CONSTRUCTIVE REASONING FOR SEMANTIC WIKIS

JAKUB KOTOWSKI

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der
Ludwig-Maximilians-Universität, München

Oktober 2011

CONSTRUCTIVE REASONING FOR SEMANTIC WIKIS

JAKUB KOTOWSKI

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der
Ludwig-Maximilians-Universität, München

Oktober 2011

Erster Berichterstatter:	Prof. Dr. François Bry
	Ludwig-Maximilians-Universität München
Zweiter Berichterstatter:	Prof. Dr. Peter Dolog
	Aalborg University
Datum der Disputation:	18. Oktober 2011

ABSTRACT

One of the main design goals of social software, such as wikis, is to support and facilitate interaction and collaboration. This dissertation explores challenges that arise from extending social software with advanced facilities such as reasoning and semantic annotations and presents tools in form of a conceptual model, structured tags, a rule language, and a set of novel forward chaining and reason maintenance methods for processing such rules that help to overcome the challenges.

Wikis and semantic wikis were usually developed in an ad-hoc manner, without much thought about the underlying concepts. A *conceptual model* suitable for a semantic wiki that takes advanced features such as annotations and reasoning into account is proposed. Moreover, so called *structured tags* are proposed as a semi-formal knowledge representation step between informal and formal annotations.

The focus of rule languages for the Semantic Web has been predominantly on expert users and on the interplay of rule languages and ontologies. *KWRL*, the KiWi Rule Language, is proposed as a rule language for a semantic wiki that is easily understandable for users as it is aware of the conceptual model of a wiki and as it is inconsistency-tolerant, and that can be efficiently evaluated as it builds upon Datalog concepts.

The requirement for fast response times of interactive software translates in our work to bottom-up evaluation (materialization) of rules (views) ahead of time – that is when rules or data change, not when they are queried. Materialized views have to be updated when data or rules change. While incremental view maintenance was intensively studied in the past and literature on the subject is abundant, the existing methods have surprisingly many disadvantages – they do not provide all information desirable for explanation of derived information, they require evaluation of possibly substantially larger Datalog programs with negation, they recompute the whole extension of a predicate even if only a small part of it is affected by a change, they require adaptation for handling general rule changes.

A particular contribution of this dissertation consists in a set of *forward chaining* and *reason maintenance methods* with a simple declarative description that are efficient and derive and maintain information necessary for reason maintenance and *explanation*. The reasoning methods and most of the reason maintenance methods are described in terms of a set of *extended immediate consequence operators* the properties of which are proven in the classical logical programming framework. In contrast to existing methods, the reason maintenance methods in this dissertation work by evaluating the original Datalog program – they do not introduce negation if it is not present in the input program – and only the affected part of a predicate's extension is recomputed. Moreover, our methods directly handle changes in both data *and* rules; a rule change does not need to be handled as a special case.

A framework of *support graphs*, a data structure inspired by justification graphs of classical reason maintenance, is proposed. Support graphs enable a unified description and a formal comparison of the various reasoning and reason maintenance methods and define a notion of a derivation such that the number of derivations of an atom is always finite even in the recursive Datalog case.

A practical approach to *implementing* reasoning, reason maintenance, and explanation in the KiWi semantic platform is also investigated. It is shown how an implementation may benefit from using a graph database instead of or along with a relational database.

ZUSAMMENFASSUNG

Ein vorrangiges Designziel von sozialer Software, wie zum Beispiel Wikis, ist es, die Interaktion und Zusammenarbeit zu unterstützen und zu erleichtern. Diese Dissertation untersucht Herausforderungen, die beim Anreichern von sozialer Software mit fortgeschrittenen Funktionen, wie dem Schließen und semantischen Annotationen, entstehen, und stellt Hilfsmittel in Form eines konzeptuellen Modells, strukturierten Tags, einer Regelsprache und einer Menge von neuartigen Methoden zum Vorwärtsschließen und zur Begründungsverwaltung zur Verarbeitung solcher Regeln zur Verfügung, um diese Herausforderungen zu meistern.

Wikis und semantische Wikis wurden üblicherweise nach einem Ad-Hoc-Ansatz entwickelt, ohne sich über die darunterliegenden Konzepte Gedanken zu machen. Es wird ein für semantische Wikis geeignetes konzeptuelles Modell vorgeschlagen, welches fortgeschrittene Funktionen wie Annotationen und Schließen berücksichtigt. Darüberhinaus werden sog. strukturierte Tags als semi-formales Bindeglied zwischen informalen und formalen Annotationen vorgeschlagen. Regelsprachen des Semantic Web waren bisher hauptsächlich auf die Benutzung durch Experten und das Zusammenspiel mit Ontologien ausgerichtet. KWRL, die KiWi Rule Language, ist ein Vorschlag einer für den Benutzer leicht verständlichen Regelsprache für ein semantisches Wiki, da sie das konzeptuelle Modell eines Wikis berücksichtigt und Inkonsistenzen toleriert. Da sie auf Konzepten von Datalog aufbaut, ist sie zudem effizient auswertbar.

Der Anforderung an schnelle Antwortzeiten von interaktiver Software wird in unserer Arbeit durch eine vorzeitige Bottom-Up-Auswertung (Materialisierung) von Regeln (Sichten) entsprochen – d.h. wenn sich Regeln oder Daten ändern, und nicht erst wenn sie angefragt werden. Materialisierte Sichten müssen aktualisiert werden, sobald sich Daten oder Regeln ändern. Während "Incremental View Maintenance" intensiv in der Vergangenheit studiert wurde und es reichlich Literatur dazu gibt, haben die existierenden Methoden überraschenderweise viele Nachteile – sie liefern nicht all die Information, die für die Erklärung der hergeleiteten Informationen wünschenswert wäre, sie erfordern die Auswertung von unter Umständen erheblich größeren Datalog Programmen mit Negation, sie berechnen die vollständige Erweiterung eines Prädikats selbst wenn nur ein kleiner Teil davon von einer änderung betroffen ist, und sie müssen angepasst werden, um allgemeine Regeländerungen verarbeiten zu können.

Ein Beitrag dieser Dissertation besteht in einer Menge von Methoden zum Vorwärtsschließen und zur Begründungsverwaltung mit einer einfachen deklarativen Beschreibung, welche effizient sind und die zur Begründungsverwaltung und Erklärung notwendigen Informationen ableiten und bereithalten. Die Methoden zum Schließen und die meisten der Methoden zur Begründungsverwaltung werden in Form einer Menge von erweiterten, unmittelbaren Folgerungsoperatoren beschrieben, deren Eigenschaften im Rahmen der klassischen Logikprogrammierung bewiesen werden. Im Gegensatz zu den bereits existierenden Methoden der Begründungsverwaltung, funktionieren die Methoden dieser Dissertation durch Auswertung des ursprünglichen Datalog Programmes – es wird keine Negation eingeführt, wenn diese nicht schon im Eingabeprogramm vorhanden war – und nur der von einer Prädikatserweiterung betroffene Teil wird neu berechnet. Darüberhinaus werden änderungen in Daten sowohl als auch Regeln direkt durch unser Verfahren behandelt; eine Regeländerung stellt also keinen Spezialfall dar.

Weiterhin wird das Konzept von Support Graphen vorgeschlagen, einer von den Rechtfertigungsgraphen der klassischen Begründungsverwaltung inspirierten Datenstruktur. Diese ermöglichen eine einheitliche Beschreibung und formale Gegenüberstellung der verschiedenen Methoden zum Schließen und zur Begründungsverwaltung und definieren einen Begriff einer Herleitung, derart dass die Anzahl der Herleitungen eines Atoms immer endlich ist, selbst im Falle von rekursivem Datalog.

Auch wird in der semantischen Plattform von KiWi ein praktischer Ansatz zur Implementierung des Schließens, der Begründungsverwaltung und Erklärung untersucht. Es wird gezeigt, wie eine Implementierung von einer Graphdatenbank anstelle oder in Ergänzung zu einer relationen Datenbank profitieren kann.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

Jakub Kotowski, François Bry, and Norbert Eisinger. A Potpourri of Reason Maintenance Methods. Submitted for publication.
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2011-9

Jakub Kotowski, François Bry, and Simon Brodt. Reasoning as Axioms Change – Incremental View Maintenance Reconsidered. In *Proceedings of the 5th International Conference on Web Reasoning and Rule Systems* (RR), Galway, Ireland (23rd - 24th August 2011).
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2011-5

Jakub Kotowski and François Bry. A Perfect Match for Reasoning, Explanation and Reason Maintenance: OWL 2 RL and Semantic Wikis. In *Proceedings of 5th Semantic Wiki Workshop (SemWiki)*, 2010. (Demo, short paper), Hersonissos, Crete, Greece (31st May 2010)
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2010-5

François Bry and Jakub Kotowski. A Social Vision of Knowledge Representation and Reasoning. In *Proceedings of the 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, 2010, Špindlerův Mlýn, Czech Republic (23rd–29th January 2010)
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2010-2

François Bry, Michael Eckert, Jakub Kotowski, and Klara Weiand. What the User Interacts With: Reflections on Conceptual Models for Sematic Wikis. In *Proceedings of the 4th Workshop on Semantic Wikis (SemWiki)*, 2009, Heraklion, Greece (31st May–4th June 2009)
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2009-2

François Bry, Jakub Kotowski, and Klara Weiand. Querying and Reasoning for Social Semantic Software. In *Proceedings of 6th European Semantic Web Conference* (ESWC), 2009. (Poster), Heraklion, Greece (31st May–4th June 2009)
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2009-3

Jakub Kotowski, Stephanie Stroka, François Bry, and Sebastian Schaffert. Dependency Updates and Reasoning for KiWi. In *Proceedings of the 4th Workshop on Semantic Wikis (SemWiki)*, 2009, Heraklion, Greece (31st May–4th June 2009)
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2009-4

Jakub Kotowski and François Bry. Reason Maintenance – Conceptual Framework. *KiWi project deliverable* D2.4, June 2009.
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2009-17

Jakub Kotowski and François Bry. Reason Maintenance – State of the Art. *KiWi project deliverable D2.3*, September 2008.
http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2008-17

ACKNOWLEDGMENTS

Many people have helped me with writing this thesis and I am grateful that I could enjoy the supportive environment of LMU Munich. I would like to thank especially to my advisor François Bry. He was always ready to provide help and support, including matters not directly related to research. I learned a lot from him about research and work in academia in general. I am also indebted to Peter Dolog for taking the time and effort to read the whole thesis thoroughly as the external reviewer. The Programming and Modelling Languages group has always been very inspiring and helpful and I am glad I could be one of its members. I want to thank especially Norbert Eisinger, from whom I have learned a lot, for careful and repeated proofreading and for spending time to discuss most of my research with me. I thank Sebastian Schaffert, the scientific coordinator of the KiWi project, for all kinds of advice and for allowing me to spend time working with the KiWi core team in Salzburg. I enjoyed working with all the members of the KiWi project at Aalborg University, Brno University of Technology, Logica, Oracle (a team which was formerly at Sun Microsystems), Salzburg Research, and the Semantic Web Company many of whom also provided me with valuable insights, discussions, and help implementing the prototype.

I cannot mention all who helped or supported me but I would like to thank namely at least those who I worked with the most: Simon Brodt, Frederico Durão, Michael Eckert, Tim Furche, Steffen Hausmann, Josef Holý, Karsten Jahn, Fabian Kneißl, Alex Kohn, Stephan Leutenmayr, Benedikt Linse, Hans Jürgen Ohlbach, Alexander Pohl, Olga Poppe, Sebastian Schaffert, Marek Schmidt, Edgar-Philipp Stoffel, Stephanie Stroka, Klara Weiand, Christoph Wieser, and Harald Zauner. I thank Martin Josko and Ingeborg von Troschke for always ensuring that day to day things run smoothly. I thank also the participants in the user study on structured tags and RDF for making the research possible. The research described in this thesis was partly funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 211932 as the "KiWi – Knowledge in a Wiki" project. This funding allowed me to pursue my research, travel to conferences, and get in touch with fellow researchers.

Last but not least, I would like to thank my family and friends for supporting me and encouraging me during the work on this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Social software, such as wikis, is affluent with user interaction and one of its main design goals is to support and facilitate interaction and collaboration. In other words, social software is often concerned with creative work and work in progress. Any work that is still in progress will be changed and it is likely to contain inconsistencies and generate disagreements. While tools and theories for handling change and inconsistencies already do exist, they have shortcomings with respect to their use by casual or non-expert users.

Extending social software with advanced facilities such as reasoning presents a set of challenges that need to be overcome in order to provide a nimble and understandable user experience for non-expert users. In this dissertation, these challenges are analysed and tools for working with a semantic wiki in form of a conceptual model, structured tags, a rule language, and a set of novel forward chaining and reason maintenance methods are presented. The potential impact of the presented forward chaining and reason maintenance methods in particular extends beyond the realm of the Semantic Web as these methods concern problems traditionally solved in the broader area of (deductive) databases.

Most of the work reported about in this dissertation has been realized in the context of the KiWi – Knowledge in a Wiki – project[1] of the Seventh Framework Programme of the European Community. The goal of the project was to provide practical social and semantic tools for software and process management in an enterprise setting. Two of the project partners, were large international IT companies, Logica and Sun Microsystems (taken over by Oracle during 2009 and early 2010), that provided practical use cases based on their actual needs. Communication with these companies revealed that advanced semantic features can indeed be difficult to fully grasp even for not-so-casual users and that the implemented KiWi 1.0 prototype mostly met their expectations.

1.1 CONTRIBUTIONS AND STRUCTURE OF THIS THESIS

Social semantic software has originated from the original Web introduced [131] by Tim Berners-Lee in 1989. Chapter 2 summarizes this evolution and highlights milestones and technologies important for this thesis. Chapter 3 analyses properties characteristic for social applications and, based on this analysis, suggests a set of requirements on social semantic software that guide the rest of this thesis.

A *conceptual model* for a semantic wiki that reflects the identified requirements is presented in Chapter 4. Wikis and semantic wikis were usually developed in ad hoc manners, without much thought about the underlying concepts. One of the few exceptions is [16] that focuses on the expressivity and formal grounding of the Semantic MediaWiki. Our conceptual model contributes a more user-oriented view with an emphasis on the use of annotations.

1 http://www.kiwi-project.eu/

A particular contribution that extends beyond the conceptual model is the proposal of *structured tags* also in Chapter 4. Structured tags are a flexible semi-formal knowledge representation step between informal annotations such as tags and formal annotations such as RDF and OWL. Structured tags are suggested as a more user-friendly addition to the standard RDF annotation formalism. Their usability in this respect is evaluated in a user study in Chapter 5.

The KiWi Rule Language, KWRL, presented in Chapter 6, is an inconsistency-tolerant rule language for a semantic wiki, the KiWi wiki, that aims to be easily understandable for users, as it is aware of the conceptual model of the wiki, and that can be efficiently evaluated as it builds upon Datalog concepts. KWRL is, to the best of our knowledge, the first rule language specifically about annotations. It employs a kind of value invention that differs from those common in Semantic Web rule languages and that is more suitable for annotations and the web of Linked Data.

The second part of this dissertation focuses on reasoning and reason maintenance.

Users expect that web applications are interactive and fast. The requirement for fast response time of interactive software translates in our work to bottom-up evaluation (materialization) of rules (views) ahead of time. Materialized views have to be updated when data or rules change which brings the need for reason maintenance. While incremental view maintenance was intensively studied in the past and literature on the subject is abundant, the existing methods have surprisingly many disadvantages – they do not provide all information desirable for explanation of derived information, they require evaluation of possibly substantially larger Datalog programs with negation, they recompute the whole extension of a predicate even if only a small part of it is affected by a change, they require adaptation for handling general rule changes. A particular contribution of this thesis consists in a set of *forward chaining* and *reason maintenance methods* presented in Chapters 7 and 8 that overcome these shortcomings.

Chapter 7 first introduces so called *support graphs* – a justification-graphs-inspired formalism that allows for a unified description of our reasoning and reason maintenance algorithms. Support graphs allow to precisely determine relations between algorithms inspired from different fields, such as reason maintenance [74, 72] and incremental view maintenance [103], that have traditionally used different and to some extent incompatible formalisms. Moreover, support graphs and derivation in support graphs are defined in such a way that the number of derivations of an atom in a support graph with respect to a recursive Datalog program is a finite number (Proposition 150 and Corollary 143). The corresponding number may be infinite in the traditional methods [158, 138]. Chapter 7 then defines several *extended immediate consequence operators* on sets and multisets that allow for a declarative and arguably simple description of the respectively extended forward-chaining algorithms that derive information useful for reason maintenance and explanation. These algorithms and operators can also be seen as providing alternative set and multiset (duplicate) semantics for Datalog. Their properties are proven in the classical logical programming framework based on fixpoint theory [136].

Chapter 8 describes several novel *reason maintenance methods* based on the extended immediate consequence operators from Chapter 7. These reason maintenance methods overcome shortcomings of the existing methods mentioned above, i.e. our reason maintenance methods use the original input program, i.e. they do not introduce negation and do not increase the size of the program, they recompute only the part of the predicate extension that is affected by a change, and they directly handle changes in both data *and* rules; a rule change does not need to be handled as a special case. Most of the methods also provide additional information that may be used for explanation. The price to pay for the above-mentioned advantages is a modification of classical forward chaining in most cases. The chapter further discusses how the methods may be combined in a real-world implementation of a semantic web application and proves minor results (batch processing and lazy evaluation in Section 8.9) that would likely facilitate an implementation.

A practical approach to an *implementation* of reasoning, reason maintenance, and explanation in the KiWi wiki is also investigated in this work and it is shown how an implementation may benefit from using a graph database in comparison to a relational database. An additional contribution in this respect is a functional implementation of reasoning, reason maintenance, and explanation in the KiWi 1.0 system that is Open Source and that can be downloaded at `http://code.google.com/p/kiwi/downloads/list`.

2 FROM WIKIS TO SEMANTIC WIKIS

This chapter provides a brief overview of the context of this dissertation the main use case of which are semantic wikis. A semantic wiki is a semantic social web application. The following sections thus introduce a path from the beginning of the Web via emergence of the Social Web and wikis up to the Semantic Web and its technologies such as RDF triple stores that can benefit from results presented in later chapters. Semantic wikis emerge from the intersection of the Social Web with the Semantic Web.

2.1 THE WORLD WIDE WEB

The World Wide Web is a system of interlinked hypertext documents accessed via the Internet [3]. It started [130] as a small project at CERN with an "Information Management" project proposal [131] by Tim Berners-Lee in 1989 at CERN. By the end of 1990, the first web browser and web server were developed. By April 1994, the size of the Web was 829 servers and it grew much faster than other Internet services such as Gopher (a campus-wide information system) and WAIS (text-based information retrieval) [132]. This original mostly non-interactive web of webpages and hyperlinks is now often referred to as Web 1.0.

The next bigger and agreed upon shift of the Web came with a rather sudden broad adoption of AJAX (Asynchronous JavaScript and XML) technologies in 2004 (Google Gmail), 2005 (Google Maps) and with web applications that facilitate user participation and collaboration on the Web. It was described as Web 2.0 by Tim O'Reily in 2005 [171]. Three conceptual parts of Web 2.0 may be distinguished [1]: rich internet applications (enabled by AJAX, Adobe Flash, and other technologies), services (mostly RSS/ATOM feeds and public web application APIs), and social web applications (the Social Web).

2.2 THE SOCIAL WEB

Social media are media meant for social communication. The Social Web is a term for social media on the Web[1], see also [78]. Social websites are characterized by enabling user participation, collaboration, information sharing, and relationship building. Currently popular website types include wikis, social networking, photo sharing, blogging, micro blogging, bookmarking, and bulk discount platforms.

Social websites typically provide a low barrier for participation meaning that beginning users may engage for example in tagging to enrich content while more advanced users edit and create content. Tagging, assigning labels to content, is employed by many social websites to simplify navigation and search for example by means of faceted browsing [230] and its success lead to a variety of research [95, 54, 141, 10, 108, 53].

1 Non-Web social media include e.g. virtual worlds such as Second Life (http://secondlife.com/)

The focus of this dissertation is predominantly on wikis and metadata (and in particular tagging) but the results may be valuable to a broader spectrum of social web applications.

2.3 WIKIS

A *wiki* is a web application for creating and managing information in the form of web pages [133]. Coincidentally, a personal wiki (i.e. a wiki typically not used for collaboration) could perhaps be seen as a re-implementation of the experiment that led to the idea of the World Wide Web:

> The idea of the Web was promted by positive experience of a small "home-brew" personal hypertext system used for keeping track of personal information on a distributed project. [132]

Wikis are characterized by a balanced mix of technological and social features. Typically, they are easy to use [68], in the sense that they do not require any special knowledge (e.g. HTML, CSS, programming, ...) other than a mere familiarity with the concept of hypertext [57] and they are open and easily accessible in the sense that any user can edit any page often even anonymously. Openness can be seen as a social feature because it is enabled by trusting that people will not misuse the system. Trust has its obvious limits which are, to certain extent, surmounted for example with the help of versioning, i.e. keeping history of each page so that malicious and inadvertent edits can always be reverted. Another notable feature is support for easy hyperlinking [50], especially within a particular wiki instance but also to external content.

Wiki was described by the inventor of the concept, Ward Cunningham, as "The simplest online database that could possibly work." [62]. Ward Cunningham designed and implemented the first wiki in 1994 and launched it as the WikiWikiWeb in 1995 [2]. It was a notable effort because at that time most web pages were "read-only", a fact about which Tim-Berners Lee complained even four years later [25]:

> " I wanted the Web to be what I call an interactive space where everybody can edit. And I started saying 'interactive,' and then I read in the media that the Web was great because it was 'interactive,' meaning you could click. This was not what I meant by interactivity."

Note that from research perspective, a wiki can serve as a model of the web – as a "web in the small" [198] – because it has the hyperlinked organic nature of the Web but it is self-contained and simplified at the same time.

Today, there is a wealth of wiki implementations [49] and wikis have a variety of applications: as a collaborative knowledge sharing software, the most prominent example of which is Wikipedia [2], for collaborative knowledge building [61], for collaborative learning [186, 212], as a support for process-driven knowledge-intensive tasks [87, 122] or as a personal knowledge management tool. In all these cases wiki serves as a knowledge management tool which is usually intended for incremental collaborative work and serves the purpose of knowledge sharing – either with other people or with a future self. This aspect of wikis is crucial and will be revisited in

2 http://en.wikipedia.org/wiki/WikiWikiWeb

the discussion of requirements for reasoning in semantic wikis in the next section.

Traditional wikis used to be aesthetically crude [129], a shortcoming still (as of 2010) visible in the WikiWikiWeb, it is however worth mentioning that Ward Cunningham regarded it as a feature which keeps people focused on the deep functional quality [129]. It has also been argued that simple user interface enables fast operation due to low resource demand [122]. Wikis usually provide a common appearance to all pages in order to convey a sense of unity and to let people wholly focus on structure and content [127]. A more substantive problem can emerge with structuring and searching when a wiki grows large [47]. Traditional full text search seems to be insufficient for large amounts of structured data [173, 198]. This, together with the fact that full text search is the primary means of navigation in wikis, may leave users to feel lost. This problem can be mitigated, to a certain extent, by implementing a concept of hierarchical workspaces such as in TWiki [187], by creating overview pages to improve the possibilities of navigation and by improving the search facility.

Another set of problems, both technological and social, arises when deploying a wiki system in a corporate environment. Technological problems include those of authentication and access rights (e.g. financial and personal data have to be protected). Social problems may arise because of different motivations of users to create content, e.g. a fear of losing a competitive advantage by sharing knowledge with colleagues. In this sense it may be useful to view a wiki as a market [127]. It should however be noted that adoption of wikis in corporate environment is heavily influenced by the ability of company leadership to understand and clarify economic benefits and by the ability to perform organizational changes that may be necessary to fully leverage the technology [60]. See for example [63] for a more thorough discussion of the role of wikis in companies.

Content in traditional wikis consists of natural language text (and possibly multimedia files) and is not directly accessible to automated semantic processing. Therefore, knowledge in wikis can be located only through simple user-generated structures (tables of contents, inter-page links) and simple full text keyword search which hinders information reuse – information in overview pages has to be manually duplicated and maintained. More advanced functionalities that are highly desirable in knowledge-intensive professional contexts such as reasoning and semantic browsing are not possible.

2.4 THE SEMANTIC WEB

Knowledge hidden in data published on the Web is, despite advances in natural language processing, not readily amenable to computer processing. The main idea behind the Semantic Web (sometimes called Web 3.0) is to mediate meaning represented on the Web to computers by means of semantic annotations. As Tim Berners-Lee puts it in the original article about the Semantic Web:

> The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users. [27]

The idea of a roaming software agent working on behalf of a user – putting together a vacation itinerary, signing up for a medical examination while considering a range of personalized criteria – is usually referred to as the Semantic Web vision. The vision is to be realized by giving data unique global identifiers, ideally dereferenceable URIs, and providing semantic annotations based on these identifiers using for example the Resource Description Framework (RDF) (Section 2.4.1) and the Web Ontology Language (OWL) (Section 2.4.3). The semantic annotations mediate precise formal meaning and enable reasoning about the data on the Web.

> The Semantic Web is a Web of actionable information – information derived from data through a semantic theory for interpreting the symbols. The semantic theory provides an account of "meaning" in which the logical connection of terms establishes interoperability between systems. [200]

While semantic annotations should ideally enable interoperability between systems, interoperability is hindered by several factors in practice: annotations do not use the same vocabularies (ontologies), the same identifier is used for (slightly) different concepts by different semantic data publishers depending on the exact use and purpose of the data. Using different ontologies for the same data then requires automated methods of ontology alignment or manual mapping, a still active research topic. Using one identifier for differing concepts by different parties is a problem of context and intended meaning. Such a problem arises also for example between different natural languages in which "the same" concept may in fact have slightly different meanings (it may be two slightly different concepts) reflecting for example cultural differences. Another limiting factor was, until recently, scarceness of semantic data on the Web which lead to the origin of the Linked Data community (Section 2.4.4). These and other challenges are discussed in the literature [22, 114, 107, 90, 4, 228] and probably contribute to the fact that the Semantic Web vision has not yet been realised, after more than 10 years since the article [27] of Tim Berners-Lee and Jim Hendler introduced it to general public.

2.4.1 *RDF*

"The Resource Description Framework (RDF) is a language for representing information about 'resources' in the World Wide Web." [139]. The vision for RDF was to provide a minimalist knowledge representation format for the web [200] and it is currently perhaps the most common format for semantic data. RDF data are suited for processing by machines but not easily human-interpretable. For practical applications, support of RDF is important in order to enable of interoperability with current Semantic Web and linked data [26, 28] applications.

"RDF's vocabulary description language, RDF Schema, is a semantic extension [...] of RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources." [36]. These basic constructs can be used to describe ontologies.

2.4.1.1 *RDF Concepts*

The central concept of RDF is an RDF triple. An RDF triple consists of a subject, a property, and an object and expresses a simple statement not unlike a simple natural language sentence of the form "subject verb object". An RDF triple has however a precise formal meaning and should not be confused with natural language statements. A set of RDF triples is called an RDF graph. An RDF graph is in fact a labelled directed graph where subjects and objects are nodes and edges are labelled with properties. Note that a URI may appear as both an edge label and a node in an RDF graph and this is in fact useful and common, consider the following two triples: "isFatherOf rdf:type rdf:Property; Peter isFatherOf Meg", see Figure 2.1.

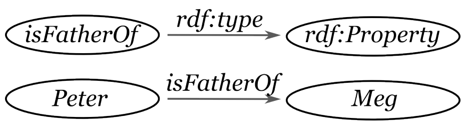


Figure 2.1: An RDF graph.

Three types of entities can take place of an object in an RDF triple: a URI reference, an RDF literal, and an RDF blank node. The subject must be either a blank node or a URI reference and the property must be a URI reference. The type restrictions of properties and particularly subjects are often disputed[3] and not always adhered to in practice.

URI references are used to refer to concepts and objects one wishes to make a statement about. It is important to keep in mind that one object can have many absolute URIs (see[4] RFC 2396 for the precise difference between URIs and URI references). While RDF makes no assumption about the URIs, the Linked Data [26] movement suggests that RDF URIs should be dereferenceable and should contain a description of the resource.

RDF literals can be used when a literal value such as a name string `"Peter"` conveys the whole meaning and inventing a URI reference for such a value would be superfluous. An example triple may be: `example:Peter foaf:name "Peter"`, using the popular FOAF[5], friend of a friend, onotology. RDF distinguishes plain and typed literals. The string `"Peter"` is a plain literal; it has no associated datatype. In comparison, `"Peter"^^xsd:string` is a typed literal of the `xsd:string` datatype.

RDF blank nodes (or often just b-nodes) can be used when it is not possible or convenient to identify a concept or an object with a URI. The RDF Abstract Syntax [121] does not provide any symbols for b-nodes, it only states that "Given two blank nodes, it is possible to determine whether or not they are the same." For example Turtle[6], a concrete syntax of RDF, uses the "_:nodeID" schema for blank nodes. The use of blank nodes in RDF asserts existence of a concept or object without identifying it.

See the "Resource Description Framework (RDF): Concepts and Abstract Syntax" W3C recommendation [121] for more details about RDF concepts.

3 See for example http://lists.w3.org/Archives/Public/semantic-web/2010Jul/thread.html#msg2
4 http://www.ietf.org/rfc/rfc2396.txt
5 http://www.foaf-project.org/
6 http://www.w3.org/TeamSubmission/turtle/

2.4.1.2 *RDF Serializations*

There are several common RDF serialization: RDF/XML, N3, Turtle, and N-Triples. RDF/XML [19] is officially the primary serialization format of RDF. It is an XML application which however is arguably not very human-reader friendly but serves well as an interchange format on the web.

Listing 2.1: An example of RDF/XML.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://www.kiwi-project.eu">
         <dc:title>KiWi - Knowledge in a Wiki</dc:title>
         <dc:publisher>KiWi</dc:publisher>
  </rdf:Description>
</rdf:RDF>
```

Notation 3 [24], or N3, is an RDF serialization format designed with human-readability in mind. It provides many shortcuts for writing complex RDF and provides bigger expressivity than RDF/XML (and in fact RDF graphs) as it allows for example for specifying rules and universal and existential quantification. Turtle [21] is a subset of Notation 3 which does not go beyond RDF graphs. While Turtle is not a W3C recommendation, it has become popular on the Semantic Web. N-Triples[20] is a line-based subset of Turtle; it is a simple one triple per line format.

Listing 2.2: An example of N-Triples.

