IRIS - Integrated Rule Inference System

Barry Bishop and Florian Fischer

Semantic Technology Institute (STI) Innsbruck,
University of Innsbruck, Austria
firstname.lastname@sti2.at

Abstract. Ontologies, fundamental to the realization of the Semantic Web, provide a formal and precise conceptualization of a specific domain that can be used to describe resources on the Web. Reasoning over such resource descriptions is essential in order to facilitate automated processing using formal descriptions that are machine interpretable. In this context, Datalog (with extensions) can be used for rule-based reasoning with ontologies described with WSML, (a subset of) OWL-DL, RDF, RDFS and extensional RDFS. Furthermore, since Java is the chosen implementation platform for the majority of software prototypes from research, it becomes clear, that a good quality, open-source, Java-based Datalog reasoner is a prerequisite for much research in the field of semantics. The purpose of this paper is to present a reasoner that fills this gap. IRIS is an open-source Datalog engine, extended with XML Schema data types, built-in predicates, function symbols and Well-founded default negation. We outline the reasoner architecture, basic evaluation algorithms and various optimizations. Additionally we provide a comparison of the performance of IRIS with similar systems.

1 Introduction

Ontologies[1] enable the reuse, sharing and portability of knowledge, coupled with a better conceptual understanding and analysis of a certain knowledge domain. Using a well defined formal language for the specification of ontologies does not only enable machine readability of knowledge, but crucially also machine interpretability and in turn automated processing. Properly employed, ontologies thus enhance the current Web with the possibility of automated reasoning about distributed knowledge, which makes it possible to derive new, and only implicitly available, knowledge. This vision finally leads to a Semantic Web, in which content has a well defined meaning.

An important observation in this regard is that resources on the Web are likely to be annotated with relatively lightweight ontologies (low number of concepts), however the number of resources annotated with these ontologies is likely to be very large (a large instance set)[2]. Reasoning with such large data sets is a well researched field in the context of deductive databases and a wealth of formal results have grown out of these Logic Programming[3] based efforts. Furthermore, it is possible to identify reasoning tasks at two different levels, namely the schema level and instance level.

On the schema level (intensional reasoning) we can reason about class properties and subclass relationships, i.e. subsumption reasoning. This task consists of checking if a certain class is more general than another one (subsumes it). By performing this for the complete knowledge base it thus becomes possible to compute a complete class hierarchy and implicit relationships with other classes become apparent. Subsumption reasoning can be reduced to satisfiability checking and Description Logic (DL) based reasoners (such as RacerPro[4], FaCT++[5], Pellet[6] and Kaon2[7]) usually implement efficient algorithms to perform this task. While subsumption reasoning can be reduced to query answering by a Logic Programming engine, DL reasoners are generally more efficient in this regard.

The second reasoning task is query answering in regard to instances in the knowledge base (extensional reasoning). Query answering can be further subdivided into instance checking (a ground query) and instance retrieval. Instance checking involves a ground fact, and the corresponding task is to check if this fact is entailed by the current knowledge base. Instance retrieval (an open query) is focused on a formula with free variables and its purpose is to give substitutions for these free variables with values from the knowledge base. As a basic naive approach, instance retrieval can be reduced to instance checking by grounding the free variables in the open query with values from the knowledge base. Thus one open query can be answered by computing several ground queries. Logic programming based techniques are very efficient and well studied in regard to query answering.

One Logic Programming based formalism that has been thoroughly analyzed is Datalog[8], 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[9].

Datalog can be used in a wide variety of applications, including Description Logic Programming (DLP)[10], rule languages from the WSML family[11] and RDF[12] reasoning. Disjunctive Datalog, which allows disjunctions in the head of a rule and is more expressive than standard Logic Programming, can be used to reason with an even larger subset of OWL DL[13].

The purpose of this paper is to present IRIS, an open-source Datalog engine, extended with XML Schema data types, built-in predicates, function symbols and Well-founded default negation. It is licensed under the GNU lesser GPL and is therefore free to use and modify by the research community and industry alike. The IRIS project is hosted by Sourceforge[1] and more detailed information is available on its home page[2].

