AALTO UNIVERSITY SCHOOL OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Vesa Luukkala

Rule-based Metaprogramming for Smart Spaces

Licentiate's thesis submitted in partial fulfillment of the requirements for the degree of Licentiate of Science in Technology

Espoo, 19th April 2015

Supervisor:	Prof. Ilkka Niemelä
Instructor:	Tomi Janhunen, Docent

Aalto University School of Science Department of Computer Science			ABSTRACT OF LICENCIATE'S THESIS
Author	Vesa Luukkala	Date	19th April 2015
		Pages	vi + 112
Title of thesis	Rule-based Metaprogramming for Smart Spaces		
Professorship	Theoretical Computer Science	Code	T-119
Supervisor	Prof. Ilkka Niemelä		
Instructor	Tomi Janhunen, Docent		

The motivation of this work is goes back to the objective of achieving interoperability in multiparty environments such as ubiquitous systems. Full interoperability in an open environment requires mutually sharing the behavior of the participants, so that the behavioral interoperability becomes as relevant as interoperability of data. This requires analysis or evaluation of behavioral descriptions from untrusted parties in a controlled manner. Furthermore we need to manage the evaluation process based on the content and provenance of the descriptions and other information on which the descriptions operate. This information allows one to choose which behaviour is to be used and which data is to be operated on. To enable this vision we propose to present behavioral descriptions as Answer Set Programming (ASP) rules. In this work we present a method for the evaluation of ASP rules based on metaprogramming: the evaluator for the rules is implemented using ASP rules themselves. To facilitate metaevaluation, we transform rules to a reified format, which enables representing rules as facts, and construct the metaevaluator to work directly on this reified format. Facts corresponding to reified rules and the metaevaluation rules are then treated by native ASP tools. We give a proof that our metaevaluator adheres to the stable model semantics for ASP evaluation. Having rules in the reified format is beneficial as behavioral rules can then be shared and manipulated as any other data. We have implemented a mechanism which maintains the provenance information of data during the rule evaluation along with hooks to allow control over the context of the use of that data. This allows attaching arbitrary metainformation to rules and facts and allows independently creating policies which control on how different data is handled in the ASP solving phase. In addition to the metaevaluation phase, we have implemented syntactical safety analysis of reified rules. These methods enable sharing, analyzing and executing behavioral descriptions in a controlled manner within the same semantic ASP framework, providing one solution for the interoperability problem. The evaluation of ASP rules has two logical phases: grounding and actual solving. We have separated provenance handling and syntactic analysis to the metagrounding phase with the intention that rules and data, which do not match the provenance criteria, are never delivered to the solving phase. To the best of our knowledge, this work presents the first implementation of a metagrounder for ASP programs. According to performance analysis, the metagrounder is not yet competitive with current grounder technology.

Keywords

Answer Set Programming, Metaevaluation, Grounding, Ubiquitous systems, Smart Spaces, Semantic Web, Rules, Logic Programming, Knowledge Representation, Semantics, Stable models

Aalto-yliopisto Perustieteiden korkeakoulu
Tietotekniikan laitos

LISENSIAATTITYÖN
TIIVISTELMÄ

Tekijä	Vesa Luukkala	Päiväys	19. huhtikuuta 2015
		Sivumäärä	vi + 112
Työn nimi	Rule-based Metaprogramming for Smart Spaces		
Professuuri	Tietojenkäsittelyteoria	Koodi	T-119
Työn valvoja	Prof. Ilkka Niemelä		
Työn ohjaaja	Dosentti Tomi Janhunen		

Tämän opinnäytteen motivaationa on yhteensopivuus ubiikkien järjestelmien kaltaisissa usean käyttäjän ympäristöissä. Täydellinen yhteensopivuus avoimissa ympäristöissä vaatii osapuolten käyttäytymisten kuvausten jakamista käyttäjien kesken. Tällöin käyttäytymisen kuvausten yhteensopivuus muodostuu yhtä tärkeäksi kuin muun tiedon yhteensopivuus. Tästä johtuen on tarpeellista analysoida tai evaluoida hallitusti käyttäytymisten kuvauksia, jotka ovat peräisin mahdollisesti epäluotettavilta tahoilta. Tämän lisäksi evaluointiprosessia täytyy hallinnoida perustuen sekä käyttäytymisten kuvausten että muun käytetyn tiedon sisältöön ja alkuperään. Tämän tiedon avulla valitaan mitä käyttäytymiskuvauksia ja mitä tietoa tullaan käyttämään evaluoinnissa. Tämän vision mahdollistamiseksi tässä työssä ehdotetaan käyttäytymiskuvausten esitettämistä sääntöpohjaisella rajoiteohjelmoinnilla (engl. Answer Set Programming, ASP). Tässä opinnäytteessä kuvataan metaohjelmointipohjainen menetelmä sääntöjen evaluoimiseen, missä itse evaluaattori on toteutettu ASP-säännöillä. Jotta metaevaluaatio olisi mahdollista, säännöt muunnetaan reifioituun muotoon, joka sallii sääntöjen esittämisen faktoina ja metaevaluaattori toteutetaan toimimaan näiden reifioitujen kuvausten kanssa. Faktoina esitetyt reifioidut säännöt ja metaevaluaattorin säännöt evaluoidaan olemassaolevilla ASP-työkaluilla. Työssä esitetään oikeellisuustodistus, jonka perusteella toteutettu metaevaluaattori noudattaa stabiilien mallien semantiikkaa. Sääntöjen esittäminen reifioidussa muodossa on hyödyllistä, sillä tällöin sääntöjä voidaan jakaa ja käsitellä samoin kuin muutakin tietoa. Tässä työssä esitetään lisäksi menetelmä, joka säilyttää sääntöjen käyttämien tietojen alkuperän sääntöjen evaluoinnissa. Tämän ohella esitellään edelliseen laajennus jonka avulla voidaan kontrolloida syötetiedon käyttökonteksti. Tämä mekanismi mahdollistaa mielivaltaisen metainformaation liittämisen sääntöihin sekä muuhun tietoon ja suo erityisesti mahdollisuuden määrittää lisätoimintaperiaatteita sääntöevaluoinnin ohjaamiseen. Nämä menetelmät mahdollistavat käyttäytymissääntöjen turvallisen ja hallitun jakamisen, analysoinnin sekä evaluaation yhdessä semanttisessa viitekehyksessä, tarjoten erään mahdollisen ratkaisun yhteensopivuusongelmaan. Työssä esitetään myös syntaktinen turvallisuusanalyysi reifioiduille säännöille. ASP-sääntöjen evaluaatiossa on kaksi loogista vaihetta: muuttujien instantiointi ja varsinainen ratkaiseminen. Tietojen alkuperän käsittely sekä syntaktinen analyysi on rajattu metatasolle. Näin varmistetaan, että tiedot tai säännöt, jotka eivät ole toimintaperiaatteiden mukaisia, eivät koskaan päädy ratkaisuvaiheeseen. Tässä työssä on esitetty käsittääkseemme ensimmäinen toteutus ASP-sääntöjen instantioinnista metatasolla. Suoritetun vertailun perusteella metatason instantioinnin tehokkuus ei ole vielä kilpailukykyinen nykyisten instantiointitekniikoiden kanssa.

Avainsanat

Sääntöpohjainen rajoiteohjelmointi, metaevaluaatio, muuttujien instantiointi, ubiikit järjestelmät, älykkäät tilat, Semanttinen web, säännöt, logiikkaohjelmointi, tietämyksen esittäminen, semantiikka, stabiilit mallit

Contents

List of Figures

List of Tables

Preface

The background for this work was laid when we were discussing with Prof. Ilkka Niemelä the need to represent and evaluate policies in a distributed blackboard environment, then under construction in the EU Artemis SOFIA project. Ilkka proposed using Answer Set Programming and telling me that "Don't come to me later saying that I didn't tell you about this." It soon became obvious that there is a need of handling rules and data from many different sources and this realization lead to the problem statement for this work. I was lucky to have Docent Tomi Janhunen as my instructor, the support he gave was of highest caliber, beyond competent, insightful, yet always in a relaxed manner. He specifically suggested focusing on metagrounding.

Parts of this work appeared within results of SOFIA project and also in the subsequent followup work supported by EIT-ICT labs and these facets partially funded this work.

In addition to Ilkka and Tomi, I'd also like to thank Prof. Axel Polleres for providing an expert review and suggesting new avenues for continuation.

I'd like to thank my wife Ingrid for all the love, support and patience during this work, it made all the difference!

This work is dedicated to my late father Mauri Luukkala.

Espoo, 18. April 2015

Vesa Luukkala

Chapter 1

Introduction

The amount of cheap embedded devices with computing power and communication facilities is growing at a rapid pace. This makes it feasible to embed these devices into otherwise non-digital objects and enabling them to offer information and resources digitally for other computers. Furthermore the line between such embedded devices, personal computers and mobile devices will blur in the sense that embedded devices increasingly gain enough capabilities and resources (memory, computing power, communication facilities) that allow them to carry out nontrivial computation. Together these communicating devices may form an aggregate system composed of multiple heterogeneous devices with some of them appearing and disappearing on ad-hoc basis. We call this system a *smart space*, an abstract entity, which makes services and information available for the user in a seamless way using the most suitable available resources. There may be several kinds of smart spaces which have a particular purpose for benefitting human users. This is a realization of the *Ubiquitous Computing* vision [111] and we can also call such systems *ubiquitous systems*. For example consider a building with sensors for temperature, sound and brightness along with controls for lighting and air conditioning. By default the system keeps a certain base temperature, but it only turns on the lights and stronger air conditioning when a person with a mobile phone is detected by the system in a particular room. Furthermore mobile devices of the users can also act as sensors which provides information at the location of the particular user. Having this information enables offering an optimized service across the building and better service for individuals. It may be that a person prefers silence over air conditioning and she makes this and other relevant preferences available via her mobile device.

In order for arbitrary devices to be able to communicate with each other, they must be *interoperable*, which requires definition of common terms for communication. An ubiquitous system also needs to maintain its integrity while still being able to accept contributing unknown actors on an ad-hoc basis. In this work we hypothesize that in order to fulfill these requirements, we need to be able to manipulate the behavioural descriptions of the actors and share them safely among the other actors. Furthermore we want to maintain the ownership of information provided by different participants, so that it is possible to control how that information is being used within the system. The origin and the changes of ownership of a piece of information is called the *provenance* of information. Taking this as a starting point we describe approaches for methods and architecture for implementing such a vision. We proceed to describe solutions for the interoperability, such as the semantic web [13] and then highlight some problems which arise from sharing behavioural descriptions, which is necessary for high-level interoperability. These lead to the necessity of being able to manipulate the descriptions of behaviour both to ensure safety and to maintain the ownership of data. In order to do this we need to decide on the type and semantics of these behavioural descriptions and in our case we select answer set programming, a logic programming paradigm. Finally we need to pick a suitable architecture for our system.

The motivation for this work, identification of the problem domain and some technology choices is based on our earlier work in building an information interoperability system combining the visions of ubiquitous systems and information interoperability by semantic web technologies. The aim of that system was to enable sharing of semantic information in an environment of multiple heterogeneous actors in accordance to the ubiquitous system [111] and semantic web [13] visions. The underlying blackboard implementation system is presented in [16] and we elaborate on it later on. Our initial implementations of mashing together two use cases, one in automotive domain and another in building maintenance domain [77] lead to a requirement for a a common distributed resource allocation framework for smart space applications and also to the requirement of sharing rules for interoperability. We implemented a resource allocation framework and described it in [78] where we used a small ontology and behaviour over that described in ASP rules. During this work we identified the need to speed up the resource allocation by means of concurrent execution. Furthermore, we noted the need for sharing behaviour for the purpose of expressing preferences as a client and policies as a service provider. We presented a solution for the distribution of the ASP resource allocation computation in [5] and this solution was elaborated, extended and analyzed by Aziz in [6]. As first stage towards managed sharing of behaviour, we presented an ASP metaevaluator for ground programs in [61]. In this work we analyze and extend that work to include grounding of ASP programs. There is earlier work on metaevaluation of answer set programs, but to our knowledge this thesis presents the first implementation of grounding of ASP programs by ASP rules. Our main contributions in this work are the mathematical analysis pertaining to manipulating the behavioural descriptions in ASP, a novel application of this manipulation to the grounding of ASP itself and identifying some generic problems arising in ubiquitous system settings concerning safety and ownership of data.

1.1 Interoperability

To handle complexity in communication, a communicating system can be divided into abstraction layers [116], where an individual layer serves the layer above it and uses the layer below it. Each layer relies on the lower level features to build a more abstract service for the higher layers, hiding the details of the lower levels. For instance a “network layer” is responsible for delivering packets of data between points on a network, but it does not guarantee reliable delivery: packet may be lost due to changes in the network, for example. A “transport layer” can then implement a reliable packet delivery using the services of the network layer below it by adding bookkeeping and error control for the packets. The logical separation of layers allows standardizing each one separately thus ensuring interoperability. This approach has been enormously successful and pervasive, it is used for instance in current mobile networks and the Internet [20, 67]. However, as the number of participating devices, vendors, requirements, product categories and produced information is large, creating a common standard is hard due to the need for collective acceptance [41]. Also, changing an existing standard is difficult partly for the same reason, but also due to the base of existing devices which may become partially or completely obsolete if the standards change. These problems occur especially on the highest layers. The purposes of the lower layers are quite restricted and there is an intuitive hierarchy for the concepts and services. This is not true for the highest layers, where the amount and combinations of possible purposes grows very large. For instance, in the OSI model [116] the highest layer is the seventh "application layer", which has standardized protocols for many diverse purposes, email, file transfer and naming systems among other things. The levels beyond the application layer are not standardized, but informally in some contexts layer 8 is referred as the "user layer" and layer 9 referred as the "organization layer". Lewis et al. [71] gives different layers as compared to the OSI model, but also identifies organizational layer as the highest layer. The semantic web vision [13] offers a partial solution to the high-level interoperability problem by mechanisms which allow handling and representing semi-structured information and giving the flexibility of augmenting existing information and definitions so that they remain compatible. However, as argued by Lewis et al. [71], even this approach is not enough for end-to-end interoperability on highest layers and that standardizing on the information only is not sufficient. To solve this problem Lewis et al. require that multiple heterogeneous participants of a smart space publish descriptions of their behaviour for other participants. Essentially, this permits an arbitrary user wanting to benefit from the services offered by the smart space system to investigate the behaviour of other system participants and choose its own behaviour according to its own interpretation. Our approach is to define the data by semantic web mechanisms and share the behaviour as logic programs, which can themselves be represented and shared using the same semantic web mechanisms. We elaborate this approach and the natural link between logic programming and semantic description of data in Section 2. Another approach to position semantic information within a layer model is presented by Decker et al. [29] where they propose a linked data layer between the seventh application layer and the sixth presentation layer. Here the linked data layer contains shared semantics as a common data model for all applications. Their goal is similar as in Lewis et al. [71]: exploit semantics to enable collaboration and interoperation between people, organizations and systems, but they do not take a stand whether a common data model without behavior is sufficient.

Finally, an alternative solution for the interoperability problem is that there is a single actor who is strong enough to force its conventions as a de-facto standard to which other actors must adhere to. This kind of situation is not in our focus, but we note that even in this context there is value in the flexibility offered by the approach we presented above: possibility of modifying an existing standard while maintaining the value of the existing base of users.

1.2 Architecture

To implement our vision of the smart space, we need to choose a suitable *architecture*, a mapping of intended roles and behaviour on the software and hardware platforms. In our ubiquitous system case, the emphasis is on the information produced and used by the participants. On a high level we have two types of actors: the participants of the smart space and the smart space itself. The participants may come and go but the smart space must exist independently of them and be able to serve the participants so that information is accepted and made available for other actors. Conceptually this is similar to a blackboard in a classroom: students and teachers may write on the blackboard, leave for an intermission and afterwards continue where they left off. Another class may later come in and read what the previous class had written and use that information for their benefit. A teacher is responsible for seeing that what is written on the blackboard is sensible and consistent. Furthermore she ensures that what is written is pertaining to its purpose; for example history class blackboard has content which is not relevant for the math class. This conceptual model is quite useful and it is known as a *logical blackboard model* [26, 40, 56], which presents a single shared, persistent memory, the blackboard, and a number of nodes, which only communicate via the blackboard. By logical we mean that the blackboard itself presents itself as a single entity, where information produced by one node is automatically available for the other nodes, even though technically the blackboard may be spread across multiple physical devices. It is the responsibility of the blackboard to maintain its consistency internally. The main benefit of using this model is that in general it admits concentration on the shared information as content on the blackboard rather than details such as node connectivity. Our starting point was interoperability on the highest levels and the earlier mechanisms of sharing information and behaviour using semantic web approaches can be implemented in a straightforward manner as information on the blackboard. The separation of concerns is not complete though, it may be that for example connectivity status of a node is essential and hence must be reflected and handled as high level information. This representation of the *state* of the nodes on the blackboard is not limited to physical status of an individual node but it can also reflect the status of the smart space itself reflecting the state of multiple nodes. Our conceptual execution model for the nodes is simple: a node that is executing a behavioural description or program reads in all relevant information from the blackboard and uses that as input for the program and after computation the produced output is the published back to the blackboard. The execution of the program may be triggered by a change on the blackboard or internally by the node. This execution model will have an effect on the behavioural descriptions. We elaborate on our architectural choices in Section 2.1.

1.3 Behavioural Descriptions and Information Ownership

In order to be able to benefit from the behavioural descriptions of other entities, we want to combine these descriptions with descriptions of our own behaviour or be able to modify our own behaviour based on the external behaviour. At this point there are several potential threats: are pieces of foreign behaviour harmful? Is the combination of many pieces of behaviour sensible or even computable? Whose behaviour description is trusted and should we handle differently behaviour and information from different parties?

We propose the following mechanisms to manage these risks: firstly, we should use a method for describing the behaviour which is safe by construction, i.e., it is not possible to define undesirable behaviour. Secondly we need to be able to track the provenance of the behavioural descriptions but also more generally any other information that the behavioural descriptions operate on. Thirdly we need to analyze the behavioural descriptions themselves before they are evaluated in the actual system. This implies that the behavioural descriptions should have well-defined semantics. There are inherent limitations in these mechanisms: the behavioural descriptions we are concerned about are essentially computer programs. A foundational result by Church and Turing [25, 109] implies that it is not possible to create a general algorithm which could decide for an arbitrary input algorithm whether that input algorithm stops or diverges. Hence a general analysis of an arbitrary program to detect undesirable behaviour is not possible. Furthermore limiting to a "safe" description method for which analysis is possible will limit the behaviour that can be expressed. It is however possible to choose a suitable compromise so that we have some guarantees of safety and we can perform meaningful analysis on the behavioural description. Our hypothesis is that Answer Set Programming (ASP) [74, 81, 84] is a suitable formalism for behavioural descriptions. It is a logic programming paradigm with similar syntax to Prolog [69, 106] and Datalog [44]. Syntactically ASP programs consist of rules and facts, but they have different semantics compared to Prolog and Datalog: the rules are treated as constraints over the possible solution sets. ASP is a *declarative* paradigm, which means that the rules describe the target solution and it is up to the ASP solver to calculate how this can be achieved. This differs from imperative paradigms, where the steps to reach the goal are explicitly given by the programmer. Declarativeness is also beneficial in that the order of computation (or syntactic presentation) for the individual rules is not relevant. This is useful in a distributed setting as we do not need to synchronize for order. Furthermore, both the rules and the data can easily be represented in a uniform way, a feature which ASP shares with Prolog as well. Finally ASP has a useful subset for which it is possible to deduce whether a piece of code can produce a result, which contributes to our safety requirement. This is a compromise we are making: the limitation of our chosen ASP-subset means that it is not a general programming language and that not all behaviour can be expressed using this ASP-subset. We describe the syntax and semantics of ASP in Sections 3.1.1 and 3.1.2.

As mentioned in the previous section, our execution model assumes that a node reads in input from the blackboard, performs a computation according to a behavioural description and publishes the results back to the blackboard. Multiple nodes may be executing at the same time but the only way they can communicate with each other is by writing and reading information on the blackboard. Any behaviour description or program which requires mutual communication must be contain such a read-write cycle for communication. In our case the basic structure of behavioural descriptions reflects this: each potential read-write operation is specified by by ASP rules and the full program then consists of chains of these read-write steps. Because we use a subset of ASP, for which we can guarantee that the computation produces a result or an error (rejection of the rules), no individual step will hang, but there is no guarantee that the steps together converge. For example it may be that two steps alternate forever, taking turns deleting results of the previous step from the blackboard. While this single-step approach requires splitting more complex behaviour and managing the transition between different steps, it has some benefits as well. The main benefit is that the program is built so that the preconditions of each step are explicit in the rules and the conditions can easily be mapped to the semantic concepts on the blackboard. This is useful in a ubiquitous system setting as we can define the context when a particular behaviour is applicable allowing us to focus to a limited subset of information in otherwise large and unpredictable environment. Even though a single read-write step is itself simple, the program which is executed in between can be complex, benefitting from the power of

ASP. We elaborate on our execution model and programs in Section 2.3.

Operating directly on rules which describe behaviour of another participant, an untrusted source, carries risks and in order to address them our basic approach is to transform the behaviour to data in such way that we can analyze the behaviour and if necessary, modify it. This gives us the possibility of controlling how the rules are evaluated enabling us to track the provenance of both the rules and the data used by them. Furthermore we can now share the behaviour in same way as any other data is shared. We translate the original behavioural rules as a set of simple facts describing the various elements of the rules. This is called a *reified* representation of the original rules. Since the reified representation is now data, a set of facts, we can create an ASP program which operates on those facts and this is called *metaprogramming*. Specifically, it is possible to create an ASP program which performs the same evaluation for the reified rules as an existing ASP tool would do for the unreified rules. We call the evaluation of ASP rules by other ASP rules *metaevaluation*. One of the key issues in our work is to make sure that the evaluation of the reified rules produces the same results as the evaluation of the unreified rules. If necessary, changing the metaevaluator allows us to modify the meaning (or *semantics* or interpretation) of the behaviour and expressing this in ASP allows us to maintain a well-defined relation between the original ASP semantics and the modified semantics. Hence our approach is based completely on ASP semantics. Our earlier work in [61] presented an ASP metaevaluator and we extend that by presenting a formal analysis of the metaevaluator showing that it produces essentially the same answer sets as would have been produced for unreified rules by existing ASP tools. We describe and analyze the reification and our metaevaluator in Sections 3.2 and 3.3.

In addition to the safety related problems, another requirement for our solution is related to the ownership of the data and control of its use. By default, once information is published, there are no guarantees concerning its use. We need to be able to have more control than that, minimally so that there are voluntary mechanisms for tracking the shared information. Also, as we have chosen to work on reified rules it means that sharing these rules is sharing of data and any provenance mechanism for data also pertains to behavioural descriptions as well. While our focus is within the ubiquitous computing context, the same problem is relevant in all distributed systems, such as the Internet or distributed databases. Since the rules we are sharing operate on information coming from different sources we need a way of deciding what to include as input for rules. The provenance can be hard-coded in the rules themselves, but we cannot assume that all rules, especially the external ones, use the same convention. Hence we need a generic approach which does not require modifying the rules and in this thesis we present a mechanism for maintaining the provenance during the metaevaluation process. ASP solving has two logical stages: *grounding* and *solving* and we are proposing to include the management of different information sources in the grounding phase, logically separating it from the solving phase. We present an ASP implementation of grounding for reified rules, *metagrounding* of ASP rules, which together with metaevaluator form a complete ASP toolchain for reified programs. We describe the metagrounding process in Section 4. The metaevaluation approach comes with costs in translating the plain rules and data to reified representation, computing in the domain of the reified representation and in translating back to unreified representation. The most obvious cost is performance and we aim to measure the performance of the proposed solution. However, as mentioned above, the computational problems for metaevaluation are in the domain where ASP solvers are known to be efficient, so it may be that the indirection induced by reification can be rectified with better domain specific rules. As we implement the grounding using rules, we hope to be able to express different evaluation and grounding approaches succinctly using rules and at the same time benefitting from the efficiency of the ASP solvers for performance. Our approach could also work as a mechanism to solve the "grounding bottleneck" problem [33] for answer set programs by mapping the rules to a restricted format for which there are known to be more efficient algorithms.

1.4 Related Work

The main focus of our work is in metaevaluation of answer set programs, but many of the related issues and problems have been researched in other fields from different perspectives. Some of the existing work on **metaevaluation of answer set programs** includes Gebser et al. [45] and Eiter et al. [33, 34]. Furthermore Delgrande et al. [30] present a framework for including preferences in logic programs, which is quite similar to metaevaluation. Previous works implement the solver part so their input are ground programs and their purpose is to take into account preferences between answer sets or modifying the default behaviour for optimizations (for instance minimize or maximize) across the answer sets. Our work can be seen as a continuation of previous metaevaluation work in the sense that we extend the metaevaluation approach to the grounding phase, whereas the previous work concentrated on solver part only. Gebser et al. [47] and Brain et al. [79] have applied ASP metaprogramming to debugging ASP programs themselves. Their approach is based on rewriting the programs to include specific tagging atoms and then use those to control the evaluation. This is similar to our metaevaluation approach, but we do not introduce any extra atoms, but rather provide hooks for any ASP code. In our analysis part we are examining the reified rules purely "syntactically", for instance, looking for names of atoms which should not be produced. The distinction of syntactic analysis and debugging is not clear and similar techniques are applicable in all kinds of analysis of ASP programs. Eiter and Polleres [37] present an automated metaevaluation based mechanism for integrating separate two-step propositional "guess" and "check" programs into a single program which performs both steps at the same time. Our metaevaluation is general, it aims to evaluate any program, furthermore our metagrounding approach handles program with variables. Furthermore Janhunen et al. [62] propose several criteria for testing answer-set programs. Although the paper does not explicitly mention metaprogramming, it builds on concepts which rely on source level access to the program being tested and hence the availability of these on the metaprogramming side could be useful for enabling the definition of tests in ASP. Grounders are essential components of an ASP toolchain and previous work for implementing a grounder is presented by Syrjänen [108] and Gebser et al. [48] among others. The **combination of rules in general and semantic web** has a good "impedance match" due to the uniform representation of the rules and data: it is straightforward to access semantic web data in rules. Hence there is a large body of previous work, including but not limited to a W3C standard proposal for a rule language [59], proposals for using Prolog as a fundament for semantic web [113] and a conference series for Web Reasoning [1]. A survey for different approaches and their relation to each other is given by Eiter et al. in [35]. Eiter et al. [36] also propose the integration of answer set programming with other semantic web reasoning mechanisms. The combinations of ASP and semantic web focus on using the rule languages for application development, which is similar approach to our work in [78]. Zimmermann, Polleres et al. [115] present a framework for adding annotations to RDF data and methods for performing RDFS reasoning with annotated RDF data. Furthermore they present an extension to SPARQL [103] query language which allows querying such annotated data. The annotations they highlight are timing and provenance annotations. These particular annotations along with the notion of annotating graphs and specifically RDF graphs is similar to our approach, but we separate the evaluation of the annotations to distinct rules and we aim to introduce the annotation handling in metaevaluation.

The related **notions of reflection and reification** have been to date studied extensively and the earliest work has been done in LISP context. This is because the syntactic representation of a program in LISP is essentially the syntax tree of the program. An early formalization for reflection for LISP is by Smith [100, 101]. An overview of reflection in different programming paradigms is given in [31]. Our work is in the field of logic programming, which starting from Prolog, have a very straightforward relation between the actual program, its representation and the representation of goals and clauses. Foundational work is by Bowen and Kowalski [18] and since then the notion of metainterpretation has been accepted to mainstream to such degree that descriptions of Prolog metainterpreters are found in textbooks like Sterling and Shapiro [106] and Prolog systems usually incorporate metaprogramming facilities as part of the standard library (see, e.g., [24, 114]). Similarly to the synergies between semantic web and logic programming, the **combination of relational databases and logic programming** have also obvious complementary benefits. Systems which augment databases with often rule-based deduction mechanisms are called *deductive databases* [44, 60, 92]. In such systems part of the data is represented *extensionally* as relations on the database itself. Data which is produced by the evaluation of rules is called *intensional* data. The goals of our work a very similar to the goals set for the deductive databases, the main difference being that we focus on the ASP rule language and de-emphasize the database side. Also in our case, we are operating in a distributed environment, where the different actors are physically separated. This setting has many commonalities to *distributed databases* [88], which however aim to hide the distribution as an implementation detail. In contrast we aim to present the relevant synchronization mechanisms openly as semantic information.

One relevant mechanism in distributed databases is the **notion of provenance**, where the need to understand how the query results were constructed is important for consistency. The general method is to annotate query results with information about which pieces of information “contributed” to the results. There are various approaches for this, ranging from the *lineage* approach [28], which explains which source tuples were used for constructing a particular result tuple, to *provenance semirings* [53], which contain the information on how the result tuple was constructed from the source tuples. Green [53] also shows that all of the different approaches can be expressed using the same underlying semiring approach as was used for provenance semirings, making it the most general approach. Our approach for provenance is similar to the lineage approach, but for a produced fact, we keep several lists which may contain the same source. This is more general than *why provenance* [22] which records how many times a particular source has been used. As mentioned earlier, one of the main uses for annotations in Zimmermann, Polleres et al. [115] is provenance and its handling in RDFS reasoning and query answering. One of our original goals for this work was to enable different stakeholders to publish their own policies and preferences thus allowing suitable ad-hoc composition of the users and services. This objective is supported by a large body of work for systems and languages for **policy definition and enforcing**, especially for access control and trust management. Keynote and Policymaker [14, 15] are early decentralized trust management systems, in addition there are several declarative policy languages based on logic programming including Binder [32], Delegation logic [72], SecPAL [9] and DKAL [54]. The first three have semantics based on Datalog whereas the first versions of DKAL had semantics based on ASP. In our case we allow arbitrary ASP rules to handle the policies, which is more general, but at the same time lacking any of the structure and support from the dedicated approaches. Our work assumes a ubiquitous environment with a logical blackboard architecture. The tight **connection between the data on the blackboard and especially logic programs** that manipulate the data is clear and has been investigated before. Schwartz et al. describe a Prolog-based blackboard system [97] with metaprogramming facilities as a central mechanism for integrating different information sources. We differ in that we focus more on controlling and managing a limited aspect of the reasoning (attached metainformation such as provenance or timing) where Schwartz et al. present more generic orchestration components, such as events, interrupts and blackboard management. In addition we apply ASP semantics. Much of the work that combines semantic web with rules referred above assume the same kind of blackboard architecture as in [97]. Logical blackboard enables separation of concerns thus making it possible to concentrate on the semantic problem while still having an architecture which can easily be mapped onto many existing distributed architectures. Furthermore it is a natural choice for various collaborating software approaches, especially when the problem space is not well defined. We present more discussion regarding the architecture in Section 2.1. However, we note here that there is a large amount of research which pertains to these systems, but that may fall under many categories, for instance coordination infrastructure or tuple space computing. We point to Nixon et al. [86] for a survey which takes the semantic web perspective. Furthermore we point to Corkill [27] who compares blackboard systems to agent-based systems and gives criteria on classification of required properties for those systems.

1.5 Thesis Structure

This thesis is is structured as follows: in Chapter 2 we present our context for this work, the smart space and interoperability problems which arise from such a dynamic environment. We put forward our solution as a key underlying component to fulfill the requirements for interoperability in a dynamic ubiquitous environment. Chapter 3 presents the Answer Set Programming paradigm, the reification of answer set programs as well as metaevaluation and metagrounding of answer set programs. This section contains a mathematical analysis of our metaevaluator. The following Chapter 4 contains description of the metagrounder and the necessary list-based mechanism. Together Chapters 3 and 4 contain the main contribution of this work. This is followed by Section 5 where we present mechanisms which allow quantitative parameters to be incorporated in metaevaluation approach, namely tracking of provenance and timing. This is application of the mechanism we developed in the previous section. Section 5.6 presents timing measurements on the overhead induced by metaevaluation when compared to direct evaluation. In Section 6 our work is contrasted with existing work and evaluate our approach and finally we present some conclusions and outline for future work in Section 7.

Chapter 2

Smart Spaces

In this section we aim to introduce our primary application domain, the *smart space*, describe one potential architecture and implementation for it, present an example based on our earlier work and highlight the link of our work to semantic web. We argue that in order to implement such an inherently unpredictable and partially ad-hoc multiagent system, interoperability of information becomes a key issue and the architecture choices should support this.

2.1 Logical Architecture

We introduced earlier the notions of *smart space* and *ubiquitous system*. They offer services and information to benefit human users as well as other devices or smart spaces, all of which attempt to use the best resources available at a given time and a place. Our view on the smart space architecture is that its essential feature is the high-level view on the information available. Thus implementation specific issues, such as connectivity, processor architecture or mass memory are necessary enablers which are assumed to exist, but the main focus is just the information and its manipulation. Nevertheless, it is possible that the lower levels are reflected on the information level and manipulation of the information may have effects on the lower levels. A natural architecture to enable this is a *logical blackboard model* [26, 40, 56], where the two main elements are are *nodes* which communicate via a shared persistent memory, the *blackboard*. This concept is similar to *actor model* [2] and *tuple space* models like Linda [49]. This allows various heterogeneous systems to represent themselves as node implementations and use the blackboard as their interface to the smart space. Hence the notion of information interoperability of the content on the blackboard becomes a crucial issue and this discussed further in Section 2.2 below. Another related perspective is found in *Internet of things* [4] or *device-to-device* (D2D) [63] concepts, which build on the notion that even the smallest digital devices such as sensors have connectivity and identity. This allows building automated, digital systems where the devices' sensors can appear as virtual objects and have a degree of autonomy and self configuration capabilities, even though they are perhaps still considered as auxiliary actors to be used by other, larger and more powerful entities. One approach for implementing the combined vision of ubiquitous systems and information interoperability is provided by Smart-M3[1], an interoperability

[1] `http://sourceforge.net/projects/smart-m3/`

platform which allows devices to share and to access semantic information in RDF format through a single logical blackboard. This is the underlying platform that we have used in our earlier work [5, 77, 78]. The concept and the architecture of Smart-M3 are explained in [16] where two kinds of logical components are identified: a *semantic information broker* (SIB) and a number of independent *knowledge processors* (KP) exchanging RDF triples with the SIB by a dedicated protocol. There is no notion of a single application, but one is formed by the KPs which share an ontology and are able to produce and consume information. A new KP committed to the same ontology may contribute to the application by reusing the information in a new way or augmenting it with new information. This is partly enabled by the semi-structured form of semantic web definitions and the self-describing nature of semantic information. There are several use case and application implementations on this architecture [43, 58, 65, 68, 77, 85, 89] which follow the approach described above and aim at semantic interoperability as described in [65]. This architecture itself is simplistic and it is clear that the nodes or KPs may perform functions that are not visible on the shared blackboard, but in one extreme it is possible to represent everything on the blackboard. In such case the smart space system can be thought of as a totally transparent application and operating system, which uses the blackboard as shared memory, representing the state of the system as RDF graphs. Another view is that the blackboard acts as a (public) database and the program consists of multiple clients of the database, which is typical for many of the current web-based systems. The main difference here is that in the typical web-based system, the database is not directly accessible.

There is an underlying architectural assumption in the description of our blackboard architecture: we assume that individual spaces can guarantee strong consistency within their own logical blackboard, even though there may be several distinct participants within that space. Strong consistency means that visibility of changes and their apparent order on the blackboard is the same for all participants. Essentially the system aims to hide its distributed nature and appear as a single logical entity. So far our work has assumed this to be the case. There is an inherent weakness in our approach we assume there is an agreement of a value on participating systems. It has been shown by Fischer et al. [42] that it is not possible to reach consensus in an asynchronous system subject to failures. As a consequence either safety (no inconsistent state) or liveness (e.g., eventual consistency) of the system must be chosen. A corollary of this is given in the CAP theorem [51] which states that it is not possible to fulfill requirements of consistency, availability and partition tolerance at the same time. At most two alternatives of the previous three can be enforced. Consistency here means strong consistency as above, availability means that a failure of component nodes can fail without affecting operation of other nodes. Partition tolerance refers to the situation where message loss between components does not prevent the system from operating properly. In our distributed environment we cannot enforce that all messages are always delivered and furthermore we may not be able to distinguish between message loss and a failed node. As we are assuming a logical blackboard system, which requires consistent data and we cannot enforce all node to be present we must give up on availability. In practical implementation we usually expect availability of some basic services, so we may need weaker consistency models, possibly so that some subset of the data has less guarantees in this regard.

