IRIS - Integrated Rule Inference System - API and User Guide

April 3, 2008

Contents

1 Introduction **4**
- 1.1 Purpose . . . 4
- 1.2 Audience . . . 4
- 1.3 Scope . . . 4

2 Description **4**
- 2.1 Datalog . . . 4
- 2.2 Features . . . 4
- 2.3 Evaluation Process . . . 5

3 Architecture **5**
- 3.1 Supported Strategies . . . 6
- 3.2 Program optimizations . . . 7
- 3.3 Rule Safety Processing . . . 7
- 3.4 Stratification Algorithms . . . 7
- 3.5 Rule Re-ordering optimizations . . . 8
- 3.6 Rule optimizations . . . 8
- 3.7 Rule Compilers . . . 9
- 3.8 Rule Evaluators . . . 10
- 3.9 Miscellaneous Components . . . 10
  - 3.9.1 Storage and Indexing . . . 10
  - 3.9.2 Built-in Predicates . . . 11
  - 3.9.3 Configuration . . . 11
- 3.10 Errors . . . 12

4 Stratification **12**
- 4.1 Stratified negation . . . 12
- 4.2 Locally stratified negation . . . 13

5 Safe Rules **14**
- 5.1 Algorithm . . . 14
- 5.2 Unsafe Rules . . . 15

6 Datatypes and Built-in Predicates **15**
- 6.1 Supported datatypes . . . 15
- 6.2 Built-in Predicates . . . 15
- 6.3 Behaviour of built-ins with incompatible datatypes . . . 15
- 6.4 Negated built-ins . . . 15
- 6.5 Arithmetic built-ins . . . 16
- 6.6 Custom built-in predicates . . . 16
  - 6.6.1 Extend one of the base classes . . . 16
  - 6.6.2 Register the custom built-in for parsing . . . 17

7 API guide **17**
7.1 Creating objects with the Java API 17
7.2 Creating objects using the parser 17
7.3 Evaluating a program 18
7.3.1 Configuration 18

8 External Data-sources **19**

A Datalog Grammar Support **20**
A.1 Datalog 20
A.2 Data types 20
A.3 Built-in predicates 20

1 Introduction

1.1 Purpose

This guide is intended to give a short introduction to using the Integrated Rule Inference System (IRIS) and its application programming interface (API).

1.2 Audience

This guide is for software developers who will be integrating IRIS in to their application as well as logicians/researchers who wish to understand the capabilities of the IRIS reasoner.

1.3 Scope

This guide describes the reasoner architecture, evaluation algorithms and various optimizations.

However, this document does not attempt to explain the theory of logic programming and only provides a brief description of the evaluation strategies employed.

2 Description

2.1 Datalog

One Logic Programming based formalism that has been thoroughly analyzed is Datalog[1], which was originally developed as a database query and rule language. Datalog is based on a simplified version of the Logic Programming paradigm (it is a syntactic subset of Prolog) with its main focus on the processing of large amounts of data from relational databases. Several relevant complexity results of Datalog in regard to query answering have been derived. Querying a static knowledge base in general has polynomial time complexity, but is exponential otherwise[2].

Datalog can be used in a wide variety of applications, including Description Logic Programming (DLP)[3], rule languages from the WSML family[4] and RDF[5] reasoning.

2.2 Features

IRIS is an open-source Datalog reasoner that can evaluate safe or unsafe datalog extended with function symbols, XML schema data types, built-in predicates and (locally) stratified or well-founded negation as failure.

It is delivered in three java 'jar' files. One contains the reasoning engine, another contains the parser and the last contains some utility programs including two applications that provide a user interface to the IRIS engine. These

applications are useful for experimenting with Datalog and various evaluation options.

IRIS is licensed under the GNU lesser GPL and hosted by Sourceforge[1]. More detailed information is available on the IRIS home page[2].

2.3 Evaluation Process

IRIS evaluates queries over a knowledge base. The knowledge-base consists of facts (ground atomic formula) and rules. The combination of facts, rules and queries is known as a logic program and forms the input to a reasoning (query-answering) task.

The creation of the knowledge-base is achieved in one of two ways:

- Create the java objects representing the components of the knowledge-base using the API (described in section 7.1 on page 17)
- Parse an entire Datalog program written in human-readable form using the parser. The grammar supported by IRIS is described in the datalog grammar guide in section 7.2 on page 17).

For each query submitted to the knowledge-base, IRIS will return the variable bindings, i.e. the set of all tuples that can be found or inferred from the knowledge-base that satisfy the query.

3 Architecture

IRIS has been designed to be as modular as possible and is comprised of a number of loosely coupled components that implement well-defined Java interfaces. This approach allows more evaluation strategies to be added over the course of time. For the initial releases of IRIS, it was decided to concentrate on bottom-up evaluation techniques. However, future top-down and hybrid techniques are envisaged and planned for. When the time comes, new implementations can be easily 'plugged-in' and used without any requirement to modify the existing code-base.