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

The WSML2Reasoner[3] framework uses IRIS for ontology reasoning for rule based WSML variants (WSML-Core, WSML-Flight and WSML-Rule) and there exists an RDFS reasoner[4] that uses IRIS for RDF, RDFS and extensional RDFS reasoning[14].

The rest of this paper is structured as follows: Section 2 provides a brief summary of features. Section 3 describes the internal design, how different components of the reasoner interact, in what ways optimization techniques are applied and how new features can be added in a non-obtrusive way. Section 4 outlines related work and identifies other reasoners that employ similar techniques and formalisms. In order to demonstrate the practical use of IRIS, a performance evaluation in the form of a comparison with other well-known reasoners is given in Section 5. Finally, plans for future development are outlined in Section 6

2 System Overview

IRIS is a Datalog reasoner that uses bottom-up[8] evaluation strategies with several optimizations. However, support for rule-based WSML variants requires several extensions to Datalog, namely:

- WSML-Core is based on plain (function-free and negation-free) Datalog with primitive XML schema types.
- WSML-Flight requires Datalog extended with inequality and locally stratified[15] default negation.
- WSML-Rule further requires the unrestricted use of function symbols, Well-Founded default negation and does not require the rule safety condition (unsafe rules).

IRIS has been designed to be as modular as possible thus allowing more evaluation strategies to be added over time. However, for the initial releases of IRIS, it was decided to concentrate on bottom-up evaluation techniques. The advantages of using bottom-up techniques are 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'[16] 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[17][18], easy to implement, fast and efficient.

[3] http://tools.deri.org/wsml2reasoner/
[4] http://tools.deri.org/rdfs-reasoner/

3 Design and System Architecture

The IRIS Datalog reasoner is highly modular and comprised of a number of loosely coupled components that implement well-defined Java interfaces. The overall strategy is to focus on fast bottom-up evaluation techniques and optimize query answering using magic sets. 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.

Broadly speaking, an evaluation strategy represents a particular combination of processing elements. There are two basic evaluation strategies currently implemented, see Figure 1.

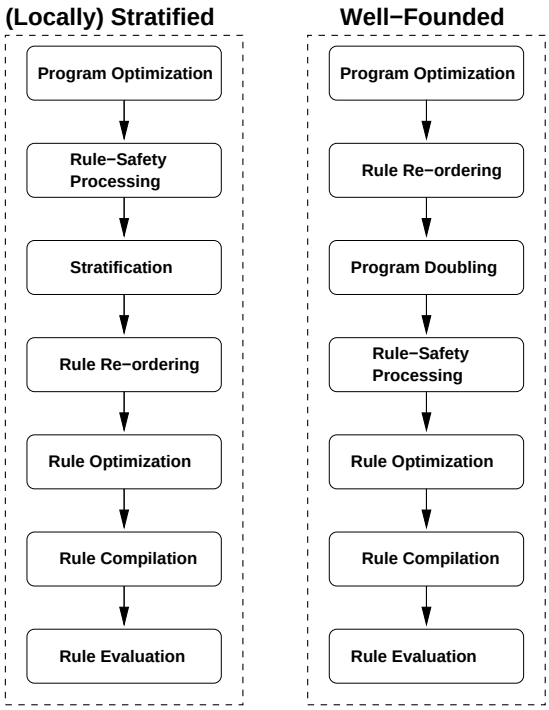

Fig. 1. Stratified and Well-founded evaluation strategies

The first is a (locally) stratified[8] 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[19] 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.

3.1 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[20]. 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.

3.2 Rule Safety Processing

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. Unsafe rules are therefore particularly problematic for bottom-up evaluation techniques that do precisely this, i.e. substitute known values into variables of rule body predicates.

When IRIS is configured not to allow unsafe rules, the standard rule-safety processor 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 results in a specific exception being thrown containing a message explaining which rule is unsafe and which variables are problematic.

In order to process unsafe rules IRIS can be configured to use a rule augmentation processor. This processor uses a technique suggested by Gelder[21] 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.