2.2 Interoperability and Semantic Web

To fully exploit the wide variety of heterogeneous devices existing today, the interoperability of the devices becomes an issue that must be addressed. In [71], four levels of interoperability are distinguished: machine-level, syntactic, semantic, and organizational interoperability. Machine-level interoperability solves the problem of exchanging data at the hardware or machine level, e.g., by means of protocols, operating systems and compilers. The syntactic interoperability deals with interoperation between systems established by the machine level. The solutions for this are provided, e.g., by Service Oriented Architecture (SOA) middleware [39]. These two layers are covered by current technologies. Semantic interoperability attempts to ensure that communicating systems have same understanding of the information (shared by means of the lower layers) by all communicating participants. For instance, it is important to know whether a price quote includes sales tax or not. Finally, organizational interoperability insists that operations performed on the information produce consistent results understood by all participants. This is needed when business processes and workflows along with information transcend organizational boundaries. An example of this is that there are two medical service providers performing the same activity, which results in a modified medical record. However one requires a blood test prior to admission and the other performs it after the admission. Changing providers midstream carries a risk as changing from latter after the admission to the first provider may cause the blood test to be omitted. The authors of [71] point out that current standards are insufficient and they also also argue that only the first two levels and a part of the third can not be achieved by standardization activities.

Semantic interoperability presumes the same understanding of the information by all participants whereas organizational interoperability insists that operations performed on the information produce consistent results understood by all participants. A partial solution for the semantic interoperability can be achieved by formalizing agreements [87] by domain-specific *ontologies*. An ontology is a formal naming and definition of the types, properties, and interrelationships which exist for a particular domain of discourse. The mechanisms for ontology definitions are provided by the Semantic Web [13] where the information is minimally described in terms of *resource description framework* (RDF)[2] triples. In addition, dedicated knowledge representation languages such as RDFS[3] and OWL[4] allow the specification of more complex structures on top of RDF triples.

In practice it is cumbersome to expose the full state of a system as RDF (or similar), but nevertheless, when the computation is no longer one way (read-only or write-only), similar patterns of behaviour to full application are needed. In order to solve this we need to describe and agree both to the state and the operations pertaining to it. Our vision for solving this is that nodes which participate in smart space publish their information according to ontologies, but they should also publish description of their own behaviour pertaining the ontological information. This enables the other participants to reason about the concepts and assumptions of others, thus increasing interoperability in the sense proposed in [55]. In our case the behaviour is described by a set of read-write steps where the information is read from the blackboard, processed by the rules and based on these results, information

[2] http://www.w3.org/RDF/

[3] http://www.w3.org/TR/rdf-schema

[4] http://www.w3.org/2004/OWL/

is written to (or deleted from) the blackboard. The new concepts described by the rules refer to information defined in ontologies which contain the agreement of the meaning of these concepts (semantic level interoperability). The rules contain the agreement of how new concepts and operations are defined using the existing semantic level concepts (organizational interoperability). The use case description in Section 2.3 elaborates on the notion of behaviour.

We choose to use rules to describe behaviour, because they have a good “impedance match” with the semantic web mechanisms: an RDF triple is easy to represent as a tuple. The rules themselves have an uniform representation with the data: a fact, smallest piece of information, can be represented as a rule with a head only. Other elements of a rule body have a similar straightforward mapping to the semantic data. The syntax of the rules can be represented using semantic web mechanisms, RIF (Rule Interchange Format) [17] being a W3C standard effort. Hence we can share the rules as data among other data on the blackboard.

Publishing the behaviour syntactically is not enough, for interoperability we also need to define semantics for the published behaviour. We have chosen to focus on ASP semantics mainly because our earlier use case of resource allocation under combined multiple preferences (in Section 2.3) requires solving a hard computational problem for which ASP is especially suited. For interoperability the vision is to have a single underlying semantic basis for all rules. We want be able to use the same semantics to interpret the rules across all participants and one way of ensuring this is to implement the evaluation by the same mechanism as we use for evaluation, i.e., we metaevaluate the rules. The semantics of the metaevaluator is naturally defined by the metaevaluator implementation, but by implementing it using ASP we remain within the same semantic base. In principle the common ground is then defined using an executable core of ASP semantics (as an ASP program). At this point other semantics can be defined using the core ASP semantics and their implementation is then based on the existing ASP toolchains. In practice ASP is less suited for general purpose computation when compared to some other rule-based options, this is visible later in this work when representing basic lists. Eiter et al. [35] present a survey of rule-based languages in conjunction with semantic web which lists several systems and semantics which could be used instead of ASP. In practice it is realistic to expect multiple different semantics for different uses, however, the same problems arise in those cases as well and metaevaluation approach is one solution for them. Most of our work in this thesis concerns defining the ASP implementations for metaevaluation but also presenting a way to modify the evaluation such that other functionality can be injected “under the hood”, in the evaluation of the rules themselves, while maintaining a compatible semantic basis.

2.3 Use Case: Preferences in Resource Allocation

Now we are ready to discuss a use case which highlights some of the issues in the previous two sections in the context of rest of the thesis. In [78] we presented an ASP based resource allocation mechanism for a distributed smart space environment. The notion of resource allocation is quite general and it underlies many typical situations in a distributed environment, especially, when multiple parties need to mutually agree on something. We present an example of sharing an audio resource among passengers in a car based on their indi-

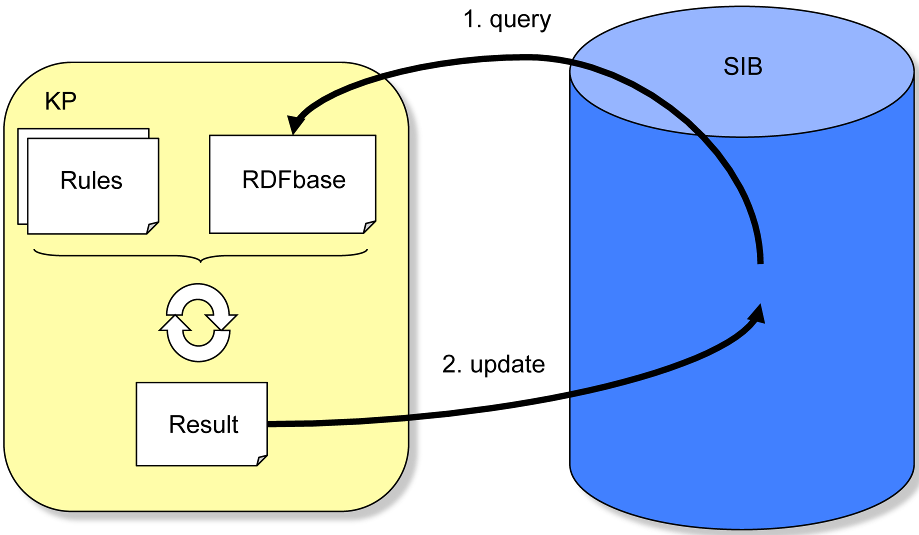

Figure 2.1: Operation of a single KP.

vidual preferences. For now we have a simplistic notion of preferences, globally mapping types of resource uses to integer values without concern for the source of the mapping.

The overall execution of a set of participants (or KPs) is an asynchronous process where the participants query the SIB, do some inferences and then make atomic updates in the SIB in an interleaved fashion. Figure 2.1 illustrates the operations of a single KP.

The individual KPs operate independently and the only guarantee for their operation is that insertions and deletions are atomic. This requires implementing synchronization primitives on top of the basic functionality and we can see our implementation of resource allocation one such primitive. Each participant reads in a subset of SIB contents and processes it locally. We have a set of predefined fact names which are interpreted as insertions, deletions and subscriptions to the SIB. These are interpreted locally for the inference results and a runtime then commits these actions atomically. As a simple example of an action consider an action to insert a single RDF-triple:

$$\mathsf{i}(123, \mathsf{temp:celsius}, "22").$$

Any semantic instance which is produced by the rules must be “rendered” as RDF-triples and then committed (or deleted by fact d) using this action mechanism. Figure 2.2 shows an example of this. In step 3 the rule execution has determined that a device should become active and this should be reflected in the user counter as well. This is reflected by a dedicated semantic instance (here a `to_commit` fact) referring to the device ID and there are rules which then have this semantic instance in the body and derive the necessary insertions and deletions to make necessary changes on the SIB. In the picture we only show a subset of the necessary deletions and insertions, but they are consistent for that particular device instance.

The interface between the rules and the information on the SIB has then two components: firstly the query which reads in the relevant content from the SIB and the aforementioned actions. The query can be constructed from the rules so that only content that is relevant is transferred. Furthermore it is possible to define a permanent query or a subscription which triggers the rules upon changes in the SIB. We can interpret each such step as a

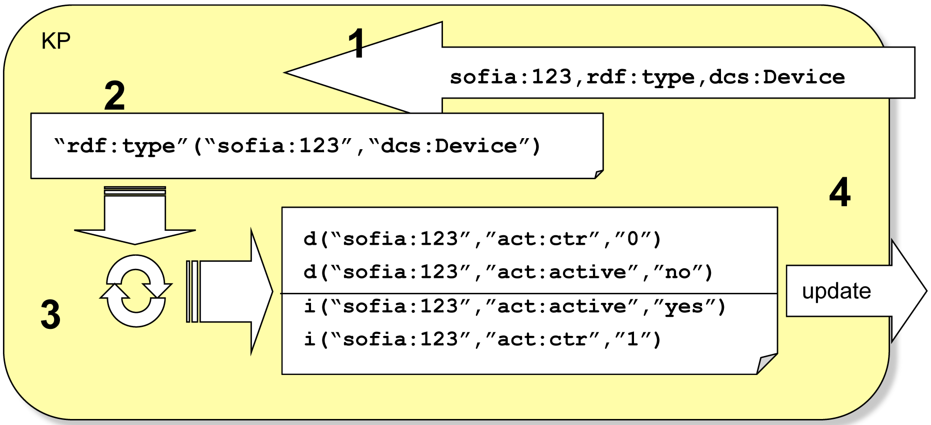
KP
1
sofia:123,rdf:type,dcs:Device
2
"rdf:type"("sofia:123","dcs:Device")
3
d("sofia:123","act:ctr","0")
d("sofia:123","act:active","no")
i("sofia:123","act:active","yes")
i("sofia:123","act:ctr","1")
4
update

Figure 2.2: Changing existing RDF facts.

Hoare triple [57] $\{P\}\ C\ \{Q\}$ where the query acts as a *precondition* P for the *command* C which consists of the rules to be evaluated and finally the actions form the *postcondition* Q which should hold after the execution. It is up to the programmer to compose the individual steps to meaningful and correct sequences or then use these as steps primitives to build higher level mechanisms such as actors [2] or tuple spaces [49]. It may be necessary to make sure that no other step can interfere between two steps and constructing the preconditions defensively. Alternatively it may be possible to create another step that is triggered by the same precondition as an existing step and hence augment an existing computation.

The purpose of our resource allocator framework was to enable multiparty synchronization and we present the following use case to motivate our work. A user starts listening to music on her mobile device (MD), using MD's loudspeaker and keypad to listen and control. When entering her car, she can seamlessly use the car loudspeakers and the steering wheel buttons and when exiting the car, she can continue the listening but only with the resources of the MD. The intuition behind this is that the user should have the best available resources at her disposal. A the same time another device (the car itself) monitors an account for messages and upon receiving one, uses text-to-speech to read the content of the message for the user. Both of these activities exist in the same smart space and compete for the use of audio resources, not necessarily for the same individual resource, but for the user's attention.

For defining the vocabulary for the entities in our system we use parts of SPICE mobile ontologies [110], which define relevant concepts for devices and capabilities. The ontologies are described using OWL (and the ontologies it refers to: RDF,RDFS,XSD), which means that any entity is identified by an URI [12]. Syntactically this means that the concepts are expressed as URLs in the http scheme (e.g., URI for an empty class: `http://www.w3.org/2002/07/owl#Nothing`) which can be accessed via the web. There are a number of *namespaces* [21] which is a name identified by a URI. For example name `owl` is identified by URI `http://www.w3.org/2002/07/owl#`, which allows rewriting the above empty class URI as `owl:Nothing`. Specifically we use classes `dcs:Device`,`dcs:AcousticModalityCapability` and `dcs:KeypadInputCapability` which represent a particular capability the participants are willing to offer. The `dcs` namespace is identified by the URI `http://ontology.ist-`

`spice.org/mobile-ontology/1/0/dcs/0/dcs.owl#`.

We describe a simple additional ontology (namespace `wp1`) with ASP rules which together perform resource management for any generic resources while trying to minimize the required extra information needed in addition to the SPICE concepts above. Whenever a device joins the smart space, it publishes information about its capabilities as instances of SPICE classes. To use these capabilities an entity can publish an *Activity* instance (from our custom `wp1` ontology) which refers to the particular capabilities it wants to use. The referring means publishing a relation between the activity and capability instances. This triggers the resource allocation mechanism, which attempts to fulfill the requirements of the activity according to preferences and priorities of the capabilities and the activity itself. The following example shows an excerpt of the rules of the resource allocator.

Example 2.3.1 *These rules from [78] are a part of a two step system which in the first step looks for requests for a resource and then in the second step assigns a resource to the requester. The following rules are part of the second step. First we have an auxiliary predicate, which reflects the fact that an existing capability is committed to some purpose. The rule refers directly to RDF-triples read in from the SIB. These are atoms where the name is same as the predicate of the RDF subject-predicate-object triple. The two terms of the atom are the subject and object of an RDF triple. Syntactically any value representing part of an RDF triple is always enclosed in quotes.*

$$\begin{aligned}\mathsf{committed\_to}(Cap, Act) \leftarrow\ & \mathsf{managed\_cap}(Cap),\\ & \mathsf{"rdf:type"}(Cap, \mathsf{"dcs:Capability"}),\\ & \mathsf{"wp1:uses"}(Act, Cap),\\ & \mathsf{"wp1:commits"}(Cap, Act).\end{aligned}$$

The following rule renders an insertion action which commits to a request (here an Activity*). The actual decision has been computed by the rules which have produced an internal* to_commit *fact, but we also explicitly require that the capability has not been committed elsewhere by the negated* committed_to *atom. Here the internal facts are converted to a relation in the custom ontology, which is published to the smart space.*

$$\begin{aligned}\mathsf{i}(Cap, \mathsf{"wp1:commits"}, Act) \leftarrow\ & \mathsf{managed\_cap(Cap)},\\ & \mathsf{"rdf:type"}(Act, \mathsf{"wp1:Activity"}),\\ & \mathbf{not}\ \mathsf{committed\_to}(Cap, Act),\\ & \mathsf{to\_commit}(Cap, Act).\end{aligned}$$

Both capability instances and activity instances are managed by dedicated rules, where the activity management takes a global view and capability management a local view. The programs which use the managed resources must observe and honor the decisions made by the resource allocator. An example of operation is given in example 2.3.2 below.

Example 2.3.2 *We have two capabilities: a loudspeaker and a keypad. We abuse the notation and name them and other entities in an URI-like manner (e.g.,* `loudspeaker:Capability` *and* `keypad:Capability`*), even though they are not URIs. We have two users for these capabilities: a music player and a message reader. The*

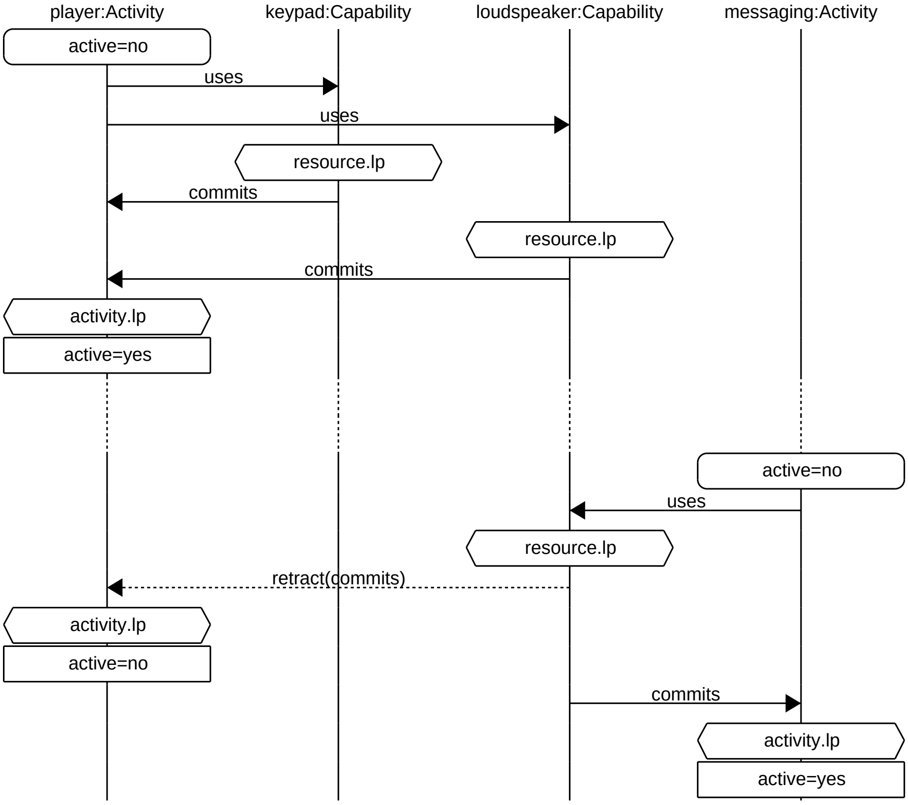

Figure 2.3: Two activities being scheduled.

music player wants to use both capabilities and the resource allocator allows this. However later on the message reader makes a request for the loudspeaker and we have set the message reading to have a higher priority than music playing. The allocator will hence retract the commitment of loudspeaker to the music player and give this commitment to the message reader. This is shown in Figure 2.3.

The resource allocator rules should enforce the following three principles: all available capability should be used, activities with higher priorities should override those with lesser priorities. We do not elaborate these further here, but refer to the original paper [78] for detailed description.

In [5] our main goal was to achieve better performance by creating a distributed resource allocator. In both this case and the original resource allocator case we recognized the need of including user specific parametrization to the computation: preferences or policies regarding foreign use of own resources or different scheduling mechanisms for different resource allocation instances. These directly lead to the questions concerning the safety and provenance of the rules and data we outline in the introduction. The common theme is that we have a set of “core” rules which are standalone but nevertheless must take into account the possibility that part of the data or even some of the rules are coming from an untrusted source. Using the terms of the previous sections, we can assume the following high-level architecture: there are several “smart spaces”, in this example, home, the car and the personal space of the user. All of these have their own resources which they own and actively manage, allowing “foreign” entities to access them. These resources are described

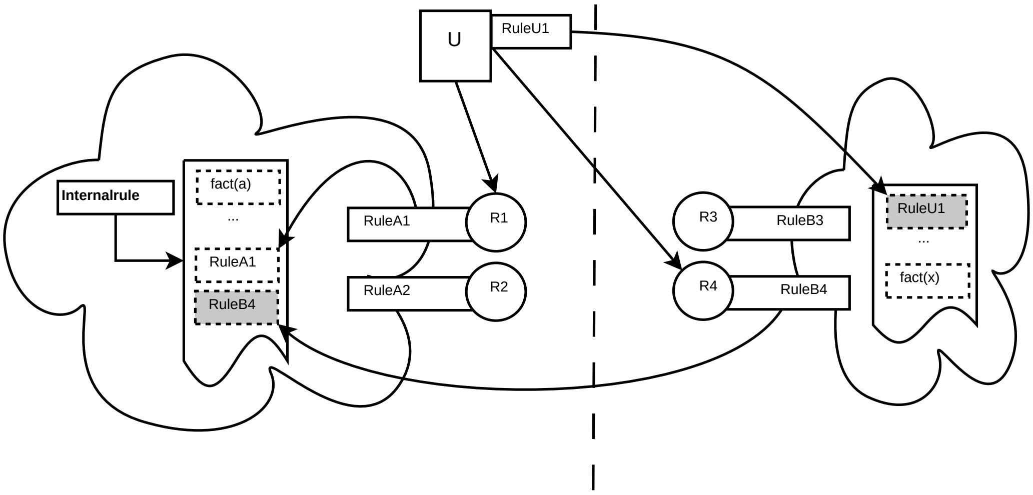

Figure 2.4: Using resources in two spaces

by semantic data and the ownership of the resources is reflected in the data. Furthermore we assume that the spaces share at least some common concepts, i.e., they use a common ontology and that the owners of the resources publish rules or policies which describe how they themselves want their resources to be used. These rules can be the same rules as used by the host system itself, in case they are different then the published and actual rules must be consistent with each other. It is possible that the rules translate between different ontologies of the different participants, enabling separate ontologies, but even in that case we assume that we have published rules that describe the resource use. For simplicity and focus we do not elaborate this case further. In addition to rules, the smart spaces and their users may also have data for which they want to set conditions of use.

Each of the individual spaces makes decisions based on its own trusted rules, but when operating with external entities must include their rules in the decisions and honor the conditions for the use of their data. This means that the entities executing the rules must have mechanisms of ensuring that the untrusted rules behave properly and that the provenance of these rules is known. This is depicted in Figure 2.4: we have two separate spaces A and B, having their own resources, for which they publish corresponding rules (e.g., resource $R1$ and RuleA1). We have a user U, that wants to use resources within its native space A and a foreign space B. Before doing this, it obtains the description of the foreign used resource RuleB4 and combines it with the description of the local resource RuleA1. These rules are then managed on the blackboard of A by its own internal mechanisms Internalrule. Here we need to make a distinction between our own, "safe" rules and the external ones. The space B may augment its rules with policies of use: here for example stating that the rule applies only for data and rules which operate in EU. The space B has also the same mechanisms at work, for instance for this case it may also require U to provide a description of its own intentions, i.e., RuleU1. This thesis concentrates on the mechanisms of the Internalrule part, which allows importing foreign rules and data, analyzing them and reasoning about the provenance of the information and maintaining it for newly produced information. This is based on manipulating the rules in reified format, essentially as data. To enable this the key mechanisms are maintaining the provenance of the data even when it is being manipulated by rules and being able to reason about the rules themselves. These mechanisms and the necessary semantics are introduced in the following sections.

Chapter 3

Answer Set Programming and Metaevaluation

The purpose of this chapter is to formally introduce Answer Set Programming (ASP), its basic syntax and semantics. This is followed by definition of reification for ASP programs and a description of the metaevaluation. We limit ourselves to normal programs with variables, which is despite the limitations a useful subset of the language so that we can use it to describe relevant behaviour in our applications. This chapter focuses on metaevaluation propositional (non-variabled) programs, the following section presents the metagrounding mechanism, which allows us to handle programs with variables. Our approach is to transform the rules to a reified format and base our operation on manipulating these reified rules. The basic functionality we need is to be able evaluate the reified rules as if they were non-reified rules and to this end we present the semantics of ASP and a metaevaluator which is consistent with these semantics. We also present the reified format, which is to be shared by different participants of a smart space. The metaevaluation and the metagrounding provide ASP implementations for both phases of ASP rule evaluation: grounding and solving.

3.1 Stable Model Semantics of Answer Set Programming

Answer Set Programming [74, 82, 84] is a declarative knowledge representation [7] formalism, especially suited to computationally difficult search problems. A problem is described by means of logic program rules with first-order logic primitives. The rules are interpreted as constraints on the solution set. A natural definition for the solution sets is given by *stable models* [50]. There are several freely available high-quality ASP solvers, which compute the stable models, including `smodels`[1] [99], `clasp`[2] [46] and `dlv`[3] [46]. The actual computation of the stable models is performed for propositional (non-variabled) logic programs. Nevertheless for practical reasons admitting variables is desirable. Before computing the stable models of variabled ASP programs, they need to be grounded, i.e., variables in rules are replaced with constant values available in the program as facts. Different variable replacements produce multiple ground versions of the same rule. At this

[1] `http://research.ics.aalto.fi/software/asp/smodels/`

[2] `http://potassco.sourceforge.net`

[3] `http://www.dlvsystem.com/dlv/`

point ground programs can be considered as propositional programs, consequently they can be solved as such. Typical ASP toolsets come with a distinct grounder program, which produces a grounded and simplified version of the input program for the solver. Many of the syntactic features of ASP programs can be resolved at grounding time. Therefore in practice the grounders act as parsers which are needed to feed any ASP program, propositional programs included, to the solvers. The grounder for `smodels` is `lparse` [107], the grounder for `clasp` is `gringo` [48], whereas `dlv` integrates both grounder and solver in same program.

3.1.1 Propositional Programs

We now introduce the basic concepts for rules and their interpretations, then give semantics for positive programs which is followed by the stable model semantics. We use the following syntactical conventions: *atoms* are written in lowercase letters, a *literal* is an atom a or its negation $\mathbf{not}\ a$, where $\mathbf{not}$ denotes *default negation*. Negated literals are also called *negative literals*. A *normal rule* is of the following form

$$a \leftarrow b_1, \ldots, b_n, \mathbf{not}\ c_1, \ldots, \mathbf{not}\ c_m. \tag{3.1}$$

where a is the *head* of the rule. The *body* of the rule consists of positive body literals $b_1, \ldots, b_n$ and negative body literals $\mathbf{not}\ c_1, \ldots, \mathbf{not}\ c_m$. A rule which only has a head is a *fact* which is true unconditionally. A *constraint* is a positive rule without head, having the form

$$\leftarrow c_1, \ldots, c_m.$$

The body of the constraint lists the literals which should not be true at the same time.

A set of rules which only have positive body literals is called a *positive program*. In order to give semantics for the evaluation of positive programs, we define additional concepts. Firstly, we define the possible components which can be used to evaluate the program; these are the atoms which may hold or not hold. From this moment forwards when we refer to atoms, they are by default defined according to Herbrand base of some program.

Definition 3.1.1 (Herbrand base) *Let P be a propositional normal logic program. The* Herbrand base *of P, denoted $\mathrm{Hb}(P)$, consists of the atoms that appear in P.*

Secondly we define what it means for one rule to be "evaluated" using the available atoms from the Herbrand base. Subsets of Herbrand base are considered as worlds for which a particular rule may hold true. The intuitive reading is that if the atoms in the body of the rule are true in a world, the head of the rule must also be true in that world.

Definition 3.1.2 (Satisfaction) *Let P be a positive program. An atom a is* satisfied *by an* interpretation *$I \subseteq \mathrm{Hb}(P)$ if $a \in I$. This is denoted by $I \models a$. Interpretation I* satisfies *a rule $a \leftarrow b_1, \ldots, b_n$ iff*

$$I \models b_1, \ldots, I \models b_n \Rightarrow I \models a.$$

This is denoted $I \models a \leftarrow b_1, \ldots, b_n$.

If all rules in a positive program are satisfied by an interpretation, then we call it a model for that program. Atoms of Herbrand base which are in the interpretation are considered true and others, still in the Herbrand base, are false.

Definition 3.1.3 (Model) *An interpretation $M \subseteq \mathrm{Hb}(R \cup C)$ is a* model *of a set of rules R and a set of constraints C iff for all $r \in R \cup C, M \models r$. This is denoted by $M \models R \cup C$.*

There may be several models for a program and if every model contains the same atom, then that atom is a logical consequence of the program.

Definition 3.1.4 (Logical consequence) *An atom a is a* logical consequence *of $R \cup C$ iff $a \in M$ for all interpretations $M \subseteq \mathrm{Hb}(R \cup C)$ such that $M \models R \cup C$.*

Among the multiple models we can find models for which there are no smaller models. The relative order is defined by set inclusion, inducing a lattice over subsets of Herbrand base.

Definition 3.1.5 (Subset minimal model) *An interpretation $M \subseteq \mathrm{Hb}(P)$ is a* minimal model *of a positive program P iff $M \models P$ and there is no model $N \models P$ such that $N \subset M$.*

Furthermore for positive programs there is only one minimal model, which is the smallest set of atoms which "justifies" all of the rules in the program. Conversely atoms in Herbrand base that are not logical consequences of the program are not included and are hence assumed to be false by default.

Theorem 3.1.1 (Least model [75]) *Every positive program P has a unique minimal model, the* least model of *P, denoted $\mathrm{LM}(P)$, which is the intersection of all of its models $\mathrm{LM}(P) = \bigcap\{M \subseteq \mathrm{Hb}(P) \mid M \models P\}$.*

Example 3.1.1 *Consider the following positive program P:*

$$a \leftarrow c. \quad b \leftarrow a. \quad a.$$

The Herbrand base of P is $\mathrm{Hb}(P) = \{a, b, c\}$. The interpretation $I_1 = \{a, b\} \subseteq \mathrm{Hb}(P)$ satisfies rule $b \leftarrow a$ that is $I_1 \models b \leftarrow a$, but $\{a\} \not\models b \leftarrow a$, for example. Note also that $I_1 \models a \leftarrow c$. Interpretations I_1 and $I_2 = \{a, b, c\}$ are models of the program as they satisfy all rules in P. However, $\{a, c\} \not\models P$, $\{b, c\} \not\models P$, $\{a\} \not\models P$, $\{b\} \not\models P$, $\{c\} \not\models P$ and $\emptyset \not\models P$.

The set of logical consequences of P is $\{a, b\}$ when restricted over $\mathrm{Hb}(P)$. The interpretation I_1 is a minimal model of P, whereas I_2 is not. The least model of P is $\mathrm{LM}(P) = I_1 \cap I_2 = \{a, b\}$ and there is no smaller model.

In order to construct the least model we define an operator whose least fixpoint equals to the least model.

Definition 3.1.6 ([75]) *Let P be a positive logic program. We define an operator $\mathrm{T}_P : 2^{\mathrm{Hb}(P)} \to 2^{\mathrm{Hb}(P)}$ on interpretations $I \subseteq \mathrm{Hb}(P)$:*

$$\mathrm{T}_P(I) = \{a \in \mathrm{Hb}(P) \mid a \leftarrow b_1, \ldots, b_n \textit{ and } \{b_1, \ldots, b_n\} \subseteq I\}$$

An interpretation is a fixpoint *of $\mathrm{T}_P(I)$ if $I = \mathrm{T}_P(I)$ and I is the* least fixpoint *of T_P iff $I \subseteq I'$ for any interpretation $I' \subseteq \mathrm{Hb}(P)$ such that $I' = \mathrm{T}_P(I')$.*

The T_P operator has the following properties. An interpretation $I \subseteq \mathrm{Hb}(P)$ is a model of a positive program iff $\mathrm{T}_P(M) \subseteq M$. The operator T_P is *monotone* for a positive program P: $M \subseteq N \subseteq \mathrm{Hb}(P) \Rightarrow \mathrm{T}_P(M) \subseteq \mathrm{T}_P(N)$. For a positive program P, T_P has the least fixpoint $\mathrm{lfp}(\mathrm{T}_P) = \bigcap\{M \subseteq \mathrm{Hb}(P) \mid M = \mathrm{T}_P(M)\}$.

Theorem 3.1.2 ([75]) *For a positive program P*

$$\mathrm{lfp}(P) = \mathrm{T}_P \uparrow \infty = \mathrm{LM}(P)$$

where we define a monotonically increasing sequence of interpretations for a positive program P: $\mathrm{T}_P \uparrow 0, \mathrm{T}_P \uparrow 1, \ldots$ such that

1. $\mathrm{T}_P \uparrow 0 = \emptyset$*,*
2. $\mathrm{T}_P \uparrow i+1 = \mathrm{T}_P(\mathrm{T}_P(i))$*, for $i > 0$, and*
3. $\mathrm{T}_P \uparrow \infty = \bigcup_{i=0}^{\infty} \mathrm{T}_P \uparrow i$*, for the* limit *of the sequence*

In order to give semantics for programs which may have negated atoms in bodies, *normal programs*, we need further definitions. The aforementioned approach is based on the stable model semantics [50] which is the core of the answer set programming paradigm. The underlying ideas are firstly, that a negative literal $\mathbf{not}\ c$ is satisfied by a model M if c is not present in M: $M \models \mathbf{not}\ c \Leftrightarrow c \notin M$. Secondly, the least model of the program should satisfy all the negative conditions in addition to the positive ones.

Definition 3.1.7 (Gelfond-Lifschitz Reduct [50]) *Let P be a normal program and $M \subseteq \mathrm{Hb}(P)$ an interpretation. The* Gelfond-Lifschitz reduct *of P with respect to M, denoted P^M is*

$$\begin{aligned} P^M = \{a \leftarrow b_1, \ldots, b_n \mid \quad & a \leftarrow b_1, \ldots, b_n, \mathbf{not}\ c_1, \ldots, \mathbf{not}\ c_m \in P, \\ & M \models \mathbf{not}\ c_1, \ldots, M \models \mathbf{not}\ c_m\}. \end{aligned}$$

In the above for any interpretation M

$$M \models \mathbf{not}\ c_1, \ldots, M \models \mathbf{not}\ c_m \Leftrightarrow M \cap \{c_1, \ldots, c_m\} = \emptyset.$$

Essentially, the reduct produces a positive program, where the negative atoms have been interpreted in accordance with a particular interpretation M.

Definition 3.1.8 (Stable Model) *Let P be a normal program. An interpretation $M \subseteq \mathrm{Hb}(P)$ is a* stable model *of P iff $M = \mathrm{LM}(P^M)$. We denote the set of stable models* $\mathrm{SM}(P)$.

Stable models are not necessarily unique, normal program P may have several stable models. These models are minimal, for a stable model $M_1 \in \mathrm{SM}(P)$ there is no smaller model $M_2 \subset M_1$ such that $M_2 \in \mathrm{SM}(P)$. We present the following example to help visualizing the concepts of reduct and answer set.

Example 3.1.2 *Consider the following normal program P, which can be used to specify how a loudspeaker should be used in car* [4]*. We state that an audio resource (*loudspeaker*) is exclusively used by either for music playing (*player*) or traffic announcements (*announcements*). Furthermore we require that controls should be available for music playing (*controls*). The stable models of the following rules correspond to valid configurations in our system.*

$$\begin{aligned}
\mathsf{announcements} &\leftarrow \mathsf{loudspeaker}, \ \mathbf{not}\ \mathsf{player}.\\
\mathsf{player} &\leftarrow \mathsf{loudspeaker}, \ \mathbf{not}\ \mathsf{announcements}.\\
\mathsf{controls} &\leftarrow \mathsf{player}.\\
\mathsf{loudspeaker}&.
\end{aligned}$$

- *The Herbrand base of P is* $\mathrm{Hb}(P) = \{\mathsf{announcements}, \mathsf{player}, \mathsf{loudspeaker}, \mathsf{controls}\}$
- *We have the following reducts P^M with respect to different interpretations $M \subseteq \mathrm{Hb}(P)$.*

$$\begin{aligned}
P^{\{\mathsf{loudspeaker}\}} :\ & \mathsf{announcements} \leftarrow \mathsf{loudspeaker}.\\
& \mathsf{player} \leftarrow \mathsf{loudspeaker}.\\
& \mathsf{controls} \leftarrow \mathsf{player}.\\
& \mathsf{loudspeaker}.
\end{aligned}$$

$$\begin{aligned}
P^{\{\mathsf{loudspeaker},\mathsf{announcements}\}} :\ & \mathsf{announcements} \leftarrow \mathsf{loudspeaker}.\\
& \mathsf{controls} \leftarrow \mathsf{player}.\\
& \mathsf{loudspeaker}.
\end{aligned}$$

$$\begin{aligned}
P^{\{\mathsf{player}\}} :\ & \mathsf{player} \leftarrow \mathsf{loudspeaker}.\\
& \mathsf{controls} \leftarrow \mathsf{player}.\\
& \mathsf{loudspeaker}.
\end{aligned}$$

$$P^{\{\mathsf{player},\mathsf{loudspeaker}\}} : P^{\{\mathsf{player}\}}$$

$$\begin{aligned}
P^{\{\mathsf{player},\mathsf{loudspeaker},\mathsf{controls}\}} :\ & \mathsf{player} \leftarrow \mathsf{loudspeaker}.\\
& \mathsf{controls} \leftarrow \mathsf{player}.\\
& \mathsf{loudspeaker}.
\end{aligned}$$

- *We apply the operator T_P to the above reducts, which are positive programs:*

[4] This program is inspired by the hard drive example from the manual of `lparse` [107].

- *Consider* $P^{\{\mathsf{loudspeaker}\}}$*:*

  $$\begin{aligned}
  &\mathrm{T}_P \uparrow 0 = \emptyset \text{ by definition.}\\
  &\mathrm{T}_P \uparrow 1 = \mathrm{T}_P(\emptyset) = \{\mathsf{announcements}, \mathsf{player}, \mathsf{loudspeaker}\}.\\
  &\mathrm{T}_P \uparrow 2 = \mathrm{T}_P(\{\mathsf{announcements}, \mathsf{player}, \mathsf{loudspeaker}\}) =\\
  &\qquad\qquad \{\mathsf{announcements}, \mathsf{player}, \mathsf{loudspeaker}, \mathsf{controls}\}.\\
  &\mathrm{T}_P \uparrow 3 = \mathrm{T}_P \uparrow 2 \text{ fixpoint reached.}
  \end{aligned}$$

  However the interpretation used for the reduct differs from the derived least model $\mathrm{LM}(P^{\{\mathsf{loudspeaker}\}})$

  $$\{\mathsf{loudspeaker}\} \neq \{\mathsf{announcements}, \mathsf{player}, \mathsf{loudspeaker}, \mathsf{controls}\}.$$

  so the fixpoint is not an answer set for P*.*

- *Consider* $P^{\{\mathsf{loudspeaker}, \mathsf{announcements}\}}$*:*

  $$\begin{aligned}
  &\mathrm{T}_P \uparrow 0 = \emptyset \text{ by definition.}\\
  &\mathrm{T}_P \uparrow 1 = \mathrm{T}_P(\emptyset) = \{\mathsf{announcements}, \mathsf{loudspeaker}\}.\\
  &\mathrm{T}_P \uparrow 2 = \mathrm{T}_P \uparrow 1 \text{ fixpoint reached.}
  \end{aligned}$$

  Now the interpretation used for the reduct $\{\mathsf{loudspeaker}, \mathsf{announcements}\}$ *equals the produced fixpoint, so it is an answer set for* P*.*

- *For* $P^{\{\mathsf{player}\}} = P^{\{\mathsf{player}, \mathsf{loudspeaker}\}}$*, we do not show the steps but only the resulting fixpoint*

  $$\{\mathsf{player}, \mathsf{loudspeaker}, \mathsf{controls}\}.$$

  which is not an answer set.