The advantage of using bottom-up techniques is that they are easily understood and implemented. The disadvantage is that for large or complex knowledge-bases, the minimal model may be too expensive to calculate in either time or storage requirements. However, 'Magic Sets'[6] is a well-researched program optimization technique that mitigates the disadvantages of bottom-up evaluation by re-writing the rules of the knowledge-base to answer a specific query. The end effect is that a far more efficient evaluation occurs where only those tuples likely to be involved in answering the query are computed.

Therefore, at present, IRIS uses a combination of bottom-up evaluation for simplicity, combined with magic sets optimization for efficiency. This particular combination is well-researched[7][8], easy to implement, fast and efficient.

[1] `http://sourceforge.net/projects/iris-reasoner`
[2] `http://www.iris-reasoner.org`

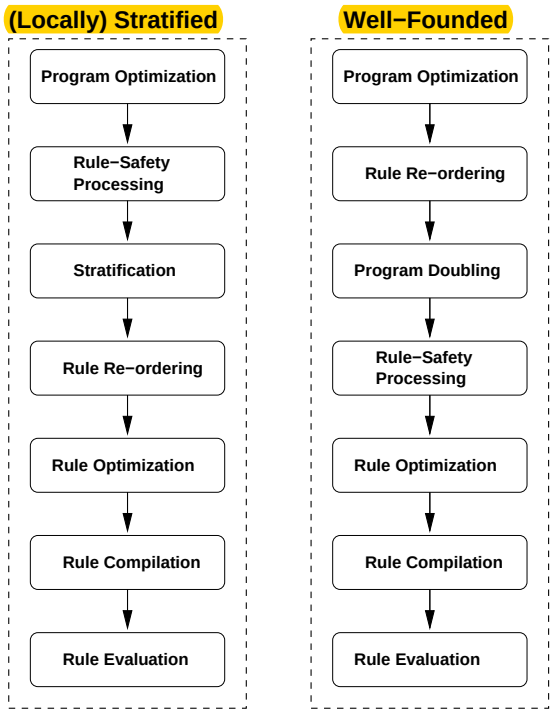

Figure 1: Stratified and Well-founded evaluation strategies

3.1 Supported Strategies

A number of evaluation strategies are supported and it is intended that more strategies will be created over time. Broadly speaking, an evaluation strategy represents a particular combination of processing elements. There are two basic evaluation strategies currently implemented, see Figure 1.

The first is a (locally) stratified[1] technique that includes a stratification step where each stratification algorithm is applied in turn until one succeeds. From then on, the rest of the processing steps are completed until a minimal model for the knowledge-base is created. Queries can then be executed against this model.

The second technique uses an alternating fixed point algorithm[9] to compute the well-founded model. This approach is required for input programs that are not stratified. Instead of stratification, a program doubling step is introduced that creates the 'positive' and 'negative' versions of the logic program for input to the alternating fixed point algorithm.

The individual processing elements are described below.

3.2 Program optimizations

As mentioned above, the Magic Sets optimization technique re-writes the rule-set according to the query so that only tuples likely to be involved in satisfying the query are computed. It can be shown that this approach allows bottom-up evaluation to rival top-down techniques in efficiency[10]. In essence, the application of magic sets allows only a sub-set of the minimal model to be computed, i.e. that part which contains all tuples that will be used to answer the query. The disadvantage, is that a new sub-set of the model must be computed for each new query. Therefore, magic sets allows faster knowledge-base initialization times at the expense of longer query times. Whether magic sets is used or not can be configured programmatically to suit the environment in which IRIS is being used.

Another simpler program optimization technique is rule-filtering. This technique is usually used in combination with Magic Sets and simply involves building a dependency graph between all rule predicates and removing those rules that can not influence the query result, thus reducing the size of the minimal model computation.

3.3 Rule Safety Processing

The algorithm for detecting unsafe rules is taken from [11] page 105. A rule is safe if all the variables occurring in the head and body are limited. A variable is limited if it appears at least once in a positive ordinary body literal, is equated with a constant in a positive equality predicate or is equated with another variable known to be limited.

In order to process unsafe rules IRIS can be configured to use a rule augmentation processor (see section 5 on page 14).

3.4 Stratification Algorithms

A globally stratified logic program is one where the rules can be arranged in to strata, where each stratum contains rules whose positive body predicates match the heads of rules that are in the same or a lower stratum and whose negated body predicates match the heads of rules that are in a lower stratum. Arranging the rules like this allows each stratum to be fully evaluated before moving to the next higher stratum. This evaluation is guaranteed to be monotone.