```
<http://www.kiwi-project.eu> <http://purl.org/dc/elements/1.1/
    title> "KiWi - Knowledge in a Wiki" .
<http://www.kiwi-project.eu> <http://purl.org/dc/elements/1.1/
    publisher> "KiWi" .
```

2.4.1.3 *RDF(S) Model Theory*

Perhaps the biggest advantage of using RDF and RDF(S) as knowledge representation languages is that their semantics is formally specified by a model theory [112]. Here, only a brief overview of RDF model theory is provided. Most of this dissertation uses only the logical core of RDF(S) as specified by Munoz et al. [160].

Definition 1. *Let* $\mathbb{U}$ *be a set of RDF URI references,* $\mathbb{B}$ *a set of blank nodes, and* $\mathbb{L}$ *a set of literals such that they are pairwise disjoint. Unions of these sets are denoted by concatenating their names in the following for brevity. A tuple* $(s, p, o) \in \mathbb{UB} \times \mathbb{U} \times \mathbb{UBL}$ *is an RDF triple. An RDF graph is a set of triples. An RDF graph is ground if it does not contain any blank nodes. The set of members of* $\mathbb{UBL}$ *that occur in the triples of a graph* G *is called the universe of* G*, denoted* $\mathrm{universe}(G)$. *The vocabulary of* G *is the set* $\mathrm{universe}(G) \cap \mathbb{UL}$.

Note that for example Munoz et al. [160] allow literals in the subject position.

The RDF model theory [112] defines four RDF interpretations: simple RDF interpretations, RDF interpretations, RDF(S) interpretations, and RDF(S) datatyped interpretations. They are theories with increasingly more restrictions on the vocabulary of a graph. For example RDF interpretations impose extra semantic conditions on `rdf:type`, `rdf:Property`, and `rdf:XMLLiteral` and RDF(S) datatyped interpretations specify a minimal semantics for typed literals in an RDF graph (given a datatype map, a mapping from lexical space to value space).

The notion of entailment between RDF graphs is defined using interpretations similarly to classical logic entailment and it also comes in four kinds: simple, RDF, RDF(S), and datatype (D-Entailment) entailments.

The main purpose of providing a formal theory of meaning of RDF graphs is to support reasoning and to determine when it is valid; or as the specification puts it:

> The chief utility of a formal semantic theory is not to provide any deep analysis of the nature of the things being described by the language or to suggest any particular processing model, but rather to provide a technical way to determine when inference processes are valid, i.e. when they preserve truth. This provides the maximal freedom for implementations while preserving a globally coherent notion of meaning. [...] The notion of entailment [...] makes it possible to define valid inference rules. [112]

The inference rules are listed in the RDF model theory specification [112] (W3C Recommendation 10 February 2004) but they are not complete as shown in [140]. [140] also provides a correction of this flaw and provides also other suggestions for improvement, mostly regarding ambiguities in the definitions, of a future revision of the specification.

2.4.2 *RDF Triple Stores*

RDF data are usually stored in dedicated data management systems called triple stores. While three years ago a study [199] found that RDF stores were not yet capable of scaling real-world queries to tens of millions of RDF triples, a recent study states [111] that certain current RDF stores such as OpenLink Virtuoso[7] or 4Store[8] can handle the volumes of the European Digital Library project Europeana[9] which are on the magnitude of hundreds of millions of RDF triples. Other popular triple stores include Sesame[10] which is also a general framework for processing RDF data, Jena[11] – also a general framework as well as a triple store, and OWLIM[12] which is also available as a SAIL – an interface layer for the Sesame framework.

Most current triple stores support some kind of reasoning. Often, query answering is sped up by forward chaining in advance and some triple stores (such as e.g. Jena) are capable of combining forward and backward query evaluation. To the best of our knowledge, no current forward chaining triple store implements incremental updates of materialization in the case of triple removals. Jena employs a general purpose RETE-based [86] forward chaining reasoner which supports incremental additions but no incremental re-

7 http://virtuoso.openlinksw.com/
8 http://4store.org/
9 http://europeana.eu/
10 http://www.openrdf.org/
11 http://www.openjena.org/
12 http://www.ontotext.com/owlim/

movals.[13] Sesame itself has a limited support for custom rule reasoning and does not offer incremental removals in the general case. There is an implementation of incremental updates for Sesame [39] inspired by reason maintenance [73] but it is specific to RDF/S reasoning and requires space to store auxiliary information about derivations. OWLIM also does not support incremental removals.[14] From personal communication with one of the authors at ESWC 2010, we know that the state-of-the-art forward chaining reasoner for the Semantic Web SAOR (Scalable Authoritative OWL Reasoner) [116, 117] also does not implement incremental removals. In most cases, removal is implemented by recomputing materialization from scratch.

Chapter 8 of this dissertation contributes several incremental maintenance algorithms that forward chaining triple stores can use to implement support for incremental removal. Note that for example Algorithm 8.15 from Section 8.4 can likely be implemented *using* the existing forward chaining reasoners while leveraging all or most of their optimizations.

2.4.3 *OWL*

The Web Ontology Language [220], OWL, is a knowledge representation language, meant for the Semantic Web, with a formally defined meaning. It can be seen as more expressive than RDF as it provides more predefined constructs for representing ontologies. An ontology is usually defined as an explicit specification of conceptualization [101]. "A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose." [101]

OWL is a subset of first order logic and the main idea behind its design is to provide guarantees about computational complexity depending on which of its subsets is used. OWL has several flavours: OWL Lite, OWL DL, and OWL Full. OWL Lite is a light version of OWL DL which is a decidable fragment of OWL; DL stands for Description Logic. Because OWL is a vocabulary extension of RDF and in order to ensure the computability properties, the Lite and DL versions of OWL put restrictions on the use of RDF Schema requiring disjointness of classes, properties, individuals, and data values. OWL Full allows arbitrary mixing with RDF Schema.

The current version of OWL, OWL 2 [220], also defines three so called profiles [157]: OWL 2 EL, OWL 2 QL, and OWL 2 RL. The profiles are sublanguages of OWL aimed at specific application scenarios. OWL 2 EL is aimed at applications requiring large ontologies as it allows for polynomial time reasoning. OWL 2 QL is suitable for applications with rather small ontologies and large amounts of instance data and allows for answering conjunctive queries in $\mathtt{LogSpace}$ ($\mathtt{LogSpace} \subseteq \mathtt{P}$, it is an open problem whether $\mathtt{LogSpace} = \mathtt{P}$). OWL 2 RL is aimed at applications relying on rule-based reasoning and database technologies; the specification provides the semantics of OWL 2 RL axiomatized using rules (first-order implications).

OWL is a language widely used on the current Semantic Web and it is in part, e.g. the GoodRelations[15] e-commerce ontology, recognized also by Internet search engines such as Google and Yahoo.

13 http://tech.groups.yahoo.com/group/jena-dev/message/43618
14 http://www.mail-archive.com/owlim-discussion@ontotext.com/msg00496.html
15 http://purl.org/goodrelations/

2.4.4 *Linked Data*

Tim Berners-Lee argued [26] that Semantic Web research projects produced many ontologies and data stores but made only very little data available on the Web. In order to promote semantic data publishing on the Web, he suggested [26] four rules to follow:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URIs, so that they can discover more things.

The idea behind linked data is to create a web of data: a piece of linked data is linked to other data that can thus be discovered by following the links. Linked data freely available on the Web is called "Linked Open Data."

Since Berners-Lee's proposal in 2006, the community took up this effort and now there are 203 data sets available that consist together of over 25 billion RDF triples and are interlinked by around 395 million RDF links, see the Linking Open Data cloud diagram[16,17] in Figure 2.2.

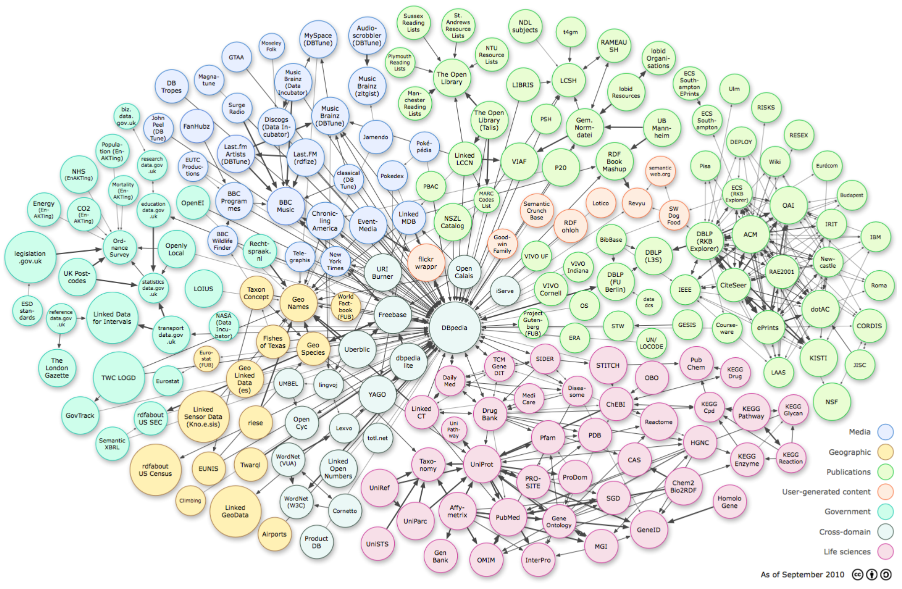


Figure 2.2: The so called Linking Open Data cloud diagram showing most of Linked Open Data as of September 22, 2010. The size of the circles corresponds to the number of triples in each dataset, there are five sizes: for <10k, 10k-500k, 500k-10M, 10M-1B, >1B triples. An arrow indicates the existence of at least 50 links between two datasets. The thickness of an arrow corresponds to the number of links: <1k, 1k-100k, >100k links.

16 http://richard.cyganiak.de/2007/10/lod/

17 http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

2.5 SEMANTIC WIKIS

Semantic wikis are ordinary wikis extended with Semantic Web technologies and, in particular, with means for specifying information in machine-processable ways.

Probably the first semantic wiki software was the Platypus wiki[18] the development of which started in December 2003 and which was introduced in 2004. Since then, many different semantic wiki systems have been developed [13, 120, 197, 198, 33, 173, 217, 180]. Some, such as Semantic MediaWiki [124], are implemented as an extension of an existing wiki, other, such as for example IkeWiki [196] and its successor KiWi[19], are built from scratch. See http://semanticweb.org/wiki/Semantic_wiki_projects/ for an overview of currently active and past semantic wiki projects.

Semantic wikis fall into two basic categories described, after [126], as "Semantic data for Wikis" and "Wikis for semantic data". The former category consists of wikis enhanced with semantic technologies while the latter one consists of wikis which support development of ontologies (wikis for ontology engineers). This distinction is currently not strict as semantic wikis exist that fall into both categories [196].

In general, semantic wikis include features such as:

- semantically annotated links,
- semantically annotated content,
- personalisation based on semantic data,
- information extraction – to enrich content with metadata,
- reasoning – to make implicit information explicit,
- semantic browsing – e.g. faceted search with facets based on automatically generated metadata,
- semantic search – querying not only by keyword but also by (possibly implicit) relations and categories,
- RDF(S), OWL, web services, linked data – to provide interoperability.

The range of technologies and concepts of the Semantic Web employed in semantic wikis enables to see them as "Semantic Web in small" [198] and thus as a small self-contained space ideal for implementing and trying out ideas and technology aimed at the Semantic Web too [126].

Although semantic wikis have been implemented and researched since more than six years, little formal research of their functionality has been done. One notable exception is [16] by a group around James Hendler. [16] formally studies Semantic MediaWiki, its modelling and query languages and specifies for them a semantics alternative to the OWL DL based semantics of [218]. Although the report focuses on Semantic MediaWiki, it illustrates that all wikis can benefit from having a formally worked out model: [16] shows the complexity of entailment and querying in Semantic MediaWiki based on a formal semantics that is more faithful to the actual wiki than the original semantics of [218], a way to implement a native reasoner (as opposed to translating to SQL or SPARQL), and a set of possible problems and extensions of Semantic MediaWiki functionality. The paper does not try to analyse decisions leading to the conceptual model of Semantic

18 http://www.c2.com/cgi/wiki?PlatypusWiki
19 http://kiwi-project.eu/

MediaWiki. A contribution to the study of conceptual models of wikis can be found in Chapter 4 and partly in Chapter 5.

Many semantic wikis support reasoning but most of them rely on conventional reasoning techniques and technologies that were developed for use in a mostly static environment with annotations, rules, and ontologies being crafted by knowledge representation experts. This is in contraposition to the ever-changing, dynamic character of wikis where content and annotations are, for the most part, created by regular users. Reasoning suitable for the wiki environment arguably should consider these differences to the traditional setting. See Chapter 3, where these aspects are explored in more detail.

Reasoning in wikis was investigated by Krötszch et al. in [126] – it compares a rather simple (at that time) but efficient reasoning in Semantic MediaWiki with a more sophisticated reasoning facility in IkeWiki and mentions several challenges with respect to reasoning: validation of formalised knowledge (with respect to a background ontology), performance, usability, closed and open world assumption (which is mentioned also by [16]). Another often mentioned concern is the expressiveness of modelling and querying formalisms – often, reasoning in semantic wikis is constrained to RDF(S) entailment as supported by their underlying RDF triple store while support of negation and user-defined rules is desirable. These problems are as important for semantic wikis as for the Semantic Web, however, the solution may differ. For example [175] suggests that the Semantic Web is better served by classical logics rather than by a Datalog-related logics[20] while [16] found that their Datalog-based semantics was a better match for Semantic-Media Wiki than the description logics based semantics proposed by [218]. In this case the essential difference probably lies in the fact that the closed world assumption is found to be better suited for a semantic wiki and the open world assumption for the Semantic Web. Let us review Semantic MediaWiki and IkeWiki in more detail now.

2.5.1 *Semantic MediaWiki*

Semantic MediaWiki (SMW) is a semantic extension of the popular open-source wiki engine MediaWiki[21] that is used to run for example Wikipedia.[22] SMW was originally developed to address three problems [217]: consistency of content, accessing knowledge (i.e. better search and search results), better knowledge reusing (which is related to consistency too).

SMW classifies pages into namespaces (individual elements, categories, properties, and types) to distinguish pages according to their function. Semantic annotations are based on properties and types in SMW. Properties express a binary relationship between a semantic entity (represented by a page) and another entity or data value. SMW also allows links to be characterized by properties. For example if one wishes to express that London is the capital of the United Kingdom then it can be done in an extension of *wiki text* (a syntax for describing wiki pages) by writing `[[capital of::England]]` on the page for London. A page "Property:capital_of" (in the Property namespace) may then contain additional information about the

20 It has to be mentioned that claims in the opposite direction have also been made, as the paper states.
21 http://www.mediawiki.org/
22 http://www.wikipedia.org/

"capital of" property. The type property may be used to specify the type of properties (their co-domain). SMW provides a formal semantics for its annotations via a mapping to OWL DL. SMW annotations mostly correspond to ABox statements in OWL DL (i.e. they describe individuals); schema (TBox) statements are limited on purpose [126].

It is possible to query the Semantic MediaWiki via direct and inline queries (embedded in pages) expressed in the SMW query language. The syntax of the language is derived from wiki text and its semantics corresponds to certain class expressions in OWL DL [217] which can be characterized [126] as concepts of the description logic $\mathcal{EL}^{++}$ [15], with addition of (Horn) disjunction that does not add to the complexity [125]. Note that $\mathcal{EL}^{++}$ is a fragment of so called Horn description logics. Horn description logics are suitable [125] for reasoning with large ABoxes which is aligned with the focus of the Semantic MediaWiki.

As already noted before, Bao et al. [16] specified an alternative minimal Herbrand model semantics for SMW and found that the entailment problem is NL-complete in SMW and that query answering is P-complete in general in SMW and it is in L if subqueries are disallowed.

2.5.2 *IkeWiki*

IkeWiki[23] is the semantic wiki predecessor of the KiWi wiki and the KiWi semantic platform that is the focus of this dissertation. IkeWiki was originally developed as a tool for collaborative development of ontologies and for knowledge management purposes [198]. The focus of IkeWiki, in contrast to the Semantic MediaWiki, is more on reasoning [198]. It supports OWL DL reasoning and also SPARQL querying via the Jena Semantic Web Framework.[24] Same as SMW, IkeWiki also supports typed links and page presentation according to page types. Editing web pages is done in the Wikipedia wiki text syntax with semantic extensions. Thanks to its full support of OWL DL, IkeWiki can be used also as an ontology engineering tool. IkeWiki is however rather a research prototype and it does not for example provide a simple way of selecting and exporting a specific ontology from its knowledge base. IkeWiki, in contrast to SMW, stores annotations separately from wiki pages. Wiki pages and annotations are combined together on page request in a rendering pipeline.

23 http://sourceforge.net/projects/ikewiki/
24 http://jena.sourceforge.net/

3 KNOWLEDGE REPRESENTATION AND REASONING FOR A SEMANTIC WIKI

This chapter identifies several characteristics important for wikis that should be maintained and even enhanced in semantic wikis. The following chapters describe concepts and methods based on this analysis.

3.1 A SOCIAL SEMANTIC WIKI – AN ANALYSIS AND MOTIVATION

Social software is affluent with user interaction and one of its main goals is to support and facilitate the interaction and collaboration. Any advanced functionality, including reasoning and reason maintenance, built into such software should further boost this defining trait, not hinder it.

This section summarizes distinguishing properties of social software and motivates the general features reasoning, reason maintenance, and explanation should have in a semantic wiki. The focus is mainly on wikis in an enterprise setting in which people seek efficient ways to "get things done" as opposed to note taking without a clear goal. Note that while the focus is on a social semantic wiki in an enterprise setting (a setting of the whole KiWi project), the observations can be generalized and transferred to other social semantic systems too.

3.1.1 *Work in Progress*

The work of knowledge workers is varied in that it does not follow strict guidelines and a large part of such work involves a creative process. Creative processes do not adhere to predefined schemes (ontologies, scenarios) and may involve inadvertent inconsistencies, i.e. they involve and result in "work in progress" which needs to be further refined. Creative thinking requires exploring alternatives, generating and testing ideas, or brainstorming to get started. *A social semantic wiki should support creative work and should not hinder it by imposing unnecessary predefined constraints.*

It is in the nature of creative process that inconsistencies arise and can even constitute a desirable contrast that can be a source of inspiration. In the ideal case inconsistencies should be tolerated and even automatically resolved. This is however not realistic. For example logic itself does not provide information how to resolve inconsistencies in a theory; all formulas are "equally important" for logic. For this very reason, the field of belief revision [8] introduces the concept of entrenchment which in effect helps to resolve inconsistencies by expressing that a formula "is more important than" some other formula (see Section 8.10 for more about belief revision). *A social semantic wiki should support users in resolving inconsistencies.*

A system can support users by drawing attention to inconsistencies and by providing additional information about them. Additional information can include information about the origin and users and facts involved in derivation of the inconsistencies. In short, the system should be able explain inconsistencies and highlight their culprits. *A social semantic wiki should provide tools that help users define and resolve inconsistencies.*

3.1.2 *Emergent Collaboration*

In an enterprise environment, collaboration is directed towards a common goal. A free unconstrained collaboration where virtual teams form spontaneously based on need is one of the main distinguishing features of social software. However, as McAfee remarks [144], it is naive to only provide users with a new software and expect that everyone will start using it and content will simply emerge. Incentives are still needed on the Internet and even more so in an enterprise environment. *A social semantic wiki should facilitate joint work as much as possible and provide further incentives for active collaboration.*

People work together by sharing and comparing their viewpoints which can frequently be in conflict. Conflicts can be seen as mutually inconsistent views that need to be settled. This is not to mean that conflicting ideas are not desirable. The same way as inconsistencies can drive a creative process, disagreements can be a means of creative problem solving [106]. It has been shown [92] that collaborative reasoning (problem solving) can be more efficient than individual reasoning. While the study in [92] presupposes a live environment where people gather together in a room and talk together, communication can also happen vicariously in a web-based social platform. In fact, another study [128] indicates that less can sometimes be more with respect to means of communication and problem solving:

> The most unexpected result of these experiments was that the participants who were limited to communicating with text-only explored three times as many ideas as those using graphic tools. Examination of the protocols suggested that the participants using the graphic tools fixated on solutions and gave themselves to exploring the early solutions and often engaging in lower level, even trivial, design decisions while ignoring the opportunity to explore more widely. Consequentially, the students using graphic communication explored the problem space less thoroughly than those using text. [128]

A social semantic wiki should support users in expressing agreements and disagreements explicitly as well as to provide them with tools to discover and explain disagreements in order to facilitate reconciliation.

3.1.3 *Knowledge Representation and Reasoning the Wiki Way*

Knowledge representation and reasoning should be easy to grasp and it should fit well with the idea of "wikiness." Traditionally, knowledge is represented in complex formalisms by experts cooperating with domain experts. Such an approach is typically neither possible nor desirable in the usual enterprise. *A motivated casual user should be able to semantically enrich content.*

Two ways of enriching content with semantics can be distinguished: automated annotation and manual annotation. Automated annotation has been mainly the domain of information extraction. Manual annotation has a long tradition in information science. Both approaches can be combined to a certain extent to form a semi-automatic annotation system. *Annotation should be as simple and as supported by the system as possible.*

A fully automated annotation is not realistic given the state of the art annotation techniques. Therefore *users should be able to create gradually more formal annotations as they become familiar with the system.*

3.1.4 *From Informal to Formal Knowledge Representation*

Traditional approaches to knowledge management impose a rigid prior structure on information. In contrast, social software enables phenomena such as social tagging that leads to emergent creation of folksonomies. The word *folksonomy*, a portmanteau word made up of folk and taxonomy, is used here to mean a user-generated taxonomy. A folksonomy is implicitly present in a bag of tags and can be extracted and represented the way traditional taxonomies are represented. Advantages and disadvantages of folksonomies compared to taxonomies and ontologies have been discussed before [209, 215, 211].

A traditional taxonomy limits the number of dimensions according to which distinctions can be made while folksonomies naturally include many dimensions[209]. Therefore different taxonomies and even ontologies can be extracted [154, 178] from a bag of simple tags using specialized algorithms based on data-mining and information extraction methods. Natural language processing is usually necessary because tags usually are simple words without structure that would provide clues about their relationships.

Informal annotations such as simple tags are easy to use but offer low expressiveness. In contrast, formal annotations, e.g. OWL, allow for greater expressiveness and better accuracy of annotation in exchange for significantly lower user-friendliness. A semi-formal annotation formalism that would e.g. provide more cues to automated information extraction methods seems to be desirable but lacking. While there already are proposals in this direction [18, 17], most are a variation of keyword-value pairs and in some cases RDF triples [229].

3.1.5 *Negative Information*

In real life, people often work with negative information. It is important to be able to express that something is not something else, something is not possible, it is excluded, or perhaps not allowed. An information about a project's approval is as important as information about its rejection and it is common knowledge that a project cannot be both approved and rejected at the same time. *A social semantic wiki should provide a way to express and handle negative information.*

3.1.6 *Inconsistency Tolerant Reasoning with Semi-formal Knowledge*

Let us assume that a company follows a process of denoting projects as risky and not risky – for example by tagging the respective project descriptions in a wiki. A manager trying to determine potential problems with projects may be interested in finding disagreements about riskiness of a project which in this case can be recognized by inconsistent taggings. A disagreement or an inconsistency may not be so direct and it may follow from a set of rules that is assumed to be correct. It is desirable that reasoning can tolerate inconsistencies and allows for pointing out incorrect assumptions. *A social semantic wiki should employ inconsistency tolerant reasoning.*

3.2 KNOWLEDGE REPRESENTATION AND REASONING WITH A SOCIAL PERSPECTIVE

Let us first summarize the essence of requirements on knowledge representation and reasoning for a social semantic wiki observed in the previous section.

Knowledge representation in a social semantic wiki:

- should not impose predefined ontologies,
- should allow for free tagging,
- should allow for gradually more formal annotation,
- should allow for expressing negative information,
- should allow for expressing agreements and disagreements,
- should provide means for describing inconsistencies.

Reasoning in a social semantic wiki:

- should be simple, understandable, and explainable,
- should be inconsistency tolerant,
- should enable processing negative information.

The requirement on simple, understandable, and explainable reasoning translates to four more explicit requirements on reasoning in this dissertation, namely: categoricity, intuitionistic reasoning, two-valued logic, and finitism. Let us shortly review them.

CATEGORICITY The reasoning should be categorical in the sense that for each possible information (formula) F, either F or $\neg F$ should be provable in order to exclude "disjunctive information." Even though disjunctive information may be useful for some applications, it basically conveys an "alternative worlds" semantics which is not the most widely assumed when annotating. Furthermore, the restriction to categorical reasoning simplifies explanations.

INTUITIONISTIC REASONING RESTRICTIONS Intuitionistic reasoning requires witnesses of existence in proving existential statements which makes it rather intuitive and simple. For example $(\exists x)F$ can only be proved if a value for x is shown for which F holds, and to prove $F_1 \vee F_2$, one of F_1 or F_2 must be proven. It excludes both refutation proofs and some forms of reasoning by cases that, arguably, are not familiar to everyone and therefore might be difficult to convey in proof explanations. It is worth stressing that the choice of a sort of intuitionistic reasoning is not philosophically motivated. It is a pragmatic choice aiming at simplifying reasoning and therefore explanations.

TWO-VALUED LOGIC A two-valued logic is desirable in order to fit well with a widespread common sense. As a consequence, fuzzy reasoning is not directly expressible and inconsistency tolerant reasoning building upon specific truth values for conveying inconsistency is also excluded.

FINITISM Finite numbers of both individuals and derivable minimal statements are assumed. This assumption is safe as many applications of social software hardly will require deductions yielding infinitely many implicit atoms. The question whether applications with possibly infinite atom generation might be meaningful is not further addressed in this dissertation. It suffices that applications limited to finitely many atoms are likely to be frequent.

The following chapters describe reasoning and reason maintenance methods that comply with these requirements. In particular, we suggest a constructive inconsistency tolerant reasoning based on KWRL (the KiWi Rule Language) rules, see Chapter 6. The proposed rule language provides two kinds of rules: constructive rules and constraint rules. Constructive rules describe how new information is derived from explicit information while constraint rules are rules that derive a special "inconsistency symbols" and thus enable description of constraints or contradictions. With range restricted (see Definition 17 in Section 7.1.1) constructive rules only, one can build up a categorical theory amounting to a set of positive variable-free atoms, that is up to the syntax a relational database. With constructive rules that are not range-restricted, more general statements can be made, namely universally quantified statements.

The requirement for a simple user-friendly system translates to the choice of forward chaining reasoning (or materialization) in this thesis. Materialization has the advantage that all information deducible from rules and explicit information is readily available which is likely to have a positive effect on user experience. Chapter 7 revisits forward chaining and describes several novel forward chaining methods that also derive information desirable and necessary for explanation.

Materialization, however, puts high demand on efficient handling of updates of explicit information. To this end, methods for handling such changes are explored in detail in this dissertation too. Chapter 8 describes several novel reason maintenance methods for updating materialized facts some of which can also serve as methods for explanation and reason maintenance.

The next chapter presents a conceptual model of the KiWi semantic wiki.

4 A CONCEPTUAL MODEL OF A SEMANTIC WIKI

This chapter presents a conceptual model of a semantic wiki from a user's point of view and with consideration of the analysis of the previous chapter. Note that there has been little research on the principles underlying wikis, their architecture and design [210].

It is necessary to know what the user-level concepts of a system are in order to design the system properly. This section reviews the basic concepts a user interacts with within a semantic wiki. It provides a common ground for the rest of this dissertation.

The conceptual model described here is not fully implemented in the KiWi 1.0 system which does not include concepts such as annotated links, negative and structured tags, and the assignment of multiple labels to a single tag concept.

4.1 CONTENT

This section outlines representation of content in the KiWi wiki. "Content" here refers to text and multimedia which is used for sharing information, most frequently through the use of natural language, between users of the wiki, and whose meaning is not directly accessible for automatic processing. Information Extraction techniques enable computerised processing of structured data extracted from text or speech, but this introduces another level of representation which is not considered "content" in this sense.

4.1.1 *Content Items*

Content items, the primary unit of information in the KiWi wiki, are composable, non-overlapping documents. Every content item has a unique URI and can be addressed and accessed individually. As a consequence, there is no inheren distinction between wiki pages and content items. Rather, by default, all content items are wiki pages.

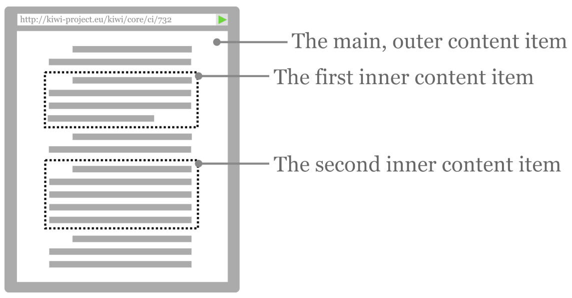

http://kiwi-project.eu/kiwi/core/ci/732
The main, outer content item
The first inner content item
The second inner content item

Figure 4.1: An example of content item nesting. The URL in the address bar is that of the main, outer content item.

An atomic textual content item can be thought of as being similar to a paragraph or a section in a formatted text in that it contains content but it also structures the text. A content item can directly contain only one type of content, for example text or video. However, content items can be nested through transclusion[1] [166] to enable the representation of complex composite content structure. Consequently, a content item may contain its textual or multimedia content and any number of inclusion declarations, both of which are optional.

Having an explicit concept of content structure in a wiki is desirable both with respect to the semantic as well as the social nature of a semantic wiki; the structural semantics of the content can be immediately used for querying and reasoning, for example for automatically generating tables of contents through queries, as well as for facilitating collaboration and planning of content. In addition, content items constitute a natural unit for assigning annotations for content (see Section 4.2).

Content items can be multiply embedded in other content items. This means that content items can be easily reused and shared, which is useful for example for schedules, contact data, or guidelines. If only a copy of a content item's content is embedded, multiple occurrences of the content item in the wiki cannot be traced as naturally or easily and data, for example changes of schedule or email address, have to be updated in multiple locations. On the other hand, updating a multiply embedded content item or reverting it to an earlier version can lead to unintuitive and undesired effects when the content item changes in all contexts it is embedded in without the editing user being aware of all these contexts. Therefore, upon modifying a content item that appears in several different locations, the user is presented with a list of the embedding locations and the choice to edit the content item or a copy its content.

Loops arise when a content item contains itself as a descendant through multiple embedding. The resulting infinite recursion is problematic with respect to the rendering of the content item[2] as well as reasoning and querying over it. Since such loops arguably have no straightforward meaningful interpretation in the wiki context, transclusions which would cause loops are generally forbidden. However, it is generally possible to embed a content item several times into another content item or one of its descendant content items as long as the embedding does not give rise to a loop.

Assuming that all links to content items included through transclusion have been resolved, the wiki content can be seen to consist of a set of finite trees. Root nodes, that is, content items that are not contained in another content item, then have a special status in that they encompass all content that forms a cohesive unit. In this, they can be seen as being alike to a wiki page in a conventional wiki.

EXTERNAL CONTENT ITEMS Linked websites that are located outside of the wiki are considered to be external content items. That means, they can be tagged and they can contain non-intrusive fragments (see below), but they are not considered to be complex, that is, nested.

4.1.2 *Fragments*

Fragments are small continuous portions of text (or, potentially, multimedia) that can be annotated. While content items allow the authors to create and organise their documents in a modular and structured way, the idea

1 Inclusion by reference.
2 At least if we assume that all of the content item is to be rendered at once.

behind fragments is that they enable the annotation of user-defined pieces of content independently of the canonical structure. If content items are like chapters and sections in a book, then fragments can be seen as passages that readers mark; they are linear and transcend the structure of the document, spanning across paragraphs or sections. Different sections of the book might be marked depending on which aspect or topics a reader is interested in.

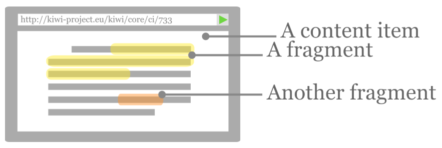


Figure 4.2: An example of fragments of a textual content item.

Fragments should be maximally flexible in their placement and size to allow for different groupings. Towards this goal, it is generally desirable that – unlike content items – fragments can overlap. The intersection between two overlapping fragments can either be processed further or it can be ignored. When two overlapping fragments f_1 and f_2 are tagged with *"a"* and *"b"* respectively, a third fragment that spans over the overlapped region and is tagged *"a, b"* can be derived automatically. Similarly, automatically taking the union of identically tagged overlapping or bordering fragments might be intuitive and expected by the user. However, this automatic treatment of fragments is a complex issue which might not always be appropriate or wanted.

Therefore, fragments are seen as co-existing but not interacting, meaning that relationships between fragments are not automatically computed and no tags are added. This view has the advantage of being simpler and more flexible in that control of fragments and their tags stays with the user. It is also in tune with the philosophy that, unlike content items that always only realise one structuring, fragments are individual in that different users can group a text in many different ways and under many different aspects.

Fragments can either be restricted to be directly contained in one content item, or they can span across content items. In the latter case, a rearrangement of content items can lead to fragments that span over multiple content items which no longer occur in successive order in the wiki and, similarly, transclusion means that content items may contain only part of a fragment with the other part being absent (but present in some other content in which the content item is used).

To avoid these problems, in the KiWi wiki, fragments start and end in the same content item, but can span over contained content items. One single content item then contains the whole fragment. More precisely, this means that fragments can span over different content items as long as they have a common parent content item and all intervening sibling content items are part of the fragment, too.

There are two possible ways of realising fragments:

- "intrusive": the insertion of markers in the text to label the beginning and the end of an fragment, and

- “non-intrusive”: external referencing of certain parts of a content items, using for example XQuery[3], XPath[4], XPointer[5], or line numbers.

As the former means that fragments are less volatile and updates to the text do not affect fragments as easily, for example when text is added to the fragment, fragments in the KiWi wiki are intrusive.

4.1.3 *Links*

Links, that is simple hypertext links as in HTML, can be used for relating content items to each other and to external resources. Links have a single origin, which is a content item, an anchor in this origin, and a single target, which is a content item or external URI. Links are primarily used for navigation but can also be considered a kind of annotation that specifies relation between the two linked resources. For an untyped link, this relation may default to the “is related to” relation. Typed links express a specific relation between two resources. The link structure within a wiki can therefore be seen as a (possibly cyclic) directed graph with typed edges.

4.2 ANNOTATION

Annotations are meta-data attached to content items, fragments, and links which convey information about their meaning or properties. Annotations can be created by users either manually or via rules. Content items and annotations also include system meta-data such as the creation and last edit date and the author(s) of a content item or an annotation. These meta-data are automatically generated and cannot be modified by the user. The KiWi wiki comes with pre-defined application independent vocabularies, taxonomies, and ontologies expressing authorship, versions, and the like. This is not further developed here as it is not in the focus of this dissertation. Note that meta-data and annotations have to be clearly distinguished. Especially in informal speech meta-data and annotations are often not distinguished and one talks about a tag of a content item while a tag *annotation* may be meant. This distinction is important for proper modelling of annotations and for example negative tags, see e.g. “Representing Tags in RDF” below.

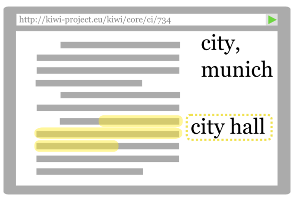


Figure 4.3: A content item tagged “munich” and “city,” and a fragment tagged “city hall.” Fragments are assumed to have URIs. Tags are attached to fragments using the URIs.

Two kinds of user generated meta-data are available in KiWi 1.0: tags and RDF triples. Tags allow to express knowledge informally, that is, without

3 http://www.w3.org/TR/xquery/
4 http://www.w3.org/TR/xpath/
5 http://www.w3.org/TR/xptr-xpointer/

having to use a predefined vocabulary, while RDF triples are used for formal knowledge representation, possibly using an ontology or some other application-dependent predefined vocabulary. Regular users are generally not confronted with RDF, which is considered an enhancement for experienced users. Here we contribute so called "structured tags" that allow for a smooth transition between informal and formal meta-data.

4.2.1 *Formal Annotations: RDF and OWL*

The Resource Description Framework (RDF, see Section 2.4.1) is currently the most wide-spread knowledge representation format on the semantic web. It enables interoperability of semantic web systems to a certain extent. Even though RDF is easily machine processable, the use of RDF alone does not guarantee full interoperability as the same RDF vocabulary can have different intended meanings in different contexts. Still, RDF provides a common ground for the semantic web and therefore it has a prominent place in the KiWi system too.

Vocabularies and richer structures such as taxonomies, thesauri, and ontologies (referred to as "terminologies" in the following) can be defined using RDF Schema or the Web Ontology Language (OWL, see Section 2.4.3). KiWi 1.0 employs both RDF(S) and OWL but only a subset of OWL reasoning is supported, namely the OWL 2 RL profile the semantics of which can be expressed using rules. Although both RDF and OWL formalisms provide a human-friendly syntax, they are not always intuitive and easy to write or read; their perhaps the main advantage lies in providing a common ground for semantic web systems and in enabling reasoning. An example of an RDF annotation is `kiwi:item/123 rdf:type kiwi:ContentItem`. Note that a triple like this is often called an annotation on the Semantic Web. With respect to our conceptual model, the triple is a piece of meta-data that can be attached to a resource for example by means of reification. The whole structure is then called an annotation.

A semantic wiki can translate flat, structured tags, and links to an RDF(S) or OWL representation using predefined terminologies in order to make them accessible to semantic querying and reasoning.

4.2.2 *Informal Annotations – Tags*

One problem that frequently arises in the context of semantic web applications that use relatively complex formalisms such as RDF for annotation is that it is hard to motivate users to annotate content since they find the process complicated, laborious, and time-consuming. In addition, predefined terminologies are not ideally suited for capturing emerging knowledge as they constitute a top-down, single-viewpoint approach to describing domain concepts and their relations. They do not and cannot account for newly arising concepts whose definition and classification might initially be vague. Social semantic software also needs less formal and more flexible modes of annotation.

Tagging is one such kind of informal or semi-formal (depending on the exact use and implementation) annotation. Tags normally consist only of keywords users associate with resources. Despite their simplicity, there are many possibilities as to how exactly tags should work and be used [204].

The ability to tag content is provided by many social websites where users can assign tags to content, for example bookmarks or pictures, thus creating groups of items that share similar characteristics, namely a relation to the tag which serves as the label for that group. Through this, ordering and retrieval of content is enabled which serves as a further incentive for tagging.

An informal tag is a simple string such as "contentItem" or "animal." In a tagging system, no meaning is assigned a priori to a string, and on the other hand, no specific string is pre-assigned to express a concept. The meaning of a tag can thus be seen to emerge from the collaborative tagging by individual users who do not have to commit to a pre-defined view on categorization. Note that this view is akin to Wittgenstein's language games [226] – meaning of a word emerges from a context and from the way it is used.

As a consequence of this flexibility and accommodation for different points of view, the same concepts are frequently expressed using different tags, for example synonyms, tags in different languages or inflected or simply differently spelled versions of the same word. Similarly, different concepts may be assigned the same string as a tag due to polysemy and homographs. Yet another problem is basic level variation, varying degrees of specificity in tagging depending on the level of expertise in an area. Since no explicit relations between tags exist in a folksonomy, tags referring to similar or identical concepts at different degrees of specificity are not recognized as being related.

These factors have a negative influence on the consistency of the emergent semantics and consequently on both recall and precision as well as users' understanding of the structure of the system. The practical and social aspects are interrelated and mutually reinforcing since users not sure about tagging conventions are likely to break them, leading to more inconsistency in the folksonomy.

As above observations indicate, social and technical factors must be taken into consideration – but can also be leveraged – when supporting the process of emerging semantics in a folksonomy.

The KiWi system employs tagging with advanced features such as URIs on the conceptual level (tags as concepts) and autocomplete and disambiguation on the user interface level. This helps to overcome the downsides of uncontrolled, ad-hoc categorization and it also helps to enable a transition between informal and formal annotation.

TAGS AS CONCEPTS The KiWi system distinguishes between tags as mere strings or labels and the abstract concepts they stand for by representing tag concepts as content items, tags in our terminology. Each tag is associated with one or more labels which may overlap between tags, thus mirroring the non-unique mapping between strings and concepts as described above. Each tag concept content item may, but does not have to, contain a textual description of its meaning and a list of resources it has been assigned to, thus making it easier for users to negotiate the meaning and use of the concept.

When a user enters a tag to be assigned to a resource, the system automatically resolves it to the corresponding tag concept, allowing the user to intervene if she disagrees with the concept association that occurs in the case of ambiguity. Various approaches for semantic disambiguation, for example based on co-occurences of concepts and distances between words in Wordnet[6], exist and can be used. Each tag concept's content item can be used as a tag label, meaning that each concept can be unambiguously addressed during tagging, even when all of its associated labels are ambiguous. This is especially relevant for the automatic assignment of tags where no user-intervention is possible or desired. In summary, a resource in the KiWi wiki is tagged not with a string or label but with a concept to which the label serves as a shortcut.

Besides offering a solution for resolving ambiguous mappings between concepts and labels, tags can also be matched to formal concepts in an ontology, allowing for the ontological grounding of a tag concept.

THE SEMIOTIC TRIANGLE. One question to ask when designing a system that includes annotations is "What is it that is being annotated?" On the purely technical level, this question may have a quick superficial answer: any resource that the system allows to be annotated. But what is the actual meaning? Let us say that the resource is a content item about an elephant. Does a tag added to the page state a fact about the page itself (a representation of an elephant in the wiki system) or does it refer to the actual elephant? This kind of analysis leads to a concept well-known as the semiotic triangle [170], Peirce's triad[177] or de Saussure's distinction between the signifier and the signified [195]. The distinction is important because it may have consequences on how annotations are interpreted and treated. In [172], the authors let the users explicitly express whether an annotation refers to a text or its meaning by providing them with a syntax that allows the users to distinguish between these two cases.

The KiWi system addresses the problem of multiple meanings of a tag label as described above. It, however, purposefully does not address the problem of what a tagging refers to – to a concept or the page that describes it. The reasons for this are that such a distinction is likely to be insignificant in most cases with respect to the practical use of tags, irrelevant to many users and, above all, forcing users to be aware of the distinction and employ it during tagging reduces the user-friendliness of tagging and thus deters users from annotating content. In summary, introducing such a distinction would be likely to lead to an unnecessary increase of complexity of the tagging process.

TAG LABEL NORMALIZATION Tag normalization defines an equivalency on the set of tag labels. Tag label normalization can mean for example that the tag labels "Wifi", "WiFi", "Wi-fi" and "WIFI" are all equivalent to the tag label "wifi" which can be the canonical form of this tag label equivalency class. The canonical form is shown in aggregated views such as tag clouds. Trailing whitespace and multiple whitespace within a tag are always removed upon saving the tag. In other cases, each user always sees the tag label that he or she entered. Further, tag label normalization in KiWi is done by converting all letters to lowercase and removing punctuation.

NEGATIVE TAGS In a collaborative context, one may be interested in tracking disagreements which presupposes some way to express negative information. Just as a user can tag a resource with tag "t" he or she may want to tag it with "–t" (meaning "not t") as a way to express disagreement

6 http://wordnet.princeton.edu/

or to simply state that the resource is not "t" or does not have the property "t." An example may be a medical doctor tagging a patient's card as "–lupus" to state that the patient definitely does not have "lupus."

Although a tag "–t" could be seen as introducing classical negation into the system, it may in fact be only a very weak form of negation because we can allow negating only pure tags, not general formulae (or sets of tags), and the only way to interpret this kind of negation would be by introducing a rule which expresses that from tag "t" and tag "–t" a contradiction symbol should be derived. See Chapter 6 for a more thorough discussion of negative tags.

TAGS AND CONTENT STRUCTURE. In presence of nested content items and fragments, it is a question whether annotations should be propagated from items and fragments to containing content items or the other way around. Propagation may be appropriate in cases where the parent-child relationship corresponds to a "isA" or "isInstanceOf" between concepts which the content items describe. In contrast, when the parent-child relationship corresponds to an "isPartOf" relation then tag propagation may be not appropriate. The content structure is accessible to the rule and query language in form of metadata. Therefore, this functionality is best left up to users who can tailor it to their needs by defining approriate rules.

TAG HIERARCHIES. Tag hierarchies constitute a step in the transition from informal to formal annotation. They are useful for example for reasoning and querying since they enable the processing of tag relationships. Tag hierarchies could be created through "tagging tags", that is, tagging a tag's content item to indicate an "is-a" relation.

REPRESENTING TAGS IN RDF. A simple tag (i.e. we mean a piece of meta-data here, not an annotation in our terminology) may be represented in RDF as the triples `tagUri rdf:type kiwi:SimpleTag` and `tagUri kiwi:hasLabel label` where `tagUri` is a unique URI of the tag and `label` is a string literal with a normalized label of the tag. Original labels used by users for the tag are then modelled as a property of the respective annotations (item-tag relationships) if they are to be preserved. We suggest that the polarity of a tag is also a property of an annotation rather than of the tag. The advantage of such modelling is that for one tag concept there is only one tag URI and not two (one for the positive concept and one for the negative concept), i.e. annotations using the tag concept (positively or negatively) can be identified more directly.

Also note that the KiWi 1.0 system represents tags and tag annotations using the Holygoat tag ontology.[7]

4.2.3 *Semi-formal Annotations – Structured Tags*

Ordinary flat tags are limited in their expressiveness. To overcome this limitation, different extensions of tagging are currently being proposed: machine tags[8], sub-tags [18] as used in the website `http://www.rawsugar.com/`, structured tags [18, 17]. Most proposals are a variation of keyword-value pairs, in some cases RDF triples [229]. One approach also considers ratings

7 `http://www.holygoat.co.uk/owl/redwood/0.1/tags/tags.n3`
8 `http://tech.groups.yahoo.com/group/yws-flickr/message/2736`

on tags [91]. Note that keyword-value pairs can be seen as triples too – the resource being annotated is the subject, the keyword is the predicate and the value is the object of the triple. More complex schemes which involve nesting of elements might be practical in some cases, e.g. "hotel(stars(3))" could express that the tagged resource is a three-star hotel. These extensions develop the structure of the tag itself and a set of tags is interpreted as a conjunction. It is conceivable to allow users to tag resources with a disjunction of tags or even with arbitrary formulae. This may be practical for some applications but it has two drawbacks: reasoning with disjunctive information is difficult and simplicity and intuitiveness would suffer (see also the requirement for categoricity in the previous chapter).

Structured tags enhance the expressive power of tags and serve as an intermediate step between informal (unstructured) tags and formal annotations.

Two basic operations lie at the core of structured tagging: grouping and characterization. Grouping, denoted "( )", allows to relate several (complex or atomic) tags using the grouping operator. The group can then be used for annotation. Example: a wiki page describes a meeting that took place in Warwick, UK on May 26, 2008, began at 8 am and involved a New York customer. Using atomic tags, this page can be tagged as "Warwick", "New York", "UK","May 26", "2008", "8am" leaving an observer in doubts whether "Warwick" refers to the city in UK or to a town near New York. Grouping can be used in this case to make the tagging more precise: "(Warwick, UK), New York, (May 26, 2008, 8am)". The annotation of a group of tags assigns only the higher unit made out of the individual atomic tags or group, but not the separate constituents. If this functionality is desired, it could be implemented for example via rules. A group of tags can also be used to describe the properties of something whose name is not yet known or for which no name exists yet.

Characterization enables the classification or, in a sense, naming of a tag. The characterization operator, denoted ":", can be used to make the tagging even more precise. For example, if we wanted to tag the meeting wiki page with a geo-location of the city Warwick, we could tag it as "(52.272135, -1.595764)" using the grouping operator. This, however, would not be sufficient as the group is unordered. Therefore we could use the characterization operator to specify which number refers to latitude and which to longitude: "(lat:52.272135, lon:-1.595764)" and later perhaps specify that the whole group refers to a geo-location: "geo:(lat:52.272135, lon:-1.595764)". Similarly, Warwick in our example could be expressed as "location:(warwick)" to differentiate it from Warwick fabric or person with the last name Warwick.

Together, grouping and characterization provide a powerful tool for structuring and clarifying the meaning of tags. Structured tags are a step between informal simple tags and formal RDF/S annotations.

The meaning of a structured tagging rests, for the most part, with the user who specified it. Structured tags do not impose rules on their use or purpose, they only allow users to introduce structure into taggings. It can be seen as a wiki-like approach to annotation which enables a gradual, bottom-up refinement process during which meaning emerges as the user's work and understanding develop.

Minimal rules are, however, necessary in order to ensure a flexible useful order and to avoid chaos:

- Groups
  - can be used on positive and negative atomic tags, groups, and characterizations,
  - are unordered,
  - cannot contain two equal members, e.g. (Bob, Bob, Anna) and ((Bob, Anna), (Anna, Bob)) are not allowed,
  - can contain arbitrarily but finitely many elements,
  - can be arbitrarily but finitely nested,
  - are identical in meaning to the atomic tag when they only contain one element, i.e. (Anna) is the same as Anna
- Characterization
  - can be used on positive and negative atomic tags and groups,
  - is not commutative, i.e. geo:x is not the same as x:geo.
  - can use atomic and complex tags as a label

Structured tags have to be syntactically correct. That means that for example "Bob:190cm,90kg" is not a valid structured tag.

The same information can be expressed in many ways using structured tags. The above described way of structuring geo-location is only one of several. Others include:

- geo:(x:y):(1,23:2,34)
- geo:(y:x):(2,34:1,23)
- geo:(1,23:2,34)
- geo:1,23:2,34

and possibly many more. Users are free to choose what suits their needs and purpose the best. Of course, for structured tags to be useful in a community, the community needs to agree on a common way of structuring information. It is important that only a very minimal structure is required by the system leaving the possibilities open; different users and different communities can agree on different ways of structuring the same information. This heterogeneity is an advantage. First, it provides users with freedom and second, it can be beneficial for different communities to encode similar kinds of information in different ways as the precise meaning of a similar concept may differ. In such a case, different encoding may facilitate automatic translation of structured tags to formal annotations because it would allow to distinguish the two concept structurally.

Introducing structure into tags increases their expressiveness and provides a means of limiting ambiguity of the intended meaning as shown for example by the above geo-location example. Note that uniqueness of meaning cannot be asked for. Even a theory of first order logic can have many different models while only one is the intended one. One example of such a theory is the Peano axioms – natural numbers are the intended (standard) model of the theory but it also has non-standard models. Also, it is for example impossible to define a transitive relation in first order logic. This should not be seen as a flaw of the theory but as a limit of the expressiveness of the formalism.

REPRESENTING STRUCTURED TAGS IN RDF. A structured tag may be represented in RDF using the following scheme. Three axiomatic triples defining the relationship of simple tags, structured tags, and grouping and characterization are assumed:

- `kiwi:TagGrouping rdfs:subClassOf kiwi:StructuredTag`
- `kiwi:TagCharact rdfs:subClassOf kiwi:StructuredTag`
- `kiwi:SimpleTag rdfs:subClassOf kiwi:StructuredTag`

Members of a structured tag of type `kiwi:TagGrouping` are described by the `kiwi:hasMember` property with `kiwi:StructuredTag` range.

The left and the right side of a structured tag of type `kiwi:TagCharact` is described by the `kiwi:hasLeft` and the `kiwi:hasRight` properties both of range `kiwi:TagGrouping` ∪ `kiwi:SimpleTag`. Both properties have to be used exactly once.

Note that this representation does not allow negative tags in structured tags. Negative tags may be useful to some: "(red, nice, –sweet)" but their meaning is not clear in general. Consider the following examples: "sweet, (red, nice, –sweet)", "(sweet, –sweet)", "apple:(ripe:sweet, crude:–sweet)", "–crude:sweet", etc. See also the following section on representing *annotations* in RDF.

STRUCTURED TAGS FROM A PSYCHOLOGICAL PERSPECTIVE. The idea of the grouping and characterization operators of structured tags is rooted in the observation of basic human skills – skills like naming and categorization that are innate to humans and as such are very natural and easy to do. This kind of observation has been first described perhaps in Gestalt psychology. Gestalt psychology developed as a theory of visual perception and it is based around the principle that people tend to perceive holistic patterns rather than individual pieces following so called gestalt laws of closure, proximity, similarity, symmetry, continuity, and common fate. This tendency can be seen as a tendency to group things according to a kind of regularity (a gestalt law). Gestalt psychology has been criticized for being merely descriptive [41] but it nevertheless provides valuable observations about natural human inclinations and aptitudes and it is often used in design and human-computer interaction research and development. More recently (in 1973), Eleanor Rosch introduced [191, 190] so called prototype theory as a mode of graded categorization. In a simplified way, it says that people determine categories by means of graded prototypes – examples that to varying extent represent a given category. It is noteworthy that these categorization and naming skill are used even by very young children [98, 99, 163, 29, 32, 137] which strengthens our assumption that grouping and characterization is likely to be well received by users.

Semi-formal annotations described in this section provide a means to transform knowledge from human-only content described in Section 4.1 to machine-processable information. Semi-formal annotations seem to be an important feature of social software because they provide a low-barrier entry point for user participation on enrichment of content with metadata which are machine processable. Users can use gradually more expressive and formal methods of annotation as they become familiar with the system. First, they only create and edit content. Then they can begin using flat tags to annotate content and later perhaps start using structured tags. Advanced users or system administrators can further enhance the metadata enrichment efforts by specifying rules for semi-formal annotations. Of course, the underlying assumption is that users in fact like using structured tags as an annotation formalism. This assumption is closer examined in a user study presented in the next chapter.

4.2.4 *Representing Annotations in RDF*

Annotations are metadata (RDF and OWL, tags, structured tags) attached to content items, fragments, and links. An RDF annotation is an RDF triple attached to an entity, i.e. it must be possible to identify the triple in order to make statements about it. This is standardly achieved by RDF reification[9] and here this standard is assumed. OWL is a vocabulary extension of RDF (see Section 2.4.3) and thus OWL statements can be represented as RDF triples and reification can also be used to express a statement about an RDF-represented OWL meta-data.

We assume that any entity (resource) that a user may want to explicitly talk about has a URI, a blank-node, or that it is a literal. Statements about such entities can be expressed as RDF triples. In particular, it can be expressed that a given entity (except a literal, which cannot be in the subject position) has some metadata attached, e.g.: `entityURI kiwi:hasMetadata metadataURI`. In this case, `metadataURI` denotes an annotation of (the denotation of) `entityURI`. Therefore statements about tags and structured tags can be easily expressed in RDF too. While entering full URIs is inconvenient, disambiguation of tag labels into specific URIs can be done in user interface with user feedback or automatically using linguistic and information extraction methods, see for example [188, 181, 169, 7]. In-depth treatment of disambiguation is out of the scope of this dissertation.

An annotation is described by at least the following properties:

property	object
`kiwi:metadata`	a tag, a structured tag, or an RDF triple
`kiwi:annotatedEntity`	the annotated entity
`kiwi:hasAuthor`	an author of the annotation
`kiwi:dateCreated`	the date of creation of the annotation
`kiwi:marker`	one of `"user"`, `"rule"`, `"system"`
`kiwi:polarity`	one of `"-"`, `"+"`

Additional properties may be used as appropriate, for example `kiwi:dateModified` for annotations that may be changed. The property `kiwi:hasAuthor` may be used multiple times if there are multiple authors of the annotation. No restriction is put on the range of the `kiwi:hasAuthor` property enabling thus an author to be a user, an automated annotating system, or some other entity. Properties `kiwi:metadata`, `kiwi:annotatedEntity`, `kiwi:dateCreated`, `kiwi:marker` have to be used exactly once. The property `kiwi:polarity` can be used to specify the polarity of an annotation. If the polarity property is not used it is assumed to be positive.

9 http://www.w3.org/TR/rdf-mt/#Reif

4.3 social content management

To facilitate social collaboration and leverage the social aspects of the semantic wiki, several options and aspects have to be considered.

groups. User groups can be used among other things for personalisation of wiki content, for querying and reasoning and to attribute wiki data to a group. Tags are an easy way to group things which is used in the wiki, so it is an obvious choice to form user groups by tagging users' content items. Groups are created by assigning at structured tag with the label 'group' to a content item, fragment or link, e.g. "group:(java)"

access rights. Users, user groups and rules for reasoning could be used to handle access rights in the wiki, but this is a complex issue which requires further investigation. Questions that arise include who owns the rules and what are the access rights on rules and who can assign the tags that restrict the access. Static rules would not be suited for rights managements in all environments. For them to function well, the organization and roles in the wiki have to be relatively stable, which may be the case in professional applications. In other areas, such as the development of open source software, such rules may not be desired or the social organization might not be static enough for rules to be adequate.

the social weight of tags. It is useful to aggregate tag assignments (and annotations in general) to provide a clearer view of them and their popularity. Tag assignments then can be seen to have weights. Some users might not agree with the assignment of a certain tag to a content item, and add a negative component to the tags' weight to express this. The overall social weight of a tag can then be calculated by assigning a value to both a tag assignment and disagreement with it and calculating the total. The social weight of a tag summarizes the users' views on the appropriateness of a specific tag assignment and thus provides a valuable measure that can be used in reasoning and querying. Note that agreeing with a tag assignment is identical to also making it but disagreement is not identical to adding the negative tag since not being content with the assignment of tag does not necessarily imply that one thinks its negation makes an appropriate tag.

Tag weights also give an overview over users' opinions and could help form a consensus on tag usage. Thom-Santinelli et al. show evidence [194] that users have a desire for consistency with other users' tag assignments and are often willing to adopt a tag that they know has been used on the same resource by other users. Weighting tags could further foster this process. To facilitate this, it should be as easy as possible to also assign a tag or to disagree with its assignment.

Finally, reinforcing or disagreeing about tag assignments constitutes a low-barrier activity in the wiki which makes it easy for users to participate.

5 EXPERIMENTAL EVALUATION: STRUCTURED TAGS AND RDF

This chapter presents a user study that compares structured tags to RDF as an established annotation formalism. The study has been conducted in a joint work with Klara Weiand; the quantitative and statistical part of this text is a joint work, the introduction, qualitative evaluation, and discussion were written by me alone. Klara's assessment of the results of the study can be found in her dissertation [223].

Structured tags, introduced in the previous chapter, are a semi-formal annotation formalism that enriches flat ordinary tags with two operators: grouping and characterization. Grouping allows for dividing tags into distinct groups while characterization allows for naming or characterizing of groups and simple tags. The meaning of a structured tag depends on its use and is not predefined. Structured tagging introduces only a minimal set of rather intuitive rules. Arguably (see Section 4.2.3), grouping and characterizing are two elementary concepts learned by all at an early age: who has not seen a three years old child grouping things according to some rule or characterizing objects by enumerating some of their properties?

RDF is a knowledge representation and annotation formalism established on the Semantic Web. RDF is rather simple in that it consists only of triples that resemble simple natural language sentences. It also provides a human-friendly syntax, and it is conceptually simpler than more advanced formalisms such as OWL and first order logic. For this reason, RDF was chosen as an alternative annotation mechanism in this user study. Structured tags are, however, not intended as a replacement of RDF. Instead, they are orthogonal to it in that they provide a semi-formal step between RDF and informal annotation using simple flat tags.

Structured tags are flexible because on the one hand they are based on only two basic concepts, grouping and characterization, yet these two concepts allow for high accuracy, and because they build upon freely chosen atomic tags with an implicit semantics. The simplest form of structured tags are ordinary flat tags that can be combined and related to each other by repeated application of grouping and characterization resulting in sophisticated structures. This flexibility enables capturing knowledge as it evolves. Starting with vague information best expressed as a simple set of flat tags one can gradually create complex and more precise (less ambiguous) descriptions. This feature of structured tags is only partially studied in this chapter. The main focus of this study is on the user-friendliness of structured tags as it is crucial to their acceptance and usefulness in social semantic applications.

This study provides first insights into the practical use of structured tags: users' opinions about structured tags, their ability to capture complex and evolving information in the formalism, the complexity of created structured tags, observations about the suitability of structured tags for expressing concepts such as causality, conditions, and negative information. While other interesting topics such as the transition from structured tags to RDF using (user-defined) rules and other automated means are not evaluated in this study, the results can serve as a valuable basis for further evaluation and implementation of structured tags.

5.1 EXPERIMENTAL SETUP AND EXECUTION

Nineteen participants were recruited through announcements in several computer science lectures at LMU. They each were rewarded with 100 Euros for their participation (during eight hours) in the study.

Participants' ages ranged from 21 to 26 with the average age being 23 years. All of the participants were students. Most studied computer science or media computer science at LMU, but some were students of related disciplines like mathematics, physics and computational linguistics. On average, participants had been students for 5.2 semesters with the individual durations ranging from two to ten semesters.

The study was performed in a single session that lasted about 8 hours and included a half-hour break at noon. Participants were split up into two groups that were balanced in terms of the fields of study and study progress of the group members. All materials that the participants were given were written in English, but comments could be given in English or German. At the start of the session, participants were asked to rate their experience with RDF and of tags on a scale from 1 ("never used/no knowledge") to 3 ("frequent use").

The participants were provided with a short introduction into the experiment scenario, reproduced here:

> You work in a software development company that runs several projects. When a project starts, information about it is added in the company wiki. The text there describes the project, its goals and projects and the people involved in it. Since several people collaborate to write the text and since more information becomes available as the project progresses, the text is changing and is also becoming more detailed. However, the company's wiki does not only contain text but also annotations that describe the content of the text, that is, it is a semantic wiki. These annotations are useful for example for retrieving wiki content and for automated reasoning, the deduction of new facts.

The first part of the experiment consisted of participants annotating different revisions of a text on project management in a software development scenario using RDF. Since structured tags are not yet implemented and available in practice, participants wrote down their annotations to the text on paper. The texts were written specifically for use in the study and represented the content of a wiki page describing a fictional software project. Two different texts of equal length with six revisions each were written, we will refer to them as "text A" and "text B" in the following. All revisions of both texts as they were presented to the participants are given in the Appendix (Chapter A).

To provide participants with an introduction into the annotation formalisms, two texts of similar length describing structured tags and RDF were prepared. These texts are also given in the Appendix (Chapter A). To ensure that the texts were comparable in quality and intelligibility, they were examined by two computer science researchers at LMU who knew both formalisms and were not involved with the study. The description of RDF was simplified in that URIs and namespaces were omitted and capitalization and quotation marks were used to distinguish between classes, instances and literals. The introduction to structured tags did not mention negative tag assignments as described in Section 4.2.2.

Participants were provided with the introductory text on RDF and instructions for the experiment. Participants could refer to the introduction at any point, and were asked not to modify their annotations once they had received the following revision of the text. Changes in the text between revisions were highlighted. Participants were told to add as many annotations as they felt appropriate.The instructions further encouraged participants to refer to annotations made to previous revisions and to describe new annotations in terms of modifications to these annotations.

The annotation process was self-paced and the next revision of the text was only handed out once a participant had finished annotating the previous revision.

In this first part of the experiment, one group annotated text A, while the other group was given text B to annotate. Participants were asked to write down the times when they started and stopped annotating each of the revisions.

After they had completed annotating the final revision, participants were provided with questionnaires where they had to indicate their agreement with ten statements about the annotation formalism and their impressions of the annotation process. For each statement, participants were asked to tick one of five boxes corresponding to a Likert scale representing different extents of agreement. The categories were "strongly disagree," "disagree," "undecided," "agree," "strongly agree."

In addition, participants could optionally provide written comments.

The second part of the experiment was executed in the same manner using structured tags instead of RDF as an annotation formalism. Additionally, the group that annotated text A with RDF was now given text B and vice versa.

When participants had completed both parts of the study, they were asked to fill out a final questionnaire describing which formalism they preferred and for which reasons.

5.2 RESULTS

The analysis of participants' previous experience with RDF and tagging shows that out of the nineteen participants, only five had any knowledge of RDF before the experiment. All five rated their amount of experience with RDF as "a little" and no participants indicated that they used RDF frequently or knew it well.

The situation is similar for tags; all but six participants stated that they had never used tags. Out of these six participants, five indicated that they occasionally assigned tags to web content, and one participant declared that he often uses tags.

5.2.1 *User judgments*

This section describes participants' level of agreement with ten statements about the annotation formalisms. For each statement, two figures are given, one showing the distribution of participants' answers ranging from 1 ("stro-

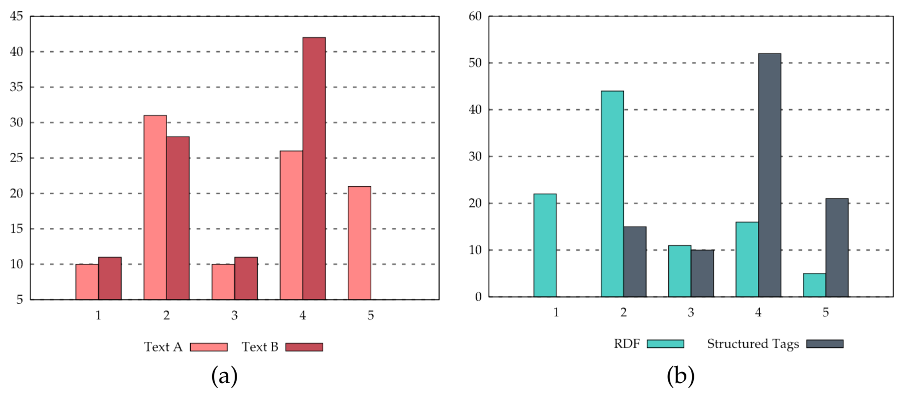


Figure 5.1: Percentages of participants' levels of agreement (1 to 5) with statement 1, grouped by (a) text and (b) annotation formalism

ngly disagree") to 5 ("strongly agree") grouped by the text type, and the second showing the same data grouped by the annotation formalism used. The former allows to determine for each statement whether there is a connection between the text, A or B, and the answer given. The latter shows how answers differ with the annotation formalism used. In addition, we performed Wilcoxon-Mann-Whitney tests to determine whether the differences are significant.

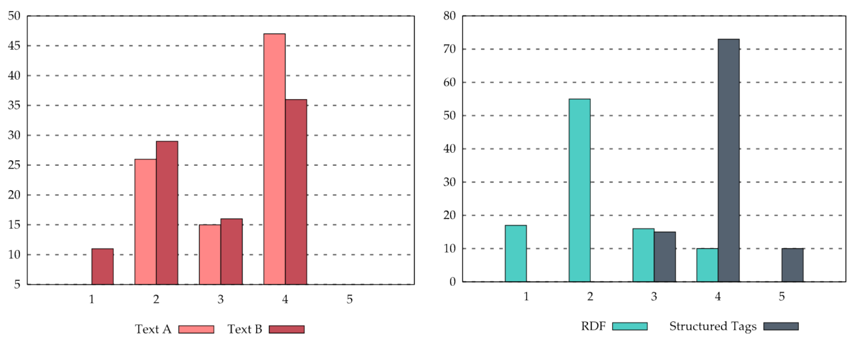
STATEMENT 1: *"After reading the introductory text, I felt I had understood how to use the annotation formalism"* This statement refers only to the introductory text and is independent of the annotated text. Figure 5.1a confirms that the answers are similarly distributed in both texts. The difference across texts was not significant (W=178, p=0.40), but a significant difference in reactions depending on the annotation formalism used was found (W=275, p=0.001).

Participants perceived the introduction on structured tags as being more helpful than that on RDF. Over seventy percent of participants agreed or agreed strongly with the statement with respect to structured tags, but about 65 percent of participants disagreed or disagreed strongly with the statement after they had read the introduction to RDF. In both cases, only about ten percent of participants were unsure whether they agreed with the statement.

Regardless of the annotation formalism, most participants' written comments state that more examples should have been included.

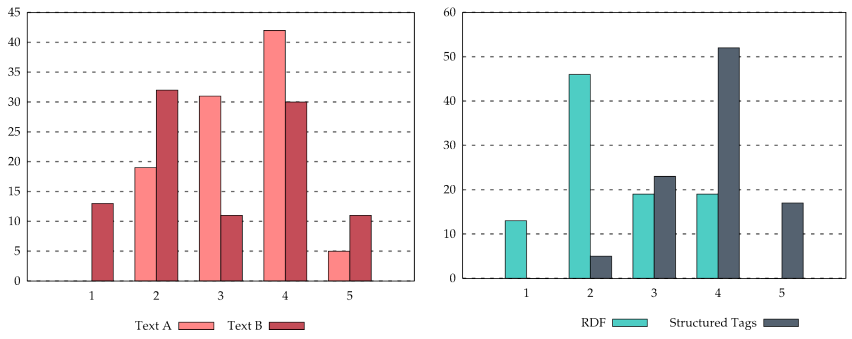
STATEMENT 2: *"The annotation formalism allowed to annotate the text in an intuitive way"* As before, the answers are similarly distributed across the texts (see Figure 5.2a), indicating that participants' opinions were independent of the text they had annotated. Indeed, the difference in answers between annotation formalisms (W=317.5, p<0.001) but not that between texts (W=188, p=0.59) was found to be significant.

More than eighty percent of the participants found structured tags to be intuitive ("agree" or "strong agree," see Figure 5.2b), and a small number was undecided. No participant found structured tags to be unintuitive. The situation is reversed for RDF which almost three quarters of participants



(a) (b)

Figure 5.2: Percentages of participants' levels of agreement (1 to 5) with statement 2, grouped by (a) text and (b) annotation formalism



(a) (b)

Figure 5.3: Percentages of participants' levels of agreement (1 to 5) with statement 3, grouped by (a) text and (b) annotation formalism

found unintuitive. Only about ten percent of participants agreed with the statement with respect to RDF.

Many participants in their comments criticized what they considered to be limitations of RDF, for example that annotations always take the shape of triples, that a clear idea of the domain is needed to formalize the concepts and relations, and that the content of long and complex sentences is hard to express as RDF.

Comments about the intuitiveness of structured tags were mostly positive, although here, too, one participant found it hard to express long sentences as structured tags.

STATEMENT 3: *"The annotation formalism allowed to annotate the text in a convenient way"* The reactions to the third statement differ slightly more between texts than those to the two statements before, but overall are comparable (see Figure 5.3a). Again, the difference between the answers given by participants using different annotation formalisms (W=43.5, p<0.001) but not that between participants annotating different texts (W=158.5, p=0.39) is significant.

Again, with respect to structured tags, most participants agree with the statement, while the majority of participants using RDF disagrees with it (see Figure 5.3b). While the difference between the two formalisms is con-

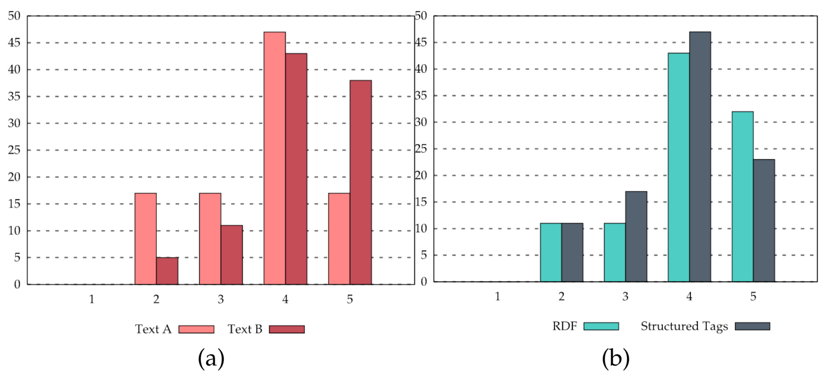


Figure 5.4: Percentages of participants' levels of agreement (1 to 5) with statement 4, grouped by (a) text and (b) annotation formalism

siderable, the reactions are less divided across formalisms than those to the previous statement.

Only few participants added further comments about this statement. One participant found writing RDF triples repetitive, while another remarked that structured tags often need to be rearranged when new information is provided.

STATEMENT 4: *"I feel that a way to represent negative facts is missing from the formalism"* The reactions to this statements are similar across texts, although participants annotating text B slightly more frequently agreed with the statement than those annotating text A (see Figure 5.4a). However, the difference between texts is not significant (W=136.5, p=1), while that between annotation formalisms is (W=85, p=0.05).

There is little difference in the reactions across formalisms (see Figure 5.4b). Overall, more participants feel that RDF is missing a way to represent negative facts, but only by a small margin. For both formalisms, more than 75% of participants agree or agree strongly with the statement and no participants disagrees strongly with it.

In the comments, one participant remarked that he considered a way to express conditions and consequences to be of greater importance than support for negation.

STATEMENT 5: *"The annotation formalism was expressive enough to let me annotate the text the way I wanted"* Here, reactions differ across texts, but less strongly than the reactions across formalisms (see Figures 5.5a and 5.5b) and, again, the difference between annotation formalisms (W=99.5, p=0.26) but not that between texts (W=48.5, p=0.001) is significant.

When using structured tags, over eighty percent of participants found structured tags expressive and agreed or strongly agreed with the statement. The levels of agreement are more varied when participants use RDF with roughly equal proportions answering "disagree," "undecided" and "agree."

STATEMENT 6: *"I feel confident about the formal correctness of the annotations I made"* Participants' answers are very similar across texts (see Figure 5.6a). As before, the difference in answers between texts is not signif-

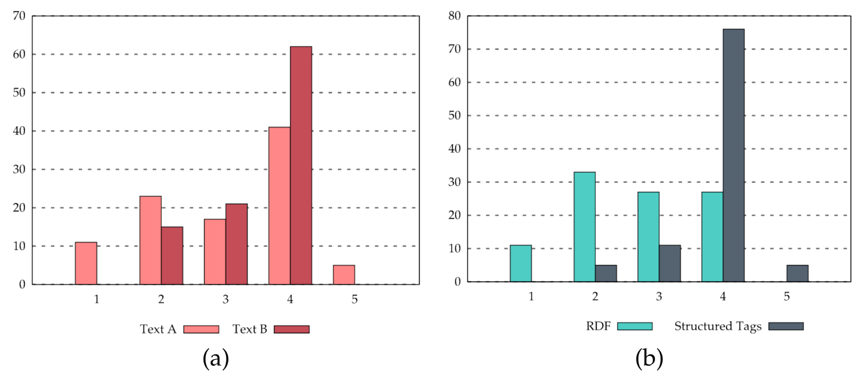


Figure 5.5: Percentages of participants' levels of agreement (1 to 5) with statement 5, grouped by (a) text and (b) annotation formalism

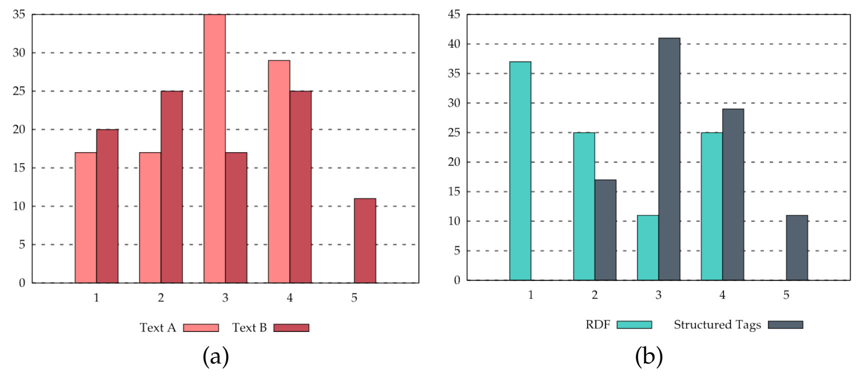


Figure 5.6: Percentages of participants' levels of agreement (1 to 5) with statement 6, grouped by (a) text and (b) annotation formalism

icant (W=130.5, p=0.85) but that between annotation formalisms is (W=69, p=0.014).

More than sixty percent of participants are not confident about the correctness of their RDF annotations ("disagree" or "strongly disagree," see Figure 5.6b), only few are undecided and about 25 percent agree with the statement. The situation is different for structured tags where more than forty percent of participants are undecided, that is, are not sure whether their annotations were formally correct. Another forty percent of participants agrees or strongly agrees with the statement and only about a fifth disagrees with it.

STATEMENT 7: *"I feel confident about the appropriateness of the annotations I made"* Again, the distributions of reactions across texts, while not identical, are highly similar and the difference between texts is smaller than that between formalisms (see Figures 5.7a and 5.7b). Neither the difference between texts (W=117.5, p=0.70) nor annotation formalisms (W=97.5, p=0.24) is significant.

The reactions of participants using RDF are distributed in comparable proportions over "disagree," "undecided," and "agree." A smaller number of participants strongly disagreed with the statement. With respect to structured tags, the reactions are very different with more than half being unde-

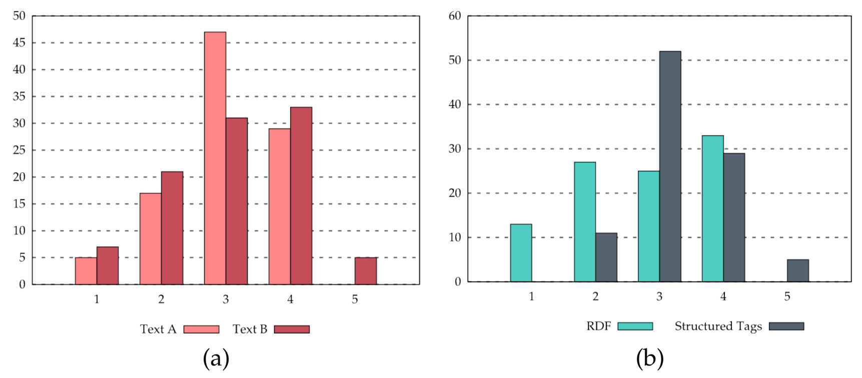


Figure 5.7: Percentages of participants' levels of agreement (1 to 5) with statement 7, grouped by (a) text and (b) annotation formalism

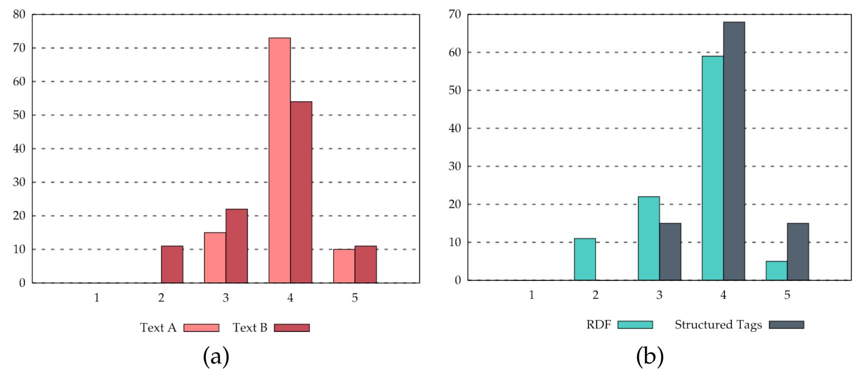


Figure 5.8: Percentages of participants' levels of agreement (1 to 5) with statement 8, grouped by (a) text and (b) annotation formalism

cided, a small number disagreeing and about a third agreeing or agreeing strongly.

STATEMENT 8: *"I feel that the annotations I made convey the important aspects of the text"* For this statement, again, the distributions of the different levels of agreement do not greatly differ across texts or annotation formalisms (see Figures 5.8a and 5.8b), and the answer distributions do not differ significantly between texts (W=200, p=0.30) or annotation formalisms (W=128, p=0.13).

The majority of participants agrees or agrees strongly, while a minority of about twenty percent is uncertain. In the case of RDF, about ten percent of participants disagree with the statement.

Several participants in both groups wrote in their comments that they felt they might have annotated too much and expressed too many unimportant details in their annotations.

STATEMENT 9: *"I enjoyed using the formalism"* Here, a small difference in reactions between the texts can be observed (see Figure 5.9a). The difference in reactions between texts is not significant (W=148, p=0.48), but that between annotation formalisms is highly significant (W=28.5, p<0.001).

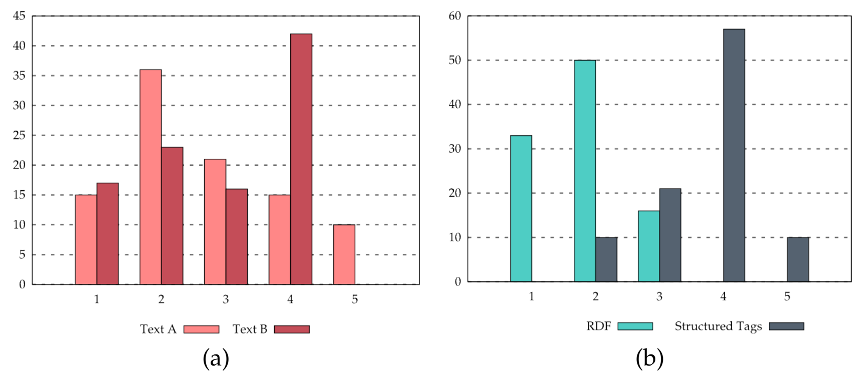


Figure 5.9: Percentages of participants' levels of agreement (1 to 5) with statement 9, grouped by (a) text and (b) annotation formalism

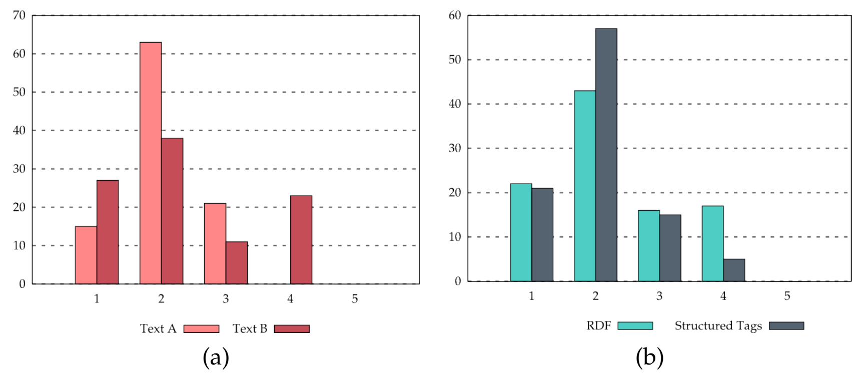


Figure 5.10: Percentages of participants' levels of agreement (1 to 5) with statement 10, grouped by (a) text and (b) annotation formalism

About 24 percent of participants annotating text A and 37 percent of the participants annotating text B disagree with the statement. Inversely, 16 percent of the participants annotating text A and 42 percent of participants annotating text B agree with the statement. However, ten percent of the participants annotating text A but no participants annotating text B agree strongly and the difference between texts is smaller than that between formalisms.

The difference in the distributions of levels of agreement between formalisms is very high for this statement (see Figure 5.9b). Over eighty percent of participants do not enjoy using RDF ("disagree" and "disagree strongly"), and the rest are undecided. No participant indicated that he enjoyed using RDF. The case is reversed for structured tags with over two thirds stating that they enjoyed using structured tags. About twenty percent are undecided and ten percent disagree with the statement.

Several participants commented that they did not enjoy using RDF since they found it too difficult, did not feel that they fully understood the formalism or because they found the process too laborious.

STATEMENT 10: *"Given more time, I would have liked to add more annotations"* Grouped by text, the levels of agreement show a difference in that the reactions of participants annotating text A overall are more negative than those of participants annotating text B (see Figure 5.10a).

A similar difference can be found for the reactions grouped by formalism where more participants using RDF would have liked to add more annotations (see Figure 5.10b). Overall, the differences found for this statement are minor compared to the differences in reactions to the other statements and neither difference is significant (text:W=173.5, p=0.95; annotation formalism: W=107, p=0.61).

In the final questionnaire, eighteen of the nineteen participants stated that they preferred structured tags to RDF. Frequently given reasons given for this choice are that structured tags are more flexible and intuitive, easier to understand and use and quicker to write.

Several participants stated that they thought that structured tags were expressive and could be updated easily, making it possible to start annotating without having a clear idea of all concepts in the domain. The latter point was a major point of criticism with respect to RDF where many participants expressed opinions similar to the following: "[W]ithout prior knowledge of the subjects to be annotated, it is quite hard to define a structure that is compact, yet flexible enough to be used in future edits."

One participant suspected that users would be more willing to write annotations in the form of structured tags than RDF, while another remarked that, when the people annotating the content are paid, RDF is better because of it can be used for reasoning, but that structured tags are better for encouraging casual users to add annotations.

While most participants were more in favor of structured tags, several participants commented on the advantages of RDF. A number of participants found that when the domain and its concepts are known, RDF should be used since it allows for a cleaner formalization that is better suited for, for example, reasoning.

One participant remarked that when several people work on one annotation, RDF is preferable to structured tags due to its more rigid, pre-defined semantics. Another participant wrote that "RDF" delivers a more "clean" feeling, whereas [structured tags] feel somewhat "quick & dirty."

Some participants criticized both formalisms because they felt that uncertainty, negation, temporal information and complex relationships could not easily be expressed.

5.2.2 *Time requirements for annotations*

The average times needed to annotate each revision, grouped by text and formalism, are shown in Figures 5.11a and 5.11b.

On average, participants spent 22.5 minutes annotating each revision of text A and 23.4 minutes each revision of text B. The differences in the time spent annotating per revision between texts are minor and the distributions of time spent over the revisions of each text resemble each other.

Annotating one revision of a text with RDF on average took participants 30.7 minutes, almost double the average time needed to annotate a revision with structured tags, 15.7 minutes.

Not only the average amount of time, but also the differences in time spent annotating across revisions differ.

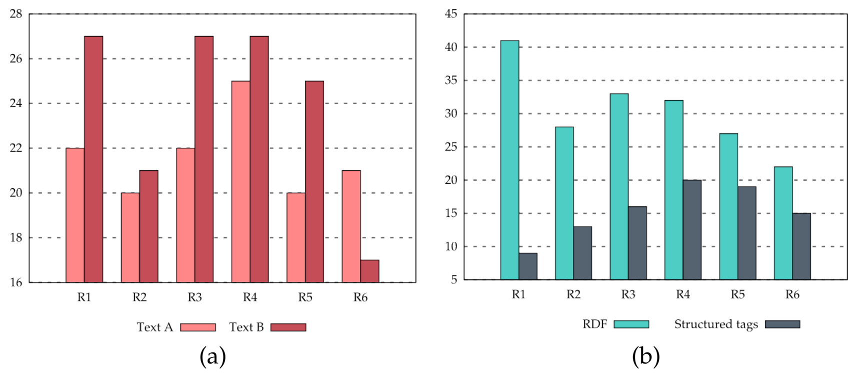


Figure 5.11: Average time taken to annotate the different revisions of the text (in minutes), grouped by (a) text and (b) annotation formalism

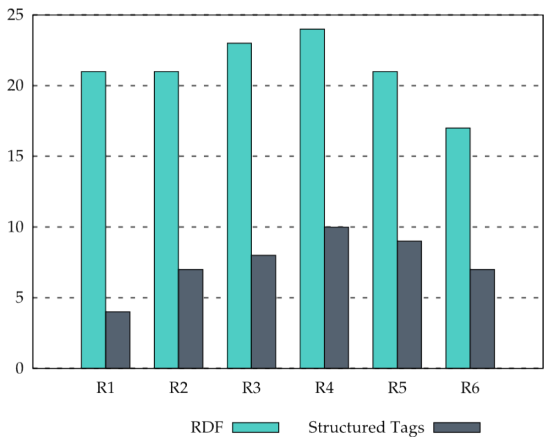


Figure 5.12: Average number of annotations per text revision (1 to 6), grouped by annotation formalism

Participants assigning RDF annotations spent 40.6 minutes on the first revision, more than on any other revision. At 28 minutes, the second revision was annotated comparatively quickly. The time spent annotating revisions three and four is higher and almost identical at 32.7 and 32 minutes respectively, and declines for the next and final two revisions (26.9 and 24 minutes).

When structured tags were used, the first revision took the shortest amount of time to annotate, 9 minutes. From there, the time taken gradually increases; revision 4 on average was annotated within 20.8 minutes, the highest amount of time across a revisions. The average time then declines to 19.1 minutes for the final revision.

5.2.3 *Analysis of the annotations*

On average, each participant wrote 178.53 annotations, 14.88 for each individual revision. 27.36 percent of those annotation are structured tags and 72.64 percent are RDF triples. 34.05 percent of the structured tags and 10.19 percent of the RDF triples were marked by participants as changes to earlier annotations.

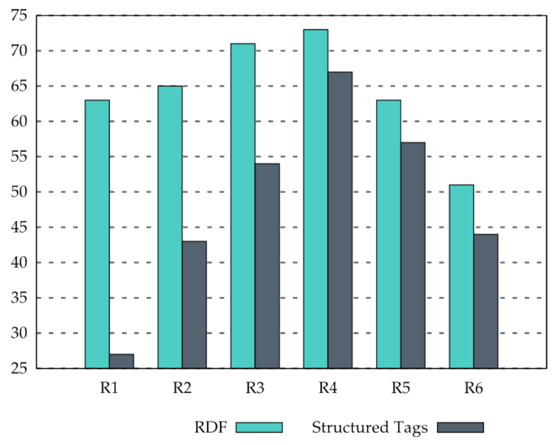


Figure 5.13: Average number of elements in annotations added per text revision (1 to 6), grouped by annotation formalism

Figure 5.12 shows the average number of annotations added per revision for both formalisms. In both cases, the number of annotations increases from the first to the fourth revision and then decreases for the last two revisions.

The initial increase in the number of annotations added is proportionally lower for the RDF annotations. Overall, considerably more RDF triples than structured tags were assigned; the number of RDF triples assigned is between two and four times higher than that of structured tags.

While RDF triples have a fixed number of elements, structured tags can be made up of an arbitrary number of simple tags. This means that comparing the overall number of annotations is not a sufficient metric to compare how much information is expressed through the annotations in each formalism. Figure 5.13 displays the average number of elements, that is, subjects, predicates, and objects, and simple tags, that were added to each revision of the text. For the first three revisions, the number of RDF elements is higher by a factor of 2 to 1.5, but for revisions 4 to 6, the number of RDF elements assigned is only about 10 percent higher than that of structured tags elements.

Note that this statistic does not fully capture the differences in the amount of information contained in the annotations either: while an RDF triple describes the relationship between a subject and a predicate, structured tags use labels and the grouping operator to describe the relationships between tags. Since structured tags can be characterized and grouping can be applied to an arbitrary number of simple or structured tags, the number of elements alone is not sufficient to determine how much information is expressed in a structured text in terms of the relations between elements. However, since the nature of relations in RDF and structured tags differ greatly, this factor cannot be easily quantified and we use the number of elements as an approximation of the information content of an annotation.

The average number of constituent simple tags in each structured tag is 8.35 across all revisions.

Grouping and characterization are used to comparable extents and often in combination; structured tags use grouping at least once in 85.39%, and characterization in 83.20% of all annotations.

The number of complex structured tags, that is, structured tags that contain at least one occurrence of labeling or characterizing a tag that itself

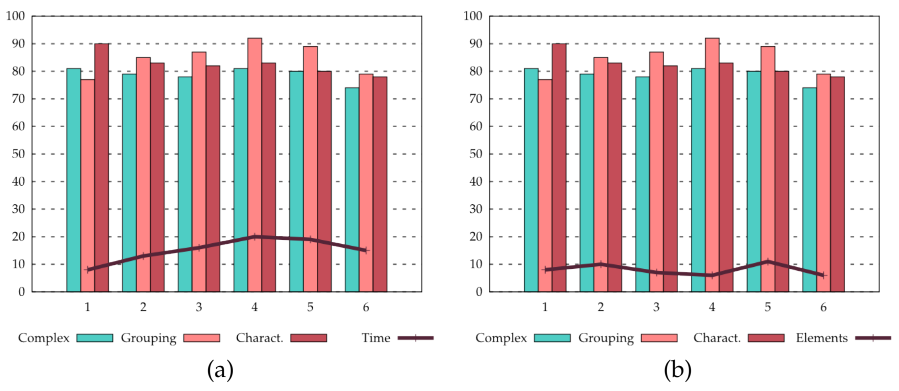


Figure 5.14: Relationship between tag complexity and time spent annotating (a) and number of total elements in the added structured tags (b), per revision

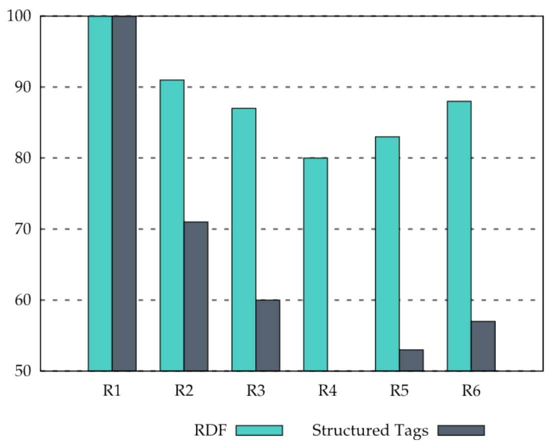


Figure 5.15: Average percentage of annotations not based on previous annotations, per revision

uses one the operations, is high, 79.62%. In most cases, it is the elements of a grouping or the tag being characterized that are complex, and 96.13 percent of labels are simple tags.

As figure 5.14 shows, the percentages of structured tags that use grouping, characterization or that are complex are not clearly related to the average time spent annotating or the average number of elements per structured tag.

When using structured tags, participants more frequently indicated that a newly added annotation was based on an existing annotation that it replaces (see Figure 5.15). Since participants wrote their annotations on paper, they had to manually specify that a new annotation was to be understood as a change to a previous annotation. It is likely that participants did not remember to add this note in all cases and consequently, the numbers shown in Figure 5.15 should be understood as a low estimate.

5.2.4 *Qualitative analysis of the annotations*

Different types of information (such as dates, time ranges, negative information, conditions, . . .) were introduced in the texts in order to get a better idea how natural structured tags are to represent various kinds of knowledge. Not all types of information were represented in both texts and the precise encoding was not always clear (mostly due to unintelligible handwriting). Therefore the results presented in this subsection should be taken only as the first indication of the actual usage. Let us first examine properties inherent to the two individual formalism.

RDF Two out of 19 participants had trouble properly distinguishing URIs and RDF literals, the same number of participants seemed to have trouble distinguishing b-nodes and URIs in some cases. 11 participants made use of blank nodes at least once and 9 participants used them mostly correctly. All participants used the "rdf:type" predicate and most used it heavily indicating that encoding type information was easy for the participants. 15 participants chose the simple triple syntax, 4 participants made use of the more compact Turtle syntax.

STRUCTURED TAGS 14 participants used grouping and characterization to describe classes of things, an example is: `Employee:(Al, Andy, Allen)`. Five out of 19 participants used a group as characterization at least once, an example is `(Spanish,Italian):(pretty well, cannot speak)`. All but two participants chose to represent dates in a detailed form, an example is: `(day:31, month:3, year:2011)`. The reason for such a high number probably is that representing date in a similar way is one of the examples in the introductory text to structured tags. In comparison, most participants represented a whole date as one literal in RDF (using various date formats). This may indicate that examples of best practices are likely to influence users' annotation behaviour significantly. Roughly 40% participants tended to form one complex structured tag instead of using many smaller ones. Such large structures bore a striking resemblance to involved JSON[1] objects.

DIFFERENT KINDS OF KNOWLEDGE 15 out of 19 participants (= ~79%) used grouping to represent a list of office days of a person, example: `John: reachable: (Mo, Tu, We)`. In comparison, only 8 participants represented such a list explicitly in RDF, 10 participants employed a literal, example: `John isReachableOn "Mo, Tu, We"`. Roughly the same number of participants (~30%) unambiguously encoded a chain of properties (e.g. A is a component of B, B is a component of C) in both formalisms with a higher total number of explicit (but partly ambiguous) encodings in structured tags. Participants were presented with the problem of representing information such as "Al maybe knows Bob" stressing the qualification of the *knows* predicate. Only one participant provided an explicit reification-based solution in RDF. Most participants (10) used a new predicate such as `maybeKnows`. Eight participants provided either a wrong solution, no solution, or used a literal in RDF. In comparison, 17 participants encoded such a statement explicitly in structured tags; 8 ambiguously such as `(a, probably, knows, b)`, 9 unambiguously such as `a:(probably:knows):b`. An if-then condition ("if project p is successful then it can be extended") was explicitly repre-

1 JavaScript Object Notation, see http://www.json.org/

sented by only one out of 19 participants in both formalisms. A comparison ("Al likes Python more than Java") was slightly more often explicitly represented in structured tags than in RDF but most participants provided either no, literal, or wrong solution in both cases. Interestingly, most participants encoded "causality" ("A likes B because C") explicitly in structured tags but almost none in RDF. Moreover, many of the structured tags solutions included a non-trivial combination of characterization, grouping and nesting. Participants encoded duration ("A project P runs from 2003 to 2006") wrongly or by a single literal in RDF in most cases while 13 participants provided a (partially) explicit solution in structured tags. Four out of the 13 solutions were unambiguous. Negation ("Al does not know Haskell") was encoded by most participants by introducing a new, implicitly dual, predicate in RDF, an example is `Al notKnows Haskell`. More participants tried to encode negation explicitly in structured tags and 5 of 19 participants encoded the negated relation unambiguously as for example: `not:knows` while the whole statement still sometimes remained ambiguous: `not:knows:(a, b)` (groups are unordered and thus it is unclear who does not know whom).

5.3 DISCUSSION

The results indicate that participants find structured tags convenient and comfortable to use and at the same time expressive.

The short introduction to structured tags was sufficient for most participants to understand how to use them and a majority of participants considers structured tags to be intuitive, convenient, and flexible. Participants were also more confident in the correctness of their annotations when using structured tags and found using structured tags more enjoyable than using RDF. Further, participants represented similar amounts of information in structured tags and RDF, but needed substantially less time when using structured tags even though the amount of explicitly encoded information was likely even higher in the case of structured tags.

Due to the experimental setup, the possibility cannot be fully excluded that the order in which the formalisms were used had an effect on the outcome of the experiment. For example, it is possible that the smaller amount of time needed for annotating the texts with structured tags is partially due to participants' experiences in creating the RDF annotations. If this practice would significantly decrease the time needed to write annotations, it could be expected that later revisions of the text annotated with RDF are annotated quicker than the earlier ones. However, this is not what we observed and with the exception of the very first text that was annotated, the relative amounts of time spent on each revisions are proportional between formalisms. This might indicate that either learning took place at the very beginning of the experiment or that RDF is initially harder to understand or use than structured tags. One might think that a different experimental setup would eliminate this issue but it would in fact introduce another uncertainty – for example either due to participant nuisance variables (personality, intelligence, previous experience, ...) if different groups were used for different formalisms, or due to the order of the used text variants which were of course created as similar but sufficiently different and as such can be a source of an order effect, or a significantly larger number of participants could be necessary. See for example [193] for a more in-depth discussion of the problems around order effects, nuisance (confounding) variables, and experimental setups in general.

More structured tags than RDF triples were identified as being based on annotations to prior revisions. Additionally, several participants remarked that, unlike structured tags, RDF requires to have a clear idea of the conceptual domain before beginning to annotate. Together, these findings indicate that structured tags are seen as more suited for expressing evolving knowledge than RDF is. At the same time, several users find that, when it is used in an appropriate environment where the domain concepts are known and where the annotations are assigned by experienced users, RDF has advantages over structured tags.

The qualitative evaluation showed that many participants had problems using for example b-nodes, which is not surprising, but some also had difficulties with properly distinguishing URI resources and literals. Participants were also more likely to correctly encode different kinds of knowledge such as causation, predicate specialization, and negation in structured tags. In RDF, they were more likely to either not encode such knowledge at all, encode it using literals, or implicitly using new predicates. The example of encoding dates in structured tags shows that if the users were presented with predefined constructs for each different knowledge kind in the introduction, they would likely use them. Moreover, the number of creative and "correct" solutions to such problems was higher with structured tags than with RDF which supports our initial hypothesis that structured tags can be seen as a more suitable candidate for (non-expert) community driven annotation than RDF.

Overall, the results of the experiment indicate that structured tags succeed at realizing a way to assign semi-formal annotations that are more expressive than simple tags but easier to use, especially for expressing evolving knowledge, than RDF.

The proportions to which the two operations, grouping and characterization, were used were roughly equal and did not change across revisions. A high number of the structured tags that participants created are complex. While, especially in the case of large structured tags, some evolution of a tag across revisions was observed, it was mostly adding new information and little or no evolution from vague to more specific concepts could be observed.

One reason for this could be that the conditions of the experiment were not realistic in that the experiment lasted several hours which were explicitly dedicated to creating annotations. Another way in which the annotations assigned likely differ from those used in practice is that they merely describe but do not enhance the content given in the text, for example with additional information, or with subjective opinions and assessments.

This experimental evaluation is concerned mainly with the participants' impressions and acceptance of structured tags. A study that focuses on the evolution of annotations, both from simple to structured tags and from structured tags to RDF, should consider this aspect in its experimental design.

ACKNOWLEDGEMENTS. This user study was realized in collaboration with my colleague Klara Weiand.

6 KWRL – THE KIWI RULE LANGUAGE

KWRL, pronounced [*quirl*], is an inconsistency-tolerant rule language based upon Datalog concepts that is aware of the conceptual model of the KiWi wiki. It aims at being simple for beginning users while staying sufficiently expressive and efficiently evaluable. KWRL consists of two sublanguages: sKWRL, pronounced [*squirl*], which is the core sublanguage, and the full KWRL which adds syntactic sugar to sKWRL to allow for a more compact conceptual-aware syntax.

The contribution of this chapter is twofold: a rule language about annotations, no other rule language for this purpose has yet been described as far as we know, and identification of the expressive power of such a language that is necessary for expressing rules about annotations: the power of a Datalog language with value invention. Moreover, value invention in KWRL is different from value invention in current Semantic Web rule languages in that the created constants are URLs instead of RDF blank nodes which makes information derived by KWRL more easily usable for example as part of Linked Data [26].

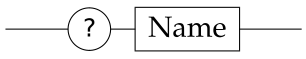
6.1 SKWRL – THE CORE SUBLANGUAGE

sKWRL is a rule language based on RDF triple patterns that can express for example the logical core of RDF(S) [160].

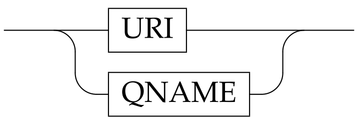
6.1.1 *Syntax*

This section presents the syntax of sKWRL. For brevity, only a simplified version is shown that for example does not allow omitting parenthesis in conjunctions with more than two conjuncts. The extension to a full syntax is a standard task and thus it is not included in this dissertation, an interested reader can have a look at the KiWi 1.0 sKWRL implementation.[1] We use the EBNF standard [119] accompanied by railroad diagrams [153, 150].[2]

Variable = '?' Name ;



Resource = URI | QNAME ;

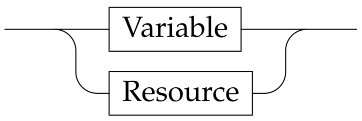


Variables are denoted by a question mark. A resource is either a full URI (an XML expanded name[3]) or an XML qualified name[4] that can be expanded to a URI; that is, while the XML Namespaces specification [34] does not

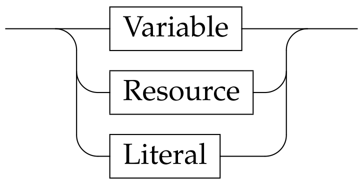
1 http://www.kiwi-community.eu/display/SRCE/Get+KiWi
2 We use white space as the concatenation symbol for better legibility. The standard uses a comma.
3 http://www.w3.org/TR/xml-names/#dt-expname
4 http://www.w3.org/TR/xml-names/#dt-qualname

require a namespace to be a URI, we do require it here so that the expanded name forms a valid RDF URI reference.[5]

Predicate = Variable | Resource ;

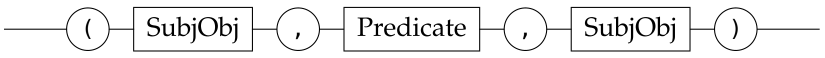


SubjObj = Variable | Resource | Literal ;



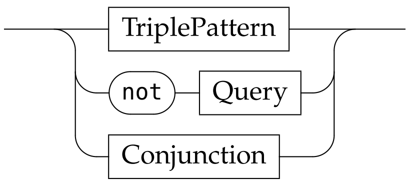
A triple pattern, `TriplePattern`, is a triple that allows a variable and a resource in any position and a literal in subject and object positions (in line with [160]).

TriplePattern = '(' SubjObj ',' Predicate ',' SubjObj ')' ;

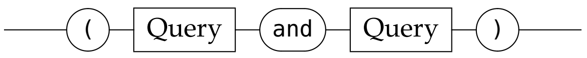


The base query is a triple pattern. Base queries can be combined using negation and conjunction to form complex queries.

Query = TriplePattern | 'not' Query | Conjunction ;

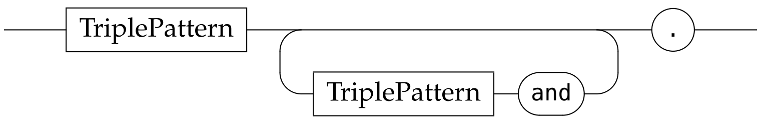


Conjunction = '(' Query 'and' Query ')' ;

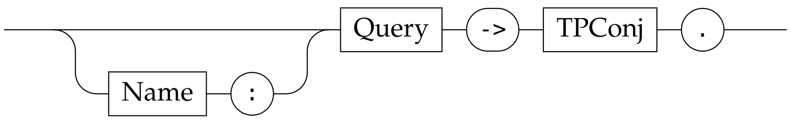


A rule can optionally have a name, a feature that is nice to have for later explanation purposes. A rule body consists of a (complex) query. A rule that has a `TriplePattern` or a conjcuntion of `TriplePatterns` as its head is a constructive rule. A rule that has the `inconsistency` keyword as its head is a constraint rule.

TPConj = TriplePattern {'and' TriplePattern} '.' ;

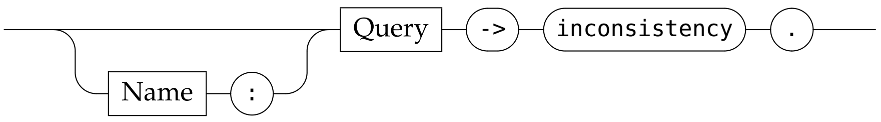


ConstructiveRule = [Name ':'] Query '->' TPConj '.' ;

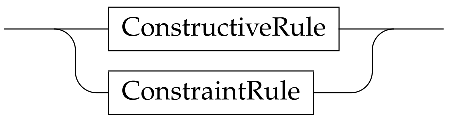


5 http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref

ConstraintRule = [Name ':'] Query '->' 'inconsistency' '.' ;

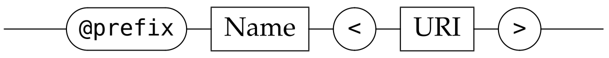


Rule = ConstructiveRule | ConstraintRule ;

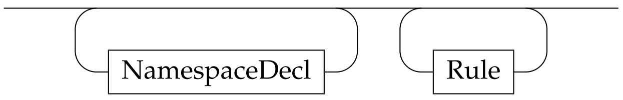


A program consists of an optional finite set of namespace declarations followed by a finite set of rules.

NamespaceDecl = '@prefix' Name '<' URI '>' ;



Program = { NamespaceDecl } { Rule }



6.1.2 *Examples*

Let us assume the following namespace declarations:

```
@prefix kiwi <http://kiwi-project.eu/ns/kiwi-core/>
@prefix rdf <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs <http://www.w3.org/2000/01/rdf-schema#>
```

Then the following sKWRL rule expresses the (arguably questionable) transitivity of the `kiwi:relatedTo` relation:

```
((?x, kiwi:relatedTo, ?y) and (?y, kiwi:relatedTo, ?z))
                                -> (?x, kiwi:relatedTo, ?z) .
```

The following example sKWRL program consists of two constraint rules and checks that a resource p is a parent of a resource c iff c is a child of p.

```
((?p, kiwi:isParentOf, ?c) and not (?c, kiwi:isChildOf, ?p))
                                  -> inconsistency .
(not (?p, kiwi:isParentOf, ?c) and (?c, kiwi:isChildOf, ?p))
                                  -> inconsistency .
```

Let us now specify a rule that creates a new tag annotation for any entity that has a tag annotation with label "squirrel." We omit the "kiwi:" namespace for brevity and only include two triple patterns in the head – the full rule would include triple patterns for all mandatory properties of an annotation in the head.

```
(?t rdf:type SimpleTag) and (?t hasLabel "squirrel") and
(?a annotatedEntity ?e) and (?a metadata ?t) ->
  (?new annotatedEntity ?e) and (?new metadata animalTagUri)
  and ...
```

Note that `?new` is only a variable (i.e. the name "new" has no special meaning in sKWRL). The variable does not occur in the rule body (i.e. it is not range restricted) and therefore it makes use of the value invention feature of sKWRL, see the next section for more information. Also note that value invention is necessary in order to create *new* taggings: intuitively, the rule head can use values bound to variables in the rule body – none of these values can represent a *new* tagging and thus a new constant has to be created in the rule head. Values created by sKWRL are URLs which means that they, in contrast to blank nodes, can be used outside of a local RDF graph too. This is important in the context of LinkedData and data sharing in general.

The full KWRL provides syntactic sugar that allows for writing rules about annotations that are much more concise.

6.1.3 *Semantics*

In this section, the semantics of sKWRL is provided by translation to Datalog $^{\neg}_{\infty}$, i.e. a deterministic Datalog language with negation and value invention. Datalog $^{\neg}_{\infty}$ is formally defined in [5] by fixpoint semantics and by translation to a procedural language. In contrast to [5], we restrict ourselves to Datalog with stratified negation [44]. See also [136, 6] for an introduction to standard Datalog without value invention.

In ordinary Datalog, a non-range-restricted rule (also called "unsafe" [5]) is a rule that has a variable in its head that is not in its body. Such a variable can be instantiated by any constant from the universe of discourse for any variable binding satisfying the body resulting in a valid rule instance. Such rules are of limited use as they inherently express a property satisfied by all constants in the universe of discourse. An example is a rule expressing that everything is an `rdfs:Resource` in RDF: $(x \;\; \mathit{rdf{:}type} \;\; \mathit{rdfs{:}Resource}) \leftarrow \top$ which however does not hold in the current version of RDF because RDF literals are not of type `rdfs:Resource`.

Datalog $_{\infty}$, i.e. ordinary Datalog with value invention, interprets non-range-restricted rules so that new constants are created for each body variable binding and variable in the rule head that is not bound. This means that a variable occurring in a rule head but not in the rule body is considered existentially quantified instead of universally quantified. Intuitively, the new constants mark the respective body variable bindings; the constant is sometimes referred to as the "witness" of the variable binding. Datalog $_{\infty}$ is strictly more expressive than ordinary Datalog because it can express queries that include new constants which is impossible in ordinary Datalog. Datalog $_{\infty}$ does not guarantee termination because, with recursive rules, new constants may be generated ad infinitum. This can be alleviated by syntactically restricting the use of value invention, e.g. by forbidding "recursion over value invention" which can be verified in a similar manner as stratifiability of a Datalog $^{\neg}$ program, see the similar notion of weakly safe programs in [5]. In the following, we use the standard logic programming syntax for Datalog [136].

Datalog $^{\neg}_{\infty}$ is defined in [5] so that only a single atom may appear in rule heads but, as [5] notes (e.g. the remark on page 90), the syntax with multiple atoms in rule heads does not increase expressiveness under deterministic semantics. Datalog $^{\neg}_{\infty}$ with unrestricted negation is complete for the class of deterministic database queries. We assume Datalog $^{\neg}_{\infty}$ with stratified negation which is likely less expressive. See for example [118, 55, 58] for an in-depth discussion of database complexity classes and the power of relational database languages.

Qualified names and namespaces in sKWRL serve only the purpose of abbreviations. Therefore we assume that qualified names are expanded into full URIs using namespace declarations of a program and we give semantics for the transformed program with expanded qualified names. Most of the recursive translation of sKWRL to Datalog $^{\neg}_{\infty}$ is straightforward:

(Variable)	$[\![?\mathtt{x}]\!]$	$:= \mathtt{var}_{\mathtt{x}}$
(Resource)	$[\![\mathtt{uri}]\!]$	$:= \mathtt{const}_{\mathtt{uri}}$
(Literal)	$[\![\mathtt{literal}]\!]$	$:= \mathtt{const}_{\mathtt{literal}}$
(TriplePattern)	$[\![(\mathtt{s}, \mathtt{p}, \mathtt{o})]\!]$	$:= \mathtt{triple}([\![\mathtt{s}]\!], [\![\mathtt{p}]\!], [\![\mathtt{o}]\!])$
(not Query)	$[\![\mathtt{not\ query}]\!]$	$:= \neg[\![\mathtt{query}]\!]$
(Conjunction)	$[\![(\mathtt{query}_1\ \mathtt{and}\ \mathtt{query}_2)]\!]$	$:= ([\![\mathtt{query}_1]\!], [\![\mathtt{query}_2]\!])$
(TPConj)	$[\![\mathtt{tp}_1\ \mathtt{and}\ \mathtt{tp}_2\ \ldots\ \mathtt{and}\ \mathtt{tp}_n]\!]$	$:= ([\![\mathtt{tp}_1]\!], [\![\mathtt{tp}_2]\!], \ldots, [\![\mathtt{tp}_n]\!])$
(ConstructiveRule)	$[\![\mathtt{query\ \text{->}\ head}]\!]$	$:= [\![\mathtt{head}]\!] \leftarrow [\![\mathtt{query}]\!]$

$\mathtt{var}_{\mathtt{x}}$ is a Datalog variable for the sKWRL variable ?x, similarly $\mathtt{const}_{\mathtt{uri}}$ is a constant corresponding to the URI uri, and $\mathtt{const}_{\mathtt{literal}}$ is a constant corresponding to the literal literal. $\mathtt{triple}(\_, \_, \_)$ is a predicate that represents RDF triples in a single predicate axiomatization of RDF(S) as for example in [140, 160].

Constraint rules translate to not range restricted Datalog $^{\neg}_{\infty}$ rules with a fresh variable in the head: for each constraint rule, x is a fresh variable that does not occur in $[\![\mathtt{query}]\!]$:

(ConstraintRule) $[\![\mathtt{query\text{->}inconsistency}]\!] := \mathtt{triple}(\mathtt{sub}, \mathtt{prop}, x) \leftarrow [\![\mathtt{query}]\!]$

Where sub stands for a URI identifying the default RDF graph (resp. named graph [51]) and prop stands for a URI for a special hasInconsistency property. This way each variable binding of the query is “marked” by a new constant – each inconsistency can be tracked back to its origin; the variable binding that “caused it.” This feature is important for explanation. For the semantics of the language, it is sufficient that for each variable binding and each not range restricted variable in the head there is a unique constant without imposing further restrictions on the constant. For explanation, it may be beneficial to enable inverting constants to the corresponding rule body instances. See Chapter 9 for an example how this can be achieved.

sKWRL is sufficiently expressive to represent for example the ρdf logical fragment of RDF(S) defined in [160] ([160] gives a deductive system for ρdf that corresponds to a set of Datalog rules, their translation to sKWRL is straightforward). sKWRL programs can however become lengthy and thus a more concise language is desirable especially for wiki users.

6.1.4 *Significance of Value Invention*

One of the motivations behind sKWRL and KWRL is the need to express rules about annotations, for example: “if there is a tag annotation ’linked-Data’ of an item then the item should also be annotated with the tag ‘semanticWeb’.” Remember that an annotation is a piece of meta-data attached to a resource (Section 4.2); ‘semanticWeb’ is the meta-data in this case and the annotation is an object that is "supposed to exist." If we constrain ourselves to declarative rule languages (we forbid e.g. production rules and Event-Condition-Action rules) then the traditional way of specifying this rule is to use existential quantification in the rule head: "if there is a tag annotation 'linkedData' of an item then there *exists* a 'semanticWeb' tag annotation of that item." Existential quantification in the head of an RDF rule language naturally translates to creation of b-nodes because b-nodes can be seen as existentially quantified variables. RDF b-nodes however are not accessible from outside their RDF graph – i.e. the 'semanticWeb' annotation could not be referred to from outside the system that created it which is a major disadvantage from the perspective of for example Linked Data. Of course, this problem can be circumvented by *creating* a new URI for the annotation and expressing that it is equivalent to the b-node. Allowing value invention directly in rules solves this problem in an arguably simpler way and the resulting rule language has moreover a clear formally defined semantics. Formal semantics of the language allows for analysing its properties and for example determining and enforcing syntactic restrictions such as safeness with respect to value invention that ensure good behaviour of the program, e.g. termination.

6.2 KWRL

KWRL is a rule language about annotations (Section 4.2) that is based on sKWRL; KWRL adds two syntactic constructs (annotation patterns and disjunction) that allow for more compact rules. KWRL programs can be expanded into equivalent sKWRL programs.

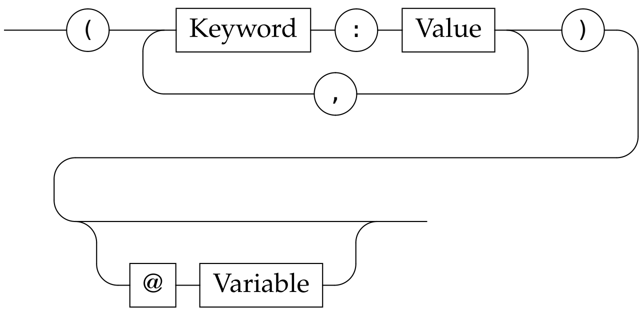
6.2.1 *Syntax*

KWRL adds Annotation Pattern atomic queries to sKWRL for easier description of annotations. An AnnotationPattern is a tuple of keyword-value pairs that correspond to RDF triples that describe annotations and metadata (Section 4.2):

There also may be a Variable at any of the values positions.

Keyword = 'taguri' | 'tag' | 'staguri' | 'stag' | 'rdfuri' | 'rdf' | 'entity' | 'author' | 'date' | 'marker' | 'polarity'
Value = Resource | Literal | Variable | StructuredTag | TriplePattern | 'user' | 'rule' | 'system' | '+' | '-'

AnnotationPattern = '(' Keyword ':' Value {',' Keyword ':' Value } ')' ['@' Variable]



property	keyword	value
`kiwi:metadata`	taguri	Resource
`kiwi:hasLabel`	tag	Literal
`kiwi:metadata`	staguri	Resource
(see Section 4.2.3)	stag	StructuredTag
`kiwi:metadata`	rdfuri	Resource
(see RDF reification)[6]	rdf	TriplePattern
`kiwi:annotatedEntity`	entity	Resource
`kiwi:hasAuthor`	author	Resource
`kiwi:dateCreated`	date	Literal
`kiwi:marker`	marker	`user`, `rule`, `system`
`kiwi:polarity`	polarity	`-`, `+`

Table 6.1: Mapping of annotation pattern keywords to RDF properties.

If a *Literal* stands at the value position of the `tag` keyword then all tags with the label corresponding to the literal are matched. The *StructuredTag* nonterminal serves an analogous purpose for structured tags.

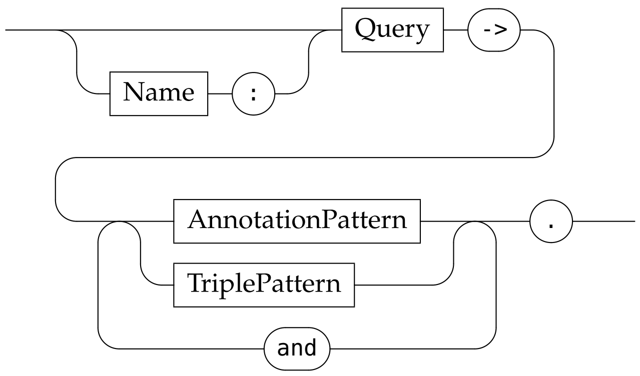
StructuredTag = SimpleTag | TagGrouping | TagCharact
SimpleTag = Literal | Resource | Variable
TagGrouping = '(' StructuredTag ',' StructuredTag ')'
TagCharact = (SimpleTag | TagGrouping) ':' (SimpleTag | TagGrouping)

In contrast to sKWRL queries, KWRL queries also allow annotation patterns and disjunction.

Query = AnnotationPattern | TriplePattern | 'not' Query | Conjunction | Disjunction
Disjunction = '(' Query 'or' Query ')'

The head of a KWRL constructive rule consists of a conjunction of annotation and triple patterns.

ConstructiveRule = [Name ':'] Query '->' (AnnotationPattern | TriplePattern) { 'and' (AnnotationPattern | TriplePattern) } '.' ;



As usual, it is also allowed to omit unnecessary parenthesis around conjunctions and disjunctions and `and` has a higher precedence than `or`.

6.2.2 *Examples*

In full KWRL, the second sKWRL program example may be rewritten as a single rule:

```
((?p, kiwi:isParentOf, ?c) or (?c, kiwi:isChildOf, ?p)) and
(not (?c,kiwi:isChildOf,?p) or not (?p,kiwi:isParentOf,?c))
                                      -> inconsistency .
```

Let us rewrite the sKWRL rule about annotations to full KWRL.

