THE UNIVERSITY *of* EDINBURGH

Meta-ontology fault detection

Juan Casanova Jaquete

Doctor of Philosophy

Artificial Intelligence and its Applications Institute

School of Informatics

University of Edinburgh

2022

Abstract

Ontology engineering is the field, within knowledge representation, concerned with using logic-based formalisms to represent knowledge, typically moderately sized knowledge bases called *ontologies*. How to best develop, use and maintain these ontologies has produced relatively large bodies of both formal, theoretical and methodological research.

One subfield of ontology engineering is *ontology debugging*, and is concerned with preventing, detecting and repairing errors (or more generally pitfalls, bad practices or *faults*) in ontologies. Due to the logical nature of ontologies and, in particular, entailment, these faults are often both hard to prevent and detect and have far reaching consequences. This makes ontology debugging one of the principal challenges to more widespread adoption of ontologies in applications. Moreover, another important subfield in ontology engineering is that of *ontology alignment*: combining multiple ontologies to produce more powerful results than the simple sum of the parts. Ontology alignment further increases the issues, difficulties and challenges of ontology debugging by introducing, propagating and exacerbating faults in ontologies.

A relevant aspect of the field of ontology debugging is that, due to the challenges and difficulties, research within it is usually notably constrained in its scope, focusing on particular aspects of the problem or on the application to only certain subdomains or under specific methodologies. Similarly, the approaches are often ad hoc and only related to other approaches at a conceptual level. There are no well established and widely used formalisms, definitions or benchmarks that form a *foundation* of the field of ontology debugging.

In this thesis, I tackle the problem of ontology debugging from a more abstract than usual point of view, looking at existing literature in the field and attempting to extract common ideas and specially focussing on formulating them in a common language and under a common approach. *Meta-ontology fault detection* is a framework for detecting faults in ontologies that utilizes semantic *fault patterns* to express schematic entailments that typically indicate faults in a systematic way. The formalism that I developed to represent these patterns is called *existential second-order query logic* (abbreviated as ESQ logic). I further reformulated a large proportion of the ideas present in some of the existing research pieces into this framework and as patterns in ESQ logic, providing a *pattern catalogue*.

Most of the work during my PhD has been spent in designing and implementing an algorithm to effectively automatically detect arbitrary ESQ patterns in arbitrary ontologies. The result is what we call *minimal commitment resolution for ESQ logic*, an extension of first-order resolution, drawing on important ideas from higher-order unification and implementing a novel approach to unification problems using *dependency graphs*. I have proven important theoretical properties about this algorithm such as its soundness, its termination (in a certain sense and under certain conditions) and its fairness or completeness in the enumeration of infinite spaces of solutions.

Moreover, I have produced an implementation of minimal commitment resolution for ESQ logic in Haskell that has passed all unit tests and produces non-trivial results on small examples. However, attempts to apply this algorithm to examples of a more realistic size have proven unsuccessful, with computation times that exceed our tolerance levels.

In this thesis, I have provided both details of the challenges faced in this regard, as well as other successful forms of qualitative evaluation of the meta-ontology fault detection approach, and discussions about both what I believe are the main causes of the computational feasibility problems, ideas on how to overcome them, and also ideas on other directions of future work that could use the results in the thesis to contribute to the production of foundational formalisms, ideas and approaches to ontology debugging that can properly combine existing constrained research. It is unclear to me whether minimal commitment resolution for ESQ logic can, in its current shape, be implemented efficiently or not, but I believe that, at the very least, the theoretical and conceptual underpinnings that I have presented in this thesis will be useful to produce more foundational results in the field.

Acknowledgements

It would be hard to acknowledge anyone without first acknowledging my supervisors: Alan Bundy and Perdita Stevens. They have not only provided me huge amounts of research methodology advice, motivated discussions and directions of research and introduced me to important fields and persons in those fields; but also they have regularly acted as counsellors, played the good-cop-bad-cop routine with me very effectively, given me ample and supportive freedom to choose my focus and perhaps tolerated a bit more from me than I should have expected them to.

Next to my supervisors, the list of people I have collaborated with during my PhD is very long, but I would like to identify a few specific individuals who have proved key in one way or another to my research.

The first of these must be Xue Li, whose PhD happened more or less at the same time as mine, and with whom I shared more than a lot of topics, ideas, discussions and projects. Similarly, Christian Kindermann has provided interesting discussions, a bridge to certain fields that would have been harder for me to reach without him, and very relevant and useful literature pointers and discussions. I am sure I will be forgetting some, but other similar collaborators that I would like to mention by name include Jacques Fleuriot, Kwabena Nuamah, Alan Smaill, Francisco Quesada, Marius Urbonas, Paul Jackson, Iain Murray and Daniel Raggi.

I would like to mention the particularly convenient and nice environment provided by the School of Informatics at the University of Edinburgh. The Informatics Forum truly is one of the best buildings to work in and it is unfortunately uncommon that the physical context of a long project does not in itself introduce its own difficulties to the project, and in this case it hasn't.

Similarly, I would like to acknowlege the financial and institutional support of the EPSRC CDT in Data Science, which this PhD has been a part of.

Other than that, the number of people who have been relevantly supportive or helpful in general throughout my life during the PhD is too large to explicitly mention here, but clearly the mere fact that I have arrived to the end of this PhD while still being sane is a testament to their value.

A last mention will go to Heriot-Watt University for hiring me as a full time assistant professor during my last year of PhD, and for overall being supportive of my finishing the PhD, by giving me space, time and trying to avoid overloading me while still giving me a job that I enjoy and hope to continue to enjoy in the future.

Thank you to all, and I sincerely hope I was not too much to bear with too often.

PS: I will obviously **not** thank the wonderful Covid-19 virus for making the last year of my PhD that little bit more stressful and anxiety inducing. If there ever was a reason to add negative acknowledgements to a thesis, this was it.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Juan Casanova Jaquete*)

Table of Contents

Chapter 1

Introduction

In the context of the semantic web and linked data, *ontologies* [Guarino et al., 2009] are knowledge bases expressed using logic-based formalisms, such as description logics [Baader et al., 2009], which allow for rich semantics encoding complex entailments that enable more advanced inference mechanisms than simple databases. These usually involve some form of automated theorem proving. For example, by explicitly representing in a logic formalism that:

- Birds fly.
- Eagles are birds.

we can infer that *eagles fly*.

One of the most common problems when utilizing ontologies is the effect of errors or misrepresentations in the ontologies on the inferences done with them. We use the word *fault* to encompass not only direct *errors* (inconsistencies, representations that do not accurately depict reality, etc.) but also other problematic situations such as inadequate representations, bad design patterns, etc. In short, anything we want to avoid in an ontology. Due to the expressive capabilities of logic formalisms, it is moderately common for even small faults to cascade into large issues with inference.

Building on the previous example, not all birds actually fly. For example, penguins do not fly. This may lead us to infer that penguins are not birds. This could escalate even further, depending on the specific ontology, to infer things like penguins being reptiles, penguins having scales, and so on; all because of a small misrepresentation (that all birds fly).

There has been a moderate body of work trying to deal with this and similar issues in different ways. Some approaches involve limiting expressivity and reach of inference to contain the effect of faults, while others may wish to utilize provenance or related notions to be able to identify sources of problems. Our work in this thesis focuses on finding automated or semi-automated ways to identify these faults that may help ontology designers limit their frequency and, perhaps more importantly, more quickly and easily debug ontologies as they evolve. This is sometimes referred to as *ontology debugging*.

While it is not central to our work, it is relevant to understand the usual sources of faults in ontologies. Most research on the topic [Rector et al., 2004, Blomqvist, 2010, Kindermann et al., 2019, Poveda-Villalón et al., 2010, Poveda-Villalón et al., 2012, Guarino and Welty, 2009, Copeland et al., 2013, Gkaniatsou et al., 2012, Haverty, 2013, Markakis, 2013, Lambrix and Liu, 2013, Prince Sales and Guizzardi, 2017, Mikroyannidi et al., 2011, Mikroyannidi et al., 2012] points to *human error* and different ontologies that work together using different *background incompatible conceptualizations*, because of their different scope, assumptions or simplifications. For example, in a simple educational ontology it may make sense to work with the simplification that all birds fly, whereas this assumption is problematic in a more serious taxonomy ontology. Another important area of research in the field is the *automatic generation of ontologies*, often by combining or *aligning* preexisting ones [Lambrix and Liu, 2013, Šváb-Zamazal and Svátek, 2008]. *Automatically generated ontologies* are also very prone to faults [Gkaniatsou et al., 2012, Haverty, 2013, Markakis, 2013].

Meta-ontology fault detection is a framework for automated detection of faults in ontologies that we have developed. Its core principle is to use automated theorem proving techniques, combined with formally encoded *patterns* to identify potentially faulty situations in ontologies. We sometimes refer to these patterns as meta-knowledge, and one of the strengths of the framework is that the technique is independent of the specific patterns. While we do identify some patterns which are likely to be useful across the board, we also identify others that are designed to be applicable only to specific kinds of ontologies. Similarly, all of the patterns we describe in this thesis are based on situations identified by previously existing research, that we have **interpreted and encoded within our framework**. We believe this showcases the strength of the

approach, and we consider the gathering and reformulation of this *pattern catalogue* to be one of the three main contributions of this work.

Clearly, another of the main contributions of this work is the meta-ontology fault detection framework itself. This is embodied by one of the two research hypotheses that we evaluated:

Hypothesis 1. *Meta-ontology fault detection, used by encoding patterns in* existential second-order query logic*, has the potential to be an effective and feasible approach to detecting common faults in ontologies formalized in first-order logic.*

This hypothesis focuses on the pragmatic value of the framework, assuming the technology for implementing it exists and works adequately. But of course, the development of this technology is another relevant contribution of this thesis. In particular, we developed a formalism and an algorithm using novel automated theorem proving techniques to efficiently find instances of fault patterns present in ontologies. We call this *minimal commitment resolution for existential second-order query logic*, or *minimal commitment resolution for ESQ logic*.

This algorithm takes patterns, encoded as *queries* containing free second-order variables (representing the ontology elements that cause the faults), and an ontology presented in first-order logic, and uses a *minimal commitment approach* to produce proofs associated with instantiations of the patterns corresponding to likely faults. We evaluate the success of minimal commitment resolution for ESQ logic via our second research hypothesis:

Hypothesis 2. minimal commitment resolution for existential second-order query logic *is a sound, complete and computationally feasible implementation of existential second-order query logic.*

Thus, the three primary contributions of this thesis are:

1. The development of the *meta-ontology fault detection framework*, and the evaluation of its effectiveness and feasibility.

2. The development of the *minimal commitment resolution for ESQ logic algorithm*, and the evaluation of its soundness, completeness and computational feasibility.

3. The recollection and reformulation of common ontology patterns from the literature into a ESQ logic *pattern catalogue*.

Unfortunately, I ran into significant computational feasibility issues with the usage of the algorithm on more realistically sized examples, meaning that the practical feasibility of this approach as a whole in its current state is unclear. Nonetheless, the framework principles, theoretical results and the pattern catalogue remain as the valuable contributions of this thesis independently of the algorithm's computational properties; and there are still ideas to be tried to attempt to overcome the computational performance issues. We discuss these issues, their causes, and potential future work mainly in chapter 9, but also in chapters 8 and 10.

The remainder of the thesis is structured as follows:

- In chapter 2 we review the existing literature on topics related to this thesis, such as automated theorem proving, ontology engineering, ontology debugging and other related topics in artificial intelligence.

- In chapter 3 we provide technical background on the standard topics that are important for the understanding of this thesis.

- Chapter 4 describes the meta-ontology fault detection framework and introduces the pattern catalogue, one of the main contributions of this work.

- In chapter 5 we discuss the formalism, algorithm and other technical tools that we developed to implement the meta-ontology fault detection framework: minimal commitment resolution for existential second-order query logic.

- Chapter 6 contains core theoretical results about the outer layers of the minimal commitment resolution for ESQ logic algorithm. Namely, the definition of ESQ logic and the resolution aspects.

- Chapter 7 contains core theoretical results about the inner layers of the minimal commitment resolution for ESQ logic algorithm. Namely, dependency graph unification, which is the main theoretical contribution of the thesis. Among other things, it includes correctness (soundness and conditional completeness) theorems.

- In chapter 8 we describe our implementation of both the technical algorithm and the framework to detect faults.

- Chapter 9 discusses the evaluation of the two research hypotheses, and analyzes the results.
- Finally, we produce some concluding remarks, compare this work with other similar research and discuss ideas for future work in chapter 10.

Chapter 2

Literature review

The initial problem that motivates this PhD is an ontology engineering (§2.1) one: automatically detecting and repairing faults in ontologies (§2.1.1), a subfield sometimes referred to as *ontology debugging* or *ontology evaluation*. The problem of automatically detecting and repairing faults (errors, bugs, mistakes, problems...) is, however, present in other fields, and we take a look at them: belief revision (§2.2.1) considers updating knowledge of an agent when new information is obtained. Other logic based approaches (§2.2.3) and experience in the software engineering field (§2.2.4) are also relevant.

Both because of the usual usage of ontologies for reasoning and for the approach to fault detection and repair that we have taken, topics in automated theorem proving (§2.3) are also highly relevant. We extensively consider foundational and theoretical aspects of logic in general (2.3.1), modern work in optimized algorithms built on these foundations (2.3.2) and description logics (2.3.4) as a theoretical basis for OWL.

2.1 Knowledge management and ontology engineering

A general and informal description of an *ontology* is that it is an "explicit specification of a conceptualization" [Guarino et al., 2009]. Practically speaking, ontologies are usually specified using languages that stem in one way or another from logic. The most straightforward case is when this language is just first-order logic. However, due both to the computational properties of first-order logic (undecidability, bad complexity for reasoning) and some usual aspects of ontologies (*classes* or *concepts*) not being an explicit part of first-order logic, it is more common to use less expressive logics with more attractive computational properties and explicit formalizations

of concepts. A big family of such restrictions are known as *Description Logics* [Brachman and Schmolze, 1989, Baader et al., 2009], among which the Web Ontology Language (OWL) [Motik et al., 2012] is a predominant one, specially in the Semantic Web and Linked Data context. It should be noted that OWL is also a particular case of RDF [Lassila and Swick, 1999, Brickely and Guha, 2014], a simple formalism for expressing *triples* connecting pairs of elements via *properties* (also called relations).

Other approaches to ontologies exist, among which we can find Frame Logic [Angele et al., 2009] or Formal Concept Analysis [Stumme, 2009], but these have not received as much attention or research effort, and thus we shall not focus on them. However, some of the ideas developed in the PhD might still apply to them, as the notion of using formalized meta-level patterns to detect faults via meta-reasoning is transferable. This applicability is, though, something that we have not explored in detail.

Within the description logics and OWL context, plenty of literature, engineering methods and tools have been developed. There are several well developed IDEs for development of ontologies in OWL, among which Protégé[1] seems to be the currently dominant one. Protégé comes with plenty of inbuilt abilities (automated reasoners, refactoring methods, metrics) as well as a healthy ecosystem of plugins and tools that can be used in conjunction with it. A good survey of the ecosystem can be found in [Kurian et al., 2013]. Methodologies have also been thoroughly studied for the development of ontologies in an industrial setting [Sure et al., 2009, Pinto et al., 2009].

In the context of the work carried out in my MSc [Casanova, 2017], the idea of utilizing anti-patterns to automatically detect faults in ontologies was explored. While perhaps not generally aimed towards fault detection, the concept of ontology patterns and anti-patterns has been long and thoroughly discussed in the literature. For example, in [Hammar and Presutti, 2017], notions of template-based or specialization-based instantiation of patterns in ontologies are discussed. These ideas also seem relevant in the context of pattern detection, representing (in my opinion) the usage of meta-level or object-level methods for fault detection. Similar discussions are carried out in [Blomqvist, 2010]. An analysis of the actual usage in practice (in medical ontologies) of ontology design patterns as typically presented in the literature was performed in [Kindermann et al., 2019]. This analysis has largely negative results, showing that the

[1] `https://protege.stanford.edu/`

vast majority of these ontologies do not incorporate these design patterns at all.

2.1.1 Ontology debugging

However, most of this work is purely *methodological*: there is tool support but there are few automated and AI-based methods for dealing with ontology development and specifically with ontology debugging. More precisely, they are typically tools to help humans find existing faults in ontologies or avoid introducing them, with little to no automatic support in its semantic aspect. For example, in [Rector et al., 2004] a taxonomy of common errors made by learners when authoring ontologies is presented, and some general high level ideas are explained as to how to find these situations. In a similar way, in [Poveda-Villalón et al., 2010] a classification of what are called *pitfalls* in ontology engineering is described, arising from a similar experience: teaching learners to develop ontologies.

OntoClean [Guarino and Welty, 2009] is a more systematic and formal methodology for finding faults in ontologies, but is still meant for a human to apply manually. OntoClean presents an interesting approach in which *abstract* and *philosophical* notions are used as a means to express general ideas that can be used to find faults. We embraced this general idea in the work in [Casanova, 2017] (from which this PhD departs), but from a "dual" point of view: instead of manually annotating the ontology and using the annotations to detect faults, we use pre-established and generic anti-patterns that are then matched against the whole ontology to flag up possible faults. The work in this thesis starts off from that overall idea.

Some more automated work exists. For example, OOPS [Poveda-Villalón et al., 2012] is an automated online system for detecting the pitfalls described in [Poveda-Villalón et al., 2010] in ontologies. This system is, however, specific to those pitfalls and it detects them programatically with largely ad-hoc procedures. Moreover, probably due to the intention of such work being more related to *ontology evaluation* from an engineering point of view than to *fault detection* from a semantic point of view, it is not accompanied by any theoretical background or semantic procedures for understanding the nature of the faults. In a similar fashion, [Copeland et al., 2013] uses versioning information of the ontology as an indicator of potential faults. In particular, they consider incoherent versioning actions such as adding

and removing an element from the ontology in subsequent versions to be indicators of faults. In [Gkaniatsou et al., 2012, Haverty, 2013, Markakis, 2013], work was carried in relation to *detecting* and *repairing* faults in the KnowItAll ontology: an ontology created from general knowledge found in the world wide web, plagued with faults. In that work, specific types of faults (for example, incorrect association of countries to their capital cities) were looked into and specific procedures were developed to find and repair them, with moderate success. Similarly, in [Lambrix and Liu, 2013], a specific approach and implementation to finding and repairing "is-a" hierarchical structures in ontologies is offered, with good theoretical underpinnings and experimental evaluation. [Prince Sales and Guizzardi, 2017] offers another specific approach for finding mereological errors in ontologies defined using the Unified Foundational Ontology.