- *For* $P^{\{\mathsf{player}, \mathsf{loudspeaker}, \mathsf{controls}\}}$*, the fixpoint equals the interpretation used for the reduct, so*

  $$\{\mathsf{player}, \mathsf{loudspeaker}, \mathsf{controls}\}.$$

  is also a stable model of P*.*

3.1.2 Programs with Variables and Grounding

In practice it is useful to be able to parametrize the rules with atomic formulas with variables in terms. A grounding process [108] substitutes variables with values from the Herbrand universe producing a new set of ground rules, which can then be evaluated using essentially the same answer set semantics as described above.

We introduce more syntactical constructs: an *atom* is of form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol starting with lowercase letters and $t_1, \ldots, t_n$ are *terms*. Terms may be *constants*, *variables* or *functions*. Constants are written starting with lowercase letters: $a, bar1$, variables are written starting with uppercase letters: $X, Y1, Zed$. Functions are of form $f(t_1, \ldots, t_m)$ where f is an m-ary function symbol and $t_1, \ldots, t_m$ are supplied as its arguments. An atom with with no variables is called a *ground atom*, likewise *ground terms* are terms with no variables. Note that we assume that our input rules for metaevaluation and metagrounding do not contain function symbols. However, the metarules operating on the input rules do use them.

A rule with variables is of form

$$r(t_1, \dots, t_n) \leftarrow p_1(u_1, \dots, u_i), \dots, p_m(s_1, \dots, s_j)$$

where predicates $r, p_1, \dots, p_m$ have terms $t_1, \dots, t_n$ $u_1, \dots, u_i$ and $s_1, \dots, s_j$ as their arguments. It may be that the same variable or constant appears in different atoms in different positions. For the example x_2, y_7 and z_3 could all be the same variable X.

Definition 3.1.9 (Herbrand universe) *Let P be a program with variables.* Herbrand universe *of P, denoted* $\mathrm{Hu}(P)$*, consists of ground terms constructible from constant and function symbols of P.*

If there are no function symbols in P, then $\mathrm{Hu}(P)$ is finite for a finite P.

Definition 3.1.10 (Herbrand Base) *Let P be a program with variables.* Herbrand Base *of P, denoted* $\mathrm{Hb}(P)$*, consists of ground atoms of form $p(t_1, \dots, t_n)$ where p is an n-ary predicate symbol in P and $t_1, \dots, t_n$ are ground terms from* $\mathrm{Hu}(P)$.

Clearly, $\mathrm{Hb}(P)$ is finite for finite P without function symbols.

Definition 3.1.11 (Ground Substitution) *Let $X_1, \dots X_i$ be variable symbols and terms $\{t_1, \dots, t_i\} \subseteq \mathrm{Hu}(P)$. A* substitution *$\sigma$ is a finite set of form $[X_1/t_1, \dots, X_i/t_i]$.*

For a given atom α, $\alpha\sigma$ denotes the replacement of the variables in α with constants from $\mathrm{Hu}(P)$ according to the substitution σ. We extend substitution for a rule straightforwardly by systematically replacing variables occurring in its literals by σ.

Definition 3.1.12 (Ground Rule) *Let P be a program with variables with a particular rule:*

$$r(t_1, \dots, t_n) \leftarrow p_1(u_1, \dots, u_i), \dots, p_m(s_1, \dots, s_j)$$

where $r, p_1, \dots, p_m$ are atomic formulas with terms $t_1, \dots, t_n, u_1, \dots, u_i, s_1, \dots, s_j$ which may be variables. Let $\{\sigma_1, \dots, \sigma_k\}$ be all possible substitutions for P over $\mathrm{Hu}(P)$*. A* ground rule *is a rule where all variable occurrences have been substituted by a substitution $\sigma \in \{\sigma_1, \dots, \sigma_k\}$.*

Example 3.1.3 *Consider a simplistic positive program A:*

$$\mathsf{edge}(X, Z) \leftarrow \mathsf{edge}(X, Y),\ \mathsf{edge}(Y, Z). \tag{3.2}$$

with a number of ground facts B describing a loop over three nodes:

$$\mathsf{edge}(1, 2). \quad \mathsf{edge}(2, 3). \quad \mathsf{edge}(3, 1). \tag{3.3}$$

The program is $P = A \cup B$, the Herbrand base $\mathrm{Hb}(P)$ consists of 2^3 possible different facts enumerating the possible combinations of predicate symbols and terms:

$\mathsf{edge}(1,1)$. $\mathsf{edge}(1,2)$. $\mathsf{edge}(1,3)$. $\ldots$ $\mathsf{edge}(3,2)$. $\mathsf{edge}(3,3)$.

One possible substitution is $\sigma = [X/3, Y/2, Z/1]$, which applied to A produces a ground rule:

$\mathsf{edge}(3,1) \leftarrow \mathsf{edge}(3,2),\ \mathsf{edge}(2,1).$

The substitutions come from $\mathrm{Hu}(P)$, so the resulting atoms are not necessary corresponding to facts in $\mathrm{Hb}(P)$, so we have $\mathsf{edge}(3,2)$, although there is no such thing among the facts. In practice the grounder will not produce such an inapplicable rule whose body is false. Many ground instances are useless and can be skipped by the grounder.

The notion of a ground rule is extended to full programs in a straightforward manner.

Definition 3.1.13 (Ground Program) *Let P be a program with variables. The respective ground program denoted $\mathrm{Gnd}(P)$ consists of all ground instances of its rules obtained by substitutions σ over $\mathrm{Hu}(P)$.*

If we assume that P has a number r of rules, a number c of constants and the maximum number of variables in any of rules in P is v, then the upper limit for the number of ground instances is rc^v. Now we are ready to define the answer set for programs where the variables are removed by means of grounding and the resulting program can then be considered as a normal program, like in Definition 3.1.8.

Definition 3.1.14 (Answer Set) *Let P be a program with variables. The unique* stable model *of P is $M = \mathrm{LM}(\mathrm{Gnd}(P))$.*

Example 3.1.4 *Consider again the earlier positive program A in (3.2) with facts B in (3.3). The steps of an intelligent grounding process are as follows; firstly we produce a ground rule instance:*

$\mathsf{edge}(1,3) \leftarrow \mathsf{edge}(1,2),\ \mathsf{edge}(2,3).$

This rule could be given to the solver as such, but as it is "obviously" true, each fact in the body is justified by an existing fact, the grounder will immediately discard the rest of this rule and produce a new fact: $\mathsf{edge}(1,3)$. *This in turn can be combined with the existing facts to produce another ground rule instance, which can be immediately evaluated to a ground predicate:*

$\mathsf{edge}(1,1) \leftarrow \mathsf{edge}(1,3),\ \mathsf{edge}(3,1).$

So the grounder produced a new ground predicate based on another ground predicate, which it had created earlier during the grounding process. Later on during metagrounding this will become an issue.

Grounding the program produces the following new ground atoms, which are also the stable model for the program.

$\mathsf{edge}(1,3)$. $\mathsf{edge}(2,1)$. $\mathsf{edge}(3,2)$.
$\mathsf{edge}(1,1)$. $\mathsf{edge}(2,2)$. $\mathsf{edge}(3,3)$.

This is essentially producing a transitive closure of the edge *relation. Note that a naive grounder implementation would produce all* 3^3 *ground rules starting from*

$$\mathsf{edge}(1,1) \leftarrow \mathsf{edge}(1,1),\ \mathsf{edge}(1,1).$$

These ground rules would then have to be evaluated by the solver.

We now introduce a new syntactic feature which is resolved at grounding stage: the *conditional literal* or *composite condition* is of form $\mathsf{a}(X)\ :\ \mathsf{b}(X)$, where the first literal $\mathsf{a}(X)$ is any basic literal and the following one, separated by ":", must be based on a *domain predicate*, i.e., a predicate that can completely evaluated at grounding time. There may be more than one condition separated by further ":" symbols. Also, new variables may be introduced in the conditions. The conditional literal is equivalent to producing a collection of literals as given by the first literal, where the variables must hold for all the literals following the ":". In our case we limit the use of conditional literals to literals in the body of the rule and do not allow them in the head. In this case the conditional literal is equivalent to a *parametrized conjunction*. The variables within the conjunction must adhere to the bound variables elsewhere in the rule. For example, consider the following rule:

$$\mathsf{a}(X) \leftarrow \mathsf{b}(X),\ \mathsf{c}(X,Y)\ :\ \mathsf{d}(Y,Z).$$

The first item in the body $\mathsf{b}(X)$ will define the value of X in the conjunction $\mathsf{c}(X,Y)\ :\ \mathsf{d}(Y,Z)$, but Y and Z are constrained only by the conditions within the conjunction. Here Y and Z are local variables and the conjunction contains a member $\mathsf{c}(X,Y)$ for every $\mathsf{d}(Y,Z)$ which is true according to other rules of the program.

Example 3.1.5 *We show the use of parametrized conjunction by continuing to extend the previous example for the graph B in (3.3) towards computing the Hamiltonian cycle for it. First we define some projections which result to domain predicates* node*:*

$$\mathsf{node}(X) \leftarrow \mathsf{edge}(X,Y).$$
$$\mathsf{node}(Y) \leftarrow \mathsf{edge}(X,Y).$$

Next we define two mutually referring rules, which state that we include the edge to the solution cycle if it is not already present in it:

$$\begin{aligned}\mathsf{oncycle}(X,Y) &\leftarrow \mathsf{edge}(X,Y),\ \mathbf{not}\ \mathsf{other}(X,Y).\\ \mathsf{other}(X,Y) &\leftarrow \mathsf{edge}(X,Y),\ \mathsf{edge}(X,Z),\ Y\ !=Z,\ \mathsf{oncycle}(X,Z).\end{aligned}$$

Now we constrain the solution stating that no node can exist outside of a cycle:

$$\leftarrow\ \mathsf{node}(X),\ \mathbf{not}\ \mathsf{oncycle}(Y,X)\ :\ \mathsf{edge}(Y,X). \tag{3.4}$$

Here we use the parametrized conjunction to form a number of negated oncycle *facts. In order to demonstrate the effects better, we introduce more edges:*

$$\mathsf{edge}(3,2).\quad \mathsf{edge}(2,1).\quad \mathsf{edge}(1,3).$$

Intuitively, the previous data in (3.3) described a loop over three nodes; these new edges represent another loop going in the opposite direction with respect to the former loop. In *this case the Hamiltonian cycles for this graph coincide with the two loops. From grounding perspective the parametrized conjunction in (3.4) will result in expanding the* oncycle *predicate within the constraint itself, but also in producing multiple constraints:*

$$\begin{aligned}
&\leftarrow \ \mathbf{not}\ \mathsf{oncycle}(1,3),\ \mathbf{not}\ \mathsf{oncycle}(2,3).\\
&\leftarrow \ \mathbf{not}\ \mathsf{oncycle}(3,2),\ \mathbf{not}\ \mathsf{oncycle}(1,2).\\
&\leftarrow \ \mathbf{not}\ \mathsf{oncycle}(2,1),\ \mathbf{not}\ \mathsf{oncycle}(3,1).
\end{aligned}$$

The complete rules for calculating Hamiltonian cycles in a graph are given in Appendix A, however the essence of the solution is in the above example.

Although there are other syntactical constructs for answer set programs, most notably choice rules and weight rules, the set of concepts we have presented here are sufficient for defining the metaevaluation.

We give another example use case for applying answer set programming below.

Example 3.1.6 *Consider the following two rules which formalize an access policy:*

$$\begin{aligned}
\mathsf{denied}(C,S) &\leftarrow \mathbf{not}\ \mathsf{granted}(C,S),\ \mathsf{customer}(C),\ \mathsf{service}(S). && (3.5)\\
\mathsf{granted}(C,S) &\leftarrow \mathsf{registered}(C),\ \mathsf{subscribed}(C,S),\\
&\qquad \mathsf{customer}(C),\ \mathsf{service}(S). && (3.6)
\end{aligned}$$

The intuitive reading of the rules is as follows. The access to a service is denied by default and granted only for registered customers who have subscribed to the service. It is easy to add further reasons why the access should be granted. This illustrates how default negation can be used to concisely encode exceptions; instead of explicitly enumerating all possibilities of granted and non-granted instances, we can halve this work. Given a particular customer and a service, denoted by constants j and g, the rule (3.5) instantiates to $\mathsf{denied}(j,g) \leftarrow \mathbf{not}\ \mathsf{granted}(j,g),\ \mathsf{customer}(j),\ \mathsf{service}(g)$ *under* $[C/j, S/g]$.

3.2 Reification

In this section, we describe the structure of abstract syntax trees which we publish for others to use and evaluate. The reification is performed by a tool, `aspreify` [5] . All of the examples here have been produced by this tool, however, for some images visual modifications have been applied to ease readability. The overall procedure is straightforward: the rules are parsed into a data structure, a *parse tree*, which is then represented as ASP facts. This representation turns a rule into data accessible by meta-level rules as facts.

In Table B.1 we enumerate the used facts and their purpose. Each non-leaf node of the parse tree has been assigned a unique identifier. For present, it is sufficient to use integers as identifiers, however it may become necessary to qualify these identifiers in order to distinguish them of content by other potential sources. The distinction can be achieved by using name spaces based on the name of the owner or host of the rules, which can be encoded as Universal Resource Identifiers [12], which support this kind of hierarchical representation. Also, integers can be replaced by practically unique random identifiers, such as Universally

[5] `http://www.github.com/vluukkal/aspreify` along with visualization utilities

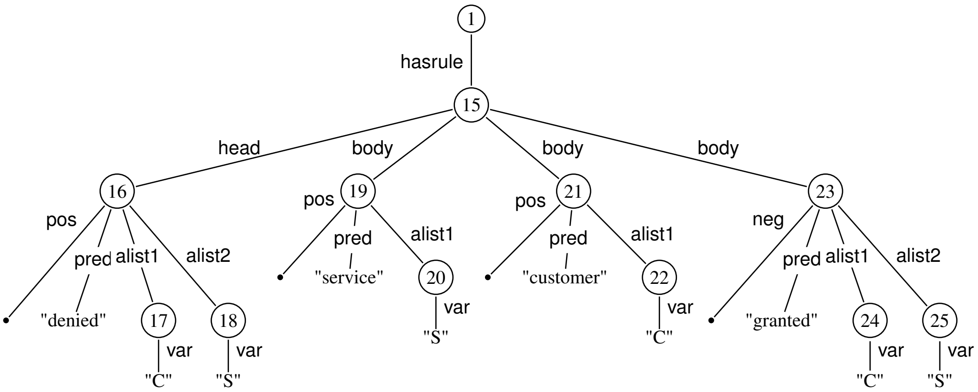

Figure 3.1: Parse tree of a single rule.

Unique Identifiers (UUID) [70] for which there are widely available standard implementations. We have made a design choice of not making any equivalence checks between the nodes, so that we may have a different identifier for an otherwise similar sub-tree. This is most evident in the representation of variables or constants, where nodes representing the same variable or constant within the same rule would get different identifiers. Changing the design so that each variable corresponds to a unique subtree is possible, yet since one of our goals is to enable different provenance for different facts while keeping different identifiers for syntactically identical entities, this design choice is justified. The declarative nature of ASP insists that the orders of rules in the program and the literals within rule bodies are not significant. Thus we take the opportunity to abstract the parse tree by not representing the original syntactic order but only listing the bodies. Eventually in the metagrounding phase we find this information useful for bookkeeping and hence such information is added. When the ordering information must be maintained (e.g., for the arguments of predicates), it is explicitly encoded using an indexing scheme: for example predicate argument is represented by $\mathsf{alist}(\cdot, n, \cdot)$ where n gives the position of an argument. The procedure is further illustrated by the following example.

Example 3.2.1 *The rule (3.5) from Example 3.1.6 can be parsed into a syntax tree shown in Figure 3.1. Note that here nodes representing variable "C" within the same rule will get different identifiers* 17*,* 22*, and* 24*. A root node, identified as the node* 1 *in the tree, collects all parsed rules under* hasrule *arcs which connect the individual rules, facts, and constraints of the program. Each rule has exactly one* head*-labeled sub-tree whereas arbitrarily many* body*-labeled sub-trees are allowed. The latter represent the body conditions of the rule. The tree shown in Figure 3.2 can be represented as a set of facts producing the* reified *rule. The predicates* $\mathsf{head}(\cdot, \cdot)$ *and* $\mathsf{body}(\cdot, \cdot)$ *associate the head and body conditions, respectively, with the rule. The predicate* $\mathsf{pred}(\cdot, \cdot)$ *is used to attach predicate symbols—treated as strings—for these conditions. Predicates* $\mathsf{pos}(\cdot)$ *and* $\mathsf{neg}(\cdot)$ *express whether the original atom is negated or not. The predicate* $\mathsf{alist}(\cdot, \cdot, \cdot)$ *represents the argument list: the order of* n *arguments is encoded by selecting the second argument from the range* $1 \ldots n$*. The third argument refers to another node capturing either a constant or a variable expressed in terms of predicates* $\mathsf{const}(\cdot, \cdot)$ *and* $\mathsf{var}(\cdot, \cdot)$*. Note that for visualization purposes the argument index has been syntactically collapsed with the* alist *identifier into one arc in the graph figure. The* $\mathsf{pos}(\cdot)$ *and* $\mathsf{neg}(\cdot)$ *predicates have been given a dummy node in the graph.*

```
hasrule(1, 15).
rule(15).
pos(16).
head(15, 16).
pred(16, "denied").
var(17, "C").
alist(16, 1, 17).
var(18, "S").
alist(16, 2, 18).
pos(19).
body(15, 19).
pred(19, "service").
var(20, "S").
alist(19, 1, 20).
pos(21).
body(15, 21).
pred(21, "customer").
var(22, "C").
alist(21, 1, 22).
neg(23).
body(15, 23).
pred(23, "granted").
var(24, "C").
alist(23, 1, 24).
var(25, "S").
alist(23, 2, 25).
```

Figure 3.2: Reified representation for a rule.

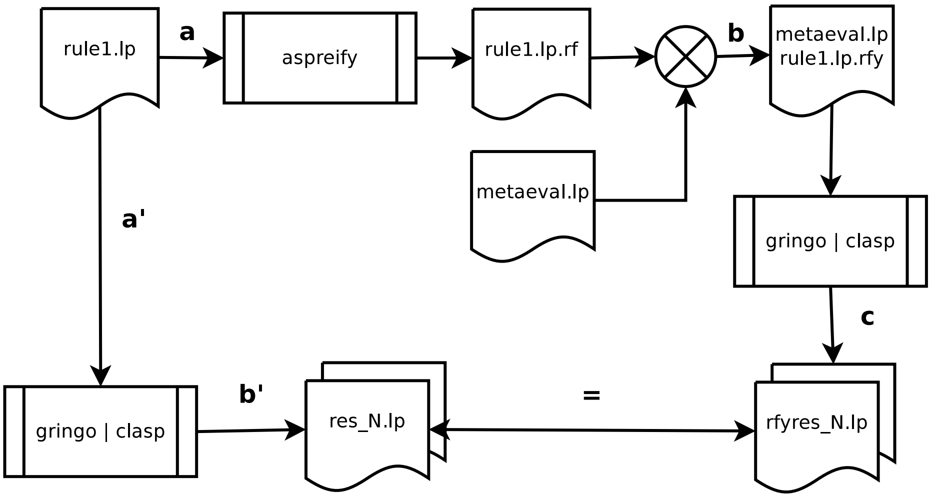

Figure 3.3: Overview of the metaevaluation process compared to plain evaluation

As mentioned earlier, it is possible to create an ontology definition for rules (in RDFS or OWL) corresponding to the concepts above in a straightforward way as explained in [78]. Existing efforts such as [17] contain definitions of the syntactic elements of rules enabling the presentation of rules by their syntax trees (e.g., in Figure 3.1), which can then be stored and shared. We present an example of a complete reified ASP program with more syntactic elements in Appendix A. We list some properties of the reified programs.

Proposition 3.2.1 *For a normal program P its reified representation* $\mathrm{Rfy}(P)$*:*

1. *The set* $\mathrm{Rfy}(P)$ *consists of ground atoms.*
2. *The set* $\mathrm{Rfy}(P)$ *is finite if P is finite.*
3. *When a rule in P refers to multiple items for the same predicates, facts, or terms in different parts of the rule, they will have different unique identifiers in* $\mathrm{Rfy}(P)$.

3.3 Metaevaluation

Metaevaluation is the evaluation of the reified answer set programming rules by means of another answer set program, the *metaevaluator*. The evaluation is performed by the same

ASP toolchain as the evaluation of the non-reified program. The metaevaluation process schematically illustrated in Figure 3.3.

(a) The non-reified set of rules is processed by `aspreify` producing reified representation of the source rules.

(b) The resulting set of facts is combined with the metaevaluator to form another set of rules. At this point we could bring in other rules or facts or filter the results of the reification.

(c) Standard ASP tools calculate the answer sets of the combined rules, which correspond to the answer sets that would be produced by directly evaluating the original rules (steps **a'** and **b'**).

We continue by describing an ASP metaevaluator for a representative subset of the ASP language. We prove that it produces the same stable models for reified programs as the direct evaluation of unreified ASP programs. Here we limit ourselves to *propositional* normal programs, i.e., all predicates are of arity zero. The used reified constructs are limited to $\mathsf{head}(\cdot,\cdot)$, $\mathsf{body}(\cdot,\cdot)$, $\mathsf{pred}(\cdot,\cdot)$, $\mathsf{neg}(\cdot)$, $\mathsf{pos}(\cdot)$ and $\mathsf{rule}(\cdot)$, which can represent all propositional normal programs.

The key idea is to introduce an auxiliary predicate $\mathsf{in}(H)$ for each rule head H. The intuitive reading of $\mathsf{in}(H)$ is that H belongs to an answer set being formalized. According to the *metaevaluation* rule (3.7) below, the atom $\mathsf{in}(H)$ is derived if and only if $\mathsf{in}(P)$ is derivable for all positive body conditions P of the rule but $\mathsf{in}(N)$ is not derivable for any negative body conditions N. We use the parametrized conjunction operator ":", which is expanded during the grounding for each positive atom in the body and likewise for each negative atom in the body. This mechanism works irrespective of the number of positive or negative atoms in the rule body, making the parametrized conjunction operator a key element of tooling.

$$\begin{aligned}\mathsf{in}(H) \;\leftarrow\; & \mathsf{rule}(R),\, \mathsf{head}(R,H),\, \mathsf{in}(P) : \mathsf{body}(R,P) : \mathsf{pos}(P),\\ & \mathbf{not}\ \mathsf{in}(N) : \mathsf{body}(R,N) : \mathsf{neg}(N). \end{aligned} \tag{3.7}$$

Since a predicate may have several occurrences in a particular rule, we need auxiliary rules to ensure that all instances of the predicate are in the corresponding relation $\mathsf{in}(P)$, essentially synchronizing the $\mathsf{in}(\cdot)$ predicates for the non-reified atoms. We consider two predicates to be the same if their name coincide (3.11), even if they have a different identifier (3.10) in the syntax tree. Both positive (3.8) and negative (3.9) instances are considered. Finally (3.11) ensures that all predicates considered to be the same are covered.

$$\begin{aligned}
\mathsf{atom}(A) &\;\leftarrow\; \mathsf{pos}(A). && (3.8)\\
\mathsf{atom}(A) &\;\leftarrow\; \mathsf{neg}(A). && (3.9)\\
\mathsf{equal}(A,A) &\;\leftarrow\; \mathsf{atom}(A). && (3.10)\\
\mathsf{same}(A_1,A_2) &\;\leftarrow\; \mathsf{pred}(A_1,P),\, \mathsf{pred}(A_2,P),\\
&\qquad \mathsf{atom}(A_1),\, \mathsf{atom}(A_2),\, \mathbf{not}\ \mathsf{equal}(A_1,A_2). && (3.11)\\
\mathsf{in}(A_1) &\;\leftarrow\; \mathsf{in}(A_2),\, \mathsf{same}(A_1,A_2). && (3.12)
\end{aligned}$$

The rule (3.12) causes a quadratic blow up during grounding. We could avoid this by modifying the reification to produce one identifier for the same predicate, allowing us to ignore the above rules. However, we want to allow different provenance for same predicates and hence we must accept this cost.

3.3.1 Mathematical Analysis of Metaevaluation

In this section we present an analysis of metaevaluation, showing that our approach essentially produces the same answer sets as the direct evaluation. We make some simplifying assumptions in order to make the proofs shorter. Firstly, we consider *normal* and *propositional programs* only, i.e., all literals are atoms with arity zero, where the predicate symbols coincide with the atoms. Secondly we consider *finite* programs and thus also finite reified programs. The results shows that our metaprogramming approach adheres to the stable model semantics and hence for our subset of ASP we can replicate results of earlier metaevaluation work such as [45].

We establish a syntactic link between the ground rules and their reified representation. This is done by mapping the syntactic elements of ground programs to syntactic elements of the reified program, which are ground together with rules for the metaevaluator.

As mentioned previously each referral to a variable or constant in a rule will get a unique identifier regardless of its name. For example atom a in head, same atom a in body or negated atom $\mathbf{not}\ a$ in body all refer to the same atom in the rule, but they will get a distinct identifier thereby we can always map atoms back to the particular source component within the rule. We use the following convention for referring to identifiers generated by the reification process for atoms: $\#_i^r(a)$ is a unique integer identifier which corresponds to atom a occurring in unreified rule $r \in P$ of form (3.1) in the ith literal in the body. The case $i = 0$ is reserved for the head occurrence. Additionally for convenience we write that *an atom a has an i-occurrence in r*, when we have an atom a occurring syntactically in the ith body literal of $r \in P$.

For completeness we introduce notation for identifiers of reified rules: $\#(r)$ refers to the unique identifier assigned to a reified rule r.

Example 3.3.1 *Consider a rule r: $z \leftarrow z, \mathbf{not}\ z$, its representation using our convention alongside its raw reification:*

$\mathsf{rule}(\#(r)).$	$\#(r) = 2$	$\mathsf{rule}(2).$
$\mathsf{head}(\#(r), \#_0^r(z)).$	$\#_0^r(z) = 3$	$\mathsf{head}(2, 3).$
$\mathsf{pred}(\#_0^r(z), "z").$		$\mathsf{pred}(3, "z").$
$\mathsf{pos}(\#_0^r(z)).$		$\mathsf{pos}(3).$
$\mathsf{body}(\#(r), \#_1^r(z)).$	$\#_1^r(z) = 4$	$\mathsf{body}(2, 4).$
$\mathsf{pred}(\#_1^r(z), "z").$		$\mathsf{pred}(4, "z").$
$\mathsf{pos}(\#_1^r(z)).$		$\mathsf{pos}(4).$
$\mathsf{body}(\#(r), \#_2^r(z)).$	$\#_2^r(z) = 5$	$\mathsf{body}(2, 5).$
$\mathsf{pred}(\#_2^r(z), "z").$		$\mathsf{pred}(5, "z").$
$\mathsf{neg}(\#_2^r(z)).$		$\mathsf{neg}(5).$

We have explicitly given the correspondences between the identifiers in the middle column. In contrast to Example 3.2 some facts are not included in favor of readability.

Note that an identifier $\#_i^r(a)$ is a metalevel reference to its non-reified counterpart, i.e., the atom a – metalevel in the sense that the identifier can always be mapped back to its counterpart. We use this reference to keep track of the counterpart later in this section. This is different from facts of form $\mathsf{pred}(\#_2^r(z), "z")$ which is a syntactical artifact of the reification, stating that the identifier is related to a non-reified atom with predicate z. On the reified side, the relevant parts of the answer sets will contain of atoms of form $\mathsf{in}(\#_2^r(z))$, whereas on the non-reified side they will consist of the atoms themselves.

Let us define a function which maps a non-reified rule to the reified representation. The function f defines a link between the "static" part of the reified representation and non-reified rules; the reified facts will always be present in reified representation (independent of any reduct) and they can always be mapped back to a non-reified rule. The identifiers are unique so obtaining f^{-1} is straightforward.

Definition 3.3.1 *Let P be a propositional normal logic program and let rule $r = a \leftarrow b_1, \ldots, b_n, \mathbf{not}\ c_{n+1}, \ldots, \mathbf{not}\ c_m$ be a rule of P.*

Let $\#(r), \#_0^r(a), \#_1^r(b_1), \ldots, \#_n^r(b_n), \#_{n+1}^r(c_{n+1}), \ldots, \#_m^r(c_m)$ be unique identifiers. We define function a f by setting:

$$\begin{aligned} f(r) = \{\ & \mathsf{head}(\#(r), \#_0^r(a)), \mathsf{pred}(\#_0^r(a), "a"), \mathsf{pos}(\#_0^r(a)), \\ & \mathsf{body}(\#(r), \#_1^r(b_1)), \mathsf{pred}(\#_1^r(b_1), "b_1"), \mathsf{pos}(\#_1^r(b_1)), \ldots \\ & \quad \mathsf{body}(\#(r), \#_n^r(b_n)), \mathsf{pred}(\#_n^r(b_n), "b_n"), \mathsf{pos}(\#_n^r(b_n)), \\ & \mathsf{body}(\#(r), \#_{n+1}^r(c_{n+1}), \mathsf{pred}(\#_{n+1}^r(c_{n+1}), "c_{n+1}"), \mathsf{neg}(\#_{n+1}^r(c_{n+1})), \ldots \\ & \quad \mathsf{body}(\#(r), \#_m^r(c_m)), \mathsf{pred}(\#_m^r(c_m), "c_m"), \mathsf{neg}(\#_m^r(c_m))\ \}. \end{aligned} \tag{3.13}$$

The reification of P is $\mathrm{Rfy}(P) = \bigcup_{r \in P} f(r)$.

We define functions which link the reified and non-reified rules. The $\mathsf{in}(\cdot)$ facts that are derived by the metaevaluation represent the reified answer sets which are compatible with the non-reified answer sets of the non-reified rules. There are other facts produced by the metaevaluation rules (3.8)–(3.11), but they are auxiliary while the interesting part is formed by the $\mathsf{in}(\cdot)$ facts. Whereas the previous Definition 3.3.1 pertained to "static" information, the following ones map between the "dynamic" information containing the essence of answer sets.

Definition 3.3.2 *Let P be a propositional normal logic program, $R = \mathrm{Rfy}(P)$ its reification, Q the metaprogram consisting of rules (3.7)–(3.12) and let $G = \mathrm{Gnd}(R \cup Q)$. Furthermore let $\#(r), \#_0^r(a), \#_1^r(b_1), \ldots, \#_n^r(b_n), \#_{n+1}^r(c_{n+1}), \ldots, \#_m^r(c_m)$ be unique identifiers. We define function g which maps rule $r = a \leftarrow b_1, \ldots, b_n, \mathbf{not}\ c_{n+1}, \ldots, \mathbf{not}\ c_m \in P$ to its reified representation:*

$$\begin{aligned} g(r) = \mathsf{in}(\#_0^r(a_0)) \leftarrow & \ \mathsf{in}(\#_1^r(b_1)), \ldots \mathsf{in}(\#_n^r(b_n))), \\ & \mathbf{not}\ \mathsf{in}(\#_{n+1}^r(c_{n+1}))), \ldots \mathbf{not}\ \mathsf{in}(\#_m^r(c_m)). \end{aligned} \tag{3.14}$$

Function g is bijective, but for convenience we define a function h for mapping reified atoms in rule heads to corresponding unreified rule:

$$h(\mathsf{in}(\#_0^r(a))) = r \in P. \tag{3.15}$$

The mechanism being presented grounds a representation of rules by means of rules. However, there is a bootstrap problem: we need an existing implementation of grounding and solving, which necessitates a reliance on properties of available tools. We cannot mathematically prove the following proposition, but based on our intuition on how a reasonable grounder works, we give the following properties:

Proposition 3.3.1 *Let P be a propositional normal logic program, $R = \mathrm{Rfy}(P)$ its reification and Q the meta program consisting of rules (3.7)–(3.12). The ground program $G = \mathrm{Gnd}(R \cup Q)$ contains following elements:*

1. *The reified rules of P as ground facts, i.e., the elements of R.*
2. *The ground facts produced by grounding rules (3.8)–(3.11) of form:*
   (a) $\mathsf{atom}(\#^r_i(a))$ *for every i-occurrence of an atom a in rule $r \in P$.*
   (b) $\mathsf{equal}(\#^r_i(a), \#^r_i(a))$ *for every i-occurrence of an atom a in rule $r \in P$.*
   (c) $\mathsf{same}(\#^r_i(a), \#^{r'}_j(a))$ *for every atom a which has an i-occurrence in r and has an j-occurrence in r', such that $r \neq r'$ or $i \neq j$.*
3. *The rule $g(r)$ of the form (3.14) for every rule $r \in P$ as given by (3.1), corresponding to the grounding at the metarule (3.7) for $f(r)$.*
4. *The rules of form*

$$\mathsf{in}(\#^r_i(a)) \leftarrow \mathsf{in}(\#^{r'}_j(a)).$$

corresponding to metarule (3.12) for both $f(r)$ and $f(r')$ where a has an i-occurrence in r and also an j-occurrence in r', such that $r \neq r'$ or $i \neq j$.

Furthermore we define function $F(G)$ to return the ground facts specified in items 1 and 2 above.

The ground fact $\mathsf{equal}(\cdot,\cdot)$ is an auxiliary literal needed for definition of $\mathsf{same}(\cdot,\cdot)$. The rule (3.12) is used for synchronizing across reified identifiers produced in the metaevaluated answer set such that all identifiers referring to the same non-reified atom have the same truth values. The facts $\mathsf{same}(\cdot,\cdot)$ and $\mathsf{equal}(\cdot,\cdot)$ are statically determined by grounding rules (3.11) and (3.10), but the $\mathsf{in}(\cdot)$ facts are produced by rules (3.7) and (3.12) which are grounded as rules rather than facts. We call the ground rules produced by rules (3.7) and (3.10) *ground metarules*.

Next we define functions which map between unreified atoms and the reified representations via the critical $\mathsf{in}(\cdot)$ atoms.

Definition 3.3.3 *Using the same definitions as in Proposition 3.3.1 we define functions t and s for mapping between essentially a non-reified atom its a reified identifier:*

$$t(a) = \{\mathsf{in}(\#^r_i(a)) \mid a \textit{ has an } i\textit{-occurrence in } r \in P\} \text{ and} \tag{3.16}$$

$$s(\mathsf{in}(\#^r_i(a))) = \textit{For an atom a having an } i\textit{-occurence in } \; r \in P \tag{3.17}$$

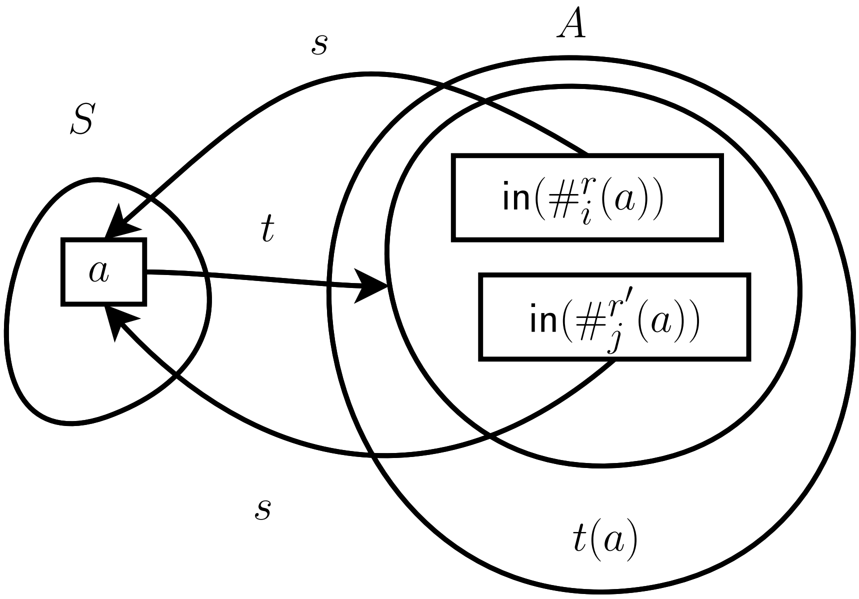

Figure 3.4: Mappings between plain and reified atoms in sets S and A respectively.