IRIS has two separate stratification algorithms. The first algorithm is the simplest and attempts to stratify the rules assuming the program is globally stratified as described above[1]. If the program is not globally stratified the algorithm fails. The second algorithm assumes the program is locally stratified[12].

Local stratification occurs when a rule has a direct or indirect dependency upon itself through negation, but the presence of constant term values allow the separation of the domain of tuples used as input to the rule and the domain of tuples produced by the rule. For example, the following rule appears to be

unstratified:

$$p(2, ?X) :- q(?X), \neg p(3, ?X)$$

because the rule head predicate has a direct negative dependency upon itself. However, the rule can only produce tuples whose first term value is 2 and can only use input tuples whose first term value is 3. Therefore no recursive dependency exists at all and this rule can be evaluated normally.

Locally stratified logic programs can be far more complicated than the simple example shown above. IRIS uses a novel technique to evaluate locally stratified logic programs that does not require the use of a well-founded semantics (see section 4.1 on page 12).

3.5 Rule Re-ordering optimizations

After rules have been allocated to strata (or not as in the case of the well-founded evaluation strategy) there can still be significant performance improvements if the rules are evaluated in a better order, i.e. rules that produce tuples that feed other rule bodies are evaluated earlier. The standard IRIS rule re-ordering optimizer simply searches for the first positive body predicate of each rule and builds a dependency graph between these positive body predicates and rule heads. Rules are then arranged following this directed graph.

3.6 Rule optimizations

A number of optimizations can be achieved on a per rule basis. The default configuration contains the following four optimizers, but more user defined optimizers can easily be added.

Join condition This optimizer attempts to use the same variable for join conditions, e.g.

$$p(?X) :- q(?X), r(?Y), ?X = ?Y$$

would be changed to

$$p(?X) :- q(?X), r(?X)$$

This can significantly reduce the number of intermediate tuples produced during a sequence of cartesian products.

Replace variables with constants This has the effect of pushing selection criteria in to the evaluation of a relation, such that fewer tuples are processed, e.g.

$$p(?X, ?Y) :- q(?X, ?Z), ?Z = 2$$

would be changed to

$$p(?X, ?Y) :- q(?X, 2)$$

Re-order literals Re-arrange the literals in a rule body so that the most restrictive literals appear first. The preferred order is: positive literals with no variables, built-ins with no variables, positive literals, built-ins and negated literals. However, negated literals and built-ins can be pushed earlier in to the rule body as soon as all their variables are bound.

Remove duplicate literals Remove any literal that appears twice within the rule with the same variables.

3.7 Rule Compilers

Compiling an input rule simply involves pre-computing all possible information required for rule evaluation. The input rule is transformed in to a compiled rule that can be quickly evaluated using a rule evaluator.

The first step is to create views on each literal. A view is analogous to a view in a relational database and is created from the underlying relation for a predicate and the tuple as it appears in the rule body predicate. A view is itself a relation for the purposes of rule evaluation, as the following examples demonstrate:

$$p(?X, ?Y) :- q(?X, ?Y), r(?Y, ?Y), s(1, ?X), t(g(?Y, ?Z))$$

$q(?X, ?Y)$ is a simple view that selects all tuples from the relation for 'q'.
$r(?Y, ?Y)$ is a view that selects only those tuples where both terms are equal. This view appears as a unary relation.
$s(1, ?X)$ is a view that selects values from the second term of the relation for 's' where the first term is equal to 1. This view also appears as a unary relation.
$t(g(?Y, ?Z))$ is a view that selects the two term parameters of constructed terms from the relation for 't'. This view converts a unary relation in to a binary view.

The next step is to assign join objects and indexes. Since all joins in Datalog are natural joins, the compiling stage looks for all matching variables between two adjacent views, calculates the join indices and creates indexes. The indexes used in the default rule compiler are hash-based and therefore this approach is equivalent to performing a hash join. The advantages of a hash join over a sort-merge join are that the underlying views are not required to be sorted in any way, rather simply grouped according to matching join indices. This approach appears to scale much better than maintaining sorted relations (as in previous versions of IRIS) and is much faster overall. When an evaluation is highly iterative, the cost of maintaining a sorted relation as tuples are added on each iteration becomes very expensive.

An important optimization that this approach allows is that of caching of indexes, views and relations. Bottom-up evaluation can be expensive computationally when the rule set is highly recursive. However, when a rule is compiled in to an object model just described, the fetching of matching tuples for joins does not have to re-evaluate an entire view of a relation, because a view need

only process the extra tuples added since the last rule evaluation and the index only need process those matching tuples from the view.

3.8 Rule Evaluators

Closely related to rule compilation is rule evaluation. A rule evaluator simply applies facts to rules to generate new facts. Two rule evaluators are provided as described in Ullman[1], the 'Naive' evaluator and the 'Semi-Naive' evaluator.