3.3 Stratification Algorithms

A globally stratified logic program is one where the rules can be arranged into 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[8]. If the program is not globally stratified the algorithm fails. The second algorithm assumes the program is locally stratified[15]. 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[22]. IRIS uses a novel technique that scans the rule bodies looking for negated predicates containing constants. If any are found then the rules are examined to discover which rule heads can match to this negated predicate. This is done by examining each rule's body to discover what tuples can be produced and adorning the rule-head with information about the range of term values produced at each position in the rule output tuple. If any rules can produce both tuples that match and tuples that do not match the original negated predicate, then the rule is 'split' into two separate rules, one that perfectly matches and one that does not match. This process is continued until no more rule-splitting can be done. After this, the normal stratification algorithm is applied, but rule head adornments are also used to indicate predicate dependencies. In this way, IRIS is able to do fast bottom-up evaluation of locally stratified logic programs, without requiring the well-founded semantics.

3.4 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.

3.5 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 be easily 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 into 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 into 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.

3.6 Rule Compilers

Compiling an input rule simply involves pre-computing all possible information required for rule evaluation. The input rule is transformed into 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 into 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 into 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.

3.7 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[8], 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. In each round of evaluation it uses the deltas, i.e. the set of new facts from the previous round, to substitute into each rule once for each positive ordinary literal.

3.8 Miscellaneous Components

The following utility components are common to all evaluation strategies.

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 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 into 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.

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 into 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.

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

4 Related Work

It is possible to make a comparison with other ontology reasoners when IRIS is used with the WSML2Reasoner adaptor for rule-based reasoning. However, ontology reasoners are mostly based on OWL(DL) and therefore any comparison will invariably favor one or the other approach.

So while DL reasoners should be mentioned as related work in the area of ontology reasoning, it makes more sense to compare IRIS with other Datalog engines.

DLV [23] is a Logic Programming based system computing answer sets according to the stable model semantics[24]. DLV is a Disjunctive Datalog engine, with support for safe rules, but without function symbols. Among other features, DLV supports several comparative and arithmetic built-ins, aggregate functions and a SQL front-end.

MINS [5] is a Datalog reasoner that supports function symbols and negation using the Well-Founded Semantics. The acronym MINS stands for 'Mins Is Not Silri', because it is based on the SILRI[25] inference engine by Stephan Decker and Jürgen Angele. It is no longer supported.

XSB [26] is a Logic Programming and deductive database system based on Prolog and more particularly on WAM[27]. However, it goes beyond Prolog by introducing several features such as different kinds of negation (stratified negation, negation under the well-founded semantics), bottom-up extensions and SLG resolution[28]. The syntactic basis of XSB is HiLog[29], which allows a great deal of flexibility in regard to (meta)modeling. XSB also has a range of built-in predicates and data types. It is openly distributed under the GNU lesser GPL license.

5 Evaluation

It makes little sense to try and compare reasoners founded on different knowledge representation paradigms, i.e. Logic Programming versus Description Logic. For this reason the evaluation results are limited to a comparison of similar Logic Programming/Datalog based systems, even though they might use different evaluation methods. The situation is further complicated by the lack of widely accepted benchmarks for Datalog.

5.1 Methodology

It was decided to use DLV and XSB for comparison, because they have similar usage semantics to IRIS. In order to make a comparison, a set of specimen logic programs were chosen that are known to be computationally expensive, in that a large number of intermediate tuples must be computed in a series of cartesian

[5] http://tools.deri.org/mins/

products. The intention is to test the basic join operation, fundamental to evaluating Datalog. Each program is identical, apart from the number of starting tuples in the relation for 'p'. The reasoners use a slightly different Datalog dialect, so separate programs were generated for each one. Shown below is the '11 starting tuples' version for IRIS:

```
p('abcd0').p('abcd1').p('abcd2').p('abcd3').p('abcd4').
p('abcd5').p('abcd6').p('abcd7').p('abcd8').p('abcd9').
p('abcd10').

ra(?A,?B,?C,?D,?E) :- p(?A),p(?B),p(?C),p(?D),p(?E).
rb(?A,?B,?C,?D,?E) :- p(?A),p(?B),p(?C),p(?D),p(?E).

r(?A,?B,?C,?D,?E) :- ra(?A,?B,?C,?D,?E),rb(?A,?B,?C,?D,?E).

q(?A) :- r(?A,?B,?C,?D,?E).
q(?B) :- r(?A,?B,?C,?D,?E).
q(?C) :- r(?A,?B,?C,?D,?E).
q(?D) :- r(?A,?B,?C,?D,?E).
q(?E) :- r(?A,?B,?C,?D,?E).

?- q(?X).
```

The comparison was conducted on a 32-bit Windows machine with a dual-core, 2.67GHz Intel processor. Timings were measured using the cygwin 'time' command. This set-up was not intended to generate rigorous results, rather simply to give a quick impression of performance characteristics.