We overload s *and* t *to operate on items in sets:*

$$t(A) = \bigcup_{a \in A} t(a) \text{ and} \tag{3.18}$$

$$s(A) = \{s(a) \mid a \in A\}. \tag{3.19}$$

Using the above we can obtain a full description (all related atoms) of an unreified rule from any reified in, pred, head or body fact.

The following theorem establishes the link between the answer sets of the reified program and the non-reified program.

Theorem 3.3.2 *Let* P *be a propositional normal logic program,* $R = \mathrm{Rfy}(P)$ *its reification, and* Q *the metaprogram consisting of rules (3.7)–(3.12) above. Also let* $G = \mathrm{Gnd}(R \cup Q)$ *and* $F(G) \subseteq G$ *the facts in G. If* S *is an answer set of* P*,* $S = \mathrm{LM}(P^S)$*, then* $R \cup Q$ *has an answer set*

$$A = F(G) \cup \{\mathsf{in}(\#_i^r(a)) \mid a \in S \textit{ has an } i\textit{-occurrence in } r \in P\} \tag{3.20}$$

and $A = \mathrm{LM}(\mathrm{Gnd}(R \cup Q)^A) = \mathrm{LM}(G^A)$.

Essentially we are saying that if there is an element $\mathsf{in}(\#_i^r(a)) \in A$ in the reified answer set, then that element corresponds to an element $a \in S$ in the non-reified answer set. In other words the metaevaluator operating on the reified program can simulate the original program. The relation between the sets S and A and especially the notion of having several reified identifiers correspond to the same non-reified atom as presented in Figure 3.4.

As the same atom may have several occurrences in the rule and thus several identifiers, synchronization between these identifiers is one of the key issues here. The metaevaluation rules have a dedicated synchronization rule (3.12) for ensuring that if some reified representation of an atom ends up in an answer set, then all of the reified representations with different identifiers, but referring to same name and also included in the answer set. This notion is the essence of the auxiliary lemma given below.

Lemma 3.3.3 *Let P be a propositional normal logic program, $R = \mathrm{Rfy}(P)$ its reification, Q the metaprogram consisting of rules (3.7)–(3.12) above and $G = \mathrm{Gnd}(R \cup Q)$. Let A be the set of atoms defined by (3.20) and $S = \mathrm{LM}(P^S)$.*

If P contains two rules r and r' such that atom a occurs in both of them in positions i and j respectively, then $A \models \mathsf{in}(\#^r_i(a)) \leftarrow \mathsf{in}(\#^{r'}_j(a))$.

Proof For rules $r \in P$ and $r' \in P$, which have the same atom a we assume that $A \not\models \mathsf{in}(\#^r_i(a)) \leftarrow \mathsf{in}(\#^{r'}_j(a))$, or that $\mathsf{in}(\#^r_i(a)) \notin A$, but $\mathsf{in}(\#^{r'}_j(a)) \in A$. Since $\mathsf{in}(\#^{r'}_j(a)) \in A$, by the definition of A (3.20) $a \in S$. But according to the same definition a has an occurrence in r, specifically $\mathsf{in}(\#^r_i(a)) \in A$, a contradiction. □

Now we proceed with the proof of the actual theorem.

Proof We want to prove $A = \mathrm{LM}(G^A)$ given $S = \mathrm{LM}(P^S)$. Note that $R = \mathrm{Rfy}(P)$ has produced a set of ground facts as explained in Section 3.2 and their reduct with respect to the model remains the same. The only nontrivial rules in $G = \mathrm{Gnd}(R \cup Q)$ are the result of grounding (3.7) and (3.12) where the heads will be of form $\mathsf{in}(\cdot)$.

1. $\mathrm{LM}(G^A) \subseteq A$: By the least model (3.1.1) being an intersection of all models, it is sufficient to show $A \models G^A$. We assume by contradiction that $A \not\models G^A$. The difference cannot be due to any ground facts of $F(G)$ because they are included in A by the definition (3.20). By Lemma 3.3.3 the difference cannot be due to an instance of rule (3.12) but it can only be caused by an instance of rule (3.7).

   (a) Thus by proposition (3.3.1) there must exist a rule $g(r)$:

   $$\begin{aligned}\mathsf{in}(\#^r_0(a)) \leftarrow &\mathsf{in}(\#^r_1(b_1)),\ \ldots,\ \mathsf{in}(\#^r_n(b_n)),\\ &\mathsf{in}(\#^r_{n+1}(c_{n+1})),\ \ldots,\ \mathsf{in}(\#^r_m(c_m)) \in G\end{aligned}$$

   and the corresponding rule in the reduct:

   $$\mathsf{in}(\#^r_0(a)) \leftarrow \mathsf{in}(\#^r_1(b_1)),\ \ldots,\ \mathsf{in}(\#^r_n(b_n)) \in G^A$$

   such that $\mathsf{in}(\#^r_0(a)) \notin A$ but $\{\mathsf{in}(\#^r_1(b)),\ \ldots,\ \mathsf{in}(\#^r_n(b))\} \subseteq A$ and $\{\mathsf{in}(\#^r_{n+1}(c_{n+1})), \ldots, \mathsf{in}(\#^r_m(c_m))\} \cap A = \emptyset$.

   (b) We map the set $\{\mathsf{in}(\#^r_1(b_1)) \ldots, \mathsf{in}(\#^r_n(b_n))\}$ to non-reified side, and by the definition of A in (3.20)

   $$s(\{\mathsf{in}(\#^r_1(b_1)), \ldots, \mathsf{in}(\#^r_n(b_n))\}) = \{b_1, \ldots, b_n\} \subseteq S.$$

   (c) We identify the corresponding non-reified rule

   $$h(\mathsf{in}(\#^r_0(a))) = r = a \leftarrow b_1, \ldots, b_n, \mathbf{not}\ c_{n+1}, \ldots, \mathbf{not}\ c_m \in P$$

   where $b_1 = s(\mathsf{in}(\#^r_1(b_1))), \ldots, b_n = s(\mathsf{in}(\#^r_n(b_n)))$ and $c_{n+1} = s(\mathsf{in}(\#^r_{n+1}(c_{n+1}))), \ldots, c_m = s(\mathsf{in}(\#^r_m(c_m)))$.

(d) The set $S = \mathrm{LM}(P^S)$ is the least model, and since

$$\{\mathsf{in}(\#^r_{n+1}(c_{n+1})), \ldots, \mathsf{in}(\#^r_m(c_m))\} \cap A = \emptyset,$$

then $\{c_{n+1}, \ldots, c_m\} \cap S = \emptyset$ by the definition of A in (3.20).

(e) Because $\{b_1, \ldots, b_n\} \subseteq S$ and $S = \mathrm{LM}(P^S)$ is the least model, the reduct of $h(\mathsf{in}(\#^r_0(a)))^S = a \leftarrow b_1, \ldots, b_n$ forces $a \in S$.

(f) By equation (3.13) and by (3.20) atom $\mathsf{in}(\#^r_0(a)) \in A$, a contradiction. Thus $A \models G^A$.

2. $A \subseteq \mathrm{LM}(G^A)$: We use the T_{P^S} operator and the related sequence indexed by l as defined in Definition 3.1.6 and characterized by Theorem 3.1.2. For each element of this sequence we can project its contents to the reified side by using the function t from (3.16) and for each of the steps we prove that this projection is included in the least model of the reduct of the reified program.

   We prove by induction on the length l of the sequence $\mathrm{T}_{P^S} \uparrow l$:

   (a) Base case: $t(\mathrm{T}_{P^S} \uparrow 0) \subseteq \mathrm{LM}(G^A)$. By definition $\mathrm{T}_{P^S} \uparrow 0 = \emptyset$ and $t(\emptyset) = \emptyset$ and $\emptyset \subseteq \mathrm{LM}(G^A)$.

   (b) Inductive hypothesis $t(\mathrm{T}_{P^S} \uparrow l) \subseteq \mathrm{LM}(G^A)$.

   (c) Inductive step $t(\mathrm{T}_{P^S}(\mathrm{T}_{P^S} \uparrow l)) = \mathrm{T}_{P^S} \uparrow l + 1 \subseteq \mathrm{LM}(G^A)$.

      i. Pick $\mathsf{in}(\#^r_i(a)) \in t(\mathrm{T}_{P^S}(\mathrm{T}_{P^S} \uparrow l))$ for some r.

      ii. We map the reified atom to the non-reified side: $a = s(\mathsf{in}(\#^r_i(a))) \in \mathrm{T}_{P^S}(\mathrm{T}_{P^S} \uparrow l)$.

      iii. By definitions of T_{P^S} and reduct there must be a rule $r' : a \leftarrow b_1, \ldots, b_n,$ $\mathbf{not}\ c_{n+1}, \ldots, \mathbf{not}\ c_m \in P$ such that $\{b_1, \ldots, b_n\} \subseteq \mathrm{T}_{P^S} \uparrow l$ and $\{c_{n+1}, \ldots, c_m\} \cap S = \emptyset$. We map the set of negative body atoms to reified side $t(c_{n+1}) \cup \ldots \cup t(c_m)$

      iv. By the definition of A in (3.20), it follows that $t(\{c_{n+1}, \ldots, c_m\}) \cap A = \emptyset$. To see this assume by contradiction that $\mathsf{in}(\#^{r'}_j(c_j)) \in t(\{c_{n+1}, \ldots, c_m\})$ and $\mathsf{in}(\#^{r'}_j(c_j)) \in A$. By the definition of A in (3.20), there must be $c_j \in S$ and $\mathsf{pred}(\#^{r'}_j(c_j), "c_j") \in R$ and also $c_j \in \{c_{n+1}, \ldots, c_m\}$, which contradicts item (iii).

      v. We map rule r' to the reified side via $g(r') \in G$:
      $$\mathsf{in}(\#^{r'}_0(a)) \leftarrow \mathsf{in}(\#^{r'}_1(b_1)), \ldots, \mathsf{in}(\#^{r'}_n(b_n)) \in G^A.$$

      vi. We map the set of positive body atoms to reified side $\bigcup_{i=1}^n t(b_i) \subseteq t(\mathrm{T}_{P^S} \uparrow l)$ and furthermore $t(\{c_{n+1}, \ldots, c_m\}) \cap A = \emptyset$.

      vii. By inductive hypothesis, $\{\mathsf{in}(\#^{r'}_1(b_1)), \ldots, \mathsf{in}(\#^{r'}_n(b_n))\} \subseteq \mathrm{LM}(G^A)$.

      viii. By metaevaluation rule (3.7) with $g(r)$ and $\{\mathsf{in}(\#^r_1(b_1)), \ldots, \mathsf{in}(\#^r_n(b_n))\} \subseteq \mathrm{LM}(G^A)$ along with the reification $f(r)$ (3.13) and with Proposition 3.3.1 item 4 we can deduce $\mathsf{in}(\#^r_i(a)) \in \mathrm{LM}(G^A)$, hence $t(\mathrm{T}_{P^S}(\mathrm{T}_{P^S} \uparrow l)) \subseteq \mathrm{LM}(G^A)$.

   (d) By the definition of T_P in Definition 3.1.6, $t(S) = t(\mathrm{lfp}(\mathrm{T}_{P^S})) \subseteq \mathrm{LM}(G^A)$, hence $A \subseteq \mathrm{LM}(G^A)$. □

Theorem 3.3.4 *Let P be a propositional normal logic program, $R = \mathrm{Rfy}(P)$ its reification, and Q the metaprogram consisting of rules (3.7)–(3.12) above. Also let $G = \mathrm{Gnd}(R \cup Q)$ and $r \in P$. If $A = \mathrm{LM}(G^A)$ then $S = \mathrm{LM}(P^S)$ where*

$$S = \{a \in \mathrm{Hb}(P) \mid \ a \textit{ has an } i\textit{-occurrence in } r \in P \textit{ and } \mathsf{in}(\#_i^r(a)) \in A\}. \qquad (3.21)$$

We give a lemma stating that a reified representation of an atom a that has an i-occurrence in r in the reified answer set forces all other reified representations of occurrences of a to be in the same answer set. Conversely, if there is no reified representation of an atom a in the reified answer set, then no other representation can be there either. This is a necessary synchronization as an unreified atom may have several representations with distinct identifiers on the reified side.

Lemma 3.3.5 *Let P be a propositional normal logic program, $R = \mathrm{Rfy}(P)$ its reification, Q the metaprogram consisting of rules (3.7)–(3.12) above, $G = \mathrm{Gnd}(R \cup Q)$ and $A = \mathrm{LM}(G^A)$. Then for an atom a having an i-occurrence in $r \in P$ and a j-occurrence in $r' \in P$:*

1. $\mathsf{in}(\#_i^r(a)) \in A$ *iff* $\mathsf{in}(\#_j^{r'}(a)) \in A$ *and*
2. $t(a) \cap A \neq \emptyset$ *iff* $t(a) \subseteq A$.

Proof

1. We prove the if-direction, the only-if-direction follows by symmetry. Given an atom a having an i-occurrence in r and a j-occurrence in r' we assume that $\mathsf{in}(\#_j^{r'}(a)) \in A$ but $\mathsf{in}(\#_i^r(a)) \notin A$. By assumption $A = \mathrm{LM}(G^A)$ and all ground instances $\mathsf{in}(\#_i^r(a)) \leftarrow \mathsf{in}(\#_j^{r'}(a))$ of rule (3.12) are in G^A. But since $A \models G^A$, then $A \models \mathsf{in}(\#_i^r(a)) \leftarrow \mathsf{in}(\#_j^{r'}(a))$ so that $\mathsf{in}(\#_i^r(a)) \in A$, a contradiction.
2. Since we assume a to have occurrences in r, $t(a) \neq \emptyset$ by its definition (3.16). For one direction assuming $t(a) \cap A = \emptyset$ but $t(a) \subseteq A$ leads to an immediate contradiction. For the other direction, suppose that $t(a) \not\subseteq A$ but $t(a) \cap A \neq \emptyset$, so there must be $\mathsf{in}(\#_i^r(a)) \in A$ and $\mathsf{in}(\#_i^r(a)) \in t(a)$. By the preceding item $\mathsf{in}(\#_i^r(a)) \in A$ forces any $\mathsf{in}(\#_j^{r'}(a)) \in A$ for any j-occurence of a in a rule $r' \in P$, thus for any $t(a) \subseteq A$, a contradiction. □

We now proceed to the proof of the actual theorem.

Proof We want to prove $S = \mathrm{LM}(P^S)$ given $A = \mathrm{LM}(G^A)$ and S defined by (3.21).

1. $\mathrm{LM}(P^S) \subseteq S$: As the least model is an intersection of all models, it is sufficient to show $S \models P^S$. We assume the contrary, i.e., $S \not\models P^S$.

   (a) By the definition of the reduct P^S there must exist a rule r:

   $$a \leftarrow b_1, \ldots, b_n, \mathbf{not}\ c_{n+1}, \ldots, \mathbf{not}\ c_m \in P$$

   and a corresponding rule in the reduct:

   $$a \leftarrow b_1, \ \ldots, \ b_n \in P^S$$

   such that $a \notin S$ but $\{b_1, \ \ldots, \ b_n\} \subseteq S$ and $\{c_{n+1}, \ldots, c_m\} \cap S = \emptyset$.

(b) We identify the corresponding rule $g(r)$ that exists by Proposition 3.3.1:

$$\begin{aligned}\mathsf{in}(\#^r_0(a)) \leftarrow \mathsf{in}(\#^r_1(b_1)), \ldots, \mathsf{in}(\#^r_n(b_n)), \\ \textbf{not } \mathsf{in}(\#^r_{n+1}(c_{n+1})), \ldots, \textbf{not } \mathsf{in}(\#^r_m(c_m)) \in G\end{aligned}$$

where $\mathsf{in}(\#^r_1(b_1)) \in t(b_1), \ldots, \mathsf{in}(\#^r_n(b_n)) \in t(b_n)$ and $\mathsf{in}(\#^r_{n+1}(c_{n+1})) \in t(c_{n+1}), \ldots, \mathsf{in}(\#^r_m(c_m)) \in t(c_m)$.

(c) Now $\{c_{n+1}, \ldots, c_m\} \cap S = \emptyset$ where each atom c_i has an i-occurrence in r. By the definition of S in (3.21) any $c_i \in S$ would require a corresponding $\mathsf{in}(\#^r_i(c_i)) \in A$, hence $\{\mathsf{in}(\#^r_{n+1}(c_{n+1})), \ldots, \mathsf{in}(\#^r_m(c_m))\} \cap A = \emptyset$.

(d) For an atom $b_i \in \{b_1, \ldots, b_n\} \subseteq S$ we have by the definition of S in (3.21) $t(b_i) \cap A \neq \emptyset$, and $t(b_i) \subseteq A$ by Lemma 3.3.5. Hence $t(\{b_1, \ldots, b_n\}) \subseteq A$ and more specifically $\{\mathsf{in}(\#^r_1(b_1)), \ldots, \mathsf{in}(\#^r_n(b_n))\} \subseteq A$.

(e) Since $A = \mathrm{LM}(G^A)$, $\{\mathsf{in}(\#^r_1(b_1)), \ldots, \mathsf{in}(\#^r_n(b_n))\} \subseteq A$, and furthermore $\{\mathsf{in}(\#^r_{n+1}(c_{n+1})), \ldots, \mathsf{in}(\#^r_m(c_m))\} \cap A = \emptyset$ we have the reduct

$$\mathsf{in}(\#^r_0(a)) \leftarrow \mathsf{in}(\#^r_1(b_1)), \ldots, \mathsf{in}(\#^r_n(b_n)) \in G^A$$

which forces $\mathsf{in}(\#^r_0(a)) \in A$ because $A \models G^A$. Thus $a \in S$ by the definition of S in (3.21), a contradiction.

2. $S \subseteq \mathrm{LM}(P^S)$: First for any $l \geq 0$ we define a projection of S which allows us to focus only on the in-atoms in A:

$$\begin{aligned}S_l = \{a \in \mathrm{Hb}(P) \mid a \text{ has an } i\text{-occurrence in } r \in P \text{ and} \\ \mathsf{in}(\#^r_i(a)) \in \mathrm{T}_{G^A} \uparrow l\}.\end{aligned} \tag{3.22}$$

S_l is monotonically increasing as it is based on monotonically increasing $\mathrm{T}_{G^A} \uparrow l$. We prove by induction on the length l of the sequence S_l that $S_l \subseteq \mathrm{LM}(P^S)$.

(a) Base case: $S_0 \subseteq \mathrm{LM}(P^S)$. By definition $S_0 = \emptyset$ and $\emptyset \subseteq \mathrm{LM}(P^S)$.

(b) Inductive hypothesis: $S_l \subseteq \mathrm{LM}(P^S)$.

(c) Induction step: $S_{l+1} \subseteq \mathrm{LM}(P^S)$.

i. We pick $a \in S_{l+1}$, by the definition of S_l (3.22) there must be an i-occurrence for atom a in $r \in P$, so we have $\mathsf{in}(\#^r_i(a)) \in \mathrm{T}_{G^A} \uparrow l + 1$.

ii. Now for $\mathsf{in}(\#^r_i(a)) \in \mathrm{T}_{G^A} \uparrow l + 1$ there must be a ground metarule in the reduct G^A having that atom as its head and by Proposition 3.3.1 the ground metarule can either be a reified representation of a rule in P ($i = 0$) or a synchronization rule ($i \geq 0$).

iii. For the case when the ground metarule is a synchronization rule ($i \geq 0$) which must exist in both G and G^A:

$$\mathsf{in}(\#^r_i(a)) \leftarrow \mathsf{in}(\#^{r'}_j(a))$$

such that $\mathsf{in}(\#^{r'}_j(a)) \in \mathrm{T}_{G^A} \uparrow l$.

iv. We can map $\mathsf{in}(\#^{r'}_j(a))$ to the reified side by the definition of S_l (3.22) obtaining $a \in S_l \subseteq \mathrm{LM}(P^S)$ by inductive hypothesis.

v. For the case where $\mathsf{in}(\#_i^r(a)) \in \mathrm{T}_{G^A} \uparrow l+1$ is corresponding to a head atom of of a rule, which is not a synchronization rule ($i = 0$) there must be

$$\begin{aligned}\mathsf{in}(\#_0^r(a)) \leftarrow\ & \mathsf{in}(\#_1^r(b_1)), \ldots, \mathsf{in}(\#_n^r(b_n)),\\ & \textbf{not}\ \mathsf{in}(\#_{n+1}^r(c_{n+1})), \ldots, \textbf{not}\ \mathsf{in}(\#_m^r(c_m)) \in G\end{aligned}$$

and a corresponding rule in reduct G^A

$$\mathsf{in}(\#_0^r(a)) \leftarrow \mathsf{in}(\#_1^r(b_1)), \ldots, \mathsf{in}(\#_n^r(b_n))$$

and we must have $\{\mathsf{in}(\#_1^r(b_1)), \ldots, \mathsf{in}(\#_n^r(b_n))\} \subseteq \mathrm{T}_{G^A} \uparrow l$ and $\{\mathsf{in}(\#_{n+1}^r(c_{n+1})), \ldots, \mathsf{in}(\#_m^r(c_m))\} \cap A = \emptyset$.

vi. Now we can map the positive body atoms back to unreified side by the definition of S_l (3.22) and by inductive hypothesis

$$s(\{\mathsf{in}(\#_1^r(b_1)), \ldots, \mathsf{in}(\#_n^r(b_n))\}) = \{b_1, \ldots, b_n\} \subseteq S_l \subseteq \mathrm{LM}(P^S).$$

vii. Assuming that $c_j \in S$ for some $n+1 \leq j \leq m$, $\mathsf{in}(\#_j^{r'}(c_j)) \in A$ by the definition of S in (3.21). By Lemma 3.3.5 $\mathsf{in}(\#_j^r(c_j)) \in A$, a contradiction with $\{\mathsf{in}(\#_{n+1}^r(c_{n+1})), \ldots, \mathsf{in}(\#_m^r(c_m))\} \cap A = \emptyset$. Hence

$$s(\{\mathsf{in}(\#_{n+1}^r(c_{n+1})), \ldots, \mathsf{in}(\#_m^r(c_m))\}) \cap S = \{c_{n+1}, \ldots, c_m\} \cap S = \emptyset.$$

viii. Now we find the non-reified rule via h (3.15) such that $h(\mathsf{in}(\#_0^r(a))) = r$:

$$a \leftarrow b_1, \ldots, b_n, \textbf{not}\ c_{n+1}, \ldots, \textbf{not}\ c_m \in P$$

and since $\{c_{n+1}, \ldots, c_m\} \cap S = \emptyset$ we have the corresponding rule $a \leftarrow b_1, \ldots, b_n$ in the reduct P^S.

ix. As$\{b_1, \ldots, b_n\} \subseteq \mathrm{LM}(P^S)$ we can deduce that $a \in \mathrm{LM}(P^S)$.

(d) Since $S = S_l$ for sufficiently large l, it follows that $S \subseteq \mathrm{LM}(P^S)$.

We have shown that our metaevaluator can simulate an answer set solver for normal and propositional programs. The answer sets produced for reified rules by the metaevaluator are compatible with the answer sets produced for the non-reified rules in the sense that the answer sets for the reified rules contain $\mathsf{in}(\cdot)$ atoms corresponding to the non-reified atoms in the answer set for the original rules. There is no one-to-one correspondence because the reified rules may contain several referrals to the same non-reified atoms, i.e., one non-reified atom may be represented by more than one $\mathsf{in}(\cdot)$ atoms on the reified side. Furthermore the reified answer sets contain reified representations of the rules and artifacts produced by the metagrounding rules for which there are no corresponding counterparts on the non-reified side.

Chapter 4

Metagrounder Implementation

Metagrounding is the grounding of the reified answer set programming rules by means of another answer set program, the *metagrounder*. In this chapter we describe an implementation of a metagrounder operating on the reified representation of variabled normal rules. The end result is a version of input rules where all of the variables have been replaced by potential constant values, together forming a variable free program, which can be submitted to a solver such as `clasp`, `smodels` or our metaevaluator in the previous section. Similarly to the metaevaluator in the previous section, input for our metagrounder is in reified format and we want the results to be in same reified format as well. In contrast to the metaevaluator our task is to produce new versions of the reified rules, which requires us to create completely new structures and identifiers and use these recursively as sources for further new structures.

4.1 Metagrounding Process

Given a set of rules and domain information (`rule1.lp` and `data.lp`) we produce ground rules by the process depicted in Figure 4.1 and listed as steps below:

(a) A non-reified set of rules is processed along with the domain information by `aspreify` to produce a reified representation of both.

(b) The resulting set of facts is combined with the metagrounder to form another set of rules.

(c) Standard ASP tools calculate a single answer set of the combined rules resulting into a reified representation of the ground rules.

(d) Finally we *dereify* the results, i.e., we transform the reified ground rules to non-reified rules.

Earlier in Example 3.1.3 we introduced an example rule and its grounding. Now we use the same example to illustrate the metagrounding process.

Example 4.1.1 *Figure 4.2 shows the essentials of reified representations of the rule (node ID 2) and two ground facts (IDs 12 and 16) in Example 3.1.3. In order to produce a ground instance, each of the literals in the body must be associated with a ground atom, which has*

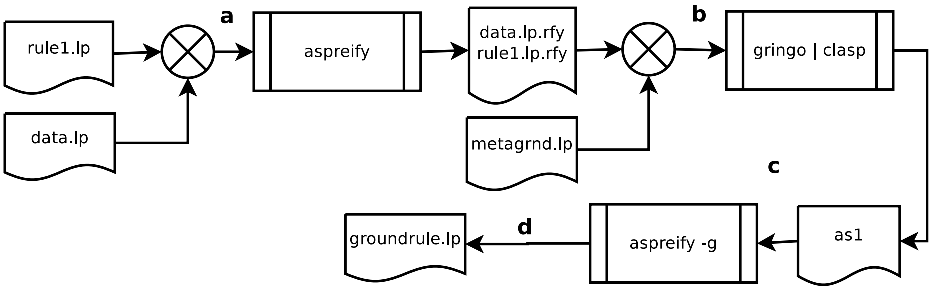

Figure 4.1: Overview of the metagrounding process

*the same name and the same number of terms (*pred, alist*). For rules with just one item in the body, this association is trivial, but when there are several items, the association between the body items and the ground atoms must take into account variables. If the same variable name is in two bodies, then the constants substituting them in the associated ground atoms must be the same.*

For the rule in Figure 4.2, we associate ground fact ID 13 with rule body ID 9 and ground fact ID 17 with body ID 6. The variable Y occurs in both bodies, hence it constrains the second argument of fact ID 13 and the first argument of fact ID 17 to be the same. Any other ground atoms must also honor this constraint.

The purpose of grounding is to produce new instances of rules and facts, in this case reified representations of them. Since the reified rules use unique identifiers for different elements of the rules, we cannot reuse (most of) the identifiers in the rule. Consequently, we need to create new identifiers for a ground instance of the rule. Furthermore the newly produced reified facts should be available as input for the same grounding process, so that they can potentially be used to produce more new facts. In practice this means that we need to generate new IDs for the new facts. As the amount of new facts to be produced cannot easily be determined beforehand, we need to do this dynamically.

Example 4.1.2 *Figure 4.3 shows a reified ground instance of a rule produced by the grounding process in Figure 4.2. The rule, head and body atoms have all new IDs. As in the previous example, the rule body is "trivially" true and we can leave only the head, i.e., the sub-tree under ID 65, which is in the same format as the ground facts. One rule may generate several ground rules with different values for the atoms thereby they will likewise have new unique IDs. For presentation purposes, here the recycled and shared IDs* 14 *and* 19 *are duplicated graphically, even though there is only one instance of them.*

In order to implement the grounding we need to introduce more information to the reified rules making the syntax trees more verbose than these presented in this section. These additional propositions will be gradually introduced below.

4.2 Metagrounder Implementation

As an example of a grounding problem, let us consider the problem of determining the mutual reachability of nodes in a graph as in Example 3.1.3. The full reachability information

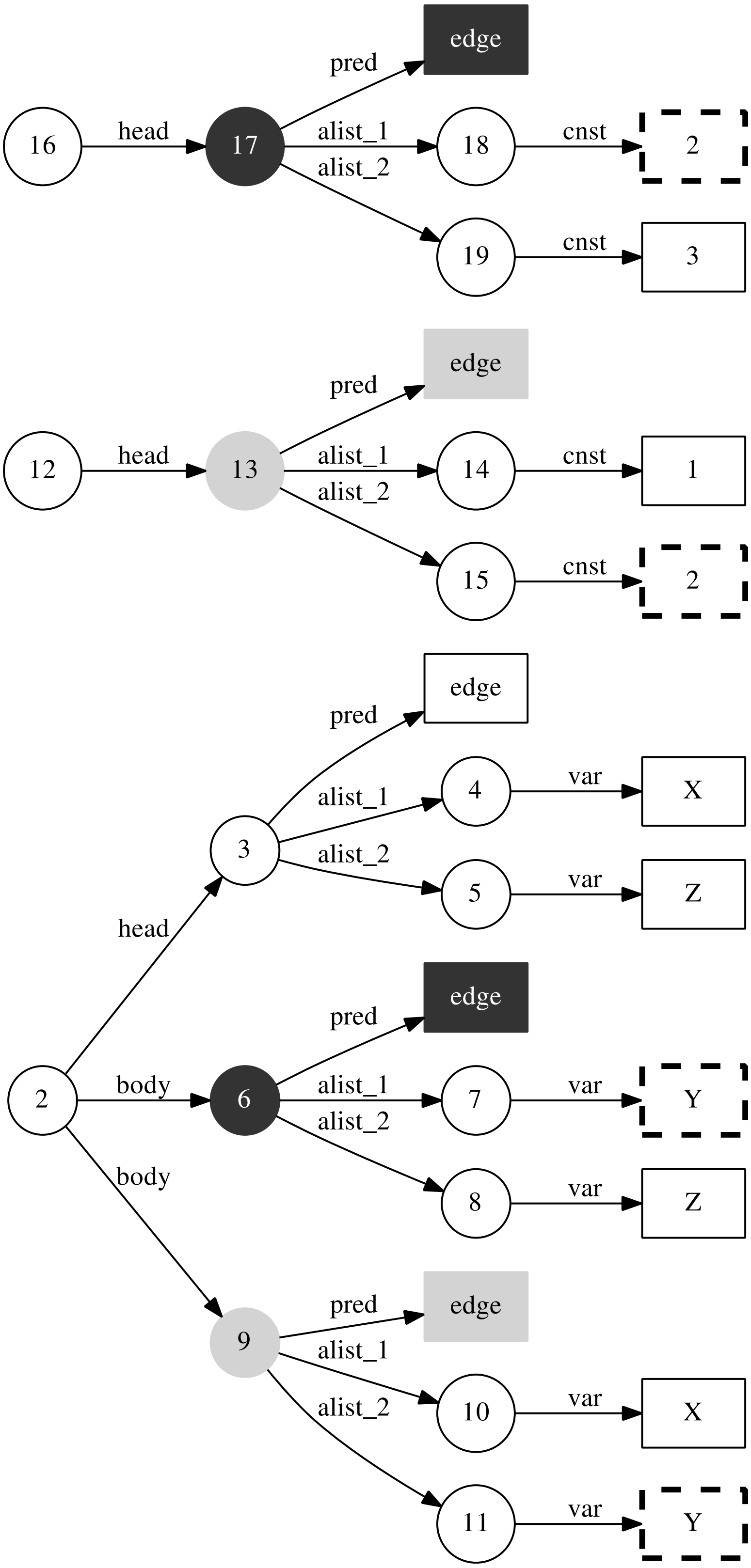

Figure 4.2: Finding suitable reified facts for grounding a reified rule.

from this graph can be produced by grounding only, without the need of the actual solver, however the calculation requires the generation of intermediate facts.

We have two approaches to implement metagrounding: we can either do incremental

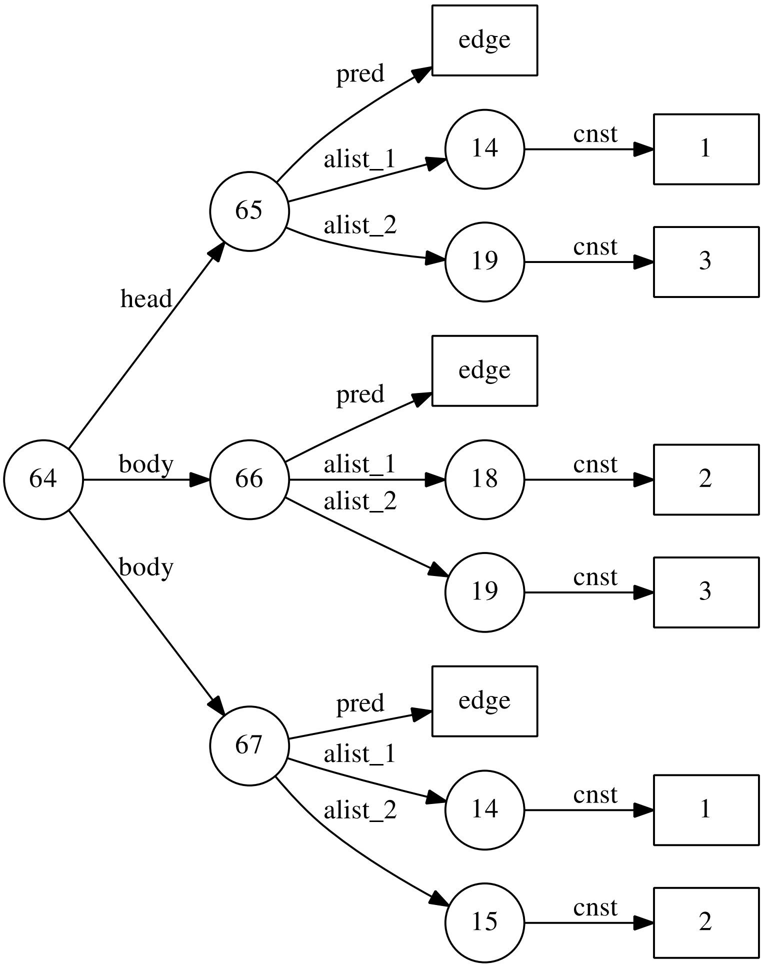

Figure 4.3: A ground instance of a reified rule produced from the reified non-ground rule and facts in Figure 4.2

grounding or we can attempt to perform the whole grounding in one step. In the first case we use existing facts to ground the metarules and produce the immediately following facts, which are combined with the old ones. The steps are re-iterated until no more new facts can be produced. The other possibility is to produce all ground facts representing the ground rules being created in one execution of the rule engine. In the former incremental case it is possible to benefit from the answer set mechanism because it allows to collect results from all resulting answer sets. In the latter single-execution case this is not possible: a fact that has been produced in one answer set is not available to other answer sets. An intermediate step of gathering all of the produced facts would be required. We concentrate here on the latter approach, where we aim to produce the groundings in one execution of the metagrounding rules. However since all the produced ground facts should be available for the same execution, we cannot use mechanisms which produce multiple answer sets. Therefore we need a mechanism to isolate and to name a subset of choices. In this section we present a list-based solution for achieving such a mechanism which comes at the cost of generating all possible sublists of the generated list.

4.3 Implementation of Identifiers as Lists

Consider the need of mapping the potential atoms of a rule body to ground atoms. Ideally, we should produce a set of facts representing the ground facts, their names and the values of the terms, for Example $\mathsf{groundpred(edge}, 1, 3)$ in Figure 4.3. This atom is identified by its arguments and there cannot be two distinct atoms with the same name and arguments. Any rule which creates or refers to atom $\mathsf{groundpred}/3$ operates on a full, coherent representation of the information for that one atom and can make a choice about how to include the values of variables from ground facts to form a new atom $\mathsf{groundpred}/3$. It is not possible for a single instance of $\mathsf{groundpred}/3$ to be incoherent in the sense that either it would be lacking a term, or it would have multiple values for a single term. Representing atoms this way is however particular to atoms of a certain arity. Consequently the metagrounding rules, which should produce these atoms, need to be duplicated for each arity separately. The reificator could produce these rules mechanically, albeit in general the metagrounding rules should be able to handle all arities. Since current ASP languages do not have support for generating a new atom with a dynamically varying number of terms, we must represent the term list separately and combine it with the atom name, while ensuring that this particular combination of values is distinct. Although in [36] the authors present a *higher order atom*, where the first item in a tuple is handled as the predicate name, they do not discuss similar mechanisms for different arities. The following example highlights the task of maintaining a coherent view when the representation does not consist of a single fact. Note that while this example closely resembles the actual implementation, we omit some details in order to focus on the essential problem. We introduce two new auxiliary atoms for the reified representation: $\mathsf{rule}(\cdot)$ and $\mathsf{assert}(\cdot)$, which identify an ID as a rule (with at least one body) or alternatively an ID as a ground fact or assertion. These auxiliary atoms are generated by the reification process.