```
(tag:"squirrel",entity:?e) -> (taguri:animalTagUri,entity:?e)
```

The full KWRL rule is apparently much more concise than the corresponding sKWRL rule from Section 6.1.2 and it moreover hides the details of representing annotations in RDF and the necessity to use value invention which arguably is not a common feature. This more compact notation is likely to benefit beginner users as they can focus on formulating rules about annotations and do not have to care much about the internal representation details.

6.2.3 *Semantics*

Semantics of KWRL is given here by translation of KWRL rules to sKWRL rules. sKWRL is moreover a sublanguage of KWRL and therefore KWRL and sKWRL have the same expressive power.

An annotation pattern query translates to a conjunction of triple pattern queries according to the annotation representation as RDF triples. Each annotation has a URI which is either implicit in the annotation pattern or it is referred to explicitly by the "@ Variable" part of the query. Each annotation pattern without the "@"-part is given one with a fresh variable. Each keyword-value pair of an annotation pattern maps to one or several triple pattern conjuncts. Table 6.2 shows the mapping derived from Table 6.1 and from the representation of annotations and metadata described in Section 4.2.

A KWRL rule containing disjunction in its body translates to possibly several sKWRL rules. For example the KWRL rule

```
((a,p,x) or (a,p,y)) -> inconsistency
```

translates to two sKWRL rules:

```
(a,p,x) -> inconsistency
(a,p,y) -> inconsistency
```

In the general case, a KWRL rule body is transformed to a disjunctive normal form by De Morgan's laws and a new rule is created (using the head of the original rule) from each of the disjuncts.

6.2.4 *Evaluation*

While the semantics of KWRL is specified by translation to sKWRL, direct evaluation of KWRL rules may be advantageous. For example rules with disjunction in the body may be evaluated more efficiently directly. It is apparent that rules about annotations can become long in sKWRL. Such rules

$\llbracket(k_1 : v_1, \ldots, k_n : v_n)@?x\rrbracket$	$= \llbracket ?x, k_1, v_1\rrbracket$ and ... and $\llbracket ?x, k_n, v_n\rrbracket$
$\llbracket ?x, taguri, res\rrbracket$	$= (?x, \texttt{kiwi:metadata}, res)$
$\llbracket ?var, tag, label\rrbracket$	$= (?x, \texttt{kiwi:metadata}, ?m)$ and $(?m, \texttt{kiwi:hasLabel}, label)$
$\llbracket ?x, staguri, res\rrbracket$	$= (?x, \texttt{kiwi:metadata}, res)$
$\llbracket ?x, stag, s\rrbracket$	= a conjunction of triple patterns that describe the structured tag s – see the StructuredTag non-terminal definition and “Representing structured tags in RDF” in Section 4.2.3.
$\llbracket ?x, rdfuri, res\rrbracket$	$= (?x, \texttt{kiwi:metadata}, res)$
$\llbracket ?x, rdf, (s, p, o)\rrbracket$	$= (?x, \texttt{kiwi:metadata}, ?m)$ and $(?m, \texttt{rdf:type}, \texttt{rdf:Statement})$ and $(?m, \texttt{rdf:subject}, s)$ and $(?m, \texttt{rdf:predicate}, p)$ and $(?m, \texttt{rdf:object}, o)$
$\llbracket ?x, entity, res\rrbracket$	$= (?x, \texttt{kiwi:annotatedEntity}, res)$
$\llbracket ?x, author, res\rrbracket$	$= (?x, \texttt{kiwi:hasAuthor}, res)$
$\llbracket ?x, date, d\rrbracket$	$= (?x, \texttt{kiwi:dateCreated}, res)$
$\llbracket ?x, marker, m\rrbracket$	$= (?x, \texttt{kiwi:marker}, m)$
$\llbracket ?x, polarity, p\rrbracket$	$= (?x, \texttt{kiwi:polarity}, p)$

Table 6.2: AnnotationPattern to sKWRL TriplePatterns mapping.

may result in a large number of joins in an SQL implementation of the language. Traditional relational database systems are not optimized for exceedingly long queries. Therefore it may be beneficial to represent annotations not only in RDF but also directly as a single entity. In such a setting, annotation pattern queries can be evaluated using significantly fewer SQL joins and thus more efficiently. This approach (taken, in part, by the KiWi 1.0 implementation), however, has disadvantages such as the need to synchronize the two distinct annotation representations.

Evaluation of structured tags poses a specific problem and that is structural matching. Structured tag queries are deliberately limited in KWRL in order to fit well with classical Datalog. More powerful constructs could be provided such as a descendant operator or a unification-like variable matching. This topic is outside the scope of this dissertation and the interested reader is referred to Klara Weiand’s dissertation [223] about the KiWi query language KWQL and structural search.

6.3 RELATED WORK

There is a variety of rule languages for the Semantic Web. SPARQL [182] is the first standardized rule language for the Semantic Web. It is a rule language for RDF that also supports value invention which is however limited to blank-node construction (see Section 10.2.1 of [182]). HEX [79, 80], "Higher-order with EXternal atoms", is a rule language for the Semantic Web that includes features such as default negation and higher-order reasoning. It also supports external sources of data via the external atoms – a feature that is closely related to value invention that allows for enriching the universe of discourse with new constants for external data. RDFLog [45] and Xcerpt^{RDF} [46] are rule languages that also support blank node construction. RDFLog supports arbitrary quantifier alternation and Xcerpt^{RDF} is an RDF extension of the powerful query language Xcerpt [134, 43] that can query both data (XML, HTML, ...) on the Web and data on the Semantic Web (RDF, TopicMaps, ...). The semantic web rule language TRIPLE [203] has some kind of RDF triple import feature and thus "value creation by importing." TRIPLE is a syntactic extension of Horn logic and thus not range-restricted rules have the standard semantics of first order logic. A more detailed model theoretic semantics of TRIPLE and blank node treatment were left for a next version of the language which probably never came. Tim Berners-Lee's N3 [23] allows existential quantification in rule heads which has the meaning of blank node value invention. To the best of our knowledge, KWRL is the first rule language that invents URLs; other semantic web rule languages with value invention invent blank nodes.

Most semantic wikis such as for example SemperWiki [174, 172] and Semantic MediaWiki [217] provide some annotation mechanism that however usually assumes that users write, in effect, RDF triples the subject of which is the annotated page. In contrast, KWRL and the conceptual model of KiWi (Chapter 4) clearly distinguish between this kind of adding triples and creating annotations as rich descriptions "attached to" existing resources.

Most rule languages for the Semantic Web are general and aim at high expressiveness. This means that they can handle a large class of use cases sometimes at the expense of usability. This is natural and not surprising as the rule languages do not and cannot provide syntactic sugar for specific uses. Therefore the annotation and wiki-related rules expressed in them are likely to be similar to the long sKWRL rules such as in Section 6.1.2. For example the last rule of the section may be expressed in N3Logic as follows (we omit namespace declarations):

```
@forAll t, a, e
{ t a SimpleTag. t hasLabel "squirrel". a annotatedEntity e.
  a metadata t }
  log:implies
{ ?new annotatedEntity e. ?new metadata animalTagUri. ... }
```

A similar TRIPLE program would look as follows (we again omit namespace declarations and the optional model specifications):

```
FORALL t, a, e
EXISTS new (new[annotatedEntity -> e; metadata ->
            animalTagUri] AND ...)
<-
t[rdf:type -> SimpleTag; hasLabel -> "squirrel"] AND
a[annotatedEntity -> e; metadata -> t]
```

Note that both the N3 program and the TRIPLE program have different semantics than the corresponding sKWRL program from Section 6.1.2. The N3 program invents blank nodes while the sKWRL program invents URLs.

The variable "new" in the TRIPLE program has the standard semantics that does not invent values. While TRIPLE allows for a slightly more compact syntax by glueing together statements with the same subject (a feature supported by N3 too), the full rule still has to explicitly list all mandatory properties of an annotation in the head. In comparison, KWRL is aware of the conceptual model of a wiki and provides syntactic sugar for it which allows for a significantly more compact rule specification that hides most technical details:

```
(tag:"squirrel",entity:?e) -> (taguri:animalTagUri,entity:?e)
```

KWRL embraces the social Semantic Web and the conceptual model of the KiWi wiki in order to simplify its use to non-expert users. These goals are shared with the KWQL query language [223] that is also aware of the conceptual model of the KiWi wiki but is more focused on querying structure than on annotations.

7 FORWARD CHAINING REVISITED

This chapter presents several extended forward chaining algorithms and builds a unifying framework of so called support graphs in which they (and reason maintenance algorithms of the next chapter) are described. First classical forward chaining is reviewed. Then a support keeping extension and several novel extensions are presented. Each is described using a new immediate consequence operator defined in such a way that the resulting naive and semi-naive algorithms are directly analogous to the classical forward chaining algorithms.

Recall that the focus on forward chaining over backward chaining in this dissertation stems from its anchoring around a wiki. Materialization of derivable facts can be seen as a completion of the wiki according to the rules defined by its users. This completion makes it easier and more efficient for wiki users to see the "current state of affairs" including possible inconsistencies immediately.

7.1 PRELIMINARIES

Most of this section follows the usual definitions as for example in [44]. Sections 7.1.1, 7.1.2, and 7.1.3 review classical definitions from logic programming for the reader's convenience. Section 7.1.4 introduces the notion of a support and a definition of the classical immediate consequence operator based on this notion. Section 7.1.5 reviews multisets extended with infinite multiplicities and defines the so called powerset of a multiset that is useful for the fixpoint theory of the multiset-based extended immediate consequence operators proposed and studied in next sections. Section 7.1.6 introduces so called "support graphs", a novel data structure for derivations inspired by data-dependency networks from the field of reason maintenance (see e.g. [74] or the next chapter).

Throughout this dissertation a first order predicate logic language with signature $\mathcal{S}$ is assumed. $\mathcal{S}$ is assumed to have no function symbols other than constants, and it is assumed to include at least one constant and the two $\mathcal{S}$-formulas: $\top$ and $\bot$ which respectively evaluate to true and false in all interpretations.

7.1.1 *Rules and programs*

First, let us review standard definitions from terms to logical formulas and programs. This subsection is based on [44] and can easily be skipped by readers familiar with these notions.

Definition 2 ($\mathcal{S}$-term)**.** *Let $\mathcal{S}$ be a signature. We define:*

1. *Each variable x is an $\mathcal{S}$-term.*
2. *Each constant c of $\mathcal{S}$ is an $\mathcal{S}$-term.*

Definition 3 ($\mathcal{S}$-atom)**.** *Let $\mathcal{S}$ be a signature. For $n \in \mathbb{N}$, if p is an n-ary relational symbol of $\mathcal{S}$ and $t_1, \ldots, t_n$ are $\mathcal{S}$-terms, then $p(t_1, \ldots, t_n)$ is an $\mathcal{S}$-atom or atomic*

$\mathcal{S}$-formula. For $n = 0$ the atom may be written $p()$ or p and is called a propositional $\mathcal{S}$-atom.

Definition 4 ($\mathcal{S}$-formula)**.** *Let $\mathcal{S}$ be a signature. We define inductively:*

1. *Each $\mathcal{S}$-atom is an $\mathcal{S}$-formula. (atoms)*
2. *$\top$ and $\bot$ are $\mathcal{S}$-formulas. (0-ary connectives)*
3. *If φ is an $\mathcal{S}$-formula, then $\neg\varphi$ is an $\mathcal{S}$-formula. (1-ary connectives)*
4. *If φ and ψ are $\mathcal{S}$-formulas, then $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$ and $(\varphi \Rightarrow \psi)$ are $\mathcal{S}$-formulas. (2-ary connectives)*
5. *If x is a variable and φ is an $\mathcal{S}$-formula, then $\forall x\varphi$ and $\exists x\varphi$ are $\mathcal{S}$-formulas. (quantifiers)*

Definition 5 (Substitution)**.** *A substitution is a function σ, written in postfix notation, that maps terms to terms and is*

- *identical on constants, i.e. $c\sigma = c$ for constants,*
- *identical almost everywhere, i.e., $\{x \mid x \text{ is a variable and } x\sigma \neq x\}$ is finite.*

The domain of a substitution σ, denoted $\mathrm{dom}(\sigma)$, is the finite set of variables on which it is not identical. Its codomain is the set of terms to which it maps its domain. A substitution σ is represented by the finite set $\{x_1 \mapsto x_1\sigma, \ldots, x_k \mapsto x_k\sigma\}$ where $x_1, \ldots, x_k$ is its domain and $x_1\sigma, \ldots, x_k\sigma$ is its codomain. $\{x_1 \mapsto x_1\sigma, \ldots, x_k \mapsto x_k\sigma\}$ can be shortly written as $\{x_1/x_1\sigma, \ldots, x_k/x_k\sigma\}$.

The application of a substitution to a formula, a set of formulas, or any other object containing terms is defined as usual (cf [44]) and also written in postfix notation.

Definition 6 (Subsumption ordering)**.** *A term s subsumes a term t, denoted $s \leqslant t$, iff there exists a substitution σ with $s\sigma = t$. One also says that t is an instance of s, or t is more specific than s, or s is more general than t.*

Analogously for atoms.

A substitution σ subsumes a substitution τ, denoted $\sigma \leqslant \tau$, iff there exists a substitution θ with $\sigma\theta = \tau$. One also says that τ is an instance of σ, or τ is more specific than σ, or σ is more general than τ.

Definition 7 (Unification)**.** *Two terms s and t are unifiable, if there exists a substitution σ with $s\sigma = t\sigma$. In this case σ is called a unifier of s and t.*

A most general unifier or mgu is a minimal element w.r.t. the subsumption ordering among the set of all unifiers of s and t. If σ is a most general unifier of s and t, the term $s\sigma$ is called a most general common instance of s and t.

Notation 8 (Function var())**.** *Let us denote $\mathrm{var}()$ a function that maps an object (a term, a formula, a substitution, ...) to the set of all variables that occur in it.*

Definition 9 (Ground)**.** *A term t is a ground term iff $\mathrm{var}(t) = \emptyset$. A formula φ is a ground formula iff $\mathrm{var}(\varphi) = \emptyset$. A substitution is ground if its codomain consists of ground terms only. A grounding substitution for a term t is a ground substitution σ whose domain includes all variables in t, such that $t\sigma$ is ground.*

Definition 10 (Instance of a formula)**.** *Let φ be a formula and σ a ground substitution. Then $\varphi\sigma$ is the formula obtained from φ by replacing each free variable occurrence x in φ by $x\sigma$.*

Notation 11 (Rule). *A* rule $r = \psi \leftarrow \varphi$ *is a notation for a formula* $(\varphi \Rightarrow \psi)$ *which can but does not have to be closed. The subformula* φ *is called the* antecedent *or* body, $\text{body}(r)$ *denotes the set of atoms in* φ, *and* ψ *the* consequent *or* head *of the rule,* $\text{head}(r)$ *denotes the set of atoms in* ψ, *and* $\text{heads}(R) = \bigcup_{r \in R} \text{head}(r)$, *where* R *is a set of rules.*

Definition 12 (Literal, Complement). *If* A *is an atom then both* A *and* $\neg A$ *are* literals *(the former positive, the latter negative). The complement of a positive literal* A*, denoted* $\overline{A}$*, is* $\neg A$*, the complement of* $\neg A$*, denoted* $\overline{\neg A}$*, is* A.

Note that $\bot, \top, \neg\bot, \neg\top$ are no literals because the are defined as formulas but not atomic formulas.

Definition 13 (Clause). *A* clause *is a disjunction of finitely many literals. It is a shorthand for its universal closure, i.e., a closed formula. A clause* $A_1 \vee \ldots \vee A_m \vee L_1 \vee \ldots \vee L_n$*, where the* A_i *are positive literals, and the* L_j *are literals, is written as* $A_1 \vee \ldots \vee A_m \leftarrow \overline{L_1}, \ldots, \overline{L_n}$ *in rule notation. For* $m = 0$*, the head of the rule is written* $\bot$*. For* $n = 0$*, the body of the rule is written* $\top$.

Definition 14 (Definite rule, normal rule). A definite rule *is a rule of the form* $A \leftarrow L_1, \ldots, L_n$*, where* L_i *are positive literals and* A *is an atom.* A normal rule *is a rule of the form* $A \leftarrow L_1, \ldots, L_n$*, where* L_i *are literals and* A *is an atom.*

Definition 15 (Program, base facts). *A* program *is a finite set of rules. A* definite program *is a finite set of definite rules. A* normal program *is a finite set of normal rules. A base fact is a rule with* $\top$ *as its body.*

Terminology 16 (Base fact). *The term "base fact" may be used for both a rule with* $\top$ *as its body and for the head atom of such a rule. It is usually clear from context which exact meaning is meant. The only exception is the case when both meanings are admissible and lead to equivalent statements such as "the number of terms in base fact* x*" because* $\top$ *is a formula that contains no terms and thus the number of terms in* a *and the number of terms in* $a \leftarrow \top$ *is the same for an atom* a.

See also Terminolgy 104.

Definition 17 (Range restricted). *A definite rule is* range restricted *if each variable occurring anywhere in it also occurs in its body. A definite program is range restricted if each of its elements is.*

Note that the base facts in a range restricted definite programs are ground atoms.

7.1.2 *Herbrand interpretations*

Let us review standard definitions relating to models of logical theories. This subsection is based on [44] and can easily be skipped by readers familiar with the notions. However, note that for example Observation 25 is often used in later sections in proofs of termination of the presented algorithms and definitions analogous to Definition 28 are given with respect to the novel operators in later sections.

Definition 18 (Variable assignment). *Let* D *be a nonempty set. A variable assignment in* D *is a function* V *mapping each variable to an element of* D*. We denote the image of a variable* x *under an assignment* V *by* x^V .

Definition 19 ($\mathcal{S}$-Interpretation). *Let $\mathcal{S}$ be a signature. An $\mathcal{S}$-interpretation is a triple $\mathcal{I} = (D, I, V)$ where*

- D *is a nonempty set called the domain or universe (of discourse) of* I*, denoted* $\mathrm{dom}(I)$*,*
- I *is a function defined on the symbols of* $\mathcal{S}$ *mapping*
  - *each* n*-ary function symbol* f *to an* n*-ary function* $f^I : D^n \to D$*. For* $n = 0$ *this means* $f^I \in D$*. Notation:* $f^{\mathcal{I}} = f^I$*.*
  - *each* n*-ary relation symbol* p *to an* n*-ary relation* $p^I \subseteq D^n$*. For* $n = 0$ *this means either* $p^I = \emptyset$ *or* $p^I = \{\emptyset\}$*. Notation:* $p^{\mathcal{I}} = p^I$*.*
- V *is a variable assignment in* D*. Notation:* $x^{\mathcal{I}} = x^V$

Notation 20. *Let* V *be a variable assignment in* D*, let* V' *be a partial function mapping variables to elements of* D*, which may or may not be a total function. Then* $V[V']$ *is the variable assignment with*

- $x^{V[V']} = x^{V'}$ *if* $x^{V'}$ *is defined*
- $x^{V[V']} = x^V$ *if* $x^{V'}$ *is undefined*

Let $\mathcal{I} = (D, I, V)$ *be an interpretation. Then* $\mathcal{I}[V'] := (D, I, V[V'])$*. By* $[x_1 \mapsto d_1, \ldots, x_k \mapsto d_k]$ *we denote the partial function that maps* x_i *to* d_i *and is undefined on other variables. In combination with the notation above, we omit the set braces and write* $V[x_1 \mapsto d_1, \ldots, x_k \mapsto d_k]$ *and* $I[x_1 \mapsto d_1, \ldots, x_k \mapsto d_k]$*.*

Definition 21 (Tarski, model relationship). *Let* $\mathcal{I}$ *be an interpretation and* φ *a formula. The relationship* $\mathcal{I} \models \varphi$*, pronounced "*$\mathcal{I}$ *is a model of* φ*" or "*$\mathcal{I}$ *satisfies* φ*" or "*φ *is true in* $\mathcal{I}$*", and its negation* $\mathcal{I} \not\models \varphi$*, pronounced "*$\mathcal{I}$ *falsifies* φ*" or "*φ *is false in* $\mathcal{I}$*", are defined inductively:*

$\mathcal{I} \models p(t_1, \ldots, t_n)$	*iff* $(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) \in p^{\mathcal{I}}$
$\mathcal{I} \models p$	*iff* $\emptyset \in p^{\mathcal{I}}$
$\mathcal{I} \not\models \bot$	
$\mathcal{I} \models \top$	
$\mathcal{I} \models \neg\varphi$	*iff* $\mathcal{I} \not\models \varphi$
$\mathcal{I} \models (\varphi_1 \wedge \varphi_2)$	*iff* $\mathcal{I} \models \varphi_1$ *and* $\mathcal{I} \models \varphi_2$
$\mathcal{I} \models (\varphi_1 \vee \varphi_2)$	*iff* $\mathcal{I} \models \varphi_1$ *or* $\mathcal{I} \models \varphi_2$
$\mathcal{I} \models (\varphi_1 \Rightarrow \varphi_2)$	*iff* $\mathcal{I} \not\models \varphi_1$ *or* $\mathcal{I} \models \varphi_2$
$\mathcal{I} \models \forall x\varphi$	*iff* $\mathcal{I}[x \mapsto d] \models \varphi$ *for each* $d \in D$
$\mathcal{I} \models \exists x\varphi$	*iff* $\mathcal{I}[x \mapsto d] \models \varphi$ *for at least one* $d \in D$

For a set of formulas S*,* $\mathcal{I} \models S$ *iff* $\mathcal{I} \models \varphi$ *for each* $\varphi \in S$*.*

Definition 22 (Semantic properties). *A formula, or a set of formulas, is*

valid or a tautology	*iff it is satisfied in each interpretation*
satisfiable	*iff it is satisfied in at least one interpr.*
falsifiable	*iff it is falsified in at least one interpr.*
unsatisfiable or inconsistent	*iff it is falsified in each interpretation*

Definition 23 (Entailment). *Let* φ *and* ψ *be formulas or sets of formulas.*

$\varphi \models \psi$	*pronounced: "*φ *entails* ψ*" or "*ψ *is a (logical) conseq. of* φ*", iff for each interpretation* $\mathcal{I}$*: if* $\mathcal{I} \models \varphi$ *then* $\mathcal{I} \models \psi$*.*
$\varphi \models\!\!\mid \psi$	*pronounced: "*φ *is (logically) equivalent to* ψ*", iff* $\varphi \models \psi$ *and* $\psi \models \varphi$*.*

Definition 24 (Herbrand universe, Herbrand base)**.** *The* Herbrand universe *HU is the set of all ground $\mathcal{S}$-terms. The Herbrand base HB is the set of all ground $\mathcal{S}$-atoms. Let P be a program. Herbrand universe HU_P is the set of ground terms that can be constructed with symbols occurring in P. Herbrand base HB_P comprises all ground atoms that can be constructed with symbols ocurring in P.*

Observation 25 (HB_P is finite)**.** *If P is a program with no function symbols other than constants then HB_P is finite because, by Definitions 15 and 24, the number of constants and predicate symbols occurring in P is finite and thus the number of ground atoms that can be generated using these constants and predicates is also finite. Note that under the same assumption HU_P is also finite.*

Definition 26 (Herbrand interpretation)**.** *An interpretation $\mathcal{I} = (D, I, V)$ is a Herbrand interpretation if $\text{dom}(I) = HU$ and $c^I = c$ for each constant c.*

Definition 27 (Herbrand model)**.** *Let S be a set of closed formulas. A* Herbrand model *for S is a Herbrand interpretation which is a model for S.*

Definition 28 (Herbrand interpretation induced by a set of ground atoms)**.** *Let V be some fixed variable assignment in HU. Let $B \subseteq HB$ be a set of ground atoms. Then $HI(B)$ is the Herbrand interpretation with variable assignment V and $p^{HI(B)} = \{(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n) \in B\}$ for each n-ary relation symbol p.*

Definition 29. *Let $\{B_i \mid i \in I, B_i \subseteq HB\}$ be a nonempty set of sets of ground atoms. Then the intersection of Herbrand interpretations $\{HI(B_i) \mid i \in I\}$ is defined as $\bigcap\{HI(B_i) \mid i \in I\} = HI(\bigcap\{B_i \mid i \in I\})$, i.e. it is the Herbrand interpretation induced by the intersection of the sets of ground atoms.*

Definition 30. *Let B_1, B_2 be two sets of ground atoms. Then $HI(B_1) \leqslant HI(B_2)$ iff $B_1 \subseteq B_2$ and $HI(B_1) < HI(B_2)$ iff $B_1 \subset B_2$.*

Definition 31. *A minimal Herbrand model of a program is a $\leqslant$-minimal member of the set of its Herbrand models.*

Proposition 32. *Let P be a definite program.*

- *P has a Herbrand model: $HI(HB)$.*
- *If $\{M_i\}_{i \in I}$ is a non-empty set of Herbrand models for P, then $\bigcap_{i \in I} M_i$ is a Herbrand model for P.*

Definition 33 (Minimal Model of a Definite Program)**.** *Let P be a definite program. The minimal Herbrand model of P, for short: its minimal model, is the intersection of the set of all Herbrand models of P.*

Note that by Proposition 32 the minimal model of a definite program exists and is unique.

7.1.3 *Fixpoint theory*

This section introduces a general fixpoint theory for operators on complete lattices, it mostly follows Lloyd's book [136], Lemma 51 is adapted from [44]. The advantage of using a lattice-based fixpoint theory is that it can be used both for the standard immediate consequence operator defined on sets, see Section 7.1.4, and for the novel extended immediate consequence operators defined on multisets introduced later in this chapter.

Definition 34 (Relation)**.** *Let* S *be a non-empty set. A relation* R *on* S *is a subset of* $S \times S$.

Notation 35. *Let* R *be a relation on a set* S. *The fact that* $(x, y) \in R$ *is denoted* xRy.

Definition 36 (Powerset of a set)**.** *Let* S *be a set. The powerset of* S, *denoted* $\mathcal{P}(S)$, *is the set of all subsets of* S; *i.e.* $\mathcal{P}(S) = \{X \mid X \subseteq S\}$.

Definition 37 (Partial order, partially ordered set)**.** *A relation* R *on a set* S *is a partial order if the following conditions are satisfied:*

- xRx, *for all* $x \in S$ *(*R *is reflexive),*
- xRy *and* yRx *imply* $x = y$, *for all* $x, y \in S$ *(*R *is antisymmetric),*
- xRy *and* yRz *imply* xRz, *for all* $x, y, z \in S$ *(*R *is transitive).*

A nonempty set with a partial order on it is called a partially ordered set, *or shortly a* poset.

Notation 38. *Partial order will be denoted* $\leqslant$ *in cases when it is not given a specific symbol.*

Definition 39 (Upper bound, lower bound)**.** *Let* S *be a set with partial order* $\leqslant$. *Then* $a \in S$ *is an* upper bound *of a subset* $X \subseteq S$ *if for all* $x \in X$ *it holds that* $x \leqslant a$.

Similarly, $b \in S$ *is a* lower bound *of* $X \subseteq S$ *if* $b \leqslant x$, *for all* $x \in X$.

Definition 40 (Least upper bound, greatest lower bound)**.** *Let* S *be a set with a partial order* $\leqslant$. *Then* $a \in S$ *is the* least upper bound *(supremum) of a subset* X *of* S, *denoted* $\text{lub}(X)$, *if* a *is an upper bound of* X *and, for all upper bounds* a' *of* X, *it holds that* $a \leqslant a'$.

Similarly, $b \in S$ *is the* greatest lower bound *(infimum) of a subset* X *of* S, *denoted* $\text{glb}(X)$, *if* b *is a lower bound of* X *and, for all lower bounds* b' *of* X, *it holds that* $b' \leqslant b$.

Definition 41 (Complete lattice)**.** *A partially ordered set* $(S, \leqslant)$ *is a* complete lattice *if* $\text{lub}(X)$ *and* $\text{glb}(X)$ *exist in* $(S, \leqslant)$, *for every subset* X *of* S.

Observation 42. *Let* S *be a set. Then* $(\mathcal{P}(S), \subseteq)$ *is a complete lattice and* $\text{lub}(X) = \bigcup X$ *and* $\text{glb}(X) = \bigcap X$ *for each* $X \subseteq \mathcal{P}(S)$.

Definition 43 (Operator)**.** *Let* $L = (S, \leqslant)$ *be a complete lattice. An* operator *on* L *is a mapping* $\Gamma : S \to S$.

Definition 44 (Monotonic operator)**.** *Let* $L = (S, \leqslant)$ *be a complete lattice and* Γ *an operator on* L. Γ *is* monotonic *iff for all* $x, y \in S$ *such that* $x \leqslant y$ *it holds that* $\Gamma(x) \leqslant \Gamma(y)$.

Definition 45 (Directed set)**.** *Let* $(S, \leqslant)$ *be a complete lattice and* $X \subseteq S$. X *is* directed *if every finite subset of* X *has an upper bound in* X.

Note that the finiteness requirement on the subset of X is meant with respect to the number of members of the subset; the elements themselves may be infinite (if they are sets). This observation is important in proofs with infinite sets and with multisets where the multiset can contain elements with infinite multiplicities. An equivalent definition is that the set X is directed iff each pair of elements has an upper bound in X.

Definition 46 (Continuous operator). *Let* $L = (S, \leqslant)$ *be a complete lattice and* Γ *an operator on* L. Γ *is continuous iff for each directed set* $Y \subseteq S$ *it holds that* $\Gamma(\mathrm{lub}(Y)) = \mathrm{lub}(\Gamma(Y))$.

Lemma 47. *Let* Γ *be a continuous operator on a complete lattice* $(S, \leqslant)$. *Then* Γ *is monotonic.*

Definition 48 (Fixpoint, least fixpoint, greatest fixpoint). *Let* Γ *be a continuous operator on a complete lattice* $(S, \leqslant)$. $a \in S$ *is a fixpoint of* Γ *if* $\Gamma(a) = a$. a *is the* least fixpoint *of* Γ, *denoted* $\mathrm{lfp}(\Gamma)$, *if it is a fixpoint and for all fixpoints* b *of* Γ *it holds that* $a \leqslant b$. *Analogically, the* greatest fixpoint $\mathrm{gfp}(\Gamma)$ *is defined.*

Proposition 49 (Knaster-Tarski, existence of the least and the greatest fixpoint). *Let* Γ *be a monotonic operator on a complete lattice* $(S, \leqslant)$. *Then* Γ *has a least fixpoint and a greatest fixpoint. Moreover:*

$$\mathrm{lfp}(\Gamma) = \mathrm{glb}(\{x \mid \Gamma(x) = x\}) = \mathrm{glb}(\{x \mid \Gamma(x) \leqslant x\}),$$
$$\mathrm{gfp}(\Gamma) = \mathrm{lub}(\{x \mid \Gamma(x) = x\}) = \mathrm{lub}(\{x \mid x \leqslant \Gamma(x)\})$$

Definition 50 (Ordinal powers of a monotonic operator). *Let* Γ *be a monotonic operator on a complete lattice* $(S, \leqslant)$. *For each finite or transfinite ordinal the power of* Γ *is defined as:*

$$\begin{array}{ll} \Gamma^0 = \mathrm{glb}(S) & \textit{(base case)} \\ \Gamma^{\alpha+1} = \Gamma(\Gamma^\alpha) & \textit{(successor case)} \\ \Gamma^\lambda = \mathrm{lub}(\{\Gamma^\beta \mid \beta < \lambda\}) & \textit{(limit case)} \end{array}$$

Let $s \in S$ *and* $n \in \mathbb{N}_1$. *Then*

$$\begin{array}{ll} \Gamma^1(s) = \Gamma(s) & \textit{for } n = 1 \\ \Gamma^{n+1}(s) = \Gamma(\Gamma^n(s)) & \textit{for } n \geqslant 1 \\ \Gamma^\lambda(s) = \mathrm{lub}(\{\Gamma^\beta(s) \mid \beta < \lambda\}) & \textit{(limit case)} \end{array}$$

Lemma 51. *Let* Γ *be a monotonic operator on a complete lattice* $(S, \leqslant)$. *For each ordinal* α *holds:*

1. $\Gamma^\alpha \leqslant \Gamma^{\alpha+1}$
2. $\Gamma^\alpha \leqslant \mathrm{lfp}(\Gamma)$
3. *If* $\Gamma^\alpha = \Gamma^{\alpha+1}$ *then* $\mathrm{lfp}(\Gamma) = \Gamma^\alpha$.

Corollary 52. *Let* Γ *be a monotonic operator on a complete lattice* $(S, \leqslant)$. *Then*

1. *if* $m \leqslant n \in \mathbb{N}$ *then* $\Gamma^m \leqslant \Gamma^n$
2. *if* $n_1, \ldots, n_k \in \mathbb{N}$ *then* $\Gamma^{\max(n_1, \ldots, n_k)}$ *is an upper bound of* $\{\Gamma^{n_1}, \ldots, \Gamma^{n_k}\}$.

Proposition 53. *Let* Γ *be a monotonic operator on a complete lattice* $(S, \leqslant)$. *There exists an ordinal* α *such that* $\Gamma^\alpha = \mathrm{lfp}(\Gamma)$.

Theorem 54 (Kleene). *Let* Γ *be a continuous operator on a complete lattice* $(S, \leqslant)$. *Then* $\mathrm{lfp}(\Gamma) = \Gamma^\omega$.

Proof. It is sufficient to show that $\Gamma^{\omega+1} = \Gamma^\omega$ by Lemma 51.

$$\begin{array}{lll} \Gamma^{\omega+1} & = \Gamma(\Gamma^\omega) & \text{by definition, successor case} \\ & = \Gamma(\mathrm{lub}(\{\Gamma^n \mid n \in \mathbb{N}\})) & \text{by definition, limit case} \\ & = \mathrm{lub}(\{\Gamma(\Gamma^n) \mid n \in \mathbb{N}\}) & \text{by continuity of } \Gamma \text{, note that} \\ & & \{\Gamma^n \mid n \in \mathbb{N}\} \text{ is directed,} \\ & & \text{see Corollary 52} \\ & = \mathrm{lub}(\{\Gamma^{n+1} \mid n \in \mathbb{N}\}) & \text{by definition, successor case} \\ & = \Gamma^\omega & \text{by definition, limit case} \end{array}$$

□

7.1.4 *Immediate consequence operator* T_P

This section introduces the notion of a support which is close to the notion of justification from the field of reason maintenance. Definitions of immediate consequence and of the classical immediate consequence operator are then given in terms of supports. They are however equivalent to the standard definitions which can be reviewed for example in [44].

Definition 55 (Support). *A* support *with respect to a definite program* P *is a pair* $s = (r, \sigma)$, *where* $r \in P$ *and* σ *is a substitution such that* $\mathrm{dom}(\sigma) \subseteq \mathrm{var}(r)$.

For $r = A \leftarrow B_1, \dots, B_n$ *the atom* $A\sigma$ *is called the* head *of* s, *written* $\mathrm{head}(s)$, *and the set* $\{B_1\sigma, \dots, B_n\sigma\}$ *of atoms is called the* body *of* s, *written* $\mathrm{body}(s)$. *If* $n = 0$ *(i.e.,* $r = A \leftarrow \top$*) then* $\mathrm{body}(s) = \emptyset$ *and* s *is called a* base *support, otherwise it is called a* derived *support. Furthermore,* $\mathrm{heads}(S)$ *for a set of supports* S *is the set* $\{\mathrm{head}(s) \mid s \in S\}$. *A support is ground if its head and its body are ground. We say that* s *supports* $\mathrm{head}(s)$.

Observation 56. *A given atom* a *may be the head of infinitely many supports with respect to* P *as long as the substitutions may be arbitrary. However, there are only finitely many rule heads in* P *of which* a *can be an instance. Thus, appropriate restrictions on the substitutions, such as that their codomains are subsets of some finite set of terms, can ensure that there are only finitely many supports with head* a. *In particular, the number of ground supports with respect to* HB_P *is finite, because the codomains of their substitutions are subsets of* HU_P, *which is finite by Observation 25.*

Also note that while the notion of a support is close to the notion of a rule instance, they are not the same as the following example shows.

Example 57. *Consider the following program:*

$r_1 = p(a, a) \leftarrow \top$

$r_2 = q(x) \leftarrow p(x, x)$

$r_3 = q(y) \leftarrow p(a, y)$

The atom $q(a)$ *is derivable both via rule* r_2 *and* r_3 *from the base fact* r_1. *In both cases the resulting rule instance is* $q(a) \leftarrow p(a, a)$. *However there are two different supports* $(r_2, \{x \mapsto a\})$ *and* $(r_3, \{y \mapsto a\})$ *which allows for distinguishing the two different derivations of* $q(a)$ *(see Section 7.1.6).*

Definition 58 (Immediate consequence). *Let* P *be a definite program and* $F \subseteq HB$ *a set of ground atoms. An* immediate consequence *of* P *with respect to* F *is a ground atom* a *such that there is a support* $s = (r, \sigma)$ *with respect to* P *with*

- $\mathrm{dom}(\sigma) = \mathrm{var}(r)$,
- $\mathrm{body}(s) \subseteq F$,
- $\mathrm{head}(s) = a$.

In general there are cases with infinitely many immediate consequences of P with respect to F. Any of the following conditions is sufficient to ensure that the number of immediate consequences is finite:

- *HB* is finite, or
- F is finite and P is range restricted (because then the codomain of σ is a subset of the set of ground terms in F).

Definition 59 (Immediate consequence operator). *Let P be a definite program. The immediate consequence operator T_P for P is*

$$\begin{aligned} T_P\colon\quad & \mathcal{P}(HB) && \to \mathcal{P}(HB) \\ & T_P(F) && = \{a \in HB \mid a \text{ is an immediate conseq. of } P \text{ w.r.t. } F\} \end{aligned}$$

where HB is the Herbrand base, i.e., $F \subseteq HB$ is a set of ground atoms.

Lemma 60 (T_P is continuous). *Let P be a definite program. The immediate consequence operator T_P is continuous.*

This implies that T_P is monotonic (Lemma 47), i.e. if $F' \subseteq F$ then $T_P(F') \subseteq T_P(F)$. In addition, the operator is also monotonic in another sense:

Observation 61 (Monotonicity of T_P w.r.t. P.). *Let P be a definite program and let F be a set of ground atoms. If $P' \subseteq P$ then $T_{P'}(F) \subseteq T_P(F)$. It is easy to see that if a ground atom is an immediate consequence of P' with respect to a set of ground atoms F then it also is an immediate consequence of a superset of P' with respect to the same set of ground atoms F.*

The immediate consequence operator can be used to characterize the minimal Herbrand model of a positive program P (see [44], page 54, for details):

Theorem 62. *Let P be a definite program. Then $\text{lfp}(T_P) = T_P^{\omega} = \{A \in HB \mid P \models A\}$ and $HI(\text{lfp}(T_P))$ is the unique minimal Herbrand model of P.*

Note that for any definite program P, by definition $T_P^0 = \emptyset$ and T_P^1 is the set of ground instances of the base facts in P. If HB is finite then $T_P^{\omega} = T_P^n$ for some $n \in \mathbb{N}$.

7.1.5 *Multisets*

Subsequent sections in this chapter define operators on multisets. A multiset is often described as a bag of elements or as a set where an element can occur multiple times. There are a number of different definitions of multisets and even formalizations of multisets, see for example [208, 31] for an overview. Most of the definitions are equivalent, see [31]. The theory of this dissertation requires that an element can occur infinitely many times in a multiset so that a certain set of multisets is a complete lattice. This kind of extended multisets has also been studied in the literature, see for example [115, 52] and [30, 185] for a more general axiomatic treatment. This section follows the naive (non-axiomatic) definitions of [52] as it describes exactly the kind of extended multisets that are needed in this dissertation. The name "root set", and the $\in_m$ operator are borrowed from [30]. The only non-standard definition in this section is Definition 75. The corresponding Lemma 84 then shows that the Lloyd's standard lattice-based fixpoint theory can be applied to the multiset-based extended immediate consequence operators defined later in this chapter.

Notation 63 (Natural numbers). *$\mathbb{N}$ is the set of all positive natural numbers,i.e. $\mathbb{N} = \{1, 2, 3, \ldots\}$. $\mathbb{N}_0$ is the set of all natural numbers inlcuding zero, i.e. $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$. The set of positive natural numbers can also be denoted $\mathbb{N}_1$ to stress that it does not include zero, i.e. $\mathbb{N} = \mathbb{N}_1$. The usual total ordering $\leqslant$ is assumed.*

Notation 64 (Infinity). *Infinity, denoted ∞, is a number greater than any natural number; i.e. $\forall n \in \mathbb{N}_0 \quad n < \infty$. It is assumed that $\infty + n = n + \infty = \infty$, for all $n \in \mathbb{N}_0$.*

Furthermore, the set $\mathbb{N}_0 \cup \{\infty\}$ is denoted $\mathbb{N}_0^{\infty}$, the set $\mathbb{N}_1 \cup \{\infty\}$ is denoted $\mathbb{N}_1^{\infty}$. Therefore also $\mathbb{N}^{\infty} = \mathbb{N}_1^{\infty}$.

Observation 65. *The infinity symbol ∞ completes the total ordering $\leqslant$ on $\mathbb{N}_0$ so that each subset of $\mathbb{N}_0^\infty$ has the least upper bound (and also the greatest lower bound)* in $\mathbb{N}_0^\infty$.

Definition 66 (Multiset). *Let D be a non-empty set. A* multiset *on D is a pair $A = (D, \mu)$, where $\mu : S \to \mathbb{N}_0^\infty$ is a function from D, the domain set, to the set $\mathbb{N}_0^\infty$. μ is called the multiplicity (or frequency) function of A and D the domain of A.*

Observation 67. *Let A be an ordinary set. Then A can be seen as the multiset (A, χ_A), where χ_A is the characteristic function of A.*

Definition 68 (Empty multiset). *Let $A = (D, \mu)$ be a multiset on D. Then A is the empty multiset on D if $\mu(d) = 0$, for all $d \in D$. The empty multiset is denoted $\emptyset_m^D$. The domain set superscript may be omitted where the domain set is clear from context.*

Operators on multisets always assume that domain sets of the operands are the same. Therefore the domain set of the empty multiset is always clear from context.

Definition 69 (Root of a multiset). *Let $A = (D, \mu)$ be a multiset. The root of A is the set* $\mathrm{root}(A) = \{d \in D \mid \mu(d) > 0\}$.

The root of a multiset A is the set of elements of A with multiplicity greater than zero.

Observation 70. *A multiset can be infinite in two different ways: its root is infinite or it contains an element with infinite multiplicity.*

Definition 71 (Membership relationship). *Let (D, μ) be a multiset.*
$a \in_m^i (D, \mu)$ iff a is a member of (D, μ) with multiplicity $i \in \mathbb{N}_1^\infty$, i.e. iff $a \in D$ and $\mu(a) = i$. $a \in_m (D, \mu)$ holds iff there is an $i > 0$ such that $a \in_m^i (D, \mu)$.

Definition 72 (Submultiset, proper submultiset). *Let $A = (D, \mu_A)$ and $B = (D, \mu_B)$ be multisets. A is a* submultiset *of B, denoted $A \subseteq_m B$, if for all $a \in D$: $\mu_A(a) \leqslant \mu_B(a)$.*

A is a proper submultiset *of B, denoted $A \subset_m B$, if in addition to $A \subseteq_m B$ there is an $a \in S_A$ such that $\mu_A(a) < \mu_B(a)$.*

Notation 73 (Multiset, multiset-builder). *Square brackets "[ ]" are used to denote multisets and to distinguish them from sets (denoted by braces "{ }"), e.g.*

$$\begin{aligned} A = [a, a, b, c] &= (\{a, b, c\}, \{(a, 2), (b, 1), (c, 1)\}),\ \mathrm{root}(A) = \{a, b, c\}, \\ B = [a, b, c] &= (\{a, b, c\}, \{(a, 1), (b, 1), (c, 1)\}),\ \mathrm{root}(B) = \{a, b, c\}. \end{aligned}$$

Analogically to the set-builder (sometimes called set comprehension) notation $\{x \mid P(x)\}$, a multiset-builder notation $[x \mid P(x)]$ is used to specify multisets. The formula $P(x)$ is called the (multiset-)builder condition *and $[x \mid P(x)]$ is the multiset of individuals that satisfy the condition $P(x)$. It is assumed that whenever the $\in_m$ relationship is used in the multiset-builder condition the multiplicities are transferred to the resulting multiset. Also, in cases like $[x \mid \exists y P(x, y)]$, the multiplicity of x in the resulting multiset is $|\{y \mid P(x, y)\}|$. See the following example.*

Example 74. *Let $A = \{a\}$ and $N = [n, n, n]$. Then*

$$\begin{aligned} N_1 &= [\{x, y\} \mid x \in A, y \in_m N] &&= [\{a, n\}, \{a, n\}, \{a, n\}], \\ N_2 &= [\{x, y\} \mid x \in A, y \in \mathrm{root}(N)] &&= [\{a, n\}], \\ N_3 &= \big\{\{x, y\} \mid x \in A, y \in_m N\big\} &&= \{\{a, n\}\}. \end{aligned}$$

Also, $n \in_m N$, $n \in_m^3 N$, $\{a, n\} \in_m^3 N_1$, $\{a, n\} \in_m^1 N_2$, $\{a, n\} \in N_3$.

Definition 75 (Multiset powerset of a set)**.** *Let* S *be a set. The multiset powerset of* S*,* $\mathbb{P}(S)$*, is the set of all multisets on* S*, i.e.* $\mathbb{P}(S) = \{(S, \mu) \mid \mu : S \to \mathbb{N}_0^\infty\}$*,* $\mathbb{P}(\emptyset) = \emptyset$*.*

Note that the value of the multiplicity function can be zero for an element. Therefore $\mathbb{P}(S)$ also includes multisets the roots of which are subsets of S. Also, the multiset powerset of a set should not be confused with the powerset of a *multiset* which is a closely related but different concept usually defined as the set of all submultisets of a multiset. Only the multiset powerset of a set is needed in this dissertation, therefore the concept of a powerset of a multiset is not introduced formally.

Example 76. *Let* $S = \{a, b, c\}$*. Then*

$$\begin{aligned}\mathbb{P}(S) = \quad & \{(S, \mu) \mid \mu : S \to \mathbb{N}_0^\infty\} = \\ & \{\emptyset_m, [a], [a, a], [a, a, a], \ldots, [b], [b, b], [b, b, b], \ldots, [c], [c, c], \\ & \quad [c, c, c], \ldots, [a, b], [a, a, b], \ldots, [a, b, b], \ldots, [a, b, c], \\ & \quad [a, a, b, c], \ldots, [a, b, b, c], \ldots, [a, b, c, c], \ldots, \\ & \quad [a, a, b, b, c], \ldots, \ldots\}.\end{aligned}$$

Let $M = [a, a, b, c]$*. The powerset of* M *is the set of all submultisets of* M*, i.e. the set*

$$\{\ [a, a, b, c], [a, b, c], [a, a, c], [a, a, b], [b, c], [a, c], [a, b], [a, a], \\ [a], [b], [c], \emptyset_m\ \}$$

Lemma 77. *Let* S *be a set. The submultiset relation* $\subseteq_m$ *is a partial order on* $\mathbb{P}(S)$*.*

Proof. Let $A = (D, \mu_A), B = (D, \mu_B), C = (D, \mu_C) \in \mathbb{P}(S)$.

$\subseteq_m$ is reflexive. $A \subseteq_m A$ because $\mu_A(s) \leqslant \mu_A(s)$ for each $s \in S_A$.

$\subseteq_m$ is antisymmetric. If $A \subseteq_m B$ and $B \subseteq_m A$ then for each $s \in D$ it holds that $\mu_A(s) = \mu_B(s)$ because $\mu_A(s) \leqslant \mu_B(s)$ and $\mu_B(s) \leqslant \mu_A(s)$. Therefore $A = B$.

$\subseteq_m$ is transitive. Let if $A \subseteq_m B$ and $B \subseteq_m C$ then for all $s \in D$ it holds that $\mu_A(s) \leqslant \mu_C(s)$ because $\mu_A(s) \leqslant \mu_B(s) \leqslant \mu_C(s)$. Therefore also $A \subseteq_m C$. □

Definition 78 (Multiset union $\cup_m$)**.** *Let* $A = (D, \mu_A)$ *and* $B = (D, \mu_B)$ *be multisets. The union of* A *and* B*, denoted* $A \cup_m B$*, is the multiset* $C = (D, \mu_C)$*, where for all* $d \in D$*:* $\mu_C(d) = \max(\mu_A(d), \mu_B(d))$*.*

Let X *be a set of multisets on a set* D*. Then* $\bigcup_m X = (D, \mu_X)$*, where* $m_X(d) = \text{lub}(\{\mu(d) \mid (D, \mu) \in X\})$*, for all* $d \in D$*.*

Lemma 79. *Let* $S \subseteq \mathbb{P}(HB)$*. Then* $\text{root}(\bigcup_m S) = \bigcup\{\text{root}(s) \mid s \in S\}$*.*

Proof. $\subseteq$: Let $a \in \text{root}(\bigcup_m S)$. Then by definition $\text{lub}(\{\mu(a) \mid (HB, \mu) \in S\}) > 0$ and there is an $s \in S$ such that $a \in_m s$. Therefore also $a \in_m \bigcup_m S$ by the definition of multiset union and therefore $a \in \text{root}(S)$ by the definition of $\text{root}()$.

$\supseteq$: Let $a \in \bigcup\{\text{root}(s) \mid s \in S\}$. Then there is an $s \in S$ such that $a \in \text{root}(s)$. Hence $a \in_m s$, by the definition of $\text{root}()$, and thus $a \in_m \bigcup_m S$ and $a \in \text{root}(\bigcup_m S)$. □

Definition 80 (Multiset intersection $\cap_m$)**.** *Let* $A = (D, \mu_A)$ *and* $B = (D, \mu_B)$ *be multisets. The intersection of* A *and* B*, denoted* $A \cap_m B$*, is the multiset* $C = (D, \mu_C)$*, where for all* $d \in D$*:* $\mu_C(d) = \min(\mu_A(d), \mu_B(d))$*.*

Let X *be a nonempty set of multisets on a set* D*. Then* $\bigcap_m X = (D, \mu_X)$*, where* $\mu_X(d) = \text{glb}(\{\mu(d) \mid (D, \mu) \in X\})$*, for all* $d \in D$*.*

Definition 81 (Multiset sum $\uplus$)**.** *Let $A = (D, \mu_A)$ and $B = (D, \mu_B)$ be multisets. The sum of A and B, denoted $A \uplus B$, is the multiset $C = (D, \mu_C)$, where for all $s \in D$: $\mu_C(s) = \mu_A(s) + \mu_B(s)$.*

Let X be a set of multisets on a set D. Then $\biguplus_m X = (D, \mu_X)$, where $\mu_X(d) = \sum_{(D,\mu) \in X} \mu(d)$, for all $d \in D$.

Definition 82 (Multiset removal $\ominus$)**.** *Let $A = (D, \mu_A)$ be a multiset and $B = (D, \mu_B)$ be a multiset with finite multiplicities, i.e. $\mu_B(d) < \infty$ for all $d \in D$. The removal of B from A, denoted $A \ominus B$, is the multiset $C = (D, \mu_C)$, where for all $d \in D$: $\mu_C(d) = \max(\mu_A(d) - \mu_B(d), 0)$.*

Lemma 83. *Let $A = (D, \mu_A)$, $B = (D, \mu_B)$, and $C = (D, \mu_C)$ be multisets. The following holds:*

$$\begin{aligned}
A \cup_m B &= B \cup_m A \\
A \cap_m B &= B \cap_m A \\
A \cup_m (B \cup_m C) &= (A \cup_m B) \cup_m C \\
A \cap_m (B \cap_m C) &= (A \cap_m B) \cap_m C \\
A \cup_m A &= A \\
A \cap_m A &= A \\
A \cup_m (B \cap_m C) &= (A \cup_m B) \cap_m (A \cup_m C) \\
A \cap_m (B \cup_m C) &= (A \cap_m B) \cup_m (A \cap_m C)
\end{aligned}$$

Proof. Follows directly from definitions by associativity, commutativity, and distributivity of the min and max functions. □

Notice that although operations on multisets are similar to their set counterparts, caution has to be taken in some cases. For example, it is not clear how to define the complement operation on general multisets. Casasnovas defines complement on finite n-bounded multisets (multisets with maximal multiplicity n) [52] and also provides results about infinite distributivity, see Proposition 33 in [52].

Lemma 84. *Let S be a set. Then $(\mathbb{P}(S), \subseteq_m)$ is a complete lattice.*

This lemma corresponds to Proposition 23 in Casasnovas et al. [52]. Casasnovas et al. use a definition of lattices more common in algebra using the meet and join operators while this report follows Lloyd's definitions using an order relation. Therefore, and because of the importance of this result for this work, a slightly different and a bit more detailed proof of the lemma is given here.

Proof. Let $X \subseteq \mathbb{P}(S)$. Let $(S, m) = \bigcup_m X$. Then $(S, \mu) \in \mathbb{P}(S)$ by definition of $\mathbb{P}(S)$ and $\bigcup_m$ and it is an upper bound of X: $(x, \mu_x) \subseteq_m (S, \mu)$, for all $(S, \mu_x) \in X$ because $\mu(s)$ is the supremum of $\{\mu_x(s) \mid (S, \mu_x) \in X\}$ for each $s \in S$ by Definition 78. Let $(S, \mu') \in \mathbb{P}(S)$ be an upper bound of X such that $(S, \mu') \subseteq_m (S, \mu)$. Then $\mu'(s) \leqslant \mu(s)$, for all $s \in S$, and also $\mu'(s) \geqslant \mu(s)$, for all $s \in S$, because $\mu(s)$ is the supremum of the set $\{\mu_x(s) \mid (S, \mu_x) \in X\}$. Therefore $(S, \mu') = (S, \mu)$ and (S, μ) is the least upper bound of X. The top element of the lattice is (S, μ_∞), where $\mu_\infty(s) = \infty$, for all $s \in S$.

Let $(S, \mu) = \bigcap_m X$. Then $(S, \mu) \in \mathbb{P}(S)$ by the definition of $\mathbb{P}(S)$ and $\bigcap_m$ and it is a lower bound of X: $(S, \mu) \subseteq_m (S, \mu_x)$, for all $(S, \mu_x) \in X$ because $\mu(s)$ is the infimum of the set $\{\mu_x(s) \mid (S, \mu_x) \in X\}$ for each $s \in S$ by Definition 80. Let $(S, \mu') \in \mathbb{P}(S)$ be a lower bound of X such that $(S, \mu) \subseteq_m (S, \mu')$. Then

$\mu(s) \leqslant \mu'(s)$, for all $s \in S$, and also $\mu(s) \geqslant \mu'(s)$, for all $s \in S$, because $\mu(s)$ is the infimum of the set $\{\mu_X(s) \mid (S, \mu_X) \in X\}$. Therefore $(S, \mu') = (S, \mu)$ and (S, μ) is the greatest lower bound of X. The bottom element of the lattice is $\emptyset_m = (S, \mu_0)$, where $\mu_0(s) = 0$, for all $s \in S$. □

Note, that because S is fixed in all pairs $(S, \mu) \in \mathbb{P}(S)$, the set $\mathbb{P}(S)$ is a complete lattice iff the set $\{\mu \mid \mu : S \to \mathbb{N}_0^\infty\}$ is a complete lattice. In fact, one of the definitions of multisets [155, 31, 52] uses functions directly (instead of a pair of a set *and* a function) to represent multisets. A multiset is defined as a pair in this dissertation because it seems to reflect the intuition of a multiset and because it stresses the domain set. Also note that the lub() and glb() on multisets correspond to pointwise lub() and glb() on $\mathbb{N}_0^\infty$ [52].

Observation 85. *Let* $X \subseteq \mathbb{P}(S)$. *The proof of Lemma 84 also proves that* $\mathrm{lub}(X) = \bigcup_m X$ *and* $\mathrm{glb}(X) = \bigcap_m X$.

7.1.6 *Support graphs*

This section introduces so called *support graphs* and related notions. A support graph is a data structure inspired by data-dependency networks of classical reason maintenance. In contrast to data-dependency networks, support graphs can accommodate a notion of derivation which is close to the notion of a proof from assumptions in classical mathematical logic. This fact in itself is a contribution as it allows a unified description of the reasoning and reason maintenance algorithms of this dissertation and the proposed definition of a derivation has the advantage that the number of derivations with respect to range restricted Datalog programs is always finite. In contrast, for example in [158] the number of so called derivation trees (a notion defined in [138]) that is used in some of the established reason maintenance methods can be infinite. See Chapter 8 for more details. The notion of a well-founded support is equivalent to the notion of a well-founded justification from reason maintenance [74].

Definition 86 (Support graph, compact support graph)**.** *Let* P *be a definite range restricted program. A* support graph *with respect to* P *is a bipartite directed graph* $G = (S, F, E, l)$ *such that*

- S *is a set of ground supports with respect to* P,
- F *is a set of nodes,*
- l *is a function labelling nodes with ground atoms,* $l : F \to HB_P$,
- $E \subseteq (F \times S) \cup (S \times F)$ *and for each* $(x, y) \in E$
  - *either* $x \in F$, $y \in S$, $l(x) \in \mathrm{body}(y)$
  - *or* $x \in S$, $y \in F$, $l(y) = \mathrm{head}(x)$,
- *for each* $s \in S$
  - *for each* $b \in \mathrm{body}(s)$ *exists an* $x \in F$ *such that* $l(x) = b$ *and* $(x, s) \in E$
  - optionally *for* $a = \mathrm{head}(s)$ *exists a* $y \in F$ *such that* $l(y) = a$ *and* $(s, y) \in E$

If the labelling function l *is moreover injective then the support graph is called a* compact support graph. *In compact support graphs, we identify atom nodes with atoms.*

Note that the definition requires that if a support graph contains a support then it also contains all its body atoms and all edges connecting the support to these atoms. It may or may not contain the head of a support and an edge connecting the head to the support because of the optional point in the definition.

When drawing support graphs, nodes in F are represented as dots and nodes in S as ovals.

A support with empty body does not have an incoming edge. However, in drawings, such supports are emphasised by marking them with the symbol $\top$ and no arrow, although $\top$ is neither a node nor an edge in the support graph.

Example 87.
Let u, v, w, x, y *be variables,* a, b *constants, and* $P = \{r_1, r_2, r_3, r_4\}$ *with*

$$\begin{aligned} r_1 &= p(a, w) \leftarrow \top, \\ r_2 &= p(u, v) \leftarrow r(u, v), \\ r_3 &= r(x, z) \leftarrow p(x, y), q(y, z), \\ r_4 &= q(b, b) \leftarrow \top, \end{aligned}$$

$G = (S, F, E, l)$ *is a compact support graph with respect to* P *for*

$$\begin{aligned} &F = \{n_1, n_2, n_3\} \\ &l = \{n_1 \mapsto p(a, b), n_2 \mapsto q(b, b), n_3 \mapsto r(a, b)\} \\ &S = \{s_1, s_2, s_3\} \textit{ with } \quad s_1 = (r_1, \{w \mapsto b\}) \\ &\qquad\qquad\qquad\qquad\quad s_2 = (r_2, \{u \mapsto a, v \mapsto b\}) \\ &\qquad\qquad\qquad\qquad\quad s_3 = (r_3, \{x \mapsto a, y \mapsto b, z \mapsto b\}) \end{aligned}$$

and E *as indicated by the edges in the graphical representation in Fig. 7.3.*

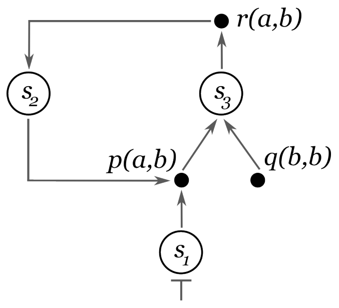


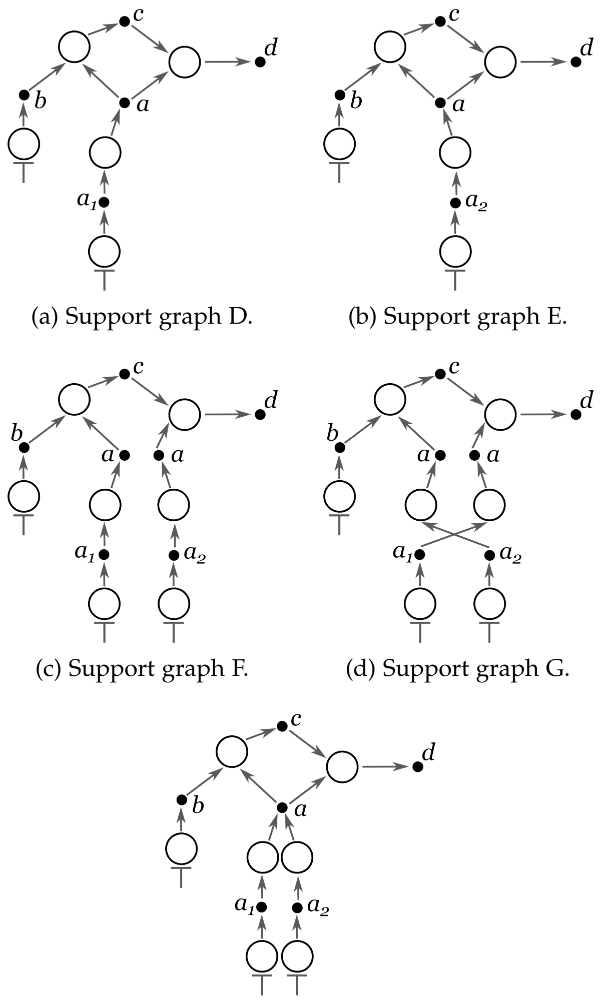
Figure 7.1: An example of a support graph $G = (S, F, E, l)$.

Note that $\mathrm{body}(s_1) = \emptyset$, $\mathrm{body}(s_2) \neq \emptyset$, $\mathrm{body}(s_3) \neq \emptyset$.

Definition 88 (Support graph induced by a set of supports)**.** *Let* T *be a set of ground supports with respect to a definite program* P*. Support graph induced by* T*, denoted* $\mathrm{SG}(T)$*, is a compact support graph* $G = (S, F, E, l)$*, where* $S = T$ *is the set of supports,* F *is a set of nodes such that* $l(F) = \mathrm{heads}(T)$ *is the set of atoms, where* l *is the injective labelling function, and* E *satisfies all the conditions in Definition 86, including the optional one.*

Observation 89. *Let* T *be a set of ground supports with respect to a definite program* P*. Then* $G = \mathrm{SG}(T)$ *is unique up to renaming of its atom nodes (those in* F*) because no atoms and no supports can be repeated in* G *and* G *also satisfies the optional requirement of Definition 86 for all supports.*

Definition 90 (Homomorphic embeddeding). *A support graph* $G = (S_G, F_G, E_G, l_G)$ *is* homomorphically embedded *in a support graph* $H = (S_H, F_H, E_H, l_H)$ *if there is a graph homomorphism* f *of* G *to* H *that also respects labellings, i.e.* $f : S_G \cup F_G \to S_H \cup F_H$ *and* $(\forall x, y \in S_G \cup F_G)(x, y) \in E_G \Rightarrow (f(x), f(y)) \in E_H$ *and* $l_G(x) = l_H(f(x))$ *whenever* $x \in F_G$ *and* $l_G(y) = l_H(f(y))$ *whenever* $y \in F_G$.



(a) Support graph D.

(b) Support graph E.

(c) Support graph F.

(d) Support graph G.

(e) Support graph H.

Figure 7.2: Compact support graphs D and E and non-compact support graphs F and G are homomorphically embedded in the compact support graph H.

Example 91. *Let* P *be the following propositional definite program:*

$d \leftarrow a, c$

$c \leftarrow a, b$

$a \leftarrow a_1$

$a \leftarrow a_2$

$a_1 \leftarrow \top; \qquad a_2 \leftarrow \top; \qquad b \leftarrow \top$

See Figure 7.2 for an example of five support graphs with respect to P. H *is compact because no two atom nodes have the same label and thus the labelling function of* H *is injective.* F *and* G *are not compact because they both have two nodes labelled* a. *It is easy to see that the respective homomorphisms "merge" the two* a *nodes into one while mapping all other nodes to their obvious counterparts.*

Note that there can be at most as many nodes for an atom in a support graph as there are supports of that atom, by Definition 86 and because support nodes cannot be repeated.

Lemma 92. *Let* $G = (S, F, E, l)$ *be a non-compact support graph. Then* G *is homomorphically embedded in* $\mathrm{SG}(S)$.

Proof. Let $H = \mathrm{SG}(S) = (S, F_H, E_H, l_H)$. We will build a homomorphism $f : S \cup F \to S \cup F_H$ from G to H. The set of supports in both G and H is S. Therefore let $f(s) = s$ for all $s \in S$. The only way G and H can differ is if $|F| > |F_H|$. Then there are nodes in F with the same labels. Nodes sharing a label a are mapped by f to the node in F_H which has label a (it is unique because H is compact). Thus f respects labellings and it is a homomorphism because G and H have the same set of supports S which means that the requirements on edges in Definition 86 are also the same (with respect to labels) for G and H. □

Notation 93 (Operations on support graphs)**.** *Let* P *be a definite program,* $G = (S, F, E, l)$ *a support graph with respect to* P, $s, s_1, s_2, \ldots$ *supports with respect to* P *that may or may not be in* S, $a, a_1, a_2, \ldots$ *atoms that may or may not be in* F.

- $G + s := (S', F', E', l')$, *where* $S' = S \cup \{s\}$, *for each* $b \in \mathrm{body}(s) \cup \{\mathrm{head}(s)\}$ *if* $(\nexists n \in F) l(n) = b$ *then add a new node* n *to* F' *and* $l'(n) = b$, $(\forall n \in F) l'(n) = l(n)$, $E' = E \cup \{(b, s) \mid b \in \mathrm{body}(s)\} \cup \{(s, \mathrm{head}(s))\}$.
- $G - s := (S', F, E', l)$, *where* $S' = S \setminus \{s\}$, $E' = E \setminus \{(x, y) \mid x = s$ *or* $y = s\}$. *Note that* F *(and thus also* l*) remains unchanged.*
- $G + \{s_1, s_2, \ldots\} := (((G + s_1) + s_2) + \ldots)$
- $G - \{s_1, s_2, \ldots\} := (((G - s_1) - s_2) - \ldots)$
- $G + a := (S, F', E, l')$, *where* $l(f) = l'(f)$ *for all* $f \in F$ *and if* $(\nexists f \in F) l(f) = a$ *then let* n *be a new node not in* F *and* $F' = F \cup \{n\}$ *and* $l'(n) = a$. *Note that* S *and* E *remain unchanged.*
- $G - a := (S', F' \setminus l^{-1}(a), E', l \restriction_{F' \setminus l^{-1}(a)})$, *where* $(S', F', E', l) = G - \{s \mid a \in \mathrm{body}(s)$ *or* $a = \mathrm{head}(s)\}$, *where* $\restriction$ *stands for "restricted to".*
- $G + \{a_1, a_2, \ldots\} := (((G + a_1) + a_2) + \ldots)$
- $G - \{a_1, a_2, \ldots\} := (((G - a_1) - a_2) - \ldots)$

Definition 94 (Path)**.** *Let* P *be a definite program and* $G = (S, F, E, l)$ *a support graph with respect to* P. *A* path *in* G *is an alternating sequence* $a_0 s_1 \ldots a_{n-1} s_n a_n$ *for* $n > 0$, *with* $s_i \in S$, $l(a_{i-1}) \in \mathrm{body}(s_i)$, $\mathrm{head}(s_i) = l(a_i)$.

The sequence may also start with s_1 *and/or end with* s_n. *The path has* length n *and is said to be:*

- *a path from* a_0 *to* a_n,
- *a path from* s_1 *to* a_n,
- *a path from* a_0 *to* s_n,
- *a path from* a_1 *to* s_n.

A path is cyclic *if* $a_i = a_j$ *for some* $i \neq j$.

A support graph G *is called* acyclic *if there is no cyclic path in* G.

Note, that an acyclic path in a non-compact support graph may homomorphically map to a cyclic path in the respective compact support graph.

Terminology 95 (Supports, depends)**.** *If there is a path from a node* x *to a node* y *in* G*, we say that* x supports y *in* G *and* y depends on x *in* G*. If the path has length 1 we may add the adverb "directly". If the path has length* > 1 *we may add the adverb "indirectly".*

A node in a support graph depends on a rule if it is a support that includes the rule or it depends on a support in the graph that includes the rule.

The convention that a support depends on a rule that it includes simplifies formalization of the dependent-supports algorithm (Algorithm 7.6) in Section 7.4 and of the following formalizations in the reason maintenance chapter, especially in Section 8.4.

Definition 96 (Subgraph)**.** *Let* $G_1 = (S_1, F_1, E_1, l_1)$ *and* $G_2 = (S_2, F_2, E_2, l_2)$ *be support graphs with respect to a definite program* P*.* G_1 *is a* subgraph *of* G_2*, written* $G_1 \subseteq G_2$*, if* $S_1 \subseteq S_2$*,* $F_1 \subseteq F_2$*,* $E_1 \subseteq E_2$*, and* $l_1(n) = l_2(n)$*, for all* $n \in F_1$*.*

Definition 97 (Well-founded support graph)**.** *A support graph* $G = (S, F, E, l)$ *with respect to a definite program* P *is* well-founded*, if it is acyclic, and for each* $n \in F$ *there is an incoming edge* $(s, n) \in E$*.*

Note that all source nodes (i.e. nodes with no incoming edge) of a well-founded support graph are supports with empty body. Being acyclic, a (finite) well-founded support graph has at least one sink node (i.e. a node with no outgoing edge).

Definition 98 (Derivation, well-founded support)**.** *Let* $G = (S, F, E, l)$ *be a support graph with respect to a definite program* P*. Let* a *be an atom and* s *a support.*

A derivation *of* a *in* G *is a minimal (w.r.t* $\subseteq$*) well-founded support graph that is homomorphically embedded in* G *and has a node labelled* a *as its only sink node.*

A derivation *of* s *in* G *is a minimal (w.r.t* $\subseteq$*) well-founded support graph that is homomorphically embedded in* G *and has* s *as its only sink node.*

A support $s \in S$ *is* well-founded in G *if there exists a derivation of* $\text{head}(s)$ *in* G *that includes* s*.*

We say that an atom a *has a derivation with respect to* P*, or that* a *is derivable from* P*, if there is a derivation of the atom in a support graph with respect to* P*.*

Definition 99 (Depth of a derivation)**.** *Let* P *be a definite program and* D *a derivation with respect to* P*. The depth of* D *is the number of supports on the longest path in* D *that ends with the only sink node.*

Example 100. *Consider Figure 7.2. Graphs* D, E, F*, and* G *are all derivations of* d *in* H*. The graph* H *is not a derivation because its proper subgraph* D *is well-founded.*

Lemma 101. *Let* P *be a definite program and let* $a \in HB_P$*. Then the number of derivations of* a *from* P *is finite up to renaming atom nodes.*

Proof. A derivation of a from P is a support graph with respect to P. Note that support graphs are defined only with respect to ground supports. There is only a finite number of ground supports with respect to P (Observation 56) and therefore there is only a finite number of support graphs with respect to P. It follows that there must be only a finite number of derivations of a with respect to P. □

Definition 102 (Depends strongly). *Let* G *be a support graph with respect to a definite range restricted program* P. *A node* x *in* G *depends strongly on a node* y *in* G *if* y *is in every derivation of* x *in* G.

Example 103. *The graph* G *from Example 87 is not well-founded for two reasons: it is cyclic and its atom* $\mathsf{q(b,b)}$ *has no incoming edge.*

Let $\mathsf{G}' = \mathsf{G} + \mathsf{s}_4$, *where* $\mathsf{s}_4 = (\mathsf{r}_4, \emptyset)$. *Then* G' *is not well-founded, but* s_3 *well-founded in* G' *because* $\mathsf{G}' - \mathsf{s}_2$ *is a derivation of* $\mathsf{head}(\mathsf{s}_3)$ *in* G'.

The only support that is well-founded in G *is* s_1.

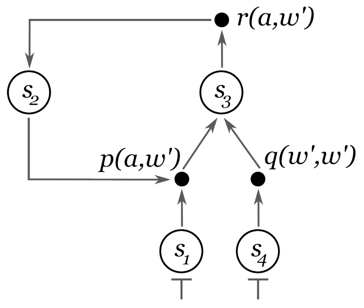


Figure 7.3: An example support graph $\mathsf{G}' = \mathsf{G} + \mathsf{s}_4$.

Terminology 104 (Base fact, derived fact). *Let* $\mathsf{G} = (\mathsf{S}, \mathsf{F}, \mathsf{E}, \mathsf{l})$ *be a support graph with respect to a definite program* P. *Let* $\mathsf{s} \in \mathsf{S}$ *be well-founded in* G *and let* $\mathsf{head}(\mathsf{s}) = \mathsf{a}$.

If $\mathsf{body}(\mathsf{s}) = \emptyset$ *then* a *is said to be (an instance of) a* base fact *in* G. *If* $\mathsf{body}(\mathsf{s}) \neq \emptyset$ *then* a *is said to be a* derived fact *in* G.

See also Definition 15 and Terminology 16.

Note that an instance of a base fact in G is an instance of a base fact in P. Moreover, an atom can have well-founded supports of both kinds.

7.2 CLASSICAL FORWARD CHAINING

Forward chaining, or data-driven evaluation of a program, is a deductive procedure that iteratively derives atoms that a program and a set of atoms entail. Theorem 62 allows for operational understanding of a definite program in terms of forward chaining. The immediate consequence operator $\mathsf{T_P}$ corresponds to one forward chaining iteration: it computes immediate consequences of a program P and a set of atoms. Reaching a fixpoint for a program means that all atoms entailed by that program are computed. The theorem indicates that a fixpoint is reached in at most ω steps.

Using the immediate consequence operator, a naive forward chaining algorithm can be directly formulated. The structure of the naive algorithm is the same for several operators defined in this chapter and therefore the common structure is shared, see Algorithm 7.1. It is then used to specify an operator-specific algorithm, see Algorithm 7.2 for the naive forward chaining algorithm for the $\mathsf{T_P}$ operator. The algorithm is naive because all previously computed results are recomputed in each iteration.

Algorithm 7.1: Naive forward chaining

