*N*`)` or `desc(`*N*`)`.

Example 3.2: The following are examples of comprehension in Rulelog:
```
setof{?X(desc) |o[p->v[?Y->cde], q->?Z], ?X=f(?Y,?Z)}
setof{?X[?Y,?Z]([asc(2),desc(1)]) |o[p->?X[?Y->cde], q->?Z]}
bagof{?X |o[p->v[?Y->cde], q->?Z], ?X=f(?Y,?Z)}
bagof{?X[?Y,?Z] |o[p->?X[?Y->cde], q->?Z]}
```
□

A ***parenthesized term*** is an expression of the form (T), where T is a term. In Rulelog a parenthesized term *(T)* is allowed wherever T is allowed.

Parenthesizing is typically used for clarity or to override associativity. For instance, we shall see that semantically a path expressions of the form `a.b.c` is the same as `(a.b).c`. If we want to change this default associativity, we could write, for example, `a.(b.c)`.

3.3.2 Basic Formulas

A Rulelog ***basic formula*** plays a role similar to first-order atomic formulas. We call them *basic* rather than atomic because these formulas are not really atomic: they are equivalent to conjunctions of atomic formulas. Basic formulas will also sometimes be called ***positive literals***.

In Rulelog *every* formula is associated with a module explicitly or implicitly. ***Explicit association*** happens when a basic formula occurs as a subformula of a *foreign formula* (see Section 3.3.3). ***Implicit association*** happens when a Rulelog rule set is loaded into the system. Every Rulelog rule set is loaded into a specific module, M, and all basic formulas that are not explicitly associated with a module become associated implicitly with that module M. This is explained in Section 3.6.

- Any Rulelog term is a formula, called ***term formula***. (Jumping ahead, only reification terms are of any significance as formulas. All other terms will uniformly be considered as false.)
- A ***HiLog formula*** is a basic formula of the form H, where H is a HiLog term. Although syntactically a HiLog formula may look identical to a HiLog term, the former is associated with a module and, as we shall see, it is interpreted differently by the Rulelog semantics.
- An ***equality formula*** has the form T `:=:` S, where T and S are HiLog terms.
- A ***disequality formula*** has the form T `!=!` S, where T, S are HiLog terms.
- A ***unification formula*** has the form T `=` S, where T, S are HiLog terms.
- A ***disunification formula*** has the form T `!=` S, where T, S are HiLog terms.
- A ***comparison formula*** has the form T *op* S, where *op* is `<`, `>`, `>=`, `=<`. Again, T and S are HiLog terms.

Editor's Note 3.6: More comparisons? E.g., `@<`, `@>`, `@=<`, `@>=` □

- An **evaluable expression** is a formula of the form T `is` *Expr*, where T is a term and *Expr* is an expression.

For example: `?X is 2+min(3,?Z)`, `14 is 5*3-1`, `a.b.c is ?X*?Y`.

Issue 3.4: The form of *Expr* needs to be sorted out. For now, this is an *arithmetic* expression as in XSB and FLORA-2, but we might want to add string manipulation and others. □

Issue 3.5: Need to add sensor/builtin formulas. We don't want their names to be returned by HiLog variables. So, we either put their names into a separate domain (or do something similar). □

◦ A ***membership formula*** has the form P`:`T where both P and T are path expressions.
◦ A ***subclass formula*** has the form P`::`T where both P and T are path expressions.

◦ A ***frame formula*** is a statement of the form $P[Spec_1, ..., Spec_n]$, where $n \geq 0$, P is a term, and each $Spec_i$ has one of these forms:

- Q_i, where Q_i is a term.
- Q_i`->`V_i, where Q_i is a term and V_i is a term or a frame formula.
- Q_i`->`$\{W_{i,1}, ..., W_{i,k}\}$, where $k \geq 0$ and each $W_{i,j}$ is a term or a frame formula.
- `=>`Q_i, where Q_i is a term.
- Q_i`=>`W_i, where Q_i and W_i are terms.

Note that $n = 0$ and $k = 0$ are possible, so `foo[]` and `foo[bar->{}]` are frame formulas.

In a frame formula, the Q_is are called ***methods*** (or ***properties***, if they are just constants) and the V_i, $W_{i,j}$ are called *values* or *results*. In the first case, when $Spec_i$ is Q_i, the method has no explicitly specified value and is called a ***Boolean method***.

Example 3.3: The following are frame formulas:

```
a[p(foo,bar)->v(?Z), q(?X,123)]
abc[pqr(1,2)]
abc[foo->{bar1,bar2,bar3}]
```

In the first, the method `p(foo,bar)` has the name `p`, the arguments `foo` and `bar`, and `v(?Z)` is its result. The second method, `q(?Z,123)`, is Boolean. It does not return any results; it is simply true or false depending on its arguments (which are `?X` and `123`). The second frame formula has only one method, and it is Boolean. The last frame formula has the method `foo` that returns a set as its result. Since `foo` is just a constant, this method is a property. □

Spec Change 3.2: Note: another restriction with respect to the old document: The Q_i's above cannot be frame formulas — only terms. □

Signature formulas are used to say something about collections of basic formulas. Such statements can be made about classes of objects (their types and default property values) or about predicates.

◦ A ***signature formula*** can have one the following forms:

C [| $Spec_1$, ... , $Spec_n$ |]
P (| T_1 , ... , T_n |)

where $n \geq 0$, C, P, T_1, ..., T_n are terms, and each $Spec_i$ has the form

- `=>`M — typing for Boolean methods
- M`=>`V — typing for non-Boolean methods
- $M\{Low..High\}$`=>`V — same as above, but also specifies min (Low) and max ($High$) cardinality constraints on the result returned by the method
- M — a default value for a Boolean method
- M`->`V — specifies the default value for the method

Here M and V are path expressions and Low, $High$ are non-negative or variables integers. $High$ can also be the symbol $*$, which denotes infinity.

Spec Change 3.3: This is what we previously called *inheritable signatures* (in case of M`=>`V) and *inheritable defaults* (in case of M`->`V). Note the significant change in the syntax. □

Spec Change 3.4: Note: using `low..high` instead of `low:high`. This makes ":" less overloaded. □

A signature of the form C[|...|] is a statement about a class, C. In such a statement, an expression of the form M`=>`V informally says that M is a property or a method whose return value has the *type* (i.e., belongs to the class) V. An expression of the form M in signature formulas is used to give the type for Boolean methods in frame formulas. An expression of the form M`->`V states that the *default value* for M in class C is V. For instance,

```
Human[| name{1..*}=>\string,
        married(Year),
        age(Year){1..2}=>\integer,
        offspring{0..*}=>Human,
        num_chromos=>\integer,
        num_chromos->46 |]
```

states that the property `name` for humans has the type `\string`, the Boolean method `married` tells us whether that person was married in a particular year. Well-typed invocations of the method `married` must be provided instances of class `Year` as an argument. The method `age` is a method that tells us person's age in a given year. Well-typed invocations of that method take arguments that are instances of class `Year` and returns an integer. The cardinality constraint says that this method can return one or two values: most people's age can change during the year except for those born on January 1. The property `offspring` returns values of type `Human`; there are no cardinality restrictions

on that method (`0..*` is redundant). The property `num_chromos` has the type `\integer` and its default value of 46.

A formula of the form $P(|...|)$ is a statement about the argument types of the predicate P where the arguments are intended to represent classes of objects. For instance, `Flight(|City,City,\integer|)` states that `Flight` is a ternary predicate whose first two arguments have the type `City` and the last `\integer`.

3.3.3 Compound Formulas

A ***compound formula*** is a formula that combines basic formulas into more complex ones. Rulelog has several different kinds of compound formulas, as described below. Some of them are allowed in the rule heads only and some both in rule heads and bodies.

A ***conjunctive formula*** (or a ***conjunction***, for short) is a formula of the form φ `and` ψ, where both φ and ψ are compound formulas. As a shortcut, Rulelog allows to use comma (",") instead of `and`. Thus, `p(?X) and q(a,b)` and `p(?X),q(a,b)` are considered to be synonymous.

A ***default-negated literal*** is a compound formula of the form `naf`φ, where φ is a basic formula or an explicitly-negated formula.

An ***explicitly-negated literal*** is a compound formula of the form `neg`φ, where φ is a basic formula other than a comparison formula or an evaluable expression.

A ***parenthesized compound formula*** is an expression of the form (φ) where φ is a compound formula. In Rulelog, a parenthesized formula, (φ), is allowed wherever φ is allowed.

A ***foreign formula*** is an expression of the form φ`@`*ModuleRef* where φ is a basic or a parenthesized compound formula, and *ModuleRef* is a *module reference* (to remind: a module reference is an abstract symbol or a variable). Foreign formulas can occur *only* in rule bodies.

3.3.4 Rules, Facts, and Queries

A ***rule*** is a statement of the form

@!{*ruleid*[`tag->`T, *DefeasibilityFlag*, *OtherProperties*]} *head* `:-` *body*.

where *body* is a compound formula, T is a pure HiLog term, *ruleid* is a term, and *DefeasibilityFlag* is either `strict` or `defeasible`. *OtherProperties* is a comma-separated, possibly empty list of items of the form *term* or *term*`->`*term*. The tag property above can appear in any order with respect to the other items inside the brackets or it may be omitted. The head of a rule, *head*, is a basic formula other than:

- a comparison formula
- an evaluable expression

The expression *head* in the above rule is called the ***head*** of the rule and *body* is called the ***body*** of that rule. The `@!{...}` part is called the ***descriptor*** of the rule. Its primary role is to provide a handle by which the rule could be used as part of the defeasible reasoning machinery. Other uses of descriptors include identification and syntax extensions. Rulelog defines a number of components of statement descriptors, which are listed in Section 3.4. The descriptor part is optional but, if omitted, a unique rule Id descriptor is assumed to be assigned b the system—See Section 3.4.

Note that rules are terminated with a period. Facts and queries, defined below, are also terminated with periods.

A ***fact*** is a statement of the form

fact.

where *fact* is a term.

A ***query*** is a statement of the form

?- *query*.

where *query* is a compound formula.