5.2 Results

Four version of the specimen program were created, using 11, 15, 17 and 19 starting tuples, which should in turn involve creating 161,051, 759,375, 1,419,857 and 2,476,099 tuples for each of the relations associated with predicates 'ra', 'rb' and 'r'.

The performance results are shown in the following table and graph. All timings are in seconds.

Tuples	XSB	DLV	IRIS
11	14.799	6.449	4.267
15	71.699	31.68	21.135
17	136.017	62.036	40.28
19	237.453	107.468	n/a

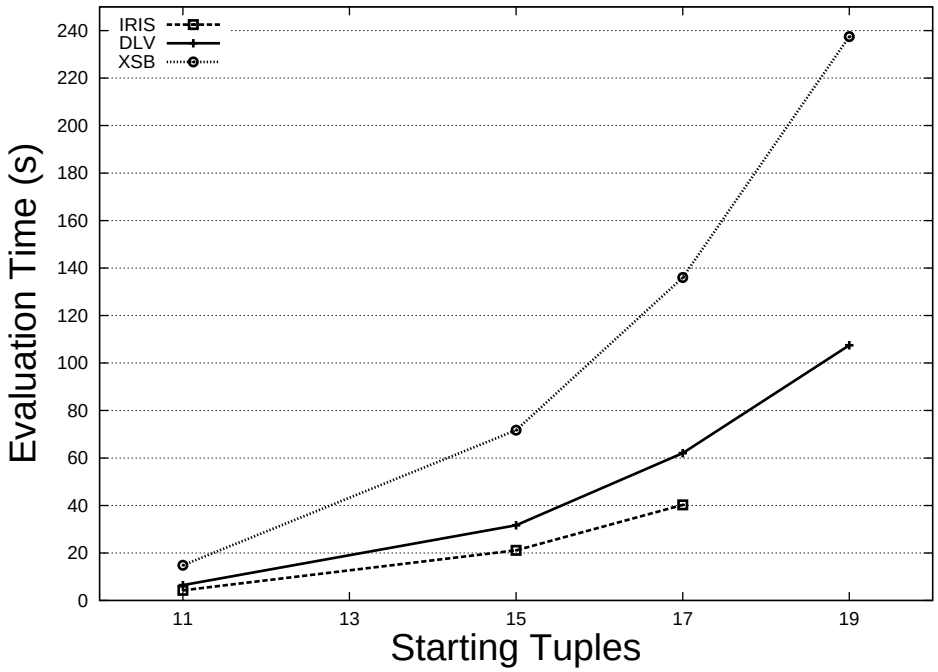

Fig. 2. Performance Comparison

Set up times were not measured, because with more complex problems the setup/load times become insignificant. As can be seen, IRIS was faster in all tests. However, IRIS was unable to evaluate the '19 starting tuples' program due to an out of memory error. In any case, it was observed that IRIS generally uses more memory during evaluation and this has been identified as an area for improvement.

6 Conclusion

This paper has attempted to elucidate the motivation for creating a free, open-source, Java based Datalog reasoner. The features that extend IRIS beyond a simple Datalog reasoner have been described, along with the internal structure and design goals. As can be seen from Section 5, the functionality and performance of IRIS compare favorably with similar systems. IRIS is currently used in the implementation of WSML2Reasoner[6] and an RDFS reasoner[7].