Example 4.3.1 *The first step of grounding is to relate a potential ground atom to an atom in the body of the rule. Our criterion for candidate facts is that the names of the ground atom and body atom must match. For brevity we do not check for the matching number of arguments. The following rule is the first attempt to fulfill this requirement. Note, that we refer here to atom* $\mathsf{assert}/1$*, which tags an identifier as a ground fact.*

$$\begin{aligned}\mathsf{map}(Ruleid, Bodyid, Groundid) \leftarrow \quad & \\ & \mathsf{rule}(Ruleid),\\ & \mathsf{body}(Ruleid, Bodyid),\\ & \mathsf{pred}(Bodyid, Pn),\\ & \mathsf{assert}(Aid),\\ & \mathsf{head}(Aid, Groundid),\\ & \mathsf{pred}(Groundid, Pn).\end{aligned} \tag{4.1}$$

We look for an atom with ID $Bodyid$ with name Pn in rule $Ruleid$ and then a ground fact or assertion Aid with the same predicate name Pn. For the example in Figure 4.2 the results are:

$\mathsf{map}(2, 6, 13).\quad \mathsf{map}(2, 6, 17).\quad \mathsf{map}(2, 9, 13).\quad \mathsf{map}(2, 9, 17).$

This result is not what we expected. The set contains all of the four possible combinations of ground atoms and their matching atoms in the rule body. The example had two terms in the body so we expect two $\mathsf{map}/3$ *facts for one grounding. We aim to choose a subset from this set such that there would be one* map *for each body ID 6 and 9. Another aim is to be able to identify and refer to these particular subsets in the metarules in order to check the consistency of the mapping. Eventually this would act as a basis for deriving new ground facts.*

Generally we seek to obtain multiple sets of instances of $\mathsf{map}/3$ where the criterion for subsets is that one term in the atom (the second term bodyid) is unique in each subset. Also we require that any individual set of $\mathsf{map}/3$ instances to be complete in the sense that no body ID is without a mapping. As the rules may have an arbitrary (but finite) amount of atoms in their bodies, we cannot restrict the size of these sets. What we require can be achieved using an existential quantifier: *there exists a subset of* r *such that second argument is unique*, see Section 4.3.1 for further discussion. It is easy to create an ASP program which produces answer sets with the required amount of atoms, for instance by using *choice rules*. In this case the identity of the choice is implicit in the answer sets themselves: each generated answer set has its own identity. We base the generation of new ground facts on $\mathsf{map}/3$ instances, so to be sure that all possible derivations for new ground instances can be produced, we need to be able to have access to the newly generated $\mathsf{map}/3$ instances within the same answer set. Yet this is not possible as each answer set is a separate world. Due to this absence such an otherwise idiomatic mechanism is not acceptable for us. Our solution to fill this gap is based on constructing a list structure of the selected subsets. By construction the list representation lends itself to be used as a unique identifier. To this end we use the *function symbols*. Syntactically the function symbol must appear in terms inside the atoms. It cannot exist on the same level as atoms, rules, or facts. The underlying idea in our list representations is using a recursive structure, similar to lists in LISP:

$$\mathsf{lst}(\mathsf{cons}(''a'', \mathsf{cons}(''b'', \mathsf{cons}(''c'', \mathsf{nil})))) \tag{4.2}$$

The function symbol $\mathsf{cons}/2$ has two arguments, the first one is a member of the list and the second one is the rest of the list, which is recursively another $\mathsf{cons}/2$ term or constant nil, indicating an empty list. As function symbols cannot appear independently, the predicate $\mathsf{lst}/1$ is chosen here arbitrarily to host the list representation. Furthermore, we can use any other function symbol name for list representations in a similar manner. In other words, even though the underlying concept is general, we are not presenting a generic, modular list library. We can handle function symbols in a straightforward manner in our rules. Function symbol can be unified with a variable and recursive structures can also be decomposed and unified to variables. Function terms can also be compared for equality, thereby potentially serving as identifiers. The example below demonstrates these features.

Example 4.3.2 *Consider the list in (4.2) along with the following rules and facts* P. *We give another list as a term inside* $\mathsf{lst2}/1$.

$$\begin{aligned}
&\mathsf{lst2}(\mathsf{cons}(''b'', \mathsf{cons}(''c'', \mathsf{nil})))).\\
&\qquad \mathsf{hd}(X, \mathsf{cons}(X,Y)) \leftarrow \mathsf{lst}(\mathsf{cons}(X,Y)).\\
&\quad \mathsf{singleton}(\mathsf{cons}(X,Y)) \leftarrow \mathsf{lst}(\mathsf{cons}(X,Y)),\ Y = \mathsf{nil}.\\
&\qquad\qquad \mathsf{taileq}(X,Y) \leftarrow \mathsf{lst}(\mathsf{cons}(X,Y)),\ \mathsf{lst2}(Z), Y = Z.
\end{aligned}$$

The Hebrand Universe for these rules is infinite:

$$\mathrm{Hu}(P) = \{”a”, ”b”, ”c”, \mathsf{nil}, \mathsf{cons}(”a”, \mathsf{nil}), \mathsf{cons}(”a”, \mathsf{cons}(”a”, \mathsf{nil})), \ldots\}$$

Solving the above rules produces the following new facts:

$$\begin{aligned} \mathsf{hd}(”a”, \mathsf{cons}(”a”, \mathsf{cons}(”b”, \mathsf{cons}(”c”, \mathsf{nil})))). \\ \mathsf{taileq}(\mathsf{cons}(”a”, \mathsf{cons}(”b”, \mathsf{cons}(”c”, \mathsf{nil}))), \mathsf{cons}(”b”, \mathsf{cons}(”c”, \mathsf{nil}))). \end{aligned}$$

For the rule in the second line ($\mathsf{hd}/2$*) we can have the following substitution:*

$$\sigma = [X/”a”, Y/\mathsf{cons}(”b”, \mathsf{cons}(”c”, \mathsf{nil}))]$$

The rule on the third line ($\mathsf{singleton}/1$*) shows that we can access and compare individual elements inside the function symbols. The rule on the fourth line (*$\mathsf{taileq}/1$*) shows how we can compare two recursive symbols.*

We need to provide constructors and accessors for lists so that we can manipulate them by our metarules. We add an explicit index for each list item because this helps us when creating the lists and also enables us to set explicit guards to avoid infinite recursion. For example the list in (4.2) is represented as

$$\mathsf{lst3}(\mathsf{c}(1, ”a”, \mathsf{c}(2, ”b”, \mathsf{c}(3, ”c”, \mathsf{nil})))) \tag{4.3}$$

For this type of lists we want to have accessors in the following style:

$$\begin{aligned} \mathsf{nth}(1, ”a”, \mathsf{c}(”b”, \mathsf{c}(”c”, \mathsf{nil})), \mathsf{c}(”a”, \mathsf{c}(”b”, \mathsf{c}(”c”, \mathsf{nil})))). \\ \mathsf{nth}(2, ”b”, \mathsf{c}(”c”, \mathsf{nil}), \mathsf{c}(”a”, \mathsf{c}(”b”, \mathsf{c}(”c”, \mathsf{nil})))). \\ \mathsf{nth}(3, ”c”, \mathsf{nil}, \mathsf{c}(”a”, \mathsf{c}(”b”, \mathsf{c}(”c”, \mathsf{nil})))). \end{aligned}$$

The first argument of $\mathsf{nth}/4$ is the desired index and the second argument is the value at this index. The next two arguments are needed for bookkeeping: the third argument has the rest of the list after the current index and the fourth argument is the list which we are indexing. The fourth argument is the key which relates all of these predicates to a particular list instance – we use this list as an identifier. The third argument enables traversal of the list by always having direct access to the rest of the list. We can define accessors for this type of lists as follows:

$$\mathsf{nth}(1, E, X, \mathsf{c}(1, E, X)) \;\leftarrow\; \mathsf{lst3}(\mathsf{c}(1, E, X)). \tag{4.4}$$

$$\mathsf{nth}(N + 1, E, X, L) \;\leftarrow\; \mathsf{nth}(N, Olde, \mathsf{c}(N + 1, E, X), L). \tag{4.5}$$

Rule (4.4) is the base case where we peel off the outermost value of the source list and construct the first accessor fact $\mathsf{nth}/4$. We continue the generation of accessors based on the existing $\mathsf{nth}/4$ facts in rule (4.5): we access the third argument which must contain index of the next item ($N + 1$), the list element E and rest of the list X. We produce a new $\mathsf{nth}/4$ accessor using these values as the first, the second and the third arguments and maintain the unchanged full list as the last fourth argument. This process ends when we generate $\mathsf{nth}/4$

where the third argument is nil and we cannot unify the third argument in (4.5). The end result is one new $\mathsf{nth}/4$ fact for each item in the list.

Here we defined the list indexing so that the outermost item of the list corresponds to the first index. If the lists are constructed so that the outermost item corresponds to the last index, we need to define the accessors differently. We reverse the order of our lists, so the list in (4.2) is represented as:

$$\mathsf{lst4}(\mathsf{c}(3, "c", \mathsf{c}(2, "b", \mathsf{c}(1, "a", \mathsf{nil})))) \tag{4.6}$$

The following rules define accessors for the reverse order:

$$\begin{aligned} \mathsf{rev}(N, E, X, \mathsf{lst4}(Val, X)) &\leftarrow \mathsf{lst4}(\mathsf{c}(N, E, X)). && (4.7)\\ \mathsf{rev}(N-1, E, X, L) &\leftarrow \mathsf{rev}(N, Olde, \mathsf{c}(N-1, Val, X), L). && (4.8) \end{aligned}$$

The structure of these rules is similar to the rules in (4.4) and (4.5). The base case is rule (4.7), which constructs the initial accessor based on the outermost item in the nested list in $\mathsf{lst4}/3$. Based on the existing $\mathsf{rev}/4$ facts we produce recursively new $\mathsf{rev}/4$ facts in rule (4.8) until the third argument is nil.

Our list generation mechanism is limited: we require some kind of existing ordering which we copy when constructing our lists. For instance, in order to re-implement our mapping of ground atoms to bodies in a rule (in Example 4.3.1), we need to know how many body items the original rule has. To this end we modify the reification and introduce a new auxiliary atom in the reified rules: $\mathsf{bodylist}/3$. It is otherwise like $\mathsf{body}/2$, but assigns an integer index for each item. For the rule in Figure 4.2 we will have the following new facts:

$$\mathsf{bodylist}(2, 1, 6). \quad \mathsf{bodylist}(2, 2, 9).$$

which are compatible with the existing reified facts:

$$\mathsf{body}(2, 6). \quad \mathsf{body}(2, 9).$$

The idea is similar to $\mathsf{alist}/3$ which represents terms of an atom. We are not interested in the syntactical order for strict evaluation purposes, but rather as a mechanism for having unique and predictable sequence of identifiers within one reified rule. We cannot anticipate that the first body in a rule will have a particular identifier, since this would depend on the number of terms in the head atom which have their own identifiers. Yet with $\mathsf{bodylist}/3$ we can consistently identify the first identifier for the body item and increment it by one to obtain the next one. Accordingly we modify rule (4.1) in the earlier Example 4.3.1 to create a list which has an element for each body item in a particular rule. The base case for our construction starts with the first available $\mathsf{bodylist}/3$.

$$\begin{aligned} \mathsf{maplst}(Ruleid, \mathsf{c}(1, Groundid, \mathsf{nil})) \leftarrow \quad & && (4.9)\\ & \mathsf{rule}(Ruleid),\\ & \mathsf{bodylist}(Ruleid, 1, Bodyid),\\ & \mathsf{pred}(Bodyid, Pn),\\ & \mathsf{assert}(Aid),\\ & \mathsf{head}(Aid, Groundid),\\ & \mathsf{pred}(Groundid, Pn). \end{aligned}$$

Now we can use the existing $\mathsf{maplst}/2$ to construct the rest of the list so that the $\mathsf{maplst}/2$ for the previous index N is combined with $\mathsf{bodylist}/3$ having the index $N+1$:

$$\begin{aligned}
\mathsf{maplst}(&\mathit{Ruleid}, \mathsf{c}(N+1, \mathit{Groundid}, \\
&\quad \mathsf{c}(N, \mathit{Oldid}, \mathit{Rest}))) \leftarrow \\
&\qquad \mathsf{maplst}(\mathit{Ruleid}, \mathsf{c}(N, \mathit{Oldid}, \mathit{Rest})) \\
&\qquad \mathsf{rule}(\mathit{Ruleid}), \\
&\qquad \mathsf{bodylist}(\mathit{Ruleid}, N+1, \mathit{Bodyid}), \\
&\qquad \mathsf{pred}(\mathit{Bodyid}, \mathit{Pn}), \\
&\qquad \mathsf{assert}(\mathit{Aid}), \\
&\qquad \mathsf{head}(\mathit{Aid}, \mathit{Groundid}), \\
&\qquad \mathsf{pred}(\mathit{Groundid}, \mathit{Pn}).
\end{aligned} \tag{4.10}$$

The rules above will generate all possible combinations of the ground atoms and their matching atoms in lists whose length is limited by the number of body items in a rule. The complete lists must be identified with the knowledge that the index equals the largest $\mathsf{bodylist}/3$ index. In addition to the full lists, all sublists within these lists will be generated. While it may be possible to modify the above rules to exclude some combinations of values, it is not possible with this approach to omit generating the intermediate lists. If a rule n there are body atoms with the same atom and k suitable ground atoms (which unify), then we get k^n full lists and $\Sigma_{i=1}^{n} j^i$ lists in total with sublists. For accessing the list items we need to use the $\mathsf{rev}/4$ rules (4.7) and (4.8). Alternatively, we could build the list the other way around, by starting with the highest bodylist index and counting down. In that case the $\mathsf{nth}/4$ rules (4.4) and (4.5) could be used. In both approaches the creation and access of the lists are tied together.

Example 4.3.3 *Using the previous rules we obtain the following lists for Example 4.3.1:*

$$\begin{aligned}
\mathsf{maplst}(2, \mathsf{c}(1, 18, \mathsf{nil})). &\quad \mathsf{maplst}(2, \mathsf{c}(1, 13, \mathsf{nil})). \\
\mathsf{maplst}(2, \mathsf{c}(2, 18, \mathsf{c}(1, 13, \mathsf{nil}))). &\quad \mathsf{maplst}(2, \mathsf{c}(2, 13, \mathsf{c}(1, 13, \mathsf{nil}))). \\
\mathsf{maplst}(2, \mathsf{c}(2, 18, \mathsf{c}(1, 18, \mathsf{nil}))). &\quad \mathsf{maplst}(2, \mathsf{c}(2, 13, \mathsf{c}(1, 18, \mathsf{nil}))).
\end{aligned}$$

If the length of the list is known in advance then it could be possible to create a correctly-sized list in one rule omitting the intermediate sublists. This problem is a different manifestation of the problem of generic representation for facts of different arities, which we discussed in the beginning of this section. Also note that instead of requiring the explicit indexing by bodylist we could infer an order for the used body items, as long as the IDs are integers and the reification is consistent. We can then compare the integers to check which is the next biggest integer and use that instead of the explicit link to the next item that is the current $N+1$. If for example we had two body declarations $\mathsf{body}(10, 25)$ and $\mathsf{body}(10, 35)$ for atoms with IDs 25 and 30 and the reification is guaranteed to generate the IDs as monotonically increasing integers, then we could deduce the order within the rule. This would require auxiliary rules for checking across all potential candidate $\mathsf{bodyitem}/3$ facts for the subsequent items. This cumbersome technique can be avoided with explicitly coding the order. After all, ordering information is readily available at the time of reification. Finally, when the list is complete it is possible to compare all items in the list for consistency. During construction it is possible to compare the previous item with the current one. However in order to compare a potential new item to all previous items, we need to access them, therefore we need to create $\mathtt{nth}/3$-style accessors for the sublists during construction. This step increases the amount of superfluous information so that the end result may be that although some unfit lists are avoided, the amount of accessors created may easily exceed the amount of savings in unfit lists.

To summarize: we have presented a list mechanism which allows us to choose subsets of existing facts and give them unique names, so that they can be referred to later or within other rules. This mechanism comes at a cost, that of generating more auxiliary facts. In the following sections we will be using variations of this mechanism for keeping track of the provenance through the grounding process. Typically the name of the function symbol for the list will differ and there will be more terms representing the "content" of the list. Nevertheless the basic mechanisms presented for construction and access will remain the same. In our method, the construction of lists is always bound by the number of items in the rule body, hence the list instances will always have a finite representation. The major compromise in this solution is that since we do not handle function symbols in the reification, we cannot use our metaevaluator to evaluate itself.

4.3.1 Interpretation of Identifier Generation as Skolemization

We mentioned earlier in this section that the problem of creating new object identifiers can be expressed by using an existential quantification over logical statements. Since answer set semantics does not recognize existential quantification, earlier in this section we presented a mechanism which used function symbols to create a unique list representation. In general the method of removing existential quantifiers from a logic formula by replacing quantified variables with a new function symbol is called *Skolemization*. The introduced function is called by extension a *Skolem function* [83, 105]. We argue that our solution is essentially a Skolemization. In our list representation, the existence of an identifier is represented by a recursive Skolem function. The Skolem function and the original formula may have different models, i.e., they are not equivalent, yet they are equisatisfiable; one formula is satisfiable if and only if the other is satisfiable.

Regrettably introducing a function symbol induces an infinite Herbrand instantiation, for example:

$$\mathsf{newhead}(\mathsf{counter}(X+1)) \leftarrow \mathsf{newhead}(\mathsf{counter}(X))$$

In our approach the function symbol based recursive list representations are by construction finite for finite rules. The generation of lists is limited by the items in the body of the rule. For parametrized conjunctions the generated list may not be limited as such. Nevertheless, the number of elements would be limited by the number of ground facts, which remains finite for finite programs instantiated over finite number of ground facts. Hence the size of any generated list is always limited by a combination of a limited number of ground facts.

4.3.2 Discussion Regarding Alternatives for Lists

There are alternatives to using the list structure and we briefly review some of them here. The first option is to use an embedded scripting language within the toolset. An example of this is the `gringo` grounder which has an embedded Lua[1] interpreter. We can define Lua scripts within the rule source code and call Lua functions within the rules. In this example we define a global counter and some functions to access it:

```
#begin_lua
local idctr = 0
function nxtctr(inc)
  idctr = idctr + inc
  return idctr
end
function getctr()
  return idctr
end
#end_lua.
```

The functions can be called with a specific syntax instead of terms and their results will be expanded in the ground atoms. We can modify the head of the $\mathsf{map}/3$ rule in (4.3.1) to produce a new counter:

$$\mathsf{map}(@\mathsf{nxtctr}(1), \mathit{Ruleid}, \mathit{Bodyid}, \mathit{Groundid}) \leftarrow \ldots$$

The aim in this approach is to produce a new identifier for each produced map. Yet we get the same number for all of the produced new facts. We speculate that this is due to the underlying declarative semantics of ASP. Since the presence of side-effects (updating a global counter) outside of the `gringo` runtime is not fully defined, this would represent an abuse of the extension mechanism. Furthermore moving the call to the Lua code within the body of the rule will produce the same results, i.e., the same counter number is generated for multiple results. Notwithstanding it is possible to trick `gringo` to produce the desired results for this case:

$$2\{\mathsf{map}(@\mathsf{getctr}(), \mathit{Ruleid}, \mathit{Bodyid}, \mathit{Groundid}), \mathsf{nxt}(@\mathsf{nxtctr}(1))\}2 \leftarrow \ldots$$

Here we use the choice mechanism to produce precisely two facts at the same time, albeit no actual choices are made. When the counter incrementation is a separate fact than the fact wherein the counter is read, the evaluation policy produces an execution whereby the modification of the global state has been “committed”. However, this only works when we make a single inference. Once we start to have new counter values which depend on previous counters, we get an unbounded recursion with the system producing an endless stream of new IDs at grounding stage. Hence we abandoned this method of identifier generation. Other options require extensive modification of the current toolset or else an adaptation of other toolsets. As we have a working solution, we did not pursue these further. One possibility to replace the Lua counter would be to modify the `gringo` grounder and solver implementations with a hardcoded counter atom available at the rule level. In addition to

[1] `http://www.lua.org/`

being limited to the modified implementations, this approach introduces a further risk of breaking the semantics by introducing an ad-hoc non-declarative element in the evaluation. Although the `dlv` [46] solver includes list features, our rules are not directly compatible with `dlv` on syntactical level. Eiter et al. [36] describe integration to external solvers that might provide a way of including an external counter in a well defined way. There is existing work that combines ASP with Prolog semantics [38, 91, 95], which may allow creating a counter in Prolog that could be used in a program which otherwise would be subject to answer set semantics. Prolog and ASP are more similar than ASP and Lua. Hence this combination seems more promising when compared to the Lua approach. The objective in all of these approaches would be to keep the counter isolated from the rest of the evaluation by essentially combining two semantics within one evaluation.

4.4 Logical Steps of Metagrounding and their Implementation

We split the process of computing the ground instances to logical steps. Essentially we represent the ground facts in an internal list format so that the list identifies uniquely the ground fact, i.e., there cannot be double instances of the ground fact representations. Given a reified rule we produce internally lists of these lists for ground facts so that they are consistent with the variables in the rules. Then we recombine the relevant parts into a third list. This end result is in the same format as the first internal list for ground facts. The end result may trigger the same process from the beginning. The process rests on three different internal *flows*: one for producing ground facts when all of the atoms in the rule body are groundable, another for producing ground rules when the rule needs to be evaluated by the solver and finally one for handling parametrized conjunction, which expands a ground rule syntactically. We separate the cases for deriving facts and deriving ground rules for clarity, even though they are quite similar. The alternative of producing both ground rules and facts within the same flow would still end up requiring additional bookkeeping information. In fact it may be that this extra scaffolding would lead to higher complexity than what is achieved by a single flow. We aim to be conservative in choosing which flow to use. If we cannot be sure that a rule can be ground as a fact, we will produce a ground rule, which is always correct but may not be optimal as the solver needs handle the resulting rule. The overall approach in each of the three flows is as follows:

1. Initialize the process by creating an internal list-based representation of available ground facts.

2. Create all possible *mappings* between the identifiers representing the atoms in the rule and ground facts with matching names. The result of this is a restriction over Herbrand instances as we limit the possible values to those which are related to the ground atoms. We can also prune ground atom sources by metainformation (e.g., provenance, timing) if such information is associated with the ground atom identifiers. At this point the mapping obtains a unique identifier.

3. We combine these mappings to multiple possible *substitutions* producing pairs of variable names and terms. The intermediate outcome may be inconsistent as the same

variable might be given two different terms. Nevertheless, checking and filtering out inconsistent assignments is straightforward and the end result is a set of (consistent) substitutions. We maintain the link to the identifier created for the mapping which has induced this substitution.

4. Create the final internal representation of the ground rule or fact, which is a list with substitutions and the source atoms for them. At this point, it is also possible to take into account metainformation, together with the terms substituted with the variables.

5. Finally, render the generated internal representation to the reified format and also to the same internal format as presented in the first step, allowing the newly generated facts to be available as input to this process. At this point we give the newly generated facts or rules their identifiers.

Deciding which flow to apply for a particular rule is based on a rudimentary syntax analysis and we tag identifiers of rules which can produce ground atoms with $\mathsf{evaluable}/1$. The flow for grounding atoms is outlined in Figure 4.4. Note that we do not elaborate all of the atoms, but concentrate on the most relevant ones. However, a full listing of the used atoms and their descriptions are given in Appendix B in Tables B.2, B.3, and B.4.

First we define different list types which we are using to keep track of our grounding. The approach is similar to the one presented in previous Section 4.3: lists are recursive structures built using function symbol based constructors and the name of the function symbol reflects the type of the list. For each list type we define relevant accessor terms in same manner as in previous section but we omit details here.

The v-*list* represents the terms of a ground fact. It is of form $\mathsf{v}(I, V, X)$, where I is the index for the list element, V is the list element itself, the ground term, and X is rest of the list as a recursive structure with nil representing the empty list. The outermost list element has the highest index of the list and this index also indicates the length of the list.

Example 4.4.1 *Consider a ground fact* $\mathsf{edge}(2,3)$*, which has a reified representation as shown in Figure 4.2. Our internal representation of the reified fact is*

$$\mathsf{apredlst}(17, \textit{"edge"}, \mathsf{v}(2, "3", \mathsf{v}(1, "2", \mathsf{nil})))$$

where 17 *is the identity of the ground fact given by the reification process. The* v*-list itself needs more information to fully describe a ground fact and its provenance and in here this information is provided by the identity and the name of the fact, which must be associated with the* v*-list.*

The z-*list* represents a substitution, which consists of representations of ground facts (with values as v-lists) which contribute to this particular substitution. This list is of form $\mathsf{z}(I, Vn, C, B, L, X)$, where I is the index for the list element, Vn is name of a variable in a rule, C represents the ground term to be substituted for Vn, B uniquely identifies a body item in a rule (and hence also the rule and the name of the relevant predicate) which contains the variable Vn. The argument L has the v-list representation of the terms of the ground fact which produced substitution for this variable. Finally X is rest of the list as a recursive structure with nil representing the empty list. Again, the outermost element has the highest index of the list and this index indicates the length of the list. In order to construct z-list there must be a mapping of a body item to a ground fact in the same style as in Example 4.3.3. A z-list is *consistent* when there is only one substituted term for each variable.

Example 4.4.2 *A representation of the substitution for the rule and facts in Figure 4.2 is given by the consistent* z*-list below:*

$$
\begin{aligned}
&\mathsf{z}(3, "Z", "3", 6, \mathsf{v}(2, "3", \mathsf{v}(1, "2", \mathsf{nil})),\\
&\quad \mathsf{z}(2, "Y", "2", 6, \mathsf{v}(2, "3", \mathsf{v}(1, "2", \mathsf{nil})),\\
&\qquad \mathsf{z}(1, "X", "1", 9, \mathsf{v}(2, "2", \mathsf{v}(1, "1", \mathsf{nil})), \mathsf{nil})))
\end{aligned}
$$

The intuitive interpretation for this z*-list is that mapping reified body item with identifier* 6 *with a ground fact has produced a substitution for variables* Z *and* Y*. Similarly mapping body item* 9 *has produced a substitution for variable* X*. This list is compatible with the* maplst/2 *facts in Example 4.3.3.*

Based on a consistent z-list (a substitution) we can create two types of lists pertaining to the new ground fact which is being produced. This is done by deriving a v-list for the terms of the new ground fact and another corresponding list keeping the provenance information of the new v-list. The list containing the provenance information is a vv-*list*. The list is of form $\mathsf{vv}(I, S, Vn, X)$, where I is the index for the list element, S is the identifier of the ground term which is used to produce a substitution for variable named *Vn*. Finally X is rest of the list as a recursive structure with nil representing the empty list. A vv-list has at least one item for each variable occurring in the head atom of the rule which is being grounded, but it can also have multiple sources for the same variable name. The v-list is generated to represent the ground fact resulting from grounding of the particular rule, so it is determined by the variables in the head atom of the rule. Specifically, the arity of the v-list is always determined by the head of the rule it is derived from.

Example 4.4.3 *Continuing the previous Example 4.4.2, we obtain the following lists from the* z*-list presented there:*

$$\mathsf{v}(2, "3", \mathsf{v}(1, "1", \mathsf{nil})) \quad \mathsf{vv}(2, 17, "Z", \mathsf{vv}(1, 13, "X", \mathsf{nil}))$$

This corresponds to ground fact $\mathsf{edge}(1, 3)$ *and the* vv*-list identifies the reified source ground facts for both of the terms in the ground fact. Looking at Picture 4.2, we can see that the head atom has two variables* X *and* Z *and the* vv*-list has items which link the variable name to the identifier of the source of the binding. The* v*-list is also based on the head atom and it is in the same format as the* apredlst/3 *in Example 4.4.1. This* v*-list is in same format as a reified ground fact.*

Any newly produced ground fact needs to be translated from the internal format to the reified format and for this reason it needs a new identifier. In case of a ground fact this identifier is the v-list augmented with a counter. This list can always be associated with a provenance list.

When a rule cannot be grounded to a fact, but needs to be grounded as rule, we use an f-list and an ff-list as which are linked in a similar way as a v-list and a vv-list, where the ff-list contains the provenance for the f-list. The f-list contains information for each bodyitem of the ground rule, mapping this identifier to a ground fact. An f-list is consistent when the induced substitution has only one substituted term for each variable. The form of an f-list is as follows: $\mathsf{f}(I, B, V, X)$ where I is the index for the list element, B uniquely identifies a body item in a rule, V stands for the v-list which represents the terms of a ground fact bound to the body item and, like before, X is rest of the list as a recursive structure with nil representing the empty list. The main property which makes f-list applicable to ground rules is that the ground facts which are related to the body items may be heads of ground rules as opposed to standalone ground facts. The form of an ff-list is as follows: $\mathsf{ff}(I, A, X)$, where I is the index for the list element and A is an identifier of a ground fact and X is rest of the list as a recursive structure with nil representing the empty list. The ff-list always refers to elements of an f-list via the index.

Example 4.4.4 *The following* f*-list describes a ground rule as shown in Figure 4.2, combining the (unique) identifiers of the body items in the rules with* v*-list representations of the terms of the ground facts. It is followed by the related* ff*-list giving the provenance for the values in the* f*-list:*

$$\begin{aligned}&\mathsf{f}(2, 6, \mathsf{v}(2, \text{"3"}, \mathsf{v}(1, \text{"2"}, \mathsf{nil})),\\ &\quad \mathsf{f}(1, 9, \mathsf{v}(2, \text{"2"}, \mathsf{v}(1, \text{"1"}, \mathsf{nil}), \mathsf{nil})))\\ &\mathsf{ff}(2, 17, \mathsf{ff}(1, 13, \mathsf{nil}))\end{aligned}$$

The logical flow of the grounding for rules which can be grounded as facts is shown in Figure 4.4 and elaborated below.

1. The process starts by deriving a list-based representation ($\mathsf{apredlst}/3$) for each reified ground atom ($\mathsf{assert}(\mathit{I1})$). Each $\mathsf{apredlst}(\mathit{I1}, \mathit{N1}, \mathit{VL1})$ consists of an identifier $\mathit{I1}$, atom name $\mathit{N1}$, and the arguments for the atom as a v-list.

2. We combine ground atom representations with body items using $\mathsf{lmap}(\mathit{R1}, \mathit{B1}, \mathit{VL1})$, where $\mathit{R1}$ is the identifier for the rule, which has a bodyitem with identifier $\mathit{B1}$. Finally, $\mathit{VL1}$ is a v-list which is associated by apredlst with a ground fact with matching atom name.

3. Based on $\mathsf{lmap}/3$ instances we produce a set of candidates for substitutions, $\mathsf{xassign}/5$, which associate a particular variable name with a constant along with the information detailing the bodyitem of the rule hosting the variable and the associated ground fact terms. The form is as follows: $\mathsf{xassign}(\mathit{R1}, \mathit{B1}, \mathit{Vn1}, \mathit{Val1}, \mathit{VL1})$, where $\mathit{R1}, \mathit{B1}$ and $\mathit{VL1}$ are copied from lmap, $\mathit{Vn1}$ is name of a variable and $\mathit{Val1}$ is the bound ground term. We can further refine $\mathsf{xassign}/5$ by producing $\mathsf{visiblexassign}/5$, which has identical arguments to $\mathsf{xassign}/5$, but it is only produced for variables which only occur in the head of rule $\mathit{R1}$. These candidate substitutions can be checked for consistency so that a variable is only substituted with a single ground term.

4. Combining the suitable candidate substitutions from the previous step, we construct a new full substitution for all variables in a particular rule: $\mathsf{vlist2}/3$. The form is as follows: $\mathsf{vlist2}(\mathit{R1}, \mathit{Pn1}, \mathit{ZL1})$, where $\mathit{R1}$ is identifier for the rule, which has a predicate with head named $\mathit{Pn1}$ and $\mathit{ZL1}$ is a z-list which contains a (consistent) list representing a substitution. This is essentially the provenance information of the newly grounded atom.

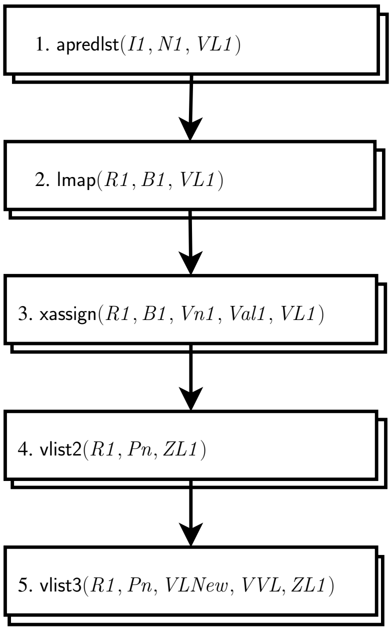
1. apredlst($I1$, $N1$, $VL1$)
2. lmap($R1$, $B1$, $VL1$)
3. xassign($R1$, $B1$, $Vn1$, $Val1$, $VL1$)
4. vlist2($R1$, Pn, $ZL1$)
5. vlist3($R1$, Pn, $VLNew$, VVL, $ZL1$)

Figure 4.4: Logical flow for grounding a rule to a ground fact.

5. The vlist2/3 is then used to construct the final internal representation of the result of the grounding: vlist3/5. The form is as follows: vlist3($R1$, $Pn1$, $VL2$, VVL, $ZL1$), where $R1$ and $Pn1$ are copied from vlist2/3. The first of the following lists, $VL2$, is a v-list and together with $R1$ and $Pn1$ it contains the information for a new apredlst/3 corresponding to a new ground fact. The provenance information for the v-list (and hence the new ground fact) is given by VVL which is a vv-list. Finally, $ZL1$ is a z-list which is copied directly from the source vlist2/3 instance. This associates vlist3/5 with the source vlist2/3.

We trace the steps of the previous process in the following example.

Example 4.4.5 *Consider again Example 4.1.1 where we have a reified rule and two reified facts. The above process produces the following facts (we only show complete lists here):*

1. *The internal representations of the two ground facts are as follows:*

   apredlst(17, *"edge"*, v(2, ”3”, v(1, ”2”, nil))).
   apredlst(13, *"edge"*, v(2, ”2”, v(1, ”1”, nil))).

2. *We combine the body items with the ground facts:*

   lmap(2, 6, v(2, ”3”, v(1, ”2”, nil))). lmap(2, 9, v(2, ”2”, v(1, ”1”, nil))).
   lmap(2, 6, v(2, ”2”, v(1, ”1”, nil))). lmap(2, 9, v(2, ”3”, v(1, ”2”, nil))).

3. *Next we produce candidates for substitutions:*

$$\begin{gathered}
\mathsf{xassign}(2, 6, ''Y'', ''2'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil}))).\\
\mathsf{xassign}(2, 6, ''Z'', ''3'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil}))).\\
\mathsf{xassign}(2, 9, ''X'', ''2'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil}))).\\
\mathsf{xassign}(2, 9, ''Y'', ''3'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil}))).\\
\mathsf{xassign}(2, 6, ''Y'', ''1'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil}))).\\
\mathsf{xassign}(2, 6, ''Z'', ''2'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil}))).\\
\mathsf{xassign}(2, 9, ''X'', ''1'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil}))).\\
\mathsf{xassign}(2, 9, ''Y'', ''2'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil}))).
\end{gathered}$$

4. *Now we combine the earlier disjoint information to produce full and consistent substitutions. In this case we get two otherwise identical ones, but with different source for substitution* $[Y/2]$.

$$\begin{aligned}
&\mathsf{vlist2}(2, ''edge'', \mathsf{z}(3, ''Z'', ''3'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil})),\\
&\quad \mathsf{z}(2, ''Y'', ''2'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil})),\\
&\qquad \mathsf{z}(2, ''X'', ''1'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil})), \mathsf{nil})))).\\
&\mathsf{vlist2}(2, ''edge'', \mathsf{z}(3, ''Z'', ''3'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil})),\\
&\quad \mathsf{z}(2, ''Y'', ''2'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil})),\\
&\qquad \mathsf{z}(2, ''X'', ''1'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil})), \mathsf{nil})))).
\end{aligned}$$

5. *Next we construct corresponding* vlist3/4 *facts, here the resulting* v*-lists inside the facts are identical, the differences are in the* z*-lists, which use different* v*-lists for substituting* Y.

$$\begin{aligned}
&\mathsf{vlist3}(2, ''edge'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''1'', \mathsf{nil})), \mathsf{vv}(2, 17, ''Z'', \mathsf{vv}(1, 13, ''X'', \mathsf{nil})),\\
&\quad \mathsf{z}(3, ''Z'', ''3'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil})),\\
&\qquad \mathsf{z}(2, ''Y'', ''2'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil})),\\
&\qquad\quad \mathsf{z}(2, ''X'', ''1'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil})), \mathsf{nil})))).\\
&\mathsf{vlist3}(2, ''edge'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''1'', \mathsf{nil})), \mathsf{vv}(2, 17, ''Z'', \mathsf{vv}(1, 13, ''X'', \mathsf{nil})),\\
&\quad \mathsf{z}(3, ''Z'', ''3'', \mathsf{v}(2, ''3'', \mathsf{v}(1, ''2'', \mathsf{nil})),\\
&\qquad \mathsf{z}(2, ''Y'', ''2'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil})),\\
&\qquad\quad \mathsf{z}(2, ''X'', ''1'', \mathsf{v}(2, ''2'', \mathsf{v}(1, ''1'', \mathsf{nil})), \mathsf{nil})))).
\end{aligned}$$

When a rule cannot be grounded to an atom (it is not tagged as evaluable/1), we will use a different flow to produce a ground rule. The flow for producing ground rules is outlined in Figure 4.5.