Spec Change 3.5: Note: exclusions will no longer be specified using the `exclusion` annotation. Instead, we'll be using a predicate or a frame. □

Issue 3.6: Need to come up with a predicate/frame/whatever to express exclusions in an argumentation-theory-independent way. □

Editor's Note 3.7: Note: overriding and exclusions are not part of this syntax section. That is because they are not new kinds of syntactic formulas. They are regular predicates syntax-wise, but their semantics is special. So, they'll be addressed in the semantics section. □

3.4 Syntactic Extensions

3.4.1 Path Expressions

A ***path expression*** is a term of the form $e_1.e_2.\cdots.e_n$, where $n \geq 1$ and

- Each e_i, is either a generalized HiLog term (see below), a frame formula (defined in Section 3.3.2), or a parenthesized path expression; and
- e_n is a generalized HiLog term.

For example, the following are path expressions: `a.b.c`, `a.b[p->c,q->d].e`, `a.b[p.r->c,q->d.h].e[abc.cde].f`, `?X(a,?Y)`, `abcde`, `?X(a)(?Y,b)`, `a.b(?X,?Y.f(s))`. In contrast, `a.b.c[d->e]` and `a.b[p->c,q->d]` are *not* path expressions.

Spec Change 3.6: The last condition in the definition of path expressions makes Rulelog path expressions a little more restricted than before (see the above negative examples). This is to avoid unintuitive semantics of formulas such as `p(a.b[c->d])`, which people often confused with `p(${a.b[c->d]})`. Also, no path expressions of the form `p!q!r`. Our experience has been that we never used them (also in FLORA-2), and we are changing the syntax of signatures anyway. □

A ***foreign path expression*** is a term of the form *T@ModuleRef*, where *T* is a path expression and *ModuleRef* is a ***module reference***, which can be an uninterpreted Rulelog symbol (a ***module name***) or a variable.

Editor's Note 3.8: Terms as module names are also useful. However, this is not implemented at present. Terms as module names are also hard to implement efficiently. □

A ***generalized HiLog term*** has one of the following forms:

- A (pure) HiLog term.
- A reification term.
- A path expression.
- An expression of the form $t(t_1, ..., t_n)$, where $n \geq 0$ and t, t_1, ..., t_n are path expressions. When $n = 0$, the term has the form $t()$.

For example: `f(?X,abc)`, `?X(a,?Y)`, `?X(a)(?Y,b)`, `a.b(?X,?Y)`, `a.b(?X,c).d(e(?Z).f,q.p[r->13].u,22)`.

Editor's Note 3.9: Note: *a.b(c.d,e.f)* is a generalized HiLog term. Check if FLORA-2 parses this correctly.
Note: need to specify associativity correctly. Flora parses the above as (a.b)((c.d),(e.f)), as expected. □

Generalized HiLog terms are permitted anywhere where pure HiLog terms are allowed in the Rulelog syntax, except:

- Statement descriptors.
- Places where specialized terms are *explicitly* required (e.g., variables, first-order terms). Example: aggregate variables are explicitly required in comprehension terms, so generalized HiLog terms are not allowed there.
- Wherever otherwise stated explicitly.

3.4.2 User-Defined Functions

User-defined functions (abbr., UDF) are a syntactic extension that permits the users to enjoy certain aspects of functional programming in the framework of a logic-based language.

A ***declaration of a user-defined function*** has one of the following forms:

`udf` $FunctionName(t1,...,tn)$ `:=` $Expr$ `if` $Definition$.
`udf` $FunctionName(t1,...,tn)$ `:=` $Expr$ `:-` $Definition$.
`udf` $FunctionName(t1,...,tn)$ `:=` $Expr$.

The first two forms are equivalent; the last is an abbreviation for the case where $Definition$ is empty (i.e., tautologically true). `Expr` is a term and `Definition` can be any formula that can be used as a rule body. The arguments of the UDF `foo` are terms, which usually are distinct variables, but generally the arguments can be any terms. `Expr` and *Definition* can contain occurrences of other UDFs, but those UDFs must be defined previously.

Example 3.4: The following example defines `father/1` as a function.

```
udf father(?x):=?y if father(?x,?y).
father(mary,tom).
father(john,sam).
```

Now, instead of writing `father(John,?y)` and then using `?y` one can simply write `father(?John)`:

```
?- ?y=father(?x).
```

will return

```
?x=mary ?y=tom
?x=john ?y=sam
```

The query

```
?- writeln(father(mary))@\pplg.
```

will output `tom`. □

Although the UDFs used in the definition of another UDF must be previously defined, it is still possible to create mutually recursive UDFs with a little syntactic workaround.

Example 3.5: The following function declaration is not allowed:

```
udf foo(a) := b.
udf foo(?X) := f(?X) if p(bar(?X)), ?X>0.
udf bar(?X) := foo(?Y) if ?X is ?Y+2.
```

because the definition of `foo` uses the UDF `bar`, which is defined only later. However, the following legal workaround captures the original intent:

```
udf foo(a) := b.
udf foo(?X) := f(?X) if newpred(?X).
udf bar(?X) := foo(?Y) if ?X is ?Y+2.
newpred(?X) :-  p(bar(?X)), ?X>0.
```

□

The following examples illustrate a number of advanced uses of the UDF feature. One of the most interesting such features is a simplification of the use of arithmetics. For instance, normally one would write

```
?- ?x is 1+2, writeln(?x)@\prolog.
```

but with UDFs we can define "+" as a function:

```
udf ?x+?y := ?z if ?z is ?x+?y.
```

and then issue the following query:

```
?- writeln(1+2)@\prolog.
```

to get the same result. The following example uses the above arithmetic functions in an elegant definition of the Fibonacci function:

Example 3.6: Functional definition of Fibbonacci:

```
udf ?x+?y := ?z if ?z is ?x+?y.
udf ?x-?y := ?z if ?z is ?x-?y.
udf fib(0) := 0.
udf fib(1) := 1.
udf fib(?x) := fib(?x-1)+fib(?x-2) if ?x>1.
```

We can now write queries like the following:

```
?- writeln(fib(34))@\prolog.
```

instead of the more cumbersome `?- fib(34,?X), writeln(?X)@\prolog`. This also illustrates how a definition of a UDF can consist of multiple `function`-statements—just like a definition of a predicate can have multiple rules. □

3.4.3 Statement descriptors

Editor's Note 3.10: This section might be moved to non-logical markup. □

Statement descriptors are used to modify the syntax and semantics of the Rulelog expressions in which they appear and to attach metadata to those expressions.

Spec Change 3.7: Note: `@[prefix]` and `@[argumentation]` are gone. These are statements that belong to non-logical markup and to semantic directives. They have global effect and are absolutely out of place among descriptors. □

Spec Change 3.8: Note: many changes to implement the decision about Rulelog annotation shortcuts: `http://silk.bbn.com/index.php/High-level_outline#Annotation_Syntax`. □

The general form of a descriptor is

`@!{`***`ruleid`***`[`*descriptor-1*, ..., *descriptor-n*`]}`

where each *descriptor-i* has one of these forms:

keyword
keyword`->`*value*.

where *keyword* and *value* are terms. The keywords `tag`, `strict`, and `defeasible` are reserved and are described in more details below.

The explicit rule id may be replaced with "\@!". The descriptor may be omitted altogether. In these two cases, a unique abstract symbol is implicitly assigned as the id of the rule. The frame part of the descriptor, [...], can also be omitted (leaving only the rule Id). At most one rule Id descriptor can precede the same Rulelog rule. However, as we shall see shortly, several shortcuts are allowed, and any number of these shortcuts can be specified in addition to or instead of the rule Id descriptor. These multiple rule descriptors must *not* conflict, however. For example, the same rule cannot be specified both as strict and defeasible. Note also that, since *value* above must be a term, frames cannot be nested inside rule descriptors. The aforesaid descriptor shortcuts are defined below.

Example 3.7: Explicit and implicit rule Ids:
Explicit rule id: `@!{123[tag->{foo,bar},defeasible]}` or, equivalently, `@!{123} @foo @bar @@defeasible` using the shortcut notation defined below. Note the use of multiple annotation blocks in the shortcut form (and also multiple tags).
Implicit rule id: `@!{\@![tag->foo,defeasible]}` or, equivalently, `@foo @@defeasible` using the shortcut notation. □

Users can introduce their own descriptors properties:

Example 3.8:

```
@!{r123[my#tag->abc,
        dc#creator->'http://ex.com/Bob',
        another_property->1]}
p(?X) :- q(?X).
```
□

The explicit ids of rules in the same file or document must be distinct. However, different documents and files can, in principle, have rules with the same id. Given the potentially decentralized nature of knowledge base development, it is impractical to assume that ids in different documents will not clash. The *full* rule ids are thus defined as triples of the form

$$(explicit\_ruleid,\ file/document\_url,\ Rulelog\_module)$$

This schema makes it possible to uniquely identify rules both at run time and statically, in a Rulelog document. Static uniqueness is ensured by the fact that ids are unique within each file, and the inclusion of file/document URLs provides the rest. Runtime uniqueness is achieved by including the modules into which the rules are loaded. (Recall that the same Rulelog file/document can be loaded into several different modules. For instance, several agents can be initialized with the same knowledge base.)

An important side effect of specifying descriptors (whether builtin or user-defined) is that certain facts are automatically generated and added to the knowledge base. These facts are accessible to the user knowledge base. The form of these facts are up to each particular implementation of Rulelog, but they must be queriable through the standard Rulelog primitive `@!{...}`, described later.

The following keywords are reserved:

`tag:` Specifies a rule tag or tags to be used by the defeasible reasoning mechanism (see Section 4.1). The tags do not have to be unique and multiple rules may have the same tag. If no tag is given, the value of the `id` descriptor is assumed.

Shortcut: `@`*sometag* or `@{`*sometag*`}` instead of `@!{\@![tag->`*sometag*`]}`. For instance, `@foobar` instead of `@!{\@![tag->foobar]}` and `@tag1 @tag2` instead of `@!{\@![tag->{tag1,tag2}]}`.

`strict:` Indicates that the rule is to be strict, i.e., non-defeasible. Rulelog allows strict rules to have tags, as some implementations might opt to make strictness a runtime property, which could be turned on and off.

Shortcut: `@@strict` or `@@{strict}`.

`defeasible:` Indicates that a rule is to be defeasible. This is a default. `strict` and `defeasible` are mutually exclusive and cannot be both specified.

Shortcut: `@@defeasible` or `@@{defeasible}`.

Note that shortcut and the unabbreviated form of the rule descriptor can be combined in various ways, as shown below.