Most of this work is *specific* and/or uses *ad-hoc* implementations. In contrast, our approach intends to be more *generic*, providing a *framework* for fault/pattern detection, while still being *practically feasible*.

A more general approach is followed in [Mikroyannidi et al., 2011, Mikroyannidi et al., 2012], where syntactic regularity is used to cluster parts of ontologies and signal areas where regularity is not present and it should be, which they understand as a symptom of an underlying problem. This approach is very distant from ours in that it is a machine learning / statistical approach, and it focuses more on the syntax of the ontology than on its semantics. The work provides, however, interesting considerations about similarity / equivalence of axioms that are relatable to our work.

A relevant concept in the context of fault detection and repair is that of an *explanation* of a proof, entailment or, more generally, an inference. Many approaches to ontology debugging rely on finding useful forms of explanation. A formal explanation of a faulty inference can guide a system to identify the source of the fault and/or how to repair it. In this context, interesting work [Horridge, 2011] considers the formalized notion of a *justification* in the context of description logics, and how it can be practically used as a foundation for explanation in ontologies of this kind and all tasks associated with it. It provides relevant algorithms for finding and manipulating justifications, and experimental results surrounding those algorithms.

2.2 Other topics in knowledge revision

2.2.1 Belief revision

Belief revision [Alchourrón et al., 1985, Gärdenfors, 1992, Gärdenfors, 2003] is a term used to refer to the study of the process of updating a *logical knowledge base* when new information is acquired. This has mostly been used in the context of developing agent systems that need to have a model of the reality within which they are operating, and where this model needs to be continuously updated as new information is obtained or the environment itself changes. However, it seems evidently related to the problem of fault detection and repair in logical ontologies: it deals with incorrect representation in a logical theory.

Belief revision works from the assumption that a knowledge base exists, containing a set of statements, usually expressed in some form of logic, and with the implicit assumption that the knowledge base is *logically closed* (i.e. all logical consequences of statements in the knowledge base must also be part of it). Its approach can be summarized as the development of strategies for updating the knowledge base that maintain a series of desired formal properties, such as *monotonicity* in the changes or *minimal change*.

There are several differences between these two research areas, the main one being that belief revision is normally specifically concerned with the revision of a local knowledge base in the event of finding new information, whereas ontology engineering is more focused on using ontologies as a (correct) model of a broader and less dynamic reality. While the semantics of these two processes are not largely different, the aspects of the problem that each of them is most concerned with, as well as the situations that typically arise in them, are indeed relevantly different. Another notable difference is that knowledge bases considered in belief revision are usually considerably smaller than ontologies in, for example, OWL. These differences are mostly contingent and of pragmatic nature, meaning that the techniques appearing in each of them can in principle be applied in the other, but their effectiveness and applicability may meander.

2.2.2 Abduction

Abduction was originally introduced by philosopher Charles Sanders Peirce during the nineteenth century [Peirce, 1901, Peirce, 1906]. A slightly more modern account of its usage in artificial intelligence can be found in [Cox and Pietrzykowski, 1986]. A much more recent description of this and related topics can be found in [Kowalski, 2014].

Generally speaking, abduction is the process of finding the most satisfactory *explanation* for a series of observations, where most satisfactory typically means that it is as simple as possible and alters our perception of reality as little as possible. It can be seen as a complement to belief revision in that belief revision typically works in making inconsistent theories/ontologies consistent again with simplicity and minimal change principles, while abduction typically works in making incomplete theories/ontologies observationally complete with simplicity and minimal change principles.

In artificial intelligence and mathematical logic, abduction usually considers possible ways in which a theory or ontology can be extended to produce a wanted theorem, with options ranging from the most straightforward (adding the wanted theorem as an axiom) to less obvious ones (such as weakening preconditions on existing axioms).

2.2.3 Conceptual change and other logic based approaches

Reformation [Bundy and Mitrovic, 2016] is an algorithm for repairing previously detected faults in first-order theories exploiting the properties of first-order unification. Reformation offers a well-founded and technically unambiguous (i.e. mathematically formulated) method for performing repairs to ontologies already considered faulty. More importantly, it differs from other techniques (such as belief revision §2.2.1) in that it repairs faults by changing the *representation* (i.e. the language) rather than the *statements* (axioms, content...). That is, the repairs performed by reformation change the language of the ontology, by, for example, adding or removing arguments from functions and predicates, renaming them, etc. In [Li et al., 2018], work on several methods of knowledge revision (conceptual change, belief revision and abduction) are combined in a single system that provides repairs suggested by each of them, enabling the usage of heuristics, repair combination methods, etc., homogeneously.

The detection method usually associated with reformation is, however, simplistic: it relies on existing *true* and *false* sets of observations and a choice of an unsuc-

cessful proof attempt or a successful proof of an unwanted theorem, meaning not only that these sets of observations need to be available, but also that the proof attempts of each of these individual faults have to be carried out. Also, in its general form, reformation suggests a vast space of possible repairs for each fault. Work in [Urbonas, 2019, Urbonas et al., 2020] explores, with initial successful results, heuristic approaches for reducing this search space. Work presented in this thesis could be used in combination with reformation, where meta-ontology fault detection offers a local detection mechanism that does not rely on full observation of reality, and reformation tackles the potential faults detected by it to suggest repairs to them.

2.2.4 Systematic and automatic approaches to software development

In software engineering, similar issues to that of faults in ontologies appear in the form of bugs in programs. The similarities are many: both problems relate to specifications made by humans in formal languages about conceptual models.

One of the most prominent ways in which the topic of avoiding problems in software through design is that of *design patterns*. The topic of software design patterns is huge and it would be impossible to summarize it all here. [Bafandeh Mayvan et al., 2017] is a recent systematic mapping study of the whole field which can help get a basic map of the subtopics it includes.

At a basic level, software design patterns were introduced in the 80s (see, for example, [Smith, 1987, Beck, 1987]) in the context of object oriented programming and taking inspiration from architecture. Software design patterns are general reusable approaches to software engineering situations. Their value arguably lies in the systematicity of applying the pattern to the situation they are designed for, reducing how much specific design needs to happen for each instance and the likeliness of problems appearing with the approach, as well as increasing the generality and interoperability of the implementation. The less general a software engineering situation or problem is, the less likely a design pattern is to be useful.

Some important topics around the idea of design patterns include: *refactoring* of implementations and designs, both to be more general or leveraging previously utilized patterns to more easily change aspects of an implementation; and *modeling languages* (see, for example, UML (`https://www.uml.org/`)) that are particularly prepared for

designing using patterns. Once again, the literature on these topics is immense and the UML website or a text book on software engineering (such as [Mall, 2018]) would be a good starting point for the basics on these.

These concepts are clearly related, and have influenced, our approach to fault detection patterns in this thesis. However, there are some important differences to note. First, fault detection patterns as presented in this thesis, while in many cases originated from design patterns in ontologies (see §2.1.1), are meant for *automatic detection*, whereas software design patterns are primarily and majoritarily aimed at informing the design process of the software, not the debugging. Second, fault detection patterns as presented in this thesis are considerably more formalized than software design patterns typically are. This is a particular strength and focus of this thesis, and derives from the first difference mentioned.

A topic in software engineering conceptually closer to the ideas presented in this project is that of bad smells [Fowler, 1999]. *Bad smells* are situations in programs that typically indicate that a bad design is present. They are found by looking for patterns in the shape of a program, and are accompanied by schematic directions on the refactorings that usually ought to be done to "fix" them. Similarly to the ideas presented in this thesis, they appeal to *abstract* notions that do not point out that an error is certain to exist but rather likely bad structuring.

Bad smells were originally introduced [Fowler, 1999] as a taxonomy of undesirable situations in software to be used by software developers to *manually* find refactorings to be made in their programs. However, similarly to fault detection, attempts have been made to try to automate their detection [Fontana et al., 2012].

The similarities between the two are mostly conceptual, though, and direct or semi-direct application of tools and specific approaches to bad smell detection and repair in software engineering to ontology fault detection does not seem like a feasible idea, partly due to the complexity of a formal translation method between programs and ontologies, but also and perhaps more importantly because of how bad smell works are mostly focused on typical patterns and situations found in software development that indicate a fault in a software development setting, but do not necessarily do so in their corresponding ontology setting (for example, because they may be related to typical

errors that programmers or software engineers make, caused by mostly contingent reasons not necessarily related to the semantics of the program). In summary, while the conceptual ideas are similar, the particularities (which are essential to an effective solution to the problem) are quite different, and therefore there are good reasons to develop specific work in each field. That said, and while not close to the topic of this thesis, it may be an interesting area of research to formally translate these methods and apply them in the corresponding setting; an area that we have *not* explored.

An interesting piece of work is [Balaban et al., 2015]. In it, the authors present a language for defining software patterns and anti-patterns, provide some patterns using such language, compare it with similar approaches and evaluate it using experiments. The idea of formalizing patterns in a higher-level language and matching those against real models is very similar to our approach, even though its field of application is software engineering and, most importantly, it is focused on the definition of patterns rather than on their detection.

2.3 Automated theorem proving and related topics

One of the main reasons for the attractiveness of ontologies, and especially those based on some form of logic, is the ability to *reason* over them. This refers to the ability to infer entailed facts about the ontology from its axioms through some form of *automated theorem proving*. Moreover, the meta-ontology fault detection approach on which this PhD is based relies heavily on automated reasoning concepts and capabilities.

2.3.1 Fundamentals of the semantics and algorithmics of logic

Logic as a formal system, as understood in this thesis, appeared originally in the form of first-order logic. The precise origins of first-order logic are multiple, broad and hard to track down, but an overview can be found in [Ferreirós, 2001]. Overall, its current shape appeared from the hand of several mathematicians and philosophers in the end of the nineteenth century and the beginning of the twentieth. Although both propositional and first-order logic as concepts had existed long before, it is the formalizations developed in that period that sustains most of today's work in logic. Miriads of introductory books to logic exist. A relatively modern and AI-oriented

approach can be found in [Bundy, 1983] or [Robinson and Voronkov, 2001]. In chapter 3 we briefly describe the foundational syntax and semantics of first-order logic that we work with.

Automated theorem proving has been a research topic almost since the efforts regarding the formalization of logic and the meta-theory of logic in the first half of the twentieth century, even before computers were a common or even a real thing; with formal approaches not just to the representation of knowledge, but also to the process of logic deduction. These approaches are in most cases based on systems of deduction rules. For first-order logic, natural deduction [Jaśkowski, 1934] is often regarded as the most intuitive of such systems. However, as an automated system, and in its most naive form, it is a practically infeasible one, due to its complexity properties, but it provides a good foundation for meta-reasoning about deduction systems and their algorithmic properties. Successful approaches build on the basis of natural deduction, while being more careful with its search properties. For example, the foundations of the higher-order theorem prover Isabelle [Paulson, 1989] are arguably in this group.

Other approaches, such as *resolution* [Robinson et al., 1965, Bundy, 1983, Robinson and Voronkov, 2001] and *tableaux* based methods [Beth, 1955, Robinson and Voronkov, 2001], have been long known to have more attractive computational properties and many automated theorem provers are based in variations of these. Both of these methods typically rely on first-order *unification* [Herbrand, 1930, Robinson et al., 1965, Bundy, 1983, Robinson and Voronkov, 2001], a symbolic technique for matching two or more formulae which has many other applications outside first-order logic, and which is a fundamental focus in this work.

Theoretically, it is well known that provability in first-order logic is, in general, semi-decidable[2]. Plenty of proofs of this fact are known, the most well-known ones being published almost simultaneously by Alan Turing [Turing, 1937] and Alonzo Church [Church, 1936]. Moreover, even for provable formulas, the complexity of any algorithm for finding proofs for them is in general at least exponential on the size of the formula. These results are relevant because they state the limits of any automated theorem proving algorithm. Of course, these limitations do not make the problem hopeless, and modern approaches (described in §2.3.2) use either *heuristics*, reductions in the *expressivity* of the logic or a combination of these to tackle the problem in a more

[2] Algorithms exist that will provide a positive result for every provable formula, but will not always terminate for non-provable formulas.

practically feasible way, although usually still using a core deduction system from the ones mentioned. A family of reduced expressivity logics of particular relevance in the ontology engineering community are description logics (see §2.3.4).

Another relevant topic in the context of this thesis is *higher-order logic* [Church, 1940, Andrews, 2010, Robinson and Voronkov, 2001], which extends first-order logic by allowing predicates and functions to also be first-class elements[3]. Automated theorem proving techniques for higher-order logic (see, for example, [Paulson, 1989, Sterling and Shapiro, 1994a, Miller, 2021]) also rely on systems of deduction rules, which normally also use unification. Higher-order unification is, however, a qualitatively more complicated problem than first-order unification. For instance, it has an infinite number of maximal solutions in general (as opposed to the unique one of first-order unification), along other highly relevant additional complexities. Therefore, the standard algorithm for higher-order unification [Huet, 1975] is not only non-terminating in general, but also has to make certain search space choices, and has very unattractive complexity properties when attempting to find all individual unifiers. While in many applications this complexity can be avoided by relying on simply checking for *unifiability*, this is not enough in others. The technical problem that the work in this thesis solves can formally be presented as a particular case of finding all maximal unifiers in higher-order logic (see chapter 5 for a more detailed discussion). Our approach shares many aspects with higher-order unification, and therefore we have taken inspiration in this fundamental work, and in particular in the higher-order unification algorithm proposed by Huet [Huet, 1975] and other algorithms based in it.

A particular case of higher-order logic is *second-order logic*, where only one level of functions and predicates over other functions is allowed. Second-order theorem proving, and in particular second-order unification, are also in general undecidable ([Levy and Veanes, 2000, Farmer, 1991, Levy, 1998]). However, a relevant amount of research effort has gone into finding subsets of second-order logic that have decidable and/or efficient algorithms. The most prominent examples include linear second-order unification [Levy, 1996], monadic second-order unification [Farmer, 1988] and bounded second-order unification [Schmidt-Schauß, 2004]. This is relevant to our work because our technical problem (chapter 5) is a subset of second-order logic, and thus the question of whether these algorithms or adaptations for these can be used for our purposes is

[3]Therefore, there can be variable functions/predicates and higher-order functions/predicates whose domain and/or range are other functions/predicates.

relevant. We discuss this in more detail in chapter 5, but a summary is that we find our problem is clearly not contained in any of these languages and the fundamental ideas behind these algorithms work are not applicable in any natural way to our problem in the way we would ideally want. It is, however, potentially interesting to consider the application of bounded second-order unification in particular to a restricted version of our problem, as future work; even though the complexity properties are still in general problematic. We discuss this in chapter 10.

2.3.2 Heuristics, domain-specific approaches and other modern topics

Modern topics in automated theorem proving usually involve at least one of three things: altered expressivity, heuristics or interactivity with automated theorem provers. These topics are, however, not in general closely related to this thesis and therefore we only provide a high level overview.

By altered expressivity we refer to constraining or extending a conventional logic language (like first-order logic or higher-order logic), altering both the scope of its semantics and its algorithmic properties. A good example of reduced expressivity is description logics, which we discuss in more detail in §2.3.4. Other examples include intuitionistic logic [Heyting, 1966, Van Dalen, 1986] and linear logic [Girard, 1987]. Examples of increased expressivity include modal logics [Kripke, 1959, Kripke, 2007, Emerson, 1991], such as temporal logic [Prior, 1962, Emerson, 1991].

Both within conventional propositional, first-order or higher-order logic, or within other altered expressivity logics, the particularities of the algorithmics are another big area of research. In conventional logics, however, the general complexity properties of the problem are well understood and overall significant changes to general purpose algorithms have not appeared in the past few decades. Thus, most of the research focus is on enhancing the general structure of the algorithms with heuristics that refine the search, computing approaches that optimize the usage of resources or in solving specific subsets of the language in particular ways, relying on domain-specific semantics. An example of the latter is SMT (Satisfiability Modulo Theories) [Nieuwenhuis et al., 2006]. SMT is a general approach to theorem proving which re-

duces non-propositional problems to propositional logic algorithms (in particular, SAT (satisfiability) [Nieuwenhuis et al., 2006, Gomes et al., 2008, Davis et al., 1962] problems), by using specific encodings of non-propositional aspects of certain domains in efficient ways. Two typical examples are the theory of arithmetic or the theory of arrays, which are much simpler and computationally efficient when handled in a domain-specific way than when encoded naively into first-order logic. There are two families of SMT approaches: *lazy* and *eager*. The eager approach clearly separates the propositional from the non-propositional aspects, by transforming each individual instance of the target problem into a traditional propositional formula. The lazy approach instead uses the structure of the propositional problem to query the non-propositional encoding in targetted ways. It can be argued that the lazy approach has more flexibility and a wider application scope. SAT and SMT have enormous practical applications and are a highly active area of research and of practical application. The effectiveness of SMT approaches depends mainly on two (interrelated) factors: the efficiency of the underlying SAT algorithm used (which in itself is in large proportion dependent on heuristics), and the power of the encoding used to represent non-propositional aspects. This second factor is ultimately related to the particular theory. An interesting idea is to encode the entire theory of first (or higher) order logic in SMT and use SMT to solve the much more general family of problems that these logics present. To a degree, this is as difficult (both in terms of decidability, complexity and mathematical prowess) a problem as it is to implement native automated theorem provers for these logics. However, it is arguable that the optimized search capabilities of SMT solvers can be of use in this, and separate the problem somehow. Some work has been done in this regard (see, for example [Barrett et al., 2002, Bongio et al., 2008], which encodes first-order logic as an SMT theory), but the results are mixed and limited. Encoding the particular logic problems contained in this work as an SMT theory is in principle a sound idea, but would require an important amount of theory development of a similar nature to that included in this thesis, and it is not one that we have chosen to explore. We discuss this issue a bit further in chapter 5.

In altered expressivity logics, while the usual algorithms still form the basis of most reasoning approaches, they are more heavily modified in consonance with the modified expressivity properties of the language, and this is still an active area of research. See, for example [Tsarkov and Horrocks, 2006, Baader and Ghilardi, 2011].

Improving the way in which humans interact with theorem provers is also an active

area of research, often related to that of heuristics: human input can be used as a heuristic, but how to capture this input and use it is not a trivial question. Topics in this area typically include interactive and/or tactical theorem provers that allow the user to control the proof search and strategies (see [Paulson, 1989], for example), but also work on the explainability of inference, such as [Horridge, 2011].

