Advanced Knowledge Base Debugging for Rulelog*

Carl Andersen**, Brett Benyo**, Miguel Calejo***, Mike Dean**, Paul Fodor†, Benjamin N. Grosof‡, Michael Kifer†, Senlin Liang†, and Terrance Swift§

Abstract. We present a novel approach to debugging expressively rich knowledge representation and reasoning (KRR) logic Rulelog. Rulelog is an extended form of declarative logic programs (LP) under the well-founded semantics, which allows higher-order logic formulas as axioms in combination with defeasibility mechanisms that include rule cancellation and priorities, along with default and explicit negation. Rulelog also supports strong knowledge interchange with all current major semantic web standards for logical KRR. Rulelog has been implemented in Flora-2 and Silk, both on top of XSB; and (less completely) in Cyc. The debugging approach described here is part of an integrated development environment, most fully implemented in Silk. The approach includes: reasoning trace analysis, based on tabled LP inferencing tables and forestlog; and justification graphs, which treat why-not and defeasibility as well as provenance. The reasoning trace analysis treats performance and runaway computations, including non-termination as well as classic subgoal-ordering issues that arise in database query optimization. Non-termination can be prevented entirely by leveraging the restraint (bounded rationality) feature of Rulelog. Revision/authoring of knowledge is interactive, based on a rapid edit-test-inspect loop and incremental truth maintenance.

1 Introduction

1.1 Rulelog

Rulelog is an expressively rich knowledge representation and reasoning (KRR) logic, based on a unique set of features that include:

1. defeasibility, based on argumentation theories (AT's) [21], i.e., *AT-defeasibility*. These theories provide features such as rule cancellation and priorities, along with default and explicit negation.

* The order of the authors is alphabetical. Copyright © 2013 by the authors.
** Raytheon BBN Technologies, USA
*** Declarativa, Portugal
† Stony Brook University, USA
‡ Benjamin Grosof & Associates, LLC, USA
§ CENTRIA, Universidade Nova de Lisboa, Portugal.

2. higher-order syntax, based on HiLog [1], and other meta-knowledge enabled by rule id's, i.e., *hidlog*;
3. classical-logic formula syntax, including existential as well as universal quantifiers, i.e., *omniformity* ([5] gives a compressed description); and
4. bounded rationality, based on *restraint* ([7] gives the basic *radial* form) which utilizes the undefined truth value of the well-founded semantics to represent "not bothering."

Omniformity together with HiLog allows higher-order logic (HOL) formulas as axioms. The omniformity feature also includes and extends the Lloyd-Topor transformation [12] on rule bodies. Omniform rules are called "omni rules" or "omnis", for short. The hidlog feature also includes reification, i.e., a formula can be treated as a term. The rule id's aspect of hidlog enables meta-info about axioms to be specified easily within the KB itself, e.g., meta-info about prioritization and about provenance. Other features include: object-based knowledge modeling (frame syntax), and aggregates (e.g., setof, sum, average, etc.).

Rulelog is the logic that was used in the Silk system [17] developed as part of Vulcan's Project Halo [8] advanced research effort, and grows out of earlier work on RuleML [15] and Semantic Web Services Framework [20]. A W3C Rule Interchange Format (RIF) dialect based on Rulelog is in draft [9], in cooperation also with RuleML.

The semantics of Rulelog is specified transformationally, into logic programs (LP) that are *normal*: those with logical functions and with default negation under the well-founded semantics. Using these transformations, Rulelog has been implemented most fully to date in Silk, which is architected as a Java layer that sits on top of Flora-2 [4]. Flora-2 sits in turn on top of XSB [22,19], which implements normal LP. Rulelog also has been implemented, less completely, in Cyc [2]. Both XSB and Flora-2 are available open source; Silk (i.e., the Java layer), purposed primarily as a scientific research effort, is proprietary.[1]

Rulelog supports strong semantic knowledge interchange with not only LP but also with first-order logic (FOL), and thus with all current major semantic technology standards for logical KRR, including RDF(S), SPARQL, SQL, XQuery, OWL-RL, OWL-DL, RIF-Core, and RIF-BLD, as well as with ISO Common Logic and thus SBVR.

Rulelog provides a good target for text-based authoring of knowledge [5], because of its ability to express defeasible HOL formulas as axioms.