Example 3.9: Various alternatives:
`@!{myid[abcd->123,tag->{tag1,tag2},strict]}` *rule.*
`@tag1 @!{myid[abcd->123, tag->tag2, strict]}` *rule.*
`@tag1 @tag2 @!{myid[abcd->123, strict]}` *rule.*
`@tag1 @tag2 @@strict @!{myid[abcd->123]}` *rule.*
□

Querying statement descriptors. Statement descriptors can be queried using the `@!{`*`frame`*`}` construct in rule bodies or queries. Here *`frame`* is a frame formula of the form *Obj[prop1,prop2,...]*. It has the same syntax as the frame in the rule Id descriptor and has the same restrictions. In particular, these frames cannot be nested. For instance,

Example 3.10: A descriptor query:
`?- ..., @!{?X[strict,tag->B,foo->1,bar->?Y]}, ...` □

3.4.4 Skolem Constants and Function Symbols

Skolemization is a logical process which introduces a completely new constant of function symbol as a replacement for existential quantifiers. Silk supports skolemization by providing syntax that the user can use to specify a skolem constant or function and the compiler will automatically take care of the uniqueness. Two type of Skolem symbols are supported: named and unnamed.

Unnamed Skolem symbol: `\#`.
Every occurrence of this symbol is replaced by the compiler with a completely new abstract symbol. This symbol can occur both as a constant and as a function symbol.

For example, `\#, \#(a,b), \#(a,\#(b,\#))`

Named Skolem symbol: `\#12`, `\#59`.
These symbols are written as an unnamed Skolem followed by a number. Each numbered Skolem symbol has the scope of the Rulelog rule/fact in which it occurs. Identically named such symbols that occur in the same rule/fact are replaced by the compiler with the *same new* symbol, while symbols that occur in different rules are treated as unrelated.

Example 3.11: Consider the following two rules:
```
\#12(?X,\#12(?X,b), \#34(a,\#34(b,\#,\#))) :- ...
\#12(?X) :- \#12(?X,d), \#34(a,\#34)) :- ...
```

In the first rule, the two occurrences of `\#12` are replaced with the same new abstract symbol. The two occurrences of `\#34` are also replaced with a new abstract symbol, but with one that differs from the symbol used for `\#12`. The two occurrences of the symbol `\#` are replaced with two different new symbols that are unrelated to each other and to the symbols used for `\#12` and `\#34`.
The two occurrences of the symbol `\#12` in the second rule are also replaced with the same new symbol, but this symbol is different from the one used for `\#12` in the first rule. Ditto for `\#34` in the second rule. □

Observe that in this example all Skolem symbols occur only in the rule heads. This is not accidental': such symbols are allowed only in rule heads. Using them in rule bodies is considered an error because, due to the uniqueness of these symbols, a body-occurrence of such a symbol cannot be satisfied by matching against of another Rulelog fact or rule.

3.4.5 Various Other

TBD

- Omni.
- Composite frame formulas (combinations of frames and subclass/membership)
- introduce notation like {obj1,obj2,obj3}`:`class likewise for {cl1,cl2,cl3}`::`cl.
- introduce class expressions like (class1 `and` class2), (class1 `or` class2). This will allow writing obj`:`(class1 `and` class2) (equivalently obj`:`(class1,class2)), i.e., it belongs to both.

3.5 Non-logical Markup

Non-logical markup consists of Rulelog statements that do not affect the semantics. The purpose of these statements is to make the use of Rulelog more convenient. Statements that belong to non-logical markup include macros, comments, import statements, and the like.

3.5.1 IRI Prefix and Base Declarations

A ***prefix declaration*** is a statement of the form

`:- prefix` *`prefix-name`* `= <`*`URI`*`>.`

Such a statement defines *prefix names* that can be later used in CURIE macros. For instance,

```
:- prefix w3 = <http://www.w3.org/TR/>.
```

defines a prefix, `w3`. As a result of this declaration, a macro such as `w3#xquery` expands into the IRI `'http://www.w3.org/TR/xquery'`. Note that the concatenated string is enclosed in single quotes. If the expansion of the prefix contains single quotes then those quotes must be escaped with a backslash.

A ***base declaration*** declares a prefix to be used with CURIE macros that have no explicitly given prefix. It has the form:

```
:-  base <URI>.
```

At most one base directive per file or document is allowed.

Issue 3.7: Or should we allow later base declarations to replace earlier ones? What about multiple declarations of the same prefix? □

A CURIE macro that has no explicit prefix looks like this: `#`*localname*, where *localname* is the local name of a CURIE. For instance, if a Rulelog document has this declaration:

```
:- prefix <http://www.w3.org/TR/>.
```

then `#xquery` will expand into `'http://www.w3.org/TR/xquery'`.

3.5.2 Comments

Rulelog has two kinds of comments: single line comments and multiline comments. The syntax is the same as in Java. A ***single-line comment*** is any text that starts with a `//` and continues to the end of the current line. If `//` appears within a string (`"..."`) or a quoted symbol (`'...'`) then these characters are considered to be part of the string or the symbol, and in this case they do not start a comment. A ***multiline comment*** begins with `/*` and ends with a matching `*/`. The combination `/*` does not start a comment if it appears inside a string or a quoted symbol.

Remark 3.3: Note that `/*...*/` may not be nested. That is, in `/*.../*...*/...*/` the comment starts with the first `/*` and ends with the first (inner) `*/`. The second `*/` will likely cause a syntax error (unless it happens to be inside a string or a symbol).

3.5.3 Import and Export of Modules

An ***import statement*** has the form

```
:- import   module1, ..., modulen.
```

where $module_i$ are Rulelog module names. Informally, this means that external formulas that match the modules' exported templates can be *invoked* (i.e., appear in rule bodies) in the module where the import statement appears.

A *formula template* is a pure HiLog, frame, membership, or subclass formula of the following form:

- *HiLog*: $t(?, ..., ?)$ (zero or more arguments), where t is either the anonymous variable ? or a term.
- *Frame*: ?[m`->`?], where m is ? or a term.
- *Membership*: ?`:`c, where c is ? or a term.
- *Subclass*: ?`::`c, where c is ? or a term.

An export ***statement*** has the form

`:- export` $Template_1$ [`>>` ($module_{11}$, ..., $module_{1k_1}$)], ...,
$Template_n$ [`>>` ($module_{n1}$, ..., $module_{nk_n}$)].

where $Template_i$ are formula templates and $moduleij$ are module names and parentheses around the module lists are needed only if more than one module is listed. The parts inside the brackets are optional. If the `>>`-clause is not present, the corresponding template is exported to all modules. If the `>>`-clause is present, the template is exported only to the listed modules.

If a module has no export statement, all HiLog, frame, membership, and subclass formulas are exported. If a module *has* an export statement then only the listed templates are exported. A runtime attempt by a rule in module M to call an atomic formula that does not unify with a template exported to M will result in an error and the current query will be terminated.

Remark 3.4: Multiple import and export statements are allowed in the same module.

Example 3.12: Here are some examples of the import and export statements in Rulelog:

```
:-import foobar.
:-export ?(?)>>(me,you), p(?,?), ?[name->?], ?::person>>him.
```

The first statement makes all exported formulas in module `foobar` available in the module where the above `import` statement occurs. The second statement exports various formulas from the current module as follows:

- `export ?(?)>>(me,you)`: any unary HiLog predicate to modules `me` and `you` (and none other).
- `p(?,?)`: binary predicate `p` to any module.
- `?[name->?]`: the method `name` (in any class) to any module.
- `?::person>>him`: the ability to query which classes are subclasses of `person` to module `him`.

□

3.5.4 Preprocessor

Rulelog has a C-like preprocessor, which can be used to make Rulelog knowledge bases more readable and flexible.

TBD

Here we intend to include something like the C language macros, i.e., #define, #include. For this we might be able to reuse gpp, the preprocessor used by XSB and Flora.

3.6 Creation of and Loading into Rulelog Modules

Rulelog modules can be created

- by loading a file or a document into a module.
- by a `newmodule` command.

On start-up, the module `main` is automatically created and all rules and facts inserted directly through the Rulelog shell are automatically in that module. These rules can be overwritten with an explicit `load` command or added to using an `add` command:

```
?- load{FileOrURL)}.
?- add{FileOrURL)}.
```

The `load` command deletes all the rules and facts that exist in module `main` and instead inserts the rules and facts from the file or document specified by

FileOrURL into that module. The `add` command works similarly, but does not erase the old contents of `main`.

The above commands can also be used to create (or refresh) other modules as follows:

```
?- load{FileOrURL>>ModuleName)}.
?- add{FileOrURL>>ModuleName)}.
```

The first command works as follows. In the module *ModuleName* does not exist, it is created empty and then the rules and facts found in *FileOrURL* are inserted in the new module. If the module does exist then the old contents is erased and the new contents is taken to be that of *FileOrURL*. The `add` command works similarly except that the old contents is not erased: *ModuleName* is created empty, if necessary, and then the contents of *FileOrURL* is added to what is currently in the module.

Modules can be deleted with the help of the `erasemodule` command:

```
?- erasemodule{Module_1, ..., Module_n}.
```

An empty module can be also created using the `newmodule` command:

```
?- newmodule{Module_1, ..., Module_n}.
```

Rulelog also supports shortcuts for loading and addition to modules: instead of `load` one can type `[...]` and instead of `add(...)` one can use `[+...]`. These commands are illustrated below.

Example 3.13: Loading, addition, deletion, and creation of modules.

```
?- load{foo}, add{'http://example.com/test'}.
?- add{bar>>moo}, load{'http://example.com/test'>>testmod}.
?- [foo].  // same as load
?- [+bar>>moo], [+bar2>>moo].  // same as add to moo
?- ['http://example.com/test'>>test].  // same as load into test
?- erasemodule{moo,test}.
?- newmodule{moo,testmod}.
```

□

it should be kept in mind that the code that is loaded into a module is compiled into very efficient virtual machine code and therefore is *much faster* than the code that is added to a module. This difference in performance is especially pronounced for rules (even small ones), but does not exist for facts. Therefore, users should structure their knowledge bases by first loading most of the rules into the relevant modules and keep subsequent rule addition in check. Adding facts does not cause significant overhead, however.

3.7 The Lloyd-Topor Syntactic Extensions

The Lloyd-Topor syntactic extensions [Llo87] make it possible to include first-order-looking formulas in the rule bodies subject to certain restrictions. They also slightly generalize the heads of the rules. We distinguish between the relatively simple *monotonic* Lloyd-Topor extensions and the more complex *non-monotonic* ones.