The field of *logic programming* is related to all topics discussed in this section, as it involves enhancing the semantics of different expressivity logics with computational aspects that, to a degree, allow the human (programmer) to control the search aspects of the problem, reducing the complexity and performance issues and the need for heuristics. The first and most important logic programming theory / language is Prolog. Plenty of academic and technical resources exist for conventional Prolog, but to cite a general one the reader may take a look at, consider [Sterling and Shapiro, 1994b]. Semantically, Prolog is a subset of first-order logic attached to a particular form of resolution theorem prover, restricted to Horn clauses [Wikipedia contributors, 2022b], with a particular set of rules for how first-order variables can be used and resolved for. Formulas in Prolog are instead presented as *rules* or *facts* (each of which is a Horn clause), with added computational notions to them; mainly to do with how variables are unified, and the specific order in which they are resolved. It is in the clearly specified rules for how resolution and unification are carried out that Prolog gives the programmer the power to determine the particular way in which they wish to solve their problem. As the name implies, logic programming is generally considered to be inbetween automated theorem proving and programming. In particular, it is more common to use logic programming to create software solutions (though normally to problems with a predominantly mathematical nature) than it is to use it to produce general automated theorem proving solutions. Many variations on the basic concept of Prolog exist. Lamdba Prolog ([Miller, 2021, Nadathur and Miller, 1988]) is a logic programming implementation with higher-order capabilities. However, its use is not very extended and it has important difficulties and limitations, both in terms of expressivity, performance and ease of use. These difficulties and limitations are mostly fundamental to the wider scope of the problem than particular issues that could be easily solved. More in general, *constraint logic programming* [Jaffar and Lassez, 1987] is a generalization of logic programming which abstracts away from the specifics of (first-order) unification and resolution, and instead embraces the constraint solving nature of logic programming to provide a general approach to a wider family of

problems that is fundamentally the same as logic programming. Conceptually speaking, our approach to existential second-order query logic (see chapter 5) is very similar to (and potentially formalizable as a particular case of) constraint logic programming, in the sense that we have goals that we process to generate (unification) constraints that we solve using particular algorithms. We discuss this in more detail in chapter 5.

In relation to constraint logic programming, the wider field of *constraint solving* needs to be looked at. In the general sense, constraint solving refers to automated or semi-automated approaches for finding solutions to sets of constraints, normally generated as an encoding of a domain problem or as a result of some form of processing of the domain problem. In constraint solving, often the aim is to find general solution approaches for families of constraint problems that avoid the specifics of the domain problems. Logic and logic-based languages are a prime choice for expressing these generic constraints. SAT [Gomes et al., 2008], SMT [Nieuwenhuis et al., 2006] and logic programming [Sterling and Shapiro, 1994b, Miller, 2021, Nadathur and Miller, 1988, Jaffar and Lassez, 1987], already mentioned, conceptually fall within this larger field, but there are other approaches. Among them, we would like to bring up Answer Set Programming (ASP) [Brewka et al., 2011, Balduccini, 2009, Baget et al., 2018]. ASP is similar to logic programming in that it expresses first order rules and facts, and to SMT in that it solves them by transforming it into a propositional problem. However, as opposed to logic programming, the resolution of ASP programs is much more automated (though some computational aspects can still be left to the programmer); and as opposed to SMT, the translation to propositional problems is completely general and automated, and is due to limits in expressivity (for example, in how negation is handled). ASP is designed to solve simpler constraint problems than those that SMT or logic programs solve, in more general and automated ways. ASP solvers are based on the same basic DPLL algorithm that SAT and SMT solvers are [Nieuwenhuis et al., 2006].

2.3.3 Rewrite systems

Rewrite systems or Rewrite rule systems are a mathematical tool used to define and reason about formal systems in which we have elements that can transform into equivalent ones through *rewrite rules*, normally to *simplify* or compare them in some way. A basic overview of standard rewrite systems can be found on [Huet and Oppen, 1980, Robinson and Voronkov, 2001].

In rewrite systems, the elements that are rewritten to and from are called *terms*, a definition which properly contains the definition of *term* in logic (that is, logic terms are a particular case of terms in general). Thus, we often refer to *term rewrite systems*. A term is simply an element that has some algebraic structure and which is, in some sense, defined by its algebraic structure.

In relation to this thesis, we use some basic notions of term rewriting in producing some standard proofs about *normalization* of first-order and second-order terms in logic, which can be found in different shape in standard reviews on logic topics, such as [Dowek, 2001]. The fundamental aspects here are of when two terms are equivalent, in what way they are equivalent, and specially about whether they can be *normalized*, that is, reduced to a *normal form*; meaning an equivalent version of a term that is clearly and intrinsically distinguished as being an equivalent version that is no longer reducible, often simpler in some sense. Normal form is related to the notion of *irreducibility*, meaning the term cannot be rewritten any more. Often, we care about normal forms being *unique*, that is, each term can only be reduced to one normal form. Very important notions in this topic are those of *confluence* and *termination* of rewrite systems. One of the main results in this area is due to [Newman, 1942], called *Newman's lemma* or the *diamond lemma*, that establishes that a terminating rewrite system is confluent if and only if it is *locally confluent*. The relevance of this lemma comes from the connection between confluence, a global property establishing the general behaviour of the rewrite system as a whole, and which can thus be difficult to prove on its own; to local confluence, which, as its name implies, is a local property that is much easier to prove, relating to the behaviour of two particular alternative reductions of the same term.

A sub-topic of rewrite systems that is particularly related to the work in this thesis is that of *term graph rewriting*. [Plump, 2002, Habel and Plump, 1995, Plump, 1999, Barendregt et al., 1987] are some good pieces of literature to get started. Standard terms based on *strings* are well known to be equivalent to *trees* in the graph theoretical sense. Indeed, graph theoretical trees are isomorphic to linear representations of information and this transformation is universal so long as there is a clear algebraic structure on the string terms. Term graph rewriting considers what happens when we extend rewrite systems to graphs that are not trees. The most studied and most interesting case is what is arguably the simplest extension of trees in this context: *directed acyclic graphs*. A

directed acyclic graph is a graph whose edges are directed, and where it is impossible to cycle through the graph while following the direction of the arrows. However, unlike trees, directed acyclic graphs can have undirected cycles; that is, cycles that do not follow the direction of the arrows.

In rewrite systems, directed acyclic graphs represent the notion of multiple dependence on the same sub-term. This is very natural, for example, when considering *variables* (such as first-order logic variables) which appear in multiple places in a term but which conceptually the represent the same element, and when substituted, must be substituted homogeneously throughout the term. It is also common (see, for example, [Habel and Plump, 1995, Plump, 1999, Barendregt et al., 1987, Plump, 2005]) to consider *hypergraphs* instead of regular graphs, meaning that edges can have multiple sources/targets. Our *unification dependency graphs* (introduced in chapter 5) and the rewrite system associated to them are a particular case of term graph rewriting, though we note that we have adopted what seems to be the opposite of the standard convention on the direction of the edges[4]. This has no effect on the theory whatsoever.

The majority of the results in term graph rewriting topics focus on the general properties of abstract term graph rewriting systems and their theoretical properties, in relation to confluence, reducibility, termination and normalization, as well as complexity; often in comparison with the simpler string/tree rewriting systems. To put it in a simplified way, results in [Habel and Plump, 1995, Dwork et al., 1984, Plump, 2005, Barendregt et al., 1987] establish that term graph rewrite systems are:

- *Sound* w.r.t. corresponding string term rewrite systems (they produce correct results).
- *Complete* w.r.t. corresponding string term rewrite systems (they produce all correct results).
- *Terminating* and *confluent* w.r.t. corresponding string term rewrite systems under some basic additional conditions dependent on the particularities of the rewrite system.
- More **efficient in space** as a general rule, due to the *sharing* of data structures.
- Have potential for more **efficient parallel algorithms** for some particular subproblems, while the general standard problems remain more or less equivalent in

[4]This was discovered after this PhD's viva, and changing it would involve a lot of work and issues, specially with respect to the implementation, so we chose to note it rather than change it.

time complexity.

We note that in chapter 7 we have proven soundness, completeness, termination and confluence (under conditions) results to our particular algorithm / rewrite system based on unification dependency graphs. Moreover, we argue (though we do not prove) the moderate efficiency advantages mentioned above, and also argue that a graph data structure allows our **search** to make decisions based on **additional information** that works as a mild heuristic for the search inherently involved with second-order unification.

2.3.4 Description logics

While most or all of the approaches described so far can and are applied to description logics (as a particular, reduced expressivity, case of first-order logic), one of the reasons for using description logics to begin with is the more attractive computational landscape that they offer.

In that regard, inference in description logics is often done using structural methods which, in some sense, work at the level of *concepts* or *classes* and their subsumption relations (as well as properties that connect them), as opposed to working at the level of *instances* of said classes and using quantification to depict subsumption. An early example of this is *structural subsumption* [Dionne et al., 1992, Dionne et al., 1993], in the context of the KL-ONE description logic [Brachman and Schmolze, 1989]. It is interesting to realize the way in which these algorithms are produced: the intuitive way in which humans would consider subsumptions by thinking about almost "graphical" relations between concepts in a graph (more precisely, reasoning about orderings in a graph) is formalized, and later compared to the first-order logic (model-theoretic) variant of it. The semantics of both approaches are connected and thus the validity of the structural subsumption algorithm for description logics is asserted, even from a model-theoretic semantics point of view, which uses the expressivity constraints of description logics to largely simplify the algorithm and allow reasoning at the *class* level.

The work in this thesis falls relatively far from this. Our approach is applicable to full first-order logic (existential second-order logic[5], in fact), but it also does not leverage the expressivity constraints of description logics, of course. A relevant area of future research lies in understanding how the core ideas of the meta-ontology fault

[5] A slight extension of first-order logic that allows existentially-quantified second-order variables.

detection framework can be applied in a description logic context and in relation to structural inference algorithms. We have chosen not to focus on that mainly because first-order logic is more foundational and, from a theoretical point of view, the results within it will carry a lot more relevance, and therefore we thought best to begin the exploration of the approach from that end.

Chapter 3

Background

In this section we discuss and provide definitions for standard concepts that are not part of the contribution of this thesis, but which this thesis builds on. Most of these are presented in more detail elsewhere, but we will try to provide references whenever possible.

3.1 Web Ontology Language (OWL)

The Web Ontology Language (OWL) [Motik et al., 2012] is one of the most used ontology languages. It is formally an instance of a *description logic* ([Baader et al., 2009], §3.2.4), but it has its own syntax(es) and approach aimed at making it accessible to people without the technical expertise in description logics.

The most important concept in OWL is that of a *class*.

Definition 3.1.1 (OWL class)**.** *In OWL, a* class *represents a family of* individuals *(*instances*) that are conceptually of the same nature.*

For example, the notion of a Pizza or a PizzaTopping might be OWL classes. Classes correspond to first-order *unary predicates* (§3.2.1). For example, an individual x being an instance of the class Pizza would correspond to the first-order statement $Pizza(x)$

The other fundamental concept in OWL are *properties*. Where classes are unary predicates, properties are first-order *binary predicates*, that link two individuals in some way.

Definition 3.1.2 (OWL property)**.** *In OWL, a* property *represents pair-wise relations between instances. An* instance *of a property is one such pair of connected individuals.*

For example, the notion of a Pizza *having a topping* can be embodied by a hasTopping property. In first-order logic, pizza x having a topping y would be represented by the statement $hasTopping(x,y)$.

One of the most important concepts relating to classes in OWL is that of one class *subsuming* another.

Definition 3.1.3 (OWL subsumption)**.** *In OWL, we say that a class A* subsumes *another class B if every individual that is an instance of class B is also an instance of class A.*

For example, the Pizza class may subsume the VegetarianPizza class.

This is similar to inheritance in object-oriented programming, and in first-order logic terms, it corresponds to the relation between two first-order predicates A and B in which A subsuming B is equivalent to the following first-order statement:

$$\forall x.B(x) \implies A(x)$$

There are some concepts in OWL that do not explicitly map to concepts in first-order logic, and which sometimes have more to do with the way the ontology is expressed than with what it formally represents itself. In this area one of the most important distinctions is that between *primitive* and *defined* classes (see [Horridge et al., 2009]). In OWL, primitive classes are usually considered to be the "baseline" classes, and defined classes are a tool used for expressivity.

Definition 3.1.4 (OWL primitive class)**.** *In OWL, we say a class is* primitive *if it is defined by a* necessary *condition but not a* sufficient *one*

Definition 3.1.5 (OWL defined class)**.** *In OWL, we say a class is* defined *if its definition consists of a condition that is both necessary and sufficient (it fully* defines *it).*

For example, Pizza could be a primitive class, meaning that some instances that satisfy its definition are not Pizzas, as long as they are not explicitly declared or inferred as Pizzas; whereas VegetarianPizza would normally be a defined class, establishing that any Pizza that only contains vegetable toppings is a VegetarianPizza, regardless of whether it has been explicitly defined as a VegetarianPizza or not.

Due to the nature of OWL, and specifically because it normally only discusses classes and properties rather than individuals, it is impossible (under standard conditions) for an OWL ontology to be *inconsistent* in the usual logical sense (§3.2.1). Instead, the much more common situation that in some sense represents a problem with representation in OWL is that of an *unsatisfiable class*.

Definition 3.1.6 (OWL unsatisfiable class)**.** *A class in OWL is said to be* unsatisfiable *if it is not possible that it has any instances.*

For example, if we defined a class VegetarianPizzasWithMeat, containing all Pizzas that are vegetarian and contain meat, it would in all likelihood be unsatisfiable.

In first-order logic terms, a class A is unsatisfiable if the following first-order statement is true:

$$\neg\exists x.A(x)$$

Finally, when discussing properties, there are two aspects often discussed about them, called the *domain* and *range* of the property.

Definition 3.1.7 (OWL domain and range)**.** *Given an OWL property, its* domain *is the OWL class whose instances are exactly those individuals that appear on the left side of one instance pair of the property.*

Similarly, its range *is the OWL class whose instances are exactly those individuals that appear on the right side of one instance pair of the property.*

For example, the domain of hasTopping is the set of all pizzas that have any toppings at all, whereas its range is the set of all toppings that appear in a pizza.

In first-order logic terms, A is the domain and B the range of property R if the following statements are true:

$$\forall x.A(x) \iff \exists y.R(x,y)$$
$$\forall y.B(y) \iff \exists x.R(x,y)$$

3.2 Logic and automated theorem proving

3.2.1 First-order logic

All of the concepts discussed in this section are described in much more detail in standard books on logic and algorithmic theorem proving, such as

[Bundy, 1983, Robinson and Voronkov, 2001].

First-order logic is what is usually abbreviated as logic or mathematical logic. It is a formalization of deductive reasoning that differentiates from higher-order logic in that functions and predicates are not themselves first-order elements. That is, there are no statements about functions, predicates or statements themselves.

The most basic definitions in first-order logic are those revolving around the notion of a *signature*. A signature is a definition of a language of discourse for a set of logical statements.

Definition 3.2.1 (First-order signature)**.** *A first-order* signature *consists of a set of* variables*, a set of* function symbols *and a set of* predicate symbols*.*

- Predicate symbols *are syntactic representations of properties that can be stated about elements in the* universe of discourse*. They can have different* arities*, indicating how many elements the statement is about. For example, an isPizza predicate would be unary (arity 1) and indicate whether a given element is a pizza or not. A predicate hasTopping would be binary (arity 2) and indicate whether a given pizza has a certain topping.*

- Function symbols *represent functional relations between elements in the universe of discourse. Similarly to predicates, functions have an arity. Functions with arity 0 are often called* constants *and treated separately, though this distinction is formally unnecessary. As an example, a unary function father could indicate the father of any other one individual, whereas a 0-ary function (constant) Mars could refer to the specific individual in the universe of discourse corresponding to the planet Mars.*

- Variables *are syntactical tools that represent unidentified elements, in the context of a formula, of the universe of discourse. They are normally used together with* quantifiers*, explained later in this section, to produce general statements about families of elements in the universe of discourse.*

When predicates, functions and variables are combined to represent the simplest statements in the language, we refer to these as *atoms* and *terms*.

Definition 3.2.2 (First-order term)**.** *First-order* terms *are formed either by variables or functions applied to other terms. For example, x, father*(x) *or Mars*$()$ *are terms.*

A term is said to be ground *if it contains no variables.*

Definition 3.2.3 (First-order atom)**.** *First-order* atoms *are formed by applying predicates to terms, such as isPizza*(x) *or hasTopping*$(father(father(x)), y)$.

The perhaps most fundamental aspect of first-order logic is the usage of logical *connectives* to produce more complex statements (*formulas*) from the simplest formulas represented by atoms.

Definition 3.2.4 (First-order connectives)**.** *The simplest connective is the* negation *of another formula, which by definition is considered to be satisfied if and only if the original formula is not. We usually write* $\neg F$ *to represent the negation of formula* F*. For example,* $\neg isPizza(x)$.

It is usual to use the terminology literal *to refer to either atoms or the negation of atoms.*

Binary connectives are usually formed by the conjunction *("and"), the* disjunction *("or") and the* implication*, though technically only any one of these three is necessary for the language to be complete, as the others can be built from that one and negation.*

The conjunction $F \wedge G$ *of two formulas* F *and* G *is satisfied if and only if both* F *and* G *are satisfied.*

The disjunction $F \vee G$ *is satisfied if and only if at least one of them is satisfied.*

The implication $F \implies G$ *is satisfied if and only if* F *is not satisfied and/or* G *is satisfied.*

In close connection to connectives are *quantifiers*. Quantifiers are used in combination with variables to express more general statements about the universe of discourse.

Definition 3.2.5 (Universal quantification)**.** *The* universal *quantifier* $\forall$ *is used together with a variable (for example x) and a formula containing the variable x, which we will write as* $\phi(x)$.

The universally quantified formula $\forall x.\phi(x)$ *is satisfied if and only if for every element a in the universe of discourse, when x is replaced by a,* $\phi(a)$ *is satisfied.*

For example $\forall x.isPizza(x)$ *is satisfied if every element in the universe of discourse is a pizza.*

Definition 3.2.6 (Existential quantification)**.** *The* existential *quantifier* $\exists$ *is used together with a variable (for example x) and a formula containing the variable x, which we will write as* $\phi(x)$.