Rulelog has been application-piloted in the domain of college-level biology for the task of question-answering in e-learning, in Project Halo. However, Rulelog is applicable to many other domains and tasks, e.g., that involve policies, contracts, law, and/or information integration.

[1] The Silk development effort, including maintenance, ended in April 2013.

1.2 Challenge of Debugging Knowledge in a Rulelog System

The expressivity of Rulelog raises a number of issues both in debugging and in understanding the behavior of Rulelog derivations.

The *justification problem* is a problem of explaining missing or unexpected (e.g., wrong or unintended) answers. This task is complicated not only by the types of inference used, but also by the transformations used to implement Rulelog reasoning. Answers to a query may be different than expected due to defeasibility or due to unexpected inferences made by the use of the higher-order reasoning provided by the HiLog component.

The *performance/termination problem* is a problem of indicating why a derivation has taken up more resources than expected — including non-termination as an extreme case. To explain the context of this problem, one of the major objectives of the Silk implementation of Rulelog was to be usable by knowledge engineers (KE's) who are competent in logic, but who are not necessarily computer programmers. Such usage can give rise to knowledge bases constructed in a declarative manner, but with little attention to procedural aspects. Queries to such knowledge bases may lead to derivations that take longer than expected. In addition, as mentioned earlier, Rulelog uses logical functions both explicitly and implicitly (the later due to existential quantification, which is part of omniformity), and this use of logical functions can lead to non-termination.[2] While some performance issues can be addressed by optimizing compilers, users still need to understand what parts of a knowledge base give rise to poor performance or non-termination, so that these parts can be remedied.

Understanding Rulelog derivations is complicated by the semantics of Rulelog, which unlike first-order logic, is a fixed-point logic that supports recursive definitions. A Rulelog derivation, therefore, can be seen as a sequence of evaluations of recursive components in which the answers to a given subquery may be mutually dependent on answers to numerous other subqueries. Such a derivation can be partially modeled via a graph whose vertices are Rulelog atoms and whose edges are direct dependencies of the truth of one Rulelog atom on another. As will be shown later, such dependencies are implicit in our solution to the justification problem, but are explicit in our solutions to the performance/termination problem.

To partially address the justification and performance/termination problems, support is given by the tabled resolution of XSB, which serves as the computational underpinning of Rulelog in Silk. Although the details of tabling are quite complex, at a high level it handles recursive query evaluation by registering each tabled subgoal in a derivation. The first time a subgoal S is encountered in a derivation, a table is created for S and program clause resolution is used to derive answers for S, which are added to the table for S as they are derived. Subsequent calls to S need only resolve against answers in its table. In addition,

[2] FOL and normal LP also have this potential for non-termination in inferencing, for the same reason.

tabling keeps partial track of dependency information in order to determine the truth values of atoms in the 3-valued well-founded semantics. Although tables are central to the derivation strategy of Silk, they can also be examined by users to help understand features of a derivation.

A basic requirement in debugging is that the edit-test-inspect loop be rapid. This is addressed in Silk (and XSB and Flora-2) by the use of incremental methods for tabling in XSB and Flora-2. Such incremental tabling essentially constitutes truth maintenance.

The considerations so far indicate that a creative approach must be taken to understanding correctness, performance, and termination. Note that because of the complications of the transformations from Rulelog to normal logic programs, together with the technical details of tabled resolution, an interactive-debugger approach like that used in Prolog and other languages is impractical. Instead, we have developed a number of novel tools, each of which has an analytic component, which examines the internal structures of the engine and produces textual output, and a presentation component that makes the textual output more comprehensible to the user. The presentation components were incorporated into an overall graphical integrated development environment (IDE) for Silk, based on Eclipse, called Silklipse [6]. All of the tools described below have either been completed or are in the advanced stages of development.