3.7.1 Monotonic Lloyd-Topor Extensions

These extensions introduce the following features:

1. Disjunction in rule bodies.
2. Conjunction in rule heads.
3. Additional forms of implication.

A ***Lloyd-Topor implication*** is a statement of one of the following forms:

formula$_1$ `~~>` *formula*$_2$
formula$_2$ `<~~` *formula*$_1$

where the *formula*$_2$ is a conjunction of basic formulas and *formula*$_1$ can be a Boolean combination of basic formulas that contains conjunctions and disjunctions only.

Intuitively, the implication f_1 `~~>` f_2 is interpreted as "***naf*** f_1 `or` f_2" while f_2 `<~~` f_1 is treated as "f_2 `or` ***naf*** f_1".

If Lloyd-Topor implication holds in both directions between formulas, their logical equivalence can be expressed in a single statement using ***Lloyd-Topor bi-implication*** of the following form:

formula$_1$ `<~~>` *formula*$_2$

In that case, both `formula`$_1$ and `formula`$_2$ must be conjunctions of basic formulas.

Horn rules are extended as follows. A rule still has the form

head `:-` *body*.

but *head* can now be a conjunction of basic formulas[1] and of Lloyd-Topor implications (including bi-implications) and *body* can have basic formulas combined using the `and` and `or` connectives.

Editor's Note 3.11: Add the implications of the form `==>`, `<==`, `<==>`. □

Monotonic Lloyd-Topor extensions are viewed as shortcuts that reduce to the base Rulelog via the following ***monotonic Lloyd-Topor transformations***:

- *head* `:-` *body*$_1$ `or` *body*$_2$ reduces to

  head `:-` *body*$_1$.
  head `:-` *body*$_2$.

- *head*$_1$ `and` *head*$_2$ `:-` *body* reduces to

  head$_1$ `:-` *body*.
  head$_2$ `:-` *body*.

[1] Subject to the restrictions of Section 3.3.4. Namely: these basic formulas cannot be comparison formulas or evaluable expressions.

- (*head*$_1$ `<~~` *head*$_2$) `:-` *body* reduces to

  head$_1$ `:-` *head*$_2$ `and` *body*.

- (*head*$_1$ `~~>` *head*$_2$) `:-` *body* reduces to

  head$_2$ `:-` *head*$_1$ `and` *body*.

- (*head*$_1$ `<~~>` *head*$_2$) `:-` *body* reduces to

  head$_2$ `:-` *head*$_1$ `and` *body*.
  head$_1$ `:-` *head*$_2$ `and` *body*.

Complex formulas in the head are broken down using the last four reductions. Rule bodies that contain both disjunctions and conjunctions are first converted into the disjunctive normal form and then are broken down using the first reduction.

3.7.2 Nonmonotonic Lloyd-Topor Extensions

Nonmonotonic Lloyd-Topor extensions include explicit bounded quantifiers (both `exist` and `forall`) and also permit Lloyd-Topor implications `<~~`, `~~>`, and `<~~>` in rule bodies.[2] These extensions essentially permit arbitrary first-order-looking formulas in the body of Rulelog, although, as we shall see, with certain restrictions. We emphasize that the semantics of Rulelog is *not* first-order and this is why we call these formulas "first-order looking" and not first-order. For example, Lloyd-Topor implication `A <~~ B` is interpreted in a non-classical way: as `(A ornaf B)` (either `A` is known or `B` is not known to be true). This is quite different from classical implication, which is interpreted as `(A orneg B)` (either `A` is known or `B` is known to be false).

Recall that without explicit quantification, all variables in a rule are considered *implicitly* quantified with `forall` outside of the rule, i.e., `forall ?X,?Y,...` (*head* `:-` *body*). A variable that occurs in the body of a rule but not in its head can be equivalently considered as being existentially quantified in the body. For instance,

```
forall(?X,?Y)^( p(?X) :- q(?X,?Y) ).
```

is equivalent to

```
forall(?X)^( p(?X) :- exist(?Y)^(q(?X,?Y)) ).
```

In the scope of the `naf` operator, unbound variables have a different interpretation under negation as failure. For instance, if `?X` is ground and `?Y` is not ground as, for instance, in

```
p(?X) :- naf q(?X,?Y), r(?Y).
```

[2] Recall that *monotonic* Lloyd-Topor extensions permit these implications only in rule heads.

then the evaluation of `naf q(?X,?Y)` is delayed until `?Y` is bound by other subgoals (such as `r(?Y)`). If `?Y` cannot be ground (e.g., if `r(?Y)` does not ground `?Y`) then a runtime error is issued. The user can also explicitly request that instead of an error the variable `?Y` should be treated as existential, if it is not grounded by subsequent subgoals:

```
forall(?X)^( p(?X) :- wish(ground(?Y))^(naf q(?X,?Y)) ).
```

Here `wish(...)` is a *delay quantifier*, which will be explained elsewhere.

Formally, Nonmonotonic Lloyd-Topor extensions include the following kinds of rules. The rule ***heads*** are the same as in the monotonic Lloyd-Topor extension. The rule ***bodies*** are defined as follows.

- Any basic formula is a legal rule body
- If f and g are legal rule bodies then so are
  - f `and` g
  - f `or` g
  - `naf` f
  - f `~~>` g
  - f `<~~` g
  - f `<~~>` g
- If f is a legal rule body then so is
  - `exist(?X`$_1$`,...,?X`$_n$`)^`f

  where `?X`$_1$, ..., `?X`$_n$ are variables that occur *positively* (defined below) in f.
- If g_1, g_2 are legal rule bodies then
  - `forall(?X`$_1$`,...,?X`$_n$`)^(`g_1`~~>` g_2`)`
  - `forall(?X`$_1$`,...,?X`$_n$`)^(`g_2`<~~` g_1`)`

  are legal rule bodies provided that `?X`$_1$, ..., `?X`$_n$ occur *positively* in g_1

The above quantifiers bind stronger than the other connectives such as `and`, `or`, `~~>`, etc., so parentheses should be used when ambiguity might arise. For instance, `forall(?X)^p(?X),q(?X)` is interpreted as `forall(?X)^(p(?X)),q(?X)` rather than `forall(?X)^(p(?X),q(?X))`.

A ***positive occurrence*** of a variable in a formula is defined as follows:

- All variables in a basic formula occur there positively
- A free variable occurs positively in f `and` g iff it occurs positively in either f or g.

- A free variable occurs positively in f `or` g iff it occurs positively in both f and g.
- A free variable occurs positively in f `~~>` g iff it occurs positively in g.
- A free variable occurs positively in f `<~~` g iff it occurs positively in f.
- A free variable occurs positively in f `<~~>` g iff it occurs positively in both f and g.
- A free variable occurs positively in `exist(?X`$_1$`,...,?X`$_n$`)^`f or `forall(?X`$_1$`,...,?X`$_n$`)^`f iff it occurs positively in f.

Editor's Note 3.12: Need rationale here. □

Issue 3.8: Need to add restrictions to the above syntax. For instance, universal quantifiers over builtins should be disallowed because one cannot generally compute things like `naf exists somebuiltin` or `forall(...)^somebuiltin`.
Similarly, universal quantification of subgoals that are in the scope of the delay quantifiers `wish` and `must` cannot be allowed for the same reason. □

Non-monotonic Lloyd-Topor Transformations

These transformations aim to eliminate the extended Lloyd-Topor syntactic forms in rule bodies and reduce these rules to the base Rulelog syntax. The intuition behind most of these transformations is that `(`f `~~>` g`)` is treated as a shorthand for `(naf` f`) or`g and `forall(?X)^f` as a shorthand for `naf exist(?X)^(naf f)`.

Note that the rules, below, must be applied top-down, that is, to the conjuncts that appear directly in the rule body. For instance, if the rule body is as follows

`forall(?X)^exist(?Y)^(foo(?Y,?Y)~~>bar(?X,?Z)) <~~ foobar(?Z)`

then one should first apply the rule for `<~~`, then the rules for `forall`, and finally the rules for `exist` and `~~>`.

The non-monotonic Lloyd-Topor transformations are listed next.

- Let the rule have the form

  $head$ `:-` $body_1$ `and (`f `~~>` g`) and` $body_2$`.`

  Then the Lloyd-Topor transformation replaces it with the following pair of rules:

  $head$ `:-` $body_1$ `and naf` f `and` $body_2$`.`
  $head$ `:-` $body_1$ `and` g `and` $body_2$`.`

The transformations for <∼∼ and <∼∼> are similar.

- Let the rule be

  head `:-` *body*$_1$ `and forall(?`X_1`,...,?`X_n`)^(`g_1 `∼∼>` g_2`)` `and` *body*$_2$`.`

  where $?X_1, \ldots, ?X_n$ are free variables that occur positively in g_1.

  The Lloyd-Topor transformation replaces this rule with the following pair of rules, where $q(?X'_1, \ldots, ?X'_n)$ is a new n-ary predicate and $?X'_1, \ldots, ?X'_n$ are new variables:

  head `:-` *body*$_1$ `and naf q(?`X'_1`,...,?`X'_n`)` `and` *body*$_2$`.`
  `q(?`X_1`,...,?`X_n`)` `:-` g_1 `and naf` g_2`.`

  The transformation for <∼∼ is similar. Bi-implications, such as g_1<∼∼>g_2, are first broken into conjunctions of unidirectional implications, e.g., g_1<∼∼g_2 and g_1<∼∼>g_2, and the the transformation rules for the latter are applied.

- Let the rule be

  head `:-` *body*$_1$ `and exist(?`X_1`,...,?`X_n`)^(`f`)` `and` *body*$_2$`.`

  where $?X_1, \ldots, ?X_n$ are free variables that occur positively in f.

  The Lloyd-Topor transformation replaces this rule with the following:

  head `:-` *body*$_1$ `and` f' `and` *body*$_2$`.`

  where f' is f in which the variables $?X_1, \ldots, ?X_n$ are consistently renamed into new variables. Since the variables in f' are new and do not appear elsewhere in the rule, the explicit existential quantification can be dropped in favor of implicit quantification.

4 The Semantics of Rulelog

Like the syntax of Rulelog, the description of its semantics is organized into several parts. The first part consists of a formal model theory for the base sublanguage of Rulelog (i.e., the sublanguage formed by Rulelog's base syntax); it is presented in Section 4.1. The second part presents a semantics for the syntactic extensions of Rulelog's language. This does not require a separate model theory. Instead, syntactic extensions are translated directly into the base syntax, as described in Section 4.2. The third part consists of background axioms that are not covered by the model theory. These axioms are given in Section 4.4. In addition, the semantics of Rulelog can be controlled by various user-level directives, such as the directives that prescribe the argumentation theory to be used. These directives are described in Section 4.5.