The existentially quantified formula $\exists x.\phi(x)$ *is satisfied if and only if there is at least one element a in the universe of discourse such that, when x is replaced by a,* $\phi(a)$ *is satisfied.*

For example $\exists x.isPizza(x)$ *is satisfied if there is at least one element in the universe of discourse that is a pizza.*

3.2.1.1 Interpretations and Herbrand semantics

So far, we have talked about the universe of discourse and formulas being satisfied, but we have not formalized it. This is done through *interpretations*.

Definition 3.2.7 (Interpretation)**.** *An* interpretation *defines a set of elements as the* universe of discourse*, associates each function symbol with a function within that universe and each predicate symbol with a set of tuples of elements in that universe (of the same arity as the predicate).*

In other words, it establishes which atoms are true and which ones are not in a coherent way.

An important type of interpretations from a theoretical point of view are *Herbrand interpretations*, associated with *Herbrand universes*, *Herbrand structures* and *Herbrand bases*.

Definition 3.2.8 (Herbrand universe)**.** *The* Herbrand universe *of a first-order signature is the universe of discourse containing the ground terms of the logical language as elements.*

Definition 3.2.9 (Herbrand structure)**.** *The* Herbrand structure *is an interpretation for function symbols on top of the Herbrand universe, that assigns to each function symbol the function result of applying it.*

For example, the interpretation of the unary function symbol f is the function f^H that, given a term α, as an element of the Herbrand universe, produces the term $f(\alpha)$. That is, $f^H(\alpha) = f(\alpha)$.

Definition 3.2.10 (Herbrand base)**.** *The* Herbrand base *extends the Herbrand structure by adding interpretations for predicate symbols, with the same rationale.*

Definition 3.2.11 (Herbrand interpretation)**.** *A* Herbrand interpretation *finally extends the Herbrand base by assigning a certain truth value to each ground atom.*

One of the reasons why Herbrand interpretations are relevant is *Herbrand's theorem* [Herbrand, 1930], that establishes a connection between Herbrand interpretations and arbitrary interpretations, fundamentally allowing us to analyze the general semantics of a logical theory by looking only at its Herbrand interpretations. It can be used to show the soundness and completeness of an inference system.

3.2.1.2 First-order resolution and unification

Definition 3.2.12 (First-order theory)**.** *A set of formulas in first-order logic that establish the basic true facts of any universe of discourse that we are considering is called a* theory.

Given a theory T, we typically consider whether some formula F holds true in *every interpretation* of the language that satisfies all formulas in T. That is, F is a logical consequence of T. We call such interpretations the *models* of the theory.

Definition 3.2.13 (First-order model)**.** *Given a theory T and an interpretation I, we say that I is a* model *of T if every formula in T is true in I.*

Definition 3.2.14 (Entailment)**.** *We say that F is* entailed *by T, written $T \models F$ if F is true in every model of T.*

Thinking about entailment allows us to use logic as a way to talk about semantics themselves without ever having to explicitly consider interpretations.

This becomes most useful when using systematic reasoning techniques.

Definition 3.2.15 (Inference system)**.** *An* inference system *is a set of rules that can be systematically applied to formulas in a theory to produce other formulas and eventually check whether a formula is entailed by a theory.*

Definition 3.2.16 (Formal proof)**.** *Given an inference system, we call a sequence of applications of its rules a* proof *within the system, and we say that the formula F can be proven from T, written $T \vdash F$.*

Definition 3.2.17 (Soundness and completeness of an inference system)**.** *An inference system is said to be* sound *and* complete *if* $T \vdash F$ *if and only if* $T \models F$.

What this means is that we can check for entailment (semantics) entirely via syntactic means (proofs). This further typically allows us to automatize inference into an *automated theorem prover*.

Resolution is one of the most commonly used inference systems for first-order logic, that relies on two important concepts: *conjunctive normal forms (CNF)* and *unification*.

A formula is said to be in *conjunctive normal form* if:

- All quantifiers are at the front of the formula.
- The formula inside the quantifiers is a conjunction of *clauses*.
- A *clause* is a disjunction of literals.

For example:

$$\forall x.\exists y.\forall z.(p(x) \vee \neg q(y)) \wedge (\neg p(c()) \vee q(x)) \wedge (r(x,y,z))$$

is in CNF, with three clauses: $(p(x) \vee \neg q(y))$, $(\neg p(c()) \vee q(x))$ and $r(x,y,z)$.

Formulas in conjunctive normal form are *Skolemized* by removing existentially quantified variables and replacing them with *Skolem functions* that depend on the universally quantified variables whose scope is bigger than the existentially quantified variable. After doing this, we normally omit universal quantifiers, since all remaining free variables in the formula are implicitly assumed to be universally quantified. For example, the Skolemization of the formula above would be:

$$(p(x) \vee \neg q(f_y(x))) \wedge (\neg(p(c()) \vee q(x)) \wedge (r(x, f_y(x), z))$$

where we note that f_y is not some parameterized version of f, but rather a new function symbol introduced into the signature that is associated with the variable y.

Every first-order logic formula can be algorithmically transformed into a Skolemized conjunctive normal form with fundamentally[1] the same semantics.

[1]The Skolemized formula is satisfiable if and only if the original is.

Unification takes two or more atoms or terms with variables and unifies them, finding the least committed substitution of the variables for which the two atoms or terms are equal. For first-order logic, it can be shown that every unification problem has a single, most general solution, called the *most general unifier*.

For example, the unification problem:

$$p(f(x,y),x) \sim p(r,g(s))$$

has as most general unifier the substitution:

$$\begin{aligned} x &\to g(s) \\ r &\to f(g(s),y) \\ y &\to y \\ s &\to s \end{aligned}$$

which, when applied, makes both sides of the above equation become

$$p(f(g(s),y),g(s))$$

Given a Skolemized CNF formula, we can apply the rule of *resolution* which takes two clauses and a chosen literal (or set of literals) for each of them (positive in one clause, negated in the other), unifies them, removes them from each clause, and joins the remainder of both clauses with the unifier applied to them; producing a new clause that is then added to the CNF. For example, we can resolve the first literal in each of the following two clauses:

$$\begin{aligned} &p(f(x),y) \vee q(y) \\ &\neg p(r,g(g(s))) \end{aligned}$$

into the following *resolvent*:

$$q(g(g(s)))$$

The resolution rule can be applied to Skolemized CNF formulas to produce a sound and complete inference system, in the following way:

1. Consider the theory T, and present it as a conjunction C_T of all formulas in T.

2. Transform C_T into a Skolemized CNF C_T^N.

3. Take the formula to try to prove F, and consider its negation $\neg F$, presented in Skolemized CNF $\neg F^N$.

4. Conjunctively join $C_T^N \wedge \neg F^N$, this will trivially be a Skolemized CNF as well.

5. Apply the resolution rule repeatedly, conjunctively joining resolvents to the CNF each time.

6. If at any point, an empty clause is produced, then we can conclude that $T \vdash F$ (and therefore $T \models F$) and finish the proof.

The reasons for which this method works can be summarized by the following facts:

- $T \models F$ if and only if $T \wedge \neg F$ is *unsatisfiable* (no interpretation satisfies this conjunction). The usage of this fact is why resolution is commonly referred to as a *refutation* proof method.

- The resolution rule preserves the set of models of a Skolemized CNF formula.

- An empty clause in a CNF formula is unsatisfiable, since it is the disjunction of zero literals (and so none of them can be satisfiable).

A relevant property of all sound and complete inference systems for first-order logic, including resolution, is that they are *semi-decidable* (see §3.5), meaning that they will always terminate with a positive result when a formula is provable, but they may sometimes not terminate when a formula is not provable.

More details about resolution can be found in [Bundy, 1983, Robinson and Voronkov, 2001].

3.2.2 Higher-order logic

Second-order logic is the extension of first-order logic by allowing functions and predicates whose universe of discourse are first-order functions and predicates themselves, as well as second-order variables that can be quantified over.

Similarly, one can further extend this to higher and higher levels of abstraction, or include arbitrary levels within the same language. This is what is called *higher-order logic*. A thorough, general overview of higher-order logic can be found in

[Paulson, 1989]

One of the most fundamental elements added in higher-order logic with respect to first-order logic is that of *lambda abstractions*, original from lambda calculus [Barendregt, 1992]. A *lambda abstraction* is an explicitly defined function, indicating how it syntactically combines its arguments in its *body* through variables. For example, the function $\lambda f.\lambda x.f(g(f(x),x))$, which would typically instead be written using the different notation $\lambda f.\lambda x.f(g(fx)x)$, is a second-order function that takes the variable f, ranging over first-order functions, and the variable x, ranging over first-order elements, and returns the result of applying f to the result of applying the constant function g to its two arguments $f(x)$ and x.

3.2.2.1 Higher-order unification

Most of the concepts of first-order logic can be extended to higher-order logic. However, the algorithmic properties of some of these become more problematic. This becomes most clear with *higher-order unification*. In higher-order logic, not only do some unification problems not have a single most general unifier (instead having multiple, and sometimes infinitely many, maximal unifiers which do not contain each other); but even checking for the *unifiability* of two or more terms (is there a unifier at all?) is a semi-decidable problem in general (see §3.5).

Nonetheless, extensive work has been done trying to provide powerful tools to higher-order logic, due to its incredibly attractive general expressivity properties. A crucial piece of work is Huet's *higher-order unification* algorithm [Huet, 1975], which, while obviously remaining semi-decidable, and inevitably intractable even when decidable, has some attractive algorithmic properties based on deep insights about the nature of higher-order unification problems. This is also one of the most complex results in traditional automated theorem proving research. A comprehensive survey of this and related algorithms can be found in [Dowek, 2001].

We can describe two of the most important topics behind Huet's algorithm in the following way:

- **Least (or minimal) commitment** - Huet's algorithm deals with a large portion of the problem's undecidability and intractability issues by prioritizing exploration

of possible solutions that do not incur so much in it. A perfect example of this is what are often called "flex-flex pairs", which concerns the unification of two higher-order expressions that both have variable heads (functions). This is one of the most important situations in which there is not a most general unifier, but there is a large number of maximal unifiers, highly dependent on the specifics. Huet's algorithm encodes these cases implicitly (as equations), avoiding the exploration of their explicit solutions, instead focusing on parts of the problem that are more likely to produce deterministic results. This is what we call a *minimal commitment* approach, because the instantiation of variables is only produced to the extent that is required to continue the execution of the algorithm. The *commitment* to *specific solutions* is *minimal*.

- **Non-determinism** - While Huet's algorithm is not often explained explicitly in terms of non-determinism, it can be understood in such a way, and this perspective is very relevant to this thesis. As mentioned in the previous point, the lack of a most general unifier can be dealt with by non-deterministically exploring multiple branches of possible solutions, that only when combined together reconstruct the original solutions.

Both of these properties are central in the work presented in this thesis. Minimal commitment resolution for existential second-order query logic, the algorithm presented as one of the main contributions of this work, leverages heavily on both of these. However, as we discuss in chapters 5 and 7, the scope of our algorithm is, in one sense, an extension of higher-order logic, but in most other senses a constrained version of it. This means that our algorithm solves a less general problem in similar, but (we argue, or we attempt) in more efficient ways than higher-order unification does.

3.2.2.2 Second-order unification

A particular case of higher-order unification is second-order unification. While second-order unification is a subset of higher-order logic; it is also, in general, only semi-decidable. However, a relevant amount of research effort has undergone on finding the fundamental limits and elements of the language that make it undecidable [Levy and Veanes, 2000, Farmer, 1991, Levy, 1998]; and on finding specific subsets of second-order logic that are decidable and potentially useful. Some of the most prominent examples of these are bounded second-order unification [Schmidt-Schauß, 2004],

monadic second-order unification [Farmer, 1988] and linear second-order unification [Levy, 1996] (which is not, in general, decidable, but certain subcases of it are).

In chapter 4 we describe our language for representing patterns as faults, which we call *existential second-order unification*. This is a subset of second-order unification. It is therefore an important question whether this subset is decidable, whether one of the decidable algorithms described for the languages above can be applied to it, or the ideas behind them can possibly be extended to them. The extent of this discussion can be found on chapter 5, but the general conclusion is that existential second-order unification is clearly **not** a subset of any of the above mentioned languages, and the ideas underlying these algorithms cannot be applied directly to our problem in a clear way that does not require further research beyond the scope of this PhD. Of course, the possibility of taking inspiration in these ideas to develop more complex algorithms that deal with our problem in ways not considered in this thesis always remains open, but I have not found any result in that respect; and we briefly introduce the fundamentals of these languages and their decidable unification algorithms here in order to enable the discussion in chapter 5.

3.2.2.2.1 Linear second-order unification

Linear second-order unification [Levy, 1996] is the problem of unifying general second-order terms, but restricting the set of unifiers under consideration to those that instantiate second-order variables to *linear terms*.

Definition 3.2.18 (Linear second-order term)**.** Linear terms *are second-order terms where bound variables in a lambda abstraction occur exactly once in the body of the lambda-abstraction.*

For example, the following is a linear term:

$$\lambda x, y.f(x,y) \tag{3.1}$$

while the following is not:

$$\lambda x.f(x,x) \tag{3.2}$$

The principal way in which algorithms for cases of linear second-order unification exploit the linearity of the problem is by forgoing two rules in general second-order

logic (referred to as *elimination* and *iteration*), which non-deterministically extend the search space of the unification problem by considering whether a variable could appear fewer or more times in the instantiation. These rules are inherently semi-decidable in that one can always apply them additional times, so removing them removes one of the most important sources of undecidability (non-termination).

3.2.2.2.2 Bounded second-order unification

From a slightly simplistic point of view, we can say that *bounded second-order unification* [Schmidt-Schauß, 2004] is an extension of linear second-order unification in which, instead of limiting the occurrence of bound variables in lambda-abstractions to **exactly one** we limit it to **any boundary** (a maximum number of occurrences). This means that the algorithm cannot completely forgo the consideration about the number of occurrences of bound variables in instantiations of second-order variables, but it can effectively limit how many times these rules are applied, to ensure termination (and thus decidability).

Definition 3.2.19 (Bounded second-order term)**.** *A second-order term* ϕ *is* bounded *with boundary n if there is no bound variable in a lambda abstraction within* ϕ *that occurs more than n times in the body of the abstraction.*

Definition 3.2.20 (Bounded second-order unification problem)**.** *A* bounded second-order unification problem *with boundary n is any second-order unification problem with the additional restriction that we only consider instantiations for second-order variables to bounded second-order terms with boundary n.*

[Schmidt-Schauß, 2004] provides a sound, complete and terminating algorithm for general bounded second-order unification. We note that while decidable, this problem is known to be NP-hard, as proven in [Schmidt-Schauß, 2004], meaning that the algorithm is (currently and probably unavoidably) computationally problematic.

3.2.2.2.3 Monadic second-order unification

Definition 3.2.21 (Monadic second-order term)**.** *A* monadic second-order term *is one which contains no function constants with arity greater than or equal to 2. A second-order language/signature is* monadic *if it contains only function symbols with arity less than or equal to 1.*

Unification in monadic second-order languages is called *monadic second-order unification* [Farmer, 1988]. For example, the following second-order term is monadic:

$$\lambda x.f(x) \tag{3.3}$$

while the following is not:

$$\lambda x,y.f(x,y) \tag{3.4}$$

The fundamental aspect of monadic second-order terms that the algorithm described in [Farmer, 1988] exploits is that it is fundamentally impossible for the instantiation of a monadic second-order term to have more than one bound variable (that is non-trivial). Because each function symbol is monadic, the body of a lambda abstraction will always consist in a (potentially empty) sequence of applications of function symbols to individual terms, and ultimately to either a single variable or a constant; but never more than one variable. This means that instantiations of monadic second-order terms are isomorphic to sequences (or *words*). For example:

$$\lambda x,y.f(g(h(x))) \tag{3.5}$$

ignores its second argument, and thus is essentially the same as:

$$\lambda x.f(g(h(x))) \tag{3.6}$$

that is, there are no non-monadic instantiations of second-order variables. Thus, this is fundamentally isomorphic to the word:

$$fgh \tag{3.7}$$

This property greatly simplifies the complexity of the problem, mainly by completely removing the aspect of function substitution and correlation between different variables from the problem.

An important aspect of monadic second-order unification, and in particular the algorithm shown in [Farmer, 1988], is that in order for the algorithm to become decidable, considering *unification schemata* as opposed to individual unifiers, is necessary. The unification schemata utilized in [Farmer, 1988] allow variable numbers of repetitions of subsequences within a sequence to represent infinite (but regular) sets of unifiers that do

not have a common most general unifier. The general concept of *unification schemata* is a very interesting one that could potentially be used in more clever ways, both in this thesis and elsewhere. We do not directly utilize unification schemata, but the underlying ideas are present in the way we discuss dependency graphs, introduced in chapter 5 and explained in more detail in chapter 7; and in the way our particular implementation handles answer sets (chapter 8).

3.2.3 SAT and SMT

Propositional logic is logic without object variables or quantifiers. It is generally considered the simplest type of logic and explained first. The reader may familiarize themselves with propositional logic in any standard logic textbook [Bundy, 1983, Robinson and Voronkov, 2001]. For simmplicity, we define it from the previously given first-order logic definition before, by removing variables and quantifiers.

Definition 3.2.22 (Propositional logic)**.** Propositional logic *is the subset of first-order logic with no object variables or quantifiers, and with added* propositional variables.

Definition 3.2.23 (Propositional variable)**.** *A* propositional variable *is a variable that functions as an atom. An* interpretation *of a propositional variable assigns it either a* true $\top$ *or* false $\bot$ *value.*

For example, the propositional formula:

$$(p \vee q) \wedge (r \vee \neg q) \tag{3.8}$$

has three propositional variables: p, q and r. If, for example, an interpretation assigns all three to $\top$, then the formula is true in that interpretation. If, for example, an interpretation assigns all three to $\bot$, then the formula is false in that interpretation.

Definition 3.2.24 (Satisfiable propositional formula)**.** *A propositional formula is* satisfiable *if there is at least one interpretation of its propositional variables that makes it true (*satisfies *it).*

Definition 3.2.25 (SAT (Satisfiability problem))**.** *The* satisfiability problem *(usually abbreviated SAT) is the problem of, given a propositional formula, deciding whether it is satisfiable.*

SAT [Gomes et al., 2008] is one of the most studied algorithmical problems that exist. The basic algorithm that most modern SAT solvers are based on is the DPLL (Davis-Putnam-Logemann-Loveland) procedure [Davis et al., 1962, Nieuwenhuis et al., 2006]. DPLL is a parametric search procedure which utilizes the relation between propositional variables as established by the propositional formula to guide the search. A large proportion of the research done in SAT and related topics [Gomes et al., 2008] has gone into studying heuristics for precisely guiding this search.