2 Justification

Explanation of inferencing results, often called *justification*, has a long history in KRR, starting with the venerable *truth maintenance systems* [13]. The most practical previous approach to justification in LP is the method proposed for XSB's tabled computations in [14]. Silk takes the previous ideas much further in several ways. First, it provides an attractive and easy-to-use visualization of the justification process through its Silklipse environment ([6] described an early version). Second, unlike XSB and other logic systems with explanation mechanisms, Silk supports defeasible reasoning through argumentation theories [21]. In the presence of defeasibility, a fact might be false or undefined because it is derived by the rules that are *defeated* by other rules. In those cases, it is necessary to explain how and why those rules were defeated. Silk provides such explanations. A key aspect is to explain why literals or rules have *false* (in the sense of NAF) truth value, i.e., *why-not*. Another key aspect is to explain how prioritization, or its lack, is involved. Third, unlike [14], justification is done not by transforming the original rules and blowing up the size of the knowledge base but through a separate small set of meta-rules, which is invoked on-demand when the user requests justification. Fourth, Silk supports rule-based transformation of the justification information: displaying it via automatically generated English text, and/or summarizing or otherwise reorganizing it.

Figure 1 shows a screenshot of a navigable justification in the Silk GUI. Some lines have been transformed into English text, while others have not been and

thus appear directly in Silk's main logical syntax. E.g., the first line has been transformed into English text: "It is not the case that cell52 has a nucleus." But lines 4 and 13 (among others) appear in the Silk logical syntax:

cell52 # red(blood(cell)))
red(blood(cell))##eukaryotic(cell)

Here "#" means "is an instance of" and "##" means "is a subclass of." Next we explain the icons that appear on the left in each line. "G" indicates a (sub)goal literal. "A" indicates an argument, i.e., a rule body supporting such a goal literal. Here, "argument" is in the sense of prioritized argumentation in defeasibility. Black bar ("—") indicates a neg-argument, i.e., an argument for the neg (strong negation) of the goal literal. "F" indicates a fact, i.e., a literal that was directly asserted. "P" indicates prioritization info, i.e., that one rule's tag has higher prioritization than another tag. Green indicates true, while red indicates false (in the naf sense). Green bang ("!") indicates a undefeated ("live") argument. Red down arrow ("↓") indicates an argument that has been refuted, i.e., defeated by another conflicting argument that has higher priority. Plus ("+") just to the right of "G" indicates that there are more arguments to see. When the "+" is black it indicates there are both pro (i.e., positive/for) and con (i.e., strong-negative/against) arguments to see; when green, it indicates there are more pro arguments but not more con arguments to see.

In this example, the relevant asserted logical rules in the KB can be described in English as follows:

cell52 is a red blood cell.
Eukaryotic cells have nuclei. (This rule has tag r1.)
Red blood cells are a subclass of eukaryotic cells.
Red blood cells do not have nuclei. (This rule has tag r2.)
r2 has higher priority than r1.

3 Trace-based Analysis

3.1 Table dump: Examining Subqueries, Answers, and Rules

Table dump is a tool that produces a report on the subgoals that are among the most heavily called and the subgoals that have the most answers. This tool also lets the user know the *rules* that are the most heavily called ones. It thus helps to identify the bottlenecks in the knowledge base and then take measures such as to add appropriate guards to rules and to reorder subgoals within rules.

Figure 2 shows a screenshot of a navigable view of table dump info in the Silk GUI.