4.1 A Model Theory for the Base Syntax

We now turn to a rigorous model-theoretic description of the semantics of the base sublanguage of Rulelog. First we define semantic structures and then the notion of entailment.

4.1.1 Semantic Structures

The ***extended Herbrand universe*** $\mathbf{U}_H$ consists of all ***ground*** (i.e., variable-free) Rulelog terms of the following kinds:

- ground *pure* HiLog terms (Section 3.3.1). This subset of $\mathbf{U}_H$ will be denoted by $\mathbf{U}_H^{pure}$.
- ground reification terms. This subset of $\mathbf{U}_H$ will be denoted by $\mathbf{U}_H^{reif}$.

Since here we will be dealing only with extended Herbrand universes, we will usually omit the adjective "extended."

Rulelog's semantics is based on a variant of the well-founded semantics for logic programs [VRS91], which is a form of three-valued logic that relies on partial interpretations. In describing the model theory of Rulelog we follow the outline of [Prz94].

A Rulelog ***semantic structure***, $\mathcal{I}$, is a tuple of the form $\langle \mathbf{D}_\mathcal{I}, \mathcal{I}_C, \mathcal{I}_V, \mathcal{I}_{true} \rangle$, where

- $\mathcal{I}_{true}$ is a mapping from the set of all ground formulas to the set of truth values $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$, which satisfies the following **congruence** property with respect to the equality as well as additional properties spelled out later in this section.
  - $\mathcal{I}_{true}(s\texttt{:=:}s) = \mathbf{t}$, for every $s \in \mathbf{U}_H^{pure}$.
  - If $\mathcal{I}_{true}(s\texttt{:=:}t) = \mathbf{t}$, $\mathcal{I}_{true}(S) = \mathbf{t}$ (respectively, $\mathcal{I}_{true}(S) = \mathbf{f}$), and $S[t/s]$ denotes S in which some occurrence of the term s is replaced with t, then $\mathcal{I}_{true}(S[t/s]) = \mathbf{t}$ (respectively, $\mathcal{I}_{true}(S[t/s]) = \mathbf{f}$). In short, $\mathcal{I}_{true}$ is closed with respect to substitution of *equals by equals*.
- $\mathbf{D}_\mathcal{I}$ is the ***domain*** of $\mathcal{I}$.
  It is defined as a union $\mathbf{D}_\mathcal{I}^A \cup (\bigcup_\alpha \mathbf{V}_\alpha)$ where $\mathbf{D}_\mathcal{I}^A$ is the *abstract domain* (defined next) and the $\mathbf{V}_\alpha$'s are the value spaces for each supported datatype α in Rulelog (see Section 3.2).

  The ***abstract domain*** of interpretation, $\mathbf{D}_\mathcal{I}^A$, is a *factor* $\mathbf{U}_H/E_\mathcal{I}$ of the *extended Herbrand Universe* $\mathbf{U}_H$ by the equivalence relation $E_\mathcal{I}$, where
  - A ***factor*** of a set, $\mathbf{U}_H$, with respect to an equivalence relation, $E_\mathcal{I}$, is the set of all equivalence classes of the elements of $\mathbf{U}_H$ with respect to $E_\mathcal{I}$.

– The equivalence relation $E_{\mathcal{I}}$ in question is the minimal (by inclusion) relation over $\mathbf{U}_H$ that contains

$$\{(s,t) \mid s,t \in \mathbf{U}_H \ \ \texttt{and}\ \mathcal{I}_{true}(s\texttt{:=:}t) = \mathbf{t}\}$$

Observe that the congruence property of $\mathcal{I}_{true}$ implies that $E_{\mathcal{I}}$ is closed with respect to substitution of equals by equals. Indeed, by the congruence of $\mathcal{I}_{true}$, $\mathcal{I}_{true}(s'\texttt{:=:}s') = \mathbf{t}$. Thus, if $(s,t) \in E_{\mathcal{I}}$ then it follows that $\mathcal{I}_{true}(s'\ \texttt{:=:}\ s'[t/s]) = \mathbf{t}$, where $s'[t/s]$ denotes s' in which an occurrence of s is replaced with t. Therefore, $(s', s'[t/s]) \in E_{\mathcal{I}}$.

Issue 4.1: The above implies that members of the abstract domain cannot be equated to datatype elements. That is, `123:=:abc` is always false. Need to decide if this is desirable. □

- $\mathcal{I}_C$ maps the set of constants $\mathcal{C}onsts$ into $\mathbf{D}_{\mathcal{I}}$ such that
  - For each Rulelog datatype α with the lexical space $\mathcal{C}onsts_\alpha$, $\mathcal{I}_C$ restricted to $\mathcal{C}onsts_\alpha$ coincides with the lexical-to-value-space mapping of α (see Section 3.2).
  - When restricted to the rest of the constants, $\mathcal{I}_C$ maps these constants to their equivalence classes: $\mathcal{I}_C(c) = [c]_{E_{\mathcal{I}}} \in \mathbf{D}^A_{\mathcal{I}}$ ($[c]_{E_{\mathcal{I}}}$ denotes the equivalence class of c with respect to $E_{\mathcal{I}}$).
- $\mathcal{I}_V$ maps $\mathcal{V}ars$ to $\mathbf{D}_{\mathcal{I}}$.

  For convenience, we extend $\mathcal{I}_C$ `and` $\mathcal{I}_V$ to a ***term-interpreting mapping*** that maps terms to $\mathbf{D}_{\mathcal{I}}$. We will denote this mapping with $\mathcal{I}$—the same symbol as the one used for semantic structures. Whether the mapping or the structure itself is meant will be clear from the context.
  - *Constants*: $\mathcal{I}(c) = \mathcal{I}_C(c)$, if $c \in \mathcal{C}onsts$.
  - *Variables*: $\mathcal{I}(?v) = \mathcal{I}_V(?v)$, if $c \in \mathcal{V}ars$.
  - *Pure HiLog terms*: $\mathcal{I}(t(t_1,...,t_n)) = [\mathcal{I}(t)^!(\mathcal{I}(t_1)^!,...,\mathcal{I}(t_n)^!)]_{E_{\mathcal{I}}}$. Here, given $elt \in \mathbf{D}_{\mathcal{I}}$, $elt^!$ stands for an arbitrary member of the equivalence class elt. Due to the congruence of $E_{\mathcal{I}}$, the choice of the actual member in an equivalence class in this definition is immaterial.

    For example, if $\mathcal{I}_V(?x) = a$ `and` $\mathcal{I}_V(?y) = b$ then $\mathcal{I}(d(?x,1)(abc,?y,3.2)) = [d(a,1)(abc,b,3.2)]_{E_{\mathcal{I}}}$.
  - *Reified terms*: $\mathcal{I}(\$\{\varphi\}) = \$\{\varphi'\}$, where φ' is obtained from φ by replacing every constant, c, with its value $\mathcal{I}(c) \in \mathbf{D}_{\mathcal{I}}$ and every variable, $?v$, with its value $\mathcal{I}(?v) \in \mathbf{D}_{\mathcal{I}}$.

    For example, if $\mathcal{I}_C(p) = p'$, $\mathcal{I}_V(?q) = q'$, $\mathcal{I}_C(a) = a'$, $\mathcal{I}_V(?X) = b'$, and $\mathcal{I}_V(?Y) = c'$, then $\mathcal{I}(\$\{p(a,?X)\ \texttt{and}\ ?q(?Y,1)\}) = \$\{p'(a',b')\ \texttt{and}\ q'(c',1)\}$.

Issue 4.2: May need to do something about evaluable terms, like path expressions and aggregation, inside reified formulas. □

- *Properties of* $\mathcal{I}_{true}$. Recall that $\mathcal{I}_{true}$ maps ground formulas to their truth values, the set $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. This set is assumed to have a total order as follows: $\mathbf{f} < \mathbf{u} < \mathbf{t}$. It is this total order that is meant when we talk about the `min` and `max` functions below.

  Besides congruence, $\mathcal{I}_{true}$ has the properties specified below, where all terms are ground.

  - *Term formulas*:
    - $\mathcal{I}_{true}(t)$ is *not defined* if t is *not* a reification term. A Rulelog system is required to issue a runtime error if it is asked to determine the truth value of a non-reified term.
    - $\mathcal{I}_{true}(\$\{t\}) = \mathcal{I}_{true}(t)$, for reification terms.
  - *Equality*: $\mathcal{I}_{true}(s\texttt{:=:}t) = \mathbf{f}$ unless s and t are identical or t and s are *both* pure HiLog terms.

    This implies that $\mathcal{I}_{true}(s\texttt{:=:}t)$ is false if $s \neq t$ and s or t is not a pure HiLog term (e.g., a reification, evaluable, aggregate list, or parenthesized term). Of course, $\mathcal{I}_{true}(s\texttt{:=:}t)$ can still be false even if both t and s are pure HiLog terms.

    Remark 4.1: This implies that if t is not a pure HiLog term then its equivalence class with respect to $E_{\mathcal{I}}$ is a singleton set that contains t only.
  - *Disequality*: $\mathcal{I}_{true}(t\texttt{!=!}s) = \sim \mathcal{I}_{true}(t\texttt{:=:}s)$, where $\sim \mathbf{t} = \mathbf{f}$, $\sim \mathbf{f} = \mathbf{t}$, and $\sim \mathbf{u} = \mathbf{u}$.
  - *Unification*: $\mathcal{I}_{true}(t\texttt{=}s) = \mathbf{t}$ iff t and s are identical as ground terms. $\mathcal{I}_{true}(t\texttt{=}s) = \mathbf{f}$ otherwise.
  - *Disunification*: $\mathcal{I}_{true}(t\texttt{!=}s) = \sim \mathcal{I}_{true}(t\texttt{=}s)$.
  - *Comparison*: $\mathcal{I}_{true}(t\ op\ s) = \mathbf{t}$, where *op* is a comparison operator defined on the value space of s datatype if t and s are members of that value space and $t\ op\ s$ is true.
    $\mathcal{I}_{true}(t\ op\ s) = \mathbf{f}$ if t and s are members of the value space on which *op* is defined and $t\ op\ s$ is false.
    Otherwise, $\mathcal{I}_{true}(t\ op\ s)$ is not defined and if the system is asked to determine the truth value of $t\ op\ s$ it should issue a runtime error and abort evaluation of the current query.
  - *Evaluable expression*: $\mathcal{I}_{true}(t\ \texttt{is}\ Expr) = \mathbf{t}$ if t is a member of a value space of a datatype, *Expr* is an expression over the value space of that datatype and $t = value(Expr)$, where $value(Expr)$ is the result of evaluation of *Expr* on the value space in question.
    If $t \neq value(Expr)$ then $\mathcal{I}_{true}(t\ \texttt{is}\ Expr) = \mathbf{f}$.