```
Name
  naive-fw-chaining($X_P$)
```

```
Input
  $X_P$ — one of $T_P$, $skT_P$, $scT_P$, $dcT_P$ operators defined on a lattice $L$
      , P is a definite range restricted program

Output
  F — a set, $F = X_P^\omega$, $F \in L$

Variables
  $F'$ — a set, $F' \in L$

Initialization
  F := $\emptyset$; $F'$ := $X_P(\emptyset)$

begin
  while $F \neq F'$
  begin
    F := $F'$
    $F'$ := $X_P(F)$
  end

  return F
end
```

Algorithm 7.2: Naive T_P forward chaining

```
Name
  naive-tp-fw-chaining(P)

Input
  P — a definite range restricted program

Output
  F — a set of ground atoms, $F = T_P^\omega$

begin
  return naive-fw-chaining($T_P$)
end
```

Proposition 105. *Let X_P be a continuous operator on a complete lattice. Then Algorithm 7.1 computes the least fixpoint of X_P.*

Proof. i-th iteration of the algorithm computes X_P^i. X_P is continuous therefore X_P^ω is the least fixpoint of X_P by Theorem 54. □

Proposition 106. *Let P be a definite range restricted program. Algorithm 7.2 computes $T_P^\omega = \text{lfp}(T_P)$ in a finite number of steps.*

Proof. Algorithm 7.2 only calls Algorithm 7.1 with T_P as input, therefore the first part of the proposition follows immediately from Proposition 105 and the fact that T_P is continuous (Lemma 60).

F either grows in each iteration or remains unchanged because T_P is monotonic by its continuity and Lemma 47. The number of ground atoms is finite because no other functional symbols other than constants are assumed and the program is finite, cf. Observation 25. Therefore F remains unchanged after a finite number of steps and the algorithm terminates. □

The idea of *semi-naive* (or *incremental*) evaluation of definite programs lies in the observation that new ground atoms can be derived only if a rule is instantiated using at least one ground atom that was new in the last iteration (otherwise it was derived in an earlier iteration). A semi-naive procedure therefore has to keep record of newly derived facts.

Definition 107 (Semi-naive T_P mapping)**.** *Let P be a definite range restricted program. The semi-naive immediate consequence mapping T_P for P is*

$$\begin{aligned} &T_P : \mathcal{P}(HB) \times \mathcal{P}(HB) \to \mathcal{P}(HB) \\ &T_P(F, \Delta) = \{ \quad a \in HB \mid (\exists s = (r, \sigma)) r = H \leftarrow B_1, \dots, B_n \in P, \\ &\qquad\qquad\qquad (\exists 1 \leqslant j \leqslant n) B_j\sigma \in \Delta, \\ &\qquad\qquad\qquad \mathrm{dom}(\sigma) = \mathrm{var}(r), \mathrm{body}(s) \subseteq F, \mathrm{head}(s) = a \,\} \end{aligned}$$

where HB is the Herbrand base, i.e., $F \subseteq HB$ and $\Delta \subseteq HB$ are sets of ground atoms.

Algorithm 7.3: Semi-naive T_P forward chaining

```
Name
  semi-naive-fw-tp-chaining(P)

Input
  P — a definite range restricted program

Output
  F — a set of ground atoms, $F = T_P^\omega$

Variables
  Δ — a set of ground atoms

Initialization
  F := ∅; Δ := $T_P(\emptyset)$

begin
  while $\Delta \neq \emptyset$
  begin
    F := $F \cup \Delta$
    $\Delta_F$ := $T_P(F, \Delta)$      {new atoms}
    Δ := $\Delta_F \setminus F$        {forbid redundant and cyclic derivations}
  end

  return F
end
```

Algorithm 7.3 is a semi-naive forward chaining procedure formulated using the semi-naive T_P mapping. In each iteration of the while cycle, new ground atoms are collected in Δ by instantiating rules of P using already known atoms from F *and* using at least one atom from Δ.

Lemma 108. *Let P be a definite range restricted program.* $T_P(T_P^{n+1}, T_P^{n+1} \setminus T_P^n) = T_P^{n+2} \setminus T_P^{n+1}$.

Proof. Follows directly from Definition 107. □

Proposition 109. *Let* P *be a definite range restricted Datalog program. Algorithm 7.3 terminates and computes the least fixpoint of* T_P*.*

Proof. Let n be the iteration number and let F_n and Δ_n be the F and Δ computed in the n-th iteration (i.e. after n executions of the while body).

We will show that $F_n = T_P^n$ and $\Delta_n = T_P^{n+1} \setminus T_P^n$ by induction on n.

$n = 0$.

$F_0 = \emptyset = T_P^0$ and $\Delta_0 = T_P(\emptyset) = T_P^1 = T_P^1 \setminus T_P^0$

$n \rightsquigarrow n+1$.

First for Δ_i:

$$\begin{aligned}
\Delta_{n+1} &= T_P(F_{n+1}, \Delta_n) \setminus F_{n+1} && \text{from Alg. 7.3 for } \Delta_{n+1}\\
&= T_P(F_n \cup \Delta_n, \Delta_n) \setminus (F_n \cup \Delta_n) && \text{from Alg. 7.3 for } F_{n+1}\\
&= T_P(\, T_P^n \cup (T_P^{n+1} \setminus T_P^n)\,, \\
&\qquad T_P^{n+1} \setminus T_P^n) \setminus (T_P^n \cup (T_P^{n+1} \setminus T_P^n)) && \text{induction hypothesis}\\
&= T_P(T_P^{n+1}, T_P^{n+1} \setminus T_P^n) \setminus T_P^{n+1} && \text{Lemma 51 (1.)}\\
&= T_P^{n+2} \setminus T_P^{n+1} && \text{Lemma 108}
\end{aligned}$$

Therefore $\Delta_i = T_P^{i+1} \setminus T_P^i$, for all $i \in \mathbb{N}_0$. Now for F_i:

$$\begin{aligned}
F_{n+1} &= F_n \cup \Delta_n && \text{from Algorithm 7.3 for } F_{n+1}\\
&= T_P^n \cup \Delta_n && \text{induction hypothesis}\\
&= T_P^n \cup (T_P^{n+1} \setminus T_P^n) \\
&= T_P^{n+1} && T_P^n \subseteq T_P^{n+1} \text{ (Lemma 51)}
\end{aligned}$$

$T_P^\omega = \text{lub}(\{T_P^\beta \mid \beta < \omega\})$ by Definition 50 and $F_\omega = \bigcup\{T_P^n \mid n < \omega\}$. Therefore the result of Algorithm 7.3 is T_P^ω which is the least fixpoint of T_P by Theorem 54 and continuity of T_P.

Δ always contains only those atoms that were not generated in a previous iteration because the set of already generated atoms is subtracted when a new Δ is computed. The number of ground atoms with no functional symbols other than constants is finite. Therefore Δ is empty and the algorithm terminates. □

COMPLEXITY. Let P be a range restricted Datalog program and let $B \subseteq P$ be the set of base facts in P. It is a well-known result [146] that T_P^ω can be computed in $O(n^k)$ time, where n is the number of terms (i.e. constants because all atoms in B are ground) in B and k is the maximum over all rules in P of the number of variables in that rule. To compute T_P^ω in the given time, one can generate all instances of all rules in P using the constants in B and then apply the well-known algorithm for the deductive closure of a set of ground Horn clauses [71], which runs in linear time (note that the satisfiability problem is NP-complete for the general class of *all* propositions). Algorithms 7.2 and 7.3 therefore run in time $O(n^k)$.

7.3 SUPPORT KEEPING FORWARD CHAINING

The original reason maintenance methods and some of the algorithms presented in Chapter 8 require a support graph. Traditionally, support graphs were recorded by a reason maintenance component in form of so-called justifications passed to the component by a reasoner. This section presents an extended immediate consequence operator which keeps record of supports directly. To the best of our knowledge, this has not yet been done in the literature and it has the advantage that compact support graphs induced by the fixpoint of this operator can thus be easily and simply described. These compact support graphs are studied in this dissertation and correspond directly to the data-dependency networks studied in classical reason maintenance.

Definition 110 (Support Herbrand base)**.** *Let* P *be a definite program. The support Herbrand base sHB is the set of all ground supports.* sHB_P *is the set of all ground supports with respect to* P*.*

Lemma 111. sHB_P *is finite.*

Proof. P comprises a finite number of rules by Definition 15 and HB_P is finite by Observation 25. In a finite number of rules, there is only a finite number of variables and for any support (r, σ) it by definition holds that $\mathrm{dom}(\sigma) \subseteq \mathrm{var}(r)$. Therefore, for each rule in P, there is only a finite number of substitutions that make it ground. Therefore there is only a finite number of ground supports with respect to P, i.e. sHB_P is finite. □

Definition 112 (Support keeping immediate consequence operator)**.** *Let* P *be a definite range restricted program. The support keeping immediate consequence operator* skT_P *for* P *is the mapping:*

$$skT_P : \mathcal{P}(sHB_P) \to \mathcal{P}(sHB_P)$$

$$skT_P(F) = \{\quad s = (r, \sigma) \in sHB_P \mid r = H \leftarrow B_1, \ldots, B_n \in P,$$
$$\mathrm{dom}(\sigma) = \mathrm{var}(r), \mathrm{body}(s) \subseteq \mathrm{heads}(F)\ \}$$

where $F \subseteq sHB_P$ *is a set of supports with respect to* P*.*

Let us first explore skT_P on an example.

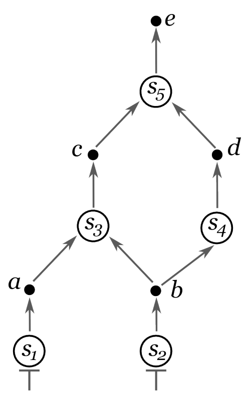


Figure 7.4: The compact support graph induced by the fixpoint of skT_P for P from Example 113.

Example 113. *Consider the following program* P*:*

$$r_1 = \quad a \leftarrow \top$$
$$r_2 = \quad b \leftarrow \top$$
$$r_3 = \quad c \leftarrow a, b$$
$$r_4 = \quad d \leftarrow b$$
$$r_5 = \quad e \leftarrow c, d$$

Let us explore ordinal powers of skT_P*:*

$$\begin{aligned}
skT_P^0 &= \emptyset \\
skT_P^1 &= \{ (r_1,\emptyset),(r_2,\emptyset) \} \\
skT_P^2 &= \{ (r_1,\emptyset),(r_2,\emptyset),(r_3,\emptyset),(r_4,\emptyset) \} \\
skT_P^3 &= \{ (r_1,\emptyset),(r_2,\emptyset),(r_3,\emptyset),(r_4,\emptyset),(r_5,\emptyset) \}
\end{aligned}$$

It is easy to verify that $skT_P^i = skT_P^3$ *for all* $i \geqslant 3$ *in this example. See the compact support graph induced by* skT_P^3 *in Figure 7.4.*

Note that $heads(skT_P(F)) = T_P(heads(F))$.

Lemma 114. skT_P *is continuous.*

Proof. It is easy to see that skT_P is monotonic: let $M \subseteq M' \subseteq sHB_P$ then $skT_P(M) \subseteq skT_P(M')$ because if $body(s) \subseteq M$ for a support s then also $body(s) \subseteq M'$.

To show that skT_P is continuous we have to show (by Definition 46 and Observation 42) that $skT_P(\bigcup Y) = \bigcup\{skT_P(M) \mid M \in Y\}$ for each directed $Y \subseteq \mathcal{P}(sHB_P)$. Let Y be a directed subset of $\mathcal{P}(sHB_P)$.

$\bigcup\{skT_P(M) \mid M \in Y\} \subseteq skT_P(\bigcup Y)$ follows directly from monotonicity of skT_P: $skT_P(M) \subseteq skT_P(\bigcup Y)$ for each $M \in Y$.

Let us prove $skT_P(\bigcup Y) \subseteq \bigcup\{skT_P(M) \mid M \in Y\}$. Let $s = (r, \sigma) \in skT_P(\bigcup Y)$. Then $r \in P$ and $body(s) \subseteq \bigcup Y$. Therefore there is $Y_i \in Y$ such that $b_i \in Y_i$ for each $1 \leqslant i \leqslant n$, where $body(s) = \{b_1, \ldots, b_n\}$. $\{Y_1, \ldots, Y_n\}$ is a finite subset of Y, therefore there is $Y' \in Y$ such that $\bigcup\{Y_1, \ldots, Y_n\} \subseteq Y'$ because Y is directed. Therefore $body(s) \subseteq Y'$ and therefore $s \in skT_P(Y') \subseteq \bigcup\{skT_P(M) \mid M \in Y\}$. □

Algorithm 7.4: Naive forward chaining with keeping supports

```
Name
  naive-sktp-fw-chaining(P)

Input
  P — a nonempty finite definite range restricted program

Output
  F — a set of supports, $F = skT_P^\omega$

begin
  return naive-fw-chaining($skT_P$)
end
```

Proposition 115. *Let* P *be a definite range restricted program. Algorithm 7.4 terminates and computes the least fixpoint of* skT_P.

Proof. Algorithm 7.4 terminates because sHB_P is finite (Lemma 111) and skT_P is monotonic because it is continuous (Lemma 47).

The rest of the claim follows directly from Proposition 105 by using Lemma 114 and the fact that $(\mathcal{P}(sHB_P), \subseteq)$ is a complete lattice (Observation 42). □

Much like in the case of classical forward chaining, a semi-naive version of the support keeping immediate consequence operator can be defined.

Definition 116 (Semi-naive support keeping immediate consequence mapping)**.** *Let P be a definite range restricted program. The semi-naive support keeping immediate consequence mapping skT_P for P is the mapping:*

$$\begin{aligned} skT_P &: \mathcal{P}(sHB_P) \times \mathcal{P}(HB_P) \to \mathcal{P}(sHB_P) \\ skT_P(F, \Delta) &= \{ \quad s = (r, \sigma) \in sHB_P \mid r = H \leftarrow B_1, \ldots, B_n \in P, \\ &\qquad (\exists 1 \leqslant j \leqslant n) B_j\sigma \in \Delta, \\ &\qquad dom(\sigma) = var(r), body(s) \subseteq heads(F) \ \} \end{aligned}$$

where $F \subseteq sHB_P$ is a set of supports and $\Delta \subseteq HB_P$ is a set of atoms.

Lemma 117. *Let P be a definite range restricted program. Then $T_P^i = heads(skT_P^i)$, for all $i \in \mathbb{N}_0$, and $T_P^\omega = heads(skT_P^\omega)$.*

Proof. By induction on i.

$i = 0$: $T_P^0 = \emptyset$ and $skT_P^0 = \emptyset$ and $heads(\emptyset) = \emptyset$.

$i \rightsquigarrow i+1$: $\subseteq$: Let $a \in T_P^{i+1}$. Then there is a support s of a such that $body(s) \subseteq T_P^i$, therefore also $body(s) \subseteq heads(skT_P^i)$ by the induction hypothesis. Therefore $s \in skT_P^{i+1}$, i.e. $a \in heads(skT_P^{i+1})$.

$\supseteq$: Let $a \in heads(skT_P^{i+1})$. Then there is a support s of a such that $body(s) \subseteq heads(skT_P^i)$, i.e. $body(s) \subseteq T_P^i$ by the induction hypothesis and thus $a \in T_P^{i+1}$.

Naive skT_P forward chaining terminates by Proposition 115 therefore there is an $i \in \mathbb{N}_0$ such that $T_P^i = T_P^\omega$ and therefore $T_P^\omega = heads(skT_P^\omega)$. □

Lemma 118. $skT_P(skT_P^n, T_P^n \setminus T_P^{n-1}) = skT_P^{n+1} \setminus skT_P^n$.

Proof. First, let us show that $skT_P(skT_P^n, T_P^n \setminus T_P^{n-1}) \subseteq skT_P^{n+1} \setminus skT_P^n$. Let $s \in skT_P(skT_P^n, T_P^n \setminus T_P^{n-1})$. Then $body(s) \subseteq heads(skT_P^n)$ from the definition of $skT_P(F, \Delta)$. Therefore $s \in skT_P^{n+1}$, from the definition of skT_P and the definition of ordinal powers of a monotonic operator. There is a $b \in body(s)$ such that $b \in T_P^n \setminus T_P^{n-1}$ by the definition of $skT_P(F, \Delta)$. Thus $b \notin T_P^{n-1}$ and therefore $s \notin skT_P^n$ by the definition of skT_P and the simple observation that skT_P cannot derive support for an atom sooner than T_P derives the atom (Lemma 117). In summary $s \in skT_P^{n+1} \setminus skT_P^n$.

Let us show that $skT_P^{n+1} \setminus skT_P^n \subseteq skT_P(skT_P^n, T_P^n \setminus T_P^{n-1})$. Let $s \in skT_P^{n+1} \setminus skT_P^n$. Then $body(s) \subseteq heads(skT_P^n)$ from monotonicity and the definition of skT_P. There must be a $b \in body(s)$ such that $b \notin heads(skT_P^{n-1})$ because $s \notin skT_P^n$. Therefore $s \in skT_P(skT_P^n, T_P^n \setminus T_P^{n-1})$ by the definition of $skT_P(F, \Delta)$. □

Algorithm 7.5: Semi-naive forward chaining with keeping supports

```
Name
  semi-naive-skTP-fw-chaining(P)

Input
  P — a definite range restricted program

Output
  F — a set of supports, $F = skT_P^\omega$

Variables
  $\Delta$ — a set of supports
  $\Delta_F$ — a set of supports
```

```
Initialization
  $F := \emptyset$; $\Delta_F := skT_P(\emptyset)$; $\Delta := skT_P(\emptyset)$

begin
  while $\Delta \neq \emptyset$
  begin
    $F := F \cup \Delta_F$
    $\Delta_F := skT_P(F, heads(\Delta))$                    {new supports}
    $\Delta := \Delta_F \setminus \{s \in \Delta_F \mid head(s) \in heads(F)\}$ {forbid redundant and
                                                     cyclic derivations}
  end

  return F
end
```

Algorithm 7.5 semi-naively computes a set of supports S which corresponds to all rule instances generated during a classical forward chaining such as in Algorithm 7.3. It differs from classical semi-naive forward chaining because some of the new supports may support an atom that was derived in a previous iteration. Such supports belong to the fixpoint but have to be removed from further computation to avoid redundant and cyclic derivations.

Proposition 119. *Let* P *be a definite range restricted program. Algorithm 7.5 terminates and computes the least fixpoint of* skT_P*.*

Proof. Analogical to Proposition 109 by using Lemma 111, Lemma 114, and Lemma 118. □

Algorithms 7.4 and 7.5 derive all the information that normal forward chaining derives plus the supports in addition. The correspondence is shown by Lemma 117, where the function heads() projects supports out and leaves only head atoms.

COMPLEXITY. Algorithm 7.5 has the same worst case time complexity as classical semi-naive forward chaining, i.e. $O(n^k)$, because it is essentially the same as classical semi-naive forward chaining only with remembering supports.

7.4 DEPENDENT SUPPORTS ALGORITHM

Algorithms in Chapter 8 that process rule removal updates of a program, and its fixpoint, often require only a specific subgraph of the whole support graph corresponding to the fixpoint. The subgraph of interest is the graph induced by the set of supports that depend on a rule to be removed.

The algorithm to compute the set of such supports is a simple modification of the semi-naive forward chaining, see Algorithm 7.6. Analogous algorithms can be easily formulated for all the other (extended) operators described in this chapter. The idea is the same and the actual differences are small and thus only Algorithm 7.6 is described here as a representative example. See also Observation 122.

Algorithm 7.6: Supports depending on a rule from a set of rules.

```
Name
  dependent-supports(P, L, D)

Input
  P — a definite range restricted program
  L — a set of supports, $L = skT_P^\omega$
  D — a set of rules, $D \subseteq P$

Output
  F — a set of supports from L that depend on a rule from D

Variables
  Δ — a set of supports
  $\Delta_F$ — a set of supports

Initialization
  F := ∅; $\Delta_F$ := $skT_D(L)$; Δ := $skT_D(L)$

begin
  while $\Delta \neq \emptyset$
  begin
    F := $F \cup \Delta_F$
    $\Delta_F$ := $skT_P(L, heads(\Delta))$                {new supports}
    Δ := $\Delta_F \setminus \{s \in \Delta_F \mid head(s) \in heads(F)\}$ {forbid redundant and
                                                  cyclic derivations}
  end

  return F
end
```

Algorithm 7.6 is similar to the semi-naive support keeping Algorithm 7.5. One difference is that Δ and Δ_F are initialized with supports to be removed. The second difference is that the skT_P mapping is applied to the fixpoint and Δ which means that the computation is in some cases faster, see Figure 7.5.

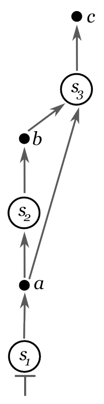


Figure 7.5: Support graph $SG(skT_P^\omega)$ for program P from Example 120.

Example 120. *Let* $P = \{a \leftarrow \top; b \leftarrow a; c \leftarrow a, b\}$. *See the corresponding induced support graph in Figure 7.5. Support* s_3 *is derived only in the second iteration of* *support keeping forward chaining while it is derived already in the first iteration of Algorithm 7.6 with the input* $L = skT_P^\omega$, $D = \{a \leftarrow \top\}$.

Lemma 121. *Algorithm 7.6 is correct in the sense that it computes exactly the supports from* skT_P^ω *that depend on a rule from* D *in* $SG(skT_P^\omega)$.

Proof. Let i be the iteration number and let F_i and $\Delta_{F,i}$ and Δ_i be the F, Δ_F and Δ computed in the i-th iteration (i.e. after i executions of the while body). Let $G = SG(skT_P^\omega)$, all uses of the "depends" relationship are with respect to G in this proof.

We will show by induction on i that $s \in F_i$ iff $s \in skT_P^\omega$ depends on a rule from D and there is a sequence of at most $\leqslant i$ supports as evidence.

$i = 0$: Trivial because $F_0 = \emptyset$.

$i = 1$: $F_1 = F_0 \cup \Delta_{F,0} = \emptyset \cup skT_D(skT_P^\omega)$. That is if $s \in F_1$ then $s \in skT_D(skT_P^\omega)$ and thus s depends on a rule from D and s itself is the evidence. Conversely, if $s \in skT_P^\omega$ and it is evidenced by a single support then the support must directly depend on a rule in D and therefore $s \in skT_D(skT_P^\omega) = F_1$.

$i \rightsquigarrow i+1$.

$\Rightarrow$: Let $s \in F_{i+1}$. $F_{i+1} = F_i \cup \Delta_{F,i}$. If $s \in F_i$ then the statement holds by induction hypothesis. If $s \in \Delta_{F,i}$ then $s \in skT_P(skT_P^\omega, heads(\Delta_{i-1}))$. Therefore there is a support $t \in \Delta_{i-1}$ such that $head(t) \in body(s)$. $\Delta_{i-1} \subseteq F_i$ and thus, by induction hypothesis, t depends on a rule from D and there is a sequence of at most i supports that shows it. Adding t to the end of the sequence creates one of length at most $i+1$ that shows that s depends on a rule from D.

$\Leftarrow$: Let $s \in skT_P^\omega$ such that it depends on a rule from D and there is a sequence of supports $t_1, \ldots, t_i, t_{i+1}$ that shows it. Then by induction hypothesis $t_i \in F_i$. Therefore there is a $j < i$ such that $head(t_i) \in heads(\Delta_j)$ (because $F_i = \bigcup_{k=0}^{i} \Delta_{F,k}$ and "Δ is Δ_F without repetitions"). Therefore $t_{i+1} \in skT_P(skT_P^\omega, heads(\Delta_j))$, and thus either $t_{i+1} \in \Delta_{F,j+1}$ or $t_{i+1} \in F_j$, in either case $t_{i+1} \in F_{j+1} \subseteq F_{i+1}$.

Let m be the maximal length of non-cyclic paths in G. If $\Delta_{\left\lceil \frac{m}{2} \right\rceil + 1} \neq \emptyset$ (one iteration of the algorithm extends paths by up to two nodes) then there is path of length at least $m+1$ in G that is either non-cyclic, which is in contradiction with the choice of m or it is cyclic, which is in contradiction with the construction of Δ in the algorithm – supports supporting an atom for which a support was previously computed are excluded. m is finite (follows from Observation 25 and groundness of support graphs) and thus Algorithm 7.6 terminates. □

Observation 122. *The definition of* $skT_P(F, \Delta)$ *uses only* $heads(F)$. *Therefore, after a trivial modification and by Lemma 117, Algorithm 7.6 can compute dependent supports also when it is given* T_P^ω *instead of* skT_P^ω *as input.*

Observation 123. *Let* SU *be the output of Algorithm 7.6 for some input* P, skT_P^ω, D. *Then* $heads(SU)$ *are all the atoms from* T_P^ω *that depend on a rule from* D *in* $G = SG(skT_P^\omega)$. *Each support* s *in* SU *depends on a rule from* D *and thus its head depends on that rule in* G. *If an atom in* T_P^ω *depends on a rule in* D *it is because at least one of its supports depends on a rule from* D *and thus such a support is in* SU.

COMPLEXITY. The worst case time complexity of Algorithm 7.6 can be estimated also as $O(n^k)$ because, in the worst case, the set of dependent supports can be simply computed from scratch. While the algorithm uses the already computed fixpoint to speed up computation, in the worst case it however runs only as fast as normal forward chaining. The worst case occurs when for each support s it holds that $\text{body}(s) \subseteq T_P^i$ and $\text{body}(s) \cap T_P^{i-1} = \emptyset$, i.e. for each support all its body atoms are derived in the same forward chaining iteration when computing from scratch.

7.5 SUPPORT COUNTING FORWARD CHAINING

Keeping record of all supports is not always necessary because they can be easily rederived as it is shown by Observation 122. Reason maintenance and explanation may however require more information – for example for each atom the number of supports it has in skT_P^ω. This section defines an extended support counting immediate consequence operator, denoted scT_P, which derives atoms and keeps track of their number of supports by means of multisets. Note that the number of supports is clearly different from the number of derivations of an atom, see also Section 8.6.

The support counting immediate consequence operator scT_P is defined as follows:

Definition 124 (Support counting immediate consequence operator scT_P). *Let* P *be a finite definite range restricted program. The* support counting immediate consequence operator scT_P *for* P *is:*

$$\begin{aligned} scT_P\colon\ & \mathbb{P}(HB_P) \to \mathbb{P}(HB_P) \\ & scT_P(S) = [\ \ H\sigma \in HB_P \mid \exists\, s = (r, \sigma), \\ & \qquad\qquad r = H \leftarrow B_1, \ldots, B_n \in P, \\ & \qquad\qquad \text{body}(s) \subseteq \text{root}(S), \text{dom}(\sigma) = \text{var}(r)\ \ ] \end{aligned}$$

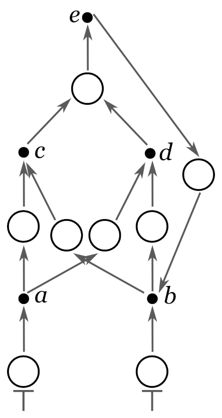


Figure 7.6: The compact support graph $SG(skT_P^\omega)$ for P from Example 125.

Example 125. *Consider the following program* P*:*

$$\begin{array}{ll} a \leftarrow \top & b \leftarrow \top \\ c \leftarrow a & d \leftarrow a \\ c \leftarrow b & d \leftarrow b \\ e \leftarrow c, d & b \leftarrow e \end{array}$$

See the corresponding support graph $SG(skT_P^\omega)$ *in Figure 7.6. Let us explore ordinal powers of* scT_P*:*

$$\begin{aligned} scT_P^0 &= \emptyset_m \\ scT_P^1 &= [\quad a, b \quad] \\ scT_P^2 &= [\quad a, b, c, c, d, d \quad] \\ scT_P^3 &= [\quad a, b, c, c, d, d, e \quad] \\ scT_P^4 &= [\quad a, b, c, c, d, d, e, b \quad] \end{aligned}$$

It is easy to verify that $scT_P^i = scT_P^4$ *for all* $i \geqslant 4$ *in this example.*

Lemma 126. scT_P *is continuous.*

Proof. It is easy to see that scT_P is monotonic: let $M, M' \in \mathbb{P}(HB_P)$ and $M \subseteq_m M'$ then $scT_P(M) \subseteq_m scT_P(M')$ because if $body(s) \subseteq root(M)$ for a support s then $body(s) \subseteq root(M')$ because $root(M) \subseteq root(M')$ by $M \subseteq_m M'$ and the definition of the submultiset relation.

We have to show that $scT_P(\bigcup_m Y) = \bigcup_m \{scT_P(M) \mid M \in Y\}$ for each directed $Y \subseteq \mathbb{P}(HB_P)$ by Definition 46 (continuous operator) and Observation 85. Let Y be a directed subset of $\mathbb{P}(HB_P)$.

For each $M \in Y$: $scT_P(M) \subseteq_m scT_P(\bigcup_m Y)$ by monotonicity of scT_P. Thus $\bigcup_m \{scT_P(M) \mid M \in Y\} \subseteq_m scT_P(\bigcup_m Y)$.

Let us show $scT_P(\bigcup_m Y) \subseteq_m \bigcup_m \{scT_P(M) \mid M \in Y\}$. Let $H\sigma \in_m^\mu scT_P(\bigcup_m Y)$.

Case $\mu \in \mathbb{N}$. Then there are μ different supports with respect to P that satisfy the condition of Definition 124 for $H\sigma$. Let $(H \leftarrow B_1, \dots, B_n, \sigma)$ be one such support. For each $1 \leqslant i \leqslant n$ it holds that $B_i\sigma \in_m \bigcup_m Y$. Thus for each $1 \leqslant i \leqslant n$ there is a $Y_i \in Y$ such that $B_i\sigma \in_m Y_i$ by the definition of the multiset union. The set $\{Y_1, \dots, Y_n\}$ is a finite subset of the directed set Y. Therefore there is an $M \in Y$ such that $\bigcup_m \{Y_1, \dots, Y_n\} \subseteq_m M$. Therefore $H\sigma \in_m \bigcup_m \{scT_P(M) \mid M \in Y\}$ and its multiplicity is at least μ by repeating the argument μ times.

Case of $\mu = \infty$. If the multiplicity of a $B_i\sigma$ is infinite in a multiset in Y then $H\sigma$ is generated infinitely many times in $\bigcup_m \{scT_P(M) \mid M \in Y\}$, from the definition of scT_P, and thus $H\sigma \in_m^\infty \bigcup_m \{scT_P(M) \mid M \in Y\}$. If, on the other hand, $H\sigma \in_m^\infty \bigcup_m Y$ while multiplicities in all multisets in Y are finite then $lub(\{f(H\sigma) \mid (X, f) \in Y\}) = \infty$, i.e. there is an infinite sequence of growing multiplicities of $H\sigma$ in $\{f(H\sigma) \mid (X, f) \in Y\}$, which means that there is an infinite sequence of growing finite multiplicities of $H\sigma$ in $\{scT_P(M) \mid M \in Y\}$ by the previous argument,i.e. $lub(\{f(H\sigma) \mid (X, f) \in \{scT_P(M) \mid M \in Y\}\}) = \infty$.

In summary, $scT_P(\bigcup_m Y) \subseteq_m \bigcup_m \{scT_P(M) \mid M \in Y\}$. □

Note that the proof of Lemma 126 does not use the assumptions that P is a range restricted and Datalog program and thus the scT_P operator is, in this sense, as powerful as the classical T_P operator while providing more information. In fact, for Datalog programs, multiplicities of atoms in $scT_P(S)$ are finite if multiplicities of atoms in S are finite. See Section 7.7, for more information.

Algorithm 7.7: Naive support counting forward chaining.

```
Name
  naive-scTP-fw-chaining(P)

Input
```

```
  P — a definite range restricted program

Output
  F — a multiset of ground atoms, $F = scT_P^\omega$

begin
  return naive-fw-chaining($scT_P$)
end
```

Proposition 127. *Let* P *be a definite range restricted program. Algorithm 7.7 terminates and computes the least fixpoint of* scT_P.

Proof. scT_P is monotonic because it is continuous (Lemma 47) which also means that $\mathrm{root}(scT_P(M)) \subseteq \mathrm{root}(scT_P(M'))$, for each $M \subseteq_m M'$. The root set is always a subset of HB_P which is finite (Observation 25) and the number of rules in P is finite by definition. Therefore Algorithm 7.7 terminates.

The rest of the claim follows directly from Proposition 105 by using Lemma 126 and the fact that $(\mathbb{P}(HB_P), \subseteq_m)$ is a complete lattice (Lemma 84). □

Definition 128 (Semi-naive scT_P mapping)**.** *Let* P *be a definite range restricted program. The semi-naive support counting immediate consequence mapping is defined as follows:*

$$\begin{aligned} scT_P\colon\ & \mathbb{P}(HB_P) \times \mathcal{P}(HB_P) \to \mathbb{P}(HB_P) \\ & scT_P(S,\Delta) = [\ \ H\sigma \in HB_P \mid \exists\, s = (r,\sigma), \\ & \qquad r = H \leftarrow B_1,\ldots,B_n \in P, \\ & \qquad (\exists 1 \leqslant j \leqslant n) B_j\sigma \in \Delta, \\ & \qquad \mathrm{body}(s) \subseteq \mathrm{root}(S), \mathrm{dom}(\sigma) = var(r)\ \ ], \end{aligned}$$

where $S \in \mathbb{P}(HB_P)$ *is a multiset and* $\Delta \subseteq HB_P$ *is a set of ground atoms.*

Lemma 129. *Let* P *be a definite range restricted program. Then* $T_P^i = \mathrm{root}(scT_P^i)$, *for all* $i \in \mathbb{N}_0$. *Also* $T_P^\omega = \mathrm{root}(scT_P^\omega)$

Proof. By induction on i.

$i = 0$: $T_P^0 = \emptyset$ and $scT_P^0 = \emptyset_m$ and $\mathrm{root}(\emptyset_m) = \emptyset$.

$i \rightsquigarrow i+1$: $\subseteq$: Let $a \in T_P^{i+1}$. Then there is a support s of a such that $\mathrm{body}(s) \subseteq T_P^i$, therefore also $\mathrm{body}(s) \subseteq \mathrm{root}(scT_P^i)$ by the induction hypothesis. Therefore $a \in_m scT_P^{i+1}$, i.e. $a \in \mathrm{root}(scT_P^{i+1})$. $\supseteq$: Let $a \in \mathrm{root}(scT_P^{i+1})$. Then there is a support s of a such that $\mathrm{body}(s) \subseteq \mathrm{root}(scT_P^i)$, i.e. $\mathrm{body}(s) \subseteq T_P^i$ by the induction hypothesis and thus $a \in T_P^{i+1}$.

All together $T_P^i = \mathrm{root}(scT_P^i)$, for all $i \in \mathbb{N}_0$.

$$\begin{aligned} T_P^\omega &= \bigcup\{T_P^\beta \mid \beta < \omega\} && \text{by definition, limit case} \\ &= \bigcup\{\mathrm{root}(scT_P^\beta) \mid \beta < \omega\} && \text{by the first part of the lemma} \\ &= \mathrm{root}(\textstyle\bigcup_m\{scT_P^\beta \mid \beta < \omega\}) && \text{by Lemma 79} \\ &= \mathrm{root}(scT_P^\omega) && \text{by definition, limit case} \end{aligned}$$

□

Lemma 130. $scT_P(scT_P^n, T_P^n \setminus T_P^{n-1}) = scT_P^{n+1} \ominus scT_P^n$.

Proof. Let us show $scT_P(scT_P^n, T_P^n \setminus T_P^{n-1}) \subseteq_m scT_P^{n+1} \ominus scT_P^n$.

Let $a \in_m scT_P(scT_P^n, T_P^n \setminus T_P^{n-1})$ and let m_a be its multiplicity. From the definition of $scT_P(F,\Delta)$, there are m_a supports of a with respect to P, let

s be one of them, such that $\mathrm{body}(s) \subseteq \mathrm{root}(scT_P^n)$. $scT_P^{n+1} = scT_P(scT_P^n)$, therefore all such supports s are generated in scT_P^{n+1} too and $a \in_m scT_P^{n+1}$ with multiplicity $\geqslant m_a$ ($\geqslant$ because there may be supports of a that do not satisfy the additional condition of the semi-naive mapping that there is a $b \in \mathrm{body}(s)$ such that $b \in T_P^n \setminus T_P^{n-1}$). There is a $b \in \mathrm{body}(s)$ such that $b \in T_P^n \setminus T_P^{n-1}$ by the definition of $scT_P(F, \Delta)$. Thus $b \notin T_P^{n-1}$ and therefore s is not generated in the computation of scT_P^n. Therefore, by the definition of scT_P, the multiplicity of a in scT_P^n is by m_a smaller than the multiplicity of a in scT_P^{n+1}. In summary $a \in_m scT_P^{n+1} \ominus scT_P^n$ with the multiplicity m_a, therefore $scT_P(scT_P^n, scT_P^n \setminus scT_P^{n-1}) \subseteq_m scT_P^{n+1} \ominus scT_P^n$.

Let us show $scT_P^{n+1} \ominus scT_P^n \subseteq_m scT_P(scT_P^n, T_P^n \setminus T_P^{n-1})$.

Let $a \in_m scT_P^{n+1} \ominus scT_P^n$ with a multiplicity m_a. Then there are m_a supports of a with respect to P, let s be one of them. Then $\mathrm{body}(s) \subseteq \mathrm{root}(scT_P^n)$ and $\mathrm{body}(s) \nsubseteq \mathrm{root}(scT_P^{n-1})$. Therefore there is a $b \in \mathrm{body}(s)$ such that $b \in \mathrm{root}(scT_P^n) \setminus \mathrm{root}(scT_P^{n-1})$, i.e. $b \in T_P^n \setminus T_P^{n+1}$ by Lemma 129. Therefore $a \in_m scT_P(scT_P^n, T_P^n \setminus T_P^{n-1})$ with the multiplicity m_a. In summary, $scT_P^{n+1} \ominus scT_P^n \subseteq_m scT_P(scT_P^n, T_P^n \setminus T_P^{n-1})$. □

Algorithm 7.8: Semi-naive support counting forward chaining.

```
Name
  semi-naive-scTP-fw-chaining(P)

Input
  P — a definite range restricted program

Output
  F — a multiset of ground atoms, $F = scT_P^\omega$

Variables
  $\Delta$ — a set of ground atoms

Initialization
  F := $\emptyset_m$; $\Delta$ := $T_P(\emptyset)$; $\Delta_F$ := $scT_P(\emptyset_m)$

begin
  while $\Delta \neq \emptyset$
  begin
    F := $F \uplus \Delta_F$
    $\Delta_F$ := $scT_P(F, \Delta)$        {new atoms}
    $\Delta$ := $\mathrm{root}(\Delta_F) \setminus \mathrm{root}(F)$ {forbid redundant and cyclic derivations
        }
  end

  return F
end
```

Proposition 131. *Let* P *be a definite range restricted program. Algorithm 7.8 terminates and computes the least fixpoint of* scT_P*.*

Proof. Analogical to the proof of Proposition 109 by using the fact that HB_P is finite in the Datalog case, Lemma 126, and Lemma 130. □

The following proposition shows that the support counting operator indeed counts the number of supports as one would expect.

Proposition 132. *Let* P *be a definite range restricted program. Then* $a \in^x_m scT_P^\omega$ *iff* $x = |\{s \in skT_P^\omega \mid \text{head}(s) = a\}|$.

Proof. $\Rightarrow$: Let $a \in^x_m scT_P^\omega$. Then there are x supports of a with respect to P such that for each such support s there is an $i_s \in \mathbb{N}_0$ such that $\text{body}(s) \subseteq \text{root}(scT_P^{i_s})$ by definition of ordinal power (the limit case), and by definition of scT_P. $\text{root}(scT_P^{i_s}) = T_P^{i_s}$ by Lemma 129, therefore $\text{body}(s) \subseteq T_P^{i_s}$ and thus $s \in skT_P^{i_s+1}$ by definition of skT_P and Lemma 117.

$\Leftarrow$: Similarly. □

COMPLEXITY. It is easy to see that Algorithm 7.8 has the same worst case time complexity as classical semi-naive forward chaining, i.e. $O(n^k)$.

7.6 DERIVATION COUNTING FORWARD CHAINING

An atom is derivable from a program if it has a derivation with respect to the program. An approach to reason maintenance, in the next section, is based on counting the number of derivations of atoms. This section defines an extended immediate consequence operator that also records all dependencies of each atom which are later used to count the number of all derivations of each atom. A dependency of an atom is informally a set of atoms that are in a derivation of that atom.

Let us assume that during a forward chaining iteration, a set of new dependencies D of an atom b_i is derived. Dependencies D correspond to new derivations of b_i and may contribute to new derivations of $\text{head}(s)$ for any s in the support graph such that $b_i \in \text{body}(s)$. A new derivation of $\text{head}(s)$ can result from a combination of a new derivation of b_i with existing derivations of $b_j \in \text{body}(s), j \neq i$. Not all combinations are admissible. No derivation of a $b_j \in \text{body}(s)$ that uses $\text{head}(s)$ can be used as a part of a derivation of $\text{head}(s)$.

Let us now define dependencies formally.

Definition 133 (Extended atom, dependency)**.** *Let* P *be a program. An* extended atom *is a pair* (a, D)*, where* $a \in HB$ *and* $D \subseteq HB$*. An* extended atom with respect to P *is a pair* (a, D)*, where* $a \in HB_P$ *and* $D \subseteq HB_P$*.* D *is called a* dependency *of* a*.*

Definition 134 (Derivation counting Herbrand base)**.** *Let* P *be a definite program. Then*

$dHB = HB \times \mathcal{P}(HB)$ *is the* derivation counting Herbrand base, *and*

$dHB_P = HB_P \times \mathcal{P}(HB_P)$ *is the* derivation counting Herbrand base with respect to P.

The derivation counting Herbrand base is non-empty for a non-empty program because at least one constant is assumed.

Observation 135. *If HB (resp. HB_P) is finite then dHB (resp. dHB_P) is.*

Lemma 136. *If* P *is a program with no function symbols other than constants then* dHB_P *is finite.*

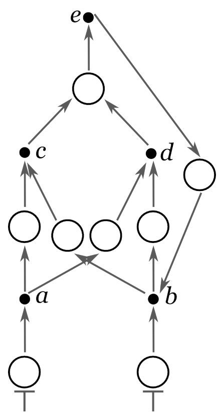
Proof. The lemma follows directly from Observation 135 and Observation 25. □

The derivation counting immediate consequence operator dcT_P is defined as follows:

Definition 137 (Derivation counting immediate consequence operator). *Let* P *be a definite range restricted program. The* derivation counting immediate consequence operator dcT_P *for* P *is the mapping:*

$$\begin{aligned} dcT_P\colon\ & \mathbb{P}(dHB_P) \rightarrow \mathbb{P}(dHB_P) \\ dcT_P(S) = [\ & (H\sigma, D) \in dHB_P \mid (\exists s = (r,\sigma)), \\ & dom(\sigma) = var(r), r = H \leftarrow B_1, \ldots, B_n \in P; \\ & \forall 1 \leqslant i \leqslant n\ \exists D_i\ (B_i\sigma, D_i) \in_m S; H\sigma \notin D_i, \\ & H\sigma \neq B_i\sigma, D = \textstyle\bigcup_{i=1}^{n} D_i \cup \{B_i\sigma\}\quad]. \end{aligned}$$

The dcT_P operator maps a multiset of extended atoms with respect to P to a multiset of extended atoms with respect to P. Repeatedly generated extended atoms are kept because the operator is defined using a multiset. Note that most cyclic derivations are avoided by the condition $H\sigma \notin D_i$ which does not cover one step cycles (cycles of length 2 in the respective support graph), hence the condition $H\sigma \neq B_i\sigma$.



A reproduction of Figure 7.6 for convenience of the reader. The compact support graph induced by the fixpoint of skT_P, where $P = \{a \leftarrow \top; b \leftarrow \top; c \leftarrow a; d \leftarrow a; c \leftarrow b; d \leftarrow b; e \leftarrow c, d; b \leftarrow e\}$.

Example 138. *Consider the example in Figure 7.6. Let us explore ordinal powers of* dcT_P*:*

$$\begin{aligned} dcT_P^0 &= \emptyset_m \\ dcT_P^1 &= [(a,\emptyset),(b,\emptyset)] \\ dcT_P^2 &= [(a,\emptyset),(b,\emptyset),(c,\{a\}),(c,\{b\}),(d,\{a\}),(d,\{b\})] \\ dcT_P^3 &= [(a,\emptyset),(b,\emptyset),(c,\{a\}),(c,\{b\}),(d,\{a\}),(d,\{b\}),(e,\{c,d,a\}), \\ &\quad (e,\{c,d,b\}),(e,\{c,d,a,b\}),(e,\{c,d,a,b\})] \\ dcT_P^4 &= [(a,\emptyset),(b,\emptyset),(c,\{a\}),(c,\{b\}),(d,\{a\}),(d,\{b\}),(e,\{c,d,a\}), \\ &\quad (e,\{c,d,b\}),(e,\{c,d,a,b\}),(e,\{c,d,a,b\}),(b,\{e,c,d,a\})] \end{aligned}$$

It is easy to verify that $dcT_P^i = dcT_P^4$ *for all* $i \geqslant 4$ *in this example.*

Lemma 139. dcT_P *is continuous.*

Proof. It is easy to see that dcT_P is monotonic: let $M, M' \in \mathbb{P}(dHB_P)$ and $M \subseteq_m M'$ then $dcT_P(M) \subseteq_m dcT_P(M')$ because if $(B_i\sigma, D_i) \in_m^x M$ then also $(B_i\sigma, D_i) \in_m^y M'$ and $x \leqslant y$ by the definition of the submultiset relation.

We have to show that $dcT_P(\bigcup_m Y) = \bigcup_m \{dcT_P(M) \mid M \in Y\}$ for each directed $Y \subseteq \mathbb{P}(dHB_P)$ by Definition 46 (continuous operator) and Observation 85. Let Y be a directed subset of $\mathbb{P}(dHB_P)$.

For each $M \in Y$: $dcT_P(M) \subseteq_m dcT_P(\bigcup_m Y)$, by monotonicity of dcT_P. Thus $\bigcup_m \{dcT_P(M) \mid M \in Y\} \subseteq_m dcT_P(\bigcup_m Y)$.

Let us show that $dcT_P(\bigcup_m Y) \subseteq_m \bigcup_m \{dcT_P(M) \mid M \in Y\}$.

Let $(H\sigma, D) \in_m^\mu dcT_P(\bigcup_m Y)$.

Case $\mu \in \mathbb{N}$. Let $s = (H \leftarrow B_1, \dots, B_n, \sigma)$ and D_i, $1 \leqslant i \leqslant n$, be any of the μ combinations that satisfy the multiset-builder condition (see Definition 137). For each $e_i = (B_i\sigma, D_i) \in_m \bigcup_m Y$ let $Y_i \in Y$ be a set with $e_i \in_m Y_i \subseteq_m \bigcup_m Y$. Such a set exists by definition of the multiset union. The set $\{Y_1, \dots, Y_n\}$ is a finite subset of the directed set Y. Therefore there is an $M \in Y$ such that $\bigcup_m \{Y_1, \dots, Y_n\} \subseteq_m M$. Therefore $(H\sigma, D) = (H\sigma, \bigcup_{i=1}^n D_i \cup \{B_i\sigma\}) \in_m \bigcup_m \{dcT_P(M) \mid M \in Y\}$ and its multiplicity is at least μ by repeating the argument μ times.

Case $\mu = \infty$. If the multiplicity of e_i is infinite in a multiset in Y then $(H\sigma, D)$ is generated infinitely many times in $\{dcT_P(M) \mid M \in Y\}$ by the definition of dcT_P, and thus $(H\sigma, D) \in_m^\infty \{dcT_P(M) \mid M \in Y\}$. If, on the other hand, $e_i \in_m^\infty \bigcup_m Y$ while multiplicities in all multisets in Y are finite then $\text{lub}(\{f(e_i) \mid (X, f) \in Y\}) = \infty$, i.e. there is an infinite sequence of growing multiplicities of e_i in $\{f(e_i) \mid (X, f) \in Y\}$, which means that there is an infinite sequence of growing finite multiplicities of $(H\sigma, D)$ in $\{dcT_P(M) \mid M \in Y\}$ by the previous argument, i.e. $m = \text{lub}(\{f(H\sigma) \mid (X, f) \in \{dcT_P(M) \mid M \in Y\}\}) = \infty$.

In summary, $dcT_P(\bigcup_m Y) \subseteq_m \bigcup_m \{dcT_P(M) \mid M \in Y\}$. □

The naive version of forward chaining with dcT_P is directly analogous to the classical naive forward chaining, see Algorithm 7.9.

Algorithm 7.9: Naive forward chaining with the dcT_P operator.

```
Name
  naive-dcTP-fw-chaining(P)

Input
  P — a definite range restricted program

Output
  F — a multiset of extended atoms, $F = dcT_P^\omega$

begin
  return naive-fw-chaining($dcT_P$)
end
```

Lemma 140. *Let* $G = (S, F, E, l)$ *be a support graph and* $s \in S$ *a support. Let* $W = (S_W, F_W, E_W, l_W)$ *be a well-founded subgraph of* G *that includes* s*. Then there is a derivation of* $\text{head}(s)$ *in* W *that includes* s*.*

Proof. We will show how to find a minimal well-founded subgraph of W that includes s and has a node labelled $\text{head}(s)$ as its only sink node.

W may not be compact – there may be several nodes labelled $\mathrm{head}(s)$, let z be one such node (note, that a derivation may be non-compact, see Definition 98 and Example 100). If z is not a sink node of W or if it is not the only sink node of W then it is easy to see that all nodes that z does not depend on make W not minimal. Let $D = \{a \in F_W \mid z \text{ does not depend on } a \text{ in } W\}$, $e = \{t \in S_W \mid z \text{ does not depend on } t \text{ in } W\}$. It is easy to verify that $W' = (W - E) - D$ is a connected well-founded subgraph of W that includes s and has $\mathrm{head}(s)$ as its only sink node.

The only admissible way (see Definition 86) how W' may be not minimal is that some of its atom nodes have more than one support.

Let A be the set of nodes in W' with more than one support: $A = \{a \mid a \in F_{W'}, |\{t \in S_{W'} \mid (t,a) \in E_{W'}\}| > 1\}$. This is a redundancy which can be reduced by removing all but one support of each such a. Let $f : A \to S_{W'}$ be a function from atom nodes to supports which for each atom node a selects one support out of all its supports $\{t \in S_{W'} \mid (t,a) \in E_{W'}\}$ and such that $f(l_{W'}^{-1}(\mathrm{head}(s))) = s$. Such a function exists because finite support graphs are assumed. Then $W'' = W' - (S_{W'} \cap f(F_{W'}))$ is a subgraph of W' it is well-founded because W' is well-founded and each atom node has exactly one support selected by f. W'' however may not be minimal w.r.t the subgraph relationship: removing a support s may leave new sink nodes labelled $\mathrm{body}(s)$. z does not depend on the new sink nodes in W'' and thus the previous approach of removing superfluous sink nodes can be applied. Let W''' be the support graph resulting from recursively removing all such sink nodes (except for z) and their supports. Then W''' is a minimal well-founded subgraph that includes s and has z as its only sink node, i.e. W''' is a derivation of $\mathrm{head}(s)$. □

Proposition 141. *Let* P *be a definite range restricted program. Algorithm 7.9 terminates and computes the least fixpoint of* dcT_P*.*

Proof. dcT_P is monotonic because it is continuous (Lemma 47). In the range restricted Datalog case, there is only a finite number of extended atoms (Lemma 136) and cycles are prevented by the conditions $H\sigma \notin D_i$ and $H\sigma \neq B_i\sigma$. Therefore Algorithm 7.9 terminates.

The rest of the claim follows directly from Proposition 105 by using Lemma 139 and the facts that dHB_P is non-empty and $(\mathbb{P}(dHB_P), \subseteq_m)$ is a complete lattice (Lemma 84). □

The following proposition shows that the dcT_P operator counts correctly in the sense that the multiplicity of extended atoms corresponds to the number of different derivations of the atom that use atoms only in the atom's dependency.

Proposition 142. *Let* P *be a definite range restricted Datalog program. Then* $(a, D) \in_m^x dcT_P^\omega$ *iff there are* x *derivations of* a *with respect to* P *that use all atoms in* D *and only atoms in* D*.*

Proof. We will prove by induction on n that $(a, D) \in_m^x dcT_P^n$ iff there are x derivations of depth (Definition 99) $\leqslant n$ of a with respect to P that use all atoms in D and only atoms in D.

$n = 1$:

$\Rightarrow$: Let $(a, \emptyset) \in_m dcT_P^1$. Any support generated in dcT_P^1 has an empty body because $dcT_P^0 = \emptyset_m$. Therefore there is exactly one ground support s of a with respect to P, by definition of dcT_P. $SG(s)$ is a derivation of depth 1 of a with respect to P.

$\Leftarrow$: If G is a derivation of depth 1 of a with respect to P then it contains only one support s of a which thus must have an empty body, from the definition of a support graph. Therefore $(a, \emptyset) \in_m dcT_P^1$. There is only one support of a with an empty body therefore there is only one derivation of a of depth 1 and the multiplicity of $(a, \emptyset)$ in dcT_P^1 is 1.

$n \rightsquigarrow n+1$:

$\Rightarrow$: Let $(a, D) \in_m^x dcT_P^{n+1}$. Then there are x ways to satisfy the multiset-builder condition for generating (a, D) in dcT_P^{n+1}. Let us explore one such combination. Let s be a support of a generated in dcT_P^{n+1} and for each $b_k \in body(s)$ there is a D_k such that $(b_k, D_k) \in_m dcT_P^n$ and $D = \bigcup_{k=1}^{|body(s)|} D_k \cup \{b_k\}$. By induction hypothesis, for each b_k, there is a derivation of length n that uses all and only atoms in D_k. Because $a \notin D_k$ for all k, derivations of body atoms $body(s)$ can be combined into a derivation of a of length $n+1$ by adding s.

Each such combination of derivations of body atoms and a support of a results in a derivation of a. Different supports of a result in different derivations of a (they differ at least in the support). By induction hypothesis, for each b_k all derivations of length n using all and only atoms in D_k are generated in dcT_P^n and the next application of the dcT_P operator generates all admissible combinations that result in a derivation of (a, D): the multiset-builder condition of dcT_P avoids only cycles and a graph with a cycle is not minimal and thus it is not a derivation. Therefore the multiplicity is correct.

$\Leftarrow$: Let G be a derivation of depth $n+1$ of a with respect to P that uses all and only atoms in D. There is exactly one support s of a in G and therefore any derivation in $G - s$ has length at most n. $G - s$ is well-founded (because G is) and it contains all atoms in $body(s)$ as well as a support for each of them. Therefore, by Lemma 140, for each atom $b_k \in body(s)$ there is a derivation in $G - s$ of b_k that uses (all and only) atoms D_k. By induction hypothesis, $(b_k, D_k) \in_m dcT_P^n$. Then, by the definition of dcT_P, $(a, D) \in_m dcT_P^{n+1}$, where $D = \bigcup_{k=1}^{|body(s)|} D_k \cup \{b_k\}$ is the set of atoms in G.

By induction hypothesis, the multiplicity of each (b_k, D_k) in dcT_P^n is correct. Any derivation of a that uses all and only atoms in D combines derivations of such b_k with a support of a and all and only such supports are generated by dcT_P, by its definition. Thus the multiplicity of (a, D) in dcT_P^{n+1} is correct. □

Proposition 142 leads to the following corollary which justifies the operators name.

Corollary 143. *Let P be a definite range restricted Datalog program. Then the number of derivations of a with respect to P is*

$$deriv_P\#(a) = \sum_{\exists D (a,D) \in_m^x dcT_P^\omega} x$$

Proof. Follows directly from Proposition 142. □

Algorithm 7.9 recomputes all the previously computed extended atoms in each step. Again, it is possible to define a semi-naive version of the dcT_P operator which is then used to specify a semi-naive dcT_P forward chaining procedure.

Definition 144 (Semi-naive dcT_P mapping)**.** *Let P be a definite range restricted program. The semi-naive derivation counting immediate consequence mapping is the mapping:*

$$\begin{array}{lll} dcT_P\text{:} & \mathbb{P}(dHB_P) \times \mathbb{P}(dHB_P) \to \mathbb{P}(dHB_P) & \\ & dcT_P(S, \Delta) = [& (H\sigma, D) \in dHB_P \mid \exists\, s = (r, \sigma), \\ & & dom(\sigma) = var(r), r = H \leftarrow B_1, \ldots, B_n \in P; \\ & & \forall 1 \leqslant i \leqslant n\ \exists D_i\ (B_i\sigma, D_i) \in_m S; H\sigma \notin D_i, \\ & & \exists 1 \leqslant j \leqslant n (B_j\sigma, D_j) \in_m \Delta, \\ & & H\sigma \neq B_i\sigma, D = \bigcup_{i=1}^{n} D_i \cup \{B_i\sigma\} \quad], \end{array}$$

where $S \in \mathbb{P}(dHB_P)$ *and* $\Delta \in \mathbb{P}(dHB_P)$ *are multisets of extended atoms.*

Lemma 145. $dcT_P(dcT_P^n, dcT_P^n \ominus dcT_P^{n-1}) = dcT_P^{n+1} \ominus dcT_P^n$

Proof. Let us show $dcT_P(dcT_P^n, dcT_P^n \ominus dcT_P^{n-1}) \subseteq_m dcT_P^{n+1} \ominus dcT_P^n$. Let $a \in_m dcT_P(dcT_P^n, dcT_P^n \ominus dcT_P^{n-1})$ and let m_a be its multiplicity. Then, the multiset-builder condition of the semi-naive dcT_P mapping is satisfied m_a times. Let $s = (H \leftarrow B_1, \ldots, B_l, \sigma)$ and D_i, $1 \leqslant i \leqslant l$, be any combination that satisfies the condition. Then they also satisfy the condition of dcT_P^{n+1} because it is less restrictive; i.e. $a \in_m dcT_P^{n+1}$ with a multiplicity at least m_a. There are at least m_a satisfying combinations that are not satisfied in dcT_P^n because in each such combination there is a $1 \leqslant j \leqslant l$ such that $(B_j\sigma, D_j) \in_m dcT_P^n \ominus dcT_P^{n-1}$. The rest follows from monotonicity of the dcT_P operator. In summary, $dcT_P(dcT_P^n, dcT_P^n \ominus dcT_P^{n-1}) \subseteq_m dcT_P^{n+1} \ominus dcT_P^n$.

Let us show $dcT_P^{n+1} \ominus dcT_P^n \subseteq_m dcT_P(dcT_P^n, dcT_P^n \ominus dcT_P^{n-1})$. Let $a \in_m dcT_P^{n+1} \ominus dcT_P^n$ and let m_a be its multiplicity. Then the multiset generating condition of dcT_P^{n+1} is satisfied m_a more times than in dcT_P^n. dcT_P^{n+1} is generated based on dcT_P^n and dcT_P^n is generated based on dcT_P^{n-1}. Therefore there are m_a occurrences of extended atoms in $dcT_P^n \ominus dcT_P^{n-1}$ that generate a. Therefore the multiplicity of a in $dcT_P(dcT_P^n, dcT_P^n \ominus dcT_P^{n-1})$ is at least m_a. In summary, $dcT_P^{n+1} \ominus dcT_P^n \subseteq_m dcT_P(dcT_P^n, dcT_P^n \ominus dcT_P^{n-1})$. □

Algorithm 7.10: Semi-naive forward chaining with the dcT_P operator.