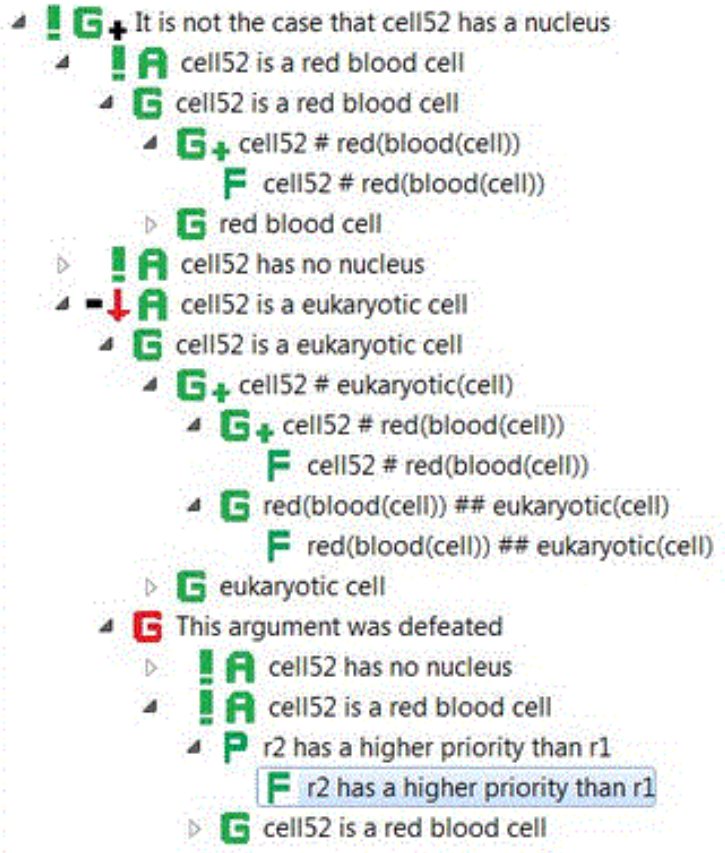

Fig. 1. Justification example

3.2 Forest logging

Although simple and powerful, the table dump approach lacks two main features needed to fully address the performance/termination problem. First, it does not provide an overview of how given subqueries in a derivation relate to one another through rules, and does not display information about the recursive components whose computation is central to a Rulelog derivation. Second, no information is provided about the order of events in a derivation, such as when subqueries were made, answers derived, and so on.

Within Silk, details of a Rulelog derivation can be reconstructed through another kind of trace-based analysis. XSB provides a mechanism to create a more dynamic trace or log of a derivation, called a *forest log* [18]. Using such a log, the structure of even very large recursive components can be analyzed, and

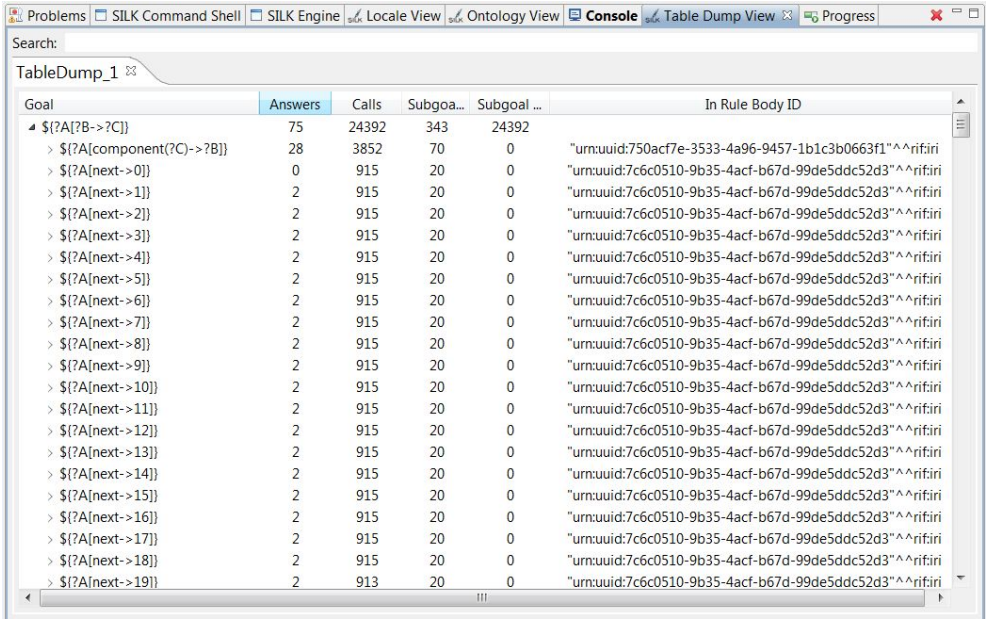

Fig. 2. Table dump example

non-terminating derivations detected. This subsection first overviews forest logs, and afterwards discusses the analysis routines based on the logs.

The form of tabling used by XSB is called SLG resolution. The operational semantics of SLG evaluation (and hence a Rulelog derivation) can be modeled as a sequence of forests of trees, where each tree corresponds to a tabled subquery S, and represents the immediate subqueries that S produces along with any answers to S. In fact, each SLG operation is modeled as a function from forests to forests that creates a new tree, or adds a node or label to an existing tree.