Otherwise, if $value(Expr)$ does not exist (which can happen if one of the arguments in $Expr$ does not belong to the appropriate value space) then $\mathcal{I}_{true}(t$ `is` $Expr)$ is not defined and the system should issue a runtime error, if it is asked to evaluate $Expr$.

- *HiLog formula*: Nothing special to be said about HiLog formulas relative to $\mathcal{I}_{true}$, since $\mathcal{I}_{true}(\varphi)$ is already defined for all ground HiLog formulas.
- *Foreign*: $\mathcal{I}_{true}(\varphi$`@`$M)$, where M is a module name, is defined according to the structure of the formula φ as defined in Section 3.3.3.
  - φ *is a basic formula*: In this case, $\mathcal{I}_{true}(\varphi$`@`$M)$ is already defined, by definition of $\mathcal{I}_{true}$.
  - $\varphi = ($`neg` $\psi)$ *is a parenthesized explicitly negated formula*: This case is similar to the previous one, as $\mathcal{I}_{true}(\varphi$`@`$M)$ would already be defined by definition.
  - $\varphi = (\psi_1$ `and` $\psi_2)$ *is a parenthesized conjunctive formula*: $\mathcal{I}_{true}(\varphi$`@`$M) =$ `min`$(\mathcal{I}_{true}(\psi_1$`@`$M)$,$\mathcal{I}_{true}(\psi_2$`@`$M))$
  - $\varphi = ($`naf` $\psi)$ *is a parenthesized default negation formula*: $\mathcal{I}_{true}(\varphi$`@`$M) = \sim \mathcal{I}_{true}(\psi$`@`$M)$.
  - $\varphi = (\psi$`@`$N)$ *is a parenthesized foreign formula*: $\mathcal{I}_{true}(\varphi$`@`$M) = \mathcal{I}_{true}(\psi$`@`$N)$.
- *Frame formula*: Let $o[spec_1, .., spec_n]$ be a frame formula. Then $\mathcal{I}_{true}(o[spec_1, ..., spec_n])$ must satisfy the following properties:
  - $\mathcal{I}_{true}(o[spec_1, ..., spec_n]) =$ `min`$\{\mathcal{I}_{true}(o[spec_1]), ..., \mathcal{I}_{true}(o[spec_n])\}$, where $n > 0$, o is a term, and ach $spec_i$ has one of the three forms spelled out in the definition of frame formulas on page 13.
  - $\mathcal{I}_{true}(o[m$`->`$\{v_1, ..., v_k\}]) =$ `min`$\{\mathcal{I}_{true}(o[m$`->`$v_1]), ..., \mathcal{I}_{true}(o[m$`->`$v_k])\}$, if $k > 0$. Again, the notation here is the same as on page 13 in the definition of frame formulas.
  - $\mathcal{I}_{true}(o[m$`->`$p[spec_1, ..., spec_k]]) =$ `min`$(\mathcal{I}_{true}(o[m$`->`$p], p[spec_1, ..., spec_k]))$, if $k > 0$.
- *Compound formula*:
  - *Parentheses*: $\mathcal{I}_{true}(\varphi) = \mathcal{I}_{true}(\varphi)$
  - *Conjunction*: $\mathcal{I}_{true}(\varphi$ `and` $\psi) =$ `min`$(\mathcal{I}_{true}(\varphi)$,$\mathcal{I}_{true}(\psi))$.
  - *Default negation*: $\mathcal{I}_{true}($`naf`$\varphi) = \sim\mathcal{I}_{true}(\varphi)$.
  - *Explicit negation*: There is no a priori relationship between $\mathcal{I}_{true}($`neg` $\varphi)$ and $\mathcal{I}_{true}(\varphi)$. In principle both can be true, which means that Rulelog admits paraconsistency.
  - *Strict rule*:
    $\mathcal{I}_{true}($`@[tag->`R`, strict]` $head$ `:-` $body) =$ `t`, if
    $\mathcal{I}_{true}(head) \geq \mathcal{I}_{true}(body)$;
    $\mathcal{I}_{true}($`@[tag->`R`, strict]` $head$ `:-` $body) =$ `f` otherwise.

* *Defeasible rule*:
  $\mathcal{I}_{true}(\texttt{@[tag->}R\texttt{, defeasible]}\ head\ \texttt{:-}\ body) = \mathbf{t}$
  if $\mathcal{I}_{true}(head) \geq \texttt{min}(\mathcal{I}_{true}(body), \texttt{naf}\ \texttt{\textbackslash prolog}(r, head, m))$, where m is the module of the rule;
  $\mathcal{I}_{true}(\texttt{@[tag->}R\texttt{, defeasible]}\ head\ \texttt{:-}\ body) = \mathbf{f}$
  otherwise.
  Here \prolog is a reserved predicate, which is used to determine the defeasibility properties of rules. It will be further discussed in Section 4.3.

4.1.2 Models

If φ is a ground formula, $\mathcal{M}$ a semantic structure, and $\mathcal{M}_{true}(\varphi) = \mathbf{t}$, then we write $\mathcal{M} \models \varphi$ and say that $\mathcal{M}$ is a ***model*** of φ or that φ is **satisfied** in (or by) $\mathcal{M}$. An interpretation $\mathcal{M}$ is a model of a Rulelog rule set $\mathbf{P}$ if and only if all the rules in $\mathbf{P}$ are satisfied in $\mathcal{M}$, i.e., if $\mathcal{M} \models R$ for every $R \in \mathbf{P}$.

If $\mathbf{A}$ is an argumentation theory (such as the ones in Section 4.3) then $\mathcal{M}$ is a ***model*** of $\mathbf{P}$ ***with respect to the argumentation theory*** $\mathbf{A}$ if and only if $\mathcal{M}$ is a model of $\mathbf{P}$ and of $\mathbf{A}$. In this case we write $\mathcal{M} \models (\mathbf{P}, \mathbf{A})$.

We now define a ***truth-order*** on Herbrand semantic structures. Let $\mathcal{M}^1$ and $\mathcal{M}^2$ be two such structures. We write $\mathcal{M}^1 \preceq_t \mathcal{M}^2$ if the following conditions hold for every ground term t in $\mathbf{U}_H$:

- $\mathcal{M}^1_{true}(t) = \mathbf{t}$ implies $\mathcal{M}^2_{true}(t) = \mathbf{t}$
- $\mathcal{M}^2_{true}(t) = \mathbf{f}$ implies $\mathcal{M}^1_{true}(t) = \mathbf{f}$

A model of $(\mathbf{P}, \mathbf{A})$ that is minimal with respect to $\preceq_t$ is called a ***least model***. In general, there can be several least models for $\mathbf{P}$ with respect to an argumentation theory.

It is known that rule sets that do not have default-negated literals (i.e., those negated with `naf`) have a *unique* least partial model [Prz94]. If $\mathbf{P}$ is such a knowledge base, then its unique least model is denoted by $LPM(\mathbf{P})$.

4.1.3 Well-founded Models

We now define *well-founded models* for Rulelog knowledge bases. There are several equivalent ways to define well-founded models. Here we adopt the definition from [WGK+09], which follows the general outline of [Prz94]. First, we define the notion of a *quotient*.

Let $\mathbf{P}$ be a set of rules, which can include defeasible as well as strict rules, and let $\mathcal{I}$ be a Rulelog semantic structure for $\mathbf{P}$. We define the ***Rulelog quotient of*** $\mathbf{P}$ ***by*** $\mathcal{I}$, written as $\frac{\mathbf{P}}{\mathcal{I}}$, via the following sequence of steps:

1. Replace every default-negated literal (i.e., negated with `naf`) that occurs in a rule body in $\mathbf{P}$ by its truth value in $\mathcal{I}$.

2. For every defeasible rule in $\mathbf{P}$ of the form
   `@[tag->`R`, defeasible]` $Head$ `:-` $Body$.
   in some module M: If $\mathcal{I}_{true}(\backslash\texttt{prolog}(R, Head, M)) = \mathbf{t}$, replace the defeasible rule with the following strict rule:
   `@[strict]` $Head$`:-`$Body$, $\mathbf{f}$.

3. For every defeasible rule in $\mathbf{P}$ of the form
   `@[tag->`R`, defeasible]` $Head$ `:-` $Body$.
   in some module M: If $\mathcal{I}_{true}(\backslash\texttt{prolog}(R, Head, M)) = \mathbf{u}$, replace the defeasible rule with the following strict rule:
   `@[strict]` $Head$`:-`$Body$, $\mathbf{u}$.

4. Remove all tags from the remaining rules.

The resulting set of rules constitute the *Rulelog quotient* $\frac{\mathbf{P}}{\mathcal{I}}$.

Note that Rulelog quotients do not have `naf`-negated literals so, as mentioned earlier, every quotient has a unique least model. In other words, $LPM(\frac{\mathbf{P}}{\mathcal{I}})$ always exists and is unique.

A Rulelog semantic structure $\mathcal{I}$ is ***empty*** if $\mathcal{I}_{true}$ maps every term in $\mathbf{U}_H$ to $\mathbf{u}$, i.e., all ground atomic formulas are undefined.

We are now ready for the main definition: The ***well-founded model*** of a Rulelog knowledge base $\mathbf{P}$ with respect to an argumentation theory, $\mathbf{A}$, denoted $WFM(\mathbf{P}, \mathbf{A})$, is the limit of the following transfinite, inductively defined, sequence. Initially, $\mathcal{I}_0$ is the empty Rulelog semantic structure. Suppose $\mathcal{I}_m$ has already been defined for every $m < k$, where k is an ordinal. Then:

- $\mathcal{I}_k = LPM(\frac{\mathbf{P} \cup \mathbf{A}}{\mathcal{I}_{k-1}})$, if k is a non-limit ordinal.
- $\mathcal{I}_k = \cup_{i<k}\mathcal{I}_i$, if k is a limit ordinal.

The limit of the aforementioned inductive sequence exists and is unique, as shown in [WGK+09], so the well-mounded model is well-defined and is unique (provided the argumentation theory is fixed).