The 'Naive' evaluator simply applies all facts to all rules in each round of evaluation and stops when no new facts are produced. 'Semi-Naive' attempts to avoid inferring the same fact twice in the same way by making use of the tuples generated in the previous step.

In each round of evaluation it uses the deltas, i.e. the set of new facts from the previous round, to substitute in to each rule once for each positive ordinary literal.

Consider a rule with two literals:

```
p(?X) :- q(?X), r(?X)
```

In the first round the whole relation for 'q' is joined with the whole relation for 'r' to produce new tuples for the relation 'p'.

During this iteration, other rules might also generate new tuples for 'q' and 'r', such that the rule must be evaluated again in the next iteration. However, in an attempt to avoid generating tuples for 'p' that are already known, the join is made twice using only the incremental tuples from the previous iteration.

Iteration 1: New tuples for p, q, r are generated: δp_1, δq_1, δr_1

Iteration 2: New tuples are generated for the rule above by joining δq_1 and r (the whole relation for 'r') and by joining q and δr_1

In other words, in each iteration a rule with N positive ordinary literals labeled l_{1-N} is evaluated N times substituting the incremental relations from the previous step for each each literal in turn.

3.9 Miscellaneous Components

The following utility components are common to all evaluation strategies.

3.9.1 Storage and Indexing

Although IRIS currently computes all inferred data in-memory, it is planned to allow for alternative implementations of relations and indexes that can use any medium, the most likely being flat files or a relational database.

New implementations for relations and indexes can easily be integrated in to IRIS by creating classes that implement the relation and index interfaces. To use these new implementations, the configuration object for the knowledge-base (see below) needs only to have new factory objects added for these new implementations.

When an IRIS knowledge-base is initialized, the complete rule-set and set of starting ground facts must be passed to the knowledge-base factory. However, this is not always convenient, especially when the data set is large. It may be that the data set does not fit in to memory or takes too long to parse and format. In any case, not all the data may not be required for evaluation. For these situations, IRIS allows the user to supply external data sources at initialization time. An external data source is simply a user supplied Java object that implements the external data source interface. The storage mechanism used is left entirely to the class implementor. The external data source must simply answer requests from the reasoner to provide facts for the given predicate and selection criteria during program evaluation.

3.9.2 Built-in Predicates

IRIS comes with a large set of built-in predicates that can be used in the bodies of rules. They include:

- Equality, inequality, assignment, unification and regular expressions.
- Less, less or equal, greater, greater or equal, that take in to account type and floating-point round-off errors.
- Unary type checking, e.g. 'is integer', for all supported data types and binary 'same type' comparison.
- Addition, subtraction, multiplication, division and modulus.

A selection of base classes are provided so that user-defined built-in predicates can be created easily. Furthermore, mechanisms are provided to allow the parser to recognize and automatically create instances of user-defined built-ins.

3.9.3 Configuration

IRIS can be configured at the point where a knowledge-base is created. All configuration parameters are collected together in a single configuration class that is passed to the knowledge-base factory, thus allowing a highly flexible combination of standard and user-provided components. The configuration class contains these categories of parameters:

- Factories for evaluation strategies, rule compilers, rule evaluators, relations and indexes.
- Termination parameters (time out, maximum tuples, maximum complexity)
- Numerical behavior, i.e. significant bits of floating point precision for comparison and divide by zero behavior
- External data source objects

- Program optimizers, rule optimizers and a rule re-ordering optimizer
- Rule set stratifiers
- Rule-safety processor for detecting unsafe rules or making unsafe rules safe

3.10 Errors

A number of problems can occur that halt the evaluation of a query. These problems are indicated by throwing an exception of one of the following types:

EvaluationException is the superclass of all exceptions that halt the evaluation process.

ProgramNotStratifiedException indicates, that the input program is not stratified (see section 4.1 on page 12) when using the stratified evaluation strategy.

RuleUnsafeException indicates, that an unsafe rule was detected (see section 5 on page 14) without unsafe rule support turned on.

4 Stratification

4.1 Stratified negation

The 'meaning' of negation in logic programs has been discussed at length in literature. Here we adopt the relational model and describe the following construct:

```
p(X) :- q(X), not r(X)
```

as meaning, that the relation associated with predicate 'p' contains all those values from predicate 'q' that are not in predicate 'r'. In other words, the set difference of 'q' and 'r'.

Traditional forward chaining methods for evaluating logic programs involve simply using the values of tuples from predicates and applying them to the program's rules to generate more tuples.

Without negation such techniques are guaranteed to be monotone. However, in the presence of negation, rules that generate tuples for a predicate that is used in negated sub-goals of other rules, must be 'fully' evaluated before evaluation of the dependant rules begins.