In the future, IRIS will be extended in several directions:

- The flexibility and configurability of the reasoner will be improved. It is envisaged that IRIS will be used in various situations that each require unique reasoner properties. For example, in some situations, fast initialization times over longer query times may be preferred. Other situations may require fast query times in preference to slower initialization times. Yet further environments may require the ability to modify the extensional database (set of starting ground facts) ad-hoc and have the intensional database (set of inferred facts) update in real time.
- The Stable Model semantics[24] can potentially reveal more information from an unstratified knowledge-base than the Well-founded semantics[21], but at the expense of a more computationally intensive evaluation process.
- At least one top-down evaluation strategy will be implemented. This will be useful when dealing with knowledge-bases with a theoretically infinite minimal model, such as can occur when using function symbols and arithmetic built-in predicates. Goals can still be proved using a top-down approach, whereas bottom-up techniques will simply fail.
- Identify a more comprehensive benchmarking strategy and derive more results.
- Research better data structures in order to improve memory usage.
- The standardization work of the Rule Interchange Format (RIF) working group[8] will lead to a common format for exchanging rules between systems. It is planned for IRIS to be one of the first systems to implement the RIF Basic Logic Dialect[9].

7 Acknowledgements

The development of IRIS has in part been funded through the European Union's 6th Framework Program, within Information Society Technologies (IST) priority under the SUPER project (FP6-026850, http://www.ip-super.org).

[6] http://tools.deri.org/wsml2reasoner/
[7] http://tools.deri.org/rdfs-reasoner/
[8] http://www.w3.org/2005/rules/wg
[9] http://www.w3.org/TR/rif-bld/

References

1. Gruber, T.: Toward principles for the design of ontologies used for knowledge sharing. International Journal of Human-Computer Studies **43**(5/6) (1995) 907–928
2. Weithoner, T., Liebig, T., Luther, M., Bohm, S.: Whats Wrong with OWL Benchmarks? (Second International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006))
3. Lloyd, J.: Foundations of logic programming. Springer-Verlag New York, Inc. New York, NY, USA (1987)
4. Haarslev, V., Moller, R.: RACER system description. Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001) **2083** (2001) 701–705
5. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006) **4130** (2006) 292–297
6. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web **5**(2) (2007) 51–53
7. Motik, B., Studer, R.: KAON2–A Scalable Reasoning Tool for the Semantic Web. Proceedings of the 2nd European Semantic Web Conference (ESWC 05), Heraklion, Greece (2005)
8. Ullman, J.: Principles of Database Systems. WH Freeman & Co. New York, NY, USA (1983)
9. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems (TODS) **22**(3) (1997) 364–418
10. 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
11. 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)
12. Lassila, O., Swick, R., et al.: Resource Description Framework (RDF) Model and Syntax Specification. (1999)
13. Hustadt, U., Motik, B., Sattler, U.: Reducing SHIQ- Description Logic to Disjunctive Datalog Programs. Proc. of the 9th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2004) (2004) 152–162
14. de Bruijn, J., Heymans, S.: (Logical Foundations of (e) RDF (S): Complexity and Reasoning?)
15. Przymusinski, T.: On the declarative semantics of stratified deductive databases and logic programs. Foundations of Deductive Databases and Logic Programming (1987) 193–216
16. Beeri, C., Ramakrishnan, R.: On the power of magic. ACM Press New York, NY, USA (1987)
17. 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
18. Chen, Y.: Magic Sets and Stratified Databases. J. Logic Programming **295** (1991) 344
19. 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

20. 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
21. 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
22. Palopoli, L.: Testing logic programs for local stratification. Theoretical Computer Science **103**(2) (1992) 205–234
23. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3) (2006) 499–562
24. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. Proceedings of the 5th International Conference on Logic Programming (1988) 1070–1080
25. Decker, S., Brickley, D., Saarela, J., Angele, J.: A query and inference service for RDF. QL98-The Query Languages Workshop (1998)
26. Sagonas, K., Swift, T., Warren, D.: XSB as an efficient deductive database engine. ACM SIGMOD Record **23**(2) (1994) 442–453
27. Warren, D., Science, C., Division, T., Center, A.: An Abstract Prolog Instruction Set. SRI International (1983)
28. Chen, W., Warren, D.: Tabled evaluation with delaying for general logic programs. Journal of the ACM (JACM) **43**(1) (1996) 20–74
29. Chen, W., Kifer, M., Warren, D.: HILOG: a foundation for higher-order logic programming. Journal of Logic Programming **15**(3) (1993) 187–230