Well-founded models are regarded as canonical models of Rulelog knowledge bases, i.e., they determine the semantics of Rulelog. If $\mathbf{P}$ is a knowledge base, $\mathbf{A}$ an argumentation theory, and φ we write $(\mathbf{P}, \mathbf{A}) \models \varphi$ if and only if φ is true in the $WFM(\mathbf{P}, \mathbf{A}) \models \varphi$.

4.2 Semantics for Rulelog Syntactic Extensions

4.2.1 Path Expressions

The following transformation gives semantics to formulas that contain generalized HiLog terms via a transformation to basic formulas. First we define a

transformation `unroll`, which takes a path expression and a pure HiLog term and replaces it with a conjunction of frames and other atomic formulas. The resulting conjunction is true if and only if the result of the path expression has an element that unifies with the term.

- `unroll`$(Expr, T) \rightsquigarrow Expr = T$, if $Expr$ is a pure HiLog term.
- `unroll`$(Expr.S, T) \rightsquigarrow$ `unroll`$(Expr, ?newvar)$ `and` $?newvar[S\texttt{->}T]$, if $Expr$ is a path expression. Here $?newvar$ denotes a new variable that does not appear anywhere in the surrounding rule.
- `unroll`$(Expr[Spec_1, ..., Spec_n].S, T) \rightsquigarrow$ `unroll`$(Expr, ?newvar)$ `and` $?newvar[Spec_1, ..., Spec_n, S\texttt{->}T]$, if $Expr$ is a path expression. Here again $?newvar$ is a new variable and $Spec_1$, ..., $Spec_n$ are the expressions that are allowed inside frame formulas, as defined on page 13.

Example 4.1: Let $Expr$ be the following path expression: $p(?X).q[foo\texttt{->}23].r(?x).s(?y)$. Then `unroll`$(Expr, f(?y))$ is $p(?X)[q\texttt{->}?V1, foo\texttt{->}23]$ `and` $?V1[r(?x)\texttt{->}?V2]$ `and` $?V2[s(?y)\texttt{->}f(?y)]$. □

4.2.2 User-Defined Functions

The semantics of UDFs is defined by the following transformation. A function definition of the form

```
function foo(t1,...,tn) := Expr if Definition.
```

is converted into the rule

$$\texttt{newpred\_foo}(\texttt{t1}, ..., \texttt{tn}, \texttt{Expr}) \texttt{ :- Definition.} \tag{1}$$

where `newpred_foo` is a new $n+1$-ary predicate. Then every occurrence of the n-ary function `foo` in a rule head

```
head(...,foo(s1,...,sn),...) :- Body.
```

is rewritten into

```
head(...,?newvar,...) :- newpred_foo(s1,...,sn,?newvar), Body.
```

where `?newvar` is a new variable. In the rule body, any literal that has an occurrence of `foo(s1,...,sn)` is rewritten as follows:

```
... :- ..., somepred(...,foo(s1,...,sn),...), ...
```

is replaced with

```
... :- ..., newpred_foo(s1,...,sn,?newvar),
                somepred(...,?newvar,...), ...
```

Again, `?newvar` is a new variable here and `newpred_foo` is the predicate introduced above in (1).

Remark 4.2: Note that `foo(s1,...,sn)` can occur at any level of nesting within `somepred(...)` so the above substitution can also happen at any level.

4.3 Argumentation Theories

In this section we describe the argumentation theories for defeasible reasoning supported by Rulelog. The user can change the default argumentation theory via a directive.

- General argumentation theory with multiway exclusions (AME) — the default.
- Courteous argumentation theory with binary exclusions (CABE) — the argumentation theory corresponding to the original Courteous logic programming [Gro99].
- Cautious courteous argumentation theory with binary exclusions (CCABE) — the argumentation theory proposed in [WGK+09].

Argumentation theories are *always* loaded in their own *separate modules* in order to not interfere with knowledge bases specified by the users.

All argumentation theories have certain things in common. First, their sole purpose is to define the predicate `\prolog`, which determines which facts derived by which rules are "defeated," i.e., the very fact of their inference is to be ignored:

- `\prolog`($RuleTag$,$RuleHead$,$Module$): If `\prolog`(t,h,m) is true, then the effect is as if every defeasible ground rule instance in module m that has h as its head literal and t as its tag is treated as if it were not in the knowledge base. The formal semantics is given in Section 4.1.1.

  Remark 4.3: Observe the role of the module argument here. Rules in different modules are independent of each other and the same tag/head may be defeated in one module, but not in the other.

The second thing that unites all argumentation theories is that they take as input certain predicates (and sometimes frame formulas) defined as part of the user knowledge base. These predicates are listed below.

- `\prolog`(Tag_1,$Head_1$,Tag_2,$Head_2$): This predicate specifies that any inference made by a rule with the tag Tag_1 and head $Head_1$ has higher precedence than inferences made by the rules having the tag Tag_2 and head $Head_2$.

For convenience, users can also use a two-argument version of this predicate, which takes tags only. The two versions of this predicate are related by the following rule:
$\texttt{\textbackslash prolog}(?Tag_1, ?, ?Tag_2, ?)$ `:-` $\texttt{\textbackslash prolog}(?Tag_1, ?Tag_2)$.
Argumentation theories use only the 4-ary version of this predicate.

- $\texttt{\textbackslash prolog}(Tag_1,Head_1,Tag_2,Head_2)$: This predicate says that inferences made by the rules with the tag Tag_1 and head $Head_1$ are inconsistent with inferences made by the rules with tag Tag_2 and head $Head_2$. The exact meaning is given by the actual argumentation theories.
  This predicate is symmetric:
  $\texttt{\textbackslash prolog}(t_1,h_1,t_2,h_2) \equiv \texttt{\textbackslash prolog}(t_2,h_2,t_1,h_1)$.
  and also $\texttt{\textbackslash prolog}(?, h, ?, \texttt{neg}\, h)$ is true for any head literal h.
  For users' convenience, there is also a 2-argument version of this predicate, which is related to the 4-ary predicate as follows:
  $\texttt{\textbackslash prolog}(?, ?Head_1, ?, ?Head_2)$ `:-` $\texttt{\textbackslash prolog}(?Head_1, ?Head_2)$.
  Argumentation theories use only the 4-ary version of this predicate.

- $Excl$: $\texttt{\textbackslash exclusion}[\texttt{\textbackslash opposers->}\{H_1, ..., H_n\}]$: This frame specifies exclusions. An exclusion is similar to opposition except that it can involve more than two rule heads, but tags are not taken into account. Such a frame says that H_1, ..., H_n are part of the exclusion $Excl$ and so they should not be derived together. Exclusions are used only by the AME argumentation theory and their precise meaning is specified by that theory.

- $\texttt{\textbackslash cancel}(Tag,Head)$: This predicate specifies when a rule is *canceled*. A canceled rule is like a defeated rule, with the difference that the reason for cancellation is given directly by the user. This stands in contrast to the `\prolog` predicate, which is defined by the argumentation theory and the user has no direct control over it.
  There is also a unary version of this predicate, which is related to the binary version as follows:
  $\texttt{\textbackslash cancel}(?Tag, ?)$ `:-` $\texttt{\textbackslash cancel}(?Tag)$.
  Argumentation theories use only the binary version of this predicate.

- `strict`$(Tag,Head)$: This predicate specifies which rules are strict based on their tag/head combination.
  This predicate cannot be changed by the user directly. Instead, it is specified indirectly through the `defeasible` statement descriptor (see Section 3.3.4).

4.3.1 Argumentation Theory with Multiway Exclusions (AME)

The AME theory uses the notions of multiway rebuttal and refutation to decide whether a particular rule is to be defeated. These notions are fairly complex and can be best understood by reading the rules of the argumentation theory. Another way to be defeated is through disqualification by being canceled. A

cancellation rule can also be defeated by virtue of being overridden by the rule that it tries to cancel.

Editor's Note 4.1: This is ATCK1 □

```
/********************** \defeated ***************************/
// using general exclusions
\defeated(?Tag,?Head,?Mod) :-
    ?Exclusion#\exclusion[\opposers->{?Head,?OtherH}]@?Mod,
    ?Head@?Mod != ?OtherH@?Mod,
    candidate(?,?OtherH,?Mod),
    naf ?Exclusion[\beater(?Mod)->?Head].
// using exclusions via \opposes
\defeated(?Tag,?Head,?Mod) :-
    \opposes(?Tag,?Head,?OtherT,?OtherH)@?Mod,
    ?Head@?Mod != ?OtherH@?Mod,
    candidate(?OtherT,?OtherH,?Mod),
    naf \beater(?Head,?OtherH,?Mod).
// defeat via disqualification
\defeated(?Tag,?Head,?Mod) :- disqualified(?Tag,?Head,?Mod).

/********************** disqualification **********************/
disqualified(?Tag,?Head,?Mod) :-
    beaten_by_strict_rule(?Tag,?Head,?TagOther,?HeadOther,?Mod).
disqualified(?Tag,?Head,?Mod) :- \cancel(?Tag,?Head)@?Mod.
disqualified((?Tag1,${\cancel(?Tag2,?H2)@?Mod}),?Mod) :-
    \overrides(?Tag2,?H2,?Tag1,${\cancel(?Tag2,?H2)})@?Mod.

/***** refutation via competition through general exclusions ****/
refutes(?H1,?T2,?H2,?Exclusion,?Mod) :-
    competes(?H1,?H2,?Exclusion,?Mod),
    \overrides(?T1,?H1,?T2,?H2)@?Mod,
    candidate(?T1,?H1,?Mod).
/******* refutation via competition through \opposes ********/
refutes(?H1,?T2,?H2,?Mod) :-
    competes(?H1,?H2,?Mod),
    \overrides(?T1,?H1,?T2,?H2)@?Mod,
    candidate(?T1,?H1,?Mod).

/****** rebuttal via competition through general exclusions *****/
rebuts(?H1,?H2,?Exclusion,?Mod)  :-
    competes(?H1,?H2,?Exclusion,?Mod),
    candidate(?T1,?H1,?Mod),
    naf refutes(?H2,?T1,?H1,?Exclusion,?Mod).
/******** rebuttal  via competition through \opposes ********/
rebuts(?H1,?H2,?Mod)  :-
```

```
    competes(?H1,?H2,?Mod),
    candidate(?T1,?H1,?Mod),
    naf refutes(?H2,?T1,?H1,?Mod).

/*************************** competition **********************/
competes(?H1,?H2,?Exclusion,?Mod) :-
    ?Exclusion#\exclusion[\opposers->{?H1,?H2}]@?Mod,
    ?H1 != ?H2.
competes(?H1,?H2,?Mod) :-
    \opposes(?T1,?H1,?T2,?H2)@?Mod,
    ?H1 != ?H2.
/*************************** candidacy ************************/
candidate(?Tag,?H,?Mod) :-
    clause{@{?Tag} ?H@?Mod, ?Body},
    ?Body.
strict_candidate(?Tag,?H,?Mod) :-
    \strict(?Tag,?H)@?Mod,
    // meta-predicate to get body of a rule
    clause{@{?Tag} ?H@?Mod, ?Body},
    ?Body.
/*************************** battery ************************/
?Exclusion[\beater(?Mod)->?H] :-
    // competition via general exclusions
    competes(?H,?Beaten,?Exclusion,?Mod),
    naf rebuts(?Beaten,?H,?Exclusion,?Mod).
\beater(?H,?Beaten,?Mod) :-
    // competition via \opposes
    competes(?H,?Beaten,?Mod),
    naf rebuts(?Beaten,?H,?Mod).
beaten_by_strict_rule(?T,?H,?Tstrict,?Hstrict,?Mod) :-
    candidate(?T,?H,?Mod),
    \opposes(?T,?H,?Tstrict,?Hstrict)@?Mod,
    strict_candidate(?Tstrict,?Hstrict,?Mod).
```

