IRIS - Integrated Rule Inference System - API and User Guide

February 28, 2008

Contents

1 Introduction

1.1 Purpose

This document 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

The creation of logic programs and their evaluation is described. The logic program can be created in one of two ways: Using the API to construct a program fragment by fragment, or using the parser to process a logic program written in human-readable format (datalog).

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 What it does

IRIS is a datalog reasoner that can evaluate safe-datalog with stratified negation as failure.

It is delivered in two java 'jar' files. One of which contains the reasoner and the other contains the parser.

2.2 What is input

IRIS evaluates queries over a knowledge base. The knowledge base consists of facts (instances of predicates) 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 logic program is achieved in one of two ways: Create the java objects representing the components of the program using the API (described in section 6.1 on page 9) or 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 6.2 on page 10).

2.3 What is output

For each query in the datalog program, 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 Evaluation Process

3.1 Supported Strategies

A number of evaluation strategies are supported and it is intended that more strategies will be created over time.

3.1.1 Naive

'Naive' evaluation is a bottom up approach where all the known facts are applied in turn to each of the rules and thus new facts are inferred. The evaluation process continues iteratively until no new facts are generated. In this way the fixed point for the knowledge base is computed before searching for the variable substitutions for the query.

3.1.2 Semi-Naive

The 'Semi-Naive' algorithm is similar to the 'Naive' strategy. However, in every iteration an attempt is made to make better use of the tuples generated in the previous step.

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 literals labeled l_{1-N} is evaluated N times substituting the incremental relations from the previous step for each each literal in turn.

3.1.3 Semi-naive with Magic Sets

The 'Magic-Sets' strategy first transforms the rules according to the variable bindings of the query so that the computation of the fixed point is restricted as

much as possible to those tuples that 'might' produce something that satisfies the query. (see 'On the Power of Magic'[1] by Beeri and Ramakrishan). After the transformation the 'Semi-Naive' evaluation is performed.

4 What can go wrong

4.1 Exceptions

A number of problems can occur that can halt the evaluation of a logic program. 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 program is not stratified (see 4.2 on page 5).

RuleUnsafeException indicates, that an unsafe rule was detected (see 4.4 on page 7).

4.2 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)
```

[1] http://portal.acm.org/citation.cfm?id=28659.28689

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 each of its positive literals and at least one higher than 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, it must be stratified.

4.3 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.

4.4 Unsafe rules

The algorithm for detecting unsafe rules is taken from 'Principles of Database and Knowledgebase Systems', Ullman, page 105. A rule is safe if all the variables occurring in the head and body are limited. Limited means, that the variable appears at least once in a positive ordinary predicate, is equated with a constant in a positive equality predicate, is equated with another variable known to be limited or occurs in an arithmetic predicate where the other two variables are limited. A more precise explanation of safeness can be found on the homepage[2]

5 Datatypes and Built-in Predicates

5.1 Supported datatypes

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

These datatypes are also discussed in the appendix A.2 on page 12.

5.2 Built-in Predicates

The built-in predicates defined in the WSML specification[4] are supported plus some IS_<type>() predicates used to confirm the type of a variable.

The complete list of built-in predicates is given in the appendix A.3 on page 12).

Additionally, user-defined built-in predicates can be created (see 5.6 on page 8).

5.3 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.

5.4 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

[2] http://www.iris-reasoner.org/saferules
[3] http://www.wsmo.org/TR/d16/d16.1/v0.21/#sec:wsml-builtin-datatypes
[4] http://www.wsmo.org/TR/d16/d16.1/v0.21/#sec:wsml-built-ins

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).
```

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 '$X \geq Y$'

5.5 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*.

5.6 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)
- Register the built-in with the BuiltinRegister class

5.6.1 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. provide a static getBuiltinPredicate() method which returns the predicate object describing your built-in (with attributes 'name' and 'arity')

Note: The BuiltinHelper class has some utility methods that might be useful. The javadoc for this class has more details. For an example, see FahrenheitToCelsiusBuiltin.java in test/org.deri.iris.builtins.

5.6.2 Register the built-in

There are two ways to register a built-in:

- automatically by adding the class name to the 'builtins.load' load file
- explicitly by calling the BuiltinRegister.registerBuiltin() method

To register the built-in automatically, the fully qualified name of the custom built-in java class must be added to a file called 'builtins.load'. This file must be accessible through the classpath and IRIS will use the first occurrence of a file with this name, ignoring any others. Only one class name can occur on each line in the file.

To register the built-in explicitly, call BuiltinRegister.registerBuiltin() passing the built-in class's class object as the parameter, e.g.

```
<IProgram object>.getBuiltinRegister()
    .registerBuiltin(FahrenheitToCelsiusBuiltin.class);
```

The built-in should now be recognised when you evaluate a logic program. Note that the built-in must be explicitly registered for every program instance.

6 API guide

6.1 Creating objects with the Java API

Most of the objects created in IRIS are created with factories. There's a factory for every purpose. 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 5.1 on page 7).

org.deri.iris.api.factory.IBuiltinsFactory creates built-in atoms provided by IRIS (see section 3 on page 15).

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[5].

6.2 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.

6.3 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.

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

6.3.1 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

7 Plug in External Datasources

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. [6]

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).`'

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

queries are 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 or variables.

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 13.

A.3 Built-in predicates

A built-in can be written instead of a literal and IRIS will do its best to evaluate it. All supported built-in predicates are described in table 3 at page 15.

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)`'.

[6] 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