Algorithmically, SAT is normally considered to be the fundamental NP-complete problem [Wikipedia contributors, 2022c], with the Cook-Levin theorem [Cook, 1971] establishing that SAT is an NP problem (a solution can be checked in polynomial time) and an NP-hard problem (if a polynomial algorithm for SAT existed, then it could be used to build a polynomial algorithm for any other NP problem), making it NP-complete. It is common to prove NP-hardness of other problems by polynomially reducing SAT to them, as a proxy for every other possible NP problem.

In relation to its NP-completeness, there is a plethora of search and constraint problems with real-world applications that can be reduced to SAT. Therefore, fast SAT solvers are highly attractive for their practical applications as much as for their theoretical insights. However, there are plenty of search and constraint problems that *cannot* be reduced to SAT. For example, first-order logic. Usually (and informally), these problems include an infinite aspect of some nature (e.g. quantifiers in first-order logic). Mainly in order to deal with some of these problems that are not, technically, reducible to SAT, but which have a lot of structure that can be exploited in simple ways (including relevant subsets of first-order logic), SMT (Satisfiability Modulo Theories) was developed. An overview of SMT can be found in [Nieuwenhuis et al., 2006]. Formally, an SMT problem is a problem with a propositional component (SAT) and a first-order component, that is implemented in a specialized way.

There are two general approaches to SMT solvers: *eager* and *lazy* approaches. In the eager approach, the problem is first reduced to a propositional problem, and the propositional problem is then solved using an off-the-shelf SAT solver. In the lazy approach, the SAT solver works closely with the specialized theory that enables exploration of the boolean value of first-order expressions in the given theory during execution of the SAT algorithm. We can define this a bit more formally.

Definition 3.2.26 (Eager SMT algorithm)**.** *An* eager SMT algorithm *is an algorithm to solve a problem consisting of two separate parts:*

1. *A part that reduces an arbitrary instance of the problem to the problem of establishing the satisfiability of a finite number of propositional formulas.*

2. *A SAT algorithm.*

Definition 3.2.27 (Lazy SMT algorithm)**.** *A* lazy SMT algorithm *is a SAT algorithm enhanced with a theory-specific oracle that the SAT algorithm can query during its execution with specific instances of the problem. The general flow and search properties of the algorithm are those of the SAT algorithm, with the oracle providing support for the non-propositional parts of the problem exclusively.*

There are a number of important theories for which SMT algorithms exist. Prominent examples include linear arithmetic [Bozzano et al., 2005], the theory of arrays [Brummayer and Biere, 2009], equality and difference constraints [Armando et al., 2004] and temporal reasoning [Armando et al., 1999]. There has even been work in utilizing the SMT approach to handle the entirety of first-order logic [Barrett et al., 2002, Bongio et al., 2008], though this comes with important limitations and is not in general preferable to standard first-order logic theorem proving approaches.

3.2.4 Description logics

We do not concern ourselves directly with *description logics* in this thesis, but we do need to acknowledge them in the context of their use for defining ontologies and defining patterns. OWL (§3.1) is a description logic, for instance.

Description logics are formally variations of first-order logic (§3.2.1) with a few important differences:

- Description logics usually have *constrained expressivity*, disallowing various forms of constructing formulas that first-order logic allows. For instance, function nesting may be disallowed, or quantification might be limited to only certain shapes, etc.

  This allows description logics to both be decidable and/or much more algorithmically attractive than first-order logic (depending on the specific description logic). That is, for many description logics, such as OWL, there are *decidable and tractable* automated inference algorithms.

- Description logics often use more human or program-like language, even if their semantics are still aligned with those of first-order logic. For example, they may use more text and overloaded syntactical constructs rather than short mathematical language and special symbols.

- Description logics typically have a considerably larger focus on *classes* and *properties* (see for example §3.1), as opposed to *individuals*. While description logics do have the ability to talk about individuals and first-order logic has the ability to talk about classes and properties, the difference is more in the way that the language is designed to be used and what aspects it focuses on.

Description logics are talked about in plural because there are many of them, with differences mostly in their expressivity and also possibly some differences in their syntax. There are often multiple alternative syntaxes for the same description logic, such as is the case with OWL (§3.1).

3.3 Constraint programming

Constraint programming is a broad term used to describe approaches to solving constraint problems by encoding them in programming languages specifically designed to this end (either specific types of constraints or more general families of constraint problems). While a lot or most of these approaches have a clear and explicit relation to logic, technically constraint programming encompasses broader approaches than those typically catalogued as logic.

The general approach of constraint programming approaches is to provide a language which is fundamentally *declarative* and aimed at expressing constraints, has very specific *semantics* and, usually, has some *computational* elements that allow the programmer to guide the search process to the problem.

SAT and SMT (§3.2.3) are both closely related to the internal execution aspects and the semantics of many constraint programming approaches, and can be considered a constraint programming approach in themselves. In this section, we consider two additional families of constraint programming approaches: Logic programming and Answer Set Programming.

3.3.1 Logic programming

Logic programming is heavily based on first-order logic and resolution theorem proving (§3.2.1.2). Prolog [Sterling and Shapiro, 1994b] is the principal and first logic programming language, and Prolog programs are declarative sets of *clauses* of two types:

- **Facts** - Unconditional statements that are defined to be "true" or "satisfied" by the program. For example, `pizza(X).`
- **Rules** - Conditional statements that have a *head* and a *body*. The head represents the statement that can be concluded, while the body indicates a set of conditions that need to be satisfied for the head to be satisfied. For example, `cheesyPizza(X):- pizza(X), hasTopping(X,Y), cheese(Y).`

Facts and rules are technically two particular cases of the same logical construct, which corresponds exactly to that of a *Horn clause*. They have *variables* in them, that can appear both in the body and the head of rules. This allows rules to represent conditional statements that are true of only those elements of the universe of discourse for which the conditions hold.

Computationally speaking, Prolog is a *goal-oriented refutation procedure*. This means that the way to run a Prolog program is to input a *query* or *goal* (which may include free variables), and Prolog systematically tries to match the goal(s) with facts or the head of rules. When a rule matches, the associated conditions are added as new goals and the process continues. In the matching procedure, *unification* (see §3.2.1.2) is utilized with respect to the variables.

The main difference between Prolog and resolution automated theorem provers is that automated theorem provers will usually try to do a full and fair exploration of the search space of potential applications of the resolution rule, and use heuristics to guide this search; whereas Prolog strictly defines the search order as a consequence of the way the program is written, with additional computational utilities (for example, *cuts*) used to give the programmer additional control over this. Automated theorem provers will also often not be limited to Horn clauses. Practically speaking, Prolog is regarded as a programming language rather than a theorem prover, and

is used to implement solutions to technical problems rather than to prove mathematical theorems, though the problems used for Prolog are often mathematical in nature.

3.3.1.1 Extensions of Prolog

There are more logic programming approaches than Prolog. We will briefly discuss two families of these.

Higher-order logic programming, and in particular *Lambda Prolog* [Miller, 2021, Nadathur and Miller, 1988], extends the notions of applying resolution theorem proving to create a declarative programming language to higher-order languages. The principal (and large) difficulty in this extension is the issue (discussed in §3.2.2) of *higher-order unification* and its much less attractive computational properties (compared to first-order unification). Mainly due to this issue, Lambda Prolog is much less popular and extensively used than regular Prolog. It is, however, particularly relevant as closely related in scope and principles to some of the technical problems in this thesis. A technical aspect of particular relevance about Lambda Prolog is that, like Prolog, it is limited to Horn clauses, and this limitation is fundamental to its implementation and algorithmical aspects.

Another related topic is *Constraint Logic Programming* [Jaffar and Lassez, 1987]. Constraint Logic Programming fundamentally understands that the declarative style and computational approach of Prolog can be applied to more general constraint solving problems and approaches than first-order resolution. At a basic level, the following extensions of the pieces of Prolog are made:

- *Facts* are replaced by the more general notion of *constraints*. They still represent unconditional elements that have to hold in any *solution* to the problem.
- *Rules* are preserved but their semantics are generalized. They still represent a connection between the body and the head, but in a more general way. A rule `head :- c1, c2.` indicates that in order for the constraint `head` to be satisfied, the constraints `c1` and `c2` need to be satisfied in the solution.
- *Unification* as the fundamental constraint solving process is replaced with a general abstract notion of any constraint solving process that relates the head and

the body of rules.

- *Goals* are still utilized as a means to guide the search.

As can be derived from the way we described it, Constraint Logic Programming is not a singular approach, but rather a family of approaches with shared notions. The particularities of what the constraints represent and the particular constraint solving process utilized change the nature and specific semantics.

3.3.2 Answer Set Programming

Answer Set Programming (ASP) [Brewka et al., 2011, Balduccini, 2009] is a language for expressing (and automatically solving) constraint satisfaction problems that focuses more on providing an easy and intuitive way to express problems than on particularly strong automated resolution capabilities. In particular, basic ASP is fundamentally based on propositional logic (§3.2.3), with a declarative language conceptually very similar to Prolog, but with an underlying resolver based on DPLL [Nieuwenhuis et al., 2006].

The target problems of ASP are search problems in *finite* spaces in which the user may utilize the declarative language's capabilities to guide the resolution of the problem, similar to how Prolog would. It provides a tool where users can express their specific problems quickly and intuitively and get automated answers with theoretical guarantees, and avoid or limit the effect of computational pitfalls through conscious choices in the definition of the program.

ASP also has first-order-like syntax, though all variables are forall quantified, which together with its handling of negation means that the programs are finite in nature and can always be automatically reduced to propositional programs that the resolver then works through. Negation in ASP takes two forms:

- *Intuitionistic* negation (inability to prove a given atom)

- *Strong negation* (classical negation, a semantic relation in every model between an atom and its negation).

The limited expressivity of ASP means that strong negation is pragmatically equivalent to adding a new atom for the negative literal, with a few implicit rules that relate them. Thus it's both not very useful and not very complicated to handle.

There are some research efforts into adding expressivity to ASP, one step at a time. A relevant example of this for the purposes of this thesis is about the possibility of adding existential rules to ASP [Baget et al., 2018]. This, naturally, has important limitations and challenges that relate to known properties of logic and algorithmics. Therefore, ASP continues to not be an ideal choice for applications that rely heavily on these semantic aspects.

3.4 Rewrite systems

A good, more complete overview on *rewrite systems* can be found on [Huet and Oppen, 1980, Robinson and Voronkov, 2001]. Here we will cover some basic notions that relate to our usage of rewrite systems, mostly as a theoretical tool to show certain results in chapter 7.

Rewrite systems or Rewrite rule systems are a mathematical tool used to define and reason about formal systems in which we have elements that can be transform into equivalent ones, normally to *simplify* them in some way. Results in rewrite system topics typically include *confluence* and *termination* results, both of which we will discuss shortly.

A simple example can be polynomials with addition and multiplication, such as $(3+x)\times(x+9)$. Rewrite systems normally begin by defining a set of *direct reduction rules*. For example:

$$
\begin{aligned}
&a \times b \to a \cdot b | a,b \in \mathbb{R} \\
&x^n \times x^m \to x^{n+m} | n,m \in \mathbb{N} \\
&a \times x^n \to a x^n | a \in \mathbb{R}, n \in \mathbb{N} \\
&a x^n + b x^n \to (a+b) x^n | a,b \in \mathbb{R}, n \in \mathbb{N} \\
&(p+q) \times r \to p \times r + q \times r
\end{aligned}
$$

Note we have omitted rules for commutativity and associativity here, and instead assumed them implicitly. We can then produce transitive chains of reductions, in which all the elements in the chain are equivalent, but the latter ones are more simplified than previous ones. For example:

$$
\begin{aligned}
&(3+x)\times(x+9)\rightarrow\\
&\rightarrow 3\times(x+9)+x\times(x+9)\rightarrow\\
&\rightarrow 3\times x+3\times 9+x\times(x+9)\rightarrow\\
&\rightarrow 3x+3\times 9+x\times(x+9)\rightarrow\\
&\rightarrow 3x+27+x\times(x+9)\rightarrow\\
&\rightarrow 3x+27+x\times x+x\times 9\rightarrow\\
&\rightarrow 3x+27+x^2+x\times 9\rightarrow\\
&\rightarrow 3x+27+x^2+9x\rightarrow\\
&\rightarrow x^2+12x+27
\end{aligned}
$$

The question then, is, what if we had applied the rules in a different order; for example:

$$
\begin{aligned}
&(3+x)\times(x+9)\rightarrow\\
&\rightarrow 3\times(x+9)+x\times(x+9)\rightarrow\\
&\rightarrow 3\times x+3\times 9+x\times(x+9)\rightarrow\\
&\rightarrow 3\times x+3\times 9+x\times x+x\times 9\rightarrow\\
&\rightarrow 3\times x+3\times 9+x^2+x\times 9\rightarrow\\
&\rightarrow 3\times x+27+x^2+x\times 9\rightarrow\\
&\rightarrow 3x+27+x^2+x\times 9\rightarrow\\
&\rightarrow 3x+27+x^2+9x\rightarrow\\
&\rightarrow x^2+12x+27
\end{aligned}
$$

The fact that we get the same final result, or in fact, that we would have gotten the same result regardless of the order in which we applied the rules, regardless of the initial expression, is called *confluence* of the rewrite system.

Definition 3.4.1 (Confluence)**.** *A rewrite system is* confluent *if for any expression that can be reduced two different ways a and b, then it is the case that a and b can both be reduced to the same expression c.*

The following is an important result in rewrite systems relating to confluence and irreducibility:

Theorem 3.4.1 (Unique normal form)**.** *If a rewrite system is confluent, and an expression can be reduced to an irreducible expression, then that irreducible expression is unique (it is the only one it can be reduced to), and it is called a* normal form*.*

For example, $x^2 + 12x + 27$ is a normal form in the above rewrite system.

It makes sense to ask whether applying rewrite rules a different way would have made us never reach a normal form. This is what we call *termination* of the rewrite rule system. For example, the following simple rewrite system is not confluent:

$$A \to B$$
$$B \to A$$

because the expression A gets rewritten to B, which in turns gets rewritten to A and so on. To be precise, a rewrite system is *terminating* if every reduction chain is finite.

A constrained notion of confluence is that of *local confluence*, which asks the confluence property only for expressions a and b which are **direct** reductions of the original expression. Local confluence is much easier to prove than confluence, and in fact confluence is usually proven by proving local confluence and termination, and then using the following fundamental result:

Theorem 3.4.2 (Newman's lemma / Diamond lemma)**.** *A terminating rewrite system is confluent if and only if it is locally confluent.*

Well behaved rewrite systems like this allow us to prove a slightly stronger result than theorem 3.4.1:

Theorem 3.4.3 (Exists and unique normal form)**.** *If a rewrite system is terminating and confluent, then any expression can be reduced to a unique normal form.*

The main usefulness of this result is that in well behaved systems, we can decide whether any two expressions are equivalent by reducing each of them to a normal form, and then comparing their normal forms.

Corollary 3.4.1 (Equivalence of normal forms)**.** *In a terminating and confluent rewrite system, two expressions a and b are equivalent if and only if they have the same normal form.*

Proof. Each of a and b have their unique equivalent normal form a_N and b_N.

If a and b are equivalent, then so are a_N and b_N. But each of a_N and b_N have a unique equivalent normal form, and they are both normal forms, so they must be exactly the same.

In the other direction, if a and b have the same normal form N, then both a and b are equivalent to N, and thus are equivalent between them. □

3.4.1 Graph rewrite systems

In what we have described so far, and in fact in almost all the literature on rewrite systems, rewrite systems operate on either sequences of symbols ("strings") or *terms*, which involve a hierarchical (tree) structure, but are still ultimately linear (can be traversed in a linear manner while going over all relevant links between elements). In both of these cases, rewrite rules can be applied locally to subsequences/subterms/subtrees because the scope of a sequence, term or tree is always locally defined. For example, if we have a string $\alpha\beta\gamma$, and we want to apply a reduction $\beta \to \beta'$ locally, we can consider this to be the result of "taking β out" of the original string, replacing it with β', and then putting that back in, for the result $\alpha\beta'\gamma$. Similarly, if we have a term $\alpha(\beta,\gamma)$, we can reduce β locally irrespective of α and γ.

However, rewrite systems can also be considered for more complex data structures, but local application can be more thorny to define. In this thesis we do this: we define a rewrite system on *graphs*. The main concern with local application of rewrite rules in graphs is what to do with edges that connect to any of the nodes in the local sub-graph after replacement. All sequences have a beginning and an end, but not all graphs have the same number of nodes or a clear way to identify them between them.

One way to deal with this is to always define rewrite rules *globally* for graphs, meaning they can only be applied to whole graphs. Then, confluence and termination results about rewrite systems can be applied to them normally. We do not do exactly this in this thesis, instead opting for an intermediate approach in which we define the rules locally, but **explicitly indicate how to handle their context**.

In the *term graph rewriting* literature, similar formalisms have been considered, even though our approach is not directly based nor inspired in any of these. For comparison, we can establish the similarities and differences with some of these.

In general, term graph rewrite systems are defined over *hypergraphs*, meaning edges can have multiple sources and/or targets. For example, see [Habel and Plump, 1995, Plump, 2002]. This is also the case for us. Another, fundamentally equivalent approach is to use *labelled edges/nodes*, such as in [Barendregt et al., 1987, Dwork et al., 1984].

A subtle difference between our approach and all approaches we've found in the literature is that a term graph in the literature usually corresponds to one term, and these can, for example, be unified into a new graph, whereas in our approach the graph represents the unification problem (in fact, multiple unification problems) at once. At the level of a graph rewrite system, this difference is inconsequential, and it only has to do with the meta-level aspects of our particular algorithm with respect to unification. An interesting concept throughout, that while not directly adopted, may help understand some of the considerations in chapter 7 is that of *graph homomorphism* [Barendregt et al., 1987]. It has a basic relation to our notion of *solution preserving rule* (definition 7.4.1) in that it establishes semantic relations between different graphs.