Consider what would happen if we have the following:

```
p(X) :- q(X), not r(X) ...(1)
r(X) :- t(X)          ...(2)
q(a)
q(b)
t(a)
```

If the known facts are applied to rule (1) first, the following new facts are generated:

```
p(a)
p(b)
```

Then applying the known facts to rule (2) produces the following:

```
r(a)
```

However, the existence of fact r(a) should have precluded the inference of fact p(a) in rule (1).

In order to ensure that rule evaluation is monotone, rules must be evaluated in a specific order.

For any general rule:

$p :- L_1 ... L_m, N_1 ... N_p$

where $L_1 ... L_m$ are positive literals and $N_1 ... N_p$ are negative literals, existing stratification algorithms require that the rule 'p' be allocated to a strata that is at least as high as any rule that has a head matching each of its positive literals and at least one higher than any rule that has a head matching each of its negative literals.

Such a scheme would therefore require that rule (2) above is evaluated before rule(1).

However, this approach precludes the evaluation of any logic program containing a rule that has a negative dependency upon itself.

In order for IRIS to evaluate a logic program when not using the well-founded semantics, it must be stratified.

4.2 Locally stratified negation

There are genuine reasoning activities that can lead to the creation of logic programs containing rules that do contain a negative dependency to themselves, but can still be evaluated in a meaningful way, because of the presence of constants in the rules that separate the domains of tuples used as input to the rule and tuples produced by the rule.

Consider:

```
p(a,X) :- q(X), not p(b,X)
```

This rule can produce tuples (a,?) for the relation associated with predicate 'p' from tuples (b,?) also associated with the relation for 'p'. However, no special treatment is required, because nothing produced by the rule can be used as input to the rule, because of the presence of constants 'a' and 'b'.

A more complicated scenario is as follows:

```
p(a,X) :- r(X), not q(b,X)
q(X,Y) :- p(X,Y)
```

Here the second rule can produce tuples for input to the first rule and vice versa. However, if we consider the second rule to be two rules:

```
q(b,Y) :- p(b,Y)
q(X,Y) :- p(X,Y), X != b
```

Then we see that the complete set of rules is still stratified.

5 Safe Rules

An unsafe rule is one in which a variable is used, but has no binding. In essence, the entire universe of possible values must be substituted for this variable, which is clearly impractical.

When IRIS is configured not to allow unsafe rules, the standard rule-safety processor (org.deri.iris.rules.RuleValidator) is used. This processor simply examines each rule and indicates if any rule is unsafe and exactly why it is unsafe. Inputting a program containing an unsafe rule without unsafe rule support turned on will result in a specific exception being thrown containing a message explaining which rule is unsafe and which variables are problematic.

5.1 Algorithm

The algorithm for detecting unsafe rules is taken from [11] page 105. A rule is considered safe if all variables are limited. A variable is limited if:

- It appears in a positive ordinary predicate
- It appears in a positive equality with a constant, e.g. ?X = 'a'
- It appears in a positive equality with another variable known to be limited, e.g. ?X = ?Y, ?Y = 'a'

However, rule validation in IRIS can be parameterised to allow the relaxation of two aspects of this algorithm, specifically:

- variables that ONLY appear in a negated ordinary predicate (and nowhere else) can still make for a safe rule, because such a rule can be re-written to move the negated sub-goal to a separate rule, see the example in [11], page 129-130
- Furthermore, variables that appear in arithmetic predicates can also be considered limited if all the other variables are known to be limited, e.g. ?X + ?Y = ?Z, ?X = 3, ?Z = 4, implies that ?Y is also limited

These two relaxations of the definition of a safe rule are configurable (on/off) in the RuleValidator associated with the Configuration object.

If an unsafe rule is detected during evaluation when unsafe rule support is not switched on then a RuleUnsafeException is thrown containing details of why the rule is considered unsafe.

5.2 Unsafe Rules

In order to process unsafe rules IRIS can be configured to use a rule augmentation processor. This processor uses a technique suggested by Gelder[13] that adds a 'universe' predicate for each unbound variable. This universe predicate automatically contains all term values that appear anywhere in the input program or that are created during program evaluation.

6 Datatypes and Built-in Predicates

6.1 Supported datatypes

IRIS supports the data types defined in the WSML specification[3], which are a subset of the XML schema datatypes.

These data types are also discussed in appendix A.2 on page 20.

6.2 Built-in Predicates

The complete list of built-in predicates is given in appendix A.3 on page 20).

Additionally, user-defined built-in predicates can be created (see section 6.6 on page 16).

6.3 Behaviour of built-ins with incompatible datatypes

Built-in predicates will evaluate to false if the operands are incompatible with the predicate, e.g. multiplying two dates.

Built-in predicates will evaluate to false if the operands are incompatible with each other, e.g. adding an integer to a string.