```
Name
  semi-naive-dcTP-fw-chaining(P)

Input
  P — a definite range restricted program

Output
  F — a multiset of extended atoms, $F = dcT_P^\omega$

Variables
  Δ — a multiset of extended atoms

Initialization
  F := $\emptyset_m$; Δ := $dcT_P(\emptyset_m)$

begin
  while $\Delta \neq \emptyset_m$
  begin
    F := $F \uplus \Delta$
    Δ := $dcT_P(F, \Delta)$ {cycles are handled by the $dcT_P$ mapping itself
        }
    end

    return F
end
```

Proposition 146. *Let* P *be a definite range restricted program. Algorithm 7.10 terminates and computes the least fixpoint of* dcT_P.

Proof. Analogical to the proof of Proposition 109 by using the fact that dHB_P is finite in the Datalog case (Lemma 136), continuity of dcT_P (Lemma 139), and Lemma 145. □

If non-empty, the Δ in semi-naive T_P forward chaining always contains new atoms – atoms that are not in the input set to which the semi-naive operator is applied. Δ in Algorithm 7.10 does not satisfy the same property with respect to extended atoms as Example 147 shows.

Example 147. *Let* P *be the following program:*

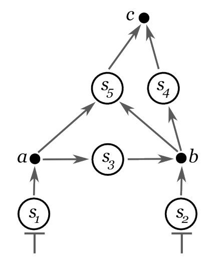
$$r_1 = a \leftarrow \top; \qquad r_2 = b \leftarrow \top$$
$$r_3 = b \leftarrow a$$
$$r_4 = c \leftarrow b$$
$$r_5 = c \leftarrow a, b$$

See Figure 7.7 for the support graph corresponding to the fixpoint of P.
Let us explore ordinal powers of dcT_P*:*

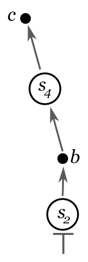
$$\begin{aligned}
\mathrm{dcT}_P^0 &= \emptyset_m \\
\mathrm{dcT}_P^1 &= [(a,\emptyset),(b,\emptyset)] \\
\mathrm{dcT}_P^2 &= [(a,\emptyset),(b,\emptyset),(b,\{a\}),(c,\{b\}),(c,\{a,b\})] \\
\mathrm{dcT}_P^3 &= [(a,\emptyset),(b,\emptyset),(b,\{a\}),(c,\{b\}),(c,\{a,b\}),(c,\{a,b\}),(c,\{a,b\})]
\end{aligned}$$

As you can see, $(c,\{a,b\}) \in_m^1 \mathrm{dcT}_P^2$ *and* $(c,\{a,b\}) \in_m^3 \mathrm{dcT}_P^3$ *– in total, there are four derivations of* c*. Three of the derivations use the same set of atoms* $\{a,b\}$ *and are of different lengths, compare (c) to (d) and (e) in Figure 7.7. The extended atom* $(c,\{a,b\})$ *is thus in* Δ *in two (subsequent) iterations of the while cycle of Algorithm 7.10. Such repeated derivations of the same extended atom are propagated further in order to derive all derivations of all atoms. Cyclic derivations are prevented by the definition of the* dcT_P *operator and* dcT_P *semi-naive mapping themselves.*
It is easy to verify that $\mathrm{dcT}_P^i = \mathrm{dcT}_P^3$ *for all* $i \geqslant 3$ *in this example.*

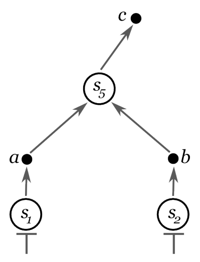
COMPLEXITY. Algorithm 7.10 is less efficient than both semi-naive T_P and semi-naive scT_P forward chaining because computation cannot stop when an atom is derived again that has already been derived. A new derivation of the atom may mean new derivations of its consequences. The algorithm derives each atom as many times as there are derivations of the atom with respect to P as the preceding example also shows. In the theoretical worst case, any atom is derivable from any atom. In this case the $O(n^k)$ computation has to be repeated $O(n^k)$ times, thus the upper bound on the worst case time complexity is $O(n^{2k})$, i.e. not that significantly worse than for computation of the classical fixpoint.



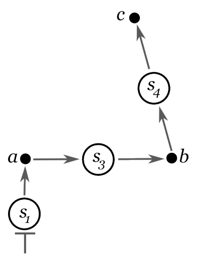
(a) The compact support graph induced by skT_P^{ω}.



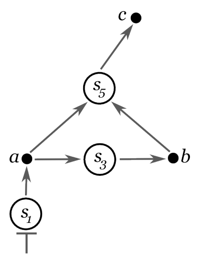
(b) The derivation of $(c, \{b\})$ in dcT_P^2.



(c) The derivation of $(c, \{a, b\})$ in dcT_P^2.



(d) A derivation of $(c, \{a, b\})$ in dcT_P^3.



(e) A derivation of $(c, \{a, b\})$ in dcT_P^3.

Figure 7.7: Support graphs illustrating Example 147. Note that each of (b), (c), (d), (e) is a subgraph of (a) and thus homomorphically embedded in (a).

7.7 MULTIPLICITIES AND MULTISET OPERATORS

A multiset can be infinite in two ways: 1) its root set may be infinite and 2) it may contain an element with infinite multiplicity and these two options do not exclude each other. This section shows that the multiset operators are well-behaved in the sense that they do not create infinite multiplicities under the assumption of Datalog programs.

The following lemma shows that infinite multiplicities can be reached only in ω steps, not sooner, beginning from a multiset with finite multiplicities.

Lemma 148. *Let* P *be a definite Datalog program. Let* $X \in \mathbb{P}(HB_P)$ *be a multiset with finite multiplicities. Then* $\mathrm{scT}_P(X)$ *is a multiset with finite multiplicities.*

Proof. There is a finite number of rules in P, a finite number of ground substitutions for each of the rules because HB_P is finite (Observation 25),

$\mathrm{root}(X)$ is finite for the same reason (Observation 25), and all multiplicities are finite in X. Therefore there is only a finite number of combinations by which an atom $H\sigma$ can be generated in $scT_P(X)$. Therefore multiplicities of all extended atoms in $dcT_P(X)$ are finite. □

Lemma 149. *Let* P *be a definite Datalog program. Let* $X \in \mathbb{P}(dHB_P)$ *be a multiset with finite multiplicities. Then* $dcT_P(X)$ *is a multiset with finite multiplicities.*

Proof. There is a finite number of rules in P, a finite number of ground substitutions for each of the rules because HB_P is finite (Observation 25), dHB_P is finite (Lemma 136), and for each $(B_i\sigma, D_i) \in_m^y X$ it holds that $y < \infty$ because all multiplicities are finite in X. Therefore there is only a finite number of combinations by which an extended atom $(H\sigma, D)$ can be generated in $dcT_P(X)$. Therefore multiplicities of all extended atoms in $dcT_P(X)$ are finite. □

It has been shown in previous sections that the least fixpoint of both scT_P and dcT_P operators is reached in less than ω steps if P is a Datalog program, hence the following corollary.

Proposition 150. *Let* P *be a definite Datalog program. Then* scT_P^ω *and* dcT_P^ω *are multisets with finite multiplicities.*

Proof. Follows directly from Lemma 148, Proposition 127, and Theorem 54 and from Lemma 149, Proposition 141, and Theorem 54. □

7.8 WELL-FOUNDEDNESS AND DERIVABILITY

One way of demonstrating that there is a derivation of an atom in T_P^ω is to show that it has a support in skT_P^ω which is well-founded with respect to the support graph induced by the fixpoint as the following two lemmas show.

Algorithms and lemmas about well-foundedness of supports described in this section were likely common knowledge in the now non-existent reason maintenance community. They are described in the framework of this dissertation and presented for the sake of completeness here.

Lemma 151. *Let* P *be a definite range restricted program and let* $s \in skT_P^\omega$*. If* s *is not well-founded in* $SG(skT_P^\omega)$ *then there is no derivation of* $\mathrm{head}(s)$ *in* $SG(skT_P^\omega)$ *that includes* s.

Proof. The statement is the contrapositive of the definition of a well-founded support, Definition 98. □

Lemma 152. *Let* P *be a definite range restricted program.* $a \in T_P^\omega$ *iff* a *has a support with respect to* P *that is well-founded in* $SG(skT_P^\omega)$.

Proof. $\Rightarrow$: Let $a \in T_P^\omega$ be an atom that has no support well-founded in $SG(skT_P^\omega)$. Then there is no derivation of a in $SG(skT_P^\omega)$ by Lemma 151. On the other hand, $T_P^\omega = \mathrm{heads}(skT_P^\omega)$ by Lemma 117 and the fact that the fixpoint is reached in a finite number of steps (datalog and range restrictedness is assumed). Thus there is a support of a in skT_P^ω and therefore there is a derivation of a in $SG(skT_P^\omega)$ – a contradiction.

$\Leftarrow$: If a has a support well-founded in $SG(skT_P^\omega)$ then $a \in skT_P^\omega$ and by Lemma 117 $a \in T_P^\omega$. □

A support s is well-founded in a support graph G if there is a derivation of head(s) in G which includes s, see Definition 98. Two algorithms are presented in this section which determine well-foundedness of a support in a support graph.

7.8.1 *Algorithms to decide well-foundedness*

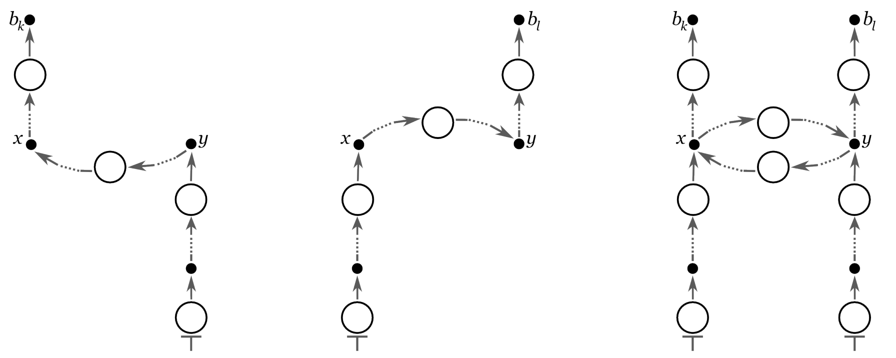
Let $G = (S, F, E, l)$ be a support graph. An obvious way to decide well-foundedness of supports in S is to try to construct a derivation backwards from s to base facts in G; if there is a derivation that includes s then it can be constructed this way. This method amounts to a depth-first search with avoiding cyclic paths, see Algorithm 7.11.

Algorithm 7.11: Deciding well-foundedness naively

```
Name
  is-well-founded(s, G, Forbidden)

Input
  $G = (F, S, E, l)$ — a finite compact support graph
  s — a support, $s \in S$
  Forbidden — a set of atoms

Output
  true — if s is well-founded
  false — if s is not well-founded

Variables
  t — a support
  f — an atom

begin
  if $head(s) \in body(s)$ or $body(s) \cap Forbidden \neq \emptyset$
    then return false

  for each $f \in body(s)$
    if not exists $t \in S$ such that
           $head(t) = f$ and
           is-well-founded($f, G, Forbidden \cup \{head(s)\}$)
    then return false

  return true
end
```

Lemma 153. *Let G be a compact support graph and s one of its supports. s is well-founded in G iff there is a derivation for each atom in* body(s) *that does not include* head(s).

Proof. ⇒: Let s be well-founded in G. Then there is a derivation D of head(s) in G that includes s. If $D - s$ includes head(s) then there is a cycle in D and thus D is not a derivation, a contradiction. Each atom in body(s) has a support in $D - s$ which is a well-founded graph and therefore the supports are well-founded too, by Lemma 140 and Definition 98.



(a) Derivation D_k.

(b) Derivation D_l.

(c) The cyclic support graph resulting from merging D_k and D_l.

Figure 7.8: Support graphs illustrating derivation merging in the proof of Lemma 153.

$\Leftarrow$: Let s be a support in G such that there is a derivation D_i for each atom $b_i \in \text{body}(s)$ that does not include $\text{head}(s)$. By Definition 98, we have to show that there is a derivation of $\text{head}(s)$ that includes s. By Lemma 140, it is sufficient to show that there is a well-founded subgraph of G that includes s.

Each D_i is a minimal well-founded subgraph of G that does not include $\text{head}(s)$. Let $D_i = (S_i, F_i, E_i, l_i)$. Then

$$D = (S, F, E, l) = \Big(\bigcup_{i=1}^{|\text{body}(s)|} S_i, \bigcup_{i=1}^{|\text{body}(s)|} F_i, \bigcup_{i=1}^{|\text{body}(s)|} E_i, \bigcup_{i=1}^{|\text{body}(s)|} l_i\Big)$$

is a subgraph of G. It is clear that for each $f \in F$ there is at least one incoming edge in E because each D_i is a subgraph of D. However, D may be cyclic, see Figure 7.8. For any cycle in D there are two derivations D_k, D_l ($0 \leqslant l, k \leqslant |\text{body}(s)|$) and two nodes $a, b \in F_k \cap F_l$ such that (WLOG) a depends on b in D_k and b depends on a in D_l. Each such cycle can be broken by removing either the path from a to b or the path from b to a. Note that both a and b have initially at least two supports and at least one support after breaking the cycle. Let D' be a support graph after subsequently breaking each cycle in D. Then D' is acyclic and for each atom node in D' there is still at least one incoming edge in D'. $\text{head}(s)$ is not in any D_i and therefore it is not in D' either. Thus $D' + s$ is a well-founded subgraph of G that includes s. □

Lemma 154. *Algorithm 7.11 called with* Forbidden $= \emptyset$ *terminates and outputs* true *if* s *is well-founded in* G *and* false *otherwise.*

Proof. The algorithm terminates because any support with a body atom in Forbidden is skipped (i.e. cycles are excluded), Forbidden increases in each recursive call, and G is finite.

The algorithm looks for well-founded supports of each atom in $\text{body}(s)$ such that do not depend strongly on $\text{head}(s)$ in G. If such supports are found then s is well-founded in G by Lemma 153. If such supports are not found then s is not well-founded in G also by Lemma 153. The algorithm excludes derivations of atoms in $\text{body}(s)$ that include $\text{head}(s)$ by means of the set $\mathtt{Forbidden}$ and the base case that stops recursion is correct: if $\mathrm{body}(s) = \emptyset$ then return true, see the proof of Lemma 153, $\Rightarrow, n = 1$. $\square$

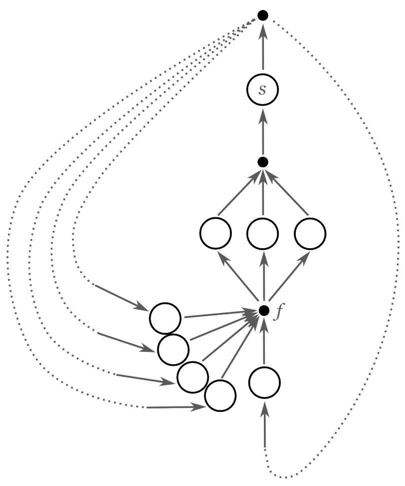


Figure 7.9: A support graph in which deciding that s is not well-founded using Algorithm 7.11 requires visiting all supports of f and their ancestors three times.

Algorithm 7.11 is inefficient because a support may be visited repeatedly, see Figure 7.9 for an illustration of the problem. The core of the inefficiency is that a derivation is constructed branch by branch and each time a new branch is explored information is not available about previous branches. In the worst case, any atom is derivable from any other atom. This means that from each of the $O(n^k)$ atoms each of the $O(n^k)$ atoms can be visited, giving a minimum worst case time complexity $O(n^{2k})$, where n is the number of terms occurring in base facts and k is the maximum over all rules in P of the number of variables in that rule (see discussion of complexity of classical forward chaining in Section 7.2). This estimate however does not account for traversing edges. To estimate the number of edges in a support graph, let us assume that b is the maximum over all rules in P of the number of body atoms in that rule. Any support has at most b body atoms and thus the corresponding support node has at most $b+1$ edges (to body atoms and to the head of the support). Therefore the number of edges in a support graph can be estimated as $O((b+1) * n^k)$. Thus given an atom, a visit of another arbitrary atom involves traversing at most $O((b+1) * n^k)$ edges. The worst case time complexity of Algorithm 7.11 is thus $O(n^{2k} * (b+1) * n^k) = O((b+1)n^{3k})$. In summary, well-foundedness of each support in a support graph can be decided in time $O((b+1)n^{4k})$ by running this algorithm for each support in a support graph.

A more efficient algorithm constructs a derivation that shows well-foundedness of s more directly. A derivation is a minimal well-founded graph. To show well-foundedness of a support, it suffices to find a well-founded graph that includes s (Lemma 140).

Well-founded graph is an acyclic subgraph of a support graph in which all source nodes are base supports. An acyclic graph that has a support s as its sink can be found by a modified topological ordering of ancestors of s. The topological ordering algorithm is a standard graph algorithm (see for example [59]) that can be used to detect cycles in a directed graph. It produces a linear ordering of nodes such that each node precedes nodes to which it has outbound edges. This ordering can be used to guide the search for a well-founded graph. Let us first specify the topological ordering algorithm adapted to support graphs, see Algorithm 7.12.

Algorithm 7.12: Topological ordering of ancestors of a support in a support graph

```
Name
  topological-sort(s, G)

Input
  $G = (F, S, E, l)$ — a finite compact support graph
  s — a support in S

Output
  L — a list with topologically sorted nodes and atoms of G,
      global variable, initially empty

Variables
  t — a support
  i — a number, a global variable, initially 0
  marked(x) — a boolean, initially false for all $x \in F \cup S$, global

begin
  marked(head(s)) := true
  to-be-sorted := $\text{body}(s) \setminus L$

  if $\exists f \in$ to-be-sorted such that marked(f) then
    return L { a cycle detected, disregard s }

  for each f ∈to-be-sorted
  begin
    for each $t \in S$ such that $\text{head}(t) = f$
      topological-sort(t, G)

    L[i] := f
    i := $i + 1$
  end

  L[i] := s
  i := $i + 1$

  return L
end
```

Algorithm 7.12 is a depth-first search beginning from a given support. The latest recursive incarnation of the function stops either when a cycle is detected or when a base support (a support with an empty body) is encountered. It numbers nodes in the support graph in postorder manner and never processes a numbered support of an atom again. The algorithm therefore terminates because the input support graph is finite.

COMPLEXITY. The worst-case time complexity of topological ordering based on depth-first search is the same as that of depth-first search, i.e. linear in the size of the input graph.

Algorithm 7.13 uses the topological ordering determined by Algorithm 7.12 to guide its search for a well-founded subgraph including a support s.

Algorithm 7.13: Deciding well-foundedness using topological ordering

```
Name
  is-well-founded-topological(s, G)

Input
  $G = (F, S, E, l)$ — a finite support graph
  s — a support in S

Output
  true — if s is well-founded in G
  false — if s is not well-founded in G

Variables
  t — a support
  L — a list of nodes in G
  well-founded(x) — a boolean for $x \in S$, initially false for
      each $x \in S$

Initialization
  L := topological-sort(s, G)

begin
  for i from 0 to $|L|$ such that $L[i] = t$ and $t \in S$
  begin
    if $\forall b \in \mathrm{body}(t) \exists w \in S$ such that $L[j] = w, \mathrm{head}(w) = b, j < i$ and
        well-founded(w)
      then well-founded(t) := true
  end

  return well-founded(s)
end
```

Algorithm 7.13 uses Algorithm 7.12 to find a candidate C for a well-founded subgraph of G that by Lemma 140 contains a derivation of s if there is one. The algorithm then verifies whether C is indeed a well-founded graph. C is acyclic because that is how it was constructed. Therefore Algorithm 7.13 verifies only whether each node in F has an incoming edge – meaning that atoms and supports of C are properly connected and source nodes of the graph are base supports. The algorithm moreover marks each well-founded support in C as such.

Lemma 155. *Algorithm 7.13 terminates and outputs* true *if* s *is well-founded in* G *and* false *otherwise.*

Proof. The algorithm terminates because G is finite, Algorithm 7.12 terminates, and it visits each sorted support exactly once.

The call to Algorithm 7.12 (L := topological-sort(s, G)) finds a maximal acyclic subgraph W of G with s as its only sink node. If there is a derivation of $\text{head}(s)$ in G that includes s then it is a subgraph of W because W contains all nodes that s depends on except for those that $\text{head}(s)$ also depends on (these are the "causes" of the possibly present cycles). Then the algorithm examines each support in W in the topological order. The topological order ensures that supports are examined in the right order; i.e. first supports in W are examined that do not depend on any other node in W and then supports are examined that depend on already examined supports. Therefore the algorithm verifies the assumption of Lemma 153 for each support in W and thus it outputs true iff s is well-founded in G. □

COMPLEXITY. Algorithm 7.13 first sorts ancestors of a support s topologically and then accesses each of its ancestor supports exactly once and for each support it performs a number of tests bounded by the maximal number of body atoms over all supports in the support graph (assuming constant time access to neighbouring nodes). Therefore Algorithm 7.13 runs in linear time in the size of the input support graph which is $O((b+2)n^k)$ because the number of nodes is $O(n^k)$ and the number of edges is $O((b+1)*n^k)$, where b is the maximum over all rules in P of the number of body atoms in that rule (see the discussion of Algorithm 7.11 after Lemma 154). Therefore well-foundedness of each support in a support graph can be decided in $O(n^k * (b+2)n^k) = O((b+2)n^{2k})$ time in the worst case by running Algorithm 7.13 for each support node of the graph.

7.9 HERBRAND INTERPRETATIONS AND EXTENDED OPERATORS

Let P be a definite program. The least fixpoint of T_P induces the unique minimal Herbrand model of P by Theorem 62. A similar result can be obtained for the skT_P, scT_P, and dcT_P operators by defining Herbrand interpretation represented by a set of ground supports, a multiset of ground atoms, and by a multiset of ground extended atoms respectively.

Definition 156 (Herbrand interpretation represented by a multiset of ground atoms)**.** *Let* $A \in \mathbb{P}(HB)$ *be a multiset of ground atoms. Then* $HI(A) = HI(\text{root}(A)$ *), i.e. the Herbrand interpretation represented by* A *is defined as the Herbrand interpretation represented by the set of ground atoms* $\text{root}(A)$.

Definition 157. *Let* P *be a definite program,* $e = (a, D)$ *be an extended atom with respect to* P*, and let* A *be a multiset of extended atoms with respect to* P*. Then* $\text{atom}(e) = a$ *and* $\text{atoms}(A) = \{\text{atom}(x) \mid x \in_m A\}$.

Definition 158 (Herbrand interpretation represented by a multiset of ground extended atoms)**.** *Let* $A \in \mathbb{P}(dHB)$ *be a multiset set of ground extended atoms. Then* $HI(A) = HI(\text{atoms}(A))$*, i.e. the Herbrand interpretation represented by* A *is defined as the Herbrand interpretation represented by the set of ground atoms* $\text{atoms}(A)$.

Lemma 159. *Let* $S \subseteq \mathbb{P}(dHB)$*. Then* $\text{atoms}(\bigcup_m S) = \bigcup\{\text{atoms}(s) \mid s \in S\}$.

Proof. $\subseteq$: Let $a \in \text{atoms}(\bigcup_m S)$. Then there is a $D \subseteq HB$ such that $(a, D) \in_m \bigcup_m S$, hence there is an $s \in S$ such that $(a, D) \in_m s$. Therefore $a \in \text{atoms}(s)$ and thus $a \in \bigcup\{\text{atoms}(s) \mid s \in S\}$.

$\supseteq$: Let $a \in \bigcup\{\text{atoms}(s) \mid s \in S\}$. Then there is an $s \in S$ such that $a \in \text{atoms}(s)$. Therefore there is a $D \subseteq HB$ such that $(a, D) \in_m s$. Therefore $(a, D) \in_m \bigcup_m S$ and $a \in \text{atoms}(\bigcup_m S)$. □

Lemma 160. *Let* P *be a definite range restricted program and let* $\beta < \omega$. *Then* $\mathrm{atoms}(dcT_P^\beta) = T_P^\beta$.

Proof. $\mathrm{atoms}(dcT_P^\beta) \subseteq T_P^\beta$: let $(a, D) \in_m dcT_P^\beta$ (thus $a \in \mathrm{atoms}(dcT_P^\beta)$). By Proposition 142 (the left to right implication), there is a derivation of a with respect to P that uses all and only atoms in D. The depth of the derivation (Definition 99) cannot be more than β because it is generated in β steps (as evidenced by $(a, D) \in_m dcT_P^\beta$) and each step can increase the depth by at most 1. Therefore there is a derivation of a of depth at most β in $SG(skT_P^\omega)$ and thus $a \in \mathrm{heads}(skT_P^\beta) = T_P^\beta$, by Lemma 117.

$\mathrm{atoms}(dcT_P^\beta) \supseteq T_P^\beta$: Let $a \in T_P^\beta$. By Lemma 152 there is a support s of a well-founded in $G = SG(skT_P^\omega)$ such that $\mathrm{body}(s) \subseteq T_P^{\beta-1}$. Because s is well-founded and because $\mathrm{head}(s) = a \in T_P^\beta$, there is a derivation D of $\mathrm{head}(s)$ in G that includes s and the depth of D is at most β. $D - s$ does not contain $a = \mathrm{head}(s)$ because D is a derivation (thus acyclic). Therefore D is a derivation of a that uses only atoms in $D - s$ and therefore, by Proposition 142 (the right to left implication), $(a, D) \in_m dcT_P^\omega$. $D - s$ is a well-founded graph that includes supports for each atom in $\mathrm{body}(s)$. For each such support there is a derivation in D by Lemma 140. Each such derivation has depth at most 1 less than D, i.e. $\beta - 1$. Therefore for each $f \in \mathrm{body}(s)$ there is a set of ground atoms S_f such that $(f, S_f) \in_m dcT_P^{\beta-1}$ and $a \notin S_f$. Therefore s satisfies the condition in Definition 137 and $(a, S) \in_m dcT_P^\beta$, i.e. $a \in \mathrm{atoms}(dcT_P^\beta)$, where S is the set of atoms in $D - s$. □

Lemma 161. *Let* P *be a definite range restricted program. Then*
$\mathrm{atoms}(dcT_P^\omega) = T_P^\omega$.

Proof.

$$\begin{aligned}
\mathrm{atoms}(dcT_P^\omega) &= \mathrm{atoms}(\textstyle\bigcup_m \{dcT_P^\beta \mid \beta < \omega\}) && \text{by def., limit case}\\
&= \textstyle\bigcup \{\mathrm{atoms}(dcT_P^\beta) \mid \beta < \omega\} && \text{by Lemma 159}\\
&= \textstyle\bigcup \{T_P^\beta \mid \beta < \omega\} && \text{by Lemma 160}\\
&= T_P^\omega && \text{by def., limit case}
\end{aligned}$$

□

Proposition 162. *Let* P *be a definite program. Then*

$$HI(\mathrm{lfp}(T_P)) = HI(skT_P^\omega) = HI(scT_P^\omega) = HI(dcT_P^\omega)$$

is the unique minimal Herbrand model of P.

Proof. $\mathrm{heads}(skT_P^\omega) = T_P^\omega$ by Lemma 117, $\mathrm{root}(scT_P^\omega) = T_P^\omega$ by Proposition 129, and $\mathrm{atoms}(dcT_P^\omega) = T_P^\omega$ by Lemma 161.

Therefore $HI(skT_P^\omega) = HI(scT_P^\omega) = HI(dcT_P^\omega) = HI(T_P^\omega)$. The rest follows from Proposition 62. □

7.10 STRATIFIABLE PROGRAMS WITH NEGATION

Previous sections mostly assumed definite programs. Many real world problems often require programs with negation. This section introduces the notion of stratified negation and shows that a fixpoint of stratifiable programs can be computed in similar manner as the least fixpoint of definite programs. Stratifiable programs are thus more expressive while the complexity of computing their fixpoint remains the same as for definite programs. See for example [44, 12] for a detailed treatment of stratifiable programs, including motivation and examples. Here, the approach is described for the classical T_P operator and it is shown that it can be applied the same way to all the extended immediate consequence operators defined in this chapter.

Definition 163 (Stratification)**.** *Let S be a set of rules. A stratification of S is a partition $S_1, \dots, S_n$ of S such that*

- *For each relation symbol p there is a stratum S_i, such that all rules of S containing p in their head are members of S_i. In this case one says that the relation symbol p is defined in stratum S_i.*
- *For each stratum S_j and for each positive literal A in the bodies of members of S_j, the relation symbol of A is defined in a stratum S_i with $i \leqslant j$.*
- *For each stratum S_j and for each negative literal $\neg A$ in the bodies of members of S_j, the relation symbol of A is defined in a stratum S_i with $i < j$.*

A set of rules is called stratifiable, if there exists a stratification of it.

Let $S_1, \dots, S_n$ be a stratification for a program P. Then the semantics of P can be defined "hierarchically" [44, 12] by a sequence of fixpoints stratum by stratum. Let $S_{\leqslant i} = \bigcup_{k=1}^{i} S_k$. S_1 is definite and thus it has the least Herbrand model: $M_1 = T_{S_1}^{\omega}$. Then $M_2 = T_{S_{\leqslant 2}}^{\omega}(M_1), \dots, M_m = T_{S_{\leqslant m}}^{\omega}(M_{m-1})$ and the semantics of the program P is the set M_m.

Example 164. *Let S be the following program:*

$\mathsf{project}(\mathsf{proj123}) \leftarrow \top$

$\mathsf{risky}(X) \leftarrow \mathsf{expensive}(X)$

$\mathsf{promising}(X) \leftarrow \mathsf{project}(X),\ \textbf{not}\ \mathsf{risky}(X)$

Then S_1, S_2, where

- $S_1 = \{\mathsf{project}(\mathsf{proj123}) \leftarrow \top, \quad \mathsf{risky}(X) \leftarrow \mathsf{expensive}(X)\}$,
- $S_2 = \{\mathsf{promising}(X) \leftarrow \mathsf{project}(X),\ \textbf{not}\ \mathsf{risky}(X)\}$

is a stratification of S.

The least Herbrand model of S_1 is $M_1 = \{\mathsf{project}(\mathsf{proj123})\}$ and $M_2 = T_{S_1 \cup S_2}^{\omega}(M_1) = \{\mathsf{project}(\mathsf{proj123}), \mathsf{promising}(\mathsf{proj123})\}$. The semantics of S is M_2.

Observation 165. *The stratum by stratum computation of a fixpoint of a program ensures that all information is available when negative literals are evaluated. In other words, if a rule with a negative literal A is used in the computation then all instances of the relational symbol of A that are in the resulting fixpoint of the whole program had already been previously derived in a fixpoint computation using lower strata.*

Therefore if something is derived by a rule in this kind of computation it also belongs to the resulting fixpoint; it will not be non-monotonically removed later. Stratification is thus a way of ordering computation of a fixpoint of a normal program so that non-monotonic changes do not occur. Once an order is determined (all stratifications of a program lead to the same fixpoint), the actual computation is a sequence of the "usual" fixpoint computations. The full account of stratification can be found in Apt et al. [12]. Stratification is well-known, the idea behind it is very intuitive, and its full account with respect to our extended operators would be mostly repetition of the work of Apt and therefore it is not worked out in detail here. We still provide one more viewpoint at Observation 165.

Observation 165 can be seen as giving an operational semantics to stratifiable normal programs. In the deductive database community, they use the notion of "built-in predicates" to provide a more declarative perspective. The idea is to consider the already computed predicates and their negations as built-in predicates. A built-in predicate is not defined in a program but in a system that evaluates the program – it usually is the case for example for the $\leqslant$ predicate. Let us assume that M_k is computed. Then all predicates in M_k and their negations are considered to be built-in predicates. Then $S_{\leqslant k+1}$ becomes a definite program and thus it has the least Herbrand model which can be computed as $T^{\omega}_{S_{\leqslant k+1}}$ (assuming the built-in predicates in $S_{\leqslant k+1}$).

The same stratification idea can be applied to all the extended immediate consequence operators defined in this chapter. First a stratification has to be found for a given program and then extended immediate consequence operators fixpoints are computed in hierarchical manner stratum by stratum. The approach is correct for the same reason it is correct for the T_P operator: negative predicates in a higher stratum depend only on information that is derived in a strictly lower stratum. Therefore no non-monotonic changes can occur during the computation and a resulting fixpoint is reached.

COMPLEXITY. M_n is polynomially computable given a stratification since each single M_i is polynomially computable (in the number of constants in base facts). Stratification is described by a simple syntactic criterion and can be found in polynomial time in the size of a program's dependency graph (a similar notion to the notion of dependency graph, but only for definite programs, is defined in the next chapter in Definition 185). Therefore admitting programs with stratified negation does not increase complexity [44].

While fixpoint computation is easily extended from definite programs to stratifiable normal programs, more adaptation of support-graphs-related notions is necessary.

7.10.1 *Support graphs for programs with negation*

Support graphs in Definition 86 are defined only for definite programs. Support graphs for programs with negation are defined in this section in order to make the framework complete. Support graphs for normal programs are not further explored in this dissertation.

The notion of a support s and the notations $\mathrm{head}(s)$ and $\mathrm{body}(s)$ are extended to normal programs in the obvious way. Note that in this case $\mathrm{head}(s)$ is an atom and $\mathrm{body}(s)$ is a set of literals.

Definition 166 (Support graph, compact support graph). *Let* P *be a normal program. A* support graph *with respect to* P *is a bipartite directed graph* $G = (S, F, E, l)$ *such that*

- S *is a set of ground supports with respect to* P*,*
- F *is a set of nodes,*
- l *is a function labelling nodes with ground literals,* $l : F \to HB_P \cup \{\, \mathbf{not}\ a \mid a \in HB_P\}$*,*
- $E \subseteq (F \times S) \cup (S \times F)$ *and for each* $(x, y) \in E$
  - *either* $x \in F$, $y \in S$, $l(x) \in \mathrm{body}(y)$,
    (i.e. $l(x)$ *may be positive or negative)*

  - *or* $x \in S$, $y \in F$, $l(y) = \text{head}(x)$,
    (i.e. $l(y)$ *is positive)*
- *for each* $s \in S$
  - *for each* $b \in \text{body}(s)$ *exists an* $x \in F$ *such that* $l(x) = b$ *and* $(x, s) \in E$
  - optionally *for* $a = \text{head}(s)$ *exists a* $y \in F$ *such that* $l(y) = a$ *and* $(s, y) \in E$

If the labelling function l *is moreover injective then the support graph is called a* compact support graph. *In compact support graphs, we identify literal nodes with literals.*

Definition 166 is only a small modification of Definition 86 that admits labelling nodes also with negative literals. The definition does not prevent that the support graph contains both an atom and its negation and thus a contradictory set of supports may also be modelled as a support graph. This in itself does not pose a problem because a support graph is only an auxiliary representation of derivations with respect to a program but it requires an additional change of the definition of a well-founded graph.

Definition 167 (Well-founded support graph)**.** *A support graph* $G = (S, F, E, l)$ *with respect to a program* P *is* well-founded, *if it is acyclic and for each* $n \in F$*: if* $l(n)$ *is a positive literal* a *then there is an incoming edge* $(s, n) \in E$ *and there is no* $m \in F$ *such that* $l(m) = \textbf{not } a$.

The definition of a well-founded support graph with respect to a program forbids that it contains both an atom and its negation. Note that nodes with a negative literal as a label do not have supports. All other support graphs related definitions for programs with negation are directly analogous to definitions from Section 7.1.6.

Graphical notation for support graphs for programs with negation also remains the same. An edge between a negative literal and a support may be crossed in order to stress the negative dependency, see the following example.

Example 168. *Let* P *be the following program:*

$r_1 = a \leftarrow \top$

$r_2 = b \leftarrow a$

$r_3 = d \leftarrow \textbf{not } c$

$r_4 = e \leftarrow b, d$

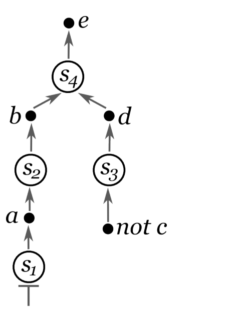
Then $G = (S, F, E, l)$*, where*

- $S = \{s_1, s_2, s_3, s_4\}$,
- $F = \{n_1, n_2, n_3, n_4, n_5\}$,
- $l = \{n_1 \mapsto a, n_2 \mapsto b, n_3 \mapsto \textbf{not } c, n_4 \mapsto d, n_5 \mapsto e\}$,
- *for* E *and* S *see the graphical representation of* G *in Figure 7.10, note that support* s_i *corresponds to an instance of rule* r_i *in this case.*

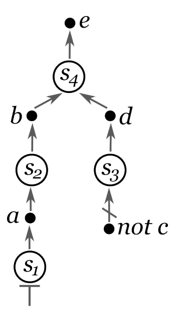
is a support graph with respect to P

7.11 DISCUSSION AND RELATED WORK

This chapter extends the classical notion of immediate consequence operator in several ways that help to retain information useful for reason maintenance of materialized fixpoints (see Chapter 8) and can also help to bring



(a) Support graph G.



(b) Support graph G with the edge from a negative literal stressed.

Figure 7.10: An example of a support graph $G = (S, F, E, l)$.

inferences closer to casual users. All the immediate consequence operators derive the same common information, as Proposition 162 shows. Naive and semi-naive (extended) forward chaining algorithms with operators skT_P and scT_P have the same time complexity, $O(n^k)$, as naive and semi-naive T_P forward chaining which is apparent because the two operators only keep some of the information that T_P derives as well but "forgets" it. Most of their value lies in the fact that they allow for a simple declarative description of reasoning and the corresponding reason maintenance methods. The dcT_P immediate consequence operator derives and retains information that is not derived by any of the other operators, namely dependencies. Dependencies provide global information about atoms with respect to a program. Consequently, the corresponding naive and semi-naive forward chaining algorithms have a worse time complexity ($O(n^{2k})$) than naive and semi-naive forward chaining for the other operators.

To the best of our knowledge, the skT_P, scT_P, and dcT_P operators and the corresponding semi-naive mappings are novel. Especially support counting seems not to be leveraged in any of the methods existing in literature. Methods similar to derivation counting do exist in the area of incremental view maintenance in deductive databases, for example [158], but the notion of a derivation is different. Mumick in [158] counts so called derivation trees which, in contrast to derivations in this work, admit "recursive repetitions" and thus there can be an infinite number of derivations even in the range restricted Datalog case (compare to Proposition 150). Derivation tree counting of [158] is used to define so called duplicate semantics of Datalog programs; i.e. it is tracked not only whether an atom is derived but also how many times it is derived. From this viewpoint, the scT_P and dcT_P operators can be seen as defining alternative multiset Datalog semantics. The skT_P operator defines an alternative set semantics from this perspective. Proposition 162 shows that they are all equivalent and differ only in the information that they provide in addition to the classical T_P set semantics. [159] extends [158] to ensure termination with recursive Datalog programs. The idea is similar to our derivation counting operator: to track the sets of atoms that an atom depends on, i.e. its derivation sets. Our contribution is three-fold: we show how the duplicate semantics with derivation sets can be computed in a declarative and agruably simple way using the dcT_P operator and either the classical naive forward chaining procedure with dcT_P or a slightly modified semi-naive procedure. We also prove the correctness of the approach in the classical logic programming way relying on properties of fixpoint operators. And we show the correspondence of the dependencies to derivations in support graphs. Moreover, and as already implied, our structured graphs framework allows us to precisely relate and compare the different Datalog semantics provided by the classical and extended immediate consequence operators.

A similar approach to defining an extended immediate consequence operator together with the corresponding naive and semi-naive methods can be found in [201] where Shiri and Zheng define a fixpoint computation with uncertainties using multisets. Their interest is in uncertainties and thus the multiplicities they use correspond to truth values in the closed unit interval $[0, 1]$. As a consequence, they for example do not have to deal with possible infinite multiplicities as it is necessary in our work. This difference is significant. While infinite multiplicities may not be essential in typical applications, they are useful and can be encountered in non-Datalog settings (i.e. when functional symbols other than constants are allowed). Infinite multiplicities also posed a new challenge in proving continuity of the multiset extended immediate operators.

8 INCREMENTAL REASON MAINTENANCE

8.1 INTRODUCTION

This chapter studies the problem of maintaining a set of materialized atoms after removing a subset of rules of a program. This problem is called the reason maintenance problem here.

The reason maintenance problem is closely related to two areas of research: reason maintenance [74] (originally called truth maintenance [72]) that is not active anymore, and incremental view maintenance [103] which is studied on and off as part of deductive databases, see Section 8.10 for a thorough review of related work. We first define the reason maintenance problem formally in terms of fixpoints. Then we show a simple solution based on the concept of well-foundedness in Section 8.3. In Section 8.4, we analyse the properties of a fixpoint and provide a method of solving the reason maintenance method without relying on support graphs based on the analysis. This method improves upon the traditional DRed [105] and PF [110] algorithms in that it makes do with the original program – it does not increase its size and it does not introduce negation. The method also takes full advantage of the already computed fixpoint and it handles both changes in data and rules. We provide a detailed comparison of the three algorithms based on two representative examples. Next we proceed to reason maintenance algorithms based on multiset semantics. In Section 8.5, we show an improvement of the algorithm from Section 8.4 that replaces its initial naive step by a direct computation that takes advantage of support counts. Then we study the reason maintenance problem restricted to non-recursive Datalog programs with support counting multiset semantics. For this special case, we formulate a purely incremental reason maintenance algorithm that is a more efficient alternative to the classical counting algorithm of [104] that counts derivation trees. We define a concept of "safeness" that is akin to the idea of local stratification [183, 231] and that could be used to optimize algorithms for the recursive Datalog case. In Section 8.6, we provide several reason maintenance algorithms based on the derivation counting multiset semantics. Although the worst case time complexity (resp. the worst case space complexity) of the derivation-counting-methods is worse than that of the other methods, it provides additional information that can be used for explanation of derived atoms to users. In Section 8.7, we discuss how the methods described in this dissertation can be used for explanation purposes. Then in Sections 8.8 and 8.9, we discuss questions related to practical use and implementation of the suggested methods.

8.2 THE REASON MAINTENANCE PROBLEM

Let P be a definite range restricted program and $D \subseteq P$ a subset of P. The *reason maintenance problem* is the problem of computing $T^{\omega}_{P \setminus D}$ given the fixpoint T^{ω}_{P}.

The reason maintenance problem has a trivial solution which is computing the fixpoint $T^{\omega}_{P \setminus D}$ directly from $P \setminus D$, disregarding T^{ω}_{P}. This solution is obviously inefficient because it necessarily repeats many of the computations used to generate T_P^ω and it can also be seen as a version of the *frame problem*[113] as noted in [165] and [64]. This chapter studies how to solve the reason maintenance problem *incrementally* by leveraging information available from computing the old fixpoint. Here, this modified task is called the incremental reason maintenance problem.

The *incremental reason maintenance problem* is the task to solve the reason maintenance problem incrementally, i.e. to leverage information available from computation of the old fixpoint to minimize the amount of resources necessary to compute the new fixpoint. A formal definition of the incremental maintenance problem is not provided here because it also is a subject of study of this chapter.

Note, that it may be beneficial to extend the standard forward chaining to store additional information with each atom which will then allow for incremental computation of the new fixpoint.

To solve the incremental maintenance problem, it is necessary to decide for each $g \in T_P^\omega$ whether $g \in T_{P\setminus D}^\omega$. This amounts to deciding whether g can be derived from $P \setminus D$. Any algorithm that decides this question has to demonstrate that there is a derivation of g with respect to $P \setminus D$. In other words, the incremental maintenance problem can be solved for example by keeping track of derivations of each atom. Later sections describe methods that avoid keeping all derivations but still can decide whether a derivation exists or not.

This chapter describes several algorithms which solve the incremental maintenance problem and which do not rely on storing the whole support graph.

8.3 REASON MAINTENANCE BY DECIDING WELL-FOUNDEDNESS

Section 7.8, Lemma 152, shows that derivability of an atom and the property of having a well-founded support are equivalent. This observation leads to a simple reformulation of the reason maintenance problem which, together with algorithms from the previous section, provides a simple (but inefficient) solution of the reason maintenance problem. This approach is explored here nevertheless because it can be improved upon and its core idea of showing that an atom has a well-founded support in the new fixpoint is the basis of the advanced algorithms described in next sections.

The problem is to "incrementally" determine the $T_{P\setminus D}^\omega$ subset of T_P^ω. By Lemma 152, it is sufficient to find those atoms in T_P^ω that have a support well-founded in $SG(skT_{P\setminus D}^\omega)$.

Algorithm 8.14: Reason maintenance by well-foundedness

```
Name
  rm-well-founded($skT_P^\omega$, D)

Input
  $G = SG(skT_P^\omega) = (S_G, F_G, E_G, l_G)$ — a finite support graph
  D — a subset of base facts in P (to remove)

Output
  $T_{P\setminus D}^\omega$
```

```
Variables
  F — a set of atoms
  U — a set of supports                         {unsure supports}

Initialization
  U := dependent-supports(P, $skT_P^\omega$, D)                {Alg. 7.6}
  F := $heads(skT_P^\omega) \setminus heads(U)$

begin
  for each $s \in U$
    if is-well-founded(s, $SG(skT_P^\omega)$, $heads(D)$) then   {Alg. 7.11}
      F := $F \cup head(s)$

  return F
end
```

Lemma 169. *Algorithm 8.14 terminates and computes* $T^\omega_{P\setminus D}$.

Proof. The algorithm terminates because Algorithms 7.6 and 7.11 terminate and G is finite.

U is the set of supports with respect to P that depend on an atom from heads(D) in $SG(skT_P^\omega)$, i.e. the set of supports that may be affected by removal of D. Therefore only supports in U need to be checked for well-foundedness in $SG(skT^\omega_{P\setminus D})$. The algorithm checks well-foundedness in $SG(skT^\omega_{P\setminus D})$ by forbidding heads(D) to be in the derivation found by Algorithm 7.11 which shows well-foundedness of s in $SG(skT_P^\omega)$ and thus the derivation, if found, shows well-foundedness of s in $SG(skT^\omega_{P\setminus D})$. □

This algorithm is not efficient for several reasons. The first, which is easily fixable, is that it uses the inefficient algorithm for determining well-foundedness. A modification of the efficient topological-ordering-based Algorithm 7.13 which would allow to forbid some atoms could be used instead (see e.g. Algorithm 8.18 for such a modification in a different context). Other possible minor improvements are also not included for the sake of brevity. More importantly, the algorithm requires that the whole support graph $SG(skT_P^\omega)$ is available. Section 8.4 presents an approach which lifts this requirement.

8.3.1 *Rule updates*

Algorithm 8.14 removes only selected base facts from P. A simple modification of the is−well−founded procedure, Algorithm 7.11, so that it forbids not only atoms but also general rules would be sufficient for generalizing the algorithm to removing any rules from P.

8.3.2 *Comparison with related work*

Algorithm 8.14 is inspired by classical reason maintenance algorithms and specifically by reason maintenance of the JTMS [73] kind. It is a less efficient but simpler version of monotonic JTMS reason maintenance, see Section 8.10 for a detailed description of the original algorithm.

8.4 REASON MAINTENANCE WITHOUT SUPPORT GRAPHS

The reason maintenance problem can be solved by finding supports for atoms in the old fixpoint which are well-founded in the new fixpoint. Previous sections present an approach to this idea but they require the full support graph corresponding to the fixpoint. This section shows how this requirement can be removed. The resulting algorithm is a modification of the classical semi-naive forward chaining where the initialization step is naive. Section 8.5 then shows how the first naive step can be replaced by a more efficient computation that uses support counts.

Let us explore the problem more formally. The following notation is used throughout the rest of this section.

Notation 170. *Let* P *be a definite range restricted Datalog program,* $D \subseteq P$ *a set of rules to remove, and let* $G = SG(skT_P^\omega)$ *be the support graph induced by* skT_P^ω*. Then*

- $U = \{a \in T_P^\omega \mid (\exists d \in D)$ *a depends on* d *in* $G\}$,
- $K = T_P^\omega \setminus U$ *– the set of atoms that do not depend on a rule from* D *in* G,
- $O = U \cap T_{P\setminus D}^\omega$ *– the set of atoms that depend on a rule from* D *in* G *but not strongly,*

U *is the set of [u]nsure atoms, i.e. the set of atoms that may have to be removed as a result of removing* D*.* K *is the set of atoms to [k]eep, i.e. the set of atoms that definitely are not affected by the removal of* D*.* O *is the set of [o]therwise supported atoms, i.e. the set of atoms from* U *that are derivable without* D*.*

T_P^ω *will also be referred to as the* old fixpoint *and* $T_{P\setminus D}^\omega$ *as the* new fixpoint.

The following lemma and Figure 8.1 illustrate relationships between U, K, and Os.

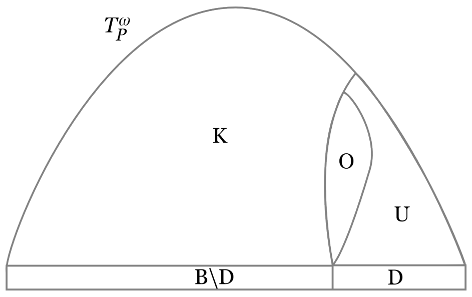


Figure 8.1: Illustration of how K, U, and O relate in the case that D is a set of base facts. B is the set of all base facts in P. With removal of arbitrary rules, there may be more "U" sets each having its own "O" subset, see Figure 8.2.

Lemma 171. *The following holds:*

1. $U \cup K = T_P^\omega$,
2. $U \cap K = \emptyset$,
3. $O \cap K = \emptyset$.

Proof. Follows immediately from definition. □

Lemma 172. *The following holds:*

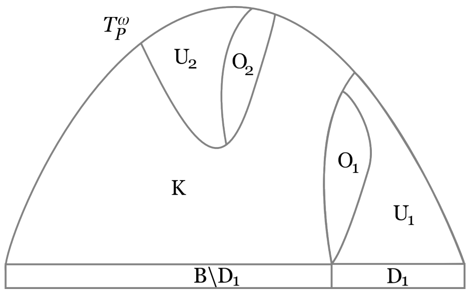


Figure 8.2: Illustration of how K, U, and O relate in the general case. B is the set of all base facts in P. D_1 is a set of base facts to remove, D_2 is a set of rules to remove, $D = D_1 \cup D_2$. U_1 is the set of atoms that depend on a fact from D_1. U_2 is the set of atoms that depend on a rule from D_2. $U = U_1 \cup U_2$.

1. $K \subseteq T^\omega_{P\setminus D}$,
2. $T^\omega_{P\setminus D} = K \cup O$,
3. $T_{P\setminus D}(K) \subseteq T^\omega_{P\setminus D}$.

Proof. Point 1: Let $a \in K$. $K = T_P^\omega \setminus U$ and therefore a does not depend on any rule in D in G by the definition of U. In other words, no derivation of a in G includes a support that includes a rule from D or any atom that depends on a rule from D in G. Therefore a is in $SG(skT^\omega_{P\setminus D})$ which, by an observation analogous to Observation 61, is a subgraph of G in which no node depends on a rule from D. Thus $a \in heads(skT^\omega_{P\setminus D})$. $heads(skT^\omega_{P\setminus D}) = T^\omega_{P\setminus D}$, by Lemma 117, therefore $a \in T^\omega_{P\setminus D}$. In summary $K \subseteq T^\omega_{P\setminus D}$.

Point 2: $K \cup O = (T_P^\omega \setminus U) \cup (U \cap T^\omega_{P\setminus D}) \supseteq T^\omega_{P\setminus D}$, from Notation 170 and because $T^\omega_{P\setminus D} \subseteq T_P^\omega$ by Observation 61. $K \cup O \subseteq T^\omega_{P\setminus D}$ by Point 1 and because $O \subseteq T^\omega_{P\setminus D}$ by definition.

Point 3 follows from Point 1 and the fact that $T^\omega_{P\setminus D}$ is the least fixpoint of $T_{P\setminus D}$. □

Lemma 172 shows a way to compute the new fixpoint by determining the sets K and O. The core of the problem lies in determining the set O, i.e. those atoms that lose some derivations after removing D but are derivable nevertheless. The following two propositions show that the new fixpoint can be computed by forward chaining on the set K.

Proposition 173. $T^\omega_{P\setminus D} = T^\omega_{P\setminus D}(K)$.

Proof. $T_{P\setminus D}(\emptyset) \subseteq T_{P\setminus D}(K)$ by monotonicity of T_P (Lemma 47 and Lemma 60). Therefore also $T^n_{P\setminus D} \subseteq T^n_{P\setminus D}(K)$, for all $n \in \mathbb{N}_1$, by monotonicity of T_P.

$\subseteq$: Let $a \in T^\omega_{P\setminus D} = \bigcup\{T^\beta_{P\setminus D} \mid \beta < \omega\}$. Then there is a β such that $a \in T^\beta_{P\setminus D} \subseteq T^\beta_{P\setminus D}(K) \in \{T^\beta_{P\setminus D}(K) \mid \beta < \omega\}$. Therefore $a \in T^\omega_{P\setminus D}(K)$. In summary $T^\omega_{P\setminus D} \subseteq T^\omega_{P\setminus D}(K)$.

$\supseteq$: Let $a \in T^\omega_{P\setminus D}(K)$. There is an α such that $a \in T^\alpha_{P\setminus D}(K)$. By Lemma 172, it holds that $K \subseteq T^\omega_{P\setminus D}$. Therefore there is a β such that $K \subseteq T^\beta_{P\setminus D}$. Thus, $T^\alpha_{P\setminus D}(K) \subseteq T^{\beta+\alpha}_{P\setminus D}$ by monotonicity of T_P. Together, $a \in T^{\beta+\alpha}_{P\setminus D} \subseteq T^\omega_{P\setminus D}$. And in summary, $T^\omega_{P\setminus D}(K) \subseteq T^\omega_{P\setminus D}$. □

An immediate corollary of Proposition 173 is the following proposition which states that all atoms in O are eventually derived by forward chaining on K.

Proposition 174. *If* $a \in O$ *then there is an* $n \in \mathbb{N}_1$ *such that* $a \in T^n_{P \setminus D}(K)$.

Proof. Let $a \in O$. $O = U \cap T^\omega_{P \setminus D}$, therefore $a \in T^\omega_{P \setminus D}(K)$ by Proposition 173. P (and therefore also $P \setminus D$) is a definite range restricted Datalog program. Therefore the fixpoint is reached in a finite number of steps and therefore there is an $n \in \mathbb{N}_1$ such that $a \in T^n_{P \setminus D}(K)$. □

The set U can easily be determined by the semi-naive forward chaining Algorithm 7.6 modified according to Observation 122, with T^ω_P as input instead of skT^ω_P. Then atoms in K are those in the old fixpoint T^ω_P that are not in U.

Therefore a reason maintenance algorithm to compute the new fixpoint can be summarized as follows:

1. Determine U (and therefore K).
2. Compute $T^\omega_{P \setminus D}(K)$

See Algorithm 8.15 for a complete specification.

Algorithm 8.15: Reason maintenance without support graphs.

```
Name
  FP-update(P, $T^\omega_P$, D)

Input
  P — a range restricted definite \datalog program
  D — a subset of P, the set of rules to remove
  $T^\omega_P$ — the least fixpoint of $T_P$

Output
  $T^\omega_{P \setminus D}$ — the new least fixpoint

Variables
  $U, K, \Delta, F$ — sets of atoms

Initialization
  U := heads(dependent-supports(P, $T^\omega_P$, D))
  K := $T^\omega_P \setminus U$
  $\Delta$ := $T_{P \setminus D}(K) \setminus K$
  F := K

begin
  while $\Delta \neq \emptyset$
  begin
    F := $F \cup \Delta$
    $\Delta_F$ := $T_{P \setminus D}(F, \Delta)$     {"new" atoms}
    $\Delta$ := $\Delta_F \setminus F$           {forbid redundant and cyclic derivations}
  end

  return F
end
```

Proposition 175. *Algorithm 8.15 terminates and computes* $T^{\omega}_{P\setminus D}$.

Proof. The algorithm correctly computes the set K, by definition of K and because the dependent supports algorithm is correct (Lemma 121).

The algorithm computes $T^{i}_{P\setminus D}(K)$ where i is the $i-th$ iteration of the while loop in the algorithm. The algorithm computes $T^{\omega}_{P}(K)$ by a similar argument as for the correctness of classical semi-naive forward chaining, see Proposition 109. HB_P and $HB_{P\setminus D}$ are finite (Observation 25), hence the algorithm terminates.

The rest follows immediately from Proposition 173. □

Let Δ_i be the Δ in the i-th iteration of the algorithm. Notice that $(\bigcup_{i=0}^{n} \Delta_i)$ $= O$, where n is the number of the iteration in which fixpoint is reached.

Algorithm 8.15 does not require the support graph to be stored in order to compute the new fixpoint from the old one. Notice that the first forward chaining step (in the initialization) is naive, the rest of the computation continues semi-naively. The first naive forward chaining step can be avoided by leveraging support counts as it is described in Section 8.5.

COMPLEXITY. The worst-case complexity of Algorithm 8.15 is the same as the worst-case complexity of semi-naive forward chaining, i.e. $O(n^k)$ – polynomial in the number of constants in base facts of the input program (recall that k is the maximum over all rules in P of the number of variables in that rule).

STRATIFIABLE NORMAL PROGRAMS. Algorithm 8.15 can be easily extended to stratifiable normal programs the same way as fixpoint computation is done for stratifiable normal programs in Section 7.10.

8.4.1 *Rule updates*

It is worth stressing that Algorithm 8.15 works for both base fact *and* general rule updates. In fact, if a rule, which is not a base fact, is to be removed then all atoms that directly depend on it can be determined by a single query; by executing the rule's body. Indeed, exactly this is the main reason for computing the least fixpoint of a program in advance.

Atoms that indirectly depend on a rule to be removed can then be computed by the usual semi-naive forward chaining algorithm. This situation is depicted in Figure 8.2 by the sets with index 2 (D_2 is not depicted there; it consists of rules of supports "on the border of U_2 and K").

8.4.2 *Comparison with related work*

Reason maintenance without support graphs is closely related to the problem of incremental view maintenance in deductive databases. The problem has been studied on and off since around 1980, see for example a survey article [103] by Gupta and Mumick, authors of the probably most popular DRed algorithm [105]. In line with the deductive database perspective, all incremental view maintenance algorithms distinguish between extensional (base facts) and intensional (rules that are no base facts) part of a database. The incremental view maintenance algorithms solve the problem of updating a view upon a change in the extensional part of the database. The algorithms do not directly handle rule changes but they can be extended to do so. It is important to clearly distinguish the incremental view maintenance problem from the view update problem [102]. The view update problem is trying to solve the question how to change a program so that a given formula cannot (resp. can) be derived from it. This section provides a detailed comparison of the incremental view maintenance methods with our approach.

The most prominent incremental view maintenance algorithms are the DRed (derive and rederive) algorithm [105] and the PF (propagate filter) algorithm [110]. Both work on the same principle of deriving an overestimation of deleted atoms (i.e. the set U) and then finding alternative derivations for them (i.e. determining the set O). The DRed algorithm first derives the whole overestimation and only then finds alternative derivations. The PF algorithm, which was originally developed in the context of top-down memoing [69], finds alternative derivations (filters) as soon as an atom to possibly be deleted is derived (propagated). Both algorithms perform the two steps by evaluating a transformed version of the original rules.

Staudt and Jarke developed [207] a purely declarative version of DRed. Their algorithm, however, transforms the original program into even more rules than DRed itself meaning that to perform maintenance a possibly substantially bigger program has to be evaluated. Also, the transformed program includes negation even if the original program does not. Recently, Volz, Staab, and Motik extended [219] Staudt and Jarke's version of DRed to handle rule changes. For example, the Volz, Staab, and Motik version of DRed transforms 12 RDF semantics Datalog rules into a maintenance program of 60 rules [219]. In comparison, Algorithm 8.15 makes do with the original 12 rules. Moreover, these purely declarative maintenance methods by Staudt and Jarke and Volz et al. handle atom insertions and deletions at one time by a single maintenance program. Their method leads [219] to complete recomputation in case of a single (ternary) predicate axiomatization of RDF(S) which is a significant disadvantage especially in the area of semantic web where this axiomatization is very common. In our case, insertions are handled by semi-naive forward chaining, deletions by Algorithm 8.15 and always only the relevant part of a predicate's extension is recomputed, including the case of a single predicate axiomatization of RDF(S).

Let us now compare DRed [105], the PF [110] algorithm and Algorithm 8.15 in detail. In the Encyclopedia of database systems [135], there is a comparison of the DRed and the PF algorithm [69] on a classic reachability example. Let us reproduce the comparison here and let us add evaluation of Algorithm 8.15 on the same example, see Example 176.

Example 176. *Let P be a Datalog program that consists of the following two rules and of a set of base facts represented as a graph in Figure 8.3.*

r_1: $\mathrm{reach}(S, D) \leftarrow \mathrm{edge}(S, D)$

r_2: $\mathrm{reach}(S, D) \leftarrow \mathrm{reach}(S, I), \mathrm{edge}(I, D)$

Note that S can stand for "source", D for "destination", and I for "intermediate".

The DRed algorithm works in three steps: first it overestimates deletions by deriving all atoms that lose at least one derivation, second it computes alternative derivations for potential deletions, and third it computes the actual changes. Step 1 and 2 are performed by evaluating rules created by transforming rules of the original program. Notation from [69] is used here. Following is the transformed program P' (excluding the base facts, which remain the same):

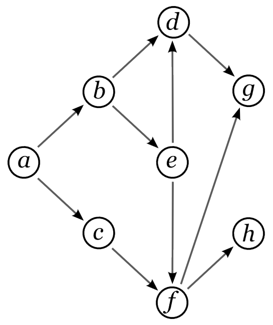


Figure 8.3: A reproduction of a diagram found in [69]. A graph depicting the (extensional) edge predicate. There is a directed edge from x to y iff $\text{edge}(x,y)$ holds. The edge (e,f) is to be removed.

$$\begin{aligned}
&\Delta^-(r_1): && \delta^-(\text{reach}(S,D)) \leftarrow \delta^-(\text{edge}(S,D))\\
&\Delta^-(r_{21}): && \delta^-(\text{reach}(S,D)) \leftarrow \delta^-(\text{reach}(S,I)), \text{edge}(I,D)\\
&\Delta^-(r_{22}): && \delta^-(\text{reach}(S,D)) \leftarrow \text{reach}(S,I), \delta^-(\text{edge}(I,D))\\
&\Delta^r(r_1): && \delta^+(\text{reach}(S,D)) \leftarrow \delta^-(\text{reach}(S,D)), \text{edge}^\nu(S,D)\\
&\Delta^r(r_2): && \delta^+(\text{reach}(S,D)) \leftarrow \delta^-(\text{reach}(S,D)), \text{reach}^\nu(S,I), \text{edge}^\nu(I,D)
\end{aligned}$$

Δ^- rules compute potential deletions, denoted as $\delta^-(\text{reach})$. There is one Δ^- rule for each body atom of each original rule. Δ^- rules use current estimates of the potential deletions, i.e. evaluation of Δ^- rules emulates the original semi-naive computation for reach. Note that rules $\Delta^-(r_1)$ and $\Delta^-(r_{22})$ intuitively say that deletions in the edge relation potentially lead to deletions in the reach relation. Rule $\Delta^-(r_{21})$ says that potential deletions in the reach relation lead to potentially more deletions in the reach relation.

Δ^r rules determine rederivations of the potential deletions. The superscript ν indicates the use of the current instance (i.e. the relation after incorporating the already known changes) of the relation corresponding to the respective subgoal.

Table 8.4 shows a run of the DRed algorithm for deletion of the atom $\text{edge}(e,f)$. For brevity, $\text{edge}(x,y)$ is shortened as xEy and xy stands for $\text{reach}(x,y)$.

You can see that the DRed algorithm identifies tuples $\{(e,f), (e,h), (b,f), (b,h)\}$ as to be removed from the reach relation.

The PF (Propagate Filter) algorithm derives (propagates) possible deletions and immediately looks for alternative derivation (i.e. immediately filters the new possible deletions) in each step. For example, the algorithm begins by propagating deletion of $\text{edge}(e,f)$ to $\text{reach}(e,f)$. $\text{reach}(e,f)$ is a possible deletion and it is not filtered because no alternative derivation is found. Thus $\text{reach}(e,f)$ is an actual deletion and is propagated further. Table 8.5 shows a whole run of the PF algorithm on the same example.

The PF algorithm also identifies the tuples $\{(e,f),(e,h),(b,f),(b,h)\}$ as to be removed from the reach relation.

Let us now explore a run of Algorithm 8.15. Algorithm 8.15 first determines the set of possible deletions U by calling Algorithm 7.6 and then it performs semi-naive forward-chaining on the set K (the complement of U in the fixpoint) and thus computes, using the new updated program, the new fixpoint. Figure 8.6 shows the old fixpoint of program P with the sets U, K,

DRed algorithm		
Step 1	Compute overestimate of potential deletions	$\delta^-(\text{reach})$
	$\Delta^-(r_1)$:$\delta^-(\text{reach}(S,D)) \leftarrow \delta^-(\text{edge}(S,D))$	*ef*
	$\Delta^-(r_{21})$:$\delta^-(\text{reach}(S,D)) \leftarrow \delta^-(\text{reach}(S,I)), \text{edge}(I,D)$	*eg eh*
	$\Delta^-(r_{22})$:$\delta^-(\text{reach}(S,D)) \leftarrow \text{reach}(S,I), \delta^-(\text{edge}(I,D))$	*af bf*
	Repeat until no change: No new tuples for $\Delta^-(r_1)$ and $\Delta^-(r_{22})$	
	$\Delta^-(r_{21})$:$\delta^-(\text{reach}(S,D)) \leftarrow \delta^-(\text{reach}(S,I)), \text{edge}(I,D)$	*ag ah bg bh*
	Last iteration does not generate any new tuples	
Step 2	Find alternative derivations to remove potential deletions	$\delta^+(\text{reach})$
	$\Delta^r(r_1)$:$\delta^+(\text{reach}(S,D)) \leftarrow \delta^-(\text{reach}(S,D)), \text{edge}(S,D)$	
	$\Delta^r(r_2)$:$\delta^+(\text{reach}(S,D)) \leftarrow \delta^-(\text{reach}(S,D)), \text{reach}^\nu(S,I), \text{edge}^\nu(I,D)$	*eg af ag ah bg*
Step 3	Compute actual changes to reach	$\Delta^-(\text{reach})$
	$\Delta^-(\text{reach}) = \delta^-(\text{reach}) - \delta^+(\text{reach})$	*ef eh bf bh*

Table 8.4: A run of the DRed algorithm for Example 176. xy stands for $\text{reach}(x,y)$. Atom $\text{edge}(e,f)$ is being removed.

and O marked. The number of a row corresponds to the forward chaining iteration in which the atoms in the row are first derived. Atoms are ordered alphabetically in each row.

Table 8.7 shows a run of Algorithm 8.15 for the same example, see also Figure 8.6.

As you can see, Algorithm 8.15 directly computes the new fixpoint. The set of tuples deleted from the reach relation is the same as in the previous two algorithms: $U \setminus (K \cup \{\textit{af bg eg ag ah}\}) = \{\textit{ef eh bf bh}\}$.

Algorithm 8.15 fully takes advantage of the already computed fixpoint when computing the set U of possible deletions.

Let us explore how the original fixpoint is leveraged in more detail on another example where it is better visible. Assume again the reachability program from Example 176 but this time, let the edge relation be as in Figure 8.8 and let the second rule be r_2: $\text{reach}(S,D) \leftarrow \text{edge}(S,I), \text{reach}(I,D)$. The exact specification of a program makes a difference. The original rule r_2 would not help to fully leverage the existing fixpoint in this case as it would first match the original unsure atom $\text{reach}(a,b)$ and the rest would be derived edge by edge in subsequent iterations.

Let us first examine a run of the DRed algorithm in the case that the atom $\text{edge}(a,b)$ is to be removed from the materialized fixpoint, see Table 8.9. The

PF algorithm				
Propagate			Filter	
	Rule	δ^-(reach)	δ^+(reach)	Δ^-(reach)
δ^-(edge):{(e, f)}	$\Delta^-(r_1)$	*ef*	{}	*ef*
Δ^-(reach):{(e, f)}	$\Delta^-(r_{21})$	*eg eh*	*eg*	*eh*
Δ^-(reach):{(e, h)}	$\Delta^-(r_{21})$	{}		{}
δ^-(edge):{(e, f)}	$\Delta^-(r_{22})$	*af bf*	*af*	*bf*
Δ^-(reach):{(b, f)}	$\Delta^-(r_{21})$	*bg bh*	*bg*	*bh*
Δ^-(reach):{(b, h)}	$\Delta^-(r_{21})$	{}		{}

Table 8.5: A run of the PF algorithm for Example 176. xy stands for reach(x, y). Atom edge(e, f) is being removed.

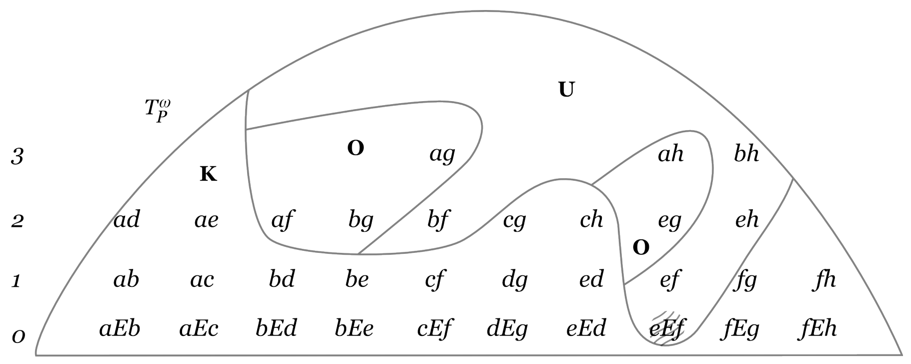


Figure 8.6: A diagram showing the relationship between sets U, K, and O as computed by Algorithm 8.15 for Example 176. xy stands for reach(x, y), xEy stands for edge(x, y). Atom eEf (i.e. edge(e, f)) is being removed.

transformed program P′ first derives δ^-(reach(a, b)) from δ^-(edge(a, b)) using rule $\Delta^-(r_1)$. There is only the (a, b) tuple in the δ^-(edge) relation and no other tuple can be derived in the relation. Therefore the rest of the derivation is done only by rule $\Delta^-(r_{22})$. The rest of the potential deletions is derived in one application of the $\Delta^-(r_{21})$ rule. The set of potential deletions determined by the first phase of the algorithm is {*aEb ab ac ae af ag*}. The second phase determines one rederivation *ag* in one step by application of the rule $\Delta^r(r_2)$.

Algorithm 8.15 also leverages the original fixpoint in computing the set of possible deletions. First, it derives reach(a, b) from edge(a, b) by applying rule r_1. Then the whole set of possible deletions U = {*aEb ab ac ae af ag*} is derived in one step by application of (the modified) rule r_2. The second step determines one rederivation *ag* by application of rule r_2. See Table 8.10.

In this last example, the overestimation phase of both algorithms derives the same information by two rule applications. With the unchanged rule, the same information would be derived in six rule applications. In general, the difference is the bigger the longer derivations exist with respect to the original program. This comparison holds also for the PF algorithm

Step 1	Algorithm 8.15: the call of Algorithm 7.6		
	F	Δ_F	Δ
Initialization	$\emptyset$	*eEf*	*eEf*
	eEf	*ef*	*ef*
	eEf ef	*bf af eg eh*	*bf af eg eh*
	eEf ef bf af eg eh	*bg bh ag ah*	*bg bh ag ah*
	eEf ef bf af eg eh bg bh ag ah	*ag*	$\emptyset$
Step 2	The rest of Algorithm 8.15		
Initialization	$U = \{eEf\ ef\ bf\ af\ eg\ eh\ bg\ bh\ ag\ ah\}$ $K = T_P^\omega \setminus U$ $\Delta = \{af\ bg\ eg\ ag\}$ $F = K$		
	$F = K \cup \{af\ bg\ eg\ ag\}$	$\Delta = \{ah\}$	
	$F = K \cup \{af\ bg\ eg\ ag\ ah\}$	$\Delta = \emptyset$	

Table 8.7: A run of Algorithm 8.15 for Example 176. xy stands for $\text{reach}(x, y)$, xEy stands for $\text{edge}(x, y)$. Atom eEf (i.e. $\text{edge}(e, f)$) is being removed.

and Algorithm 8.15 because the overestimation phase of the PF algorithm is comparable.

8.5 SUPPORT COUNTING

The initial naive step in reason maintenance without support graphs can be removed by keeping and leveraging the number of supports per atom, see Section 8.5.1.

Section 8.5.2 shows how support counts can be used to improve algorithms that decide well-foundedness of a support.

Section 8.5.3 presents a novel support counting reason maintenance algorithm specialized to non-recursive Datalog programs. It also defines a "safeness" criterion, important for the *recursive* Datalog case, that enables identification of certain acyclic subgraphs of the whole $SG(skT_P^\omega)$ graph which can be processed more efficiently than the algorithms for recursive Datalog programs allow without this criterion.

8.5.1 *Support counting reason maintenance*

Algorithm 8.15 requires an initial naive forward chaining step. The following two lemmas show how this step can be replaced by a simple comparison of atom support counts. The resulting modification of Algorithm 8.15 computing the T_P^ω fixpoint is then presented as Algorithm 8.16 and a modification to compute the scT_P^ω fixpoint is presented as Algorithm 8.17.

Let us first extend Notation 170 and introduce the set of supports SU analogous to the set U.

Notation 177. *Let* P *be a definite range restricted Datalog program,* $D \subseteq P$ *a set of rules to remove, and let* $G = SG(skT_P^\omega)$ *be the support graph induced by* skT_P^ω*. Then* $SU = \{t \in skT_P^\omega \mid t$ *depends on a rule from* D *in* $G\}$.

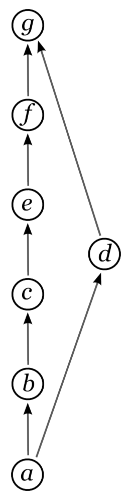


Figure 8.8: A graph depicting the (extensional) edge predicate. There is a directed edge from x to y iff $\mathrm{edge}(x, y)$ holds. The edge (a, b) is to be removed.

The set SU has the following properties:

- $\mathrm{heads}(SU) = U$, by Observation 123,
- $\mathrm{supports}(D) \subseteq SU$, directly from Notation 177 and Terminology 95 ("Supports, depends").

Lemma 178. *Assume Notations 170 and 177 and let* $a \in^x_m scT^\omega_P$. *Then* $a \in O$ *and* $a \in T_{P\setminus D}(K)$ *iff* $x > |\{t \in SU \mid \mathrm{head}(t) = a\}|$.

Proof. $\Rightarrow$: Let a be as in the lemma. $a \in T_{P\setminus D}(K)$ and $T_{P\setminus D}(K) \subseteq T^\omega_{P\setminus D}$ by Lemma 172 point 3. Therefore there is a support $s = (r, \sigma)$ of a (i.e. $\mathrm{head}(s) = a$) such that $r \in P \setminus D$ and $\mathrm{body}(s) \subseteq K$. In addition, $U \cap K = \emptyset$ (Lemma 171) and thus $(\forall x \in \mathrm{body}(s))x \notin U$ which means that s does not depend on a rule from D in G because it does not directly depend on any atom that depends on a rule from D (i.e. atoms in U) and $r \notin D$. Therefore $s \notin SU' = \{t \in SU \mid \mathrm{head}(t) = a\}$ and we found at least one support of a that is in skT^ω_P and not in SU'. Therefore $x > |\{t \in SU \mid \mathrm{head}(t) = a\}|$ by Lemma 132.

$\Leftarrow$: Let a be as in the lemma. Then there is a support $s = (r, \sigma)$ of a (i.e. $\mathrm{head}(s) = a$) in G which does not depend on a rule from D in G, i.e. no derivation of s in G includes a support that includes a rule from D or any atom that depends on a rule from D. Thus s is in $skT^\omega_{P\setminus D}$ and therefore $\mathrm{head}(s) = a \in O$ because $O = \mathrm{heads}(SU) \cap T^\omega_{P\setminus D}$. s does not depend on any atom in U in G (otherwise it would depend on a rule in D) therefore no atom from $\mathrm{body}(s)$ is in U. Because also $T^\omega_P = U \cup K$, it must hold that $\mathrm{body}(s) \subseteq K$. Hence $a = \mathrm{head}(s) \in T_{P\setminus D}(K)$. □

Notice that Lemma 178 does not say whether or not the extra support of a is well-founded. The following lemma shows that any extra support of a, i.e. a support of a that is not in SU, is well-founded because, intuitively, all not well-founded supports of a depend strongly on a and thus they depend on a rule from D, hence they are in SU. See Example 179 for an example of a not well-founded support.

DRed algorithm		
Step 1	Compute overestimate of potential deletions	$\delta^-(\mathrm{reach})$
	$\Delta^-(r_1)$:$\delta^-(\mathrm{reach}(S,D)) \leftarrow \delta^-(\mathrm{edge}(S,D))$	*ab*
	$\Delta^-(r_{21})$:$\delta^-(\mathrm{reach}(S,D)) \leftarrow \delta^-(\mathrm{edge}(S,I)), \mathrm{reach}(I,D)$	*ac ae af ag*
	$\Delta^-(r_{22})$:$\delta^-(\mathrm{reach}(S,D)) \leftarrow \mathrm{edge}(S,I), \delta^-(\mathrm{reach}(I,D))$	
	Repeat until no change: No new tuples for $\Delta^-(r_1)$ and $\Delta^-(r_{22})$	
	Last iteration does not generate any new tuples	
Step 2	Find alternative derivations to remove potential deletions	$\delta^+(\mathrm{reach})$
	$\Delta^r(r_2)$:$\delta^+(\mathrm{reach}(S,D)) \leftarrow \delta^-(\mathrm{reach}(S,D)), \mathrm{reach}^\nu(S,I), \mathrm{edge}^\nu(I,D)$	*ag*
Step 3	Compute actual changes to reach	$\Delta^-(\mathrm{reach})$
	$\Delta^-(\mathrm{reach}) = \delta^-(\mathrm{reach}) - \delta^+(\mathrm{reach})$	*ab ac ae af*

Table 8.9: A run of the DRed algorithm for the example in Figure 8.8. xy stands for $\mathrm{reach}(x,y)$. Atom $\mathrm{edge}(a,b)$ is being removed.

Example 179. *Let* P *be the program* $\{r_1 = a \leftarrow \emptyset,\ \ r_2 = b \leftarrow a,\ \ r_3 = a \leftarrow b\}$. *Then* $skT_P^\omega = \{s_1 = (r_1,\emptyset), s_2 = (r_2,\emptyset), s_3 = (r_3,\emptyset)\}$ *and the support* s_3 *is not well-founded while supports* s_1 *and* s_2 *are well-founded. See also Figure 8.11.*

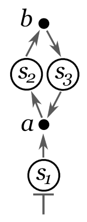


Figure 8.11: The compact support graph $SG(skT_P^\omega)$ for P from Example 179

Lemma 180. *Let* P *be a definite range restricted program. Each support of an atom* $g \in T_P^\omega$ *not well-founded in* $G = SG(skT_P^\omega)$ *depends strongly on the node labelled* g *in* G *(note that* G *is compact by Definition 88).*

Proof. Let s be a support of g not well-founded in $G = SG(skT_P^\omega)$. Assume, by contradiction, that s does not strongly depend on $l_G^{-1}(g)$. Then there is a derivation $D = (S_D, F_D, E_D, l_D)$ of s which does not include $l_G^{-1}(g)$. D is a minimal well-founded subgraph of G with s as its only sink node. Then $D' = (S_D, F_D \cup \{l_G^{-1}(g)\}, E_D \cup \{(s, l_G^{-1}(g))\}, l_G \restriction D)$ is a derivation of $g = \mathrm{head}(s)$ which includes s. Therefore s is well-founded which is a contradiction. □

Lemma 178 can be used to speed up computation of the new fixpoint in Algorithm 8.15 by computing the first delta using support counts instead of

Step 1	Algorithm 8.15: the call of Algorithm 7.6		
	F	Δ_F	Δ
Initialization	$\emptyset$	*aEb*	*aEb*
	aEb	*ab*	*ab*
	aEb ab	*ac ae af ag*	*ac ae af ag*
	aEb ab ac ae af ag	$\emptyset$	$\emptyset$
Step 2	The rest of Algorithm 8.15		
Initialization	$U = \{aEb\ ab\ ac\ ae\ af\ ag\}$ $K = T_P^\omega \setminus U$ $\Delta = \{aEb\ ab\ ac\ ae\ af\ ag\}$ $F = K$		
	$F = K \cup \{ag\}$	$\Delta = \emptyset$	

Table 8.10: A run of Algorithm 8.15 for the example in Figure 8.8. xy stands for $\text{reach}(x,y)$, xEy stands for $\text{edge}(x,y)$. Atom aEb (i.e. $\text{edge}(a,b)$) is being removed.

by naive forward-chaining, see Algorithm 8.16 for the respective modification.

Algorithm 8.16: Reason maintenance with support counts and classical fixpoint output.

```
Name
  $scT_P$-$T_P$-update(P, $scT_P^\omega$, D)

Input
  P — a range restricted definite \datalog program
  D — a subset of P, the set of rules to remove
  $scT_P^\omega$ — the least fixpoint of $scT_P$

Output
  $T_{P\setminus D}^\omega$ — the new least fixpoint

Variables
  $F, K, \Delta$ — sets of atoms
  $SU$ — a set of supports

Initialization
  $T_P^\omega$ := $\text{root}(scT_P^\omega)$
  SU := dependent-supports(P, $T_P^\omega$, D)
  K := $T_P^\omega \setminus \text{heads}(SU)$
  $\Delta$ := $\{\, a \in \text{heads}(SU) \mid a \in_m^x scT_P^\omega$ and $x > |\{t \in SU \mid \text{head}(t) = a\}| \,\}$
  F := K

begin
  while $\Delta \neq \emptyset$
  begin
    F := $F \cup \Delta$
    $\Delta_F$ := $T_{P\setminus D}(F, \Delta)$     {"new" atoms}
    $\Delta$ := $\Delta_F \setminus F$           {forbid redundant and cyclic derivations}
  end
  return F
end
```

Proposition 181. *Algorithm 8.16 terminates and computes $T^{\omega}_{P\setminus D}$.*

Proof. For the initial delta it holds that $\Delta = \{a \in O \mid a \in T_{P\setminus D}(K)\}$ by Lemma 178. Then

Δ	$= \{a \in O \mid a \in T_{P\setminus D}(K)\}$	Lemma 178
	$= O \cap T_{P\setminus D}(K)$	trivial
	$= U \cap T_{P\setminus D}(K)$	Notation 170 for O, Lemma 172 point 3
	$= T_{P\setminus D}(K) \setminus K$	Lemma 171, $T_{P\setminus D}(K) \subseteq T^{\omega}_{P}$

Let us review the last step in detail: $U \cap K = \emptyset$ and $U \cup K = T^{\omega}_{P}$ by Lemma 171 and "intersecting with a set (i.e. U) is the same as subtracting its complement (i.e. K)".

Thus the variables Δ, Δ_F, and F after initialization have the same value as variables in Algorithm 8.15 after initialization. The rest follows from Proposition 175. □

Algorithm 8.16 takes the old fixpoint with support counts and computes the new fixpoint without support counts. The new fixpoint is computed without support counts for clarity reasons only, obviously, in a real-world setting one will want to compute the new support counting fixpoint. This insufficiency is remedied in Algorithm 8.17 which computes a multiset initialization and then continues with support counting semi-naive forward chaining.

Algorithm 8.17: Reason maintenance with support counts and support counting fixpoint output.