An aspect of term graph rewriting, that in [Habel and Plump, 1995, Plump, 2002, Barendregt et al., 1987] is informally called "garbage collection", and relates to what in [Dwork et al., 1984] is called "propagation" and "transitivity", is the notion that the non-locality of graph rewriting means that the application of rewrite rules can have important effects on the state in which the global graph is left, and may need to be dealt with. One way to deal with this (that we adopted) is to have minor rewrite rules specifically designed to clean these aspects, and have a clear priority between the rules. We achieve this through most of our *prefactorizing* rules (see definition 7.5.3), which have the exact purpose of cleaning up the graph; the dependency graph normalization levels (§7.3) and the normalization preconditions on most of the rewrite rules of our system.

3.5 Termination / decidability and enumeration procedures

While we have discussed termination and decidability throughout this chapter without issue, this aspect is central enough in this thesis to provide some clear background as to our terminology. A thorough discussion on these topics can be found on any introductory book to theoretical computer science, such as [Sipser, 1996]. We will omit fundamental definitions here such as the definition of an *algorithm*.

Note, however, that it is common in theoretical computer science to reserve the word *algorithm* only for *terminating processes*. An algorithm is some computational process that terminates, computing some result from its inputs.

We say that a problem is *decidable* if there exists an algorithm that computes it. Similarly, a *decision problem* (that is, a problem with a positive or negative result) is *semi-decidable* if there is a computational process that terminates if and only if the input produces a positive result[2].

There is less concern in conventional literature, however, with the problem of *enumerating* infinite sets. Similar concerns appear sometimes in the context of computing with real numbers or when considering non-deterministic processes, but these often have a different scope and perspective than the one followed in this thesis.

In our case, we consider problems whose answer is an infinite (countable) set, and extend the usual definition of *algorithm* to include non-terminating processes that, however, produce iterative and complete (fair) results (enumerate the infinite set). Since most of these results are not exactly standard, we produce them in chapter 7. However, there are a couple of basic notions that would be relevant for the reader to understand thus far.

Definition 3.5.1 (Enumeration procedure)**.** *An* enumeration procedure *E for a set A is a computational process that keeps producing results $a \in A$, and such that every $a \in A$ is eventually produced after a finite time by E.*

Note that this does not mean that E terminates at all, or that there is necessarily ever a moment where all $a \in A$ have been produced: while each individual a is produced after finite time, if A is infinite, then E will necessarily never completely enumerate it.

Second, consider that given any two enumeration procedures E_1 and E_2 for sets A_1 and A_2 respectively, it is easy to produce an enumeration procedure $E_{1\cup 2}$ that enumerates A_1 and A_2, even when they are both infinite. Of course, naively enumerating A_1 first is problematic because A_2 never gets enumerated. But we can enumerate both at the same time by producing one result from each set at a time (this is often called *interleaving*).

For example, consider the set $A_3 = \{3,6,9,12,...\}$ of all multiples of 3, and consider the set $A_5 = \{5,10,15,20,...\}$ of all multiples of 5, and respective enumeration procedures A_3 and A_5 that enumerate them in ascending order. Then, we can

[2]Note that for most semi-decidable problems, the reasonable procedure will also terminate on some negative results, but it is always possible to make this match the theoretical definition by forcing the procedure into an infinite loop when the negative result is found. In particular, every decidable problem is also semi-decidable.

produce an *interleaving* procedure $A_{3\vee5}$ that enumerates all multiples of 3 or 5 by alternating which of A_3 or A_5 we pick next; effectively producing the enumeration $\{3,5,6,10,9,15,12,20,...\}$.

Finally, this can be extended to a more general situation: Given an enumeration procedure E_0 for a set A, and one parametric enumeration procedure E_a that enumerates a set B_a for each $a \in A$, it is possible to build a single enumeration procedure E_d (we will often call this the *diagonalization*) that enumerates the set $\bigcup_{a\in A} B_a$. The specifics are tedious to describe, but entirely analogous to the usual proof that $\mathbb{Q}$ and $\mathbb{N}$ are equipotent sets; a construction that can be implemented in a general algorithm that works with arbitrary enumeration procedures. This result can be chained multiple times to essentially combine infinite sets of infinite sets any number of times, while keeping the enumeration sound and complete.

We use this approach extensively in our work.

Chapter 4

Faults as patterns

The basic idea of using patterns of some kind to detect errors or faults is not novel at all. This comes from the idea that an ontology (like any other formal system) is mostly correct and errors and faults are likely to be isolated instances "hidden" underneath everything else.

A brief discussion about wording would be adequate. Throughout this thesis, we use the term "fault", as opposed to other options such as "error", "pitfall", "bad smell", "problem", "warning" or "mistake". While none of these terms have a formal and/or clearly agreed upon definition, to the best of my knowledge, they usually have slightly different connotations. On one hand, "error" seems to indicate that some aspect of the definition of the ontology is necessarily incorrect. "Fault" is more generic in that it also conceptually (and in practice, as described in this chapter) includes situations that are not necessarily incorrect but rather suboptimal or potentially problematic in the future. The word "mistake" carries a similar connotation to "error", except it additionally assumes that the error is the product of a simple human issue during the definition of the ontology (a mistake). "Pitfall" is used in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016, Poveda-Villalón et al., 2012] and has the connotation of referring to a particular thought pattern or human mistake that leads to the definition of the ontology in a suboptimal way. In some ways, "pitfall" is to "fault" as "mistake" is to "error". This is arguably in part because of the focus in the cited work on design patterns to be applied *by humans* during the definition of the ontology to avoid these pitfalls (even if they do incorporate some level of automated methods of detection, which are aimed primarily at pointing them out to the ontology developer). Our focus is broader, trying to include any fault, regardless of the source or what we will want to do with it. "Bad smell" is probably

closest to what we do here, but this terminology is a bit more specific and (to our knowledge) not used in the ontology debugging literature, instead being relegated to the software debugging literature (such as [Fowler, 1999]). Similarly, "bad smell" would be more adequate to refer to our patterns than to the *faults* that we detect with them, since the bad smell is a sign of a fault, not a fault in itself. "Warning" has a similar situation and would probably be a suitable nomenclature for our patterns (or the result of their application), but not for what they are trying to detect: *faults*.

All of this said, these definitions are small and conceptually the differences are not big between these concepts. The main reason we use the "fault detection" terminology is to try to establish the particular perspective from which we look at it: that there are patterns that are suboptimal or potential problematic in the ontology, and we are trying to detect them.

A large portion of the literature on ontology debugging falls in one way or another into this abstract and general concept (see chapter 2), and so does related work like bad smells in software engineering [Fowler, 1999].

The interesting questions are:

1. What does a fault pattern look like?
2. How do you detect a pattern?
3. What are good patterns to use?

We answer the first and third questions in this chapter, and the second one in chapter 5.

4.1 Semantic patterns

Part of the issue with faults in logical ontologies is their ability to quickly spread through inference due to the high expressivity of logic. For example, the following set of statements, paraphrased and slightly modified from the original example on [Rector et al., 2004] (and which can be seen in more detail in appendix A):

- Toppings are things you put on pizzas.

- Chocolate ice creams are ice creams that have chocolate toppings.
- Chocolate toppings are toppings.

leads to the conclusion that chocolate ice creams are pizzas. This is a simple inference but the point is there is no statement explicitly talking about ice creams and pizzas at the same time, yet the fault arises as such.

However, we can embrace this property of logical inference in our pattern detection framework, by considering *semantic patterns* of *entailed* situations rather then explicit patterns (e.g. syntactic). In the example above, a semantic pattern that matches any situation where two primitive classes subsume[1] another primitive class and do not subsume each other, would match this and identify the fundamentally problematic situation, even if it does not correspond to any of the explicit axioms in the ontology. In particular, pizza, ice cream and chocolate ice cream would likely be primitive classes, and chocolate ice cream is subsumed by the two others.

Following this notion, it makes sense to express the semantic patterns as some form of extension of first-order logical formulas that represent the structure of the situation we are looking for while leaving some room for interpretation (variables) that can match multiple situations. We talk more about this in §4.3.

4.2 Reasoning outside of the box

Logics are an extremely expressive formalism, in which most other formalisms can be embedded. More precisely, logical inference is one of the most far reaching kinds of inference. When using the same base knowledge, logical inference has the potential to produce more conclusions than, for example, taking the knowledge base to be a simple list of facts, strictly causal reasoning, etc.

The reason this is relevant is that, if we intend to use exclusively the information strictly semantically entailed by the original ontology, then any conclusions we would draw would in some way already be entailed by the ontology itself. With (small) caveats, what we are saying is that *every conclusion that could be drawn, through any means of*

[1] Whenever we discuss subsumption in this chapter, we refer to *inferred* subsumption unless explicitly stated that we are talking about *asserted* subsumption.

inference, exclusively from the semantics of the ontology, is already an implicit part of the ontology.

We thus need to incorporate *additional information* to be able to detect faults other than the direct inconsistency of the ontology.

A large source of additional information are the patterns themselves. This is another of the reasons why using patterns for fault detection makes sense. The patterns are supposed to be general knowledge of situations that are not necessarily inconsistent but which often signal faults; knowledge that is not semantically contained in the original ontology.

However, there are other important sources of additional information that we can and indeed do use. In the example from the previous section, we used the notion of classes being *primitive* to identify the situation. However, and as described in §3.1, the notion of a class in OWL being primitive or defined is one about the way the class is expressed, rather than about the mathematical semantics of the class. Similarly, we may want to use the notion of an axiom being expressed explicitly in a particular way, or a class in OWL being *defined*, or the existence of a *domain* or *range* axiom in a class, just to name a few examples. This information is part of the ontology in terms of its explicit representation, but not of its semantics, and thus cannot be obtained merely through logical inference.

Our framework incorporates the notion of *contextual knowledge*, an explicit representation of specific facts about how the ontology is expressed and other information about its context that may be relevant to detect faults. In many cases, this corresponds to explicit declarations about the *syntactic structure* of the ontology (for example, as with the primitive and defined classes discussed earlier). Note that this contextual information will often be dependent on the specifics of the ontology, such as its formalism, methodological context, purpose, etc. As such, our framework does not specify how to obtain this contextual information in general, but all the contextual knowledge used in patterns that we have collected is very simple to extract from the ontologies with little more (and sometimes less) than a simple scan of the explicit representation of the ontology. In the pattern catalogue in appendix A, the specific contextual knowledge utilised in each pattern and its meaning are explicitly discussed.

4.3 Existential second-order query logic

As explained in §4.1, we want to use a logical formalism to represent patterns so that we can embrace the entailment capabilities of the ontology itself.

The main reason that we need to use second-order logic to represent these has to do with the desired generality of the patterns. A pattern that only matches an individual fault in an individual ontology is both unrealistic to have (it would require us knowing this ontology beforehand) and incredibly useless (only works for this situation). In other words, in order for something to be a pattern, it generally involves some form of *variability*. However, this variability usually comes in the elements of the language of the ontology itself (for example, the names of the predicates / classes). Thus, first-order variables are unsuited for the task. Second-order variables, on the other hand, can be used as "wildcards" in patterns. We can indicate that any entailed formula, in the combination of the ontology with the contextual information, that matches the pattern by replacing the second-order variables with *some* set of predicates or functions, is an instance of the pattern. Furthermore, the specific replacement of second-order variables (we call this the *instantiation* of the pattern) for specific predicates and functions is a very adequate way to express to the user what the fault may be.

For instance, we could produce a formula with second-order variables A, B and C (class / predicate variables), indicating that if A, B and C are all primitive, A and B both subsume C but neither A nor B subsume each other, then a fault is likely. In such a case, the specific instantiation of the variables A, B and C would indicate the primitive classes that produce this issue, producing a great way to describe the fault.

However, we are not embracing all capabilities and expressivity of second-order logic, by far. Instead, the only true element of second-order logic that our meta-ontologies contain are second-order variables in patterns to indicate source of variability. Instantiations of second-order variables used as wildcards will be compositions of first-order functions and predicates. This is described in a lot more detail in chapters 6 and 7.

A relevant notion about the formal aspect of this usage of second-order variables is that finding all instantiations of patterns with second-order variables is equivalent to finding all constructive proofs of the existentially quantified formula; which in a theory containing no axioms with quantified second-order variables, is equivalent to finding all proofs of the existentially quantified formula, as every proof will be constructive[2]

[2] further discussed in chapter 5

(each proof will correspond to a family of instantiations of the variables that make the resulting first-order formula provable). For example, consider the theory:

$$\forall x.p(f(x))$$
$$\forall x.p(x) \implies q(x)$$

and the pattern

$$\forall x.P(x)$$

where P is a second-order predicate variable. Two instantiations of this pattern are $P \equiv p \circ f$ and $P \equiv q \circ f$. Each of these instantiations corresponds to one proof of the second-order existentially quantified formula $\exists P.\forall x.P(x)$.

Due to the fact that all second-order variables appearing in our patterns will be conceptually existentially quantified, we call our logic *existential*.

It is important to note, however, that we are not only interested in finding that the formula $\exists P.\forall x.P(x)$ is provable or not. Instead, we wish to **explicitly** find **every** instantiation of P for which the pattern $\forall x.P(x)$ holds. Indeed, a pattern is an indication of a potential fault, but there may be more than one in an ontology or some indications may be false alarms while others aren't. In debugging terms, fault patterns indicate *warnings* rather than necessarily *errors*. Moreover, the user needs to check which of these correspond to actual faults and which do not, which is achieved by finding all explicit instantiations of the patterns. Thus, completion in the search is fundamental.

We will write this with the notation:

$$P \models \forall x.P(x)$$

which should be read as "The P's for which $\forall x.P(x)$ is entailed", and which is not a formula that is entailed or not entailed, but rather, it is a *query* with a result set (all the P's that entail $\forall x.P(x)$).

One additional note regarding non-first-order elements has to do with the aforementioned *contextual knowledge*. For example, consider how we could express that a class (a first-order predicate) p is *primitive*. The most natural way would be to have a statement of the form $primitive(p)$. What is *primitive* here? We could represent these

as second-order predicates / functions, but[3] this would then make any second-order expression that is equivalent to p to also be primitive (through extensionality). Moreover, we note that (as explained before) contextual knowledge is extracted from the ontology through some additional process and assumed to be simple, and that patterns do not include second-order operations on the contextual knowledge predicates themselves. Thus, it is much more adequate to represent these as *meta-predicates* than second-order predicates. This requires that statements of contextual knowledge are always stated and processed separate from statements in the underlying existential second-order query logic, which is not at all a problem in all the examples and patterns that we have compiled.

Thus, our language contains two types of statements:

- First-order statements with existentially quantified second-order variables.
- Meta-statements with simple meta-predicates applied to first-order functions, predicates and second-order variables standing for these.

These two types of statements are always kept separate and processed separately. Unification of meta-statements corresponds to a simplified version of first-order unification and we do not even discuss it any further in this thesis. Both of these statements may appear in the described queries. The separation of the meta-predicates exclusively in meta-statements ensures that the syntatic type of each element is always clear and thus we do not require the usage of quotes.

The specific formalism for representing these queries is what we call *existential second-order query logic* (or ESQ logic for short), and is described in more detail in chapter 6. We have, however, explained the adjectives “existential”, “second-order” and “query”.

We can explain the “computational” adjective as well. In the course of exploring existing research on detecting faults in ontologies and identifying patterns, we realized that in a lot of situations, there is a great benefit (performance, presentational, intuitive, etc.) in expressing patterns as a *prescriptive* computation rather than a merely *declarative* expression. Thus, the queries that we use to represent our patterns are not comprised of individual formulas, but rather combinations of these, each with their own

[3] as pointed out by Paul Jackson, internal examiner of this thesis, during the viva and in discussions afterwards

result set that is reutilized by the other queries in specific ways. This is also explained in more detail in chapter 6.

An additional reason to consider separate queries that are combined in an explicit way is that some patterns may try to find *entailed* formulas while others may try to find *satisfied* formulas, and we may want to combine them.

We will present here an example of a full ESQ logic query. Consider the chocolate ice cream example described above and the pattern "two primitive classes that subsume another primitive class but such that they do not subsume each other". We will express this as the following combined query:

$$\begin{aligned}&((P,Q,R) \models^{*} (\exists X.P(X) \wedge \neg Q(X)) \wedge (\exists X.Q(X) \wedge \neg P(X))) \bowtie \\ &\bowtie ((P,Q,R) \models (\forall X.R(X) \implies P(X)) \wedge (\forall X.R(X) \implies Q(X))) \bowtie \\ &\bowtie ((P,Q,R) \models_{M} \mathit{primitive}(P) \wedge \mathit{primitive}(Q) \wedge \mathit{primitive}(R))\end{aligned}$$

which first (read from right to left like function evaluation) considers combinations of classes that are all primitive, checks whether two of them subsume the third one, and then checks whether it is possible that those two classes do not subsume each other. This streamlining aids both with expressing the query succinctly, controlling the branching factor of the algorithmical search for solutions, and combine different types of queries (entailment and satisfiability).

A complete theoretical definition of ESQ logic and discussion about its computational aspects can be found in §6.2.

4.4 The meta-ontology fault detection framework

All of what has been described so far in this chapter comes together into the *meta-ontology fault detection framework* to automatically detect faults in an ontology O:

1. Translate / ensure O is in the adequate formalism[4]
2. Extract contextual knowledge $\mathcal{C}$ from the ontology and encode it.
3. Combine O, $\mathcal{C}$ and the fault patterns into a second-order ontology (which we often also call the *meta-ontology*).

[4] In our work, this is first-order logic, but the abstract framework could potentially be used with other formalisms, such as OWL or other description logics.

4. Apply the detection mechanism to find instantiations of the patterns.

The reason we call it *meta-ontology fault detection* is that the combination of O, the contextual knowledge and the fault patterns is technically also an ontology in the more general formalism required to detect patterns that has O embedded into it, and the detection of patterns conceptually consists in performing inference within this formalism.

This approach has a couple of important attractive properties, some of which we have already mentioned:

- The pieces are independent and interchangeable. The patterns, the method to extract the contextual knowledge, the specific algorithm to detect patterns and the ontology O itself are not necessarily coupled with each other. It is true, however, that often the contextual knowledge needs to have some level of compatibility with the patterns, and so does the algorithm with the patterns. From a conceptual point of view though, they are clearly separate pieces, which allows for easier improvements on just some parts of the process.

  For example, the same set of patterns is likely to be applicable to many, if not all ontologies conceivable.

- By embedding O into a meta-ontology in an adequate way, we ensure that it only combines with the contextual knowledge and the patterns in exactly the intended ways. Language overlaps are not an issue (e.g. O contains only first-order statements using first-order predicates, but contextual knowledge statements are normally second-order statements using second-order predicates).