6.4 Negated built-ins

Negation in IRIS means 'negation as failure', so the meaning of the expression 'p(X) and not q(X)' is the relation containing every value of X for which p() is true, removing every value of X for which q() is true. In this context, care must be taken when using negation with built-in predicates. Consider the following program:

```
p(1,2).
p(2,3).
p(4,3).
p('a',4).

q(x,y) :- p(x,y), x >= y.

?-q(x,y).
```

[3] http://www.wsmo.org/TR/d16/d16.1/v0.21/#sec:wsml-builtin-datatypes

This produces the result set:

```
p(4,3)
```

However this program:

```
p(1,2).
p(2,3).
p(4,3).
p('a',4).

q(x,y) :- p(x,y) and not x < y.

?-q(x,y).
```

Produces this result set:

```
p(4,3) p('a',4)
```

As can be seen from this example, 'not $X < Y$' is not the same as '$X \geq Y$'

6.5 Arithmetic built-ins

IRIS will automatically convert the result of an arithmetic evaluation to the most precise type for both terms, e.g. a *double* value + a *float* value will result in a *double*, and a *float* value + an *integer* value will also result in a *double*.

6.6 Custom built-in predicates

To create and use a custom built-in predicate there are a few steps to follow:

- Extend one of the built-in base classes (AbstractBuiltin, ArithmeticBuiltin, BooleanBuiltin)
- If the parser is required to recognise the custom built-in, it must be registered with the BuiltinRegister object associated with the parser.

6.6.1 Extend one of the base classes

There are 3 things that must be implemented:

1. a constructor taking an ITerm array as input that will contain the constants and variables occurring during evaluation
2. depending on which base class was extended, implement one of:
   - AbstractBuiltin.evaluateTerms(ITerm[] terms, int[] variableIndexes)
   - BooleanBuiltin.computeResult(ITerm[] terms)
   - ArithmeticBuiltin.computeMissingTerm(int missingTermIndex, ITerm[] terms)
3. override IAtom.getPredicate() to return the predicate object describing your built-in (with attributes 'name' and 'arity')

6.6.2 Register the custom built-in for parsing

The Parser has a BuiltinHelper member object that is used to test if a predicate symbol is a built-in in or not. If it is a built-in, the BuiltinRegister will create a new object of the correct type. By default, the Parser will contain a BuiltinRegister with all the standard built-ins registered and this can be obtained from the Parser and modified (either to add new built-ins or remove ones currently registered). The javadoc for BuiltinRegister has more details. For an example, see FahrenheitToCelsiusBuiltin.java in test/org.deri.iris.builtins.

7 API guide

7.1 Creating objects with the Java API

Rules, facts, queries and their components are created using factories. The most important ones are described below:

org.deri.iris.api.factory.IProgramFactory creates programs with or without initial values.

org.deri.iris.api.factory.IBasicFactory creates tuples, atoms, literals, rules and queries.

org.deri.iris.api.factory.ITermFactory creates variables, strings and constructed terms.

org.deri.iris.api.factory.IConcreteFactory creates all other sorts of terms (see section 6.1 on page 15).

org.deri.iris.api.factory.IBuiltinsFactory creates built-in atoms provided by IRIS (see section 3 on page 23).

The `org.deri.iris.factory.Factory` class holds `static final` instances of all the factories, so they can be easily
(e.g `import static org.deri.iris.factory.Factory.CONCRETE;`).

For a more complete list of methods, input parameters and return values it is recommended to read the javadoc[4].

7.2 Creating objects using the parser

Instead of creating the java objects by hand, the `org.deri.iris.compiler.Parser` can be used to parse a datalog program. The grammar used by the parser is described in the grammar guide.

[4] http://www.iris-reasoner.org/snapshot/javadoc/

7.3 Evaluating a program

After a the components of a logic program have been created, either step by step using the API factories or using the parser, a knowledge base can be created and queries evaluated by following these steps:

Choose a configuration A default configuration object can be obtained from the KnowledgeBaseFactory class. Modify this object to change the KnowledgeBase behaviour.

Instantiate a KnowledgeBase Pass the configuration object, starting facts and rules to the KnowledgeBaseFactory.createKnowledgeBase() method.

Execute queries After initialisation queries can be executed against the KnowledgeBase by calling execute(). Two variations of this method are available. The first one just accepts a query and the second accepts a query and an array for variable bindings. This second method can be useful if the query is complex and the order of variables is not obvious.

7.3.1 Configuration

IRIS can be configured at the point where a knowledge-base is created. All configuration parameters are collected together in a single configuration class that is passed to the knowledge-base factory, thus allowing a highly flexible combination of standard and user-provided components.