1. We produce apredlst/3 similarly to the previous flow.

2. Likewise, the structure for lmap/3 is similar to the previous flow, but we also allow heads of ground rules to be mapped to the rule body.

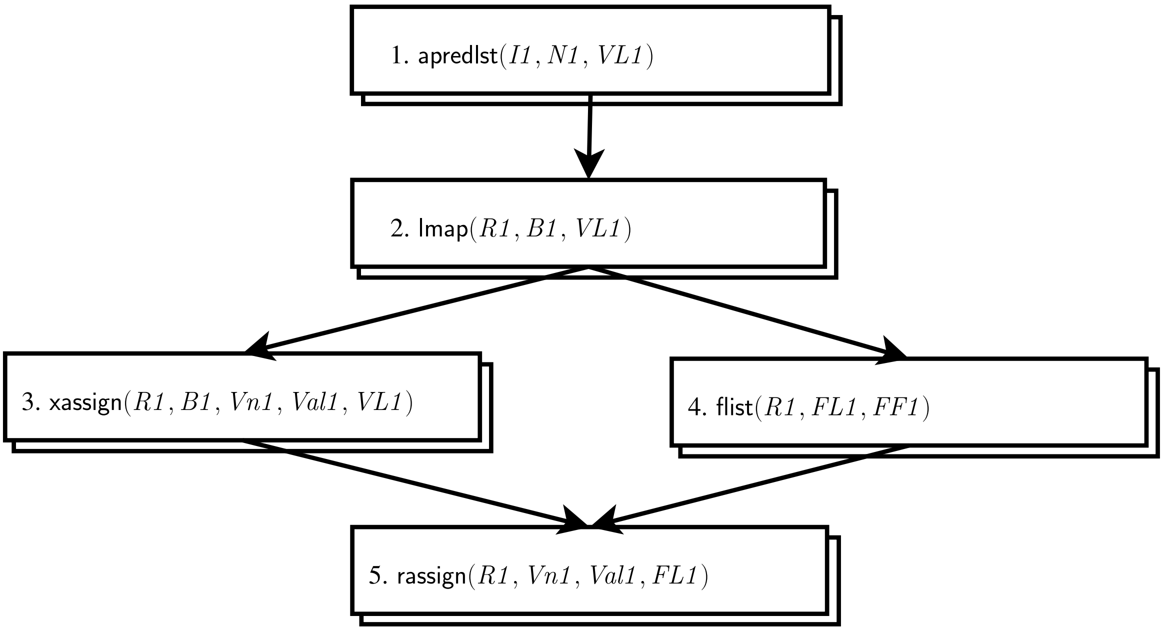
1. apredlst($I1$, $N1$, $VL1$)
2. lmap($R1$, $B1$, $VL1$)
3. xassign($R1$, $B1$, $Vn1$, $Val1$, $VL1$)
4. flist($R1$, $FL1$, $FF1$)
5. rassign($R1$, $Vn1$, $Val1$, $FL1$)

Figure 4.5: Flow for producing a ground rule.

3. We use the same approach and structure for xassign/5 as we did for the previous flow.

4. Independent of xassign/5, we create an f-list to map each bodyitem to a v-list. The structure is as follows: flist($R1$, $FL1$), where $R1$ is identifier for the rule we are grounding and $FL1$ is an f-list having the mapping for all of the body items in the rule.

5. Next we combine flist/2 and xassign/5 to obtain a substitution rassign/4 for all variables in the rule. The structure is as follows: rassign($R1$, $Vn1$, $Val1$, $FL1$), where $R1$ is identifier for the rule we are grounding, $Vn1$ is name of a variable, and $Val1$ is the bound ground term. Finally $FL1$ is the f-list used as basis for this substitution and $FF1$ is the corresponding ff-list for provenance. The f-list is checked for consistency, which also includes checking against numerical comparisons in the rule.

We continue the previous Example 4.4.5 to illustrate the flow for grounding entire rules.

Example 4.4.6 *The rule and facts in Example 4.1.1 can be grounded to facts, but in here we show the steps for grounding the same example to rules. Even though this is superfluous, the result is correct: feeding the produced ground rules to a solver will produce the same ground facts as the earlier grounding process.*

1.-3. These are the same as the corresponding steps in Example 4.1.1.

4. We generate the f*-lists (which may be inconsistent at this point) as well as the related* ff-*lists:*

flist(2, f(2, 9, v(2, ”3”, v(1, ”2”, nil)), f(1, 6, v(2, ”3”, v(1, ”2”, nil)), nil)),
  ff(2, 17, ff(1, 17, nil))).
flist(2, f(2, 9, v(2, ”3”, v(1, ”2”, nil)), f(1, 6, v(2, ”2”, v(1, ”1”, nil)), nil)),
  ff(2, 17, ff(1, 13, nil))).
flist(2, f(2, 9, v(2, ”2”, v(1, ”1”, nil)), f(1, 6, v(2, ”3”, v(1, ”2”, nil)), nil)),
  ff(2, 13, ff(1, 17, nil))).
flist(2, f(2, 9, v(2, ”2”, v(1, ”1”, nil)), f(1, 6, v(2, ”2”, v(1, ”1”, nil)), nil)),
  ff(2, 13, ff(1, 17, nil))).

5. *We combine the previous steps to produce a variable substitution:*

rassign(2, ”X”, ”1”,
  f(2, 9, v(2, ”2”, v(1, ”1”, nil)),
    f(1, 6, v(2, ”3”, v(1, ”2”, nil)), nil))).
rassign(2, ”Y”, ”2”,
  f(2, 9, v(2, ”2”, v(1, ”1”, nil)),
    f(1, 6, v(2, ”3”, v(1, ”2”, nil)), nil))).
rassign(2, ”Z”, ”3”,
  f(2, 9, v(2, ”2”, v(1, ”1”, nil)),
    f(1, 6, v(2, ”3”, v(1, ”2”, nil)), nil))).

Parametrized conjunctions are handled as special cases as they cannot exist independently of a rule, but it must always occur syntactically on the same level as other literals. Unlike basic literals, a parametrized conjunction may nevertheless result to a ground rule with an unpredictable number of ground atoms. A parametrized conjunction may contain atoms with variables which are present in other body items of a rule and the substitutions due to these atoms must be honored. Any other variables in the list of conjunctions can be assigned independently of the surrounding rule. We implement the grounding by using two concurrent flows. The first reuses the previous flow for the rule parts that are outside of parametrized conjunctions. This flow produces the substitutions for any particular rule without parametrized conjunctions. For the second flow we treat parametrized conjunctions within a rule as rules themselves, which have their own substitutions. Intuitively, we are treating the first element of the conjunction list as a head of a rule and the following atoms as body literals of that rule. Any substitution for this “pseudo-rule” is checked for consistency with respect to the substitution produced by the first flow. Eventually both substitutions are combined for the resulting ground rules. We do not describe this flow as it duplicates the previous flows, introducing differently named predicates with the same purpose as in earlier flows. Predicates relevant for grounding parametrized conjunctions are enumerated in Appendix B in Table B.4.

As a convention we provide *accessors* in the style of (4.4) to the lists presented here, notably nthinapredlst/5, nthinvlist2/10, nthinflist/6 and nthintypedlist/6. These are all described in the tables of Appendix B.

4.4.1 Producing New Identifiers

Next we describe the creation of new identifiers based on the results of the flows in the previous section. The generated identifiers are used as a basis for constructing the reified representations of the newly generated ground facts or rules.

The actual generation of the reified representations is mostly plumbing and hence we only give an example of the process. Essentially we generate the new identifier based on either the v-list or the f-list, depending whether we generate new ground facts or new ground rules respectively. Secondly we embed the list to another function symbol id with the first argument being the unique list and the second one an integer, for example $\mathsf{id}(\mathsf{v}(2, ”7”, \mathsf{v}(1, ”9”, \mathsf{nil})), 0)$. The list part forms a namespace for the new reified ground entity and the integer part enables having unique identifiers for reified structures within that namespace. By convention we always start from integer 0 and we generate new integers by incrementing this integer. Note that in our case the ground fact always consists of constants, which are represented in reified form by a $\mathsf{cnst}/2$. It is sufficient to generate a single identifier for such a constant. The following example shows a ground fact with a generated identifier based on the examples in the previous section.

Example 4.4.7 *Continuing from Example 4.4.5 based on the* $\mathsf{vlist3}$ *instances we need to use the* v*-list to produce a new ground fact. In this case we have two* $\mathsf{vlist3}$ *instances with the same* v*-lists and since we use* v*-lists as unique identifiers, we get only one new ground fact. First we create the initial new identifier and use it as the identifier for the root of the syntax tree for the reified fact:*

$$\mathsf{assert}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 0)).$$

Now we can continue building the reified ground fact. We have rules which are triggered by this $\mathsf{assert}/1$ *and they will produce the rest of the syntax tree in a fixed order: first we create a new identifier for the head predicate and then each argument of this predicate gets its own identifier. As these are always paired with the unique list based identifiers, it is not possible to mix them with other s For the reified representation of the ground fact* $\mathsf{edge}(1, 3)$ *we generate the following facts in addition to the above* $\mathsf{assert}/0$*:*

$$\mathsf{head}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 0), \mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 1)).$$
$$\mathsf{pos}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 1)).$$
$$\mathsf{pred}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 1), ”edge”).$$
$$\mathsf{alist}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 1), 1, \mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 2)). \tag{4.11}$$
$$\mathsf{cnst}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 2), ”1”).$$
$$\mathsf{alist}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 1), 2, \mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 3)). \tag{4.12}$$
$$\mathsf{cnst}(\mathsf{id}(\mathsf{v}(2, ”3”, \mathsf{v}(1, ”1”, \mathsf{nil})), 3), ”2”).$$

The identifiers for arguments represented by $\mathsf{alist}/3$ *facts are generated based on the index of the particular argument. In the fact (4.11) the index is* 1 *and the identifier for the new constant term to be created increments the integer part, so here we obtain new integer part* $1 + 1 = 2$ *and for the second fact (4.12) with index* 2 *we get* $1 + 2 = 3$*. In this way we can be sure that the generated identifiers are unique. Here we generate a fact which has only head but no body items. If we would need to generate body items, we would use the arity* *information available in the reified rules to get the next free integer value and generate the predicate in same manner as the head predicate here.*

Prior to this example all of the identifiers were integer numbers. The presented approach for grounding works also when identifiers are nested forms based on id/1 function symbols instead of integers. All of the rules used for grounding we have presented treat identifiers as opaque and unique elements. The inner structure of identifiers is only handled in the rules used for creating new identifiers and even in these rules, existing identifiers are treated as opaque values, irrespective of their possible inner structure.

Use of the provenance information is elaborated in the next section, but the basic idea is that for the internal identifier of a single ground fact (an id/2 term containing a v-list or f-list) or ground rule, we may have several different related provenances as shown in the above examples.

4.5 Termination of Metagrounding

One of our original goals was to use a safe input language so that it should not be possible to incapacitate a node in a distributed system by submitting a never-ending computation for evaluation. We limited our source language to finite normal programs without function symbols as for any such program we can guarantee that groundings are finite and hence produced in finite number of steps. Especially, the source language does not allow function symbols, because introducing them leads to infinite Herbrand universes, and hence infinite groundings of the source program. However, the list mechanism we use in our metagrounder in Section 4.4 is based on function symbols, so we must consider the termination of metagrounding. The reification of the input program is finite for a finite input program, so we need to only consider the list handling in our metagrounder rules.

The lengths of the constructed lists are bounded by the arity of the atoms, number of the variables in the rule and the number of the body items in the rules that we have reified. Each reified fact is represented by an apredlst/3 (see Example 4.4.1) containing a v-list which has one list item for each argument of the fact. For each reified rule the v-lists are mapped to identifiers of the body items in that rule (as shown in (4.1) and (4.9)) producing a finite number of possible mappings. These mappings are used to construct potential variable substitutions, which are then collected to (consistent) z-lists (Example 4.4.2), which are essentially ground substitutions. The number of potential bindings and the size of z-lists are limited by the number of variables and number of mappings. The individual values we use for the substitution are reified constants and there is a one-to-one mapping between these reified constants and constants in the Herbrand universe of the unreified program. The z-lists are transformed directly to v-lists and vv-lists by essentially a projection and a format change, no new combinations are produced in this step. The resulting v-lists combined with head predicate name from the rule represent a potential newly produced ground fact. Its arguments are selected from a finite combination of arguments of existing ground facts. This list is compared with the existing ground facts and only if no similar ground fact (same predicate name and same arguments) exist, we render it as a new ground fact with a new identifier. The flow for grounding rules is similar: the f-list (Example 4.4.4) contains mapping of v-lists to identifiers of body items and related ff-list contains the provenance information. The variable substitution is produced based on the f-list. Each reified ground fact or ground rule produced this way is thus justified by a combination of reified ground facts used as a variable substitution for a reified rule.

We have restricted our input language to finite normal programs without function symbols and we can always produce a finite groundings for such programs. By Definition 3.1.13 ground program is produced by applying substitutions over Herbrand universe to the rules. As this process yields finite groundings for the unreified rules, we argue that similar applying of substitutions on the reified side exhibits same behavior, i.e., a finite grounding.

There has been previous work on embedding functions in ASP programs. Especially Calimeri et al. [23] propose the class of *finitely ground* subset of answer set programs, which allow using possibly recursive function terms in the program. The authors present *intelligent instantiation* for program P. The intelligent instantiation is a finite and computable subset of grounding for program P with the same answer sets as program P. Hence the answer sets of the finitely ground programs are computable. However deciding whether a given program is finitely ground is not decidable. There are various proposals for subsets of finitely ground programs which can be detected: *finite domain* programs [23], ω*-restricted* programs [108] λ*-restricted* programs [48] and *argument restricted* programs [73]. Furthermore Baselice and Bonatti introduce a decidable subclass of *finitary programs* [8], which aims to enable predicates with infinite extensions, such as list manipulation predicates. These approaches count on syntactically analyzing the program, which was left outside the scope of this thesis.

Another view to the termination of metagrounding is the observation we made in the beginning of Section 4.4: we can create for any given input program a metagrounder, which is a finite normal program without function symbols. This *fixed metagrounder* operates in the same way as the current metagrounder, but loses the flexibility to evaluate general reified programs. The fixed metagrounding approach requires definition of distinct atoms to represent the different arities in the unreified rules. In practice, the $\mathsf{apredlst}/3$ facts would be replaced by different versions for each arity. The intuition is that the list based approach is a shorthand for the fixed approach. This requires a larger amount of code due to duplication of the handling rules for each arity and coordination of the rules for different arities. Furthermore, one would lose the possibility of grounding rules with atoms of arbitrary arity.

Example 4.5.1 *Recall Example 4.4.1, where the ground fact* $\mathsf{edge}(2,3)$ *was represented as:*

$$\mathsf{apredlst}(17, \textit{"edge"}, \mathsf{v}(2, ”3”, \mathsf{v}(1, ”2”, \mathsf{nil})))$$

For the fixed metagrounder approach we can represent the same fact by explicitly encoding the arity in the predicate name:

$$\mathsf{apredlst2}(17, \textit{"edge"}, ”2”, ”3”)$$

The accessor rules for the v*-lists in the metagrounder need to be replaced accordingly. Furthermore, all of the rules for* vv, f *and* z*-lists must also be modified to operate on fixed aritites.*

Using this approach the resulting metagrounder is tailored to the particular source rules, whereas our earlier metagrounder was generic.

Chapter 5

Applications of Metagrounding and Metaevaluation

In the previous sections we introduced the reified presentation of the rules, the interpretation of the reified rules (metainterpretation) and maintenance of the identifiers of the information used to derive a particular piece of information during metainterpretation. In this section we use the mechanisms presented in the previous section to propose solutions which fulfill our original goals. Specifically we present a mechanism to implement provenance management which allows introducing policies for information use without modifying the actual rules. We show that a similar approach can also be used for introducing timing constraints and policies for rule evaluation. The reified representation allows plain syntactic analysis as well as program analysis for interoperability checking, which are both introduced in this section. We also present a mechanism for using the provenance information for securing the reasoning results. Finally we report some performance measurements.

Our approach in this section relies on attaching information to the reified rules, specifically to nodes with identifiers. The metagrounders we described earlier can then be augmented to take the attached information into account. The augmented metagrounders serve to filter the existing facts and rules based on the tags associated with them by blocking these facts and rules from entering the grounding pipeline. The underlying mechanism involves first tracking the identifiers of ground facts and newly generated ground facts, then defining hook points within the grounding flows, which allow rules to be constructed, thereby accessing the ID bound metadata. We start this section by explaining the provenance mechanism which is then used to link to quantitative values such as timing information. Finally, we describe an orthogonal, yet important mechanism for applying operations to rules on a purely syntactical level. This interface allows writing metarules for checking the names of predicates in the rules by controlling what predicates are produced and used by untrusted rules.

5.1 Provenance

Provenance for a piece of information refers to metainformation which specifies the origin of that piece of information. In the case of inferred information, provenance refers to the provenances of the information we used for inferring that piece of information. One solution to maintaining provenance in a rule programming context is to modify the existing facts explicitly by adding an extra argument to atoms and requiring a convention that, for example, the first argument always contains the provenance information. With this approach two propositions with the same "payload" information but different provenances are different and hence their combination must take this difference into account in the results, as illustrated in the following example:

Example 5.1.1 *Consider combining two different data items, while maintaining the provenance fully in the first term of the resulting fact:*

$$\begin{aligned}\mathsf{combine}(\mathsf{p}(P1, P2), X, Y) \leftarrow & \\ & X \,!= Y,\\ & \mathsf{data}(P1, X),\\ & \mathsf{data}(P2, Y).\end{aligned} \tag{5.1}$$

We have data, which is tagged by its provenance in the first term:

$$\mathsf{data}(''a'', 10). \quad \mathsf{data}(''b'', 10). \quad \mathsf{data}(''x'', 20). \quad \mathsf{data}(''a'', 20).$$

With the following results:

$$\begin{array}{ll}\mathsf{combine}(\mathsf{p}(''a'', ''b''), 20, 10). & \mathsf{combine}(\mathsf{p}(''a'', ''a''), 20, 10).\\ \mathsf{combine}(\mathsf{p}(''x'', ''b''), 20, 10). & \mathsf{combine}(\mathsf{p}(''x'', ''a''), 20, 10).\\ \mathsf{combine}(\mathsf{p}(''b'', ''a''), 10, 20). & \mathsf{combine}(\mathsf{p}(''b'', ''x''), 10, 20).\\ \mathsf{combine}(\mathsf{p}(''a'', ''a''), 10, 20). & \mathsf{combine}(\mathsf{p}(''a'', ''x''), 10, 20).\end{array}$$

The actual payload data results are the pairs $(10, 20)$ *and* $(20, 10)$ *and in this case they are duplicated while differing only in the source of the data.*

Even though the payload data has the same values, treating them as distinct pieces of information is still intuitively justified to maintain the ownership information. The cost for this is that we need to rewrite the rules and embed possible auxiliary rules for reasoning about the provenance within the rules themselves. We also need to decide generally on how we represent the newly derived combined provenance information in the first argument. This is quite a similar problem to the general representation of a tuple, discussed earlier in Section 4.3: we want to represent all provenance information within one term in a generic way. The list-based solution can be used as a mechanism to carry the combined provenance, regardless of whether it is the simple *lineage model* [28] or the most general *semiring approach* [52]. The lineage provenance associates each result with a set of identifiers contributing to that particular result. Provenance semirings are polynomials with integer coefficients, where the allowed algebraic operators are multiplication and addition. Each result is associated with a polynomial which expresses how the source information was combined to produce the result, basically encoding the evaluation of the rules with particular information sources. The model we present below is similar to the lineage model, except that instead of a single set we maintain multiple lists. Furthermore as we keep the information for mapping between the atoms in rule body and ground facts (as mapping between identifiers), we can use that information to get a more precise understanding of how the result was achieved, as required by the semiring approach. Of course this understanding assumes that the rules are available for consultation. For the reified representation, we do not explicitly need to change the rules or facts, as we can associate provenance information to the reified identifiers and keep the newly produced results associated with their identifiers. In this way we separate the handling and representation of the provenance information from the payload data.

We have two kinds of provenance for the reified representation: the first one is the internal mechanism of identifiers in a syntax tree, the second one is the user level association of the internal identifiers with a higher level description of the source of the information. In this section we show how to associate the first kind of provenance information to the ground results with small changes to the earlier rules. After this association, linking any kind of metainformation (e.g., timing or location) to the source information becomes straightforward and we can decouple the rules operating on the metainformation from the rest of the metagrounding machinery. In this sense handling of the provenance and timing information are done by the same mechanism. We separate the provenance information late in the flows: when we derive the final internal representation of the newly produced ground fact or rule ($\mathsf{apredlst}/3$ or $\mathsf{flist}/3$), we also derive the provenance information $\mathsf{subprovenance}/3$ which also refers to the new identifier. The predicate $\mathsf{subprovenance}/3$ binds two pieces of information to the ID: the identity of the rule which produced this ID and the combination of IDs or provenances, which were used to derive this piece of information. There may be more than one different provenance combinations for the same piece of derived information, hence there may be more than one $\mathsf{subprovenance}/3$ facts for a single new identifier. The next example shows the result of metagrounding approach for the 5.1 example. Our purpose here is to show the nonreified results for easier comparison with the previous example.

Example 5.1.2 *In this example we replicate the previous Example 5.1.1, applying the metagrounding approach, without explicit provenance information. The* combine *rule is straightforward:*

$$\begin{aligned}\mathsf{combine}(X, Y) \leftarrow&\\ &X \mathrel{!}= Y,\\ &\mathsf{data}(X),\\ &\mathsf{data}(Y).\end{aligned} \tag{5.2}$$

For the $\mathsf{data}/1$ *facts we drop the first argument and we also reify them. The underlying assumption is that the facts themselves are combined from multiple sources and this is reflected in the reification: instead of two* $\mathsf{data}/1$ *facts, we have four reified versions with distinct identifiers. This corresponds with the aim of combining information from many different sources and owners. For space reasons we omit the reified representation, but in order to understand the provenance information and its use, we make the reified identifiers available in the comments of the unreified versions. For this example the provenance information equals the identifiers. The identifiers are given here in comments (lines beginning with %) preceding the actual fact or rule. The four* $\mathsf{data}/1$ *facts and their internal identifiers* *are below. The identifiers are unique and assigned by the reification process.*

$\%ID$: ”14”
$\mathsf{data}(10).$
$\%ID$: ”18”
$\mathsf{data}(10).$
$\%ID$: ”22”
$\mathsf{data}(20).$
$\%ID$: ”26”
$\mathsf{data}(20).$

(5.3)

The results of metagrounding, the reified facts and rules are given below. We present the grounded rules rather than the resulting facts, because this allows an easier illustration of the use of provenance information.

Like in the case of the $\mathsf{data}/1$ *facts above we show the identifiers in comments. Rather than a single number, these identifiers are generated according to the scheme described earlier in Section 4.4.1. The identifier* id *has an* f*-list and an integer, which is always zero for the root of the newly generated reified representation of the ground rule. Integers larger than zero are used for other reified structures within the newly produced rule. The identifier is followed by one or more associated* ff*-lists giving the provenance information. As described in Section 4.4 an* f*-list presents a mapping between a bodyitem of a rule and a ground value. A* v*-list represents the values of a ground fact and an* ff*-list contains the identifiers of ground facts which are used to produce this grounding. We provide the dereified results of grounding below and continue to elaborate on the provenance information thereafter.*

$\%ID$: ”$\mathsf{id}(\mathsf{f}(3, 6, \mathsf{v}(1,$ ”20”$, \mathsf{nil}), \mathsf{f}(2, 8, \mathsf{v}(1,$ ”10”$, \mathsf{nil}), \mathsf{f}(1, 10, \mathsf{nil}, \mathsf{nil}))), 0)$”
$\%\mathsf{ff}(3, 22, \mathsf{ff}(2, 18, \mathsf{ff}(1, 0, \mathsf{nil})))$
$\%\mathsf{ff}(3, 26, \mathsf{ff}(2, 18, \mathsf{ff}(1, 0, \mathsf{nil})))$
$\%\mathsf{ff}(3, 22, \mathsf{ff}(2, 14, \mathsf{ff}(1, 0, \mathsf{nil})))$
$\%\mathsf{ff}(3, 26, \mathsf{ff}(2, 14, \mathsf{ff}(1, 0, \mathsf{nil})))$

$\mathsf{combine}(10, 20) \leftarrow$
$\qquad\qquad \mathsf{data}(10),$
$\qquad\qquad \mathsf{data}(20).$

$\%ID$: ”$\mathsf{id}(\mathsf{f}(3, 6, \mathsf{v}(1,$ ”10”$, \mathsf{nil}), \mathsf{f}(2, 8, \mathsf{v}(1,$ ”20”$, \mathsf{nil}), \mathsf{f}(1, 10, \mathsf{nil}, \mathsf{nil}))), 0)$”
$\%\mathsf{ff}(3, 14, \mathsf{ff}(2, 26, \mathsf{ff}(1, 0, \mathsf{nil})))$
$\%\mathsf{ff}(3, 18, \mathsf{ff}(2, 26, \mathsf{ff}(1, 0, \mathsf{nil})))$
$\%\mathsf{ff}(3, 14, \mathsf{ff}(2, 22, \mathsf{ff}(1, 0, \mathsf{nil})))$
$\%\mathsf{ff}(3, 18, \mathsf{ff}(2, 22, \mathsf{ff}(1, 0, \mathsf{nil})))$

$$\begin{aligned} &\mathsf{combine}(20, 10) \leftarrow \\ &\qquad\qquad \mathsf{data}(20), \\ &\qquad\qquad \mathsf{data}(10). \end{aligned}$$

We walk through the latter identifier, which is an f*-list as the first argument of* id *fact:*

$$\mathsf{id}(\mathsf{f}(3, 6, \mathsf{v}(1, ”10”, \mathsf{nil}), \mathsf{f}(2, 8, \mathsf{v}(1, ”20”, \mathsf{nil}), \mathsf{f}(1, 10, \mathsf{nil}, \mathsf{nil}))), 0)$$

Two of the three bodyitems (with identifiers 6 and 8, with predicate name data*) of the rule have been associated with values from a ground fact. The third bodyitem, the comparison between the variables, is not associated with any fact and the variable substitution is caused by the two previous associations. In the identifier above our only concern are the values, yet the source of the values is explained in the provenance. Consider the last piece of provenance information:*

$$\mathsf{ff}(3, 18, \mathsf{ff}(2, 22, \mathsf{ff}(1, 0, \mathsf{nil})))$$

We have a list of three entries, which is in the same order as the f*-list in the identifier. The list item is the identifier of the reified ground fact which produces the actual value. The third item in the provenance list is zero for the same reason as in the identifier: there is no ground fact which can be associated with the arithmetic comparison. It is easy to see that we have four possible combinations of the ground facts which produce the same newly ground rule with the same bindings. This is compatible with the results of the previous Example 5.1.1. On the reified side the association between the identifier and the* ff*-lists for provenance are given by the* $\mathsf{subprovenance}/3$ *facts.*

Since we are able to track the flow of the identifiers for both the rules and facts, we can assign them with arbitrary user specified *tags*. We can reason about those tags with separate rules, as long as we have a well-defined interface towards the grounding rules. The underlying assumption is that this kind of provenance information and the rules are added to the rules at the time of reification by a separate mechanism linked to the certified provenance of these artifacts.

Example 5.1.3 *We can now track both the source of rules and the facts by tagging them: we use* $\mathsf{prov}/2$ *to associate the identifier with a string tag representing a geographic region. Continuing the previous example we modify the reified representations to associate information to the identifiers of the ground facts:*

$$\mathsf{prov}(14, ”\mathsf{EUR}”).\ \mathsf{prov}(18, ”\mathsf{USA}”).\ \mathsf{prov}(22, ”\mathsf{USA}”).\ \mathsf{prov}(14, ”\mathsf{CHN}”).$$

These should be added to the reified representation of the original data as metainformation at the time of reification. To use this information we may tag a rule with a whitelist *that specifies the regions to what that particular rule can apply to. So the rule can be attached with a policy of what kind of data is allowed to be used. For example, we may state that the rule with identifier 2 should only handle data from China or Europe.*

$$\mathsf{whitelist}(2, ”\mathsf{EUR}”).\ \mathsf{whitelist}(2, ”\mathsf{CHN}”).$$

The results of the reasoning are the same as before but the provenances are different, now there is only one way of combining the data for each of the two results as they are the only combinations admitted by Chinese and European data.

$\%ID$"$\mathsf{id}(\mathsf{f}(3, 6, \mathsf{v}(1, "20", \mathsf{nil}), \mathsf{f}(2, 8, \mathsf{v}(1, "10", \mathsf{nil}), \mathsf{f}(1, 10, \mathsf{nil}, \mathsf{nil}))), 0)$"
$\%\mathsf{ff}(3, 26, \mathsf{ff}(2, 14, \mathsf{ff}(1, 0, \mathsf{nil})))$
$\ldots$
$\%ID$"$\mathsf{id}(\mathsf{f}(3, 6, \mathsf{v}(1, "10", \mathsf{nil}), \mathsf{f}(2, 8, \mathsf{v}(1, "20", \mathsf{nil}), \mathsf{f}(1, 10, \mathsf{nil}, \mathsf{nil}))), 0)$"
$\%\mathsf{ff}(3, 14, \mathsf{ff}(2, 26, \mathsf{ff}(1, 0, \mathsf{nil})))$

We describe rules to implement changes to metagrounding below. Note that in addition to the existing accessor rules for lists, we also introduce a new accessor for ff-lists: nthinffprov/4. The actual payload of the ff-list is the identifier of the provenance, which is the second argument of nthinffprov/4. First we define internal provenance representation isubprovenance/3, which is to be checked before any new fact can be used to produce results. The result is that the first argument, the new identifier, is associated with the identifier of the rule which produced this new grounded fact in the second argument. Furthermore as third argument we associate the new identifier with the f-list describing the mapping between bodyitems and ground facts (shown in Example 4.4.4) which essentially describes the provenance information involved.

$$\begin{aligned}\mathsf{isubprovenance}(\mathsf{id}(Id, 0), Rid, Aid) \leftarrow & \\ & \textbf{not } \mathsf{cflctrassign}(Rid, Id), \\ & \mathsf{flist}(Rid, Id, Aid), \\ & \mathsf{rassign}(Rid, Vn, Val1, TmpAid, Id).\end{aligned} \tag{5.4}$$

The following metarules tag the provenance lists according to whether they conform to the whitelists. We check for admittance and denial separately. The rules pertain to lists longer than one item, we omit the definitions for the singular list, which are similar.

$$\begin{aligned} \mathsf{admitprov}(Id, Rid, P) \leftarrow & \\ & \mathsf{isubprovenance}(Id, Rid, P), \\ & \mathsf{nthinffprov}(N1, C1, R1, P), \\ & \mathsf{prov}(C1, P1), \\ & \mathsf{whitelist}(Rid, P1), \\ & \mathsf{nthinffprov}(N2, C2, R2, P), \\ & \mathsf{prov}(C2, P2), \\ & \mathsf{whitelist}(Rid, P2), \\ & N1\ ! = N2. \end{aligned} \tag{5.5}$$

$$\begin{aligned} \mathsf{denyprov}(Id, Rid, P) \leftarrow & \\ & \mathsf{isubprovenance}(Id, Rid, P), \\ & \mathsf{nthinffprov}(N1, C1, R1, P), \\ & \mathsf{prov}(C1, P1), \\ & \mathbf{not}\ \mathsf{whitelist}(Rid, P1). \end{aligned} \tag{5.6}$$

$$\begin{aligned} \mathsf{allow}(Id) \leftarrow & \\ & \mathsf{admitprov}(Id, Rid, P1), \\ & \mathbf{not}\ \mathsf{denyprov}(Id, Rid, P1). \end{aligned} \tag{5.7}$$

If the provenances of a newly produced fact have been whitelisted, we tag the new identifier with admitprov/3 and if a fact identifier in the provenance list has not been whitelisted, we tag the identifier with denyprov/3. These implement a strict policy where each fact must explicitly be whitelisted in order to be used in grounding, otherwise the identifier is not tagged with allow/1. Other policies can easily be constructed. For example, we could allow everything except for cases which are explicitly blacklisted. We only produce subprovenance if we have an allowance for the particular provenance:

$$\begin{aligned} \mathsf{subprovenance}(Aid, Rid, P) \leftarrow & \\ & \mathsf{isubprovenance}(Aid, Rid, P), \\ & \mathsf{allow}(Aid). \end{aligned} \tag{5.8}$$

Finally, we use the predicate allow/1 to guard the generation of new rules. In contrast to the results without any provenance, we get three provenance lists per new fact. We can refine the same approach by assigning a whitelist to the facts themselves. This means that the source of the fact can explicitly state with what kind of facts it can be combined. For instance we add another whitelist for rule 2 making all three provenances eligible for the rule. Secondly we add a whitelist to data 18 stating that it can only be connected with other data with USA provenance.

$\mathsf{whitelist}(2, \text{"USA"}).\ \mathsf{whitelist}(18, \text{"USA"}).$

We have the following metarules for admitting and denying, where we check that any other fact in the provenance list has a provenance which matches (or does not match) the data specific whitelist. We modify the definition of $\mathsf{allow}/1$ correspondingly and define admission and denial for individual data items.

$$\begin{aligned}\mathsf{admitdata}(Id, Rid, P) \leftarrow \quad & \\ & \mathsf{isubprovenance}(Id, Rid, P),\\ & \mathsf{nthinffprov}(N1, C1, R1, P),\\ & \mathsf{prov}(C1, P1),\\ & \mathsf{whitelist}(C1, P2),\\ & \mathsf{nthinffprov}(N2, C2, R2, P),\\ & \mathsf{prov}(C2, P2),\\ & N1\ ! = N2.\end{aligned} \tag{5.9}$$

The rule defining $\mathsf{admitdata}/3$ is otherwise quite similar to that of $\mathsf{admitprov}/3$ in (5.5), but here we require that if one identifier (of a ground fact) has been tagged with a whitelisted provenance, then any other identifier in the same provenance list must have the same provenance.

$$\begin{aligned}\mathsf{denydata}(Id, Rid, P) \leftarrow \quad & \\ & \mathsf{isubprovenance}(Id, Rid, P),\\ & \mathsf{nthinffprov}(N1, C1, R1, P),\\ & \mathsf{prov}(C1, P1),\\ & \mathsf{whitelist}(C1, P1),\\ & \mathsf{nthinffprov}(N2, C2, R2, P),,\\ & \mathsf{prov}(C2, P2),\\ & \mathbf{not}\ \mathsf{whitelist}(C1, P2).\end{aligned} \tag{5.10}$$

$$\begin{aligned}\mathsf{allow}(Id) \leftarrow \quad & \\ & \mathsf{admitprov}(Id, Rid, P1),\\ & \mathbf{not}\ \mathsf{denyprov}(Id, Rid, P1),\\ & \mathsf{admitdata}(Id, Rid, P2),\\ & \mathbf{not}\ \mathsf{denydata}(Id, Rid, P2)\\ & P1 == P2.\end{aligned} \tag{5.11}$$

For $\mathsf{denydata}/3$ we follow similar approach and we modify the definition of $\mathsf{allow}/1$ to take into account the new admission criteria. With this approach, no user-provided definition can stop derivation of a fact or ground rule based on facts of other users, yet it can still stop that particular fact to be included in the ground rules. Note that here we implicitly assume that each rule has a whitelist and ignore other rules. In the next section we show mechanisms for handling cases where some rules or facts may or may not have tags. Note also that in principle we can tag any identifiers, even the individual body literal identifiers within rules, so that we can have metarules over them.

We can continue in similar manner for even more fine grained access control. For example we could provide individual facts with a list of allowed provenances. This would allow the user to define rules for particular facts as they see fit, allowing or denying only its own inclusion in a provenance. The downside of this approach is that we would have to compare the provenance lists of any two facts. The gain in flexibility may not be worth the effort.

5.2 Timing

In [61] we presented a notion that the reified rules would only be grounded if the time-stamp associated with them was suitable. We introduced a $\mathsf{time}/2$ predicate which associates a rule ID with a timestamp. The same approach can be generalized, we can tag any identifier in the same way. Here we tag each ground fact with a timestamp and present amendments to the metagrounder so that associated timing information can be taken into account at grounding time. For this discussion we assume that the timestamps are created during the reification process and the time is presented as an integer value. Furthermore we assume that the integer value represents POSIX time, i.e., number of seconds elapsed since the first of January 1970, but the presented approach can be adapted to other kinds of time as well, provided that there exist well-defined comparisons and basic arithmetics on them. Our filtering is done in a straightforward manner by modifying the starting point of each flow, i.e., the point where ground facts are translated to the internal representation as explained in Section 4.4. We follow the same approach for augmenting the identifiers of the reified ground facts as we did in the previous section, but instead of using a $\mathsf{prov}/2$ tag, we use a $\mathsf{time}/2$ tag, which relates a reified identifier with a number representing the POSIX time (in seconds), for example $\mathsf{time}(18, 1383830800)$.

