IRIS - Integrated Rule Inference System - API and User Guide

Richard Pöttler

November 8, 2007

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 8) 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 8).

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 minimal 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 except that in every iteration only the new tuples generated in the previous step for each relation are used instead of the whole relation for this predicate.

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 as restricted as much as possible (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.

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

ProgramNotStratifiedException indicates, that the program is not stratified (see 4.2 on page 5).

RuleUnsafeException indicates, that an unsafe rule was detected (see 4.3 on page 5).

4.2 Stratified negation

The logic program must be stratified. This is a property of a logic program such that for any rule with a negated predicate, it is possible to first evaluate all possible tuples for the negated predicate. In this way it is possible to show that logic program evaluation is monotone (increasing). In other words, any tuples for any predicate generated during one iteration, will never be removed again in a later iteration.

4.3 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 9.

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 9).

Additionally, user-defined built-in predicates can be created (see 5.6 on page 7).

[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

5.3 Behaviour of built-ins with incompatible datatypes

The built-in predicates will evaluate to false if the operands are incompatible with the predicate, e.g. multiplying two dates, or 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 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, $notX < Y$ 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 only 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 only 3 things you must to implement:

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 the use 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 object.

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 5).

org.deri.iris.api.factory.IBuiltinsFactory creates built-in atoms provided by IRIS (see section 3 on page 12).

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, you could also use the
`org.deri.iris.compiler.Paser` to parse a datalog program. The grammar used by the parser is described in the grammar guide.

6.3 Evaluating a program

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

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 10.

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 12.

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

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