The configuration class contains these categories of parameters:

factories for evaluation strategies, rule compilers, rule evaluators, relations and indexes.

termination parameters for termination conditions (time out, maximum tuples, maximum complexity)

numerical behaviour significant bits of floating point precision for comparison, divide by zero behaviour

external data sources collection of external data source objects

optimisers collections of program optimisers, rule optimisers and a rule reordering optimiser

stratifiers collection of rule stratifiers

rule-safety processor for detecting unsafe rules or making unsafe rules safe

8 External Data-sources

IRIS allows the user to reason with data stored outside of the reasoner, i.e. not passed in at the point where the KnowledgeBase is instantiated. To use an external data source, simply create a class that implements the `org.deri.iris.api.storage.IDataSource` interface and add an instance of this class to the `externalDataSources` list in the `Configuration` object prior to instantiating the `KnowledgeBase`.

This approach allows the data to be stored in any format. The user-defined external data source is simply required to answer requests from the reasoner to provide tuples for predicates as and when they are needed during query evaluation.

A Datalog Grammar Support

Datalog is a database query language that is syntactically a subset of Prolog. Its origins date back to around 1978 when Herve Gallaire and Jack Minker organized a workshop on logic and databases. [5]

A.1 Datalog

IRIS evaluates logic programs that contain rules and facts (the knowledge base) and queries to be evaluated against this knowledge base.

All rules, facts and queries must be terminated by a '`.`'.

rules consist of a head and a body. Both, the head and the body, are lists of literals where the literals are separated by '`,`' and the head and the body are separated by '`:-`'. The '`,`' means 'and', e.g.
'`ancestor(?X, ?Y) :- ancestor(?X, ?Z), ancestor(?Z, ?Y).`'
IRIS requires that the head contains exactly one literal, whereas the body can contain zero or more literals.

facts are instances of predicates with constant terms, e.g. '`ancestor('john', 'odin').`'

queries are a list of literals prefixed with a '`?-`'.

literals are positive or negative atoms '`<atom>`' or '`not <atom>`'.

atoms have the format '`<predicate-symbol>(<terms>)`'. The terms must be separated by '`,`', e.g. '`ancestor('john', 'garfield')`'.

terms are either constants, variables or constructed terms (function symbols).

variables are simple strings prefixed with a '`?`', e.g. '`?VAR`'.

A.2 Data types

The data types that IRIS supports are described in table 1 on page 21.

A.3 Built-in predicates

When the parser reads a predicate name that corresponds to a built-in predicate, it will translate it to an object of the correct built-in type. All supported built-in predicates are described in table 3 at page 23.

Custom built-ins can also be registered, e.g. if a built-in with the predicate symbol '`ATOI`' is registered then the syntax will be '`ATOI(?MY_STRING, ?MY_INT)`'.

[5] http://en.wikipedia.org/wiki/Datalog

datatype	syntax
string	'<string>' _string('<string>')
decimal	'-'?<integer>.<fraction> _decimal('-'?<integer>.<fraction>)
integer	'-'?<integer> _integer('-'?<integer>)
float	_float(<integer>.<fraction>)
double	_double(<integer>.<fraction>)
iri	_'<iri>' _iri('<iri>')
sqname	<string>#<string> _sqname('<string>#<string>')
boolean	_boolean(<string>)
duration	_duration(<year>, <month>, <day>, <hour>, <minute>, <second>) _duration(<year>, <month>, <day>, <hour>, <minute>, <second>, <millisec>)
datetime	_datetime(<year>, <month>, <day>, <hour>, <minute>, <second>) _datetime(<year>, <month>, <day>, <hour>, <minute>, <second>, <tzHour>, <tzMinute>) _datetime(<year>, <month>, <day>, <hour>, <minute>, <second>, <millisec>, <tzHour>, <tzMinute>)
date	_date(<year>, <month>, <day>) _date(<year>, <month>, <day>, <tzHour>, <tzMinute>)
time	_time(<hour>, <minute>, <second>) _time(<hour>, <minute>, <second>, <tzHour>, <tzMinute>) _time(<hour>, <minute>, <second>, <millisec>, <tzHour>, <tzMinute>)
gyear	_gyear(<year>)
gyearmonth	_gyearmonth(<year>, <month>)
gmonth	_gmonth(<month>)
gmonthday	_gmonthday(<month>, <day>)
gday	_gday(<day>)
hexbinary	_hexbinary(<hexbin>)
base64binary	_base64binary(<base64binary>)