To filter out facts which are "too old" we use the following metarule, where we only allow those identifiers, which are within a predefined time window from the current moment in time:

$$\begin{aligned} \mathsf{allow}(Id) \leftarrow & \\ & \mathsf{time}(Id, T), \\ & \mathsf{currenttime}(C), \\ & \mathsf{maxage}(M), \\ & C - T < M. \end{aligned} \tag{5.12}$$

We assume that as a parameter we have defined a maximum age by $\mathsf{maxage}/1$ and that we have the current time available using $\mathsf{currenttime}/1$. We can use the Lua integration to obtain the current time from the operating system dynamically[1]. The assumption that each reified fact has a timestamp may be too strong and the policies should reflect this. One approach is that any constraints apply only to those facts which have a related timestamp and facts without timestamps are used without any constraints. The following metarules implement this allowance by default so that any fact that has not been tagged with $\mathsf{time}/2$ is included in grounding while facts which have timing information are constrained.

[1] We use atom $\mathsf{currenttime}(@\mathit{gettime}())$ with the corresponding Lua function `gettime()` calling `os.time()`.

$$\begin{aligned} \mathsf{allow}(Aid) \leftarrow & \\ & \mathsf{assert}(Aid), \\ & \mathbf{not}\ \mathsf{hastime}(Aid). \\ \mathsf{hastime}(Aid) \leftarrow & \\ & \mathsf{time}(Aid, T). \end{aligned} \tag{5.13}$$

As an alternative to the previous rule we can list the names of the atoms which are grounded even without a timestamp. We use the predicate $\mathsf{timeless}/1$ to enumerate the names of the atoms which are grounded even without a time tag.

$$\begin{aligned} \mathsf{allow}(Aid) \leftarrow & \\ & \mathsf{head}(Aid, Hid), \\ & \mathsf{pred}(Hid, Pn), \\ & \mathsf{timeless}(Pn), \\ & \mathbf{not}\ \mathsf{hastime}(Aid). \\ & \\ & \mathsf{timeless}("reached"). \end{aligned} \tag{5.14}$$

Note that this same approach is also applicable for the provenance information. In the previous section we required that for a ground fact to be included in the grounding, it must have associated provenance information. It is straightforward to adapt the above rules to use $\mathsf{prov}/2$ instead of $\mathsf{time}/2$. Similarly we can refine the applicability of the timing policy to be more fine-grained, for example by associating each (reified) rule with age constraint particular to that rule itself.

To finalize this section we sketch two extensions enabled by the timing metarules. The first one provides explicit time intervals for both the facts and the rules, which allows limiting policies for different ages of information, e.g., stating that for certain policies should apply to information that was known last week. Distinguishing information in this way permits us to implement *temporal data models* [10, 102] such as *bitemporal data* where a piece of information has a timestamp for validity and transaction. This is not quite straightforward as it is necessary to detect and handle overlapping intervals. The second extension is a different interpretation of the $\mathsf{time}/2$ relation. So far it has represented a timestamp for a fact, but we may also consider it as the time that was used for the computation producing the fact. This enables reasoning about the performance of the system.

5.3 Syntactic Analysis

The metagrounding process described earlier does not produce any results for unsafe rules. However, we can syntactically detect unsafe variables within one rule and mark them as unsafe. These act as rudimentary error messages. “Safety” here means that any variable referred in the rule must occur in at least one positive body literal in the rule. The following rules implement this analysis:

$$
\begin{aligned}
\mathsf{safevar}(Rid, Vn) \leftarrow & \\
& \mathsf{rulevar}(Rid, Vid), \\
& \mathsf{var}(Vid, Vn), \\
& \mathsf{alist}(Predid, N, Vid2), \\
& \mathsf{var}(Vid2, Vn), \\
& \mathbf{not}\ \mathsf{head}(Rid, Predid), \\
& \mathbf{not}\ pos(Predid).
\end{aligned} \tag{5.15}
$$

$$
\begin{aligned}
\mathsf{unsafevar}(Rid, Vn) \leftarrow & \\
& \mathbf{not}\ \mathsf{safevar}(Rid, Vn), \\
& \mathsf{rulevar}(Rid, Vid), \\
& \mathsf{var}(Vid, Vn).
\end{aligned} \tag{5.16}
$$

$$
\mathsf{unsaferule}(Rid) \leftarrow \mathsf{unsafevar}(Rid, Vn). \tag{5.17}
$$

The dereifier has been modified such that rules tagged unsafe are not rendered at all. The same approach can easily be extended to check the names of the heads of reified rules for blacklisted names so they are not ground. These kind of rules may also emit facts which can be interpreted as warnings or errors, which can then be logged at the evaluator or even submitted back to the owner of the rules.

5.4 Checking for Interoperability

In this section, we outline a method for reasoning about the interoperability of reified rules. We have two sets of rules A and B and we want to check their interoperability. The most simple interoperability criteria can be stated as follows: if a rule in A refers to a predicate in its body and that predicate is not in any of the heads in A, then it must be in the head of a rule in B. It is also possible to derive a dependence relation for these predicates and since A and B are assumed to be distinct entities, this relation forms an ordering for the communication. From this ordering we can deduce whether it is possible to derive a particular fact when A and B are evaluated together. If a fact cannot be derived we must either conclude that the rules are not interoperable or then we need to add more information to decide whether the omission is acceptable. A straightforward approach to this is to list fact names which belong to the external interface and only consider them in the analysis. We present an example showing this approach below.

Example 5.4.1 *We define a simple protocol where a client initiates a connection to a server, sends a data packet and finally disconnects itself. Here sending a message occurs when a new fact is derived and receiving a message is indicated by using a fact in the body of a rule. The server is expected to acknowledge each message from the client. The messages or* Protocol Data Units *(PDU) are named accordingly. This is a simplistic model, but it nevertheless allows us to demonstrate features of two-way communication and how they can be analysed. Note that here we have named the client as "a" and the server as "b".*

The rules for the client a are as follows:

$$\begin{aligned}
\mathsf{a\_init\_c}(\mathsf{client1}).& \\
\mathsf{a\_init\_s}(\mathsf{server1}).& \\
\mathsf{connect}(A, \mathit{Target}, 1) \leftarrow& \\
&\mathsf{a\_init\_c}(A), \\
&\mathsf{a\_init\_s}(\mathit{Target}). \\
\mathsf{data}(\mathit{This}, \mathit{Target}, N+1) \leftarrow& \\
&\mathsf{connect\_ack}(\mathit{Target}, \mathit{This}, N), \\
&\mathsf{a\_init\_c}(\mathit{This}). \\
\mathsf{disconnect}(\mathit{This}, \mathit{Target}, N+1) \leftarrow& \\
&\mathsf{data\_ack}(\mathit{Target}, \mathit{This}, N). \\
&\mathsf{a\_init\_c}(\mathit{This}). \\
\mathsf{stop}(\mathit{This}, N) \leftarrow& \\
&\mathsf{disconnect}(\mathit{This}, \mathit{Target}, N), \\
&\mathsf{a\_init\_c}(\mathit{This}).
\end{aligned}$$

The rules for the server b are as follows:

$$\begin{aligned}
\mathsf{b\_init\_s}(\mathsf{server1}).& \\
\mathsf{connect\_ack}(\mathit{This}, \mathit{Client}, N+1) \leftarrow& \\
&\mathsf{connect}(\mathit{Client}, \mathit{This}, N), \\
&\mathsf{b\_init\_s}(\mathit{This}).
\end{aligned} \tag{5.18}$$

$$\begin{aligned}
\mathsf{data\_ack}(\mathit{This}, \mathit{Client}, N+1) \leftarrow& \\
&\mathsf{data}(\mathit{Client}, \mathit{This}, N), \\
&\mathsf{b\_init\_s}(\mathit{This}). \\
\mathsf{stop}(\mathit{This}, N) \leftarrow& \\
&\mathsf{disconnect}(\mathit{Client}, \mathit{This}, N), \\
&\mathsf{b\_init\_s}(\mathit{This}).
\end{aligned}$$

We create an incompatible version of the server by modifying the above basic server so that we send a wait *message before sending a* data_ack*:*

$$\begin{aligned}
\mathsf{data\_ack}(\mathit{This}, \mathit{Client}, N+1) \leftarrow& \\
&\mathsf{wait}(\mathit{This}, \mathit{Client}, N+1), \% \quad \mathit{added} \\
&\mathsf{data}(\mathit{Client}, \mathit{This}, N), \\
&\mathsf{b\_init\_s}(\mathit{This}). \\
\mathsf{wait}(\mathit{This}, \mathit{Client}, N+1) \leftarrow& \\
&\mathsf{data}(\mathit{Client}, \mathit{This}, N), \\
&\mathsf{b\_init\_s}(\mathit{This}).
\end{aligned}$$

The execution model of these rules is the same as described in our use case description in Section 2.3. Here the client and server are both nodes which read and publish their PDUs via a common blackboard.

As mentioned earlier, we first need to define what subset of the facts are to be considered. One option is to augment the original rules with facts declaring those fact names which belong to the interface. Another option is to attach this information to the reified representation of the rules. For brevity we choose the latter one: we assume that the reified representation contains a set of facts based on $\mathsf{interface}/1$ which identifies the facts we should consider. Next we define a basic criteria for interoperability over these interfaces for two sets of rules, which can both be deduced for any rules. This can be stated in two ways, both having the same meaning in this system.

1. "Sent message can be received": an interface fact which can be derived one side is used by another side in rule bodies.

2. "An expected message is sent": an interface fact which is used in rule bodies on one side can be derived on the other side.

In order to make stronger statements about the compatibility, we need to augment rules with more information. The rules themselves can be thought of as containing an order for the facts: a fact in the head comes after the facts in the body. Assuming this we can produce timelines of produced (sent) and used (received) facts for both sets of rules. With this information we can produce generic information such as whether the rules admit a certain message exchange to take place so that we can compare the mutual compatibility of the timelines from both sides. However other correctness properties of timelines need to be specified by providing additional information. Such an additional specification could for example be that when A sends a connection request $\mathsf{connect}$ and receives a positive acknowledgement, then A will always be able to send a disconnection request $\mathsf{disconnect}$.

As an example we present a simple but generic analysis and its application to the rules of the protocol in Example 5.4.1 above.

Example 5.4.2 *The first two rules define a message exchange and an ordering of messages. Message $PDU2$ (PDU stands for Protocol Data Unit identifying the message) is followed by another $PDU1$ if $PDU2$ is referred to in the body and $PDU1$ is referred to in the head.*

Here the predicate $\mathsf{msgexch}/3$ *formalizes the criteria "Sent message can be received'.*

$$
\begin{aligned}
\mathsf{msgexch}(R1, R2, PDU) \leftarrow \quad & \\
& \mathsf{head}(R1, P1), \\
& \mathsf{pred}(P1, PDU), \\
& \mathsf{interface}(PDU), \\
& \mathsf{body}(R2, P2), \\
& \mathsf{pred}(P2, PDU). \\
\mathsf{nxtmsg}(R1, \mathit{PDU2}, \mathit{PDU1}) \leftarrow \quad & \\
& \mathsf{head}(R1, P1), \\
& \mathsf{pred}(P1, \mathit{PDU1}), \\
& \mathsf{interface}(\mathit{PDU1}), \\
& \mathsf{interface}(\mathit{PDU2}), \\
& \mathsf{body}(R2, P2), \\
& \mathsf{pred}(P2, \mathit{PDU2}).
\end{aligned}
$$

A message is received when it is found in a rule body and sent when it is found in the rule head. Message that is sent, but cannot be received is flagged by the predicate $\mathsf{norec}/3$*. Here we also assume that the rule identifiers are flagged with* $\mathsf{prov}/2$ *and we use this provenance information to identify the sender and the receiver.*

$$
\begin{aligned}
\mathsf{procname}(N) \leftarrow \quad & \\
& \mathsf{prov}(R1, N). \\
\mathsf{msgsend}(\mathit{Prov1}, PDU) \leftarrow \quad & \\
& \mathsf{head}(R1, P1), \\
& \mathsf{prov}(R1, \mathit{Prov1}), \\
& \mathsf{interface}(PDU), \\
& \mathsf{pred}(P1, PDU).
\end{aligned}
\tag{5.19}
$$

$$
\begin{aligned}
\mathsf{msgrec}(\mathit{Prov1}, PDU) \leftarrow \quad & \\
& \mathsf{body}(R1, P1), \\
& \mathsf{prov}(R1, \mathit{Prov1}), \\
& \mathsf{interface}(PDU), \\
& \mathsf{pred}(P1, PDU). \\
\mathsf{norec}(A, B, PDU) \leftarrow \quad & \\
& \mathsf{msgsend}(A, PDU), \\
& \mathsf{interface}(PDU), \\
& \mathbf{not}\ \mathsf{msgsend}(B, PDU), \\
& \mathsf{body}(R1, X), \\
& A \,! = B, \\
& \mathsf{prov}(R1, B).
\end{aligned}
$$

Finally we define the predicate timeline/4 *which presents the timeline for each side using send-receive-send triplets and flags timeline fragments* incompatible/5*, when one side expects an exchange which the other side cannot fulfill. This indicates a mutual incompatibility as outlined above.*

$$
\begin{array}{ll}
\mathsf{timeline}(\mathit{Id}, \mathit{PDU1}, \mathit{PDUT}, \mathit{PDU2}) \leftarrow & \\
 & \mathsf{prov}(R1, \mathit{Id}), \\
 & \mathsf{prov}(R2, \mathit{Id}), \\
 & \mathsf{msgexch}(R1, T, \mathit{PDU1}), \\
 & \mathsf{nxtmsg}(T, \mathit{PDU1}, \mathit{PDUT}), \\
 & \mathsf{nxtmsg}(R2, \mathit{PDUT}, \mathit{PDU2}). \\
\mathsf{incompatible}(\mathit{Id1}, \mathit{Id2}, & \\
\qquad \mathit{PDU1}, \mathit{PDU2}, \mathit{PDU3}) \leftarrow & \\
 & \mathsf{procname}(\mathit{Id1}), \\
 & \mathsf{procname}(\mathit{Id2}), \\
 & Id1 \; ! = Id2, \\
 & \mathsf{timeline}(\mathit{Id1}, \mathit{PDU1}, \mathit{PDU2}, \mathit{PDU3}), \\
 & \mathsf{norec}(\mathit{Id1}, \mathit{Id2}, \mathit{PDU3}).
\end{array}
$$

Applying these rules to the incompatible reified protocol in Example 5.4.2 produces the following results.

procname(”b”). procname(”a”).
norec(”b”,”a”,”wait”).
timeline(”b”,”connect_ack”,”data”,”data_ack”).
timeline(”b”,”connect_ack”,”data”,”wait”).
timeline(”a”,”connect”,”connect_ack”,”data”).
timeline(”a”,”data”,”data_ack”,”disconnect”).
incompatible(”b”,”a”,”connect_ack”,”data”,”wait”).

The produced results identify the incompatibilities: norec/3 *indicates that “a” cannot receive a $wait$ message and* incompatible/5 *indicates the context of the incompatibility.*

5.5 Cryptographically Securing Results

As we can maintain the provenance of individual facts used for reasoning, we have the possibility of benefiting from existing cryptographic algorithms on a fine-grained level. In order to secure results cryptographically, we need to ensure that a result that uses our data is not published to parties whose data was not used in the computation of that result. The provenance rules outlined in the previous section serve to control whose data can be combined with ours. Conceptually, we construct a system in which user's data is considered as an admission fee to the result: if you are not contributing to the result, then you should not have access to it. Note that implementing this solution is not feasible by ASP only, since minimal requirements involve cryptographic functions for validating and encrypting content. These functions could be implemented either via an external tools, such as that provided by the Lua interface or otherwise within the solver or grounder in a similar manner as arithmetic operations are currently implemented. We assume the availability of a public-key crypto system such as RSA [94], where each participant has a public-private key pair. As a starting point each fact can be encrypted or attached with a digital signature [66] calculated for that fact. Although each fact can be combined with the public key of the owner of the fact, in reality this practice is inefficient. Therefore we require to the public key for the owner of a fact available by other means. In this section we assume that the public keys of any participant of our ubiquitous system are available for all nodes, especially the entity performing the metaevaluation. We also assume that the provenance information is complete so that the provenance of each value that contributed to the result is maintained. This is important because it ensures that no participant can block other participants by presenting a fact with exactly the same arguments. Hence it would not possible that a participant provided the information (or "paid the price") but did not get access because someone else presented the same fact. Treating individual facts and rules as participants in a cryptographic system enables multiple designs. Nevertheless here we present one straightforward application: we would like to publish the results of our metaevaluation or metagrounding process but in such a manner that only those parties whose data has been used in the reasoning process are able to read the results. As shown earlier in Section 5.1, we can attach policies to the data so that we can control what data can be combined by our rules. We can for example require that a fact is filtered out if its signature cannot be verified or its content cannot be decrypted. However once we have obtained a ground rule with a list of provenances, we can encrypt the result targeting only those sources of the facts which are in the provenance list. This is known as *multiple-key public-key cryptography*. If we have a fixed number of participants N, we can use the approach described by Boyd [19] to target the encryption to any subset of the N participants, so that only the targeted participants can decrypt the result. However if we do not want to limit ourselves to a fixed number of participants, we use the following conference key distribution and broadcasting algorithm described by Schneier in [96]. The metaevaluator that performs the computation performs the following steps for any new piece of information with a provenance list:

1. Pick a random key K.
2. Encrypt the newly produced content with K.
3. For each unique identity in the provenance list encrypt the key K with the public key associated with the identity.
4. Publish the content and the encrypted keys along with the list of whom they are targeted to.

The recipients can now search the keys for their own identity to find the decryption key which is targeted to them and then use that key to decrypt the content. If it is unacceptable to publish the list of recipients, it is possible to use a threshold scheme, as developed by Berkovits in [11]. We do not present further mechanisms in this work, but we highlight an interesting point in the design space. As presented by Shamir [98] threshold schemes allow flexibly setting the policies on the conditions in which the decryption can occur. For instance it may be possible to state that for a rule with five body literals and hence five possible sources of facts, the decryption requires the keys of at least two owners of the facts. Furthermore we may require that if one of the facts pertains to a specific field, such as geolocation, the owner of that fact is always required for decryption.

5.6 Measurements

We performed measurements of our metagrounder over the Hamiltonian circuit program given in Appendix A with increasing sizes of the same pseudorandomly generated input graphs[2]. We also verified the results of our metaevaluator against the results produced for the unreified rules by gringo. For measurements we used the basic metagrounder, which tracks the provenance but we did not include any handling of metainformation.

We ran the tests on a MacBook 2,26 GHz Intel Core 2 Duo with 4GB of memory running OS X 10.8.5. We generated random graphs with 3, 4, 5, and 6 nodes and reported the average of 5 runs for each. The whole process is implemented as a shell script orchestrating reification, dereification, and actual solving. We measured the time using the time command but we also report timing reports from the solver implementation for more details. The results are given in Table 5.1. We tabulate the number of nodes in the random graph and the number of answer sets or Hamiltonian cycles for that graph. Then we give the total time of running the metagrounder and distinguishing between the amount of time used by the solver and the metagrounder. The solver part is the most relevant as it produces the ground results in reified format, but since in practice the dereification is also needed to produce non-reified results and so it is also measured.

Unfortunately, doing a reasonable performance comparison between our approach and gringo (or any of the other mentioned solvers) is hard to realize because the metagrounder implementation exhibits an exponential (or worse) growth of running time even on small problems giving us only few measurements. Grounding the same Hamiltonian circuit for three nodes by gringo is reported to take 0.002s, whereas the metagrounder takes around 28 seconds to produce the same result. For four nodes the metagrounder takes around 30 minutes, whereas gringo takes 0.004s. The metagrounder execution for graphs with five nodes took eight hours and for six nodes it ran out of memory after about ten hours and was interrupted. We give the total time taken for the whole metagrounding process and also the time taken by the metagrounding and the dereification (converting the reified representation back to rules) separately.

The number of answer sets produced in the metagrounding step is one by construction as everything is in the same answer set. Therefore it seems that the number of choices and conflicts reported by –stats $=$ 2 flag for clasp are always zero.

Observing the running metagrounder processes for four nodes, the maximum use by gringo was 180 MB of memory, whereas clasp used at most 45 MB of memory. In cases where the computation produced a result (for graphs under six nodes) the system had free physical memory available so thrashing was not a bottleneck. The CPU use reported by the OS X “Activity Monitor” utility for gringo was high, between 90 and 100 percent,

[2] Produced by planar-1.2 at http://research.ics.aalto.fi/software/asp/misc.

Nodes	Answer sets	MG total	MG solver	MG dereify	Solver total	clasp
3	2	38	28	10	0.020	0.001
4	6	1863	1835	28	0.021	0.001
5	12	24671	22819	1852	0.022	0.003
6	14	-	-	-	0.024	0.003

Table 5.1: The elements of reified rules, all times in seconds

Node #	AS #	A time	A choices	A conflicts	B time	B choices	B conflicts
10	76	0.154	322	221	0.026	311	201
15	2504	2.390	12730	8885	0.484	13332	9624
20	28400	22.619	85978	48067	4.337	119266	74867
21	64876	46.673	160288	79581	8.908	261977	161876
22	99130	75.822	255827	130288	15.047	436729	278288

Table 5.2: Metaevaluation measurements, all times in seconds. Case A refers to the metaevaluation and B to the non-reified evaluation.

but for `clasp` it remained under 5 percent. This seems to indicate that most of the work was spent grounding. We reran the tests informally on other machines with essentially the same results: for three nodes metagrounding takes seconds, for four it takes tens of minutes and for five it takes hours.

We continued by measuring the performance of our metaevaluator as described in Section 3.3. We generated graphs using the same mechanism as above starting from 10 nodes and ending at 30 nodes. These graphs were combined with the Hamiltonian circuit evaluator presented in Appendix A and then grounded by using `gringo`. These programs were then “flattened” syntactically so that any ground atom is rewritten as an atomic proposition. For Example $\mathrm{edge}(1, 2)$ is converted to edge_1_2. This program is then reified, combined with the metaevaluator and then executed with `gringo` and `clasp`. We only report the time reported by `clasp` as the programs are ground already and the time used by `gringo` is negligible. In addition to the measured times we report two internal statistics provided by `clasp` in its extended statistics.

For these results we find that metaevaluation takes roughly five times longer than the evaluation of the non-reified representation. The memory footprint of `clasp` remained below 10 MB for all cases. We can also perceive that the metaevaluated program makes less choices and has less conflicts than the non-reified program.

Chapter 6

Discussion and Related Work

To the best of our knowledge this thesis presents the first representation of grounding of ASP programs in terms of ASP rules. We have shown that it is possible to create a metagrounder using ASP rules with function symbols, but at a very high cost in performance. There is earlier work on metaevaluation of answer set programs by Eiter et al. [34, 33] and Gebser et al. [45]. Furthermore Delgrande et al. [30] present a framework for including preferences in logic programs, which is quite similar to metaevaluation. Previous works implement the solver part where their input are ground programs. Their purpose is to take into account preferences between answer sets or modifying the solver behaviour for optimizations (for instance minimizing or maximizing an objective function) across the answer sets. Our work can be seen as a continuation of previous work on metaevaluation in the sense that we extend the metaevaluation approach to the grounding phase, whereas the previous work concentrated on the solving phase only. In addition to the metagrounder, we also implemented a similar metaevaluator for propositional programs and proved its compatibility with the existing metaevaluators. Metaevaluators resemble each other in the sense that they do not need to generate new syntactical elements, because they only need to decide whether an existing ground rule is applied or not. The key issue in our approach is the generation of new identifiers, which allows us to produce multiple ground versions of one non-ground rule. We aimed to be able to use our metaevaluator in the metagrounder work. However these two have remained largely separate entities. The main reason for this separation is that most of the metagrounder work is related to the handling of the identifiers.

In contrast to existing tools (especially `gringo`), we aim to give unique identifiers to all elements of a rule in our representation of reified rules. The contrast is minor, but it reflects our slightly different starting point and requirements for metaevaluation. We expect that the rules are handled in reified format and that they may come from many sources, including untrusted ones. Similarly the data itself over which the rules operate, may come from different sources. Therefore we need to keep track of provenance when applying metaevaluation. Our metaevaluation approach builds on the management of identifiers by the ASP rules. While this is more cumbersome for metarules, it allows clear separation of the evaluation of rules and management of metainformation, such as provenance or timing, which may be included in the information. Another important implication is that our approach allows the same data from multiple sources to have a different provenance. Early papers on metaevaluation, such as Eiter et al. [34], presented a similar kind of representation to ours, but without variables. The implementation for reification in `gringo` is different and follows the format in Gebser et al. [45] whereby the reified rules are constructed using a combination of nested function symbols and identifiers, thereby partially constructing the parse tree inside the symbols. For example the rule

$$s \leftarrow \textbf{not}\ q,\ \textbf{not}\ r$$

is reified by `gringo` as

$$\begin{array}{r}\mathsf{rule}(\mathsf{pos}(\mathsf{atom}(s)), \mathsf{pos}(\mathsf{conjunction}(2))).\\ \mathsf{set}(2, \mathsf{neg}(\mathsf{atom}(q))).\quad \mathsf{set}(2, \mathsf{neg}(\mathsf{atom}(r))).\end{array}$$

Gebser et al. [47] and Brain et al. [79] present methods for debugging ASP programs. Both describe a *tagging technique* where the program under investigation is compiled into another one containing *control atoms* or tags added to the original rules. For instance, the authors in [79] propose adding new atoms referring to a rule r: $\mathsf{ap}(r)$ and $\mathsf{bl}(r)$ for expressing whether rule r is *applicable* or *blocked*. In addition to these they propose an $\mathsf{assign}/2$ predicate for variable bindings. The debugging primitives in [47] are similar to tags in our work and these primitives can be used to express unsatisfied rules, violated integrity constraints, unsupported atoms, and unfounded loops pertaining to some atoms. The representation of the ASP programs is in a similar reified format as in our work. Although our approach of modifying the metaevaluator and metagrounder are very similar, we do not specify a dedicated debugging or control language. Instead we propose a data driven policy mechanism for controlling evaluation. It seems that the debugging mechanism and our approach are compatible in that both could simulate each other. However in contrast to our approach the debugging mechanism does not take into account the creation of new ground rules with new identifiers so that their result could be used in(meta)evaluation. Janhunen et al. [62] propose several criteria for testing answer-set programs. Although this approach does not explicitly mention metaprogramming, it builds on concepts which rely on metalevel access to the program subject to testing.

There is existing work on the combination of ASP rules over semantic web information by Eiter et al. [35, 36], which is similar to our work in [78]. Both focus on using ASP for a straightforward implementation rather than metaprogramming. However in [36] the authors do present some metalevel constructs, but their objective is to integrate ASP with external reasoners and libraries. In a similar vein, Wielemaker et al. [113] suggest to use Prolog as an application platform for Semantic Web.

In addition to debugging and database-centric provenance, there is a related concept of *explanations* [112, 104, 3] for queries or rules in the deductive database field. Initially an explanation consists of a trace from the database internals. This serves to construct a proof tree which is offered to the user, possibly with visualization aids to be used as a debugging tool. There can be several kinds of explanations at different abstraction levels. These are more complex entities than the simple provenance mechanisms presented in this thesis. A typical mechanism for producing the explanations is a Prolog metainterpreter, instrumented to include database internal information. Such a system along with mechanisms for three different abstraction layers is described by Mallet et al. in [80]. The same approach has been applied to to ASP by Pontelli et al. [90], where the authors present a justification graph explaining how a particular atom has been derived. They implement their approach within the `smodels` solver and use ASP-Prolog [38]. In contrast to the explanation or justification based systems, our provenance approach is much simpler as we present a list of contributing atoms and how they have been mapped to existing rules.

As previously mentioned there are different provenance mechanisms which aim to explain what pieces of information were used to produce particular query results. The *lineage* approach [28] presents as a set of *source tuples* (or facts in logic programming terms) contributing to the result. The *why provenance* [22] extends this by allowing multisets for sources. Finally *provenance semirings* [53] encode the sources in the most general way, presenting a polynomial which encodes the used source tuples. Our approach maintains a set of lists for each produced fact, which can be thought of as sets of variable assignments. These lists combined with the access to the reified rules describe the provenance of a fact as accurately as the provenance semirings. However the actual provenance information can be easily accessed in filtering rules and other metarules without parsing the polynomials present in provenance semirings. If the rules are not available, then we lose information, making our approach similar to the lineage model.

The framework for adding generic annotations to RDF by Zimmermann, Polleres et al. [115] introduces annotated RDF triples, essentially 4-tuples, where the additional element is the *annotation value* associated with an *annotation domain* D, where

$$D = \langle L, \oplus, \otimes, \bot, \top \rangle$$

The annotation domain D must be an idempotent, commutative semiring and different purposes for the annotations require different choices for the domain. Some examples are as follows:

$$D_{01} = \langle \{0,1\}, \max, \min, 0, 1 \rangle \qquad\qquad D_{\text{fuzzy}} = \langle [0,1], \max, \cdot, 0, 1 \rangle$$

The article presents a set of inference rules for deducing RDFS subclass, subproperty and typing relations. RDFS annotated with values from Boolean annotation domain D_{01} corresponds to classical RDFS. In addition to RDFS inference the authors present a query language AnQL, which is an extension of SPARQL allowing querying of annotated RDF graphs. Furthermore, they present semantics of AnQL and a generic approach for combining and evaluating multiple annotation domains (e.g., fuzzy+temporal) domains. The annotations resemble *provenance semirings* [53] and the main differences to our work are as explained above: we keep less information. However in contrast to our work, [115] provides a generic framework for presenting and combining different annotations. In our case we allow arbitrary rules to evaluate our annotations. Lopes, Polleres et al. also use this annotation method to implement a triple level access control mechanism in [76].

In our early work in [78] we implemented a resource allocator using ASP in a distributed setting. We identified a need for the service users and service providers to express what can be done and by whom. As referred earlier there are several existing policy languages which use the logic programming paradigm: Binder [32], Delegation logic [72], SecPAL [9], and DKAL [54]. All of these languages are eventually translated into straightforward rules. For instance in SecPAL [9] the statements “bob *says* alice *can* read the /home directory” could be rewritten as

$$\mathsf{can}(\mathsf{right}(\mathsf{read}, \mathsf{alice}, /\mathsf{home})) \leftarrow \mathsf{says}(\mathsf{bob}, \mathsf{right}(\mathsf{read}, \mathsf{alice}, /\mathsf{home}))$$

and then evaluated as part of any other ASP program. We can extend this approach in two possible ways: firstly by offering the provenance information as a first class citizen in the underlying ASP evaluation and secondly by adapting concepts from these languages in the metarules, thereby effectively re-implementing parts of the policy languages. Furthermore our current approach intends to inject the policies (or other definitions) within the grounding or general evaluation of the rules themselves, whereas earlier work compiles the policies directly into rules. Our provenance approach allows the extension the policies to individual pieces of information.

Chapter 7

Conclusions and Future Work

In this thesis we have implemented both the grounding and solving phases of an ASP toolchain using ASP-metaprogramming techniques. Both programs operate directly on a reified format and also produce the results in the same reified format. This mechanism allows us to track the provenance of individual atoms through the rule metaevaluation and control their use with other ASP rules which formalize policies. We propose our ASP metaevaluation approach to be used as an enabler for sharing rules and data from multiple potentially untrustworthy sources. We can inspect the reified rules syntactically by banning them if needed and we can control what data these rules can access based on the ownership of the data. Any participant of a ubiquitous system can enhance its interoperability by publishing the rules that it uses itself. The described mechanism allows any other participant to evaluate the rules in a safe manner: we can exclude foreign rules which produce facts that might interfere with our own internal evaluation. Furthermore we can use the provenance mechanism to include and operate on other kinds of quantitative metadata, such as timing. We have argued that our metaevaluation mechanism will always produce a result or reject the shared behavioural descriptions, contributing to safety of the system and its individual participants.

Regarding the general applicability of our approach there are some boundary conditions. The main benefits of ASP manifest in optimization or constraint solving in the presence of combinatorial complexity, for instance making a choice under multiple preferences. The subset of ASP we use is limited so it cannot describe arbitrary behaviour; this will require a Turing complete programming language. Furthermore our own challenges in implementing a simple counter inside `gringo` show that relying only on ASP is not feasible. In addition to the lack of expressivity, our main problem currently is the unnecessarily slow performance of metagrounding. The most likely culprit is the list mechanism, which is used in representing the IDs and facts. It allows us to embed multiple answer sets within one answer set, but at the same time it forces us to re-implement many of the ASP mechanisms in a suboptimal way. Yet the performance of our metaevaluator shows that the metaevaluation approach is not completely unfeasible, since it seems that for grounding purposes we need support for both expressing identifiers and lists in order to achieve an agreeable performance. For rules with variables, the provenance data must be resolved at grounding time. The feasibility of tracking provenance via ASP only is also tied to solving this issue.

One of our design decisions in Section 4.2 was to implement the metagrounder in one step, grounding all rules at the same time. The alternative we considered, but abandoned, was to ground each rule to its own answer set. The metagrounding implementation itself would have been straightforward, but by its nature it would have required multiple iterations of the grounding step. This is because ground facts or rules produced in the different answer sets must be collected to a single set of facts so that newly produced ground results are available to be used at the next iteration. At each iteration we must compare against the previous results to detect if new results were produced and we can stop only when we have reached a fixpoint. The additional computational costs here are the combination of new content from answer sets and the detection of change. This may become inefficient when we have recursive rules, such as the Hamiltonian circuit example, where a new ground fact produced in one answer set could be used in combination with existing facts by some other rule, but it is not visible to any other answer set. There are some benefits: firstly this approach is easy to parallelize: each rule can be handled by an independent process. Secondly we can avoid the costly tracking and managing the identifiers as it is straightforward to create the system so that one answer set represents one new identifier. The actual generation of unique identifiers would have to be done outside of the rules when combining the answer sets.

Earlier in Section 4.3 we noted that if we knew the arity of each atom, we could generate reified atoms which contain the predicate name followed by terms of the original. This approach would eliminate the need to use lists for representing facts, but at the cost of losing the flexibility of adapting to arbitrary arities. We would have to generate rule stubs for all of the known arities without the certainty of another arity appearing in the future. For larger aritites it is possible to automatically split relations by introducing artificial keys for each sub-relation and hence rewrite the rules to have atoms of a fixed maximum arity. This would require a mechanism for producing identifiers for the artificial keys. Even with the limit of certain arities, we would still need an identifier mechanism for provenance purposes.

The possibility of safe identifier generation or safe and relatively efficient metaprogramming facilities for ASP are mostly tooling issues. There is work in combining Prolog and ASP, which has been used in conjunction with modifications of `smodels` [90] and this could offer a solution for both generation of identifiers and perhaps use of lists. As an additional benefit, facilities of a full programming language would also be available. However this would require nontrivial changes to the metaevaluator. Furthermore an interface to a generic programming language would allow the use of various cryptographic primitives. Whether this would bring more benefits than using the embedded Lua interpreter is unclear. Once the performance of the metagrounder is acceptable, it is then possible to combine the metaevaluation and metagrounding phases in an attempt to optimize the whole chain and to find opportunities for increasing its performance there. There are many possibilities of benefitting from combining the provenance with cryptographic primitives. The gold standard to aim for is *homomorphic encryption* [93] which allows reasoning over encrypted data so that the end result of the reasoning is also encrypted. This allows computing remotely so that the entity performing the computation only operates on encrypted data and hence cannot access the data or the rules. A more architectural line of research is to take into account different consistency guarantees for different data and its effects on the evaluation. As mentioned in Section 2.1, it is conceivable that some parts of the blackboard and the data on it could reflect the underlying consistency properties for different sources of data, which may differ depending on the data source.

Appendix A

Examples of Reified Rules

In this appendix we illustrate some rules and their reified format. We will focus on presenting the complete and verbose reified results. The presented Hamiltonian cycle example also serves as a running example for grounding in the earlier sections starting from 3.1.2. We also give a tabulated description of the predicates used to represent the reified rules.

Hamiltonian cycle

Here we present a simple answer set program for computing all Hamiltonian cycles , i.e., cycles which visit each node of a graph only once, of an arbitrary directed graph. Checking the existence of Hamiltonian cycles in a graph is known to be an NP-complete decision problem [64]. The program we present here is based on an example given in [84].

Answer Set programming Rules

The following rules assume that the graph is represented by directed edges using edge/2 predicate. The program will produce answer sets for each valid solution using oncycle/2 predicate. Furthermore we assume that the graph nodes are represented by numbers, always starting from 1. We walk through the program below.