4.3.2 Cautious Argumentation Theory with Multway Exclusions (CAME)

Editor's Note 4.2: TODO. This will be a fixed-up ATCK2. □

4.3.3 Argumentation Theory with Binary Exclusions (ABE)

The ABE theory also uses the notions of refutation, rebuttal, and disqualification in order to determine if a rule is to be defeated. The disqualification rules are the same as before, but the refutation and rebuttal rules are much simpler and straightforward.

Editor's Note 4.3: This is the old GCLP □

```
/********************** \defeated ***************************/
\defeated(?Tag,?Head,?Mod) :-
    defeats(?OtherTag,?OtherHead,?Tag,?Head,?Mod).
\defeated(?Tag,?Head,?Mod) :- disqualified(?Tag,?Head,?Mod).

defeats(?T1,?H1,?T2,?H2,?Mod) :- refutes(?T1,?H1,?T2,?H2,?Mod).
defeats(?T1,?H1,?T2,?H2,?Mod) :- rebuts(?T1,?H1,?T2,?H2,?Mod).

/********************** disqualification ***********************/
disqualified(?Tag,?Head,?Mod) :-
    beaten_by_strict_rule(?Tag,?Head,?TagOther,?HeadOther,?Mod).
disqualified(?Tag,?Head,?Mod) :- \cancel(?Tag,?Head)@?Mod.
disqualified((?Tag1,${\cancel(?Tag2,?H2)@?Mod}),?Mod) :-
    \overrides(?Tag2,?H2,?Tag1,${\cancel(?Tag2,?H2)})@?Mod.

/******************* refutation and rebuttal *******************/
refutes(?T1,?H1,?T2,?H2,?Mod) :-
    \overrides(?T1,?H1,?T2,?H2)@?Mod,
    conflicts(?T1,?H1,?T2,?H2,?Mod).
rebuts(?T1,?H1,?T2,?H2,?Mod)  :-
    conflicts(?T1,?H1,?T2,?H2,?Mod),
    naf refuted(?T1,?H1,?Mod),
    naf refuted(?T2,?H2,?Mod).

conflicts(?T1,?H1,?T2,?H2,?Mod) :-
    \opposes(?T1,?H1,?T2,?H2)@?Mod,
    candidate(?T1,?H1,?Mod),
    candidate(?T2,?H2,?Mod).

refuted(?T,?H,?Mod)   :- refutes(?OtherT,?OtherH,?T,?H,?Mod).

/***************************** candidacy ************************/
candidate(?Tag,?H,?Mod) :-
    clause{@{?Tag} ?H@?Mod, ?Body},
    ?Body.
strict_candidate(?Tag,?H,?Mod) :-
    \strict(?Tag,?H)@?Mod,
    // meta-predicate to get body of a rule
    clause{@{?Tag} ?H@?Mod, ?Body},
    ?Body.
/***************************** battery **************************/
beaten_by_strict_rule(?T,?H,?Tstrict,?Hstrict,?Mod) :-
    \opposes(?T,?H,?Tstrict,?Hstrict)@?Mod,
    strict_candidate(?Tstrict,?Hstrict,?Mod).
```

4.3.4 Cautious Argumentation Theory with Binary Exclusions (CABE)

The CABE theory is similar to the ABE theory, but it is more cautious in deciding when a rule is to be defeated. First, in order for another rule to be a cause of defeat of a rule, that other rule must not itself be refuted and defeated. Second, a rule may be disqualified for one additional reason: it may transitively defeat itself. This theory typically results in fewer rules being defeated and a larger number of facts might become undefined compared to ABE.

Editor's Note 4.4: This is the new GCLP □

```
/********************** \defeated ***************************/
\defeated(?Tag,?Head,?Mod) :-
    defeats(?OtherTag,?OtherHead,?Tag,?Head,?Mod),
    naf compromised(?OtherTag,?OtherHead,?Mod).
\defeated(?Tag,?Head,?Mod) :- disqualified(?Tag,?Head,?Mod).

defeats(?T1,?H1,?T2,?H2,?Mod) :- refutes(?T1,?H1,?T2,?H2,?Mod).
defeats(?T1,?H1,?T2,?H2,?Mod) :- rebuts(?T1,?H1,?T2,?H2,?Mod).

/********************** disqualification ***********************/
disqualified(?Tag,?Head,?Mod) :-
    transitively_defeats(?Tag,?Head,?Tag,?Head,?Mod).
disqualified(?Tag,?Head,?Mod) :-
    beaten_by_strict_rule(?Tag,?Head,?TagOther,?HeadOther,?Mod).
disqualified(?Tag,?Head,?Mod) :- \cancel(?Tag,?Head)@?Mod.
disqualified((?Tag1,${\cancel(?Tag2,?H2)@?Mod}),?Mod) :-
    \overrides(?Tag2,?H2,?Tag1,${\cancel(?Tag2,?H2)})@?Mod.

/******************** refutation and rebuttal *******************/
refutes(?T1,?H1,?T2,?H2,?Mod) :-
    \overrides(?T1,?H1,?T2,?H2)@?Mod,
    conflicts(?T1,?H1,?T2,?H2,?Mod).
rebuts(?T1,?H1,?T2,?H2,?Mod)  :-
    conflicts(?T1,?H1,?T2,?H2,?Mod),
    naf compromised(?T1,?H1,?Mod).

conflicts(?T1,?H1,?T2,?H2,?Mod) :-
    \opposes(?T1,?H1,?T2,?H2)@?Mod,
    candidate(?T1,?H1,?Mod),
    candidate(?T2,?H2,?Mod).
compromised(?T,?H,?Mod)  :-
    refuted(?T,?H,?Mod),
    defeated(?T,?H,?Mod).

refuted(?T,?H,?Mod)   :- refutes(?OtherT,?OtherH,?T,?H,?Mod).
rebutted(?T,?H,?Mod)  :- rebuts(?OtherT,?OtherH,?T,?H,?Mod).
```

```
/*************************** candidacy ************************/
candidate(?Tag,?H,?Mod) :-
    clause{@{?Tag} ?H@?Mod, ?Body},
    ?Body.
strict_candidate(?Tag,?H,?Mod) :-
    \strict(?Tag,?H)@?Mod,
    // meta-predicate to get body of a rule
    clause{@{?Tag} ?H@?Mod, ?Body},
    ?Body.
/*************************** battery ************************/
beaten_by_strict_rule(?T,?H,?Tstrict,?Hstrict,?Mod) :-
    \opposes(?T,?H,?Tstrict,?Hstrict)@?Mod,
    strict_candidate(?Tstrict,?Hstrict,?Mod).
/************************** transitive defeat ******************/
transitively_defeats(?T1,?H1,?T2,?H2,?Mod) :-
    defeats(?T1,?H1,?T2,?H2,?Mod).
transitively_defeats(?T1,?H1,?T3,?H3,?Mod) :-
    transitively_defeats(?T2,?H2,?T3,?H3,?Mod),
    defeats(?T1,?H1,?T2,?H2,?Mod).
```

4.4 Additional Background Axioms

This section describes the axioms that are part of the Rulelog semantics, but are not covered by the direct model-theoretic semantics of Section 4.1.

4.5 Semantic Directives

This section defines the directives that directly affect the semantics of Rulelog.

- Directive that affects the argumentation theory.
- Directive for turning off equality maintenance.
- Directives to turn on various optimizations.

Editor's Note 4.5: The above directives are to be decided upon. □

5 Rulelog Builtins and Their Semantics

References

[BBB+05] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and

S. Tabet. SWSL: Semantic Web Services Language. Technical report, W3C, April 2005. http://www.w3.org/Submission/SWSF-SWSL/.

[BM04] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. Recommendation 28 October 2004, W3C, 2004.

[BM08] M. Birbeck and S. McCarron. CURIE Syntax 1.0: A syntax for expressing compact URIs. Technical report, W3C, May 2008. http://www.w3.org/TR/curie/.

[DG05] M. Duerst and M. Guignard. Internationalized resource identifiers (iris). Technical report, Internet Engineering Task Force, January 2005. http://www.ietf.org/rfc/rfc3987.txt.

[GDG+] B. Grosof, M. Dean, S. Ganjugunte, S. Tabet, , and C. Neogy. SweetRules: An open source platform for semantic web business rules. Web site. http://sweetrules.projects.semwebcentral.org/.

[Gro99] B.N. Grosof. A courteous compiler from generalized courteous logic programs to ordinary logic programs. Technical Report RC 21472, IBM, July 1999.

[KYWZ13] M. Kifer, G. Yang, H. Wan, and C. Zhao. FLORA-2: User's manual. The FLORA-2 Web Site, 2013. http://flora.sourceforge.net/docs/floraManual.pdf.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer-Verlag, 1987.

[Prz94] T. Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:141–187, 1994.

[SW11] T. Swift and D.S. Warren. Xsb: Extending the power of prolog using tabling. *Theory and Practice of Logic Programming*, 2011.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[WGK+09] H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang. Logic programming with defaults and argumentation theories. In *Int'l Conference on Logic Programming*, July 2009.

Index