Table 1: All supported datatypes

name	syntax	supported operations
add	`?X + ?Y = ?Z` `ADD(?X, ?Y, ?Z)`	numeric + numeric = numeric date + duration = date duration + date = date time + duration = time duration + time = time datetime + duration = datetime duration + datetime = datetime duration + duration = duration
subtract	`?X - ?Y = ?Z` `SUBTRACT(?X, ?Y, ?Z)`	numeric - numeric = numeric date - duration = date date - date = duration time - duration = time time - time = duration datetime - duration = datetime datetime - datetime = duration duration - duration = duration
multiply	`?X * ?Y = ?Z` `MULTIPLY(?X, ?Y, ?Z)`	numeric x numeric = numeric
divide	`?X / ?Y = ?Z` `DIVIDE(?X, ?Y, ?Z)`	numeric / numeric = numeric
equal	`?X = ?Y` `EQUAL(?X, ?Y)`	any type = same type numeric = numeric any type = different type (always false)
not equal	`?X != ?Y` `NOT_EQUAL(?X, ?Y)`	any type $\neq$ same type numeric $\neq$ numeric any type $\neq$ different type (always true)
less	`?X < ?Y` `LESS(?X, ?Y)`	any type $<$ same type numeric type $<$ numeric type
less-equal	`?X <= ?Y` `LESS_EQUAL(?X, ?Y)`	any type $\leq$ same type numeric type $\leq$ numeric type
greater	`?X > ?Y` `GREATER(?X, ?Y)`	any type $>$ same type numeric type $>$ numeric type
greater-equal	`?X >= ?Y` `GREATER_EQUAL(?X, ?Y)`	any type $\geq$ same type numeric type $\geq$ numeric type
same type	`SAME_TYPE(?X, ?Y)`	any type same_type_as any type
regular expression match	`REGEX(?X, ?Y)`	string matches pattern (string term) any other type is false

Table 2: All supported binary and ternary built-in predicates

name	syntax	supported operations
is base64binary	`IS_BASE64BINARY(?X)`	true iff ?X is of type base64binary
is boolean	`IS_BOOLEAN(?X)`	true iff ?X is of type boolean
is date	`IS_DATE(?X)`	true iff ?X is of type date
is datetime	`IS_DATETIME(?X)`	true iff ?X is of type datetime
is decimal	`IS_DECIMAL(?X)`	true iff ?X is of type decimal
is double	`IS_DOUBLE(?X)`	true iff ?X is of type decimal
is duration	`IS_DURATION(?X)`	true iff ?X is of type duration
is float	`IS_FLOAT(?X)`	true iff ?X is of type float
is gday	`IS_GDAY(?X)`	true iff ?X is of type gday
is gmonth	`IS_GMONTH(?X)`	true iff ?X is of type gmonth
is gmonthday	`IS_GMONTHDAY(?X)`	true iff ?X is of type gmonthday
is gyear	`IS_GYEAR(?X)`	true iff ?X is of type gyear
is gyearmonth	`IS_GYEARMONTH(?X)`	true iff ?X is of type gyearmonth
is hexbinary	`IS_HEXBINARY(?X)`	true iff ?X is of type hexbinary
is integer	`IS_INTEGER(?X)`	true iff ?X is of type integer
is iri	`IS_IRI(?X)`	true iff ?X is of type iri
is numeric	`IS_NUMERIC(?X)`	true iff ?X is of any numeric type (integer, float, double, decimal)
is sqname	`IS_SQNAME(?X)`	true iff ?X is of type sqname
is string	`IS_STRING(?X)`	true iff ?X is of type string
is time	`IS_TIME(?X)`	true iff ?X is of type time

Table 3: All supported unary built-in predicates

References

[1] Ullman, J.: Principles of Database Systems. WH Freeman & Co. New York, NY, USA (1983)

[2] Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems (TODS) **22**(3) (1997) 364–418

[3] Grosof, B., Horrocks, I., Volz, R.: Description logic programs: combining logic programs with description logic. Proceedings of the 12th international conference on World Wide Web (2003) 48–57

[4] de Bruijn, J., Lausen, H., Krummenacher, R., Polleres, A., Predoiu, L., Kifer, M., Fensel, year=2005, D.: (D16. 1v0. 2 The Web Service Modeling Language WSML)

[5] Lassila, O., Swick, R., et al.: Resource Description Framework (RDF) Model and Syntax Specification. (1999)

[6] Beeri, C., Ramakrishnan, R.: On the power of magic. ACM Press New York, NY, USA (1987)

[7] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic sets and other strange ways to implement logic programs. Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems (1986) 1–15

[8] Chen, Y.: Magic Sets and Stratified Databases. J. Logic Programming **295** (1991) 344

[9] Gelder, A.V.: The alternating fixpoint of logic programs with negation. In: PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, New York, NY, USA, ACM (1989) 1–10

[10] Ullman, J.D.: Bottom-up beats top-down for datalog. In: PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, New York, NY, USA, ACM (1989) 140–149

[11] Ullman, J.: Principles of Database and Knowledgebase Systems Vol.1. WH Freeman & Co. New York, NY, USA (1989)

[12] Przymusinski, T.: On the declarative semantics of stratified deductive databases and logic programs. Foundations of Deductive Databases and Logic Programming (1987) 193–216

[13] VAN GELDER, A., ROSS, K., SCHLIPF, J.: The Well-Founded Semantics for General Logic Programs. Journal oi the AssocM1on for Computmg Machinery **38**(3) (1991) 620–650