We construct the cycle so that by default we include an edge of the original graph, if it is not already present in the cycle.

```
oncycle(X,Y) :- edge(X,Y), not other(X,Y).
```

We define the exception to the above default rule, expressing that an edge is already in the cycle.

```
other(X,Y) :- edge(X,Y), edge(X,Z), Y != Z, oncycle(X,Z).
```

We enforce the reachability of each node along the cycle.

```
reached(Y) :- reached(X), oncycle(X,Y), edge(X,Y).
```

For base case we always assert that the first node belongs to the cycle:

```
reached(1).
```

We project existing information of the edges to derive the nodes of the graph.

```
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).
```

Finally we constrain the solution so that every node must be reachable and that no node can exist outside of a cycle. This second constraint contains a parametrized conjunction, a list separated by “:” characters.

```
:- node(X), not reached(X).
:- not oncycle(Y,X) : edge(Y,X), node(X).
```

These rules constitute the actual program. In practice we would hide all other facts except the result in oncycle. The data is provided as facts, which are rules without bodies (uniform representation). As mentioned earlier the rules expect that the reached/1 closure has a starting point and we need to indicate one node as the starting point. Edges are represented as source node and target node. Below is a graph with three nodes and loops in two directions over the nodes.

```
edge(1,2). edge(2,3). edge(3,1).
edge(1,3). edge(3,2). edge(2,1).
```

The solutions for this instance has two answer sets, each containing a solution representing a Hamiltonian cycle. Here we show only the relevant oncycle/2 atoms:

```
{oncycle(2,1) oncycle(3,2) oncycle(1,3)}
{oncycle(3,1) oncycle(2,3) oncycle(1,2)}
```

The ground instance of the above rules with the above data is shown below. We omit the data, which are already ground facts and remain in the grounding. We have used `gringo` to perform the grounding with the following command line (assuming that both data and rules are contained in single file `hamilton_toground.lp`):

```
gringo -t hamilton_toground.lp
```

For other tools the results may differ. The order is not relevant as such and we present the results in the order most suitable for us. First, the projections can be fully evaluated:

```
node(2). node(3). node(1).
```

Next we have the tightly coupled oncycle/2 and other/2 rules. The “obviously true” parts of the rule body, those which map to existing ground facts, have been dropped thereby leaving only those rules that need to be evaluated by the solver:

```
oncycle(2,1) :- not other(2,1).
oncycle(3,2) :- not other(3,2).
oncycle(1,3) :- not other(1,3).
oncycle(3,1) :- not other(3,1).
oncycle(2,3) :- not other(2,3).
oncycle(1,2) :- not other(1,2).
other(2,1) :- oncycle(2,3).
other(3,2) :- oncycle(3,1).
other(1,3) :- oncycle(1,2).
other(3,1) :- oncycle(3,2).
other(2,3) :- oncycle(2,1).
other(1,2) :- oncycle(1,3).
```

As reached(1) is a ground fact, it is not present in the bodies of the first two ground rules. However reached(2) and reached(3) are not ground facts, so they must be present in the bodies of the following ground rules.

```
reached(2) :- oncycle(1,2).
reached(3) :- oncycle(1,3).
reached(2) :- reached(3),oncycle(3,2).
reached(3) :- reached(2),oncycle(2,3).
:- not reached(3).
:- not reached(2).
```

Finally demonstrating the effects of the parametrized conjunction, we get a syntactic augmentation of the constraint to include those atoms which match the criteria in edge given as parameter.

```
:- not oncycle(2,1), not oncycle(3,1).
:- not oncycle(1,3), not oncycle(2,3).
:- not oncycle(3,2), not oncycle(1,2).
```

Reified Representation

In this section we briefly comment the reification results. Figure A.1 shows rules which only have the very basic building blocks; body literals formed in terms of basic predicates. Each rule is prefixed with a comment with the non-reified representation. The final predicate belonging to any reified rule is maxrulectr/1, which has the "identifier space" or the smallest and the largest identifier which are related to this particular rule. In addition to the treelike representation of the rule's actual structure, we can *tag* identifiers using different predicates of arity one. Figure A.1 presents basic reified rules where most of the concepts have already been discussed in Section 3.2. Here we have two tags: rule/1 and pos/1, where the latter always pertains to an atom. Furthermore we produce auxiliary predicates bodycount/2, bodylist/3, rulepred/2, and rulevar/2, which allow direct access to various syntactic elements of the reified rules. They are redundant as the same information is contained in the reified representation, but they are provided as shortcuts for easier implementations of meta rules. For instance, finding all variables in a rule can be cumbersome if they appear in nested syntactic structures and this shortcut allows accessing them directly without implementing scaffolding rules for the search. Figure A.2 introduces two new concepts, the negation and a binary operator, which refer to an arithmetic comparison. The negation is simply a neg/1 tag for the associated atom, whereas the binary operator is a new syntactic construct, bexpr/2, which relates the rule identifier to a representation of the comparison. Here we introduce a representation for a binary operator bop/2 and its two arguments larg/2 and rarg/2. In A.2 we show constraints (headless rules), which are tagged as constraint/1 instead of rule/1 and the construction for parametrized conjunction. Such an element is typed using the composite/1 tag and the list of colon-separated items is represented by tlist/3, which is modeled after alist/3. The count of these items is given by compositenum/2. tlist/3 refers to atoms (or potentially other syntactic elements) which are defined in the usual style. Finally we have the grounded facts in A.4 and A.5. Syntactically facts are rules without bodies, the only difference is tagging them with assert/1 rather than rule/1. The reified rules and data in this section can be produced by the following command-line:

```
aspreify hamilton_toground.lp
```

producing an output file named `hamilton_toground.lp.reified`.

```
%  node(X) :-
%    edge(X,Y).
hasrule(1,30).
rule(30).
bodycount(30,1).
pos(31).
head(30,31).
rulepred(30,31).
arity(31,1).
pred(31,"node").
var(32,"X").
rulevar(30,32).
alist(31,1,32).
bodylist(30,1,33).
pos(33).
body(30,33).
rulepred(30,33).
arity(33,2).
pred(33,"edge").
var(34,"X").
rulevar(30,34).
alist(33,1,34).
var(35,"Y").
rulevar(30,35).
alist(33,2,35).
maxrulectr(30,36).
```

```
% node(Y) :-
%   edge(X,Y).
hasrule(1,37).
rule(37).
bodycount(37,1).
pos(38).
head(37,38).
rulepred(37,38).
arity(38,1).
pred(38,"node").
var(39,"Y").
rulevar(37,39).
alist(38,1,39).
bodylist(37,1,40).
pos(40).
body(37,40).
rulepred(37,40).
arity(40,2).
pred(40,"edge").
var(41,"X").
rulevar(37,41).
alist(40,1,41).
var(42,"Y").
rulevar(37,42).
alist(40,2,42).
maxrulectr(37,43).
```

```
% reached(Y) :-
%   reached(X),
%   oncycle(X,Y),
%   edge(X,Y).
hasrule(1,48).
rule(48).
bodycount(48,3).
pos(49).
head(48,49).
rulepred(48,49).
arity(49,1).
pred(49,"reached").
var(50,"Y").
rulevar(48,50).
alist(49,1,50).
bodylist(48,3,51).
pos(51).
body(48,51).
rulepred(48,51).
arity(51,2).
pred(51,"edge").
var(52,"X").
rulevar(48,52).
alist(51,1,52).
var(53,"Y").
rulevar(48,53).
alist(51,2,53).
bodylist(48,2,54).
pos(54).
body(48,54).
rulepred(48,54).
arity(54,2).
pred(54,"oncycle").
var(55,"X").
rulevar(48,55).
alist(54,1,55).
var(56,"Y").
rulevar(48,56).
alist(54,2,56).
bodylist(48,1,57).
pos(57).
body(48,57).
rulepred(48,57).
arity(57,1).
pred(57,"reached").
var(58,"X").
rulevar(48,58).
alist(57,1,58).
maxrulectr(48,59).
```

Figure A.1: Reified basic rules

```
% oncycle(X,Y) :-
%   edge(X,Y),
%   not other(X,Y).
hasrule(1,2).
rule(2).
bodycount(2,2).
pos(3).
head(2,3).
rulepred(2,3).
arity(3,2).
pred(3,"oncycle").
var(4,"X").
rulevar(2,4).
alist(3,1,4).
var(5,"Y").
rulevar(2,5).
alist(3,2,5).
bodylist(2,2,6).
neg(6).
body(2,6).
rulepred(2,6).
arity(6,2).
pred(6,"other").
var(7,"X").
rulevar(2,7).
alist(6,1,7).
var(8,"Y").
rulevar(2,8).
alist(6,2,8).
bodylist(2,1,9).
pos(9).
body(2,9).
rulepred(2,9).
arity(9,2).
pred(9,"edge").
var(10,"X").
rulevar(2,10).
alist(9,1,10).
var(11,"Y").
rulevar(2,11).
alist(9,2,11).
maxrulectr(2,12).
```

```
% other(X,Y) :-
%   edge(X,Y),
%   edge(X,Z),
%   Y != Z ,
%   oncycle(X,Z).
hasrule(1,13).
rule(13).
bodycount(13,4).
pos(14).
head(13,14).
rulepred(13,14).
arity(14,2).
pred(14,"other").
var(15,"X").
rulevar(13,15).
alist(14,1,15).
var(16,"Y").
rulevar(13,16).
alist(14,2,16).
bodylist(13,4,17).
pos(17).
body(13,17).
rulepred(13,17).
arity(17,2).
pred(17,"oncycle").
var(18,"X").
rulevar(13,18).
alist(17,1,18).
var(19,"Z").
rulevar(13,19).
alist(17,2,19).
bodylist(13,3,20).
bexpr(13,20).
bop(20,"!=").
larg(20,21).
var(21,"Y").
rulevar(13,21).
rarg(20,22).
var(22,"Z").
rulevar(13,22).
bodylist(13,2,23).
pos(23).
body(13,23).
```

```
rulepred(13,23).
arity(23,2).
pred(23,"edge").
var(24,"X").
rulevar(13,24).
alist(23,1,24).
var(25,"Z").
rulevar(13,25).
alist(23,2,25).
bodylist(13,1,26).
pos(26).
body(13,26).
rulepred(13,26).
arity(26,2).
pred(26,"edge").
var(27,"X").
rulevar(13,27).
alist(26,1,27).
var(28,"Y").
rulevar(13,28).
alist(26,2,28).
maxrulectr(13,29).
```

Figure A.2: Reified rules with negation and binary operations

```
% :-
%  node(X),
%  not reached(X).
hasrule(1,73).
constraint(73).
bodycount(73,2).
bodylist(73,2,74).
neg(74).
body(73,74).
rulepred(73,74).
arity(74,1).
pred(74,"reached").
var(75,"X").
rulevar(73,75).
alist(74,1,75).
bodylist(73,1,76).
pos(76).
body(73,76).
rulepred(73,76).
arity(76,1).
pred(76,"node").
var(77,"X").
rulevar(73,77).
alist(76,1,77).
maxrulectr(73,78).
```

```
% :-
% not oncycle(Y,X):edge(Y,X),
% node(X).
hasrule(1,60).
constraint(60).
bodycount(60,2).
bodylist(60,2,61).
pos(61).
body(60,61).
rulepred(60,61).
arity(61,1).
pred(61,"node").
var(62,"X").
rulevar(60,62).
alist(61,1,62).
bodylist(60,1,63).
body(60,63).
composite(63).
rulecomposite(60,63).
compositenum(63,2).
tlist(63,1,64).
neg(65).
qual(64,65).
rulepred(64,65).
arity(65,2).
pred(65,"oncycle").
var(66,"Y").
rulevar(60,66).
alist(65,1,66).
var(67,"X").
rulevar(60,67).
alist(65,2,67).
tlist(63,2,68).
pos(69).
qual(68,69).
rulepred(68,69).
arity(69,2).
pred(69,"edge").
var(70,"Y").
rulevar(60,70).
alist(69,1,70).
var(71,"X").
rulevar(60,71).
alist(69,2,71).
maxrulectr(60,72).
```

Figure A.3: Constraints and parametrized conjunction

```
% reached(1).
hasrule(1,44).
assert(44).
pos(45).
head(44,45).
rulepred(44,45).
arity(45,1).
pred(45,"reached").
cnst(46,"1").
alist(45,1,46).
maxrulectr(44,47).

% edge(1,2).
hasrule(1,79).
assert(79).
pos(80).
head(79,80).
rulepred(79,80).
arity(80,2).
pred(80,"edge").
cnst(81,"1").
alist(80,1,81).
cnst(82,"2").
alist(80,2,82).
maxrulectr(79,83).
```

```
% edge(2,3).
hasrule(1,84).
assert(84).
pos(85).
head(84,85).
rulepred(84,85).
arity(85,2).
pred(85,"edge").
cnst(86,"2").
alist(85,1,86).
cnst(87,"3").
alist(85,2,87).
maxrulectr(84,88).

% edge(3,1).
hasrule(1,89).
assert(89).
pos(90).
head(89,90).
rulepred(89,90).
arity(90,2).
pred(90,"edge").
cnst(91,"3").
alist(90,1,91).
cnst(92,"1").
alist(90,2,92).
maxrulectr(89,93).
```

```
% edge(1,3).
hasrule(1,94).
assert(94).
pos(95).
head(94,95).
rulepred(94,95).
arity(95,2).
pred(95,"edge").
cnst(96,"1").
alist(95,1,96).
cnst(97,"3").
alist(95,2,97).
maxrulectr(94,98).

% edge(3,2).
hasrule(1,99).
assert(99).
pos(100).
head(99,100).
rulepred(99,100).
arity(100,2).
pred(100,"edge").
cnst(101,"3").
alist(100,1,101).
cnst(102,"2").
alist(100,2,102).
maxrulectr(99,103).
```

Figure A.4: Reified facts

```
% edge(1,3).
hasrule(1,94).
assert(94).
pos(95).
head(94,95).
rulepred(94,95).
arity(95,2).
pred(95,"edge").
cnst(96,"1").
alist(95,1,96).
cnst(97,"3").
alist(95,2,97).
maxrulectr(94,98).

% edge(3,2).
hasrule(1,99).
assert(99).
pos(100).
head(99,100).
rulepred(99,100).
arity(100,2).
pred(100,"edge").
cnst(101,"3").
alist(100,1,101).
cnst(102,"2").
alist(100,2,102).
maxrulectr(99,103).
```

```
% edge(2,1).
hasrule(1,104).
assert(104).
pos(105).
head(104,105).
rulepred(104,105).
arity(105,2).
pred(105,"edge").
cnst(106,"2").
alist(105,1,106).
cnst(107,"1").
alist(105,2,107).
maxrulectr(104,108).
freectr(109).
```

Figure A.5: More reified facts

Appendix B

Description of Reified Atoms

The following tables enumerate the reified atoms which are produced by the reification process and also during the metaevaluation and metagrounding processes. The first table B.1 describes the atoms for representing rules in reified format. The following tables B.2 and B.3 describe the atoms produced by the metaevaluation flows.

hasrule(X, Y)	X represents the set of rules, Y an individual rule
rule(X)	X represents a basic rule with head and multiple bodies
assert(X)	X represents a fact with just a head
pos(X)	X is not negated, this may be omitted as it is assumed to be default
neg(X)	X is negated
arithexpr(X)	X is an arithmetic expression
bexpr(X)	X is a logical binary expression
card(X)	X has a cardinality constraint (optional mincard,maxcard have the constraints)
weigh(X)	X has a weight constraint (optional mincount,maxcount have the constraints
constraint(X)	X represents a constraint with just no head but many bodies
composite(X)	X represents a composite ordered list of predicates (tlist collects the predicates)
head(X, Y)	X represents the rule, Y represents the head, only one head is allowed for a rule
body(X, Y)	X represents the rule, Y represents a body
larg(X, Y)	the left argument of expression X is represented by Y
rarg(X, Y)	the right argument of expression X is represented by Y
qual(X, Y)	Y represents a qualifier for X in a list of constraints
constn(X, Y)	X is a constant assignment where Y is the constant to be assigned to Y (constval is the right hand side)
constval(X, Y)	X is a constant assignment with Y being the value expression (constn is the left hand side)
weight(X, Y)	X represents a component inside a weighed rule, Y represents the weighing valued
mincard(X, Y)	Y represents the minimum cardinality for X
maxcard(X, Y)	Y represents the maximum cardinality for X
mincount(X, Y)	Y represents the minimum count for X
maxcount(X, Y)	Y represents the maximum count for X
pred(X, A)	X represents a predicate with name A, which must be a string (alist has the arguments)
op(X, A)	X represents an arithmetic operator with name A, which must be a string (larg,rarg have the arguments)
bop(X, A)	X represents a logical operator with name A, which must be a string (larg,rarg have the arguments)
alist(X, Y, Z)	represents an argument of predicate X with argument Z at position Y
var(X, A)	X represents a variable with name A, which must be a string
cnst(X, A)	X represents a constant with value A
tlist(X, Y, Z)	represents an argument of composite X with element Z at position Y
altlist(X)	X has a number of alternatives altlist/3 has the elements
altlist(X, Y, Z)	represents an argument of alternative X with entry Z at position Y

Table B.1: The elements of reified rules

apredlst($Fid, Pn, Args$)	Internal representation of a ground fact, or head of a ground rule. Fid is the ID of the fact, Pn is the name of the atom and $Args$ has a recursive v-list containing the index and value (see nthinapredlst).
nthinapredlst($Fid, Pn, N,$ v(N, $Value, Rest$), $List$)	Accessor function for the lists in apredlst, where Fid and Pn are ID of the fact and name of the atom respectively. The third argument is the Nth element of list $List$, represented by function symbol v(N, $Value, Rest$) where N is the index, $Value$ the content of list index at N and $Rest$ the rest of the list, nil or another v-list.
lmap($Rid, Bodydid, Groundid, List$)	Maps a particular body element $Bodyid$ inside rule Rid to an eligible grounded atom $Groundid$. $List$ has the arguments of the grounded atom using same v-list as in apredlst.
xassign($Rid, Bodyid$, Vn, Val, $Groundid, List$)	Contains a mapping of variable named Vn to value Val within rule Rid. The mapping is due to an earlier lmap combining the body of a rule $Bodyid$ with a ground atom $Groundid$ with a fact with arguments in $List$.
visiblexassign(Rid, Vn, Val, $Groundid, List$)	Like xassign, but only for variables which are found in head of rule Rid.
xconflict(Rid, Vn, $Val1$, $Bodyid1$, $Val2$, $Bodyid2, List1, List2$)	Indicates that two xassigns identified by their last arguments $List1$ and $List2$ have different values for variable named Vn. $Bodyid1$ and $Bodyid2$ have the identifiers of the source ground facts.
vlist2($Rid, Pn, List$)	List of variable-value pairs for variables occurring in the head of rule Rid. The name of the predicate in the head is Pn. The list function symbol is z and it contains the variables in the same order as in the head, see nthinvlist2 for details. Values in this list are not conflicting. vlist2 attempts to keep all possible, even redundant, information leading to the variable assignment.
nthinvlist2($Rid, Pn,$ N, Vn, Val, Bid, Gid, $Vlist$, z($N-1$, Vn', Val', Bid', Gid', Vl', R), $List$)	Accessor for vlist2, Rid and Pn as in vlist2, the following 6 arguments are the contents of the list element: N is the index, Vn and Val are variable name and value, Bid is the ID for the body in the rules, Gid identifies the source ground fact and $Vlist$ has the arguments of Gid. The next term has the rest of the z-list after index N, R is the rest of the recursive list. Lastly, $List$ contains the full vlist2 list.

Table B.2: The atoms within the metaevaluation flows

vlist3($Rid, Pn, Alist,$ vv(N, Gid, $Vn, Rest$), $Vlist2$)	Separates a vlist2 to two lists: Alist, containing a list of values in the same v format as in apredlst and provenance vv-list, where N is the index of the list, Gid is the source ground fact ID, Vn is the name of the variable obtaining the value from this ground fact, and the rest of the list is in $Rest$. The last $Vlist2$ contains the source vlist2. Conceptually the two lists contain the actual values and the metainformation for the newly derived fact. The accessor atoms nthinvlist2 can be used to access this list as well.
evaluable(Rid)	Indicates that rule Rid can be grounded fully to a new fact. An apredlst with fact ID Rid which is marked as evaluable triggers the generation of a reified fact using the values of that apredlst.
prnonevaluable(Rid)	Indicates that rule Rid cannot be grounded to a fact, but needs to be grounded as a rule. A rassign for fact ID Rid which is marked as prnonevaluable triggers the generation of a reified rule which also includes apredlst.
flist(Rid, f(N, Bid, $Vlist$, $Restf$), ff($N, Gid, Restff$))	Similar to vlist3, containing two lists: f-list where N is the list index, Bid is the item in the body of rule Rid, $Vlist$ contains the values of the mapped to the ground atom. ff-list contains the source ground atom IDs in Gid. The recursive rest of the lists are in $Restf$ and $Restff$.
nthinflist($Rid, N, Bid\,Vlist,$ f($N-1, Bid'$, $Vlist', Restf'$), $List$)	Accessor for flist, Rid is the rule ID, the following three arguments are the list item, where N is the list index, Bid is the associated item in the rule and $Vlist$ contains the values mapped ground atom. The next f argument contain the previous index in the body and $List$ contains the list we are accessing.
rassign(Rid, Vn, $Val, Groundid, List$)	Contains a mapping of variable named Vn to a value Val within a rule Rid. The mapping is due to an earlier lmap combining a body element with a ground atom $Groundid$ with a fact with arguments in flist, which is in $List$.
cflctrassign($Rid, List$)	Marks a conflict within flist for rule Rid. Occurs when two rassign referring to the same $List$ have a different value for a variable with the same name or when two rassign violate a constraint (e.g., equal, smaller than, ...) within the rule.

Table B.3: More atoms within the metaevaluation flows, especially concerning ground rules.

typedlist$(Rid, \mathsf{f}(N, Bid, Vlist, Restf))$	Similar to flist, however it is restricted to those atoms within rule Rid, which are indicated by tlist belonging to a parametrized conjunction.
nthintypedlist$(Rid, N, Bid, Vlist,$ $\mathsf{f}(N-1, Bid', Vl', Restf), List)$	Accessor to typedlist, however it is restricted to those atoms within rule Rid, which are indicated by tlist belonging to a parametrized conjunction. Rid is the rule ID, the following three arguments are the list item, where N is the list index, Bid is the associated item in the rule and $Vlist$ contains the values of mapped ground atoms. $Restf$ has the rest of the list and $List$ contains the accessed list.
tvarpospre$(Tid, Vn, Predid, Idx)$	Binds a variable named Vn within a conjunction tlist to its atom ID $Predid$ at position Idx within that atom.
tassign$(Tid, Pid, Vn, Val, Vlist)$	Substitution similar to xassign, however it is restricted to those atoms and variables within parametrized conjunction Tid. Pid is the atom ID referring to the variables, Vn and Val are the name and value of the variable respectively and $Vlist$ is the list of arguments from a ground atom.
testassign$(Tid, Vn, Val, Vlist)$	Substitution similar to rassign, combining tassign and typedlist. Tid, Vn and Val like in tassign $Vlist$ are obtained from typedlist.
cflcttestassign$(Tid, Vlist)$	Indicates a conflict between two testassign variable assignments in the same $Vlist$ within parametrized conjunction Tid.
compatibletandx$(Rid, Tid,$ $Rassign, Tassign)$	Indicates that the substitutions in a rule Rid produced by rassign are compatible with the assignments $Tassign$ within a parametrized conjunction Tid.
subprovenance$(Fid, Rid, Groundid)$	Provenance for the fact with ID Fid is defined by $Groundid$, which may be a ff-list as in flist. The Rid gives the rule which produced this result.
isubprovenance$(Fid, Rid, Groundid)$	An internal intermediate version of subprovenance with the same terms. This atom is filtered by provenance metarules.

Table B.4: Atoms within the metaevaluation flow for handling parametrized conjunction

Bibliography

[1] *Web Reasoning and Rule Systems - 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013. Proceedings*, volume 7994 of *Lecture Notes in Computer Science*. Springer-Verlag, 2013.

[2] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press Series In Artificial Intelligence, 1990.

[3] T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, and D. Srivastava. Explaining program execution in deductive systems. In *Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science*, pages 101–119. Springer Berlin Heidelberg, 1993.

[4] K. Ashton. That 'internet of things' thing. *RFID Journal*, July 2009.

[5] R. A. Aziz, T. Janhunen, and V. Luukkala. Distributed deadlock handling for resource allocation in smart spaces. In *Proceedings of ruSMART/NEW2AN'11*, volume 6869 of *Lecture Notes in Computer Science*, pages 87–98. Springer-Verlag, 2011.

[6] R.A. Aziz. Distributed Rule-Based Resource Allocation in Smart Spaces. Master's thesis, Helsinki University of Technology, Aug 2011.

[7] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19:73–148, 1994.

[8] S. Baselice and P.A. Bonatti. A decidable subclass of finitary programs. *Theory and Practice of Logic Programming*, 10(4-6):481–496, 2010.

[9] M.Y. Becker, C. Fournet, and A.D. Gordon. Design and semantics of a decentralized authorization language. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 3–15, 2007.

[10] J. Ben-Zvi. *The Time Relational Model*. University Microfilms, 1983.

[11] S. Berkovits. How To Broadcast A Secret. In *Advances in Cryptology — EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 535–541. Springer Berlin Heidelberg, May 1991.

[12] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986, uniform resource identifier (URI): Generic syntax. http://rfc.net/rfc3986.html, 2005.

[13] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[14] M. Blaze, J. Feigenbaum, J. Ioannidis, and A.D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer Berlin Heidelberg, 1999.

[15] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[16] S. Boldyrev, I. Oliver, R. Brown, J-M. Tuupola, A. Palin, and A. Lappeteläinen. Network and content aware information management. In *Proceedings of the 4th International Conference for Internet Technology and Secured Transactions, 2009*, pages 1–7. IEEE, 2009.

[17] H. Boley, M. Kifer, P-L. Pătrânjan, and A. Polleres. Rule interchange on the web. In *Reasoning Web*, volume 4636 of *Lecture Notes in Computer Science*, pages 269–309. Springer Berlin Heidelberg, 2007.

[18] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In *Logic Programming*, pages 153–172. Academic Press, 1982.

[19] C. Boyd. Some applications of multiple key ciphers. In *EUROCRYPT*, volume 330 of *Lecture Notes in Computer Science*, pages 455–467. Springer, 1988.

[20] R. T. Braden. RFC 1122: Requirements for Internet hosts — communication layers. ftp://ftp.internic.net/rfc/rfc1122.txt, October 1989.

[21] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.1 (second edition). `http://www.w3.org/TR/2006/REC-xml-names11-20060816`, 2006.

[22] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, pages 316–330, London, UK, UK, 2001. Springer-Verlag.

[23] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP: theory and implementation. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424. Springer-Verlag, 2008.

[24] M. Carlsson and P. Mildner. SICStus Prolog - the first 25 years. *Theory and Practice of Logic Programming*, 12:35–66, 2012.

[25] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[26] D. D. Corkill. Blackboard Systems. *AI Expert*, 6(9), January 1991.

[27] D. D. Corkill. Collaborating Software: Blackboard and Multi-Agent Systems & the Future. In *Proceedings of the International Lisp Conference*, New York, October 2003.

[28] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, June 2000.

[29] S. Decker, S. Handschuh, and M. Hauswirth. Towards networked knowledge. In *Foundations for the Web of Information and Services*, pages 155–174. Springer, 2011.

[30] J. P. Delgrande, T. Schaub, and H. Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, 2003.

[31] F-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[32] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 105–, Washington, DC, USA, 2002. IEEE Computer Society.

[33] T. Eiter, W. Faber, M. Fink, and S. Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007.

[34] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory and Practice of Logic Programming*, 3(4-5):463–498, 2003.

[35] T. Eiter, G. Ianni, T. Krennwallner, and A. Polleres. Reasoning web. chapter Rules and Ontologies for the Semantic Web, pages 1–53. Springer-Verlag, 2008.

[36] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05*, pages 90–96. Professional Book, 2005.

[37] T. Eiter and A. Polleres. Towards automated integration of guess and check programs in answer set programming: A meta-interpreter and applications. *Computing Research Repository*, abs/cs/0501084, 2005.

[38] O. El-Khatib, E. Pontelli, and T. C. Son. Integrating an answer set solver into Prolog: ASP-PROLOG. In *LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, pages 399–404. Springer, 2005.

[39] T. Erl. *SOA Principles of Service Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[40] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy. The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.

[41] J. Farrell and G. Saloner. Standardization, compatibility, and innovation. *RAND Journal of Economics*, 16(1):70–83, Spring 1985.

[42] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[43] K. Främling, I. Oliver, J. Honkola, and J. Nyman. Smart spaces for ubiquitously smart buildings. In *Proceedings of the 2009 Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, UBICOMM '09, pages 295–300, Washington, DC, USA, 2009. IEEE Computer Society.

[44] H. Gallaire and J. Minker, editors. *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977*, Advances in Data Base Theory. Plemum Press, 1978.

[45] M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4-5):821–839, 2011.

[46] M. Gebser, B. Kaufmann, A Neumann, and T. Schaub. *clasp* : A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[47] M. Gebser, J. Pührer, T. Schaub, and H. Tompits. A meta-programming technique for debugging answer-set programs. In *Proceedings of AAAI 2008*, pages 448–453. AAAI Press, 2008.

[48] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 266–271. Springer-Verlag, 2007.

[49] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[50] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium in Logic Programming*, pages 1070–1080. MIT Press, 1988.

[51] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[52] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM.

[53] T.J. Green. Containment of conjunctive queries on annotated relations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 296–309, New York, NY, USA, 2009. ACM.

[54] Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 149–162, Washington, DC, USA, 2008. IEEE Computer Society.

[55] R.J. Hall. Open modeling in multi-stakeholder distributed systems: Research and tool challenges. In *Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002.

[56] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, August 1985.

[57] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[58] J. Honkola, H. Laine, R. Brown, and I. Oliver. Cross-domain interoperability: A case study. In *Proceedings of ruSMART/NEW2AN*, Lecture Notes in Computer Science, pages 22–31. Springer, 2009.

[59] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3C member submission, World Wide Web Consortium, 2004.

[60] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM.

[61] T. Janhunen and V. Luukkala. Meta programming with answer sets for smart spaces. In *Web Reasoning and Rule Systems - 6th International Conference, RR 2012*, volume 7497 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2012.

[62] T. Janhunen, I. Niemelä, J. Oetsch, J. Pührer, and H. Tompits. On testing answer-set programs. In *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 951–956. IOS Press, 2010.

[63] B. Joy. The six webs. `http://video.mit.edu/watch/the-six-webs-10-years-on-9110/`.

[64] R.M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[65] A. Katasonov and M. Palviainen. Towards ontology-driven development of applications for smart environments. In *Eigth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 696–701. IEEE, 2010.

[66] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series).* Chapman & Hall/CRC, 2007.

[67] J. Korhonen. *Introduction to 3G Mobile Communications.* Artech House mobile communications series. Artech House, 2003.

[68] D. G. Korzun, I. V. Galov, A. Kashevnik, N. Shilov, K. Krinkin, and Y. Korolev. Integration of smart-m3 applications: Blogging in smart conference. In *Proceedings of ruSMART/NEW2AN'11*, volume 6869 of *Lecture Notes in Computer Science*, pages 51–62. Springer-Verlag, 2011.

[69] R.A. Kowalski. Predicate logic as programming language. In *Proceedings of the IFIP Congress*, volume 74, pages 569–574, 1974.

[70] P. Leach, M. Mealling, and R. Salz. RFC 4122: A universally unique identifier (UUID) URN namespace. http://www.ietf.org/rfc/rfc4122.txt, 2005.

[71] G.A. Lewis, E.J. Morris, S. Simanta, and L. Wrage. Why standards are not enough to guarantee end-to-end interoperability. In *Proceedings of ICCBSS 2008*, pages 164–173. IEEE, 2008.

[72] N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 27–, Washington, DC, USA, 2000. IEEE Computer Society.

[73] Y. Lierler and V. Lifschitz. One more decidable class of finitely ground programs. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 489–493, Berlin, Heidelberg, 2009.

[74] V. Lifschitz. Answer set planning. In *Proceedings of the 16th International Conference on Logic Programming*, pages 25–37. The MIT Press, December 1999.

[75] J.W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.

[76] N. Lopes, S. Kirrane, A. Zimmermann, A. Polleres, and A. Mileo. A logic programming approach for access control over RDF. In A. Dovier and V.S. Costa, editors, *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 381–392. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[77] V. Luukkala, D.-J. Binnema, M. Borzsei, A. Corongiu, and P. Hyttinen. Experiences in implementing a cross-domain use case by combining semantic and service level platforms. In *Proceedings of the The IEEE symposium on Computers and Communications*, ISCC '10, pages 1071–1076, 2010.

[78] V. Luukkala and I. Niemelä. Enhancing a smart space with answer set programming. In *RuleML*, volume 6403 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2010.

[79] Brain M., Gebser M., Pührer J., Schaub T., Tompits H., and Woltran S. Debugging ASP programs by means of ASP. In *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2007.

[80] S. Mallet and M. Ducassé. Generating deductive database explanations. In *Proceedings of the International Conference on Logic Programming*, pages 154–168. MIT Press, 1999.

[81] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[82] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[83] A. Nerode and R.A. Shore. *Logic for applications (2. ed.)*. Graduate texts in computer science. Springer, 1997.

[84] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[85] G. Niezen, B. J. J. van der Vlist, J. Hu, and L. M. G. Feijs. From events to goals: Supporting semantic interaction in smart environments. In *Proceedings of the 15th IEEE Symposium on Computers and Communications, ISCC 2010*, pages 1029–1034. IEEE, 2010.

[86] L. J. B. Nixon, E. Simperl, R. Krummenacher, and F. Martin-Recuerda. Tuplespace-based computing for the semantic web: A survey of the state-of-the-art. *The Knowledge Engineering Review*, 23(2):181–212, June 2008.

[87] A.M. Ouksel and A.P. Sheth. Semantic interoperability in global information systems: A brief introduction to the research area and the special section. *SIGMOD Record **28**(1)*, pages 5–12, 1999.

[88] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems, 3rd Edition*. Springer, 3rd edition, 2011.

[89] S. Pantsar-Syväniemi, E. Ovaska, S. Ferrari, T. S. Salmon Cinotti, G. Zamagni, L. Roffia, S. Mattarozzi, and V. Nannini. Case study: Context-aware supervision of a smart maintenance process. In *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*, pages 309–314. IEEE, 2012.

[90] E. Pontelli, Son T. C., and O. El-Khatib. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming*, 9(1):1–56, 2009.

[91] E. Pontelli, T. C. Son, and N-H. Nguyen. Combining answer set programming and prolog: the ASP-PROLOG system. In *Logic programming, knowledge representation, and nonmonotonic reasoning*, Lecture Notes in Computer Science, pages 452–472. Springer-Verlag, 2011.

[92] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[93] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[94] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[95] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. 23(2):442–453, May 1994.

[96] B. Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[97] David G. Schwartz and Leon S. Sterling. Using a prolog meta-programming approach for a blackboard application. *SIGAPP Applied Computing Review*, 1(1):26–34, January 1993.

[98] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[99] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[100] B. C. Smith. Reflection and semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 23–35. ACM, 1984.

[101] B.C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, 1982.

[102] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[103] W3C recommendation: SPARQL query language for RDF. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115`.

[104] G. Specht. Generating explanation trees even for negations in deductive database systems. In *Proceedings of the 5th Workshop on Logic Programming Environments*, pages 8–13. IRISA, 1993.

[105] Springer. *Hazewinkel Encyclopedia of Mathematics*. Springer, 1995.

[106] L. Sterling and E. Shapiro. *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994.

[107] T. Syrjänen. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`, 2000.

[108] T. Syrjänen. Omega-restricted logic programs. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference*, volume 2173 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 2001.

[109] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[110] C. Villalonga, M. Strohbach, N. Snoeck, M. Sutterer, M. Belaunde, E. Kovacs, A. V. Zhdanova, L. W. Goix, and O. Droegehorn. Mobile ontology: Towards a standardized semantic model for the mobile domain. In *ICSOC Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 248–257. Springer, 2007.

[111] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, 1991.

[112] C. Wieland. Two explanation facilities for the deductive database management system dedex. In *Proceedings of the 9th International Conference on Entity-Relationship Approach (ER'90)*, pages 189–203. ER Institute, 1990.

[113] J. Wielemaker, M. Hildebrand, and J. van Ossenbruggen. Using Prolog as the fundament for applications on the semantic web. In *Proceedings of the 2nd Workshop on Applications of Logic Programming and to the web, Semantic Web and Semantic Web Services*, volume 287 of *CEUR Workshop Proceedings*, pages 84–98, 2007.

[114] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[115] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia. A general framework for representing, reasoning and querying with annotated semantic web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 11:72–95, March 2012.

[116] H. Zimmermann. OSI reference model–the ISO model of architecture for open systems interconnection. In *IEEE Transactions on Communications*, volume 28, pages 425–432. IEEE, 1980.