Within XSB, SLG resolution is executed using a byte-code virtual machine analogous to that used by Java. An internal XSB flag can be set so that any byte-code instruction that corresponds to a tabling operation will log information about itself and its operands as a Prolog-readable term. For instance, if (tabled) subgoal S_1 is called in the context of subgoal S_2, and it is the first time S_1 is called in an evaluation, a fact of the form

table_call(S_1,S_2,*new*,*ctr*)

is logged, where *ctr* (mnemonic for "counter") is a sequence number for the fact. When a derivation ends or is interrupted, the log can be loaded into XSB and analyzed as a set of Prolog facts. Within XSB, the logging system is written at a very low level for efficiency. Turning on full logging usually does not slow down Flora-2 performance by more than 70-80%. XSB also provides routines to load

logs and index their facts on various arguments. Based on the logging libraries, logs containing hundreds of millions of facts have been loaded and analyzed.

3.3 Analyzing Recursive Components

Once a log has been loaded, a user may ask for an overview of a computation, which provides information on the total number of calls to tabled subgoals, the number of distinct tabled subgoals, the number of answers, and so on. In addition, the overview provides aggregate information on the number of mutually recursive components, and the number of subgoals in the components. Finally, the overview contains information indicating how stratified the negation (negation-as-failure, i.e., naf) was in a derivation by listing the total number of atoms whose truth value was undefined, along with a count of the various SLG operations used to evaluate well-founded negation.

Some derivations may give rise to very large recursive components—due to an unanticipated effect of higher-orderness, a knowledge base that is not sufficiently modularized, or other reasons. The analysis routines allow given recursive components to be examined, by listing the subqueries in the component, along with the pairs of calling and called subgoals within the components.

By examining this output, users can usually fix whatever problems gave rise to large recursive components. However for a very large component $\mathcal{C}$, the number of subqueries in $\mathcal{C}$ may be on the order of 10^5 or more and the number of calling/caller pairs may be on the order of 10^6. In such a case diplaying every subquery or pair may be confusing at best. The analysis routines thus provide several abstraction routines that allow a user to coalesce similar atoms. For instance, if a component contained the subqueries *p(a,X)*, *p(b,X)*, *p(c,X)* ..., the analysis routines could use mode abstraction to coalesce all of these terms to *p(bound,free)*, or even predicate abstraction to coalesce all these terms to *p/2*. Recursive component analysis together with abstraction of atoms has been used to analyze the behavior of reasoning that was translated from Cycorp's inference engine into the Silk implementation of Rulelog, for example.

3.4 Analyzing Runaway: Terminyzer

Runaway computation occurs when a query does not terminate or takes too long to come back with an answer. The first type of problem occurs typically due to the presence of function symbols and the second is largely due to computations that produce very large intermediate results most of which could be avoided with smarter evaluation strategies, such as subgoal reordering. The problem of determining whether a query is terminating or not has long been known to be undecidable, and the known sufficiency tests for them are weak for practical purposes. Cost-based optimization of LP via subgoal reordering has not been well studied for the case when recursion and logical functions are present.

The first tools we have developed for runaway give the user the means to interrupt the computation and inspect various statistics and the table dump, as described earlier. The user can also request the computation to stop after producing the desired number of answers.

One sophisticated diagnostic tool we have developed to tackle the non-termination problem is called the *Terminyzer* (short for "(non-)Termination Analyzer") [11,10]. This tool relies on the previously described *forest logging* mechanism, which records the various tabling events that occur in the underlying inference engine XSB [19]. Among others, forest logging records when the different subgoals are called and when they receive answers. Terminyzer performs different kinds of analysis, such as *call-sequence analysis* and *answer-flow analysis*, and identifies the sequences of subgoals and rules that are being repeatedly called and in this way cause non-terminating computation.

Terminyzer also has a heuristic that may suggest the user to allow the system to reorder subgoals at run time and this avoid non-termination. For instance, in a composite subgoal `p(?X,?Y), q(?X)` , Terminyzer may detect that `p(?X,?Y)` is an infinite predicate. However, this infinity may be due to the infinite number of `?X`-values. If `q(?X)` binds `?X` to a concrete value first, non-termination will not occur. In such a case, Terminyzer may suggest the user to wrap the offending subgoal with a suitable delay quantifier—a novel facility supported by Flora-2 and Silk. For instance, if the above subgoal is rewritten as `wish(ground(?X))^p(?X,?Y), q(?X)`, the system will not try to evaluate `p(?X,?Y)` unless `?X` is bound. If it is not bound, the evaluation of `p(?X,?Y)` is postponed and `q(?X)` will be evaluated next. If this binds `?X` then all is well and `p(?X,?Y)` can be evaluated next without a runaway. If `?X` is still unbound, some other subgoal may, perhaps, bind it, so `p(?X,?Y)` remains delayed. Only when the system determines that `?X` cannot be bound no matter what, `p(?X,?Y)` is submitted for evaluation. If this happens, the user would have to use the information provided by Terminyzer to decide whether the runaway is a mistake or is semantically justified. In the first case, this information will help the user fix the mistake; in the second, restraint could be used to prevent the runaway.