```
Name
  $scT_P$-$scT_P$-update(P, $scT^{\omega}_P$, D)

Input
  P — a range restricted definite \datalog program
  D — a subset of P, the set of rules to remove
  $scT^{\omega}_P$ — the least fixpoint of $scT_P$

Output
  $scT^{\omega}_{P\setminus D}$ — the new least fixpoint

Variables
  SU — a set of supports
  $\Delta, \Delta_F$ — a set of atoms
  F, K — multisets of atoms

Initialization
  $T^{\omega}_P$ := $root(scT^{\omega}_P)$
  SU := dependent-supports(P, $T^{\omega}_P$, D)
  K := $[a \mid a \in_m scT^{\omega}_P$ and $a \notin heads(SU)]$
  $\Delta_F$ := $[\, a \in heads(SU) \mid a \in^x_m scT^{\omega}_P$ and $x > |\{t \in SU \mid head(t) = a\}|\,]$
  $\Delta$ := $root(\Delta_F)$
  F := K
```

```
begin
  while $\Delta \neq \emptyset$
  begin
    F := $F \uplus \Delta_F$
    $\Delta_F$ := $scT_{P \setminus D}(F, \Delta)$        {repetitions add to F}
    $\Delta$ := $root(\Delta_F) \setminus root(F)$     {forbid cyclic derivations}
  end

  return F
end
```

Proposition 182. $scT^{\omega}_{P \setminus D} = scT^{\omega}_{P \setminus D}(MK)$.

Proof. Analogous to the proof of Proposition 173. □

Proposition 183. *Algorithm 8.17 terminates and computes* $scT^{\omega}_{P \setminus D}$.

Proof. Δ is initialized correctly by the same argument as in the proof of Proposition 181. It is easy to see that for K after initialization it holds that root(K) is equal to the K from Notation 170, $K \subseteq_m scT^{\omega}_P$, and for all $a \in^x_m K$ also $a \in^x_m scT^{\omega}_P$. The rest follows from correctness of semi-naive support counting forward chaining (Proposition 131) and from Proposition 182. □

COMPLEXITY. Algorithms 8.16 and 8.17 function analogously to classical forward chaining. It is easy to see that the worst-case complexity of Algorithm 8.16 is the same as that of classical semi-naive forward chaining. Algorithm 8.17, in contrast to classical forward chaining, also counts supports. Counting supports, however, does not increase the worst-case complexity (see Section 7.5) and therefore the worst-case complexity of Algorithm 8.17 is also the same as the worst-case complexity of semi-naive forward chaining, i.e. polynomial in the number of terms occurring in base facts.

STRATIFIABLE NORMAL PROGRAMS. Both Algorithm 8.16 and Algorithm 8.17 can be easily extended to stratifiable normal programs the same way as fixpoint computation for stratifiable normal programs, see Section 7.10.

RULE UPDATES. Both algorithms in this section can handle base fact updates as well as general rule updates for the same reason that is explained in Section 8.4.1.

8.5.2 *Well-foundedness and support counts*

The reason maintenance algorithm (Algorithm 8.14) based on well-foundedness from Section 8.3 uses the auxiliary algorithm Algorithm 7.11 to decide well-foundedness of a support. Algorithm 7.13, which could be used instead of Algorithm 7.11, can be modified to leverage support counts, see Algorithm 8.18.

Algorithm 8.18: Deciding well-foundedness with support counts

```
Name
  is-well-founded-support-counts(s, D, SU, $scT^{\omega}_P$)
```

```
   Input
5    D — a subset of base facts in P (to remove)
     SU — the set of supports from $skT_P^\omega$ that depend on an atom
         from $heads(D)$
     s — a support in SU
8    $scT_P^\omega$ — fixpoint of $scT_P$ for a definite range restricted \
         datalog program P

   Output
11   true — if s is well-founded in $SG(skT_{P\setminus D}^\omega)$
     false — if s is not well-founded in $SG(skT_{P\setminus D}^\omega)$

14 Variables
     t — a support
     L — a list of nodes
17   well-founded(x) — a boolean for $x \in SU$, initially false for
         each $x \in SU$

   Initialization
20   L := topological-sort(s, SG(SU))

   begin
23   for i from 0 to $|L|$ such that $L[i] = t$ and $t \in SU$
     begin
       if $body(t) = \emptyset$ and $head(t) \in heads(D)$ then continue
26     if $\forall b \in body(t)$
           either $(\exists w \in SU)$  $head(w) = b$, well-founded(w), and
                                     $L[j] = w,\ j < i$
29              or $b \in_m^x scT_P^\omega$ and $x > |\{l \in SU \mid head(l) = b\}|$
         then well-founded(t) := true
     end
32
     return well-founded(s)
   end
```

Lemma 184. *Algorithm 8.18 terminates and outputs* true *if* s *is well-founded in* $SG(skT_{P\setminus D}^\omega)$ *and* false *otherwise.*

Proof. The algorithm terminates because it visits each sorted support exactly once, $SG(SU)$ is finite (P is a Datalog program and HB_P is finite by Observation 25), and Algorithm 7.12 terminates.

Algorithm 8.18 works analogically to Algorithm 7.13 with the difference that 1) it forbids derivations that include an atom from $heads(D)$, 2) only a subgraph of the whole support graph $SG(skT_P^\omega)$ is available. Point 1) assures that any found derivation of $head(s)$ is a subgraph of $SG(skT_{P\setminus D}^\omega)$. Point 2) is solved by applying Lemma 178 – i.e. support counts of an atom are used to determine whether it has a well-founded support in $skT_P^\omega \setminus SU$ (line 29).

Line 25 prevents any base supports that include a rule from D from being marked as well-founded. It can be shown by induction on i that therefore any support marked by the algorithm as well-founded has a derivation with respect to $P \setminus D$ and thus it is well-founded in $SG(skT_{P\setminus D}^\omega)$. □

RULE UPDATES. The algorithm works for removing base facts. The modification to removing general rules from P is a matter of forbidding supports using the rules to be removed in the search for a well-founded subgraph.

8.5.3 *Reason maintenance specialized to non-recursive Datalog programs*

Most of the complexity of reason maintenance algorithms stems from the fact that there can be directed cycles in a support graph for a general recursive program. If there are no cycles in a support graph then all supports are well-founded and the support graph can be examined fully incrementally in contrast to first computing the whole overestimation (i.e. the subgraph that depends on a rule to be removed). This section explores this idea and formulates a criterion, *safeness*, that can be used to determine acyclic subgraphs in the case of recursive programs thus enabling the use of the more efficient fully incremental algorithm for parts of a support graph for a recursive program.

Definition 185 (Rule dependency, rule dependency graph). *Let* P *be a definite range restricted program. A rule* $r_1 = H_1 \leftarrow A_1, \ldots, A_m \in P$ depends on *a rule* $r_2 = H_2 \leftarrow B_1, \ldots, B_n \in P$ *if there is an* $1 \leqslant i \leqslant m$ *such that* H_2 *and* A_i *are unifiable. We assume that* r_1 *and* r_2 *have no variables in common (otherwise we rename the variables of* r_2*).*

A rule dependency graph *for* P *is a directed graph* $G_P = (V, E)$*, where* $V = P$*, and* $E = \{(r_2, r_1) \mid \exists r_1, r_2 \in P, r_1$ *depends on* $r_2\}$*.*

Graphically, rules are represented by dots and edges by arrows.

The direction of edges in a rule dependency graph is in the "forward chaining direction", i.e. from a rule the head of which unifies with a body atom to the rule that includes the body atom. It is a trivial observation that any rule that is in a cycle in G_P can lead to a cycle in the support graph $G = SG(skT_P^\omega)$. Also, any rule on which a rule from a cycle depends can lead to a cycle in G. See the example in Figure 8.12. Rule 3 is in a cycle and it depends on rule 6. Therefore rule 6 may derive a fact which can be re-derived in the cycle.

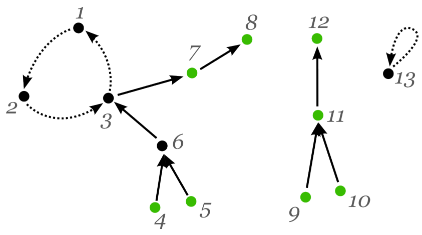


Figure 8.12: An example of a rule dependency graph that shows all cases possible with respect to cycles. Rules 4, 5, 7, 8, 9, 10, 11, 12 (denoted by a green dot) are safe rules. Rule 13 is in a cycle with itself (i.e. its head unifies with one of its body atoms). Edges in a cycle are dotted.

This observation can be used to define a heuristic useful to recognize some atoms that cannot be part of any cycle with respect to a given program.

Definition 186 (Safe rule, safe atom). *Let* P *be a definite range restricted program. A rule* $r_1 \in P$ *is safe in* P *if there is no rule* $r_2 \in P$ *such that* r_2 *depends on*

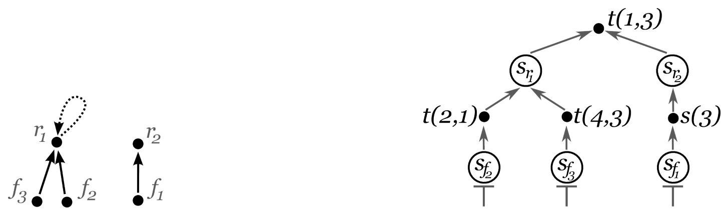
r_1 *and* r_2 *is in a cycle of a rule dependency graph* G_P. P *is safe if all its rules are safe in* P.

An atom is safe with respect to P *if it does not unify with any body atom of any rule in* P *that is not safe in* P.

It is easy to see that a program is safe if and only if its rule dependency graph is acyclic, i.e. the program is a non-recursive Datalog program. Note that a safe atom can be derived also by a rule that is not safe as the following example shows.

Example 187. *Consider the following program* P *and the corresponding rule dependency graph in Figure 8.13:*

$$r_1 = t(X, Y) \leftarrow t(4, Y), t(2, X)$$
$$r_2 = t(1, X) \leftarrow s(X)$$
$$f_1 = s(3) \leftarrow \top$$
$$f_2 = t(2, 1) \leftarrow \top$$
$$f_3 = t(4, 3) \leftarrow \top$$



(a) The rule dependency graph (b) The graph $SG(skT_P^\omega)$

Figure 8.13: Diagrams for the program P from Example 187.

Rule r_1 *is not safe because* $t(X, Y)$ *unifies for example with* $t(4, Y)$ *and thus it is in cycle in the rule dependency graph for* P. r_2 *is safe because it is not in a cycle and* $t(1, X)$ *unifies neither with* $t(4, Y)$, *nor with* $t(2, X)$. *Both rules however derive the atom* $t(1, 3)$ *which is safe because it unifies neither with* $t(4, Y)$ *nor with* $t(2, X)$ *– i.e. with no body atom of the only unsafe rule* r_1 *in* P.

Note that base facts f_2 *and* f_3 *and the corresponding atoms* $t(2, 1)$ *and* $t(4, 3)$ *are not safe because* r_1, *which is in a cycle, depends on them.*

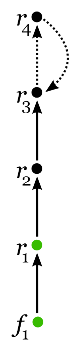
Note that an unsafe atom may depend on an atom that is safe. Consider the following example.

Example 188. *Consider the following program* P *and the corresponding rule dependency graph in Figure 8.14:*

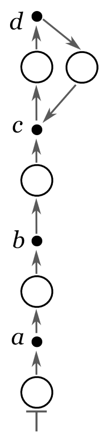
$$f_1 = a \leftarrow \top$$
$$r_1 = b \leftarrow a$$
$$r_2 = c \leftarrow b$$
$$r_3 = d \leftarrow c$$
$$r_4 = c \leftarrow d$$

Atom a *is safe in* P *because* r_1 *is safe and no body atom of any other rule unifies with* a. *All the other atoms* b, c, *and* d *are not safe in* P *and they depend on* a *in* $SG(skT_P^\omega)$.

It is important that atoms that directly depend on an atom safe in P have only well-founded supports in $SG(skT_P^\omega)$, see the following lemma



(a) The rule dependency graph for the program P. Rules f_1 and r_1 are safe in P. Rules r_2, r_3, and r_4 are not safe in P.



(b) The support graph $SG(skT_P^\omega)$ for the program P.

Figure 8.14: Graphs illustrating Example 188.

that justifies the definition of a safe atom and explains its meaning. It gives a simple syntactic criterion for recognizing when an atom cannot have not well-founded supports.

Lemma 189. *Let* P *be a definite range restricted program and let* $a \in T_P^\omega$ *be an atom safe with respect to* P. *Then all supports of any atom that directly depends on* a *in* $G = SG(skT_P^\omega)$ *are well-founded in* G.

Proof. Let G_P be the rule dependency graph for P. No rule in a cycle in G_P depends on a rule safe in P, by Definition 186. This means that any atom derived by a rule safe in P cannot be derived by a rule in a cycle in G_P (remember that, intuitively, "rules feed atoms to other rules" in direction of the directed edges in G_P). An atom is safe w.r.t. P iff it unifies only with body atoms of rules safe in P, by Definition 186. Together, any atom that directly depends on a safe atom in G is derived by a safe rule and cannot be derived by a rule in a cycle in G_P. Such an atom therefore does not depend on itself in G (if it depended on itself it would be derived by a rule in a cycle in G_P). Thus no support of such an atom depends on the atom and thus all its supports are well-founded in G by Lemma 180. □

It is a simple observation that a support graph corresponding to the least fixpoint of a safe program contains no cycles.

Lemma 190. *Let* P *be a safe definite range restricted program. Then there are no cycles in* $SG(skT_P^\omega)$.

Proof. All rules in P are safe in P. There is no cycle in the rule dependency graph for P because rules in a cycle are not safe. For a contradiction, let there be a cycle $a_1, s_1 = (r_1, \sigma_1), \ldots, a_{n-1}, s_{n-1} = (r_{n-1}, \sigma_{n-1}), a_1$ in $G = SG(skT_P^\omega)$. Then a body atom of r_1 is unifiable with a_1 (because $a_1 \in body(s_1)$) and $a_1 = head(s_{n-1})$ and therefore r_1 depends on r_{n-1}. Similarly, r_{i+1} depends on r_i for $1 \leqslant i \leqslant n-2$. Thus rules r_i create a cycle in the rule dependency graph for P which is a contradiction. □

A corollary of this lemma is that all supports in a support graph corresponding to a fixpoint of a safe program are well-founded (see Lemma 180).

The special structure of support graphs for safe programs allows for a specialized incremental reason maintenance algorithm that processes only atoms that are eventually removed or at most one step in the support graph further, see Algorithm 8.19. Recall that, in the general case, the other reason maintenance algorithms process all atoms in U while Algorithm 8.19 only processes atoms in O (see Notation 170).

In the initialization, Algorithm 8.19 first determines atoms that directly depend on a rule to remove (Δ_F is computed using the scT_D operator, notice the set D) and out of these it takes atoms (the set Δ) that lose all supports and therefore are not in the new fixpoint and it propagates their deletion. The while cycle works similarly, with the difference that any affected atom directly depends on an atom that lost all supports in the previous iteration (Δ_F is computed using the scT_P mapping and Δ, notice the set P).

Algorithm 8.19: Reason maintenance with support counts and no cycles.

```
Name
  reason-maintenance-support-counts-no-cycles(P, $scT_P^\omega$, D)

Input
  P — a safe range restricted definite \datalog program
  D — a subset of P, the set of rules to remove
  $scT_P^\omega$ — the least fixpoint of $scT_P$

Output
  $scT_{P\setminus D}^\omega$ — the new least fixpoint

Variables
  $\Delta$ — a set of atoms
  $F, \Delta_F$ — multisets of atoms

Initialization
  F := $\emptyset_m$
  $\Delta_F$ := $scT_D(scT_P^\omega)$      {atoms losing a support}
  $\Delta$ := $\{a \mid (\exists x \in \mathbb{N}_1) a \in_m^x scT_P^\omega$ and $a \in_m^x \Delta_F\}$
  $\Delta_\Sigma$ := $\Delta$                {atoms that lost all supports}

begin
  while $\Delta \neq \emptyset$
  begin
    F := $F \uplus \Delta_F$
    $\Delta_F$ := $scT_P(scT_P^\omega, \Delta)$ {atoms losing a support}
    $\Delta$ := $\{a \mid (\exists x \in \mathbb{N}_1) a \in_m^x scT_P^\omega$ and $a \in_m^x F \uplus \Delta_F\} \setminus \Delta_\Sigma$
    $\Delta_\Sigma$ := $\Delta_\Sigma \cup \Delta$        {atoms that lost all supports}
  end

  return $scT_P^\omega \ominus F$
end
```

Example 191. *Consider the following non-recursive program P and the corresponding support graph in Figure 8.15:*

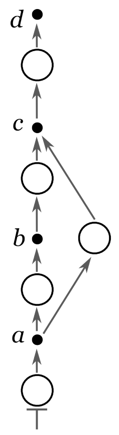


Figure 8.15: The support graph $SG(skT_P^\omega)$ for program P from Example 191. Fact $a \leftarrow \top$ is to be removed.

$a \leftarrow \top$

$b \leftarrow a$

$c \leftarrow a$

$c \leftarrow b$

$d \leftarrow c$

See the following table for a run of Algorithm 8.19 for the input P, $scT_P^\omega = [a, b, c, c, d]$, $D = \{a \leftarrow \top\}$.

F	Δ_F	Δ	Δ_Σ
$\emptyset_m$	$[a]$	$\{a\}$	$\{a\}$
$[a]$	$[b, c]$	$\{a, b\} \setminus \{a\} = \{b\}$	$\{a, b\}$
$[a, b, c]$	$[c]$	$\{a, b, c\} \setminus \{a, b\} = \{c\}$	$\{a, b, c\}$
$[a, b, c, c]$	$[d]$	$\{a, b, c, d\} \setminus \{a, b, c\} = \{d\}$	$\{a, b, c, d\}$
$[a, b, c, c, d]$	$\emptyset_m$	$\{a, b, c, d\} \setminus \{a, b, c, d\} = \emptyset$	$\{a, b, c, d\}$

In this case, the algorithm thus outputs the empty multiset: $scT_P^\omega \ominus F = [a, b, c, c, d] \ominus [a, b, c, c, d] = \emptyset_m$.

Proposition 192. *Algorithm 8.19 terminates and computes* $scT_{P\setminus D}^\omega$.

Proof. Let i denote the iteration number of the algorithm, where $i = 0$ is initialization, and let variables with index i denote value of the respective variable in step i (i.e. after i executions of the while body). We will prove by induction on i that no atom in Δ_i is in $T_{P\setminus D}^\omega$ and no support "generated" in (the multiset builder condition of) $scT_P(scT_P^\omega, \Delta_{i-1})$ is in $skT_{P\setminus D}^\omega$.

$i = 0$: $\Delta_F^0 = scT_D(scT_P^\omega)$ generates all supports that include a rule from D, none of these supports is in $skT_{P\setminus D}^\omega$. If $a \in \Delta_0$ then all supports of a are generated in Δ_F^0 and thus $a \notin T_{P\setminus D}^\omega$.

$i = 1$: $\Delta_F^1 = scT_P(scT_P^\omega, \Delta_0)$, i.e. each support generated in Δ_F^1 depends directly on an atom in Δ_0. No atom in Δ_0 is in $T_{P\setminus D}^\omega$, thus no support generated in Δ_F^1 is in $skT_{P\setminus D}^\omega$ by Lemma 117. If $a \in \Delta_1$ then $a \in_m^x scT_P^\omega$ and all x supports of a depend directly on an atom in Δ_0. Hence $a \notin T_{P\setminus D}^\omega$.

$i \rightsquigarrow i+1$: $\Delta_F^{i+1} = scT_P(scT_P^\omega, \Delta_i)$. By induction hypothesis, no atom in Δ_i is in $T_{P\setminus D}^\omega$. Therefore no support generated in $scT_P(scT_P^\omega, \Delta_i)$ is in $skT_{P\setminus D}^\omega$.

Let $a \in \Delta_i$. Then $a \in^x_m scT_P^\omega$ and all x supports of a depend directly on an atom in Δ_i none of which is in $T_{P\setminus D}^\omega$ as already proved. Hence $a \notin T_{P\setminus D}^\omega$.

Each support in $skT_P^\omega \setminus skT_{P\setminus D}^\omega$ is generated in a Δ_F^i because each such support depends on a rule from D. Also, $F = \biguplus_i \Delta_F^i$ and thus $scT_{P\setminus D}^\omega = scT_P^\omega \ominus F$, by Proposition 132.

The algorithm terminates because HB_P is finite (Observation 25) and there are no cycles in $SG(skT_P^\omega)$ by Lemma 190. $\square$

The concept of safeness is important of course in the case of recursive programs. A support graph of a recursive program may have large acyclic subgraphs which could be processed more efficiently than by Algorithm 8.15. A modified algorithm can take advantage of the fact that all atoms that directly depend on a safe atom have only well-founded supports (Lemma 189) and thus for atoms directly depending on a safe atom an incremental algorithm like Algorithm 8.19 could be applied instead of automatically deriving the whole set of "unsure" atoms. This idea is similar to the idea of local stratification [183, 231].

8.5.4 *Comparison with related work*

Support is a concept closely related to the concept of a justification known from the reason maintenance literature (see Section 8.10 for more about reason maintenance). To our best knowledge, counting supports has not yet been described in the literature. A closely related topic is counting derivations, which is treated in the next section in detail. Counting derivations leads to a so called duplicate (or multiset) semantics [159] for Datalog in which all atoms have a multiplicity corresponding to the number of their derivations. In this respect, support counting gives an alternative duplicate semantics which provides the same information about derivability and which can be computed more efficiently than numbers of derivations in the original duplicate semantics. Duplicate semantics enables certain counting approaches to incremental view maintenance [104] which are similar to the approach described in Section 8.5.3 ([104] provides no solution for general recursive Datalog programs – i.e. for Datalog programs in which the derivation count in the sense of [104] is infinite). Counting supports is inherently more efficient than counting derivations as it can be done by extending standard semi-naive forward chaining to keep count of generated rule instances per atom. In contrast, counting derivations is in general akin to generating all derivations. From this perspective, counting supports of an atom amounts to determining the in-degree of the atom node in the respective compact support graph while counting derivations amounts to generating all specific subtrees (derivations) of the support graph. See the next section for more details.

8.6 DERIVATION COUNTING

To solve the incremental maintenance problem it is necessary to be able to show whether there is a derivation of an atom or not (as indicated in the introduction to this chapter). This observation leads to the idea of counting derivations of each atom and updating the counts upon insertion and deletion. This section explores the idea and presents two different, in a way complementary, approaches to derivation counting and incremental reason maintenance based on derivation counts.

The methods of this section are important especially in the wiki context as they provide users with provenance information for each atom. Provenance information can be used for explanation and thus it can help to mitigate the problem that users feel lost even in wikis without complex features such as reasoning [173, 198].

8.6.1 *Reason maintenance with extended atoms*

Reason maintenance becomes a simple task in the case of *base fact* updates when a dcT_P fixpoint is available. Such a fixpoint provides information about derivations of any atom in the form of extended atoms (see Definition 133). Any atom derived from a base fact has a corresponding extended atom in the dcT_P fixpoint that has the base fact in its dependency. To compute the new fixpoint for a base fact removal, it is only necessary to remove all extended atoms from the fixpoint that have the base fact in its dependency. See Algorithm 8.20.

Algorithm 8.20: Reason maintenance for base fact updates using derivation counts

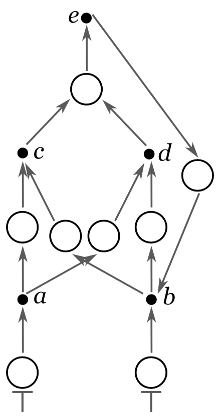
```
Name
  $dcT_P$-$dcT_P$-update(P, $dcT_P^\omega$, D)

Input
  P — a definite range restricted \datalog program
  $dcT_P^\omega$ — the least fixpoint of $dcT_P$
  D — a set of base facts from P to remove

Output
  $dcT_{P\setminus D}^\omega$ — the least fixpoint of $dcT_{P\setminus D}$

Variables
  Invalidated — a multiset of extended atoms

begin
  Invalidated := $[\,(a,S) \mid (a,S) \in_m dcT_P^\omega,\ (\exists d \in heads(D))\ d \in S\,] \cup_m$
                 $[\,(a,\emptyset) \mid a \in heads(D)\,]$

  return $dcT_P^\omega \ominus$ Invalidated
end
```

Example 193. *Consider the example in Figure 7.6. Let us remove the base fact* $a \leftarrow \top$ *from the fixpoint using Algorithm 8.20. Remember that the fixpoint of* dcT_P *is:*

$$dcT_P^\omega = [\,(a,\emptyset),(b,\emptyset),(c,\{a\}),(c,\{b\}),(d,\{a\}),(d,\{b\}),(e,\{c,d,a\}),\\ (e,\{c,d,b\}),(e,\{c,d,a,b\}),(e,\{c,d,a,b\}),(b,\{e,c,d,a\})\,].$$

Algorithm 8.20 computes:

$$\begin{aligned} \textit{Invalidated} = \quad & [\quad (c,\{a\}),(d,\{a\}),(e,\{c,d,a\}),(e,\{c,d,a,b\}),\\ & \quad\ (e,\{c,d,a,b\}),(b,\{e,c,d,a\})\ \ ] \cup_m \\ & [\quad (a,\emptyset)\ \ ] \end{aligned}$$



A reproduction of Figure 7.6 for convenience of the reader. The compact support graph induced by the fixpoint of skT_P, where $P = \{a \leftarrow \top; b \leftarrow \top; c \leftarrow a; d \leftarrow a; c \leftarrow b; d \leftarrow b; e \leftarrow c, d; b \leftarrow e\}$.

And thus it returns:

$$\mathrm{dcT}^{\omega}_{P\setminus\{a\leftarrow\top\}} = [\,(b,\emptyset),(c,\{b\}),(d,\{b\}),(e,\{c,d,b\})\,].$$

Proposition 194. *Algorithm 8.20 terminates and computes* $\mathrm{dcT}^{\omega}_{P\setminus D}$.

Proof. dHB_P is finite by Lemma 136. The fixpoint is also finite by Lemma 141. Algorithm 8.20 only removes a certain submultiset of the fixpoint from the fixpoint. Therefore it terminates.

After D is removed from P, all derivations of atoms in dcT^{ω}_P that include an atom from heads(D) are invalidated. Derivations correspond to dependencies of extended atoms by Proposition 142. Therefore the multiset $\mathit{Invalidated}$ represents exactly the invalidated derivations. Thus $\mathrm{dcT}^{\omega}_P \ominus \mathit{Invalidated} = \mathrm{dcT}^{\omega}_{P\setminus D}$ is the new fixpoint. □

Algorithm 8.20 does not count the number of derivations of an atom directly; the counting is implicit via multisets. It is easy to see that if the number of derivations of an atom represented in the multiset $\mathit{Invalidated}$ is the same as the number of derivations of the atom in $SG(\mathrm{skT}^{\omega}_P)$ then all derivations of the atom are invalidated by removal of D from P and it has zero derivations with respect to $P \setminus D$ and thus it is not in the new fixpoint.

While derivation counting keeps track of which atoms any atom depends on, it does not keep track of rules that are used to derive an atom. Reason maintenance for removing rules that are not base facts therefore requires either tracking rules as well or a more involved approach. The dcT_P operator and the current fixpoint can be used to determine all extended atoms that were derived using a rule to remove (let us call it the initial set of extended atoms). All derivations that use any atom from the initial multiset of extended atoms are thus invalidated. They can be computed by semi-naive forward-chaining using the initial multiset of extended atoms as the Δ (analogously to determining all dependent supports in Algorithm 7.6).

Note that the semi-naive forward chaining is necessary. It is not sufficient for example to look for extended atoms the dependencies of which are supersets of a dependency of one of the initial extended atoms. For the counterexample, look at Example 147 in Section 8.6 and consider removing rule $c \leftarrow a, b$. The initial multiset of extended atoms then is $\Delta_{\mathit{initial}} =$

$dcT_{\{c\leftarrow a,b\}}(dcT_P^\omega) = [(c,\{a,b\}),(c,\{a,b\})]$ representing derivations (c) and (e) in Figure 7.7 which are invalidated by the removal. Their dependencies are the same as the dependency of the extended atom $(c,\{a,b\})$ corresponding to derivation (d) in Figure 7.7 which remains valid. Therefore a simple search for dependency supersets would remove even the extended atom corresponding to the derivation that remains valid. In comparison, the Algorithm-7.6-like semi-naive computation $\Delta = dcT_P(dcT_P^\omega, \Delta_{initial}) = \emptyset_m$ shows that only the initial multiset $\Delta_{initial}$ is to be removed from the old fixpoint in this case.

See Algorithm 8.21 for the algorithm for reason maintenance for general rule updates.

Algorithm 8.21: Reason maintenance for general rule updates using derivation counts

```
Name
  $dcT_P$-$dcT_P$-update-rules(P, $dcT_P^\omega$, D)

Input
  P — a definite range restricted \datalog program
  $dcT_P^\omega$ — the least fixpoint of $dcT_P$
  D — a set rules from P to remove

Output
  $dcT_{P\setminus D}^\omega$ — the least fixpoint of $dcT_{P\setminus D}$

Variables
  F, $\Delta$ — a multisets of extended atoms

Initialization
  F := $\emptyset_m$; $\Delta$ := $dcT_D(dcT_P^\omega)$

begin
  while $\Delta \neq \emptyset_m$
  begin
    F := $F \uplus \Delta$
    $\Delta$ := $dcT_P(F,\Delta)$ {cycles are handled by the $dcT_P$ mapping}
  end

  return $dcT_P^\omega \ominus F$
end
```

Example 195. *Consider the example in Figure 7.6. Let us remove the rule* $d \leftarrow b$ *from the program and fixpoint using Algorithm 8.21. Remember that the fixpoint of* dcT_P *is:*

$$\begin{aligned} dcT_P^\omega = [\ &(a,\emptyset),(b,\emptyset),(c,\{a\}),(c,\{b\}),(d,\{a\}),(d,\{b\}),(e,\{c,d,a\}),\\ &(e,\{c,d,b\}),(e,\{c,d,a,b\}),(e,\{c,d,a,b\}),(b,\{e,c,d,a\})\ ]. \end{aligned}$$

Algorithm 8.21 computes:

F	Δ
$\emptyset_m$	$[(d,\{b\})]$
$[(d,\{b\})]$	$[(e,\{c,d,b\}),(e,\{c,d,a,b\})]$
$[(d,\{b\}),(e,\{c,d,b\}),(e,\{c,d,a,b\})]$	$\emptyset_m$

And thus it returns:

$$\begin{aligned} dcT^{\omega}_{P\setminus\{d\leftarrow b\}} = [\ & (a,\emptyset),(b,\emptyset),(c,\{a\}),(c,\{b\}),(d,\{a\}),(e,\{c,d,a\}) \\ & (e,\{c,d,a,b\}),(b,\{e,c,d,a\})\ ]. \end{aligned}$$

Proposition 196. *Algorithm 8.21 terminates and computes* $dcT^{\omega}_{P\setminus D}$.

Proof. There is only a finite number of extended atoms (Lemma 136), thus dcT^{ω}_{P} is finite, so is $dcT_D(dcT^{\omega}_{P})$, and cycles are prevented by the conditions $H\sigma \notin D_i$ and $H\sigma \neq B_i\sigma$ in the definition of the dcT_P mapping (Definition 144), and dcT_P is monotonic because it is continuous (Lemma 47, Lemma 139). Therefore Algorithm 8.21 terminates.

After initialization, Δ contains all extended atoms that directly depend on a rule from D in $SG(skT^{\omega}_{P})$. The while cycle then derives exactly those extended atoms from dcT^{ω}_{P} that depend on a rule from D by an argument similar to the one in the proof of Lemma 121 (correctness of the algorithm to compute supports dependent on a rule from a given subset of P). □

COMPLEXITY. Algorithm 8.20 identifies extended atoms that are not in the new fixpoint by directly accessing them using their dependencies and the set of base facts to remove. The time compexity of the algorithm is therefore linear in the number of atoms removed from the old fixpoint (assuming constant time access to extended atoms via their dependencies) and it is optimal because such atoms have to be removed by any reason maintenance algorithm.

The worst case space complexity of Algorithm 8.20 is at most exponential in the number of atoms in a fixpoint because dependencies are sets of atoms and for each atom there may be an indefinite number of extended atoms having different dependencies. However, not all sets of atoms are admissible dependencies; some sets of atoms do not correspond to a derivation. The number of dependencies of an atom is greatly affected by the structure of rules and data. The worst case occurs in the case that each atom has many different derivations which suggests big redundancies in rules. This case is quite unlikely in the wiki context where user rules can be assumed to be local, e.g. focused on data by a specific user or associated with a specific page.

The worst case time complexity of Algorithm 8.21 is apparently the same as that of semi-naive dcT_P forward chaining, i.e. $O(n^{2k})$.

It is not necessary to keep dependencies to count derivations to do reason maintenance, as the following section shows. Not keeping dependencies however means that a better space complexity of reason maintenance is traded for its bigger time complexity.

8.6.2 *Derivation counting without keeping dependencies*

The derivation counting immediate consequence operator dcT_P provides a way to count derivations in forward manner. It can be beneficial, especially for the purpose of explanation in combination with other reason maintenance methods, to count (and possibly enumerate) derivations for a specific atom on demand backwards. A backward derivation counting procedure then also leads to an alternative reason maintenance algorithm which first determines atoms that depend on an update and then recomputes numbers of derivations for each of them with respect to the updated program.

Numbers of derivations are not associated directly with atoms but rather with their individual supports in this method.

Definition 197 ($\text{deriv}_P\#(s)$)**.** *Let* P *be a definite range restricted Datalog program and let* $s = (r, \sigma)$ *be a support with respect to* P*. Then* $\text{deriv}_P\#(s)$ *is a variable that represents the number of derivations of* $\text{head}(s)$ *with respect to* P *that include* s*.*

Algorithm 8.22, which is central to this method, counts derivations of a given atom f which do not include any atom from the set of "forbidden" atoms F. Each well-founded support of an atom contributes at least one derivation to the atom (by the definition of a well-founded support, Definition 98). The total number of derivations of an atom is therefore the sum of derivations contributed by its supports. Derivations contributed by a support are computed by counting the number of derivations of each of its body atom. However, only such derivations of a body atom can be combined into a derivation of the head atom which do not use the head atom. Therefore the head atom becomes forbidden when searching for derivations of body atoms. The input parameter G is used to increase efficiency of the computation. $SG(G)$ is a subgraph of the whole compact graph $SG(skT_P^\omega)$ such that it contains all supports that depend on an atom from $F \cup \{f\}$. Derivation counts have to be computed only for atoms in $SG(G)$, see the proof of Proposition 200 for an explanation. The idea is that the set of atoms possibly affected by a removal (the set U of Notation 170) is likely to be small in comparison with the whole compact support graph. The reason maintenance algorithm based on derivation counting without keeping dependencies can first determine this set and use the corresponding support graph as the input G to Algorithm 8.22. Note that $SG(skT_P^\omega)$ is a *compact* support graph and thus Algorithm 8.22 does not distinguish between atoms and atom nodes for the sake of simplicity.

Algorithm 8.22: Compute the number of derivations of f which do not include any atom from F

```
Name
  count-derivations-without($P$, $skT_P^\omega$, $G$, $f$, $F$, $\text{deriv}_P\#()$)

Input
  $P$ — a definite range restricted \datalog program
  $S = skT_P^\omega$ — a set of supports
  $G$ — a set of supports, $G \subseteq skT_P^\omega$. $G$ must contain at least all
      supports that depend on any atom from $F \cup \{f\}$ in $SG(skT_P^\omega)$
  $f$ — an atom
  $F$ — a set of forbidden atoms
  $\text{deriv}_P\#()$ — an up-to-date number of derivations for
      each support in $S$

Output
  The number of derivations of $f$ that do not include any atom from
       $F$

Variables
  $count(g)$ — a number temporarily assigned to a node $g$, local to
      the procedure

begin
```

```
  for each support $s \in S$ such that $head(s) = f$
  begin
    if $body(s) \cap F \neq \emptyset$ then count(s) := 0; continue
       {the body of s contains a forbidden atom}

    for each $g \in body(s)$
      begin
        $count(g) := \sum_{t \in (S \setminus G), head(t)=g} deriv_P\#(t) + \sum_{t \in G, head(t)=g} \text{count-derivations-without}(P, S, G, g, F \cup \{f\}, deriv_P\#())$
      end
    count(s) := $\prod_{g \in body(s)} count(g)$ {assuming $\prod_{g \in \emptyset} count(g) = 1$}
  end
  return $\sum_{t \in G, head(t)=f} count(t) + \sum_{t \in (S \setminus G), head(t)=f} deriv_P\#(t)$
end
```

Lemma 198. *Let* P *be a definite range restricted program, let* $G = (S, F, E, l) = SG(skT_P^\omega)$ *be a support graph, and let* $a \in F$*. Then the number of derivations of* a *with respect to* P *that do not use any atom from a set of atoms* X *is* $deriv(f, X) = \sum_{t \in S, head(t)=f} \prod_{g \in body(t)} deriv(g, X \cup \{f\})$*.*

Proof. For an atom f and a set of atoms X, let $deriv_i(f, X)$ be the number of derivations of f in G with depth $\leqslant i$ that do not include any atom from X. Similarly, let $deriv_i(s, X)$ be the number of derivations of $head(s)$ in G that include s, do not include any atom from X, and are of depth $\leqslant i$.

We will show by induction on i that

$$\begin{aligned} deriv_i(f, X) &= \sum_{\substack{t \in S \\ head(t)=f}} deriv_i(t, X) \\ &= \sum_{\substack{t \in S \\ head(t)=f}} \prod_{g \in body(t)} deriv_{i-1}(g, X \cup \{f\}) \end{aligned}$$

$i = 1$: a derivation of f of depth 1 consists only of a support s of f ($head(s) = f$) with an empty body. Thus $deriv_1(s, X) = 1$ if $body(s) = \emptyset$ (an empty product is equal to 1) and $deriv_1(s, X) = 0$ if $body(s) \neq \emptyset$ because $deriv_0(s, X \cup \{f\})$ is trivially 0. Because there is at most one support of f with an empty body, it also trivially holds that $deriv_1(f, X) = \sum_{t \in S, head(t)=f} deriv_1(t, X)$, i.e. $deriv_1(f, X) = 1$ if there is a base support of f in G and $deriv_1(f, F) = 0$ otherwise.

$i \rightsquigarrow i + 1$: Let s be a support of f. s corresponds to derivations of f only if it is well-founded (Definition 98). Let D be a derivation of f of depth at most $i+1$ that includes s and does not use any atom from X. D consists of s and derivations of its body atoms are of length at most i. No such body atom derivation includes f (and trivially no atom from X) because otherwise D would not be a derivation, see Lemma 153 and its proof for a similar and more detailed argument and consider that if two body atom derivations share an atom a but with different supports then the resulting combined derivation includes two atom nodes labelled a and the resulting acyclic well-founded graph is still a derivation because it is homomorphically embedded in G. From the induction hypothesis, the number of such derivations of a body atom $g \in body(s)$ is $deriv_i(g, X \cup \{f\})$. Each combination of such derivations of body atoms can be combined into a derivation of f because they do not use f, see the proof of Lemma 153 for details. Therefore $deriv_{i+1}(s, X) = \prod_{g \in body(s)} deriv_i(g, X \cup \{f\})$. Each well-founded support of f corresponds to different derivations of f (they differ at least in the support). Therefore $deriv_{i+1}(f, X) = \sum_{\substack{t \in S \\ head t = f}} deriv_{i+1}(t, X)$.

For $i = \omega$ we get

$$deriv_P\#(f, X) = \sum_{\substack{t \in S \\ head(t) = f}} \prod_{g \in body(t)} deriv(g, X \cup \{f\})$$

□

Corollary 199. *Let* P *be a definite range restricted program, let* $G = (S, F, E, l)$ *be the support graph* $SG(skT_P^{\omega})$, *and let* $f \in F$. *Then* $f \in_m^{deriv_P\#(f,\emptyset)} dcT_P^{\omega}$.

Proof. Follows directly from Lemma 198 and from correctness of dcT_P, Proposition 142. □

The following proposition proves that Algorithm 8.22 is correct.

Proposition 200. *Algorithm 8.22 terminates and computes the number of derivations of* f *with respect to* P *that do not include any atom from* F.

Proof. The algorithm terminates because the size of the set of forbidden atoms increases with each recursive call, there is only a finite number of supports in S (HB_P is finite, Lemma 111), and cycles are avoided by adding the head of s to F.

First assume that $S = G$. In this case the algorithm computes exactly $deriv_P\#(f, F)$ which is correct by Lemma 198.

Let $G \subset S$. G contains all supports that depend on an atom from $F \cup \{f\}$ in $SG(skT_P^{\omega})$. Let $s \in (S \setminus G)$. We will show by contradiction that no derivation of $head(s)$ uses a forbidden atom. If a derivation of $head(s)$ includes a forbidden atom then s is a depends on that forbidden atom. By assumption of the algorithm, supports that depend on a forbidden atom are in G. Therefore s must also be in G which is a contradiction. Thus no derivation of $head(s)$ uses a forbidden atom. The number of derivations of $head(s)$ via s not using any forbidden atom is thus the same as the number of all derivations of $head(s)$ via s which is equal to $deriv_P\#(s)$, the number of derivations associated with s (assumed to be up-to-date by the algorithm). In other words, $deriv_P\#(s)$ does not have to be recomputed and can be used directly in this case. □

Note that if G is a proper subset of S then potentially a lot of computation is avoided by relying on derivation counts of supports in $S \setminus G$. It is also worth noting that Algorithm 8.22 shows how to enumerate all proofs of an atom based on a support graph – it explores all dependencies of the atom back to base facts.

Algorithm 8.23 is the main reason maintenance procedure of this method. It finds supports that depend on a support to be removed and runs Algorithm 8.22 on them. Supports and atoms are not actually removed by this algorithm; only their derivation counts are updated.

Algorithm 8.23: Recomputing derivation counts

```
Name
  reason-maintenance-derivation-count-update($P$,$skT_P^\omega$,$deriv_P\#()$,$D$)

Input
  $P$ — a definite range restricted \datalog program
  $S = skT_P^\omega$ — a set of supports
  $deriv_P\#()$ — up-to-date derivation counts of supports in $S$
  $D$ — a set of rules to remove

Output
  $deriv_{P\setminus D}\#(s)$ for each $s \in skT_P^\omega$

Variables
  $f$ — an atom
  $V$ — a set of supports
  $count(g)$ for a $g \in T_P^\omega$ — a number temporarily assigned to an
      atom $g$

Initialization
  $S_D$ := $\{(A \leftarrow \top, \emptyset) \mid A \leftarrow \top \in D\}$
  for each $s \in S_D$ $deriv_P\#(s)$ := 0
  $SU$ := dependent-supports($P$, $skT_P^\omega$, $D$)$\setminus S_D$
  $G$ := $SU$
  for each $s \in skT_P^\omega$ $deriv_{P\setminus D}\#(s) := deriv_P\#(s)$

begin
  for each $s \in SU$
  begin
    $deriv_{P\setminus D}\#(s)$ :=
       $\prod_{f \in body(s)}$ count-derivations-without($P$,$S$,$G$,$f$,$\{head(s)\}$,$deriv_P\#()$)
  end

  return $deriv_{P\setminus D}\#(s)$
end
```

Proposition 201. *Algorithm 8.23 terminates and computes* $deriv_{P\setminus D}\#()$ *such that*

$$skT^{\omega}_{P\setminus D} = skT_P^\omega \setminus \big\{s \in skT_P^\omega \mid (deriv_P\#(s) \neq 0 \text{ and } deriv_{P\setminus D}\#(s) = 0) \text{ or } (deriv_P\#(s) = 0 \text{ and } (\nexists t \in skT_P^\omega) head(s) = head(t), deriv_{P\setminus D}\#(t) > 0)\big\}.$$

Proof. Algorithm 8.23 terminates because all input sets are finite, the algorithm processes each support at most once, and Algorithm 8.22 terminates.

G contains all supports s that depend on a support to be removed (because algorithm dependent−supports is correct, Lemma 121) and thus its derivation count, $\text{deriv}_P\#(s)$, may change after removing S_D. For the same reason, derivation counts of supports in $skT_P^\omega \setminus G$ are not affected by the removal of S_D. Algorithm 8.22 recomputes derivation counts of any support in G. Algorithm 8.23 computes derivation counts of supports in SU correctly by Lemma 198 and Proposition 200.

Let $s \in skT_P^\omega$. If $\text{deriv}_{P\setminus D}\#(s) = 0$ and $\text{deriv}_P\#(s) \neq 0$ then all derivations of $\text{head}(s)$ that include s also include a support from S_D and are invalidated after S_D is removed.

If $\text{deriv}_P\#(s) = 0$ then s is not well-founded in $SG(skT_P^\omega)$ and remains in the new fixpoint as long as there is some support t of $\text{head}(s)$ such that $\text{deriv}_{P\setminus D}\#(t) > 0$ because then $\text{head}(s)$ is still derivable in the new fixpoint and s depends strongly on it in $SG(skT_P^\omega)$ by Lemma 180. That is, a support s such that $\text{deriv}_P\#(s) = 0$ and all supports of $\text{head}(s)$ have zero derivation count in the new fixpoint is not in the new fixpoint obviously either. $\square$

COMPLEXITY. Let us give a rough estimate of the time complexity of Algorithm 8.22. Let us assume the worst case, i.e. for the input it holds $G = skT_P^\omega$ and thus derivation counts have to be computed recursively for each support. Let us also assume that each atom in the fixpoint is derivable from any other atom in the fixpoint. The size of T_P^ω is bounded by $O(n^k)$ (see the complexity of forward chaining described in Section 7.2). This means that from the input atom f at most $O(n^k)$ atoms can be visited and because derivation counts are computed recursively for each of the visited atoms, the time to compute each such derivation count is also $O(n^k)$ (with a smaller constant because the number of forbidden atoms grows with each recursive descent). The number of edges traversed during computation of each such derivation also has to be taken into account. The number of edges in a support graph is $O(n^k)$ – any support has at most $k+1$ edges and k is a constant with respect to a given program. Thus the time complexity of Algorithm 8.22 is $O(n^{3k})$.

Algorithm 8.23 calls Algorithm 8.22 for each support in the worst case. The number of supports is $O(n^k)$ and thus the worst case time complexity of Algorithm 8.23 is $O(n^{4k})$. This means that Algorithm 8.23 can compute derivation counts for each atom in the fixpoint in time by factor n^{2k} bigger than the forward derivation counting Algorithm 7.10. This result represents a classical time-space tradeoff, where the better time complexity of Algorithm 7.10 is paid for by increased space complexity needed for keeping dependencies for each atom. In comparison, Algorithm 8.23 does not keep dependencies for each atom and thus has to perform more work to recompute them. The space required by Algorithm 8.23 is linear in the size of the support graph because only the support graph and one number per each support have to be kept. Note that the algorithm subtracts any processed support s from G and thus, in the next step, the execution of Algorithm 8.22 does not recompute the number of derivations corresponding to s again lowering the actual amount of computation needed. The advantage of not counting derivations of any support outside of G is likely to be substantial because in the usual case G, which corresponds to a set of consequences of rules to remove, can be assumed to be much smaller in size than the whole fixpoint if we assume that the algorithm is used for "small" updates most of the time.

The worst case analysis assumes a highly connected graph with large numbers of cycles which means a high number of redundant derivations caused by redundancies in rules. In a semantic wiki context, such a support graph is unlikely as rules are likely to be either a set of rules for example for RDF(S) semantics well crafted to limit redundancies or rules specified by users which are likely to be local to specific pages and users.

8.6.3 *Discussion*

Derivation and support counts provide similar information about derivability of atoms. An atom a is derivable with respect to a program P iff there is a *well-founded* support of a in $SG(skT_P^{\omega})$ (Lemma 152) and an atom a is derivable with respect to a program P iff there is at least one derivation of a with respect to P. This information provided support counts and derivation counts is not the same in the non-recursive Datalog case. A derivation count says how many ways there are to derive an atom. A support count expresses an upper bound on the number of well-founded supports of an atom, i.e. if one out of x supports of an atom is invalidated and $x > 1$ then it is not possible to say, using this information alone, whether or not the atom remains derivable. In contrast, derivation and support counts provide the same information about derivability in the case of non-recursive Datalog programs because in that case all supports are well-founded, i.e. if one out of x supports of an atom is invalidated and $x > 1$ then the it is clear that the atom remains derivable.

A support graph for a recursive Datalog program can contain not well-founded supports, i.e. supports that do not provide evidence of derivability. For this reason, a reason maintenance algorithm for non-recursive Datalog programs based on support counts cannot work in the ideally incremental way – exploring only those atoms which will eventually be removed and at most one step further as in the case of Algorithm 8.19. While derivation counts and support counts alone are not sufficient for an "ideally incremental" reason maintenance algorithm in the non-recursive Datalog case, they can still help to improve computation as in the case of Algorithm 8.17, where support counts are used to avoid the initial naive step which would otherwise be necessary.

The cost of counting derivations is higher than the cost of counting supports. It is inherently cheaper to count supports as it amounts to classical forward chaining with increasing a counter each time an atom is derived (an operation implicit in Algorithm 7.8 thanks to the use of multisets). In contrast, counting derivations is more expensive as each derivation of an atom has to be propagated to all atoms that depend on it.

Derivation counts and support counts provide different perspectives on derivability. Derivation count of an atom is a "global measure" of derivability, it is with respect to a whole support graph; it cannot be computed by examining only a local part of a support graph. In contrast, support counts provide a "local measure" of derivability; it depends only on the local structure of a support graph.

Another difference between the two kinds of counts can be important for explanation. A derivation count of an atom includes the number of *all* derivations, even those that may seem unintuitive or unexpected to casual users, see Example 202.

Example 202. *Let P be the following program:*

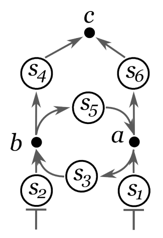


Figure 8.16: The support graph $SG(skT_P^\omega)$ for P in Example 202.

$r_1 = a \leftarrow \top$

$r_2 = b \leftarrow \top$

$r_3 = b \leftarrow a$

$r_4 = c \leftarrow b$

$r_5 = a \leftarrow b$

$r_6 = c \leftarrow a$

Consider the support graph $G = SG(skT_P^\omega)$ *in Figure 8.16. Atom* c *has two supports (*s_4 *and* s_6*) and four derivations in* G*. Following are the derivations of* c *in* G*:*

- $d_1 = s_1 a s_6 c$
- $d_2 = s_1 a s_3 b s_4 c$
- $d_3 = s_2 b s_4 c$
- $d_4 = s_2 b s_5 a s_6 c$

In this case, the longer derivations d_2 *and* d_4 *may seem to be "redundant" or perhaps even unexpected by casual users and thus the information that* c *has two well-founded supports may be sufficient and more intuitive. On the other hand, for example derivation* d_2 *provides information that* b *(and thus any atom that strongly depends on* b*) is derivable even if the base fact* r_2 *is removed. This kind of distinction may be useful in ranking derivations for the purpose of explanations.*

Note that this example is only a simple example to illustrate the problem; in a more complex support graph, the difference can be more significant.

Although only information about derivability of atoms is necessary for the purpose of solving the maintenance problem, computing the numbers of derivations, enumerating dependencies, or enumerating even whole derivations potentially provides additional valuable information to wiki users.

8.6.4 *Comparison with related work*

Derivation counting is referred to in the deductive database community. Duplicate semantics as defined in [158], uses a notion of derivation trees defined by Michael Maher in [138]. Maher's definition allows (see Definition 4.1 on page 11 and the proof of Theorem 6.1 on page 16 in [138]) for repetitions which is excluded in our definition of derivation (Definition 98) by the minimality condition. As a result, it is possible that an atom has an infinite number of Maher's derivation trees with respect to a Datalog program while the number of derivations is always finite for Datalog programs in our case. While derivation counts are always finite in the Datalog case, the problem, which essentially is detecting cycles in support graphs of recursive

Datalog programs, remains the same. This problem and related decidability problems are studied by Mumick and Shmueli in [159].

The derivation counting methods described in this section can also be seen as a novel extension of traditional reason maintenance in that they rely on derivation counts instead of sets of assumptions as the ATMS (assumption-based truth maintenance) systems [65], or they use and generate dependencies which are better suited for explanation than the more coarse-grained assumptions. Traditional reason maintenance systems of the ATMS kind, record only sets of base facts (assumptions) from which an atom can be derived which is not sufficient to count the number of derivations. They usually also are independent of the actual reasoner. In contrast, the approach presented in this section takes advantage of the additional knowledge about reasoning to speed up computation in cases where it can be justified for example by properties of a fixpoint (if a fixpoint contains an atom a then it contains also all atoms that can be derived using a) as exemplified by the parameter G in Algorithm 8.22. This dissertation also explicitly shows the correspondence between derivations, well-founded supports, and support graphs which is not discussed in literature (the reason may be that the focus in the reason maintenance context is solely on derivability).

As noted in Section 8.6.2, Algorithm 8.22 can also be used for proof enumeration which is a subject studied for example in [224] using an intuitionistic logic calculus.

Very recently, Nigam et al. [167, 168] developed an extension of [159] in the area of distributed (networked) Datalog view maintenance. Our extended immediate consequence operators, especially the scT_P operator, and the corresponding reason maintenance methods may provide insights for developing their distributed counterparts in the spirit of [167, 168] that shows (formally proves) especially how to deal with "out of order" computations that may occur in a realistic distributed setting.

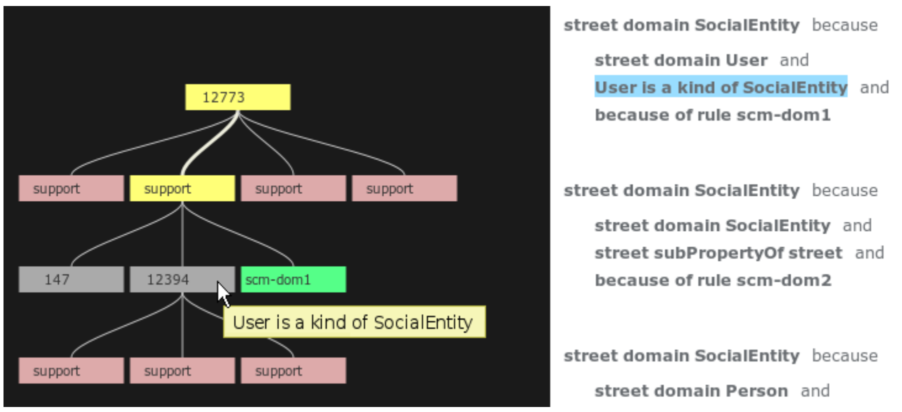
8.7 EXPLANATION

Information derived by skT_P, scT_P, and dcT_P operators, as well as information derived by the backward derivation counting algorithms can be used for explanation in semantic wikis. Even in traditional wikis users may feel lost [173, 198], and semantic wikis usually only increase the perceived complexity. Explanation is one of the ways to mitigate the added complexity by providing reasons for whatever happens in the wiki. In this section, the focus is on explaining derived information, although similar techniques can be used for other purposes too. One such example is general provenance tracking – tracking the origin and the full history of an information: where it comes from originally, how it was modified, when and by whom. Provenance has been studied in general [94, 202] and it has already been applied in fields such as scientific workflows [232, 67], databases [48], wikis [70], and with specific technologies such as RDF(S) [83, 88, 176, 221]. The concept of provenance is important because it is more general than that of a support or a derivation. For example, the provenance and explanation community even proposed a markup language for representing deduction processes – PML, the Proof Markup Language [151, 161].

Reasoning coupled with explanation can help to solve traditional problems of even classical wikis such as detecting inconsistencies. Wikis are well suited for work in progress which inevitably means that at some points of time they are likely to include contradictory information. For example automatic tagging using the kind of constructive reasoning with KWRL constructive and constraint rules as described in Chapter 6 may detect such contradictions. Explanation can then help pinpoint the root causes of such automatically detected inconsistencies because an inconsistency is only a special (reserved) kind of atom in KWRL and thus it can be explained as any other derived atom.

The methods described in this thesis provide a range of information that can be leveraged for explanation: from simple yes or no answers about derivability of an atom (all methods), via presenting direct supports (Section 7.3) or support counts (Section 7.5) together with information about their well-foundedness (Section 7.8), sets of atoms from which a given atom can be derived (i.e. dependencies of extended atoms corresponding to the atom, Section 8.6), to presenting whole derivations by walking a stored support graph (Sections 7.3 and 8.10) or by extracting them backwards on demand (Section 8.6.2). Moreover, different methods may be suitable and can be used for different parts of data, see the next section for more details.

An approach to explanation using stored support graphs and a backward rendering of derivation trees was explored in the KiWi project. In this implementation, a whole skT_P^ω fixpoint is stored in form of the compact support graph $SG(skT_P^\omega)$ in the open source graph database Neo4j[1] (see Chapter 9 for more details about the implementation). Parts of the support graph then can be rendered on user's request for explanation of a derived information. Figure 8.17 shows one such example rendering. The interactive graph, implemented in JavaScript, allows a user to interactively fold and unfold nodes upon which actions the user's browser dynamically loads required parts of the graph from the server. Note that rules are depicted as green nodes connected to supports and deviating slightly from the support graph diagrams introduced in this dissertation. Alongside the graph, there is also a textual representation of the related supports on the right side.



street domain SocialEntity because

street domain User and
User is a kind of SocialEntity and
because of rule scm-dom1

street domain SocialEntity because

street domain SocialEntity and
street subPropertyOf street and
because of rule scm-dom2

street domain SocialEntity because

street domain Person and

Figure 8.17: An example rendering of a part of a support graph using the explanation facility in the KiWi wiki. Atoms are represented by their database id numbers in the graph. Pointing at an atom node shows the corresponding statement as a tooltip and highlights it in the textual part of the explanation on the right hand side.

1 http://neo4j.org/

The textual representation of supports is done by a simple translation of RDF triples using a predefined vocabulary for properties. Such explanations can be conveniently displayed as tooltips for derived information in the wiki, see Figure 8.18. Note that, because the skT_P^ω fixpoint is precomputed and stored, such tooltips are available as soon as a page loads and thus users can get explanations immediately. Initial feedback shows that this feature is well appreciated by users. At occasions, the explanation facility helped the KiWi developers themselves in figuring out some unexpected derivation due to the RDF(S) or OWL 2 RL semantics. A user study, however, has not yet been carried out to analyse the advantages and disadvantages scientifically.

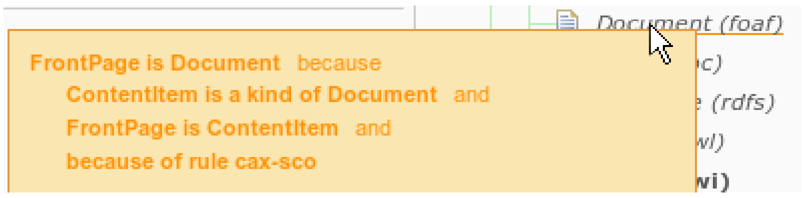

FrontPage is Document because
ContentItem is a kind of Document and
FrontPage is ContentItem and
because of rule cax-sco

Figure 8.18: An example of a support rendered as a tooltip in the KiWi wiki.

8.8 DIFFERENT ALGORITHMS FOR DIFFERENT PARTS OF DATA

One reason maintenance method may not fit all needs especially in a system containing as diverse kinds of information as a semantic wiki. For example, if a semantic wiki uses RDF to represent its data then basic RDF(S) reasoning may be ubiquitous and thus ought to be fast. Some parts of the data contained in the wiki may be used only internally by the wiki for example for navigation purposes. Explanation may not be necessary in such cases and a reasoning and reason maintenance method that does not use support graphs or dependencies may be used. Other parts of the data may be frequently viewed by users as it is the case for example with tags or categorical data. In such cases a better choice is reasoning with storing whole support (sub-)graphs or deriving dependencies.

Semantic wikis employ RDF(S) and OWL ontologies and often use many different ontologies to represent different kinds of data. As Weaver and Hendler showed [222], it is often easy to split RDF(S) data in parts for which the fixpoint can be computed in parallel. A similar strategy can be taken for reasoning and reason maintenance in a semantic wiki. For each independent part a specific reasoning and reason maintenance algorithm can be chosen according to the system and user needs. Such an approach is justified by Proposition 162 which shows that the same basic information is derived by all the methods from Chapter 7.

8.9 BATCH UPDATES AND LAZY EVALUATION

All previous sections consider the problem of incrementally processing one single update, this section explores the problem of efficient processing of sequences of updates. It may happen that a set of facts is removed and added again shortly after. In such a case, postponing updates and processing them in a batch may save computation. Note that this section studies how updates in a sequence of updates relate to each other and how they affect query evaluation rather than how to ensure consistent query results. That is, while the aim is not for a transaction concept, the results may be useful for designing a transactional system with materialization. See for example [156] for an in-depth discussion of transactions and deductive databases.

The problem of processing a sequence of updates is a very practical problem that any real world reasoner and reason maintenance system likely has to support, especially in the wiki context and in general in any application which interacts with users a lot. Large amounts of facts may be scheduled to be added and removed as a result of a few mouse clicks. A straightforward implementation may process updates as they come and let a user access whatever partially updated state that a database is currently in. A more sophisticated implementation will process whole sequences of updates and, in addition, inform a user which information may change after all updates are processed.

8.9.1 *Batch updates*

Notation 203 (Update)**.** *An update is either*

- *a simple adding update, denoted* $+A$*, where* A *is a set of rules, or*
- *a simple removing update, denoted* $-A$*,* A *is a set of rules, or*
- *a complex update, denoted* $\langle \delta_1, \dots, \delta_n \rangle$*, which is an ordered list of simple updates* $\delta_1, \dots, \delta_n$*.*

Set A *is called the update set. If* A *contains only base ground facts then we talk about a base fact update. If* A *contains rules that are not base facts we talk about a rule update. If all rules in* A *are definite we talk about a definite update. We say that two simple updates are disjoint if their update sets are disjoint.*

Definition 204 (Application of an update to a program, equivalent updates)**.** *Let* P *be a program and let* δ *be an update. Then application of* δ *to* P*, denoted* $P\delta$*, is defined as follows:*

$$\begin{array}{ll} P\delta = P \cup A & \text{if } \delta = +A, \\ P\delta = P \setminus A & \text{if } \delta = -A, \\ P\delta = ((\dots(P\delta_1)\dots\delta_{n-1})\delta_n) & \text{if } \delta = \langle \delta_1, \dots, \delta_n \rangle. \end{array}$$

Parentheses may be omitted for brevity.

Let δ *and* σ *be two updates.* δ *is equivalent to* σ *with respect to* P *if* $P\delta = P\sigma$*.*

Updating a program with a set of rules is simply making a set union or set subtraction. A trivial observation is that if a part of an update adds rules that already are in the program or removes rules that are not in the program then this part of the update is irrelevant to the result.

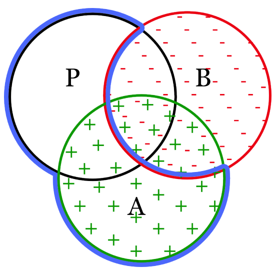
Lemma 205. *Let* P *and* A *be two programs. Then*

- $P + A = P + (A \setminus P)$*, and*
- $P - A = P - (A \cap P)$*.*

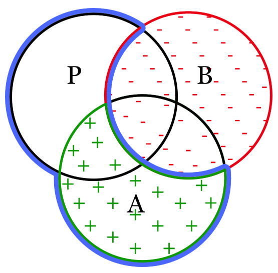
Proof. Follows immediately from Definition 204. □

Lemma 206. *Let* A, B, P *be sets. Then*

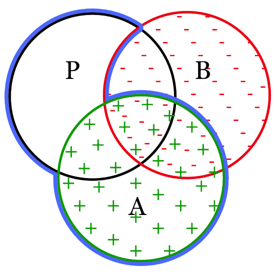
- $(P \cup A) \setminus B = (P \cup (A \setminus B)) \setminus B = (P \setminus B) \cup (A \setminus B)$*.*
- $(P \setminus B) \cup A = (P \setminus (B \setminus A)) \cup A = (P \cup A) \setminus (B \setminus A)$*.*



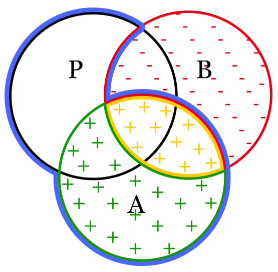
(a) $(P \cup A) \setminus B$



(b) $(P \cup (A \setminus B)) \setminus B$ and $(P \setminus B) \cup (A \setminus B)$



(c) $(P \setminus B) \cup A$



(d) $(P \setminus (B \setminus A)) \cup A$ and $(P \cup A) \setminus (B \setminus A)$

Figure 8.19: Venn diagrams illustrating the proof of Lemma 206. The resulting sets are marked in blue.

Proof. Point 1: Let $f \in (P \cup A) \setminus B$, see Figure 8.19a. Then $f \in P \cup A$ and $f \notin B$. Thus $f \in (P \cup (A \setminus B)) \setminus B$ and also $f \in (P \setminus B) \cup (A \setminus B)$, see Figure 8.19b. The other direction is similar.

Point 2: Let $f \in (P \setminus B) \cup A$, see Figure 8.19c. Then either ($f \notin A$ and $f \notin B$) or $f \in A$.

If f is neither in A nor in B then it remains in P on the right sides too. If $f \in A$ then $f \notin B \setminus A$ and therefore $f \in (P \setminus (B \setminus A)) \cup A$ and $f \in (P \cup A) \setminus (B \setminus A)$, see Figure 8.19d.

Similarly the other direction. □

Lemma 207. *Let* P, B, *and* A *be programs. Then*

- $P + B + A = P + (A \cup B)$,
- $P - B - A = P - (A \cup B)$,
- $P + A - B = P + (A \setminus B) - B = P - B + (A \setminus B)$,
- $P - B + A = P + A - (B \setminus A) = P - (B \setminus A) + A$.

Proof. Point 1 and 2 follow directly from Definition 204.

Points 3 and 4 follow directly from Lemma 206. □

Lemma 207 says that a sequence of two updates where one is an adding update and the other a removing update can be evaluated to two, possibly smaller, commuting updates. The resulting updates commute because they have an empty intersection.

Lemma 208. *Let* δ *be a finite complex update. Then there is an update* $+A$ *and an update* $-B$ *such that* $A \cap B = \emptyset$ *and* $P\delta = P - B + A = P + A - B$ *for any program* P.

Proof. Let P be a program. The lemma is trivially true because all simple updates are union and set difference operations. Thus, after all updates in the sequence are applied, the net result is that some elements are removed from P and some are added to P. Let us now present a constructive proof which shows in detail how to actually compute the updates.

$\delta = \langle \delta_1, \dots, \delta_n \rangle$, thus $P\delta = (((P\delta_1)\dots\delta_{n-1})\delta_n)$ by Definition 204. If all δ_i are pairwise disjoint then the order of update application is irrelevant, thus δ_i can be reordered so that first all adding updates are applied and then all removing updates are applied. Let δ^a be the set of adding updates in δ and δ^r the set of removing updates in δ. Then $P\delta = P - \bigcup \delta^r + \bigcup \delta^a$, by Lemma 207 points 1 and 2.

In the non-disjoint case, two consecutive adding or removing operations can be merged into one by Lemma 207 points 1 and 2. A sequence of updates $P + A - B + C$, where $B \cap C \neq \emptyset$, can be transformed to the sequence $P + A + C - (B \setminus C)$, by Lemma 207 point 4, which is equal to $P + (A \cup C) - (B \setminus C)$, by Lemma 207 point 1. After at most $n - 2$ such operations $P\delta$ can be transformed to $P\delta^+\delta^-$ and $P\delta = P\delta^+\delta^-$, where δ^+ is a simple adding operation and δ^- is a simple removing operation. If δ^+ and δ^- are not disjoint then one additional application of Lemma 207 results in two disjoint simple update operations to P. □

Notation 209. *Let δ be a complex update. Then let δ^+, resp. δ^-, denote the adding, resp. removing, update from Lemma 208.*

The notation is justified because δ and $\langle \delta^-, \delta^+ \rangle$ are equivalent updates with respect to any program.

Definition 210 (Application of an update to a fixpoint)**.** *Let P be a program and let δ be an update. Then application of δ to T_P^ω, denoted $T_P^\omega\delta$, is defined as $T_P^\omega\delta = T_{P\delta}^\omega$.*

Definition 210 formalizes the idea that applying an update to a fixpoint means updating the program and computing the corresponding new fixpoint.

A complex update can be transformed to an equivalent sequence of two updates which are disjoint, by Lemma 208. Disjointness of updates ensures that no unnecessary redundant computation is done when applying the updates to a fixpoint.

8.9.2 *Answering queries and lazy evaluation*

Processing an update may take considerable time during which an application is likely to run queries on behalf of users. Until processing of an update is completed, answers to some queries of the currently materialized fixpoint may be "unsure" in the sense that they may be not in the new fixpoint. This section explores this problem.

Let P be a definite range restricted program let δ be a complex definite range restricted update. Assume that T_P^ω is computed and a sequence of updates δ is pending, i.e. the fixpoint $T_{P\delta}^\omega$ is to be computed. By Lemma 208, there are two simple updates $+A$ and $-B$ such that $T_{P\delta}^\omega = T_{P+A-B}^\omega$. Because P and A are definite and T_P is monotonic, it holds that $T_P^\omega \subseteq T_{P+A}^\omega$ and thus an answer to a query of T_P^ω is an answer also in a new fixpoint resulting from an adding update. The more interesting case is the removing update.

Let $U(D)$ be the set of atoms in T_P^ω that depend on a rule from B in $G = SG(skT_P^\omega)$, and let $K(D) = T_P^\omega \setminus U$. Then any answer to a query that is in $U(D)$ is unsure, an atom in $U(D)$ may be removed after the update $-D$ is processed, while any answer that is in $K(D)$ is safe; it is a valid answer also after $-D$ is processed. It also means that processing removing updates can be postponed as long as queries can be answered using only atoms from $K(D)$.

An application that postpones updates should thus maintain the sets $U(D)$ and $K(D)$ as well. This maintenance amounts to marking and unmarking atoms that depend on a rule from D in G. This computation corresponds to the initialization step in reason maintenance by the FP−update algorithm (Algorithm 8.15) and thus is useful in any case.

This kind of lazy evaluation may be beneficial especially when $K(D)$ is big in comparison with $U(D)$ but $U(D)$ is still significantly big. When $U(D)$ is small then processing the $-D$ update is likely to be fast and there is less advantage to putting it off. This is however only a heuristic.

8.9.3 *Related Work*

This section studies practical problems around implementation of reasoning and reason maintenance in a real world system such as a wiki. It is essentially a problem of reasoning and materialization strategies and as such it is related to a variety of techniques from query modification to query cost analysis and materialization deferring. See for example [109] and [56] for an overview of such techniques.

8.10 RELATED WORK

The problem studied in this chapter is essentially a problem of changing knowledge and especially a problem of updating derived facts or beliefs upon changes in base facts or assumptions. As such, the problem is related to a wealth of literature ranging from epistemology, to logic, to databases. One of the overarching concepts is defeasible reasoning which includes two subfields important for this work: belief revision (an epistemological approach) and reason maintenance (a logical approach). The Stanford Encyclopedia of Philosophy describes defeasible reasoning as a reasoning where "the corresponding argument is rationally compelling but not deductively valid" [123]. Note that deductive validity is meant here as synonymous to logical validity[2] which is synonymous to logical truth. A more correct term is probably "soundness"[3]. See also Pollock's discussion [179] of the history and difference of defeasible and deductive reasoning. In this sense, reasoning described in Chapter 7 is defeasible because the reasoning procedure is rationally compelling but the reasoning is not deductively valid (resp., more correctly, it is not *sound*) because the underlying assumptions, base facts, can change. This section introduces the two defeasible reasoning subfields and gives an overview of related work from the field of reason maintenance. Related work from the field of deductive databases is covered in Section 8.4.2.

The terms belief revision and reason maintenance are sometimes treated as synonymous, sometimes one of them is taken as the more general con-

2 http://en.wikipedia.org/wiki/Validity
3 http://en.wikipedia.org/wiki/Soundness

cept. In fact, the fields are closely related and are compared in the literature for example by Jon Doyle [75], the founder of reason maintenance, and by Alvaro Val [214]. Belief revision, as an epistemological approach, studies properties of belief change and defeasible reasoning as a form of inference – a process by which knowledge is increased. In comparison, reason maintenance, as a logical approach, focuses on the consequence relation between sets of propositions.

Belief revision is developed as a formal theory by Alchourrón, Gärdenfors and Makinson [8, 9], often it is called the *AGM theory*. It tries to answer the question what to do in the case when an agent believes a set of propositions and a change (addition or retraction of a proposition) has to be carried out. If a proposition is added which is inconsistent with the original belief set then the agent has to revise the belief set by retracting some beliefs so that the belief set is consistent again. Logic itself does not provide any reason to prefer removal of one belief over another. Therefore it is necessary to rely on additional information about these propositions. The AGM theory states so called "rationality postulates" which characterize general properties that the revision should conform to. The main idea is that the revision should make a *minimal change* to accommodate the new information consistently. Note that for belief revision there is, by default, no difference between base facts and derived facts. This is in stark contrast to reason maintenance and deductive databases.

The term *reason maintenance* refers to a variety of knowledge base update techniques which share a common conceptual design – they distinguish between an inference engine and a separate reason maintenance system which communicate with each other via an interface [149]. The original term for these techniques was *truth maintenance* as Jon Doyle named it in his Master thesis and technical report at MIT [72]. Later, he suggested the name reason maintenance as a better alternative for "being more precise and less deceptive" [74]. Both terms are used in this section due to the fact that all abbreviations of the names of the algorithms still end with the TMS (truth maintenance system) suffix.

Reason maintenance was originally devised for use in problem solvers and therefore it puts stress on different aspects of handling updates than for example incremental view maintenance in deductive databases. It can also be seen as an implementation technique for belief revision [164]. Note that, in problem solving, a belief is a fact that can be derived from base facts (facts that cannot be refuted) and assumptions (facts that can be refuted). Classical reason maintenance addresses the following problems [143]:

- deriving new beliefs from old ones (the inference problem),
- methods of recording dependencies between beliefs (dep. recording),
- maintaining beliefs after a fact is disbelieved (disbelief propagation),
- changing beliefs in order to remove a contradiction (belief revision),
- dependency of beliefs on the absence of other beliefs (the nonmonotonicity problem).

Reason maintenance systems usually consist of two components: a reasoner and a reason maintenance component. The reasoner component addresses the inference problem, i.e. the derivation of new facts based on base facts, and notifies the reason maintenance system of new derivations and contradictions. The core responsibility of a reason maintenance system is to record dependencies between facts in order to be able to solve the disbelief propagation problem: to update the recorded dependencies when a fact was removed or added. Most authors of the original reason maintenance systems claim that this separation of responsibilities is advantageous, especially because of the increased modularity. However, it limits capabilities of the system. For example, the reason maintenance component cannot itself detect contradictions, it has to be notified of them.

Dependency records are usually called justifications and correspond to the notion of a support (Definition 55) in this dissertation. The original reason maintenance systems only invalidated justifications, they did not remove them in order to save computation in case a problem solver changes assumptions back. This was motivated by the fact that derivation steps could involve costly computations such as solving a differential equation.

Not all systems address all the listed problems. For example KiWi (version 1.0) does not address the nonmonotonicity problem and handles the belief revision problem only partially by supporting users in resolving inconsistencies.

Note that the disbelief propagation problem becomes a problem in fact only if derived facts are materialized (i.e. stored or cached). There is no such problem if derivation always happens at query time.

There is a simple solution to the disbelief propagation problem: computing a whole materialization anew after each update. If this obviously inefficient solution is left aside then one of the options is to keep records about derivations. In the reason maintenance jargon, the derivation record structure is called a data-dependency network.

8.10.1 *Monotonic JTMS reason maintenance*

This section presents an overview of the classical justification-based reason maintenance (JTMS) and related concepts. Note that the JTMS algorithm assumes that a data-dependency network is recorded and available. This corresponds to storing a compact support graph during extended forward chaining and, in fact, if a full support graph is stored then the algorithm in Listing 8.24 in this section (also in [73, 145, 96, 97, 165]) can (and perhaps should) be used for incremental view maintenance. In contrast, most of the work in this dissertation assumes that the full compact support graph is not stored. The memory cost of storing a support graph can be large. Let us now review the concept of a data-dependency network which is similar to but different than the concept of support graphs.

A data-dependency network (DDN) is a directed graph $G = (V, E)$, where the set of vertices V consists of two disjoint sets N, a set of nodes representing atomic facts and rules, and J, a set of justifications, i.e. records of the dependency of a fact on facts that were used in its derivation. Edges are always between a node and a justification: $E \subset (J \times N) \cup (N \times J)$. Justification with an outgoing edge to a node representing a fact is said to *support* the fact. A fact *participates* in a justification if there is an edge from the fact's node to the node of the justification. To distinguish DDNs from support graphs in diagrams, fact nodes are denoted with ovals and justification nodes with rectangles. A justification with no incoming edges is an empty justification that denotes a base fact (an assumption).

Note that most of the reason maintenance systems do not "understand" the content (atoms, literals) of nodes in a DDN. This is the reason why they, for example, cannot detect inconsistencies on their own.

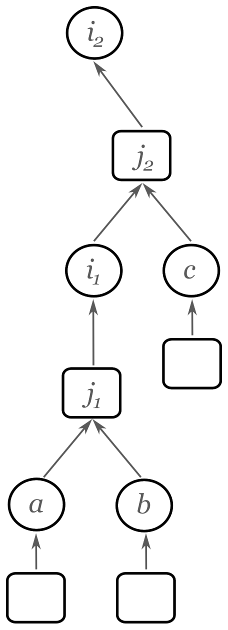


Figure 8.20: Data-dependency network.

It is also worth mentioning that there is also a dual representation of JTMS dependencies proposed by Dung [76, 77]. This alternative representation makes dependencies between *justifications* explicitly. It is equivalent to the DDN representation described here and it makes it possible to separate the monotonic and non-monotonic parts of the dependency graph in the case of the non-monotonic JTMS.

Let us now examine a specific DDN, consider the following example.

Example 211. *Consider a set of atoms* $F = \{a, b, c, i_1, i_2\}$, *where*

- $a =$ (*kiwi:StructuredTag rdfs:subClassOf kiwi:Tag*),
- $b =$ (*kiwi:Tag rdfs:subClassOf kiwi:ContentItem*),
- $c =$ (*kiwi:ContentItem rdfs:subClassOf kiwi:Document*),
- $i_1 =$ (*kiwi:StructuredTag rdfs:subClassOf kiwi:ContentItem*),
- $i_2 =$ (*kiwi:StructuredTag rdfs:subClassOf kiwi:Document*),

and a rule $r =$
(*$x rdfs:subClassOf $z*) ← (*$x rdfs:subClassOf $y*), (*$y rdfs:subClassOf $z*).
Atoms a, b, c *are base facts, atoms* i_1 *and* i_2 *are derived using the subClassOf transitivity rule* r, *see Figure 8.20 for the corresponding data-dependency network. Suppose that fact* a *should be removed.*

The idea of the JTMS algorithm is based on an idea similar to the overestimation and rederivation phases of incremental view maintenance algorithms in deductive databases. The basic idea leads to the following (incorrect) sketch of the monotonic JTMS algorithm:

1. Determine all justifications J in which a removed fact participates.
2. For each justification in J invalidate the justification and add the fact that the justification supports to F.
3. For each fact in F check if it still has a justification. If yes do nothing. If not then remove the fact recursively.

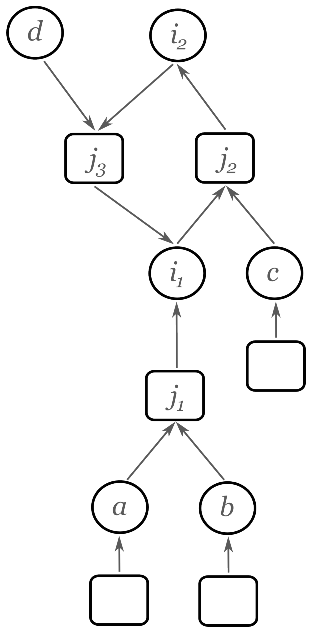


Figure 8.21: Data-dependency network with a cycle.

The basic algorithm does not handle cycles correctly. Let us assume that another base fact d = (kiwi:Document rdfs:subClassOf kiwi:ContentItem) is asserted. This leads to a new derivation of i_1, see the resulting data-dependency network in Figure 8.21.

To illustrate the problem, let us again assume that the node a is to be removed. This means that justification j_1 is removed in step 2 of the basic algorithm and node i_1 is checked for an alternative justification in step 3. The alternative justification exists; it is justification j_3. Therefore the algorithm stops although node i_2 is not derivable from the current set of base facts ($\{b, c, d\}$). In reason maintenance parlance, justification j_3 is not *well-founded* – it supports node i_1 on which it also depends via justification j_2. In other words, the subtree rooted in j_3 does not represent a derivation of i_1 because it uses i_1.

A more sophisticated algorithm [72, 73, 145, 96, 97, 165] that uses the idea of *current support* can handle cycles correctly (the description here is based mainly on the nice description of Nebel [165]). The idea is that for each derived fact node one justification is marked as current and is guaranteed to be well-founded. Let us assume there is a DDN in which all current supports are well-founded and the current support of a fact node f is to be removed. See Listing 8.24 for the algorithm to remove a justification and to maintain the set of current well-founded supports.

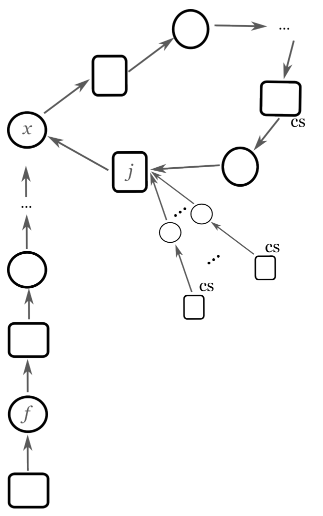
Algorithm 8.24: Current support algorithm