- The use of contextual knowledge gives a lot more power to what the patterns may encode. This allows encoding much more generic patterns into the framework.

- The computational nature of patterns allows for pragmatic concerns to be encoded and partly dealt with within them. For example, by using the more constraining, less computationally expensive queries first, as a filter for the search space of the more expensive queries.

4.5 Pattern catalogue

Apart from defining the framework, we have carried out a task of examining a portion of the literature on ontology debugging and interpreting the ideas contained in it as specific ESQ logic queries that encode fault patterns for our framework.

This task has been quite successful, showing the expressive power of our approach and how it may be a way to generalize and systematize a lot of the ideas in ontology debugging.

In this section we include two examples of patterns extracted from the literature. The remainder (18 patterns in total) is too long to be included in the body of the thesis, but is included in the appendix A. In chapters 9 and C, among other things, we examine the level of success of the process of encoding patterns as ESQ queries.

For each example, the following information is given:

- Description of the example.
- Reasoning behind considering it a fault.
- Formalism in which the example is originally represented. For example, OWL, first-order logic, etc.
- Conceptual source of the fault. For example, an inadequate blend of ontologies, an imprecision in a natural language processing technique or a misconception by an ontology designer.
- Specific source of the example. A reference to an existing ontology where it was found, a paper where it was mentioned, etc.
- Detection strategy
- (Optional) Repair suggestions. These are always quite informal and non-systematic, but they may still be quite useful in many cases. We do not explore repair mechanisms in this thesis, but it is an obvious avenue of future work and when compiling the pattern catalogue it made sense to make some notes about it.
- Formal fault pattern. Expressed using ESQ logic.

- Additional contextual information that is used at the meta-level for defining the patterns.

As an additional note regarding formulation: take into account that whenever the present tense is used to make a statement (for example, "a chocolate ice-cream is not a pizza"), we mean that *in the preferred model*[5] a chocolate ice-cream is not a pizza, whereas the faulty ontology might entail so.

The following are two representative examples from the catalogue:

4.5.1 Fault pattern 1: Assuming universal quantification implies existential quantification (Empty pepper pizza)

In [Rector et al., 2004], it is mentioned that one of the most common sources of errors for ontology designers unfamiliar with OWL is to assume that universal quantifiers imply existential quantifiers. A class which only has a universal quantifier as an axiom for a property may have that axiom satisfied trivially by having an empty range for that property.

As an example, consider the following definition of a PepperPizza:

class(PepperPizza complete
Pizza
restriction(hasTopping allValuesFrom PepperTopping))

which means that every pepper pizza is also a pizza and all of its toppings are pepper toppings, or, in first-order logic; $\forall x.PepperPizza(x) \implies (Pizza(x) \wedge \forall y.hasTopping(x,y) \implies PepperTopping(y))$.
And now consider the following definition of MargheritaPizza:

class(MargheritaPizza complete
Pizza
restriction(hasTopping allValuesFrom Nothing))

which implies that every margherita pizza is also a pizza and has no toppings. In first-order logic: $\forall x.MargheritaPizza(x) \implies (Pizza(x) \wedge \neg\exists y.hasTopping(x,y))$.

The ontology then entails that a margherita pizza is a pepper pizza, which is not conceptually what we refer to when we talk about pepper pizzas.

[5] In some sense, what we consider *to be really true.*

4.5.1.0.1 Why is it a fault

A margherita pizza is not a pepper pizza. A pepper pizza needs to have **some** pepper in it.

4.5.1.0.2 Formalism

OWL

4.5.1.0.3 Conceptual source of the fault

A failure, by the ontology designer, to express all conditions that define a PepperPizza. An assumption that universal quantification entails existential quantification.

4.5.1.0.4 Specific source of the example

I have concocted this example based on the ideas expressed in [Rector et al., 2004].

4.5.1.0.5 Detection strategy

A good approach is to check if there is any class subsumption (PepperPizza subsumes MargheritaPizza) which is *only* enabled by a trivial satisfaction of a universal quantifier in the definition of PepperPizza. While there may be some legitimate cases where this kind of subsumption is correct (for example, a MargheritaPizza is arguably a VegetarianPizza), it is a good start to signal these cases.

An important note should be made here. If tomato and mozzarella were considered toppings in the ontology, then this fault would be harder to detect, at least with the approach described here. While having no toppings is an extreme case that seems easier to flag, having no toppings except mozzarella or tomato (or any other kind of composite predicate like that) seems too broad to signal as a fault. For example, if we had a concept PepperPizza which could have green, red or yellow peppers and then a RedPepperPizza that could have only red peppers, it would be foolish to suggest that it is a fault that the subsumption between RedPepperPizza and PepperPizza is spurious: it is not. There is no structural difference, without lexical knowledge about what pepper pizzas are, that differentiates this from the Margherita pizza being subsumed by Pepper pizza.

4.5.1.0.6 Repair suggestions

A sensible suggestion is to explicitly remove the trivial case from the definition of the class. This is easily done by adding a condition that there are at least some toppings in pepper pizzas.

4.5.1.0.7 Fault pattern

To put the idea described previously more precisely, we are looking for classes A (Pizza), B (MargheritaPizza) and C (PepperPizza), property R (hasTopping) and class P (PepperTopping) such that:

1. C subsumes B. We can use the previously defined macro $subsumes(C,B)$.

2. C has a universal property restriction to it. We require contextual information to identify this as being something explicitly indicated in the ontology[6]. $class\_property\_restriction(C,R,P)$. There is a class property restriction indicating that for every x of class C (every pepper pizza), it is related through relation R only to elements y of class P (it only has pepper toppings).

3. All instances of B have that universal property restriction fulfilled trivially. $\forall x.B(x) \implies \neg\exists y.R(x,y)$, which is read as, for every element of class B, there is no element to which it is related through property R.

A key difference with this case, and one that showcases the usefulness of the instantiation set approach as compared to more ad-hoc procedures, is that the subsumed class (X) is not constrained by the domain constraint, and so any class may work there. However, we can still leverage the computational properties of the rest of the pattern to ideally find instances quick, using equational reasoning while keeping account of what we are limited to.

The resulting pattern would be:

$$\begin{aligned}&((X,Y,R) \vDash (\forall x.X(x) \implies \neg\exists y.R(x,y)) \wedge (\forall x.X(x) \implies Y(x))) \bowtie \\ &\bowtie ((Y,R) \vDash_M class\_property\_restriction(Y,R,P))\end{aligned} \tag{4.1}$$

[6]We could find any entailed universal property restriction for a class, but that would make the purpose of the pattern moot as we are looking for definitions that are not used, in some sense, not for entailments that are not used.

4.5.1.0.8 Related contextual information

We have introduced the new contextual predicate *class_property_restriction* to express that something is expressed as a class property restriction explicitly in the OWL ontology. It is fundamental for the intuition behind this fault pattern.

4.5.2 Fault pattern example 2: Missing domain or range properties

Pitfall P11 in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016], properties which have no domain or range constraints are regarded as prone to faults.

For example, consider the following ontology:

class(Writer)
class(LiteraryWork)
ObjectProperty(writesLiteraryWork)

which simply states there are two unary predicates (classes) *Writer* and *LiteraryWork* and a binary property *writesLiteraryWork*.

4.5.2.0.1 Why is it a fault

Without domain or range constraints, *writesLiteraryWork* could in theory apply to any object in the semantics. This has many unwanted consequences, such as the inability to infer useful theorems about the property because its wide domain and range mean that strange situations can appear, the potential for the property itself to interfere with the inference of other parts of the ontology for the same reason, and all the secondary consequences these may have.

4.5.2.0.2 Formalism

OWL

4.5.2.0.3 Conceptual source of the fault

Most likely the author forgot to include the domain and range constraints.

4.5.2.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

4.5.2.0.5 Detection strategy

It is in principle easy to detect. Look for explicit properties in the ontology that have no domain or range constraints.

It is very interesting that in [Poveda-Villalón et al., 2012], the authors acknowledge that their detection mechanism is limited in the sense that it will not count inherited domain or range constraints for these purpose. Our approach in principle goes even further to consider any inferred domain or range constraints as valid, enabling the author of the ontology to rely on reasoning to provide these, and meaning that our automated detection mechanism is even more useful for working with these situations while still having a way to check whether we forgot to provide them.

However, because we are looking for properties that *do not have* range or domain constraints, we are not finding an instantiation of these domain or range constraints that makes them provable, but rather, that are satisfiable.

4.5.2.0.6 Repair suggestions

It is very hard to provide useful repair suggestions for this fault pattern, as that would imply having knowledge about what the domain or range of the property should be.

4.5.2.0.7 Fault pattern

$$\begin{gathered}((P) \models^* \forall x.\exists y.P(x,y)) \bowtie ((P) \models_M explicit\_property(P)) \\ ((P) \models^* \forall x.\exists y.P(y,x)) \bowtie ((P) \models_M explicit\_property(P))\end{gathered} \tag{4.2}$$

This last example involves a *satisfiability* check rather than an entailment check. Indeed, a domain or range constraint involves an axiom entailing that the property only applies to some elements. The lack of such an axiom involves that the limitation is *not entailed*. This is not the same as the negation being entailed, as there could be models of theories in which a property has no domain or range properties, but not every instance in that model is related through the property (in fact, this would be true in most cases). For example, only people legally own property; but some models of a theory without this domain constraint may in fact be such that every element that owns another is a person. It is not entailed that every element *must* own property.

Satisfiability queries are discussed in more detail in §6.2.

4.5.2.0.8 Related contextual information

We have introduced the predicate *explicit_property* to indicate that a property is explicitly defined in the ontology.

4.6 Summary

In this chapter we have explained in detail two of the three main contributions of this thesis: The meta-ontology fault detection framework and the pattern catalogue. The meta-ontology fault detection framework uses fault patterns encoded as logical queries using a meta-ontology containing the original ontology plus contextual knowledge about how the ontology was expressed to automatically detect instantiations of those patterns.

We call the language that we use to describe the fault patterns *existential second-order query logic* (ESQ logic). It is logic because we want to consider entailment / satisfiability when checking for a pattern match, as opposed to explicit or simple syntactic checks. It is a query language because the answers are not yes or no, but rather sets of instantiations of the pattern. It is existential second-order because we need some second-order expressivity capabilities to properly describe our patterns and the contextual knowledge, but only to a certain degree (second-order variables are only existentially quantified).

One of the main attractive aspects of meta-ontology fault detection is that the ontology, the methods to extract the contextual knowledge, the patterns themselves, and the algorithm to detect the patterns in the ontology are, to a degree, independent of each other, and so each of them can be modified or exchanged for a different one.

The pattern catalogue is a set of patterns expressed in ESQ logic that we have collected from multiple sources in the literature on ontology debugging. We have understood the contexts and rationales behind these fault patterns and encoded them in the common framework that we have developed. This has been quite a successful task, showing that a majority of the nuances of these approaches can be effectively expressed in ESQ logic.

Chapter 5

Automatic detection of patterns

In chapter 4, we introduced the *meta-ontology fault detection* framework and the *ESQ logic* formalism for expressing patterns of faults within this framework.

We did not, however, specify the algorithmical process by which, from an ontology and contextual knowledge associated with it, and a formal description of a pattern, one **produces** an enumeration of instantiations of this pattern: *how does one use the patterns to find faults?*

This chapter deals with this aspect of the problem in detail, and also contains the informal description of the third (and perhaps largest) contribution of this thesis: *minimal commitment resolution for ESQ logic*, an algorithm and background theory that we have developed precisely to solve the detection problem in the most satisfactory way.

5.1 Automated theorem proving

A fundamental point that needs to be clearly established is the following: Finding instantiations of ESQ logic queries in a logical theory *is an automated theorem proving task*. As described in §4.1, ESQ logic queries are **semantic** patterns encoding sought **entailments**. Thus, in order to find their instantiations, we need to consider the full length of the entailments of the ontology.

Let us look at the example presented in §4.3. Consider the theory:

$$\forall x.p(f(x))$$
$$\forall x.p(x) \implies q(x)$$

and the logical query

$$P \vDash \forall x.P(x)$$

In order to find the two desired instantiations of this pattern ($P \equiv p \circ f$ and $P \equiv q \circ f$), it is necessary to produce *proofs* of each of these statements. Specifically, $P \equiv p \circ f$ is associated with a proof by an axiom:

$$\forall x.p(f(x))$$

while for $P \equiv q \circ f$, we need a one-step proof:

$$(\forall x.p(f(x)), \forall x.p(x) \implies q(x)) \\ \vdash \forall x.q(f(x))$$

Therefore, it makes sense to (and we do) use all of the existing knowledge, techniques and approaches of the automated theorem proving and constraint solving fields and subfields. However, we will argue that conventional approaches to these are not directly suited to directly deal with this problem. This is the reason why we have developed our algorithm, which however is heavily based on previously existing theory and approaches.

5.2 Brute force search

In this section we describe a simple, yet ineffective, approach to finding instantiations of ESQ logic queries, based on **reducing the problem to first-order theorem proving**.

Before we proceed to the meat of this section, let us briefly remind that meta-statements in the logic are solved separately from pure ESQ statements, and that this process is a simplified version of first-order unification. Thus, the only obstacle to reducing ESQ logic queries to first-order theorem proving are second-order variables.

The brute force approach then, considers a formula[1] that is part of a ESQ logic query, that contains second-order variables, and reduces finding its instantiations to the following process:

1. Consider the set of all functions and predicates present in the signature.

[1] From now on we use a single formula to represent the entire theory, since the theory can be presented as a conjunction, and then implication and/or negation can be used to join this with the potential theorem. See §3.2.1.

2. Extend this set to the (in general, infinite) set of all compositions of functions and predicates in the signature (the second-order Herbrand universe, see §6.2).

3. For each free second-order variable, and each element in the second-order Herbrand universe, consider the formula that replaces each variable for each such element. The resulting formula contains no second-order variables.

4. For each such formula, use first-order theorem proving methods to check whether it is a theorem or not.

5. For those formulas that are provable, output the corresponding instantiation of second-order variables that was used.

In other words, try every potential instantiation of second-order variables, producing a first-order formula for each of them, and try to prove it. Let's see an example. Consider the theory:

$$\forall x.p(f(x))$$
$$\forall x.p(x) \implies q(x)$$

and the logical query

$$P \vDash \forall x.P(x)$$

The signature here contains two predicates: p and q and one function f. Thus, the (infinite) set of possible instantiations of P looks like this: $\{p, q, p \circ f, q \circ f, p \circ f \circ f, q \circ f \circ f, ...\}$. For each of those, the pattern in the query is a first-order formula. For example:

$$\forall x.p(x)$$
$$\forall x.q(x)$$
$$\forall x.p(f(x))$$
$$\forall x.q(f(x))$$
$$\forall x.p(f(f(x)))$$
$$\forall x.q(f(f(x)))$$
$$...$$

We can then individually apply first-order theorem proving techniques to each such formula to attempt to prove them. Some will be provable (for example, $\forall x.p(f(x))$, and

others will not (for example, $\forall x.p(x)$).

While this approach works in theory, it has horrendous algorithmical properties. First of all, note that each attempted proof of a first-order formula is a semi-decidable and potentially intractable problem. These proofs would thus need to be run in parallel or diagonalized (see §3.5). Second, the size of this set of formulas is infinite as long as there is at least one function (that is not 0-ary) in the signature. But worse still, the rate of growth of this space as the depth of the terms increases is *exponential* on the size of the signature.

This becomes even more abhorrent in the face of realizing facts such as that any proof of $\forall x.p(f(x))$ is automatically a proof of $\forall x.p(f(f(x)))$, but the converse is not true. Even when the proofs themselves are not strictly contained in each other, it is likely that a lot of the proof steps will be common, and repeating them for each individual attempt at a first-order proof seems entirely unnecessary.

These properties are why the brute force approach is unconscionable in practice; and when considering the seemingly natural idea of doing common parts of the proof first and only enumerating second-order variables whenever necessary, we arrive at the fundamental idea of *minimal commitment resolution for ESQ logic*, described in §5.5.

5.3 Technical challenges

There are fundamentally two reasons why conventional automated theorem proving and constraint solving approaches are not best suited to solve this problem: The *second-order* nature and the *query* nature of ESQ logic (see §4.3). In the following sections we discuss the general issues with these, while later on we consider specific potential approaches and the particular problems with each of them.

5.3.1 Second-order

A very large proportion of research and results in automated theorem proving is strictly constrained to first-order logic or more constrained formalisms (this includes propositional logic and description logics, among others). There are several good reasons for this:

- A large proportion of real world applications of automated reasoning can be

expressed in first-order logic or more constrained formalisms.

- While first-order logic has intractable and semi-decidable worst case computational properties, efficient algorithms and powerful heuristics exist that deal with most practically appearing situations within it.

- Moreover, a lot of practical applications of automated theorem proving can be and are expressed using more constrained formalisms like description logics that have even more attractive algorithmical properties.

- Second or higher-order automated theorem proving, while being possible and indeed there are well-known technologies that use it (see, for example, [Paulson, 1989]), has much more unattractive algorithmical properties that make its applicability much more limited.

Existential **second-order** logic is, as its name implies, a form of second-order logic, and thus research limited to strict first-order theorem proving cannot be directly applied to solve our problem.

An issue of higher-order logic is its immense expressivity. This severely affects limitations in existing general approaches to it, either by constraining the expressivity in specific ways (for example, Lambda Prolog (see §3.3.1.1), or by offering very little in the way of algorithmical guarantees (see §3.2.2). In ESQ logic, however, we are constrained to existentially quantified second-order variables (see definition 6.1.2 and following discussions). In this thesis, we attempt to leverage these limitations in expressivity to produce more efficient algorithms than those possible for general higher-order logic by cutting some corners.

All of this thus leaves us in a point where using tested first-order approaches is technically insufficient, but using conventional higher-order approaches is excessive in expressive power, and unsatisfactory in practical terms.

5.3.2 Queries

ESQ logic is a *query* logic. This was discussed in §4.3, but we present it here again. The answer to a ESQ logic query is **a set of instantiations** that satisfy the query, whereas the answer to a potential theorem in first or higher-order logic is **true or false**:

"is it a theorem, or not", or, at most, **a proof** of the theorem.

It is important to understand how this is not at all a trivial problem that can be overcome with a simple tweak of the existing algorithms or approaches. We will present this by comparison with two distinct alternatives:

5.3.2.1 ESQ queries and first-order theorem proving

The main element that distinguishes ESQ logic from first-order logic is the presence of second-order variables for which we wish to find instantiations. In other words, for each possible instantiation of second-order variables, the resulting query is a first-order logic formula (this is discussed in more detail in §5.2).