The presentation component of Terminyzer is integrated with Silklipse.

4 Restraint: Bounded Rationality and Prevention of Runaway

Another advanced way to control runaways is to use *restraint*, an approach to bounded rationality (and pragmatic incompleteness) that is semantically sound despite non-monotonicity [7]. With restraint, the semantics of inferencing—and thus corresponding computation—is limited in well-defined way; answers derived after the limits have been reached are given the truth value of *undefined*.

While Terminyzer is used for finding *mistakes* in user's knowledge base, i.e., in situations when runaway computation is not intended, restraint is used when the knowledge base is *correct*. This typically occurs when the user query of

interest or one of its subqueries has an infinite number of answers, but only the first few need to be returned to the user.

One type of restraint is to limit a norm on subgoals, e.g., term size or depth, to be upper-bounded by a constant, which is called the *radius*. By setting the radius to a small enough value, radial restraint can be used to prevent runaways altogether.

There are several other useful types of restraint as well. In *skipping* restraint, conditions are specified (via rules) for when some other rules instances should be skipped, i.e., treated as having undefined truth value. In *unsafety* restraint: a literal that is (irremediably) unsafe with respect to NAF is treated as having undefined truth value. Likewise, an external-query (a.k.a. *sensor*) literal that is unsafe with respect to binding mode requirements is treated as having undefined truth value. In *unreturn* restraint, an external-query literal that does not return—e.g., due to network failure or server failure — is treated as having undefined truth value. In some situations, unsafety and unreturn restraints are preferable to throwing an error. Radial and skipping restraint are *voluntary* kinds of bounded rationality: the user specifies desired limits on reasoning via meta-rules knowledge. The limitation is cleanly semantic and specified as part of the knowledge base itself. By contrast, unsafety and unreturn are *involuntary*: limitations on reasoning are imposed by the circumstances of the inferencing mechanism and/or external environment.

All the above types of restraint straightforwardly combine with each other. They furthermore straightforwardly combine with the "anytime" approach to temporally bounded rationality [3,16]. In *anytime* restraint, a series of increasingly complete inferencing-result sets are computed and when a time limit is reached, the best one computed so far is returned. For instance a restraint radius is progressively incremented until the time limit is reached.

5 Overall Process of Knowledge Debugging

The tools we have described can be combined in a number of ways. The typical process of knowledge debugging goes as follows. A user runs a (test) query of interest. If the execution of the query does not take an unexpectedly/undesirably large amount of time or space, there is no performance/termination issue. The user looks at the answers to the query, and employs the justification tools to examine the explanation of those answers in terms of supporting conclusions and their associated assertions (rules knowledge). Along the way, the user looks for wrong or missing conclusions, and wrong or missing rules. The user may issue some other related queries as part of this investigation, and look at their explanations as well.

However, if execution of the query does take an unexpectedly/undesirably large amount of time or space, there is a performance/termination issue. At this point, the user needs to determine whether the runaway is due to non-termination or merely due to an inefficient computation. The first step in de-

termining the culprit is to look at the table dump of the trace. If these show very large terms with deeply nested repeated function symbols, non-termination is the likely problem, and Terminyzer can be further employed to find the actual rule sequences that cause the problem. Otherwise, the user would use the table dump and the forest log tools to identify foci of computational effort, by looking for large tables (via table dump) or large recursive components (via forest log). The user next drills down progressively from the macroscopic (more aggregated and general) to the microscopic (more detailed and specific). Once sufficiently microscopic, the user then also employs the justification tools (as described above)—and/or employs restraint, especially in order to ensure termination (e.g., by limiting term size).

As usual in any kind of debugging, the above steps are iterated as needed.