```
Name
  current-support-algorithm-mark(f, G)

Input
  f — a fact node
  $G = (N \cup J, E)$ — a data-dependency network

Output
```

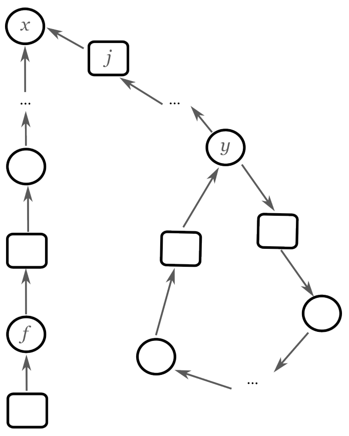
```
  marked() — a boolean variable for each node of G

Variables
  j — a justification
  x — a fact node

begin
  marked(f) := true
  for each j that f participates in
  begin
    marked(j) := true
    if j is the current support of x then
      current-support-algorithm-mark(x, G)
  end

  return marked()
end
```

```
Name
  current-support-algorithm-remove(f, G)

Input
  f — a fact node
  $G = (N \cup J, E)$ — a data-dependency network

Output
  updated G

Variables
  x — a fact node

begin
  current-support-algorithm-mark(f, G)

  for each $x \in N$ such that $x \neq f$ and marked(x) == true
  begin
    if $\exists j \in J$ such that marked(j) == false and j justifies x then
    begin
      make j the current support of x
      recursively-unmark(x)
    end
  end

  invalidate all nodes that remained marked

  return the accordingly updated G
```

To remove a fact f, first all facts and justifications that depend on f via current supports are marked. Then each marked fact is checked whether it has some unmarked justification. If yes then the justification is made the current support of the fact. The fact being checked and facts and justifications that depend on it are unmarked. Finally, all nodes that remain marked are invalidated.



(a) x depends on itself.



(b) x depends on y that depends on itself.

Figure 8.22: Illustrative examples of simplified data-dependency networks.

The following argument is, to the best of our knowledge, not available in the literature. It is given here for the sake of completeness and for the reader's convenience.

It may not be apparent why an unmarked justification of a marked fact is well-founded, i.e. why line 47 of the algorithm is correct. Let us assume that it is not the case – that there is a not well-founded *unmarked* justification j of a marked fact x. j being not well-founded means that (a) it supports x while it also depends on x or (b) it depends on a fact y that depends on itself, by definition of well-foundedness.

- If j depends on x, see Figure 8.22a, each fact that participates in j has a current support. If none of these current supports depend on x then j is well-founded. Therefore one of the current supports must depend on x. By repeating this argument there has to be a path via current supports from x to j. This is a contradiction because in this case the whole path would have been marked because x was marked.
- If j depends on y that depends on itself, see Figure 8.22b, there are three possibilities for y:
    1. y has only one justification that depends on y. This is in contradiction with the assumption that each fact has a current well-founded support.
    2. y has a current support j_y that depends on f. Then it would have been marked and so would j (by the argument from the previous point).
    3. y has a current support j_y that does not depend on f. Then it must depend on other base facts which means that y has a well-founded support and therefore j is well-founded too which is a contradiction.

When a DDN is being built, the first justification of a node is made its current support. It is easy to see that such a justification derived by forward chaining is well-founded.

The current support algorithm is more expensive than the basic one. In the basic algorithm, only those vertices are visited which are eventually deleted. The current support algorithm generally marks more nodes (overestimates) than the number of nodes that are going to be deleted and visits them twice – first to mark them and then to delete them. Note that the first algorithm is essentially an algorithm for non-recursive Datalog programs while the second algorithm works correctly in the recursive Datalog case too and thus the efficiency difference is not surprising. The current support algorithm requires the whole DDN (resp. support graph) to be stored, i.e. it requires more space than algorithms in Section 8.4 (T_P reason maintenance without support graphs), but potentially less space than the dependency keeping algorithm from Section 8.6.1 (dcT_P reason maintenance with keeping dependencies). With respect to time complexity, the current support algorithm is in general more efficient than the algorithms of Section 8.4 because it has more information (the support graph) available, hence a classical time-space tradeoff. See Chapter 10 for a discussion of a possible graph-based modification of the reason maintenance algorithms without support graphs of Section 8.4.

Another way to cope with the well-foundedness problem is to ensure that no cyclic justifications are created. This approach has been applied for example in the ITMS reason maintenance system [162]. In fact, current forward chaining reasoners for the semantic web [116, 213] aim to increase speed by eliminating duplicate inferences which of course includes cyclic inferences. If only well-founded justifications were found and stored the basic algorithm could be used. A classical semi-naive forward chaining reasoner cannot however easily ensure that only well-founded justifications are generated. To ensure well-foundedness, either dependencies have to be stored or it has to be proved that a given program has the property that no not well-founded justifications can be generated by forward chaining – this is trivially true in the case of non-recursive Datalog programs.

The monotonic JTMS [149] approach has been implemented by Broekstra et al. [39] in the area of semantic web, although only in a limited way. Their reasoning and reason maintenance system is specific to RDF(S) reasoning. In particular, it is limited to a fixed Datalog axiomatization of RDF(S) semantics which uses at most two body atoms per rule. This limitation allows them to hand-optimize semi-naive forward chaining to avoid redundant derivations by analysing dependencies between rules. It was implemented as part of the Sesame triple store [38].

8.10.2 *Other reason maintenance methods*

Monotonic JTMS reason maintenance is a simplification of the original JTMS reason maintenance system which is non-monotonic. Other significant reason maintenance systems are ATMS, LTMS, HTMS, and ITMS, see Table 8.23. Refer for example to [142, 85, 206] for an introduction and comparison of some of them.

LTMS, the Logical Truth Maintenance System, was first developed by Mc Allester in [147, 148]. LTMS is more powerful than JTMS because it can represent general formulas as justifications and it can also represent negated nodes. On the other hand it also means that its implementation requires a more sophisticated algorithm.

Nonmon. JTMS	The original non-monotonic, justification-based TMS.
Mon. JTMS	Monotonic justification-based TMS. A simplification of the original non-monotonic JTMS.
LTMS	Logic TMS – a JTMS-inspired TMS modified to to accept arbitrary clauses.
ATMS	Assumption based TMS – de Kleer's TMS that tracks sets of assumptions for all derived facts.
HTMS	Hybrid TMS – combines advantages of JTMS, LTMS, and ATMS. By de Kleer.
ITMS	Incremental TMS – improves assumption switch performance of LTMS.

Table 8.23: An overview of different reason maintenance approaches.

ATMS [65, 84, 66], the Assumption based TMS, was developed in order to support fast switching of sets of assumptions which was required by certain problem solvers. Many problem-solving tasks require the inference engine to rapidly switch among contexts (sets of assumptions) or to work in multiple contexts at once. Two examples of such tasks are qualitative reasoning and diagnosis [85]. Switching context in JTMS and LTMS requires relabelling nodes which can be costly and it is the main reason for developing and using ATMS. ATMS is similar to monotonic JTMS but it avoids such relabelling by tracking sets of assumptions for all derived facts. The price is exponential increase in memory consumption.

ITMS [162], the Incremental Truth Maintenance System, was designed for real-time use in embedded devices. Therefore ITMS includes an optimization to perform faster context switches than the LTMS. ITMS was used for example in NASA's project called Livingstone [225] which was a real-time execution kernel that performs a variety of functions such as commanding, monitoring, diagnosis, recovery and safe shutdown automatically with required response times on the order of hundreds of milliseconds.

The ITMS, works only based on approximations which means that it does not always perform optimally. This problem was adressed by Guofu Wu and George Coghill in their Propositional Root Antecedent ITMS [227]. Their method is based on the concept of *root antecedents*, which are assumptions that help to determine the dependency relationship between a rules accurately. Their algorithm however still requires that the data-dependency network is stored.

The use of reason maintenance in systems with limited memory led to the development of a "Fact-Garbage-Collecting" technique [81] to remove facts and justifications that are unlikely to be used in the future again. The algorithm adds an additional step to the normal LTMS retraction algorithm which is the garbage collection.

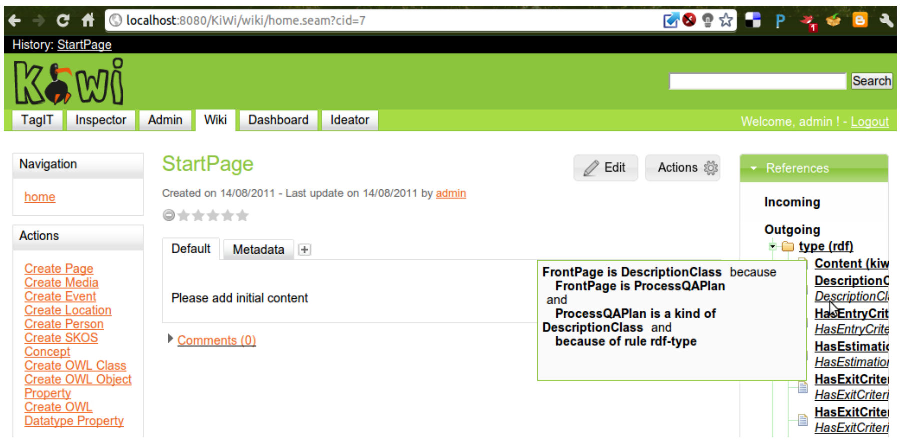
Each of the approaches has its own problem domain where it performs the best. An exception is perhaps the HTMS which combines features of the JTMS and ATMS and behaves as the former or the latter depending on its usage.

Note that, unlike belief revision, reason maintenance is more pragmatic; it started with practical implementations and formal description and theory came only later. Detlef Fehrer created a unifying logical framework [82] which describes all the various reason maintenance algorithms using one logical framework based on Gabbay's Labelled Deductive Systems [89].

9 IMPLEMENTATION

Many real-world semantic web applications process millions and even billions [116, 213, 14] of RDF triples. This makes scalability important and a pure in-memory implementation of triple stores impractical.

Data intensive applications have a long tradition of using relational database management systems as persistent storage. This chapter compares the traditional approach with a graph database approach [11, 192] to implementing reasoning and reason maintenance methods that make use of graph traversals. A hybrid approach that uses a relational database for reasoning and a graph database for reason maintenance and explanation is implemented as part of the KiWi 1.0 system.[1]



FrontPage is DescriptionClass because
FrontPage is ProcessQAPlan
and
ProcessQAPlan is a kind of
DescriptionClass and
because of rule rdf-type

Figure 9.1: A screenshot of the KiWi 1.0 wiki after installation and after assigning the Process QA Plan type from the Logica ontology to the start page. Immediately after the type was assigned, the forward chaining reasoner derived consequences and stored supports which are then shown for example in the form of the tooltip explaining why the Front Page is also of type Description Class.

9.1 A RELATIONAL DATABASE REPRESENTATION

Graph structures play an important role in reasoning and reason maintenance. Yet many current data management systems that provide a reasoning and reason maintenance functionality, such as triple stores, rely mostly on traditional relational database systems.

Processing graph structured data in a relational database system is not straightforward because of the paradigm mismatch between tables and graphs. To represent a graph in a relational database it has to be described in terms of tables. This in itself presents no obstacle but it has repercussions on the efficiency with which data can later be handled due to the need for joins.

1 http://kiwi-project.eu/

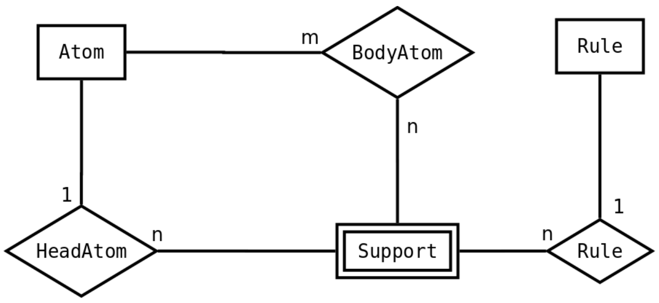


Figure 9.2: An entity-relationship diagram for a support representation.

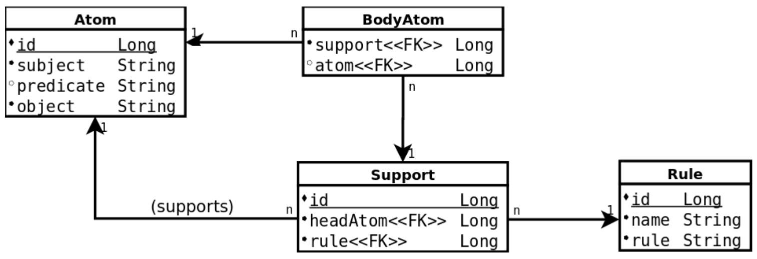


Figure 9.3: A relational database schema modelling atoms, supports and their relationships. ≪FK≫ denotes a foreign key.

Let us quickly review how a graph structure can be represented in a relational database. Two possible modelling approaches can be distinguished. A general one that admits modelling a generic graph structure which covers a wide range of specific problems and a problem-specific representation. Let us explore the latter approach as it allows for a clearer comparison to a graph database representation.

The support (r, σ) named s can be seen as the triple $(r, \mathrm{body}(s), \mathrm{head}(s))$ where rule r together with atoms $\mathrm{body}(s)$ can be used to derive the atom $\mathrm{head}(s)$. Three distinct entities can be distinguished: atoms, rule names, and supports. Figure 9.2 shows how the entities and their relationships can be modelled.

The entity-relationship model in Figure 9.2 can be represented using four tables: three for the three entities and one for the $\mathit{BodyAtom}$ relation. See Figure 9.3 for the corresponding relational database schema. Alternatively, one could use a single relation. Such an approach has the classical disadvantage of denormalized databases: more complicated and thus perhaps less efficient updates. This approach is not further investigated here as updates are as important as reads in our use-case.

One operation that is elementary in most of the algorithms of previous chapters is determining which atoms depend directly on a given atom. Let us express the operation as an SQL query assuming the relational model in Figure 9.3. See Listing 9.3.

Listing 9.3: An SQL query to find atoms depending directly on a given atom with id ATOMID

```sql
SELECT adep.id, adep.subject, adep.predicate, adep.object
FROM Atom adep, Support s, BodyAtom ba
WHERE ba.atom = ATOMID AND ba.support = s.id AND s.headAtom =
    adep.id;
```

As the query in Listing 9.3 shows, three tables have to be joined in order to determine the set of atoms that directly depend on a given atom. Although it is possible to use indexes to improve the efficiency of join computation, the operation still is rather expensive especially when a whole graph traversal needs to be performed – then each traversal of an additional edge adds more joins to the query. Let us first consider whether it is possible to reduce the number of joins.

9.1.1 *Reducing the number of joins*

Broekstra and Kampman [39] limit the number of body atoms per rule to two in the implementation of reasoning in the triple store Sesame [38] as they are interested only in RDF(S) reasoning. In our case this would mean that the `BodyAtom` relation could be included directly in the `Support` table removing one join per edge traversal. Putting a limit on the number of body atoms is however not desirable for a general-purpose rule language with user defined rules such as KWRL.

It is conceivable, that the `BodyAtom` relation could be encoded directly in the `Support` table in the general case too. The problem is to represent a variable number of body atoms using a fixed number of columns if we do not want to put an upper limit on the number of body atoms in a rule body. Each body atom is identified by a unique id in the `Atom` table. The body of a support can be encoded as a single number using these ids. It is necessary that the encoding is easily invertible and also that it is simple so that it can possibly be computed by the database itself (e.g. as a stored procedure). Given such an encoding, supports can be represented using a single table.

One candidate encoding is a generalization of the Cantor's pairing function. The Cantor pairing function is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$, i.e. it uniquely and invertibly maps pairs of natural numbers to individual natural numbers. It is also easily computable:

$$c(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$$

The Cantor pairing function can be generalized to a bijection $\mathbb{N}^n \to \mathbb{N}$ for example as follows

$$c_n(x_1, x_2, ..., x_n) = c_2(c_{n-1}(x_1, x_2, ..., x_{n-1}), x_n)$$

where $c_2(x, y)$ is the original Cantor pairing function $c(x, y)$. Other recursive schemes[2] are also possible and some of them produce smaller numbers (in the sense that small inputs lead to a small result).

The number of body atoms per rule can vary and it can be encoded as the last number in the sequence of body atom ids in order to make decoding possible. The first decoding step then retrieves the number of body atoms that need to be decoded. Computing the Cantor's pairing function as well as its inverse is simple (the inverse requires computing a square root) and can be implemented as a stored procedure in many current relational database systems. However, the function grows rapidly with the number and with the size of its arguments. This poses a major disadvantage when the number of stored atoms can easily be in the millions or even billions making the input atom ids potentially very large. It is unlikely that other pairing functions

2 See for example http://mathworld.wolfram.com/PairingFunction.html

would mitigate the problem significantly as the Cantor pairing function is bijective and in this sense it is "compact."

It may be useful to use the above described scheme to reduce the number of joins in some cases but the number of joins will still grow linearly with the number of edge traversals. Let us now review a not relational database approach designed with graph traversals in mind.

9.2 A GRAPH DATABASE REPRESENTATION

Recently, many unconventional new approaches to data management have appeared and are usually referred to as "NoSQL."[3] NoSQL stands either for "no SQL" to contrast the approach to traditional relational databases or for "not only SQL" to acknowledge the importance of relational databases and to stress NoSQL as an orthogonal approach. Indeed, there is evidence [152] that SQL and NoSQL may be just dual approaches (note that the analysis of [152] does not include graph databases). Following is the definition of NoSQL from `http://nosql-database.org/` which currently is the main NoSQL portal (footnotes were added):

> Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontal scalable. The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: schema-free, easy replication support, simple API[4], eventually consistent / BASE[5] (not ACID[6]), a huge data amount, and more.

The main motivation behind NoSQL databases is the insufficiency of relational databases in some use cases that handle large amounts of data very rapidly. Examples include the short message social network Twitter[7] and Facebook[8] which are willing to trade the `ACID` [100] properties for the different `BASE` [35, 93, 40] guarantees that mean a higher data throughput or smaller latency. Brewer's CAP theorem [35], formally proved by Gilbert and Lynch in [93], says that an application can achieve only two of the following three properties: Consistency, Availability, Partition tolerance. `ACID` drops partition tolerance while `BASE` drops consistency. The acronym `BASE` has been chosen by Brewer and his students to indicate the opposite of `ACID` [40]; "base" is synonymous to "alkali."

A graph database is a kind of NoSQL database that is optimized for handling graph structured data. The term is used loosely both for systems that use native data structures optimized for storing and traversing graphs and for systems that are built on top of relational database systems. This section is based predominantly on properties of the Neo4j[9] system – a prominent graph database that uses a native data storage designed for fast graph traversals [189] and that has been[10] in commercial development and deployment since 2003.

3 `http://nosql-database.org/`
4 Application Programming Interface
5 Basically Available, Soft state, Eventual consistency
6 Atomicity, Consistency, Isolation, Durability
7 `http://twitter.com/`
8 `http://facebook.com/`
9 `http://neo4j.org/`
10 `http://neotechnology.com/about-us`

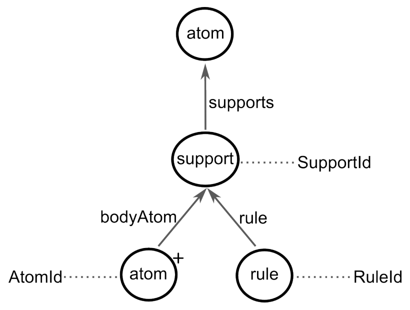


Figure 9.4: A graph database "schema" for support graphs. There are three different node types: `atom`, `rule`, and `support`. There are two atom nodes in this diagram for clarity reasons; there is only one atom node type. There are three different edge types: `bodyAtom` (meaning "is body atom of"), `rule` (meaning "is rule of"), and `supports`. Properties are denoted with dotted lines. The plus by the bottom left atom node is to indicate that there can be multiple atom nodes connected to a support node via `bodyAtom` edges.

Neo4j is designed for handling directed property graphs: directed graphs the nodes and edges of which can be adorned with multiple properties, edges (relationships in the Neo4j terminology) are typed. The types of edges and the number and type of properties need not be specified in advance (i.e. Neo4j is schema-less). There is no query language directly in Neo4j, instead, one has to traverse graphs manually by navigating between nodes and edges using the Neo4j API (natively in Java but bindings are provided for other languages too). Alternatively, Neo4j provides a so called Traverser class which can be used to traverse a graph in breadth-first or depth-first manner. It is possible to specify a starting node, a stopping criterion, a condition to determine which nodes are to be returned, and relationship types to traverse.

There also is an "XPath inspired graph traversal and manipulation language"[11] called Gremlin[12] as a third-party alternative to manual traversals.

The Neo4j API for nodes provides access to their associated relationships by type and direction. A Neo4j relationship provides methods that return its beginning node and its end node. Properties of nodes and edges must be of one of Java's primitive types.

Support graphs can easily be modelled using property graphs, see Figure 9.4. Atoms, rules, and supports are represented by nodes each having an `id` property the value of which uniquely identifies the represented object (a specific atom, a rule, or a support respectively). Note that there are at most as many rule nodes in a specific support graph as there are rules in the corresponding program. This has the advantage that a rule node gives direct access to all supports that use it. If a rule of a support was represented as a property of the support node then there would be no direct access to supports that use a specific rule.

Let us again explore how to determine atoms depending on a given atom "f", see Listing 9.4.

Listing 9.4: Finding atoms depending directly on a given atom `f` in Neo4j

11 http://wiki.neo4j.org/content/Gremlin
12 http://github.com/tinkerpop/gremlin

```
Set<Node> dependingNodes = new HashSet<Node>();
Iterable<Relationship> bodyAtomEdges =
  f.getRelationships(EdgeType.bodyAtom);

for (Relationship bae : bodyAtomEdges) {
    Node support = bae.getEndNode();
    Relationship supports =
      support.getRelationship(EdgeType.supports);
    Node depAtom = supports.getEndNode();
    dependingNodes.add(depAtom);
}
```

Note that new edge and node types as well as new properties can be added to an existing graph any time without a need to modify existing data. A disadvantage is that graph traversing has to be specified purely imperatively and it is Neo4j-specific.

There is no concept of a "join" in graph databases. A graph is essentially an already "joined structure" where each node and edge has references to its adjacent nodes and edges. This has two important implications: first, it is difficult to shard[13] a graph, and second, adjacent vertices and edges are retrieved in constant time [189]. In comparison, retrieving an adjacent node from a relational database takes at least logarithmic time if a join is involved and indexes are used [189].

Let us now compare the relational and graph database representations and the respective query evaluation complexity more closely.

9.3 COMPARISON

First, let us inspect the relational database approach, see Listing 9.3. The problem is to find ids of atoms that depend directly on a given atom with id ATOMID:

1. find all rows with `BodyAtom.atom` = ATOMID
2. for each found `BodyAtom` row ba, find a supp. s with $s.id = ba.support$
3. for each found support s, find the atom it supports, i.e. $s.atom = adep.id$

Assuming indexes are used, each step takes logarithmic time in the size of the respective table. If there are b rows in the `BodyAtom` table, s rows in the `Support` table, and a rows in the `Atom` table then the time to process the query is $O(log_2(b) * log_2(s) * log_2(a))$. It usually will be the case that $b > 2 * s$ because it can be assumed that there are at least two body atoms per support. Therefore the overall time is $O(log_2^2(b) * log_2(a))$.

Let us do the same analysis for the graph database approach, analogously to the more general account given in [189]. The problem is to find all atoms directly depending on an atom represented by a node "f":

1. find all outgoing relationships of type `EdgeType.bodyAtom` of the node f

13 To scale a database horizontally, i.e. across multiple machines.

2. for each found `bodyAtom` relationship find its end node (representing a support)
3. for each found support node find its `EdgeType.Supports` relationship
4. for each found `Supports` relationship find its end node (representing an atom)

It should be noted that the atom node f may have to be first found by its ATOMID. This is however only a one-time operation in an algorithm looking for all atoms (indirectly) depending on a given atom. After the atom node is found by the atom id, other atom nodes are found by traversing edges. The first search has to be fast too. This is usually achieved by indexing nodes by a property – Neo4j offers a Lucene[14]-based indexing service. Looking up the atom node f by its id can thus be assumed to take $O(\log_2(a))$ time.

Let us first assume that the graph database indexes node relationships by relationship type and properties by property type. In a graph database, adjacent nodes and edges are accessible via direct references and thus step 1 requires $O(r)$ operations, where r is the number of `EdgeType.bodyAtom` relationships of the node f. Steps 2-4 each require also $O(r)$ operations because each edge has only one end node and each support has only one `supports` relationship. If atoms are represented externally then an additional step of $O(r)$ operations has to be executed which will query properties of the found nodes for their external identifiers.

If indexes for relationships and properties are not used then the first step requires a sequential scan of $O(r + x)$ operations, where x is the number of incoming `bodyAtom` relationships of each atom node assuming this number is the same for all atom nodes (i.e. node f has r outgoing `bodyAtom` relationships, x incoming `bodyAtom` relationships, and it has no other relationships). The second step requires $O(r)$ operations. The third step requires $O(r * (1 + y + 1))$ operations, where y is the number of `bodyAtom` edges of a support. Step 4 requires again $O(r)$ operations. The additional step of finding external identifiers requires $O(p * r)$ operations if there are p properties per atom node, in this example $p = 1$.

Overall, the time to retrieve atoms depending on a given atom is $O(r)$ if indexes for relationships and properties are used and $O(r * y + x)$ if indexes are not used.

Note that both r and $r * y + x$ are much smaller than any of b, s, a. The time required by a graph database implementation of a graph traversal algorithm depends only on the number of nodes and vertices visited by the algorithm. In particular, it is independent of the size of the graph. By contrast, the relational database implementation requires time dependent on the size of the tables, i.e. it depends on the size of the whole graph. The difference is likely to be significant in cases where an algorithm traverses a graph along long paths. As noted earlier, a price to pay for faster traversals is substantial difficulty of horizontal scaling due to the difficulty of automatic splitting of a graph into (independent) parts.

See for example [216] for a comparison (including benchmarks) of Neo4j and MySQL from the perspective of storing and querying provenance data. Soussi et al. [205] discuss how relational data related to social networks may be converted to a graph database representation.

14 `http://lucene.apache.org/`

9.4 THE KIWI 1.0 IMPLEMENTATION

KiWi 1.0 is an enterprise Java application built using the Seam framework[15], a framework for building web applications, and the JBoss[16] application server. It is a platform for building social semantic web applications (one example of which is the KiWi wiki). As such it has to support a wide range of use cases which is reflected also in its pragmatically hybrid and often redundant architecture.

The implementation of reasoning, reason maintenance, and explanation also employs a hybrid approach. Atoms are stored in a relational database so that they can easily be queried by SQL and automatically mapped to Java objects using the JPA[17] Java technology. A graph database is used to store the compact graph $SG(skT_P^\omega)$ which enables an efficient implementation of reason maintenance and explanation algorithms which are predominantly graph algorithms. The link between the two data management systems is the primary key value of the atom in the relational database which is also stored as a property of atom nodes in the graph database.

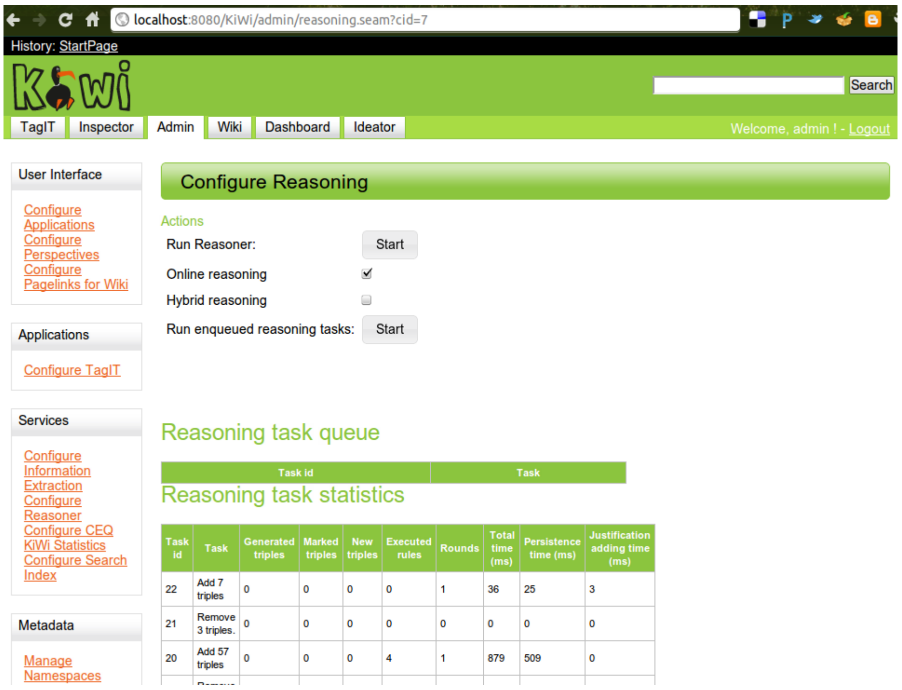
KiWi 1.0 implements semi-naive reasoning with sKWRL rules by translating sKWRL rule bodies into JQL (a Java Peristence API version of SQL) in order to query the relational database system. The retrieved results are also stored as supports in the Neo4j graph database according to the scheme in Figure 9.4. The advantage of using JPA and JQL over a specific relational database system is the possibility to switch between different relational databases without changing source code. On the other hand, it hinders efficiency due to the inability to use a native database access. The implementation is a proof of concept not aiming for high efficiency in all cases. Note that other semantic web applications and especially triple stores often implement native data storage to improve efficiency. Designing and building an efficient triple store is not the goal of this dissertation or of the KiWi project. KiWi in fact uses the Sesame[18] triple store to enable SPARQL querying.

The value invention feature of sKWRL rules is implemented using the Rabin fingerprinting method [184, 37]. A character string created from a rule body instance is hashed using the Rabin's fingerprint function that provides probabilistic guarantees on the improbability of a collision. This way it is guaranteed that new (reasonably-sized) constants are created when necessary without keeping track of rule body instances explicitly.

A graph database was chosen for two reasons: first, it is schema-less which facilitates development and second, it is optimized for graph traversals which is important for implementation of graph algorithms for reason maintenance and explanation. KiWi 1.0 implements the current support algorithm (Algorithm 8.24) because the whole compact graph is stored and available in the Neo4j database in order to be able to provide fast explanations. It would be desirable to implement also Algorithm 8.15 (reason maintenance without support graphs) in a future release of KiWi. A graph database can then still be used for example in order to provide fast explanations for data often used by users (see also Section 8.8).

Explanation services in KiWi 1.0 provide access to parts of a support graph on demand. Any atom can be readily and quickly "explained" which

15 `http://seamframework.org/`
16 `http://www.jboss.org/`
17 Java Persistence API – a Hibernate-based object-relational mapping system.
18 `http://www.openrdf.org/`



Configure Reasoning

Actions

Run Reasoner: Start

Online reasoning ✓

Hybrid reasoning

Run enqueued reasoning tasks: Start

Reasoning task queue

Task id	Task

Reasoning task statistics

Task id	Task	Generated triples	Marked triples	New triples	Executed rules	Rounds	Total time (ms)	Persistence time (ms)	Justification adding time (ms)
22	Add 7 triples	0	0	0	0	1	36	25	3
21	Remove 3 triples.	0	0	0	0	0	0	0	0
20	Add 57 triples	0	0	0	4	1	879	509	0

Figure 9.5: A screenshot of the KiWi 1.0 reasoning configuration and status page. The "Online reasoning" option means that reasoning tasks (triple additions and removals) are processed as they come. The "Hybrid reasoning" option refers to an experimental optimization of the reasoner with respect to the current TBox analogously to a similar optimization employed in the SAOR reasoner [116].

allows for a nimble user interface that is easily understandable for users. For example the explanation tooltips (see Section 8.7) are praised by users of the KiWi wiki. The KiWi user interface is rendered based on information loaded from a relational database. Therefore atoms' primary keys are available during rendering and they are used to query the graph database to obtain explanations of the atom by traversing from corresponding atom node against edge directions. This means that the starting point of the traversal, an atom node corresponding to the body atom to be explained, has to be quickly accessible in the graph database for example by the primary key of the corresponding body atom. For this purpose and as already mentioned, Neo4j provides an indexing mechanism based on the Lucene full-text search engine library which is used in the KiWi 1.0 implementation too. For a more thorough discussion of explanation see Section 8.7.

The hybrid approach allows to leverage a relational database for implementation of rule-based (sKWRL) reasoning by translation of rule bodies to SQL while explanation and the graph-based reason maintenance algorithm are implemented using Neo4j and thus make use of fast graph traversals.

10 CONCLUSION AND PERSPECTIVES

In this thesis, we have investigated theoretical and practical problems stemming from extending social software with advanced semantic functionalities such as annotation, reasoning, and explanation. First we presented social semantic software, analysed its defining properties, and formulated requirements that guided the rest of the thesis.

The conceptual model of a semantic wiki described in Chapter 4 has been implemented to a high degree in the KiWi 1.0 release. Structured tags are the most significant part of the model that has not been implemented and that has been evaluated only in the user study described in Chapter 5. An implementation of structured tags poses interesting challenges with respect to user interface, querying, and reasoning. In addition to the structural querying of structured tags already mentioned in Chapter 6 and partly addressed in [223], it is also the problem of propagating and accumulation of information along paths in structured graphs. For example, the question of authorship of derived annotations and its querying is left open in this thesis. It has been in part addressed in our article [42] where we show that the impending combinatorial explosion may be sidetracked by keeping information only for groups of authors that will likely be a natural part of an enterprise wiki system. This problem is closely related to the area of provenance and provenance querying [202, 94, 232] that is becoming a more and more active area of research with the increasing availability of data.

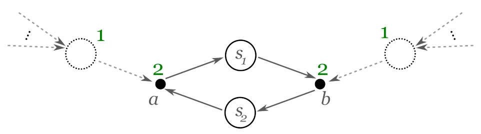
The KWRL language is to the best of our knowledge the first rule language focused on annotations. It provides a compact syntax aware of the conceptual model and it provides a value invention feature that fits the Linked Data requirements and annotations. KWRL has been implemented only in part in KiWi 1.0. The syntax of KWRL adopts the keyword:value style which is close to the syntax of the KWQL query language developed in [223]. KWQL allows also for specification of boolean queries which provides a possible point of intersection of the two languages. Another integration possibility is a loose integration of KWRL and KWQL via special built-in constructs aimed solely at annotation querying in KWQL with KWRL constructs and at rule-based reasoning with structural and fuzzy queries in KWRL with KWQL constructs. A closer investigation of the relationship and possible integration points of the two languages is desirable as it could result in a more coherent and powerful system.

The presented extended forward chaining reasoning methods cover a broad range of possibilities with respect to the information that is derived in addition to the standard forward chaining. We have designed and analysed a reason maintenance algorithm that works without support graphs and that provides significant advantages over the traditional DRed and PF algorithms for the price of modifying the classical forward chaining procedure. The skT_P, scT_P, dcT_P operators and the related theorems clarify the relationship between supports and derivations which has not been previously done and the analysis leads to a formulation of novel reason maintenance algorithms based on counting supports. We have provided several derivation counting algorithms that offer different space-time trade-offs and that provide information that can be used for explanation. The properties of all the methods are formally investigated and proved within the classical logical programming framework based on fixpoint theory and the methods are formally compared thanks to their formulation within the unifying framework of support graphs.

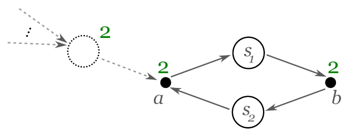
This dissertation does not go into full details with respect to the question of optimality of the suggested derivation counting methods. There however are reasons to believe that the methods are nearly optimal. The non-recursive Datalog case enables formulation of purely incremental derivation counting and reason maintenance algorithms – there is no need to detect cycles and thus the termination condition is simple. In the recursive Datalog case, it is necessary to detect cycles and there seems to be no simple way of formulating a similarly efficient divide and conquer algorithm. The reason is that following different paths in a support graph necessarily leads to formulation of different sub-problems which makes optimization hard: if we want to count the number of derivations of an atom a that directly depends on atoms b and c then we can count the number of derivations of the atom b that do not use a and the number of derivations of c that do not use a, i.e. solving the sub-problem for b and c depends on how they were reached during the previous execution of the algorithm making it hard to reuse information computed on the way to a; no two sub-problems are the same unless the algorithm explores exactly the path it had explored before. Another point of view is that the derivation count of an atom is a global property that in general cannot be computed from a local subgraph. See Figure 10.1a. Let a, b, s_1, s_2 be the local subgraph. Then the numbers of derivations of atoms "to the left of a" do not help in determining the number of derivations of a because it is not clear for example whether the atoms depend also on b and thus on a. Indeed, the number of derivations of an atom provides little information about the structure of the whole graph, e.g. the two situations in Figure 10.1a and Figure 10.1b, where in both cases a and b have two derivations but for different reasons, are indistinguishable from only the local subgraph and the numbers of derivations. Note that this situation is distinguishable with respect to support counts. In Figure 10.1a, both a and b have two supports, in Figure 10.1b, a has two supports (only one is well-founded) and b has one support. This however does not help because supports are not evidence of derivability unless they are well-founded. And well-foundedness is again a global property that cannot be determined only from a local subgraph in general. Despite these difficulties, the concept of safeness introduced and discussed in Section 8.5.3 seems to offer a promising direction for finding improvements to reason maintenance algorithms.

It would be interesting to see how our reason maintenance algorithms could be applied in a distributed Datalog setting as discussed for example in [167] and in Section 8.6.4. The method of [167] is based on the DRed algorithm [105] for which we have provided alternatives – mainly in Section 8.4.

We have also offered and implemented an approach to explaining derived data to a non-expert user. Explanation would deserve a more in-depth treatment that would consider different ways of transforming and presenting derivations to make them more accessible to the casual user. For this purpose, alternative definitions of a derivation can be explored that would further limit the amount of redundant (and possibly confusing) information.



(a) a has one derivation "from outside" and one via s_2. Similarly for b.



(b) a has two derivations "from outside," b has two derivations via s_1.

Figure 10.1: Support graphs with derivation counts. Numbers by supports indicate the number of derivations that the support contributes to the supported atom.

Such a limitation may however make derivation counting more expensive even in the non-recursive Datalog case and further investigation would be desirable.

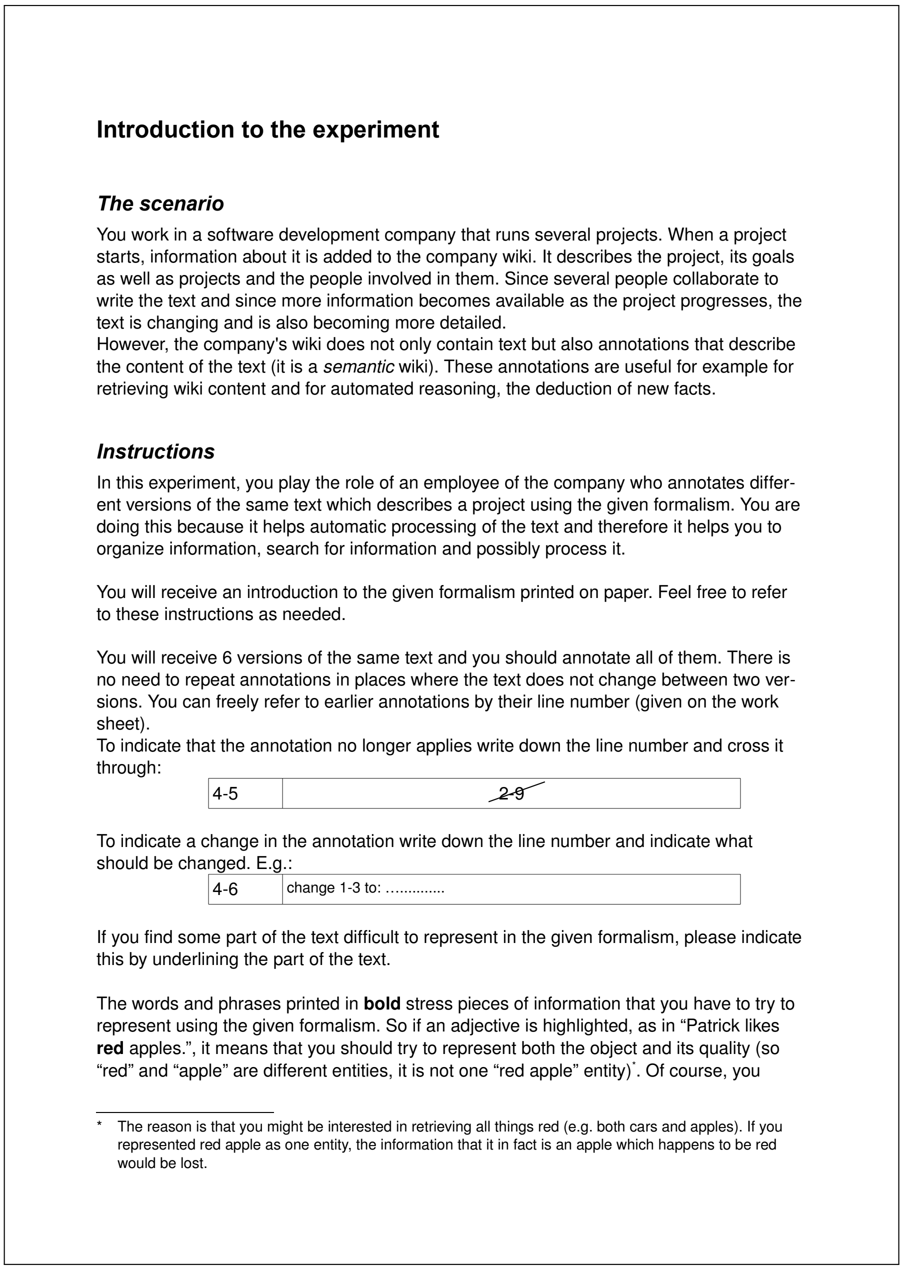
Lastly, this dissertation focuses on forward chaining and it would be interesting to explore a combination of some of the algorithms with backward methods. Backward methods can for example be used for explanation when no support graph is available. Note that the fact that a program is materialized means that the backward procedure becomes a simple lookup of atoms that are already materialized.

APPENDIX

A STRUCTURED TAGS USER STUDY MATERIALS

A.1 INTRODUCTORY TEXTS

A.1.1 *General Instructions*



Introduction to the experiment

The scenario

You work in a software development company that runs several projects. When a project starts, information about it is added to the company wiki. It describes the project, its goals as well as projects and the people involved in them. Since several people collaborate to write the text and since more information becomes available as the project progresses, the text is changing and is also becoming more detailed.
However, the company's wiki does not only contain text but also annotations that describe the content of the text (it is a *semantic* wiki). These annotations are useful for example for retrieving wiki content and for automated reasoning, the deduction of new facts.

Instructions

In this experiment, you play the role of an employee of the company who annotates different versions of the same text which describes a project using the given formalism. You are doing this because it helps automatic processing of the text and therefore it helps you to organize information, search for information and possibly process it.

You will receive an introduction to the given formalism printed on paper. Feel free to refer to these instructions as needed.

You will receive 6 versions of the same text and you should annotate all of them. There is no need to repeat annotations in places where the text does not change between two versions. You can freely refer to earlier annotations by their line number (given on the work sheet).
To indicate that the annotation no longer applies write down the line number and cross it through:

4-5	~~2-9~~

To indicate a change in the annotation write down the line number and indicate what should be changed. E.g.:

4-6	change 1-3 to: ....

If you find some part of the text difficult to represent in the given formalism, please indicate this by underlining the part of the text.

The words and phrases printed in **bold** stress pieces of information that you have to try to represent using the given formalism. So if an adjective is highlighted, as in “Patrick likes **red** apples.”, it means that you should try to represent both the object and its quality (so “red” and “apple” are different entities, it is not one “red apple” entity)*. Of course, you

* The reason is that you might be interested in retrieving all things red (e.g. both cars and apples). If you represented red apple as one entity, the information that it in fact is an apple which happens to be red would be lost.

should also annotate the rest of the text, but annotating the bold sections should be a priority.

The gray text indicates for your convenience what information changed in the current revision compared to the previous revision.

You can add as many annotations and as detailed annotations as you feel is appropriate.

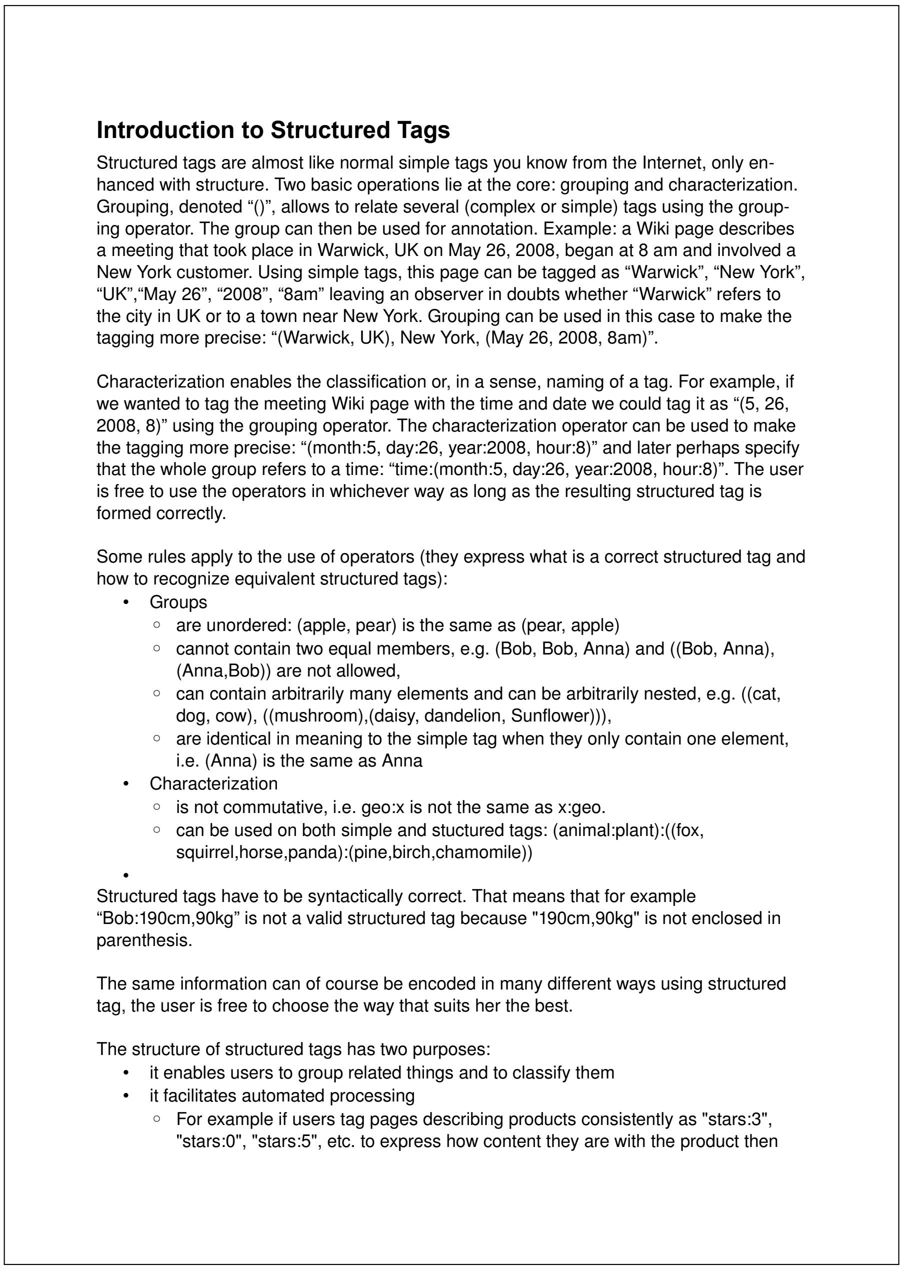
Once you are finished annotating a section, it is not allowed to modify it anymore in any way (adding, changing, ...).

You will only receive the next revision, that is, the new worksheet, after you have finished annotating the previous version.

After this part of the experiment, you will be asked to fill out a questionnaire about your experiences.

Most of the questions should only be answered when you have annotated all six versions, but the first sections requires you to fill in the times when you started and finished annotating. Please remember to fill this in as soon as you can.

A.1.2 *An Introduction to Structured Tags*



Introduction to Structured Tags

Structured tags are almost like normal simple tags you know from the Internet, only enhanced with structure. Two basic operations lie at the core: grouping and characterization. Grouping, denoted “()”, allows to relate several (complex or simple) tags using the grouping operator. The group can then be used for annotation. Example: a Wiki page describes a meeting that took place in Warwick, UK on May 26, 2008, began at 8 am and involved a New York customer. Using simple tags, this page can be tagged as “Warwick”, “New York”, “UK”,“May 26”, “2008”, “8am” leaving an observer in doubts whether “Warwick” refers to the city in UK or to a town near New York. Grouping can be used in this case to make the tagging more precise: “(Warwick, UK), New York, (May 26, 2008, 8am)”.

Characterization enables the classification or, in a sense, naming of a tag. For example, if we wanted to tag the meeting Wiki page with the time and date we could tag it as “(5, 26, 2008, 8)” using the grouping operator. The characterization operator can be used to make the tagging more precise: “(month:5, day:26, year:2008, hour:8)” and later perhaps specify that the whole group refers to a time: “time:(month:5, day:26, year:2008, hour:8)”. The user is free to use the operators in whichever way as long as the resulting structured tag is formed correctly.

Some rules apply to the use of operators (they express what is a correct structured tag and how to recognize equivalent structured tags):

- Groups
  - are unordered: (apple, pear) is the same as (pear, apple)
  - cannot contain two equal members, e.g. (Bob, Bob, Anna) and ((Bob, Anna), (Anna,Bob)) are not allowed,
  - can contain arbitrarily many elements and can be arbitrarily nested, e.g. ((cat, dog, cow), ((mushroom),(daisy, dandelion, Sunflower))),
  - are identical in meaning to the simple tag when they only contain one element, i.e. (Anna) is the same as Anna
- Characterization
  - is not commutative, i.e. geo:x is not the same as x:geo.
  - can be used on both simple and stuctured tags: (animal:plant):((fox, squirrel,horse,panda):(pine,birch,chamomile))
-

Structured tags have to be syntactically correct. That means that for example “Bob:190cm,90kg” is not a valid structured tag because "190cm,90kg" is not enclosed in parenthesis.

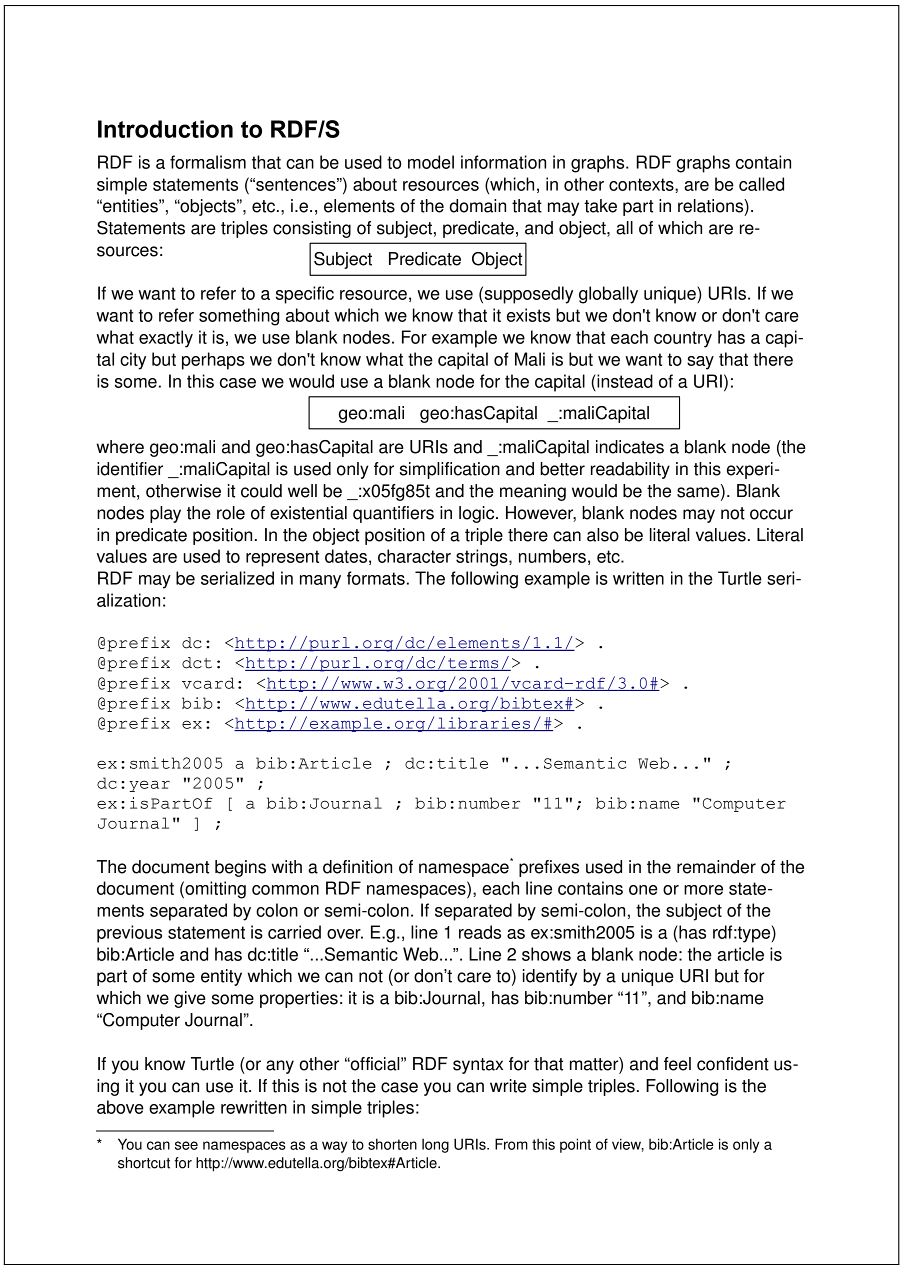
The same information can of course be encoded in many different ways using structured tag, the user is free to choose the way that suits her the best.

The structure of structured tags has two purposes:

- it enables users to group related things and to classify them
- it facilitates automated processing
  - For example if users tag pages describing products consistently as "stars:3", "stars:0", "stars:5", etc. to express how content they are with the product then

these tags can be automatically processed to compute for example average product ratings. And because the characterization operator is not commutative it would know that it should ignore tags such as "3:stars" because they can mean something else. (In case of grouping (3, stars) and (stars, 3) would be considered equal.)

A.1.3 *An Introduction to RDF*



Introduction to RDF/S

RDF is a formalism that can be used to model information in graphs. RDF graphs contain simple statements (“sentences”) about resources (which, in other contexts, are be called “entities”, “objects”, etc., i.e., elements of the domain that may take part in relations). Statements are triples consisting of subject, predicate, and object, all of which are resources:

Subject Predicate Object

If we want to refer to a specific resource, we use (supposedly globally unique) URIs. If we want to refer something about which we know that it exists but we don't know or don't care what exactly it is, we use blank nodes. For example we know that each country has a capital city but perhaps we don't know what the capital of Mali is but we want to say that there is some. In this case we would use a blank node for the capital (instead of a URI):

geo:mali geo:hasCapital _:maliCapital

where geo:mali and geo:hasCapital are URIs and _:maliCapital indicates a blank node (the identifier _:maliCapital is used only for simplification and better readability in this experiment, otherwise it could well be _:x05fg85t and the meaning would be the same). Blank nodes play the role of existential quantifiers in logic. However, blank nodes may not occur in predicate position. In the object position of a triple there can also be literal values. Literal values are used to represent dates, character strings, numbers, etc.
RDF may be serialized in many formats. The following example is written in the Turtle serialization:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix bib: <http://www.edutella.org/bibtex#> .
@prefix ex: <http://example.org/libraries/#> .

ex:smith2005 a bib:Article ; dc:title "...Semantic Web..." ;
dc:year "2005" ;
ex:isPartOf [ a bib:Journal ; bib:number "11"; bib:name "Computer
Journal" ] ;
```

The document begins with a definition of namespace* prefixes used in the remainder of the document (omitting common RDF namespaces), each line contains one or more statements separated by colon or semi-colon. If separated by semi-colon, the subject of the previous statement is carried over. E.g., line 1 reads as ex:smith2005 is a (has rdf:type) bib:Article and has dc:title “...Semantic Web...”. Line 2 shows a blank node: the article is part of some entity which we can not (or don’t care to) identify by a unique URI but for which we give some properties: it is a bib:Journal, has bib:number “11”, and bib:name “Computer Journal”.

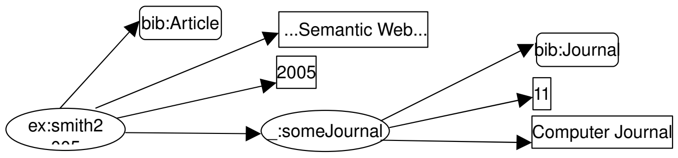
If you know Turtle (or any other “official” RDF syntax for that matter) and feel confident using it you can use it. If this is not the case you can write simple triples. Following is the above example rewritten in simple triples:

* You can see namespaces as a way to shorten long URIs. From this point of view, bib:Article is only a shortcut for http://www.edutella.org/bibtex#Article.

ex:smith2005	rdf:type	bib:Article
ex:smith2005	dc:title	“...Semantic Web...”
ex:smith2005	dc:year	“2005”
ex:smith2005	ex:isPartOf	_:someJournal
_:someJournal	rdf:type	bib:Journal
_:someJournal	bib:number	“11”
_:someJournal	bib:name	“Computer Journal”

See the end of this introduction for details of the conventions used in this simplified syntax.

The following figure shows a visual representation of the above RDF data, where we distinguish literals (in square boxes) and classes, i.e., resources that can be used for classifying other resources, and thus can be the object of an rdf:type statement (in square boxes with rounded edges) from all other resources (in plain ellipses).



RDF graphs contain simple statements about resources. RDF Schema lets us specify vocabularies (terms) that we wish to use in those statements. In the example above, we would want for example classes bib:Article, bib:Journal and perhaps also bib:Author and bib:Person. We could also want to use properties such as bib:hasAuthor, ex:isPartOf, ... to describe the classes. RDF schema allows resources to be instances of one or more classes. Classes can form a hierarchy (a little similar to object oriented type systems, but not quite) so for example if a resource ex:resourceABCD is of type zoo:Fox and it is specified that zoo:Fox is rdfs:subClassOf zoo:Mammal then we know that the resource is of type zoo:Mammal too (because that is how the meaning of the subclass relationship is defined: any instance of the subclass is also an instance of the superclass). In RDF Schema, class is any resource of type rdfs:Class. Hence, to assert that zoo:Fox is a class we would assert the following triple: zoo:Fox rdf:type rdfs:Class. The rdfs:subClassOf property is transitive. So if we also knew that zoo:Mammal rdfs:subClassOf ex:LivingThing then according to RDF Schema zoo:Fox would be a subclass of ex:LivingThing.

RDF Schema also allows for description of properties. Properties are described using the class rdfs:Property and properties rdfs:domain, rdfs:range, and rdfs:subPropertyOf. Therefore, to introduce new property ex:hasMother one would assert the triple: ex:hasMother rdf:type rdfs:Property. It is possible to further specify the property - it describes people: ex:hasMother rdfs:domain ex:Person, ex:hasMother rdfs:range ex:Person. Then if there is a triple of the form ex:a ex:hasMother ex:b then according to RDF Schema and our description of the property ex:hasMother we could derive that both ex:a and ex:b have type ex:Person (to be more precise, the following two triples could be derived: ex:a rdf:type ex:Person, ex:b rdf:type ex:Person).

Remarks

Following simplifying conventions hold for the use of the simple triples:

- The first letter of a class name is upper case. E.g.: `Person, Car, Animal`
- The first letter of an instance name is lower case. E.g.: `spoon, camera, pine`
- Literals are enclosed in quotation marks. E.g.: `"Red fox", "3 stars", "31415"`
- Blank nodes are indicated by `"_:"` prefix. E.g.: `_:b1, _:SomeBlankNodeName, _:blank`
- It is not necessary to use whole URIs or namespaces in RDF. But always be careful to put literals in quotation marks and to clearly distinguish blank nodes from other resources.
- Introduce new namespaces using the Turtle syntax:

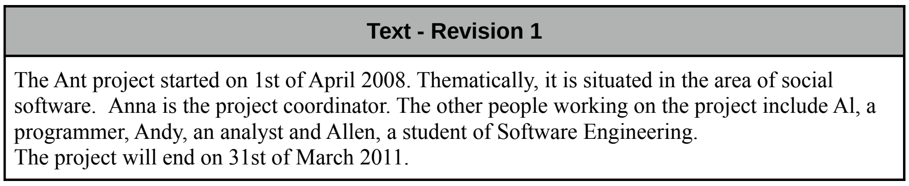
```
@prefix ex: <http://company.com/concepts/>
```

Examples

- `"New York"` - a literal string
- `NewYork` - a class which in real RDF has a URI and is of type rdfs:Class (it means that the following triple is true: NewYork rdfs:type rdfs:Class)
- `newYork` - an instance which in real RDF has a URI and is not of type rdfs:Class
- `_:NewYork` - a blank node

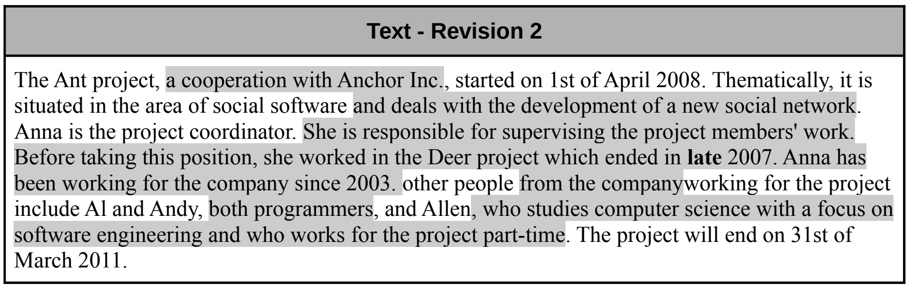
A.2 texts for annotation

A.2.1 *Text A*



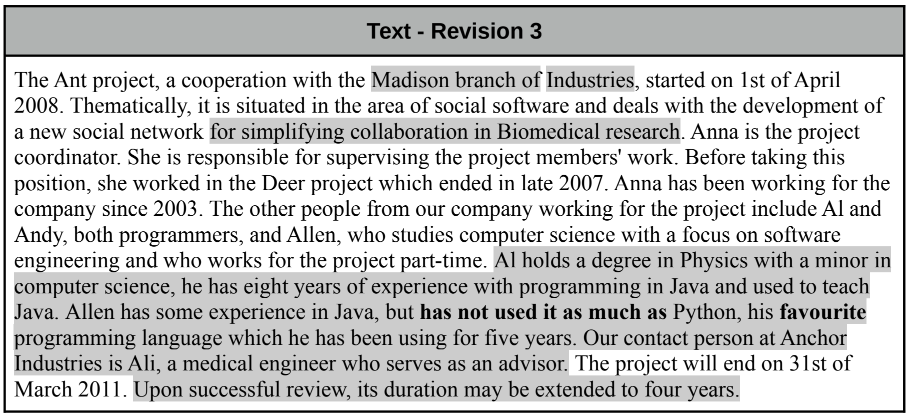
Text - Revision 1

The Ant project started on 1st of April 2008. Thematically, it is situated in the area of social software. Anna is the project coordinator. The other people working on the project include Al, a programmer, Andy, an analyst and Allen, a student of Software Engineering.
The project will end on 31st of March 2011.



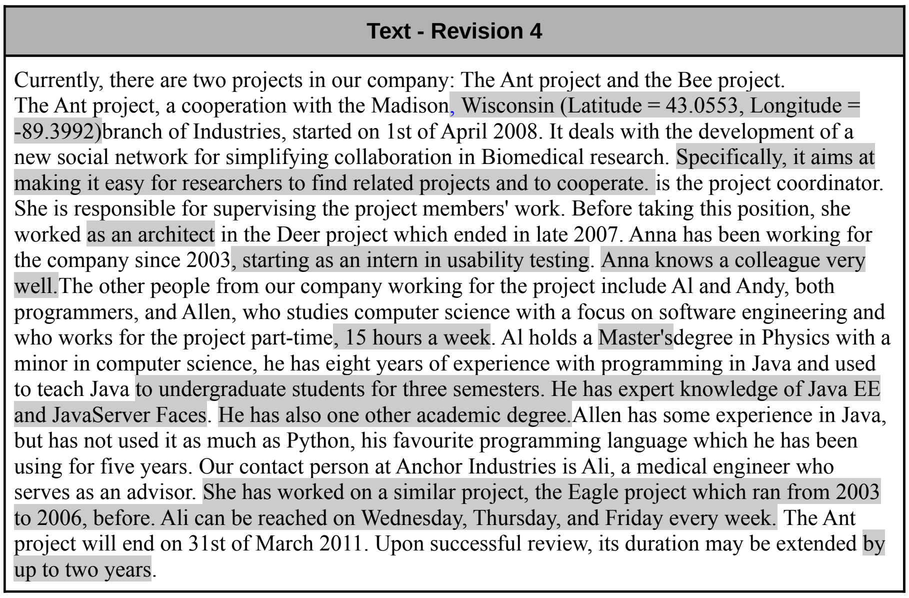
Text - Revision 2

The Ant project, a cooperation with Anchor Inc., started on 1st of April 2008. Thematically, it is situated in the area of social software and deals with the development of a new social network. Anna is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked in the Deer project which ended in **late** 2007. Anna has been working for the company since 2003. other people from the companyworking for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time. The project will end on 31st of March 2011.