6 Discussion: Scale, Skill

The debugging tools and process we have described have been used effectively for expressively rich Rulelog knowledge bases (KB's) of substantial size, ranging up to tens of thousands of (non-fact) rules. "Expressively rich" here means with expressiveness beyond that of (normal) LP. Trace-based analysis has been used for forest logs ranging up to hundreds of millions of facts, as mentioned earlier.

An important direction for future work is how to empower Subject Matter Experts (SME's), who lack skills in logic, to most effectively and efficiently debug knowledge, e.g., KB's that they author via text-based techniques [5], including in collaboration or review with KE's who do have skills in logic. This area requires considerable further research.

7 Acknowledgements

This work was supported by Vulcan, Inc., as part of the Halo Advanced Research project. Thanks to the rest of the Silk team, especially Paul Haley (Automata, Inc.) and Keith Goolsbey (Cycorp), for helpful discussions. Michael Kifer and Senlin Liang were also supported, in part, under the NSF grant 0964196.

References

1. W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.
2. Cyc. Cyc. http://www.cyc.com (project begun in approx. 1984), 2013.
3. T. Dean and M. Boddy. An Analysis of Time-dependent Planning. In *AAAI Conference on Artificial Intelligence*, pages 49–54, 1988.
4. Flora-2. Flora-2. http://flora.sourceforge.net (project begun in approx. 2000), 2013.

5. B. Grosof. Rapid Text-based Authoring of Defeasible Higher-Order Logic Formulas, via Textual Logic and Rulelog (Summary of Invited Talk). In *Proc. RuleML-2013, the 7th Intl. Web Rule Symposium*, 2013.
6. Benjamin Grosof, Mark Burstein, Mike Dean, Carl Andersen, Brett Benyo, William Ferguson, Daniela Inclezan, and Richard Shapiro. A SILK Graphical UI for Defeasible Reasoning, with a Biology Causal Process Example. In *Proc. of RuleML-2010, the 4th Intl. Web Rule Symp. (Demonstration and Poster)*, 2010.
7. Benjamin Grosof and Terrance Swift. Radial Restraint: A Semantically Clean Approach to Bounded Rationality for Logic Programs. In *Proc. AAAI-13, the 27th AAAI Conf. on Artificial Intelligence*, July 2013.
8. Halo. Project Halo. http://projecthalo.com (project begun in approx. 2002), 2013.
9. J. Sherman and M. Dean. RIF-SILK. http://silk.semwebcentral.org/RIF-SILK.html (project begun in approx. 2009), 2013.
10. Senlin Liang and Michael Kifer. A Practical Analysis of Non-Termination in Large Logic Programs. Technical report, Stony Brook University, 2013. `http://www.cs.stonybrook.edu/~sliang/iclp2013-tr.pdf`.
11. Senlin Liang and Michael Kifer. Terminyzer: An Automatic Non-Termination Analyzer for Large Logic Programs. In *PADL*, Berlin, Heidelberg, New York, 2013. Springer-Verlag.
12. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin Germany, 1984.
13. D. McAllester. Truth maintenance. In Reid Smith and Tom Mitchell, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 2, pages 1109–1116, Menlo Park, California, 1990. AAAI Press.
14. G. Pemmasani, H.-F. Guo, Y. Dong, C.R. Ramakrishnan, and I.V. Ramakrishnan. Online Justification for Tabled Logic Programs. In *International Symposium on Functional and Logic Programming (FLOPS)*, number 2998 in Lecture Notes in Computer Science, pages 24–38, 2004.
15. RuleML. Rule Markup and Modeling Initiative. http://www.ruleml.org (project begun in approx. 2000), 2013.
16. S. Russell and E. Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, 1991.
17. SILK. SILK: Semantic Inferencing on Large Knowledge. http://silk.semwebcentral.org (project begun in 2008), 2013.
18. T. Swift. Profiling Large Tabled Computations using Forest Logging. In *CICLOPS*, 2012. Available at `http://www.cs.sunysb.edu/~tswift`.
19. Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12:157–187, January 2012.
20. SWSF. Semantic Web Services Framework. http://www.w3.org/Submission/SWSF/, 2005.
21. H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang. Logic Programming with Defaults and Argumentation Theories. In *Int'l Conference on Logic Programming*, July 2009.
22. XSB. XSB. http://xsb.sourceforge.net (project begun in approx. 1993), 2013.