Text - Revision 3

The Ant project, a cooperation with the Madison branch of Industries, started on 1st of April 2008. Thematically, it is situated in the area of social software and deals with the development of a new social network for simplifying collaboration in Biomedical research. Anna is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked in the Deer project which ended in late 2007. Anna has been working for the company since 2003. The other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time. Al holds a degree in Physics with a minor in computer science, he has eight years of experience with programming in Java and used to teach Java. Allen has some experience in Java, but **has not used it as much as** Python, his **favourite** programming language which he has been using for five years. Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor. The project will end on 31st of March 2011. Upon successful review, its duration may be extended to four years.



Text - Revision 4

Currently, there are two projects in our company: The Ant project and the Bee project.
The Ant project, a cooperation with the Madison, Wisconsin (Latitude = 43.0553, Longitude = -89.3992)branch of Industries, started on 1st of April 2008. It deals with the development of a new social network for simplifying collaboration in Biomedical research. Specifically, it aims at making it easy for researchers to find related projects and to cooperate. is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked as an architect in the Deer project which ended in late 2007. Anna has been working for the company since 2003, starting as an intern in usability testing. Anna knows a colleague very well.The other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time, 15 hours a week. Al holds a Master'sdegree in Physics with a minor in computer science, he has eight years of experience with programming in Java and used to teach Java to undergraduate students for three semesters. He has expert knowledge of Java EE and JavaServer Faces. He has also one other academic degree.Allen has some experience in Java, but has not used it as much as Python, his favourite programming language which he has been using for five years. Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor. She has worked on a similar project, the Eagle project which ran from 2003 to 2006, before. Ali can be reached on Wednesday, Thursday, and Friday every week. The Ant project will end on 31st of March 2011. Upon successful review, its duration may be extended by up to two years.

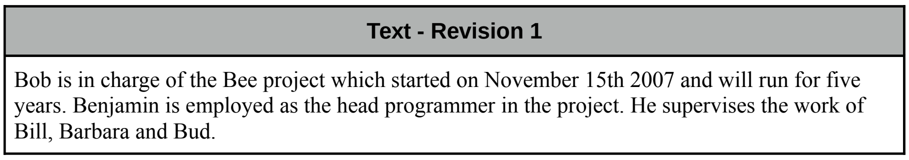
Text - Revision 5

Currently, there are two projects in our company: The Ant project and the Bee project.
The Ant project, a cooperation with the Madison, Wisconsin (Latitude = 43.0553, Longitude = -89.3992) branch of IndustriesAnchor Industries, started on 1st of April 2008. It deals with the development of a new social network for simplifying collaboration in Biomedical research. Specifically, it aims at making it easy for researchers to find projects that are similar in their topic, participating researchers and institutions or location and to cooperate by sharing access to expensive experimental equipment that not all research facilities own and by sharing the contact data of participants which might be interested in participating in further experiments. Anna is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked as an architect in the Deer project which ended in late 2007. Anna has been working for the company since 2003, starting as an intern in usability testing. Anna knows a colleague very well, the colleague is Andy. other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time, 15 hours a week. Al holds a Master's degree in Physics with a minor in computer science. He has eight years of experience with programming in Java and used to teach Java to undergraduate students for three semesters. He has expert knowledge of Java EE and JavaServer Faces. He also has a degree in mechanical engineering. Andy is a **self-taught** programmer who holds a Bachelor's degree in Biology. He has five years of experience in Programming in Java, and seven years of experience with web programming overall. has taken three classes in Java at university,but has not used it as much as Python, his favourite programming language which he has been using for five years. Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor in questions concerning biomedical practice. She has worked on a similar project, the Eagle project which ran from 2003 to 2006, before. Ali can be reached by phoneon Wednesday, Thursday, and Friday every week but can reply to emails from Monday to Friday. Ali **most probably** knows Andy. Ant project will end on 31st of March 2011. Upon successful review, its duration may be extended by up to two years.

Text - Revision 6

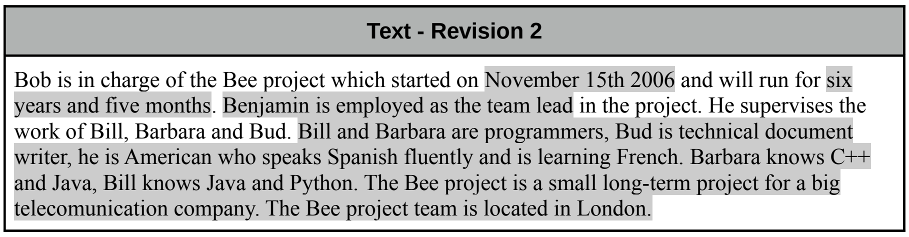
The Ant project, a cooperation with the Madison, Wisconsin (Latitude = 43.0553, Longitude = -89.3992) branch of Anchor Industries, started on 1st of April 2008. It deals with the development of a new social network for simplifying collaboration in Biomedical research (it is not about **inorganic** chemistry but it is also about **organic** chemistry). Specifically, it aims at making it easy for researchers to find projects that are similar in their topic, participating researchers and institutions or location and to cooperate by sharing access to expensive experimental equipment that not all research facilities own and by sharing the contact data of participants which might be interested in participating in further experiments. On the other hand, it does not help with process management. The software will be written in Java using the JBoss Seam framework.Anna is the senior project coordinator. She is responsible for supervising the project members' work and overseeing the evaluation of the final product. Before taking this position, she worked as an architect in the Deer project which ended in late 2007. Anna has been working for the company since 2003, starting as an intern programmer. Anna knows a colleague very well, the colleague is Andy. The other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time, 15 hours a week (it is full hours, not the **45 min.** teaching units). Al holds a Master's degree in Physics with a minor in computer science. He has eight years of experience with programming in Java and used to teach Java to undergraduate students for three semesters. He has expert knowledge of Java EE and JavaServer Faces. He also has a degree in mechanical engineering. Before joining our company, he worked as a Perl programmer for three years.Andy is a self-taught programmer who holds a Bachelor's degree in Biology. He has five years of experience in Programming in Java, and seven years of experience with web programming overall. He does not know **the programming language** Haskell. Allen has taken three classes in Java at university, but has not used it as much as Python, his favourite programming language which he has been using for five years. He does not know any other programming language.Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor in questions concerning biomedical practice. She has worked on a similar project, the Eagle project which ran from 2003 to 2006, before. Ali can be reached by phone on Wednesday, Thursday, and Friday every week but can reply to emails from Monday to Friday. Ali is not a programmer and does not want to consult information technology problems.Ali most probably knows Andy. The Ant project will end on 31st of March 2011. Upon successful review, its duration may be extended by up to two years.

A.2.2 *Text B*



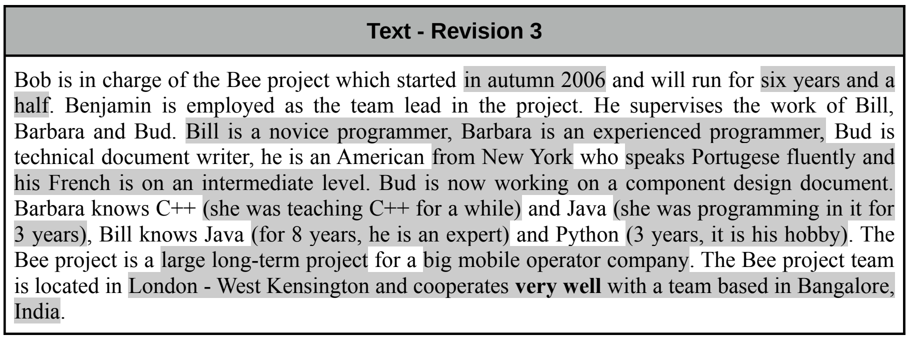
Text - Revision 1

Bob is in charge of the Bee project which started on November 15th 2007 and will run for five years. Benjamin is employed as the head programmer in the project. He supervises the work of Bill, Barbara and Bud.



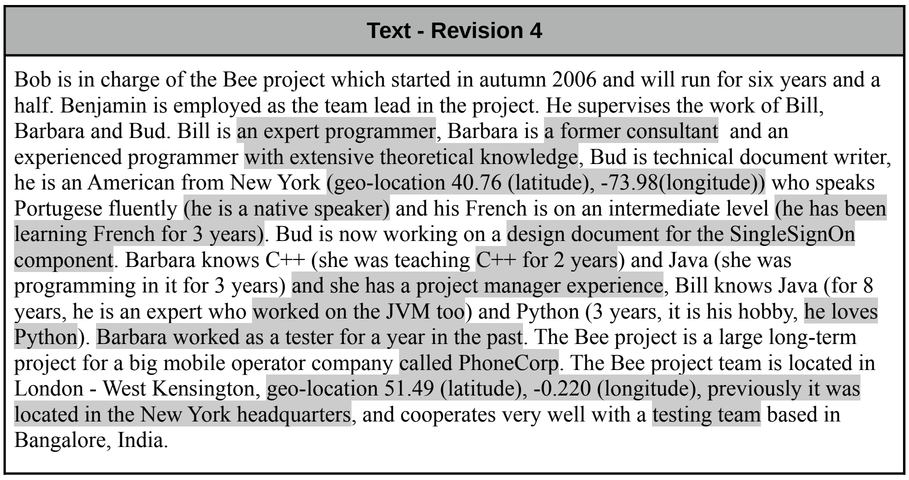
Text - Revision 2

Bob is in charge of the Bee project which started on November 15th 2006 and will run for six years and five months. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill and Barbara are programmers, Bud is technical document writer, he is American who speaks Spanish fluently and is learning French. Barbara knows C++ and Java, Bill knows Java and Python. The Bee project is a small long-term project for a big telecomunication company. The Bee project team is located in London.



Text - Revision 3

Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill is a novice programmer, Barbara is an experienced programmer, Bud is technical document writer, he is an American from New York who speaks Portugese fluently and his French is on an intermediate level. Bud is now working on a component design document. Barbara knows C++ (she was teaching C++ for a while) and Java (she was programming in it for 3 years), Bill knows Java (for 8 years, he is an expert) and Python (3 years, it is his hobby). The Bee project is a large long-term project for a big mobile operator company. The Bee project team is located in London - West Kensington and cooperates **very well** with a team based in Bangalore, India.



Text - Revision 4

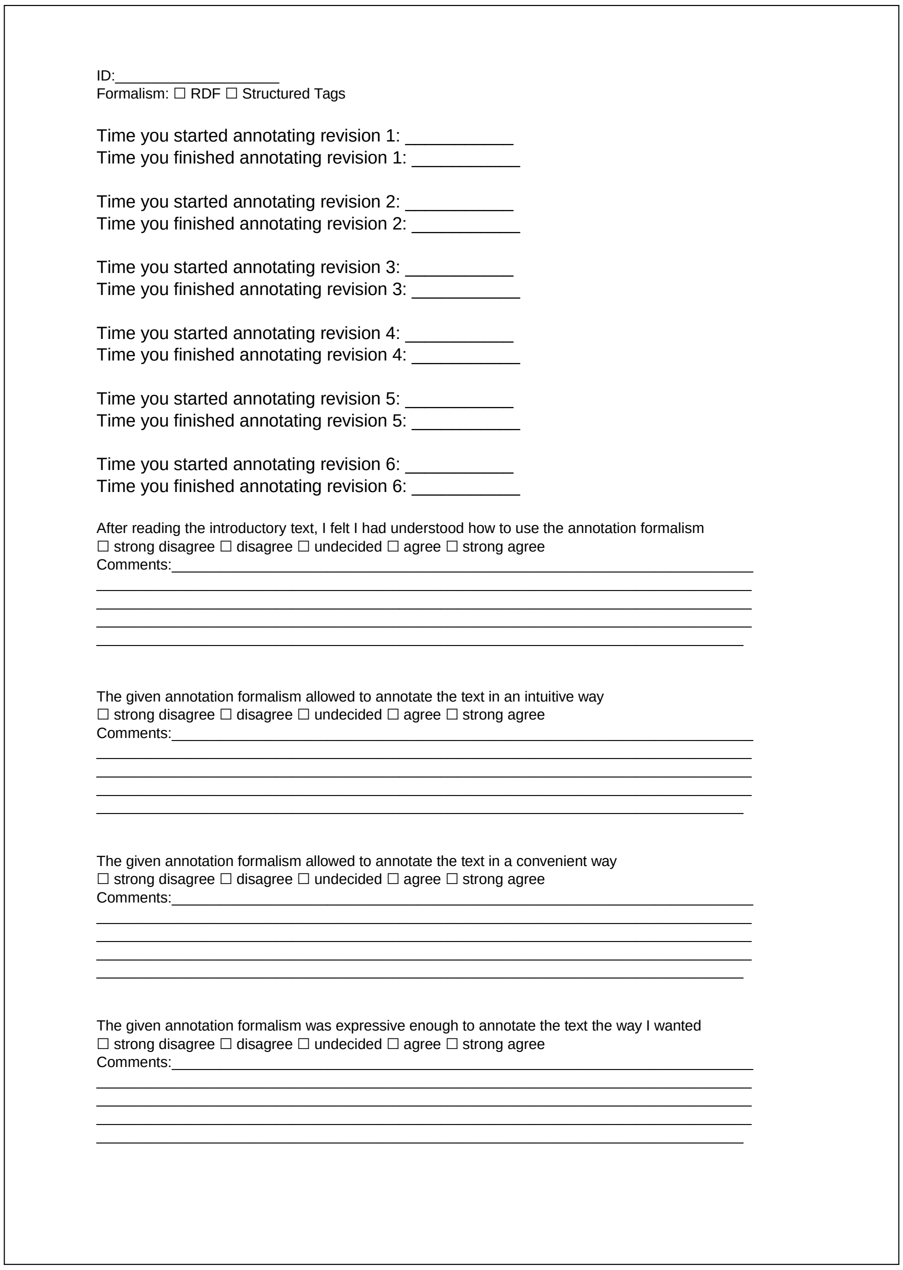
Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill is an expert programmer, Barbara is a former consultant and an experienced programmer with extensive theoretical knowledge, Bud is technical document writer, he is an American from New York (geo-location 40.76 (latitude), -73.98(longitude)) who speaks Portugese fluently (he is a native speaker) and his French is on an intermediate level (he has been learning French for 3 years). Bud is now working on a design document for the SingleSignOn component. Barbara knows C++ (she was teaching C++ for 2 years) and Java (she was programming in it for 3 years) and she has a project manager experience, Bill knows Java (for 8 years, he is an expert who worked on the JVM too) and Python (3 years, it is his hobby, he loves Python). Barbara worked as a tester for a year in the past. The Bee project is a large long-term project for a big mobile operator company called PhoneCorp. The Bee project team is located in London - West Kensington, geo-location 51.49 (latitude), -0.220 (longitude), previously it was located in the New York headquarters, and cooperates very well with a testing team based in Bangalore, India.

Text - Revision 5
Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill is an expert programmer, Barbara is a former junior consultant and an experienced programmer with extensive theoretical knowledge (she has a background in theoretical computer science, esp. the theory of complexity), Bud is technical document writer, he is an African-American from New York (geo-location 40.76 (latitude), -73.98(longitude)) who speaks Portugese fluently (he is a native speaker) and his French is on an intermediate level (he has been learning French for 3 years), he understands Spanish and Italian pretty well but cannot speak. Bud is now working on a design specification document for the SingleSignOn subcomponent of the security component. Barbara knows C++ (she was teaching C++ for 2 years) and Java (she was programming in it for 3 years, her focus is on frontend programming, especially in the JSF technology) and she has a project manager experience, Bill knows Java (for 8 years, he is an expert who worked on the JVM of some company too) and Python (3 years, it is his hobby, he loves Python, Java not so much). Barbara worked as a tester for a year in the past. Barbara knows a colleague very well. The Bee project is a large long-term project for a big mobile operator company called PhoneCorp. The Bee project team is located in London - West Kensington, geo-location 51.49 (latitude), -0.220 (longitude), previously it was located in the New York headquarters, and cooperates very well with and manages a testing team based in Bangalore, India. The other people working for the project include Bao and Bert. Bob **maybe** knows Bert.

Text - Revision 6
Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. The project can be extended by up to two years upon a successful review. Bob is not in charge of any other project. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Before joining our company he worked as a consultant for an international company. Bill is an expert programmer, Barbara is a former junior consultant and an experienced programmer with extensive theoretical knowledge (she has a background in theoretical computer science, esp. the theory of complexity), Bud is technical document writer, not a programmer, he is an African-American from New York (geo-location 40.76 (latitude), -73.98(longitude)) who speaks Portugese fluently (he is a native speaker) and his French is on an intermediate level (he has been learning French for 3 years), he understands Spanish and Italian pretty well but cannot speak. Bud is now working on a design specification document for the SingleSignOn subcomponent of the security component. Barbara knows C++ (she was teaching C++ for 2 years) and Java (she was programming in it for 3 years, her focus is on frontend programming, especially in the JSF technology), she has a project manager experience and she does not speak French and does not like Bud **because he's a chauvinist**, Bill knows Java (for 8 years, he is an expert who worked on the JVM of Sun Microsystems too) and Python (3 years, it is his hobby, he loves Python, Java not so much). Barbara worked as a tester for a year in the past. Barbara knows a colleague very well, the colleague is Bob. The Bee project is a large long-term project for a big mobile operator company called PhoneCorp. The Bee project team is located in London - West Kensington, geo-location 51.49 (latitude), -0.220 (longitude), previously it was located in the New York headquarters, and cooperates very well with and manages a testing team based in Bangalore, India. Project management is not the responsibility of Bill. Bill is reachable on Monday, Tuesday, and Wednesday by phone but he can reply to e-mails from Monday to Friday. The other people from the company working for the project include Bao and Bert. Bob maybe knows Bert.

A.3 QUESTIONNAIRES

A.3.1 *After Each Round Questionnaire*



ID:___________________
Formalism: ☐ RDF ☐ Structured Tags

Time you started annotating revision 1: ___________
Time you finished annotating revision 1: ___________

Time you started annotating revision 2: ___________
Time you finished annotating revision 2: ___________

Time you started annotating revision 3: ___________
Time you finished annotating revision 3: ___________

Time you started annotating revision 4: ___________
Time you finished annotating revision 4: ___________

Time you started annotating revision 5: ___________
Time you finished annotating revision 5: ___________

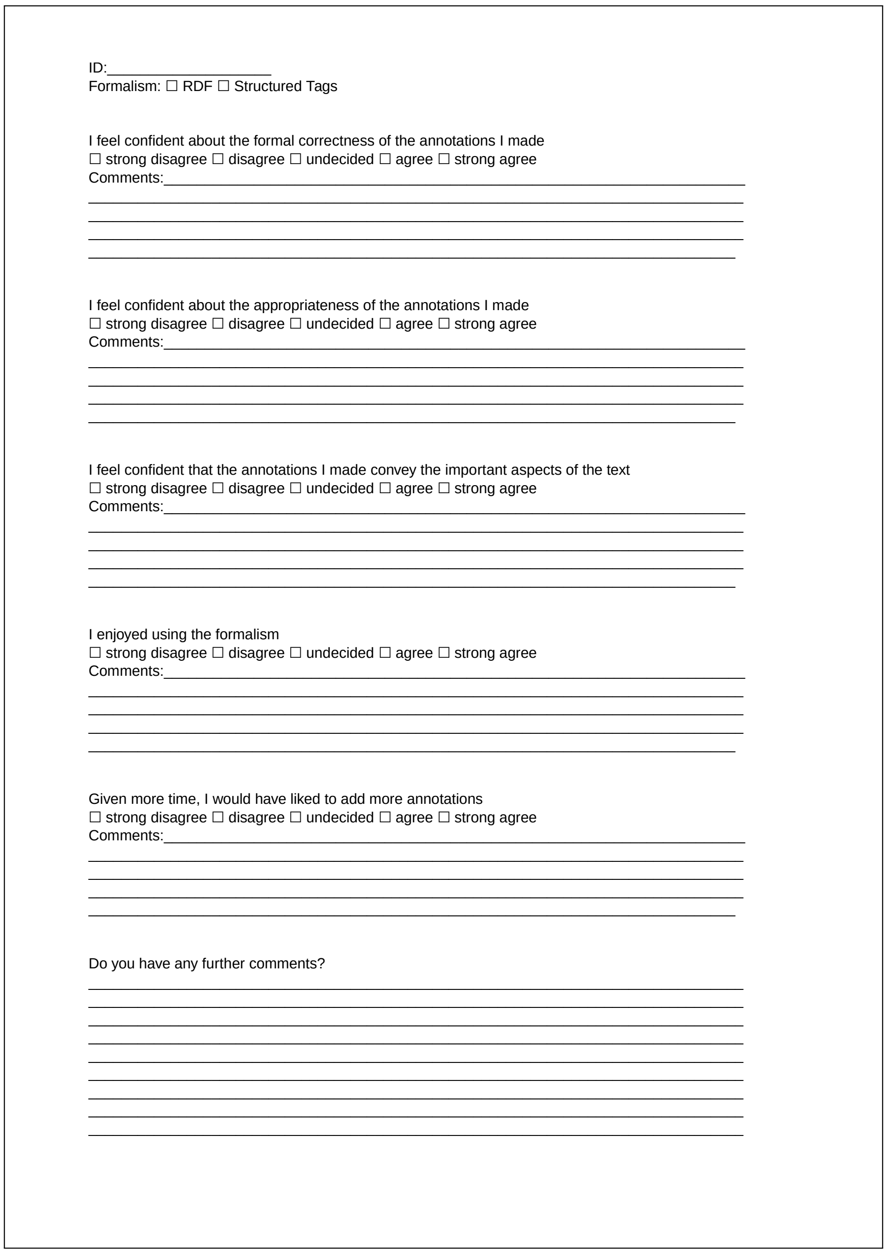
Time you started annotating revision 6: ___________
Time you finished annotating revision 6: ___________

After reading the introductory text, I felt I had understood how to use the annotation formalism
☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree
Comments:

The given annotation formalism allowed to annotate the text in an intuitive way
☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree
Comments:

The given annotation formalism allowed to annotate the text in a convenient way
☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree
Comments:

The given annotation formalism was expressive enough to annotate the text the way I wanted
☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree
Comments:



ID:____________________

Formalism: ☐ RDF ☐ Structured Tags

I feel confident about the formal correctness of the annotations I made

☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree

Comments:____

____

____

____

____

I feel confident about the appropriateness of the annotations I made

☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree

Comments:____

____

____

____

____

I feel confident that the annotations I made convey the important aspects of the text

☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree

Comments:____

____

____

____

____

I enjoyed using the formalism

☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree

Comments:____

____

____

____

____

Given more time, I would have liked to add more annotations

☐ strong disagree ☐ disagree ☐ undecided ☐ agree ☐ strong agree

Comments:____

____

____

____

____

Do you have any further comments?

____

____

____

____

____

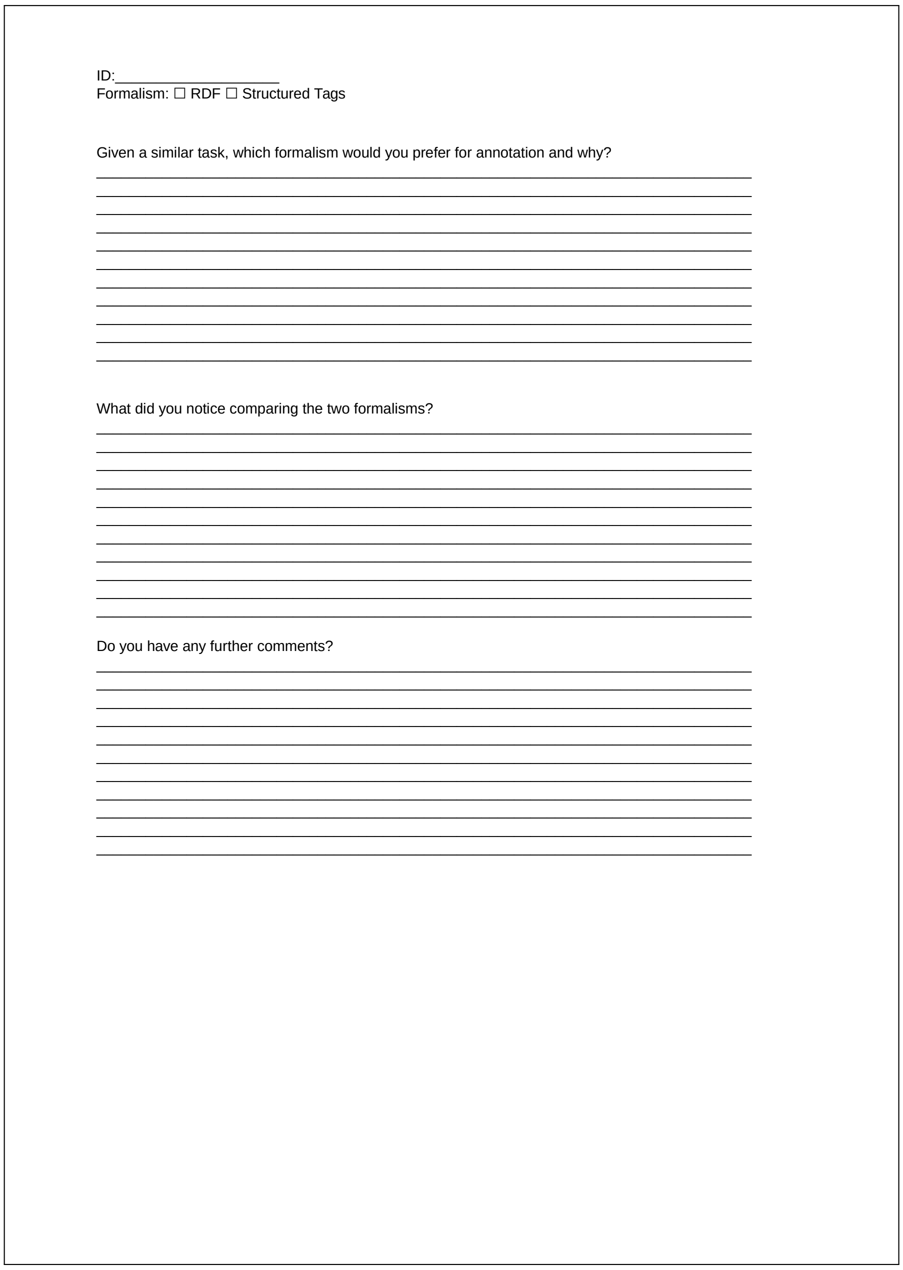
____

____

____

____

A.3.2 *Final Questionnaire*



ID:____________________

Formalism: ☐ RDF ☐ Structured Tags

Given a similar task, which formalism would you prefer for annotation and why?

What did you notice comparing the two formalisms?

Do you have any further comments?

BIBLIOGRAPHY

[1] Web 2.0. http://en.wikipedia.org/wiki/Web_2.0. Retrieved August 16, 2010.

[2] Wikipedia. URL http://en.wikipedia.org/wiki/Wikipedia. Retrieved August 16, 2010.

[3] World Wide Web. http://en.wikipedia.org/wiki/World_Wide_Web. Retrieved April 12, 2011.

[4] Karl Aberer, Philippe C. Mauroux, Aris M. Ouksel, Tiziana Catarci, Mohand S. Hacid, Arantza Illarramendi, Vipul Kashyap, Massimo Mecella, Eduardo Mena, Erich J. Neuhold, and Et Al. Emergent semantics principles and issues. In *DASFAA*, LNCS, pages 25–38. Springer, March 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.6378.

[5] S. Abiteboul. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, August 1991. ISSN 00220000. doi: 10.1016/0022-0000(91)90032-Z. URL http://dx.doi.org/10.1016/0022-0000(91)90032-Z.

[6] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.

[7] Eneko Agirre and German Rigau. Word sense disambiguation using conceptual density. In *Proceedings of the 16th conference on Computational linguistics - Volume 1*, COLING '96, pages 16–22, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics. doi: 10.3115/992628.992635. URL http://dx.doi.org/10.3115/992628.992635.

[8] Carlos E. Alchourron, Peter Gardenfors, and David Makinson. On the logic of theory change: Contraction functions and their associated revision functions. *Theoria*, 48:14–37, 1982.

[9] Carlos E. Alchourron, Peter Gardenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symbolic Logic*, 50:510–530, 1985.

[10] Morgan Ames and Mor Naaman. Why we tag: motivations for annotation in mobile and online media. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 971–980, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240772. URL http://dx.doi.org/10.1145/1240624.1240772.

[11] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, February 2008. ISSN 0360-0300. doi: 10.1145/1322432.1322433. URL http://dx.doi.org/10.1145/1322432.1322433.

[12] K. R. Apt, H. A. Blair, and A. Walker. *Towards a theory of declarative knowledge*, pages 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0-934613-40-0. URL http://portal.acm.org/citation.cfm?id=61352.61354.

[13] S. Auer, S. Dietzold, and T. Riechert. Ontowiki-a tool for social, semantic collaboration. *International Semantic Web Conference*, 4273:736–749, 2006.

[14] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, volume 4825, chapter 52, pages 722–735. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-76297-3. doi: 10.1007/978-3-540-76298-0_52. URL http://dx.doi.org/10.1007/978-3-540-76298-0_52.

[15] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL envelope. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 364–369, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. URL http://portal.acm.org/citation.cfm?id=1642351.

[16] Jie Bao, Li Ding, and James A. Hendler. Knowledge representation and query in semantic mediawiki: A formal study. Technical report, Tetherless World Constellation (RPI), 2009. URL http://www.cs.rpi.edu/~baojie/pub/2009-06-02_iswc-bao_tr.pdf.

[17] J. Bar-Ilan, S. Shoham, A. Idan, Y. Miller, and A. Shachak. Structured versus unstructured tagging: a case study. *Online Information Review*, 32(5):635–647, 2008.

[18] Judit Bar-Ilan, Snunith Shoham, Asher Idan, Yitzchak Miller, and Aviv Shachak. Structured vs. unstructured tagging - a case study. In *Proceedings of the WWW 2006 Collaborative Web Tagging Workshop*, 2006.

[19] Dave Beckett. RDF/XML syntax specification (revised). Technical report, W3C, 2004.

[20] Dave Beckett and Art Barstow. N-Triples. Technical report, W3C, 2001.

[21] David Beckett and Tim Berners-Lee. Turtle - terse RDF triple language. Technical report, W3C, 2006.

[22] V. Richard Benjamins, Jesús Contreras, Oscar Corcho, and Asunción Gómez-pérez. Six challenges for the semantic web. In *In KR2002 Semantic Web Workshop*, volume 1, 2002. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.8902.

[23] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler. N3Logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming*, 8(03):249–269, 2008.

[24] Tim Berners-Lee. Notation 3. Technical report, W3C, 1998.

[25] Tim Berners-Lee. Transcript of Tim Berners-Lee's talk to the LCS 35th Anniversary celebrations. http://www.w3.org/1999/04/13-tbl.html, April 1999.

[26] Tim Berners-Lee. Linked data. *International Journal on Semantic Web and Information Systems*, 4(2), 2006. URL http://www.w3.org/DesignIssues/LinkedData.html.

[27] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):28–37, May 2001.

[28] Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *In Proceedings of the 3rd International Semantic Web User Interaction Workshop*, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.950.

[29] Agnès Blaye and Françoise Bonthoux. Thematic and taxonomic relations in preschoolers: The development of flexibility in categorization choices. *British Journal of Developmental Psychology*, 19(3):395–412, 2001. doi: 10.1348/026151001166173. URL http://dx.doi.org/10.1348/026151001166173.

[30] W. Blizard. Dedekind multisets and function shells. *Theoretical Computer Science*, 110(1):79–98, March 1993. ISSN 03043975. doi: 10.1016/0304-3975(93)90351-S. URL http://dx.doi.org/10.1016/0304-3975(93)90351-S.

[31] Wayne D. Blizard. The development of multiset theory. *The Review of Modern Logic*, 1(4):319–352, 1991.

[32] Amy E. Booth and Sandra Waxman. Object names and object functions serve as cues to categories for infants. *Developmental Psychology*, 38(6), 2002. URL http://www.sciencedirect.com/science/article/pii/S0012164902014985.

[33] Philip Richard Boulain. Swiki: A semantic wiki wiki web. Master's thesis, University of Southampton, 2005.

[34] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, and Henry S. Thompson. Namespaces in XML 1.0 (Third Edition). Technical report, W3C, 2009.

[35] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: 10.1145/343477.343502. URL http://dx.doi.org/10.1145/343477.343502.

[36] Dan Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF schema. Technical report, W3C, 2004.

[37] A. Broder. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Citeseer, 1993.

[38] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. *LNCS*, 2342, 2002.

[39] Jeen Broekstra and Arjohn Kampman. Inferencing and truth maintenance in RDF Schema – exploring a naive practical approach. *Workshop on Practical and Scalable Semantic Systems (PSSS)*, 2003.

[40] Julian Browne. Brewer's CAP Theorem, January 2009. URL http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

[41] Vicki Bruce, Patrick R. Green, and Mark A. Georgeson. *Visual perception: Physiology, psychology, & ecology*. Psychology Press, 2003.

[42] François Bry and Jakub Kotowski. A social vision of knowledge representation and reasoning. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, pages 235–246. Springer-Verlag, 2009.

[43] François Bry, Tim Furche, and Benedikt Linse. Data model and query constructs for versatile web query languages: State-of-the-Art and challenges for xcerpt. In *In Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR*, pages 90–104, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.4497.

[44] François Bry, Benedikt Linse, Tim Furche, Clemens Ley, Thomas Eiter, Norbert Eisinger, Georg Gottlob, Reinhard Pichler, and Fang Wei. Foundations of rule-based query answering. *Springer LNCS 4636*, Reasoning Web, Third International Summer School 2007, 2007.

[45] François Bry, Tim Furche, Clemens Ley, Benedikt Linse, and Bruno Marnette. Taming existence in RDF querying. In Diego Calvanese and Georg Lausen, editors, *Web Reasoning and Rule Systems*, volume 5341 of *Lecture Notes in Computer Science*, chapter 22, pages 236–237. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-88736-2. doi: 10.1007/978-3-540-88737-9_22. URL http://dx.doi.org/10.1007/978-3-540-88737-9_22.

[46] François Bry, Tim Furche, Benedikt Linse, and Alexander Pohl. XcerptRDF: A pattern-based answer to the versatile web challenge. In *Proceedings of 22nd Workshop on (Constraint) Logic Programming, Dresden, Germany (30th September–1st October 2008)*, pages 27–36, 2008. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2008-10.

[47] Michel Buffa and Fabien Gandon. Sweetwiki : Semantic web enabled technologies in wiki. *Proceedings of the international symposium on Symposium on Wikis, ACM Press*, 2006.

[48] Peter Buneman and Wang C. Tan. Provenance in databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1171–1173, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247646. URL http://dx.doi.org/10.1145/1247480.1247646.

[49] C2.com. Wiki engines, . URL http://c2.com/cgi/wiki?WikiEngines. Retrieved August 16, 2010.

[50] C2.com. Elements of wiki essence, . URL http://c2.com/cgi/wiki?ElementsOfWikiEssence. Retrieved August 16, 2010.

[51] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 613–622, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9. doi: 10.1145/1060745.1060835. URL http://dx.doi.org/10.1145/1060745.1060835.

[52] J. Casasnovas and G. Mayor. Discrete t-norms and operations on extended multisets. *Fuzzy Sets and Systems*, 159(10):1165–1177, May 2008. ISSN 01650114. doi: 10.1016/j.fss.2007.12.005. URL http://dx.doi.org/10.1016/j.fss.2007.12.005.

[53] Claudio Castellano, Santo Fortunato, and Vittorio Loreto. Statistical physics of social dynamics. *Reviews of Modern Physics*, 81(2):591–646, May 2009. ISSN 0034-6861. doi: 10.1103/RevModPhys.81.591. URL http://dx.doi.org/10.1103/RevModPhys.81.591.

[54] Ciro Cattuto, Dominik Benz, Andreas Hotho, and Gerd Stumme. Semantic analysis of tag similarity measures in collaborative tagging systems. In *Proceedings of the 3rd Workshop on Ontology Learning and Population (OLP3)*, pages 39–43, Patras, Greece, July 2008. URL http://olp.dfki.de/olp3/.

[55] A. Chandra. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, August 1982. ISSN 00220000. doi: 10.1016/0022-0000(82)90012-5. URL http://dx.doi.org/10.1016/0022-0000(82)90012-5.

[56] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal S. Mumick, and Howard Trickey. Algorithms for deferred view maintenance. *SIGMOD Rec.*, 25:469–480, June 1996. ISSN 0163-5808. doi: 10.1145/235968.233364. URL http://dx.doi.org/10.1145/235968.233364.

[57] J. Conklin. Hypertext: An introduction and survey. *Computer Supported Cooperative Work: A Book of Readings*, pages 423–476, 1988.

[58] Luca Corciulo, Fosca Giannotti, and Dino Pedreschi. Datalog with non-deterministic choice computes NDB-PTIME. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science*, pages 49–66. Springer Berlin / Heidelberg, 1993. doi: 10.1007/3-540-57530-8_4. URL http://dx.doi.org/10.1007/3-540-57530-8_4.

[59] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Science / Engineering / Math, 2nd edition, December 2003. ISBN 0072970545. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0072970545.

[60] Mariano Corso, Antonella Martini, Luisa Pellegrini, and Andrea Pesoli. Emerging approach to e2.0: The case of social enterprise - first results from a 1-year field research. In *The Open Knowlege Society. A Computer Science and Information Systems Manifesto*, Communications in Computer and Information Science, chapter 12, pages 92–100.

Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-87783-7_12. URL http://dx.doi.org/10.1007/978-3-540-87783-7_12.

[61] Ulrike Cress and Joachim Kimmerle. A systemic and cognitive view on collaborative knowledge building with wikis. *International Journal of Computer-Supported Collaborative Learning*, 3(2):105–122, June 2008. ISSN 1556-1607. doi: 10.1007/s11412-007-9035-z. URL http://dx.doi.org/10.1007/s11412-007-9035-z.

[62] Ward Cunningham. What is a wiki. WikiWikiWeb. URL http://wiki.org/wiki.cgi?WhatIsWiki. Retrieved August 16, 2010.

[63] Ilana Davidi. Web 2.0 wiki technology: Enabling technologies, community behaviors, and successful business techniques and models. Master's thesis, Massachusetts Institute of Technology, February 2007.

[64] J. De Kleer. Choices without backtracking. In *Proceedings of AAAI-84*, pages 79–85, 1984.

[65] Johan de Kleer. An Assumption-based TMS. *Artificial Intelligence*, 28, 1986.

[66] Johan de Kleer. A general labeling algorithm for assumption-based truth maintenance. *AAAI 88*, 1988.

[67] Nicholas Del Rio and Paulo da Silva. Probe-It! Visualization Support for Provenance. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Nikos Paragios, Syeda-Mahmood Tanveer, Tao Ju, Zicheng Liu, Sabine Coquillart, Carolina Cruz-Neira, Torsten Müller, and Tom Malzbender, editors, *Advances in Visual Computing*, volume 4842 of *Lecture Notes in Computer Science*, chapter 72, pages 732–741. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-76855-5. doi: 10.1007/978-3-540-76856-2_72. URL http://dx.doi.org/10.1007/978-3-540-76856-2_72.

[68] Alain Désilets, Sébastien Paquet, and Norman G. Vinson. Are wikis usable? In *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*, pages 3–15, New York, NY, USA, 2005. ACM. ISBN 1-59593-111-2. doi: 10.1145/1104973.1104974. URL http://dx.doi.org/10.1145/1104973.1104974.

[69] S. W. Dietrich. Maintenance of Recursive Views. In *Encyclopedia of Database Systems*, pages 1674–1679. Springer Verlag, 2009.

[70] L. Ding, D. DiFranzo, A. Graves, J. R. Michaelis, X. Li, D. L. McGuinness, and J. Hendler. Data-gov wiki: Towards linking government data. In *AAAI Spring Symposium on Linked Data Meets Artificial Intelligence*, 2010.

[71] W. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, October 1984. ISSN 07431066. doi: 10.1016/0743-1066(84)90014-1. URL http://dx.doi.org/10.1016/0743-1066(84)90014-1.

[72] Jon Doyle. Truth maintenance systems for problem solving. Technical Report AI-TR-419, Dep. of Electrical Engineering and Computer Science of MIT, 1978.

[73] Jon Doyle. A Truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[74] Jon Doyle. The ins and outs of reason maintenance. *Proc. IJCAI'83*, 1983.

[75] Jon Doyle. *Reason maintenance and belief revision – Foundations vs. Coherence theories*, pages 29–51. Cambridge University Press, 1992.

[76] Truong Quoc Dung. A new representation of jtms. *LNCS*, 990/1995, 1995.

[77] Truong Quoc Dung. A revision of dependency-directed backtracking for jtms. *LNCS*, 1137, 1996.

[78] Anja Ebersbach, Markus Glaser, and Richard Heigl. *Social Web*. UVK Verlagsgesellschaft mbH, 2008. in German.

[79] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 90–96, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. URL http://portal.acm.org/citation.cfm?id=1642308.

[80] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for Semantic-Web reasoning. In York Sure and John Domingue, editors, *The Semantic Web: Research and Applications*, volume 4011 of *Lecture Notes in Computer Science*, chapter 22, pages 273–287. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-34544-2. doi: 10.1007/11762256_22. URL http://dx.doi.org/10.1007/11762256_22.

[81] John O . Everett and Kenneth D . Forbus. Scaling up logic-based truth maintenance systems via fact garbage collection. *Proc. AAAI-96*, pages 614–620, 1996.

[82] Detlef Fehrer. A Unifying Logical Framework for Reason Maintenance. In *ECSQARU '93: Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 113–120, London, UK, 1993. Springer-Verlag. ISBN 3-540-57395-X. URL http://portal.acm.org/citation.cfm?id=646560.695273.

[83] Giorgos Flouris, Irini Fundulaki, Panagiotis Pediaditis, Yannis Theoharis, and Vassilis Christophides. Coloring RDF Triples to Capture Provenance. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823, chapter 13, pages 196–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04929-3. doi: 10.1007/978-3-642-04930-9_13. URL http://dx.doi.org/10.1007/978-3-642-04930-9_13.

[84] Kenneth D. Forbus. The qualitative process engine, A study in assumption-based truth maintenance. *International Journal for Artificial Intelligence in Engineering*, pages 200–215, 1988.

[85] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1993.

[86] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982. ISSN 00043702. doi: 10.1016/0004-3702(82)90020-0. URL http://dx.doi.org/10.1016/0004-3702(82)90020-0.

[87] Frank Fuchs-Kittowski and André Köhler. Wiki communities in the context of work processes. In *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*, pages 33–39, New York, NY, USA, 2005. ACM. ISBN 1-59593-111-2. doi: 10.1145/1104973.1104977. URL http://dx.doi.org/10.1145/1104973.1104977.

[88] Joe Futrelle. Harvesting RDF Triples. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data*, volume 4145, chapter 8, pages 64–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-46302-3. doi: 10.1007/11890850_8. URL http://dx.doi.org/10.1007/11890850_8.

[89] D. M. Gabbay. *Labelled deductive systems*. Oxford University Press, USA, 1996.

[90] P. Gärdenfors. How to make the semantic web more semantic. In *FOIS*, 2004.

[91] F. Gedikli and D. Jannach. Rating items by rating tags. In *Proc. of the 2nd Workshop Recommender Systems and the Social Web, RSWEB 2010*, 2010.

[92] David M. Geil. Collaborative reasoning: Evidence for collective rationality. *Thinking & Reasoning*, 4(3):231–248, 1998. doi: 10.1080/135467898394148. URL http://dx.doi.org/10.1080/135467898394148.

[93] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://dx.doi.org/10.1145/564585.564601.

[94] B. Glavic and K. Dittrich. Data provenance: A categorization of existing approaches. In *BTW07 - 12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web*, 2007.

[95] Scott Golder and Bernardo A. Huberman. The structure of collaborative tagging systems. *Journal of Information Science*, 32(2):198–208, August 2005. URL http://arxiv.org/abs/cs.DL/0508082.

[96] James W. Goodwin. An improved algorithm for non-monotonic dependency net update. Technical report, Linkoping University, Department of Computer and Information Science, 1982.

[97] James W. Goodwin. *A Theory and System for Non-Monotonic Reasoning*. PhD thesis, Linkoping University, 1987.

[98] Alison Gopnik and Andrew Meltzoff. The development of categorization in the second year and its relation to other cognitive and linguistic developments. *Child Development*, 58(6), 1987. ISSN 00093920. doi: 10.2307/1130692. URL http://dx.doi.org/10.2307/1130692.

[99] Alison Gopnik and Andrew N. Meltzoff. Categorization and naming: Basic-Level sorting in Eighteen-Month-olds and its relation to language. *Child Development*, 63(5):1091–1103, 1992. ISSN 00093920. doi: 10.2307/1131520. URL http://dx.doi.org/10.2307/1131520.

[100] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '1981, pages 144–154. VLDB Endowment, 1981. URL http://portal.acm.org/citation.cfm?id=1286846.

[101] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993. ISSN 1042-8143. doi: 10.1006/knac.1993.1008. URL http://dx.doi.org/10.1006/knac.1993.1008.

[102] A. Guessoum and J. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, June 1990. ISSN 0288-3635. doi: 10.1007/BF03037514. URL http://dx.doi.org/10.1007/BF03037514.

[103] Ashish Gupta and Inderpal S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Data Engineering Bulletin*, 18(2):3–18, 1995. URL http://portal.acm.org/citation.cfm?id=310737.

[104] Ashish Gupta, Dinesh Katiyar, and Inderpal S. Mumick. Counting solutions to the View Maintenance Problem. In *In Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8990.

[105] Ashish Gupta, Inderpal S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *SIGMOD Rec.*, 22:157–166, June 1993. ISSN 0163-5808. doi: 10.1145/170035.170066. URL http://dx.doi.org/10.1145/170035.170066.

[106] Mandy Haggith. Disagreement in creative problem solving. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1993.

[107] Harry Halpin. The semantic web: The origins of artificial intelligence redux. In *HPLMC-04*, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.6799.

[108] Harry Halpin, Valentin Robu, and Hana Shepherd. The complex dynamics of collaborative tagging. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 211–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242602. URL http://dx.doi.org/10.1145/1242572.1242602.

[109] Eric N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 440–453, New York, NY, USA, 1987. ACM. ISBN 0-89791-236-5. doi: 10.1145/38713.38759. URL http://dx.doi.org/10.1145/38713.38759.

[110] John V. Harrison and Suzanne W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Workshop on Deductive Databases, JICSLP*, pages 56–65, 1992.

[111] Bernhard Haslhofer, Elaheh M. Roochi, Bernhard Schandl, and Stefan Zander. Europeana RDF store report. Technical report, University of Vienna, Vienna, March 2011. URL http://eprints.cs.univie.ac.at/2833/.

[112] Patrick Hayes. RDF semantics. Technical report, W3C, 2004.

[113] Patrick J. Hayes. The frame problem and related problems in artificial intelligence. Technical report, Stanford University, Stanford, CA, USA, 1971.

[114] Martin Hepp. Possible ontologies: How reality constrains the development of relevant ontologies. *IEEE Internet Computing*, 11(1):90–96, 2007. ISSN 1089-7801. doi: 10.1109/MIC.2007.20. URL http://dx.doi.org/10.1109/MIC.2007.20.

[115] J. L. Hickman. A note on the concept of multiset. *Bulletin of the Australian Mathematical Society*, 22(02):211–217, 1980. doi: 10.1017/S000497270000650X. URL http://dx.doi.org/10.1017/S000497270000650X.

[116] A. Hogan, A. Harth, and A. Polleres. Scalable authoritative owl reasoning for the web. *International Journal on Semantic Web and Information Systems*, 5(2), 2009.

[117] Aidan Hogan and Stefan Decker. On the ostensibly silent 'w' in OWL 2 RL. In Axel Polleres and Terrance Swift, editors, *Web Reasoning and Rule Systems*, volume 5837, chapter 9, pages 118–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-05081-7. doi: 10.1007/978-3-642-05082-4_9. URL http://dx.doi.org/10.1007/978-3-642-05082-4_9.

[118] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1-3):86–104, January 1986. ISSN 00199958. doi: 10.1016/S0019-9958(86)80029-8. URL http://dx.doi.org/10.1016/S0019-9958(86)80029-8.

[119] ISO/IEC. ISO/IEC 14977:1996(E) First edition – Information technology – Syntactic metalanguage – Extended BNF. Technical report, ISO/IEC, 1996.

[120] Kensaku Kawamoto, Yasuhiko Kitamura, and Yuri Tijerino. Kawawiki: A template-based semantic wiki where end and expert users collaborate. In *Proceedings of 5th International Semantic Web Conference (ISWC2006)*, 2006.

[121] Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. Technical report, W3C, 2004.

[122] André Köhler and Frank Fuchs-Kittowski. Integration of communities into process-oriented structures. *Journal of Universal Computer Science*, 11(3):410–425, 2005. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.98.7209.

[123] Robert Koons. Defeasible reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2005. URL http://plato.stanford.edu/archives/spr2005/entries/reasoning-defeasible/.

[124] Markus Krötzsch, Denny Vrandečić, and Max Völkel. Semantic mediawiki. In *The Semantic Web - ISWC 2006*, volume 4273, chapter 68, pages 935–942. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-49029-6. doi: 10.1007/11926078_68. URL http://dx.doi.org/10.1007/11926078_68.

[125] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Complexity boundaries for horn description logics. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, pages 452–457. AAAI Press, 2007. ISBN 978-1-57735-323-2. URL http://portal.acm.org/citation.cfm?id=1619718.

[126] Markus Krötzsch, Sebastian Schaffert, and Denny Vrandecic. Reasoning in semantic wikis. In *Reasoning Web Summer School 2007*, Lecture Notes in Computer Science, pages 310–329. Springer Berlin / Heidelberg, 2007.

[127] Clif Kussmaul and Roger Jack. Wikis for knowledge management: Business cases, best practices, promises, & pitfalls. In *Web 2.0*, chapter 9, pages 1–19. Springer US, 2009. doi: 10.1007/978-0-387-85895-1_9. URL http://dx.doi.org/10.1007/978-0-387-85895-1_9.

[128] T. Kvan. Designing collaborative environments for strategic knowledge in design. *Knowledge-Based Systems*, 13(6):429–438, November 2000. ISSN 09507051. doi: 10.1016/S0950-7051(00)00083-6. URL http://dx.doi.org/10.1016/S0950-7051(00)00083-6.

[129] Brian Lamb. Wide open spaces: Wikis ready or not. *EDUCAUSE Review*, 39(5), 2004. URL http://www.eric.ed.gov/ERICWebPortal/detail?accno=EJ710632.

[130] Tim B. Lee. How It All Started: Pre-W3C Web and Internet Background. http://www.w3.org/2004/Talks/w3c10-HowItAllStarted/?n=13, .

[131] Tim B. Lee. WorldWideWeb: Proposal for a HyperText Project. http://www.w3.org/Proposal.html, .

[132] Tim B. Lee, Robert Cailliau, Ari Luotonen, Henrik F. Nielsen, and Arthur Secret. The World-Wide web. *Commun. ACM*, 37(8):76–82, August 1994. ISSN 0001-0782. doi: 10.1145/179606.179671. URL http://dx.doi.org/10.1145/179606.179671.

[133] Bo Leuf and Ward Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley Professional, April 2001. ISBN 020171499X. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/020171499X.

[134] Benedikt Linse and Andreas Schroeder. Beyond XML and RDF: the versatile web query language xcerpt. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 1053–1054, New York, NY, USA, 2006. ACM. ISBN 1595933239. doi: 10.1145/1135777.1136011. URL http://dx.doi.org/10.1145/1135777.1136011.

[135] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer Verlag, 2009. ISBN 978-0-387-35544-3.

[136] J. Lloyd. *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1987.

[137] C. Fergus Lowe, Pauline J. Horne, and J. Carl Hughes. Naming and categorization in young children: III. vocal tact training and transfer of function. *Journal of the experimental analysis of behavior*, 83(1):47–65, January 2005. ISSN 0022-5002. doi: 10.1901/jeab.2005.31-04. URL http://dx.doi.org/10.1901/jeab.2005.31-04.

[138] M. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In *North American Conference on Logic Programming*, 1989.

[139] Frank Manola and Eric Miller. RDF primer. Technical report, W3C, 2004.

[140] Draltan Marin. A formalization of RDF (Applications de la logique á la sémantique du Web). Technical report, Ecole Polytechnique – Universidad de Chile, 2004.

[141] Cameron Marlow, Mor Naaman, Danah Boyd, and Marc Davis. HT06, tagging paper, taxonomy, flickr, academic article, to read. In *Proceedings of the seventeenth conference on Hypertext and hypermedia*, HYPERTEXT '06, pages 31–40, New York, NY, USA, 2006. ACM. ISBN 1-59593-417-0. doi: 10.1145/1149941.1149949. URL http://dx.doi.org/10.1145/1149941.1149949.

[142] João P. Martins. The Truth, the Whole Truth, and Nothing But the Truth: An Indexed Bibliography to the Literature of Truth Maintenance Systems. *AI Magazine*, 11, 1990.

[143] Joao P. Martins and Stuart C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35, 1988.

[144] A. P. Mcafee. Enterprise 2.0: the dawn of emergent collaboration. *Engineering Management Review, IEEE*, 34(3), 2006. doi: 10.1109/EMR.2006.261380. URL http://dx.doi.org/10.1109/EMR.2006.261380.

[145] D. A. McAllester. Reasoning utility package user's manual, version one. Technical report, Massachusetts Institute of Technology - Artificial Intelligence Laboratory, 1982. URL http://dspace.mit.edu/handle/1721.1/5683.

[146] David McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002. ISSN 0004-5411. doi: 10.1145/581771.581774. URL http://dx.doi.org/10.1145/581771.581774.

[147] David A. McAllester. A three valued truth maintenance system. *AI-Memorandum rept., MIT*, 1978.

[148] David A. McAllester. An outlook on truth maintenance. Technical report, MIT, 1980.

[149] David A. McAllester. Truth maintenance. *AAAI90*, 1990.

[150] Daniel D. McCracken and Edwin D. Reilly. Backus-Naur form (BNF). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK, 2003. ISBN 0-470-86412-5. URL http://portal.acm.org/citation.cfm?id=1074155.

[151] Deborah L. Mcguinness and Paulo P. da Silva. Explaining answers from the semantic web: the Inference Web approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1, 2004.

[152] Erik Meijer and Gavin Bierman. A Co-Relational model of data for large shared data banks. *Communications of the ACM*, 54(4):49–58, 2011.

[153] Andrew B. Mickel, James F. Miner, Kathleen Jensen, and Niklaus Wirth. *Pascal user manual and report (4th ed.): ISO Pascal standard*. Springer-Verlag New York, Inc., New York, NY, USA, 1991. ISBN 0-387-97649-3. URL http://portal.acm.org/citation.cfm?id=116409.

[154] Peter Mika. Ontologies are us: A unified model of social networks and semantics. In Yolanda Gil, Enrico Motta, V. Benjamins, and Mark Musen, editors, *The Semantic Web - ISWC 2005*, volume 3729 of *Lecture Notes in Computer Science*, chapter 38, pages 522–536. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-29754-3. doi: 10.1007/11574620_38. URL http://dx.doi.org/10.1007/11574620_38.

[155] GP Monro. The concept of multiset. *Mathematical Logic Quarterly*, 33 (2):171–178, 1987. ISSN 1521-3870.

[156] D. Montesi, E. Bertino, and M. Martelli. Transactions and updates in deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):784–797, September 1997. ISSN 10414347. doi: 10.1109/69.634755. URL http://dx.doi.org/10.1109/69.634755.

[157] Boris Motik, Bernardo C. Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 web ontology language – profiles. Technical report, W3C, 2009.

[158] Inderpal S. Mumick. *Query Optimization in Deductive and Relational Databases*. PhD thesis, Stanford University, 1991.

[159] Inderpal S. Mumick and Oded Shmueli. Finiteness Properties of Database Queries. *Fourth Australian Database Conference*, 1993. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.9206.

[160] Sergio Munoz, Jorge Perez, and Claudio Gutierrez. Minimal deductive systems for RDF. *LNCS*, 4519, 2007.

[161] William J. Murdock, Deborah L. Mcguinness, Paulo P. da Silva, Chris Welty, and David Ferrucci. Explaining Conclusions from Diverse Knowledge Sources. *ISWC 2006, LNCS*, 4273, 2006.

[162] Pandurang P. Nayak and Brian C. Williams. Fast context switching in real-time propositional reasoning. *Proc. 14th Nat. Conf. AI*, 1997.

[163] T. Nazzi and A. Gopnik. Linguistic and cognitive abilities in infancy: when does language become a tool for categorization? *Cognition*, 80 (3):B11–B20, July 2001. ISSN 00100277. doi: 10.1016/S0010-0277(01) 00112-3. URL http://dx.doi.org/10.1016/S0010-0277(01)00112-3.

[164] Bernhard Nebel. *Belief revision*, volume 422 of *Lecture Notes in Computer Science*, chapter 6, pages 149–186. Springer-Verlag, Berlin/Heidelberg, 1990. ISBN 3-540-52443-6. doi: 10.1007/BFb0016451. URL http://dx.doi.org/10.1007/BFb0016451.

[165] Bernhard Nebel. *Reasoning and revision in hybrid representation systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1990. ISBN 0-387-52443-6. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.1537.

[166] Ted Nelson. *Literary Machines*. Mindful Press, 1981.

[167] Vivek Nigam, Limin Jia, Boon T. Loo, and Andre Scedrov. Maintaining distributed logic programs incrementally. In *Proc. 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, Odense, Denmark, July 2011.

[168] Vivek Nigam, Limin Jia, Boon T. Loo, and Andre Scedrov. Maintaining distributed recursive views incrementally. Technical report, University of Pennsylvania, 2011.

[169] Zheng Y. Niu, Dong H. Ji, and Chew L. Tan. Word sense disambiguation using label propagation based semi-supervised learning. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 395–402, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics. doi: 10.3115/1219840.1219889. URL http://dx.doi.org/10.3115/1219840.1219889.

[170] C.K. Ogden, I.A. Richards, B. Malinowski, and F.G. Crookshank. *The Meaning of Meaning: A Study of the Influence of Language upon Thought and of the Science of Symbolism*. Harcourt, Brace & Company, 1938.

[171] Tim O'Reilly. What is web 2.0. design patterns and business models for the next generation of software. http://oreilly.com/web2/archive/what-is-web-20.html, 2005.

[172] E. Oren. Semantic Wikis for Knowledge Workers. *WWW2006, poster*, 2006. URL http://library.deri.ie/resource/499ixefG.

[173] Eyal Oren. Semperwiki: a semantic personal wiki. In *Proceedings of 1st Workshop on The Semantic Desktop - Next Generation Personal Information Management and Collaboration Infrastructure*, Galway, Ireland, 2005.

[174] Eyal Oren. Semperwiki: a semantic personal wiki. In *Proceedings of 1st Workshop on The Semantic Desktop - Next Generation Personal Information Management and Collaboration Infrastructure*, Galway, Ireland, 2005.

[175] Peter F. Patel Schneider and Ian Horrocks. Position paper: a comparison of two modelling paradigms in the semantic web. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 3–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-323-9. doi: 10.1145/1135777.1135784. URL http://dx.doi.org/10.1145/1135777.1135784.

[176] P. Pediaditis, G. Flouris, I. Fundulaki, and V. Christophides. On explicit provenance management in RDF/S graphs. In *TAPP'09: First workshop on on Theory and practice of provenance*, pages 1–10, Berkeley, CA, USA, 2009. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1525936.

[177] C.S. Peirce. On a new list of categories. In *Proceedings of the American Academy of Arts and Sciences*, volume 7, pages 287–298, 1868.

[178] Isabella Peters and Wolfgang G. Stock. Folksonomy and information retrieval. *Proc. Am. Soc. Info. Sci. Tech.*, 44(1):1–28, 2007. ISSN 1550-8390. doi: 10.1002/meet.1450440226. URL http://dx.doi.org/10.1002/meet.1450440226.

[179] John L. Pollock. Defeasible reasoning. *Cognitive Science*, 11:481–518, October 1987. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.111.

[180] Niko Popitsch, Bernhard Schandl, Arash Amiri, Stefan Leitich, and Wolfgang Jochum. Ylvi - multimedia-izing the semantic wiki. In *Proceedings of the 1st Workshop "SemWiki2006 - From Wiki to Semantics"*, Budva, Montenegro, 2006.

[181] B. Popov, A. Kiryakov, D. Manov, A. Kirilov, D. Ognyanoff, and M. Goranov. Towards semantic web information extraction. In *Workshop on Human Language Technology for the Semantic Web and Web Services*, 2003.

[182] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF – W3C recommendation. Technical report, W3C, 2008.

[183] T. C. Przymusinski. *On the declarative semantics of deductive databases and logic programs*, pages 193–216. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0-934613-40-0. URL http://portal.acm.org/citation.cfm?id=61352.61357.

[184] Michael O. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University., 1981.

[185] Richard Rado. The cardinal module and some theorems on families of sets. *Annali di Matematica Pura ed Applicata*, 102(1):135–154, December 1975. ISSN 0373-3114. doi: 10.1007/BF02410602. URL http://dx.doi.org/10.1007/BF02410602.

[186] Ruth Raitman and Naomi Augar. Employing wikis for online collaboration in the e-learning environment: Case study. In *ICITA '05: Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2*, volume 2, pages 142–146, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2316-1. doi: 10.1109/ICITA.2005.127. URL http://dx.doi.org/10.1109/ICITA.2005.127.

[187] R.E. Raygan and D.G. Green. Internet collaboration: Twiki. In *SoutheastCon, 2002. Proceedings IEEE*, pages 137–141. IEEE, 2002.

[188] Lawrence Reeve and Hyoil Han. Survey of semantic annotation platforms. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1634–1638, New York, NY, USA, 2005. ACM. ISBN 1-58113-964-0. doi: 10.1145/1066677.1067049. URL http://dx.doi.org/10.1145/1066677.1067049.

[189] Marko A. Rodriguez and Peter Neubauer. The graph traversal pattern. *arXiv*, April 2010. URL http://arxiv.org/abs/1004.1001.

[190] Eleanor Rosch, Carolyn B. Mervis, Wayne D. Gray, David M. Johnson, and Penny Boyes-Braem. Basic objects in natural categories. *Cognitive Psychology*, 8(3):382–439, July 1976. ISSN 00100285. doi: 10.1016/0010-0285(76)90013-X. URL http://dx.doi.org/10.1016/0010-0285(76)90013-X.

[191] Eleanor H. Rosch. Natural categories. *Cognitive Psychology*, 4(3):328–350, May 1973. ISSN 00100285. doi: 10.1016/0010-0285(73)90017-0. URL http://dx.doi.org/10.1016/0010-0285(73)90017-0.

[192] Sherif Sakr and Ghazi Al-Naymat. Graph indexing and querying: a review. *International Journal of Web Information Systems*, 6(2), 2010. URL http://www.emeraldinsight.com/journals.htm?articleid=1864789&#38;show=abstract.

[193] F. Sani, J. Todman, and J.B. Todman. *Experimental design and statistics for psychology: a first course*. Wiley-Blackwell, 2006.

[194] Jennifer T. Santelli, Michael J. Muller, and David R. Millen. Social tagging roles: publishers, evangelists, leaders. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1041–1044, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357215. URL http://dx.doi.org/10.1145/1357054.1357215.

[195] F. Saussure. Course in general linguistics (W. Baskin, Trans.). *New York: Philosophical Library*, 1916.

[196] S. Schaffert. IkeWiki: A semantic wiki for collaborative knowledge management. In *1st International Workshop on Semantic Technologies in Collaborative Applications (STICA'06), Manchester, UK*, 2006.

[197] Sebastian Schaffert, Rupert Westenthaler, and Andreas Gruber. Ikewiki: A user-friendly semantic wiki. In *3rd European Semantic Web Conference (ESWC06)*, Budva, Montenegro, 2006.

[198] Sebastian Schaffert, François Bry, Joachim Baumeister, and Malte Kiesel. Semantic wikis. *IEEE Software*, 25, 2008.

[199] Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph Pinkel. An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In Amit Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy Finin, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2008*, volume 5318 of *Lecture Notes in Computer Science*, chapter 6, pages 82–97. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-88563-4. doi: 10.1007/978-3-540-88564-1_6. URL http://dx.doi.org/10.1007/978-3-540-88564-1_6.

[200] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 21(3):96–101, 2006. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1637364.

[201] Nematollaah Shiri and Zhi H. Zheng. Challenges in Fixpoint Computation with Multisets. In Dietmar Seipel and a, editors, *Foundations of Information and Knowledge Systems*, volume 2942 of *Lecture Notes in Computer Science*, chapter 18, pages 273–290. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-20965-2. doi: 10.1007/978-3-540-24627-5_18. URL http://dx.doi.org/10.1007/978-3-540-24627-5_18.

[202] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance Techniques. Technical report, Computer Science Department, Indiana University, Bloomington IN, 2005. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.6294.

[203] Michael Sintek and Stefan Decker. TRIPLE - a query, inference, and transformation language for the semantic web. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 364–378, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43760-6. URL http://portal.acm.org/citation.cfm?id=646996.711416.

[204] Gene Smith. *Tagging: People-powered Metadata for the Social Web (Voices That Matter)*. New Riders Press, December 2007. ISBN 0321529170. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&amp;path=ASIN/0321529170.

[205] Rania Soussi, Marie-Aude Aufaure, and Hajer Baazaoui. Towards social network extraction using a graph database. *Advances in Databases, First International Conference on*, 0:28–34, 2010. doi: http://doi.ieeecomputersociety.org/10.1109/DBKDA.2010.19. URL http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/DBKDA.2010.19.

[206] Mladen Stanojevic, Sanja Vranes, and Dusan Velasevic. Using truth maintenance systems - a tutorial. *IEEE*, 1994.

[207] Martin Staudt and Matthias Jarke. Incremental Maintenance of Externally Materialized Views. In *Proc. 22th Int. Conf. VLDB*, San Francisco, CA, USA, 1996. ISBN 1-55860-382-4.

[208] Apostolos Syropoulos. Mathematics of multisets. In Cristian Calude, Gheorghe PAun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, chapter 17, pages 347–358. Springer Berlin / Heidelberg, Berlin, Heidelberg, December 2001. ISBN 978-3-540-43063-6. doi: 10.1007/3-540-45523-X_17. URL http://dx.doi.org/10.1007/3-540-45523-X_17.

[209] G. Thomas. Ontology of folksonomy: A mashup of apples and oranges. *Intl Journal on Semanic Web and Information Systems*, 2007.

[210] Robert Tolksdorf and Elena Paslaru Bontas Simperl. Towards wikis as semantic hypermedia. In *Proceedings of the 2006 international symposium on Wikis*, WikiSym '06, pages 79–88, New York, NY, USA, 2006. ACM. ISBN 1-59593-413-8. doi: 10.1145/1149453.1149470. URL http://dx.doi.org/10.1145/1149453.1149470.

[211] Jennifer Trant. Studying social tagging and folksonomy: A review and framework. *Journal of Digital Information*, 2009. URL http://hdl.handle.net/10150/105375.

[212] G. Trentin. Using a wiki to evaluate individual contribution to a collaborative learning project. *Journal of Computer Assisted Learning*, 0(0): 43–55, February 2009. ISSN 0266-4909. doi: 10.1111/j.1365-2729.2008.00276.x. URL http://dx.doi.org/10.1111/j.1365-2729.2008.00276.x.

[213] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable distributed reasoning using mapreduce. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823, chapter 40, pages 634–649. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04929-3. doi: 10.1007/978-3-642-04930-9_40. URL http://dx.doi.org/10.1007/978-3-642-04930-9_40.

[214] Alvaro D. Val. On the relation between the coherence and foundations theories of belief revision. *AAAI94*, 1994.

[215] Céline Van Damme, Martin Hepp, and Katharina Siorpaes. FolksOntology: An integrated approach for turning folksonomies into ontologies. In *Bridging the Gep between Semantic Web and Web 2.0 (SemNet 2007)*, pages 57–70, 2007. URL http://www.kde.cs.uni-kassel.de/ws/eswc2007/proc/FolksOntology.pdf.

[216] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0064-3. doi: 10.1145/1900008.1900067. URL http://dx.doi.org/10.1145/1900008.1900067.

[217] Max Völkel, Markus Krötzsch, Denny Vrandecic, Heiko Haller, and Rudi Studer. Semantic wikipedia. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*,

pages 585–594, New York, NY, USA, 2006. ACM. ISBN 1-59593-323-9. doi: http://doi.acm.org/10.1145/1135777.1135863. URL http://portal.acm.org/ft_gateway.cfm?id=1135863&type=pdf&coll=GUIDE&dl=GUIDE&CFID=64724004&CFTOKEN=29620398.

[218] Max Völkel, Markus Krötzsch, Denny Vrandecic, Heiko Haller, and Rudi Studer. Semantic wikipedia. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 585–594, New York, NY, USA, 2006. ACM. ISBN 1-59593-323-9. doi: 10.1145/1135777.1135863. URL http://dx.doi.org/10.1145/1135777.1135863.

[219] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. In Stefano Spaccapietra, Elisa Bertino, Sushil Jajodia, Roger King, Dennis McLeod, Maria E. Orlowska, and Leon Strous, editors, *Journal on Data Semantics II*, volume 3360 of *Lecture Notes in Computer Science*, chapter 1, pages 1–34. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-24208-6. doi: 10.1007/978-3-540-30567-5_1. URL http://dx.doi.org/10.1007/978-3-540-30567-5_1.

[220] W3C. OWL 2 web ontology language. Technical report, W3C, 2009.

[221] E. Watkins and Denis Nicole. Named Graphs as a Mechanism for Reasoning About Provenance. In Xiaofang Zhou, Jianzhong Li, Heng T. Shen, Masaru Kitsuregawa, and Yanchun Zhang, editors, *Frontiers of WWW Research and Development - APWeb 2006*, volume 3841, chapter 99, pages 943–948. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-31142-3. doi: 10.1007/11610113_99. URL http://dx.doi.org/10.1007/11610113_99.

[222] Jesse Weaver and James Hendler. Parallel materialization of the finite rdfs closure for hundreds of millions of triples. In Abraham Bernstein, David Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, chapter 43, pages 682–697. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04929-3. doi: 10.1007/978-3-642-04930-9_43. URL http://dx.doi.org/10.1007/978-3-642-04930-9_43.

[223] Klara Weiand. *Keyword-based querying for the social semantic web – the KWQL language - concept, algorithm, and system*. PhD thesis, December 2010.

[224] J. B. Wells and Boris Yakobowski. Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis. In Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation*, volume 3573 of *Lecture Notes in Computer Science*, chapter 17, pages 262–277. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-26655-6. doi: 10.1007/11506676_17. URL http://dx.doi.org/10.1007/11506676_17.

[225] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. *Proceedings of AAAI-96*, 1996.

[226] L. Wittgenstein and G. E. M. Anscombe. *Philosophical investigations: the German text, with a revised English translation*. Wiley-Blackwell, 2001.

[227] Guofu Wu and George Macleod Coghill. A propositional root antecedent itms. *Proceedings of the 15th International Workshop on Principles of Diagnosis*, 2004.

[228] Jakub Yaghob and Filip Zavoral. Semantic web infrastructure using DataPile. In *Web Intelligence and Intelligent Agent Technology Workshops*, pages 630–633, December 2006. doi: 10.1109/WI-IATW.2006.119. URL http://dx.doi.org/10.1109/WI-IATW.2006.119.

[229] Jie Yang, Yutaka Matsuo, and Mitsuru Ishizuka. Triple tagging: Toward bridging folksonomy and semantic web. In *Proceedings of ISWC07*, 2007.

[230] Ka P. Yee, Kirsten Swearingen, Kevin Li, and Marti Hearst. Faceted metadata for image search and browsing. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '03, pages 401–408, New York, NY, USA, 2003. ACM. ISBN 1-58113-630-7. doi: 10.1145/642611.642681. URL http://dx.doi.org/10.1145/642611.642681.

[231] Shen Yidong, Tong Fu, and Cheng Daijie. On local stratifiability of logic programs and databases. *Journal of Computer Science and Technology*, 8:97–107, 1993. doi: 10.1007/BF02939472. URL http://dx.doi.org/10.1007/BF02939472.

[232] Yong Zhao and Shiyong Lu. A Logic Programming Approach to Scientific Workflow Provenance Querying. In *Provenance and Annotation of Data and Processes*, Lecture Notes in Computer Science, chapter 5, pages 31–44. Springer Berlin / Heidelberg, 2008. doi: 10.1007/978-3-540-89965-5_5. URL http://dx.doi.org/10.1007/978-3-540-89965-5_5.