It is precisely the fact that queries *have many possible results* that prevents ESQ logic from being solved directly by first-order theorem proving methods. In other words, if the problem was simply to ascertain whether a given instantiation satisfies the formula or not, this would be exactly a first-order theorem proving problem. The presence of free second-order variables makes first-order unification inapplicable directly. It does make a lot of sense to adapt the ESQ logic problem to a variation / extension of the first-order problem, and this is what we discuss both in §5.2 and in §5.5, the latter of which is our implemented approach and one of the main contributions of this thesis. In other words, while first-order theorem proving is not enough, we have followed an approach of extending it only as much as necessary to solve our problem, treating the presence of second-order variables as a specific challenge to overcome (which it is, given that ESQ logic was always designed to find patterns **in first-order logic theories**).

5.3.2.2 ESQ queries and higher-order theorem proving

It is in fact theoretically possible to find instantiations of ESQ logic queries using traditional higher-order logic, but we should examine what this involves.

Because higher-order logic does include second-order variables as part of its language, we can directly encode the body of a ESQ query as a higher-order formula, and seek to use higher-order theorem proving to find instantiations. In order to attempt to prove a formula, however, it needs to be *closed*, meaning no free variables are left. We thus need to *quantify* the second-order variables from the ESQ query.

It is clear that universally quantifying these variables would not have the intended

semantics of finding instantiations of a query, and it is existentially quantifying them which properly represents our problem from a higher-order logic point of view. Consider the possible **proofs** of the existentially quantified formula. It is traditional to categorize proofs of existential formulas (that are not *valid*) into two families:

- **Non-constructive or classical proofs** - That is, proofs which only prove the existence of an instantiation, but do not provide any insight on what that instantiation may look like. Formally, these proofs can be traced back to *atoms with quantified variables*. For example, if every person has a father and we know of the existence of at least one person (but nothing else about them), then we can prove the existence of at least one father (but nothing else about who that father may be). These proofs are completely irrelevant for ESQ logic, because in ESQ logic we will never have *axioms* containing quantified second-order variables.
- **Constructive proofs** - That is, proofs that something exists that build it. These are exactly what we are looking for. Therefore, the following would be a fundamentally valid approach to finding instantiations of ESQ queries:
  1. Existentially quantify all free second-order variables in the ESQ query.
  2. Produce **all proofs** (they will all be constructive because we do not have second-order quantified variables in axioms) in higher-order logic of the existentially quantified formula.
  3. For each of those proofs, inspect the construction of the instantiation to produce an instantiation of the ESQ query. This is similar to the usage of *witnesses* in resolution proofs, such as, for example, logic programming [Sterling and Shapiro, 1994b].

To summarize: If the theory contains no quantified second-order variables and the part of the goal pattern containing second-order variables is not valid, then every proof of it will be constructive. For example, for a theorem $\exists P.P(a)$ to be provable from a theory that contains no second-order variables, the only possibility would be to find particular instantiations of P that satisfy $P(a)$. For example, if the theory contains axioms $p(a)$ and $\forall x.p(x) \implies q(x)$, then we can find instantiations $P \equiv p$ and $P \equiv q$, but every proof of the theorem will provide a witness for P.

Thus, a standard higher-order theorem prover could be used to find instantiations of the pattern by finding all such proofs and generating witnesses from them. The issue

with this method is double and compounded, even if in theory it is perfectly valid. First, the fact that the usual higher-order automated theorem proving methods are in general not computationally attractive and tailored to the particularities of ESQ logic. Second, and more importantly, the issue with finding **all proofs** of an existentially quantified formula. Usual higher-order automated theorem proving methods are not geared at all towards this goal, instead focusing on finding **one proof**.

This is extremely problematic in that a usual higher-order automated theorem proving method will omit entire spaces of proofs when it is known that an alternative, cheaper proof exists; and these omissions are fundamental for both the success and the algorithmical properties of the algorithm (see the issue with flex-flex pairs as discussed in §3.2.2). A direct application of usual higher-order theorem proving techniques to find instantiations of a ESQ query would result in even more issues than usual with intractability, non-termination and unfairness of the search process.

5.4 Utilizing existing approaches

In the following we discuss in more particular detail the issues with each of a series of candidate methods for solving our problem using existing approaches. We note that in general these are not issues that completely prevent the application of these approaches. Instead, they indicate large challenges either in terms of expressivity, computational feasibility or simply the fact that a significant development of additional original research would be needed for the approach to be applicable to our problem. Some of these may turn out to be productive avenues of research, and we try to outline the promises that they afford as much as their challenges.

In other words, the purpose of this section is not to say these approaches are hopeless, but rather to justify that they were not obvious solutions to our problem. I chose to develop the approach that I did based on this preliminary exploration and the realization that important issues would need to be faced in every possible approach. This was an informed and I feel justified decision, but certainly not the only one I could have taken.

5.4.1 Higher-order logic

We begin by discussing general higher-order logic or second-order logic approaches, and some specific implementations of it. The challenges come in three main shapes:

Not enough expressivity, no ability to ask instantiation queries, or computational issues. We will look at three particular implementations that fall within this category: Isabelle automated theorem prover, Leo III automated theorem prover, and Lambda Prolog. A technical comparison between dependency graph unification and standard higher-order unification can be found in §7.7.

5.4.1.1 Isabelle

Isabelle[2] [Paulson, 1989] is an interactive higher-order theorem prover with full expressivity and focused on tactics and giving the user the ability to structure proofs themselves and utilize tactics and lemmas in an effective way, rather than give the theorem prover full control over the proof search. It is quite powerful in the hands of an experienced mathematician and has a robust foundation and a lot of technical treats.

The issue with off-the-shelf Isabelle is that it is fundamentally incapable of providing a list of results. An Isabelle program consists of a series of logical statements at varying levels of abstraction that the automated theorem prover *verifies*, either generating errors or accepting it as valid. For example, the following is an Isabelle program that proves that

$$(\forall x.(p(x) \implies q(x))) \wedge (\forall y.p(f(y))) \implies (\forall z.q(f(z))) \tag{5.1}$$

```
theory example
  imports Main
begin

theorem
  fixes p q :: "'a \<Rightarrow> bool"
  fixes f :: "'a \<Rightarrow> 'a"
  assumes A1: "\<forall>x :: 'a. p(f(x))"
  assumes A2: "\<forall>y :: 'a. p(y) \<longrightarrow> q(y)"
  shows "\<forall>z. q(f(z))"
proof
  fix z

  from A1 have L1: "p(f(z))" by simp
```

[2] https://isabelle.in.tum.de/

```
  from A2 have L2: "p(f(z)) \<longrightarrow> q(f(z))" by simp
  from L1 L2 show "q(f(z))" by simp
qed

end
```

This program was written mostly manually, and the reason why we know the theorem is proven is because Isabelle accepts the program with no errors. The automated theorem prover of Isabelle has two fundamental ways in which the user interacts with it:

1. By calling proof methods (for example, *simp* in the example).
2. By providing error outputs to the user when proof steps are invalid or proof goals are not fulfilled.

The output of the second one is only an accept or an error message. But in no case will it return instantiations of variables. The only hope would come from the first one. While the output is still an accept or error message, the proof methods can be complex and involve relatively complex proofs (automated). Even if it's not usually output, one could argue it would be simple to output these proofs. However, this is still largely problematic. To see why, let us try to have Isabelle output q as an instantiation of a free existential variable. We start by existentially quantifying it, while keeping a specific proof for q:

```
theory existential
  imports Main
begin

theorem
  fixes p q :: "'a \<Rightarrow> bool"
  fixes f :: "'a \<Rightarrow> 'a"
  assumes A1: "\<forall>x :: 'a. p(f(x))"
  assumes A2: "\<forall>y :: 'a. p(y) \<longrightarrow> q(y)"
  shows "\<exists>r. \<forall>z. r(f(z))"
proof
  show "\<forall>z. q(f(z))" proof
    fix z
```

```
    from A1 have L1: "p(f(z))" by simp
    from A2 have L2: "p(f(z)) \<longrightarrow> q(f(z))" by simp
    from L1 L2 show "q(f(z))" by simp
  qed
qed
```

This works absolutely fine, but we had to tell Isabelle that q was the instantiation we were looking for. Let us have it produce that automatically for us. We could start by completely automating the proof for q:

```
theory existentialauto
  imports Main
begin

theorem
  fixes p q :: "'a \<Rightarrow> bool"
  fixes f :: "'a \<Rightarrow> 'a"
  assumes A1: "\<forall>x :: 'a. p(f(x))"
  assumes A2: "\<forall>y :: 'a. p(y) \<longrightarrow> q(y)"
  shows "\<exists>r. \<forall>z. r(f(z))"
proof
  show "\<forall>z. q(f(z))" by (simp add: A1 A2)
qed

end
```

which also works perfectly. The only step left is to not tell it the instantiation q is the one we're after. This actually also works:

```
theory existentialsearch
  imports Main
begin

theorem
  fixes p q :: "'a \<Rightarrow> bool"
  fixes f :: "'a \<Rightarrow> 'a"
  assumes A1: "\<forall>x :: 'a. p(f(x))"
  assumes A2: "\<forall>y :: 'a. p(y) \<longrightarrow> q(y)"
```

```
  shows "\<exists>r. \<forall>z. r(f(z))"
  by blast

end
```

While we do not get the actual instantiation of r that satisfies the proof, extracting a witness from the proof would probably not be very hard. It could be either $r = p$ or $r = q$ that *blast* utilizes to find the proof. But herein lies the fundamental issue. There are other instantiations, like $p \circ f$, $q \circ f$, $p \circ f \circ f$, etc. How can we ask Isabelle to output *all* instantiations? We cannot. A simple idea would be to explicitly exclude the ones we already know of (so iteratively using Isabelle to extract solutions sequentially). We can try this:

```
theory existentialsearch
  imports Main
begin

theorem
  fixes p q :: "'a \<Rightarrow> bool"
  fixes f :: "'a \<Rightarrow> 'a"
  assumes A1: "\<forall>x :: 'a. p(f(x))"
  assumes A2: "\<forall>y :: 'a. p(y) \<longrightarrow> q(y)"
  shows "\<exists>r. r \<noteq> p & (\<forall>z. r(f(z)))"
  by blast

end
```

This program does not work. Moreover, when using the catch-all *sledgehammer* procedure to try to find any method that works to prove this, Isabelle produces no results. The inequality makes the situation hard enough that Isabelle's automated elements are at a loss.

The reason behind this is at the heart of the inadequacy of off-the-shelf Isabelle to solve our problem, and relates to the standard higher-order unification algorithm [Huet, 1975], which Isabelle fundamentally utilizes (in varying ways, depending on proof method). As described previously, one of the most important aspects of this algorithm is that it focuses on *unifiability* rather than on finding *all unifiers*. Therefore, the inequality is checked by Isabelle *a posteriori* after unification has found one unifier

for the rest of the formula. It cannot go back and find *a different* unifier. Moreover, and more importantly, even if we modified the algorithm (or Isabelle's proof methods) to produce all unifiers, or at least sequentially produce those that we asked, it would become intractable. *No known algorithm produces all maximal unifiers of a higher-order logic problem in an computationally feasible way for medium sized problems*. Avoiding this search is a known *feature* of Huet's algorithm, as it avoids computational issues. But in our problem, we fundamentally need it to go back to this constraint. The algorithm is simply not designed for this, and thus neither is Isabelle.

Of course, a very sensible idea is to try to modify the algorithm to output all unifiers in a computationally reasonable way, possibly by leveraging the expressivity limitations of ESQ logic. But at this point, we are fundamentally doing exactly what this thesis ends up doing, and whether or not we use Isabelle on top is only an additional liability and not an advantage for the problem at hand. The outcomes of this thesis could be applied to extend Isabelle to do this, but I considered that using Isabelle would not have been an advantage to begin with.

5.4.1.2 Leo III

Leo III[3] [Steen and Benzmüller, 2018, Steen, 2020] is a fully automated higher-order theorem prover based on extensional higher-order paramodulation. The main difference with Isabelle is that it is fully automated rather than being interactive / tactical. However, at its core it follows similar principles: uses particular tactics to guide the search, and is ultimately reliant on a variation of the standard higher-order unification algorithm [Huet, 1975].

We can follow a similar approach to what we did with Isabelle to see the limitations with Leo III. Leo III uses TPTP[4] input files to describe its problems. We can produce the following file to describe the same problem as above, that is, prove:

$$(\forall x.(p(x) \implies q(x))) \wedge (\forall y.p(f(y))) \implies (\forall z.q(f(z))) \tag{5.2}$$

```
thf(p_type,type,
    p: $i > $o ).
```

[3] https://github.com/leoprover/Leo-III
[4] https://tptp.org/

```
thf(q_type,type,
    q: $i > $o ).

thf(f_type,type,
   f: $i > $i).

thf(a1,axiom,
    ( ! [X: $i] :
   ( ( p @ (f @ X)))
   )
   ).

thf(a2,axiom,
   ( ! [X: $i]:
      ( ( ~ (p @ X))
         | (q @ X))
   )
   ).

thf(c,conjecture,
   ( ! [X : $i] :
      ( q @ (f @ X))
   )
   ).
```

This program successfully terminates almost instantaneously with a proof. Leo III can also be easily prompted to output the proof, which ultimately also will allow us to generate witnesses for existentially quantified variables. The following is the proof output for this program:

```
thf(p_type, type, p: ($i > $o)).
thf(q_type, type, q: ($i > $o)).
thf(f_type, type, f: ($i > $i)).
thf(sk1_type, type, sk1: $i).
thf(4,axiom,((! [A:$i]: (~ (p @ A) | (q @
    A)))),file('leo3_test.txt',a2)).
thf(9,plain,((! [A:$i]: (~ (p @ A) | (q @
    A)))),inference(defexp_and_simp_and_etaexpand,[status(thm)],[4])).
```

```
thf(10,plain,(! [A:$i] : ((~ (p @ A)) | (q @
    A))),inference(cnf,[status(esa)],[9])).
thf(1,conjecture,((! [A:$i]: (q @ (f @ A)))),file('leo3_test.txt',c)).
thf(2,negated_conjecture,((~ (! [A:$i]: (q @ (f @
    A))))),inference(neg_conjecture,[status(cth)],[1])).
thf(5,plain,((~ (! [A:$i]: (q @ (f @
    A))))),inference(defexp_and_simp_and_etaexpand,[status(thm)],[2])).
thf(6,plain,((~ (q @ (f @ sk1)))),inference(cnf,[status(esa)],[5])).
thf(11,plain,(! [A:$i] : ((~ (p @ A)) | ((q @ A) != (q @ (f @
    sk1))))),inference(paramod_ordered,[status(thm)],[10,6])).
thf(12,plain,((~ (p @ (f @
    sk1)))),inference(pattern_uni,[status(thm)],[11:[bind(A, $thf(f @
    sk1))]])).
thf(3,axiom,((! [A:$i]: (p @ (f @ A)))),file('leo3_test.txt',a1)).
thf(7,plain,((! [A:$i]: (p @ (f @
    A)))),inference(defexp_and_simp_and_etaexpand,[status(thm)],[3])).
thf(8,plain,(! [A:$i] : ((p @ (f @
    A)))),inference(cnf,[status(esa)],[7])).
thf(16,plain,(~ ($true)),inference(rewrite,[status(thm)],[12,8])).
thf(17,plain,($false),inference(simp,[status(thm)],[16])).
```

As we did with Isabelle, we can existentially quantify the predicate on the conjecture, to push Leo III and its unification algorithm to instantiate it, producing the instantiation of the pattern that we search. The following is the program:

```
thf(p_type,type,
    p: $i > $o ).

thf(q_type,type,
    q: $i > $o ).

thf(f_type,type,
   f: $i > $i).

thf(a1,axiom,
    ( ! [X: $i] :
   ( ( p @ (f @ X)))
   )
```

```
   ).

thf(a2,axiom,
   ( ! [X: $i]:
      ( ( ~ (p @ X))
         | (q @ X))
   )
   ).

thf(c,conjecture,
   ( ? [Q : $i > $o] :
      ( ! [X : $i] :
         ( Q @ X)
      )
   )
   ).
```

which also finishes almost instantaneously, and when prompted for a proof allows us to extract a witness of the instantiation:

```
thf(p_type, type, p: ($i > $o)).
thf(f_type, type, f: ($i > $i)).
thf(3,axiom,((! [A:$i]: (p @ (f @ A)))),file('leo3_test_2.txt',a1)).
thf(7,plain,((! [A:$i]: (p @ (f @
    A)))),inference(defexp_and_simp_and_etaexpand,[status(thm)],[3])).
thf(8,plain,(! [A:$i] : ((p @ (f @
    A)))),inference(cnf,[status(esa)],[7])).
thf(1,conjecture,((? [A:($i > $o)]: ! [B:$i]: (A @
    B))),file('leo3_test_2.txt',c)).
thf(2,negated_conjecture,((~ (? [A:($i > $o)]: ! [B:$i]: (A @
    B)))),inference(neg_conjecture,[status(cth)],[1])).
thf(5,plain,((~ (? [A:($i > $o)]: ! [B:$i]: (A @
    B)))),inference(defexp_and_simp_and_etaexpand,[status(thm)],[2])).
thf(6,plain,(! [A:($i > $o)] : ((~ (A @ (sk1 @
    (A)))))),inference(cnf,[status(esa)],[5])).
thf(11,plain,($false),inference(rewrite,[status(thm)],[8,6])).
thf(12,plain,($false),inference(simp,[status(thm)],[11])).
```

The important proof step here is step 11, which combines the result of steps 8 and 6. This ultimately indicates the unification of the existential variable (A / Q) with the composition function $p \circ f$, producing the instantiation (not the one we originally looked for, but rather a simpler one, which is also valid).

We note that this step is a fundamentally higher-order unification step, that therefore relies on some variation of Huet's algorithm, and produces only one unifier. Similar to Isabelle, the only simple way to prompt it to find a different unifier is to explicitly indicate the different instantiation as part of the problem:

```
thf(p_type,type,
    p: $i > $o ).

thf(q_type,type,
    q: $i > $o ).

thf(f_type,type,
   f: $i > $i).

thf(a1,axiom,
    ( ! [X: $i] :
   ( ( p @ (f @ X)))
   )
   ).

thf(a2,axiom,
   ( ! [X: $i]:
      ( ( ~ (p @ X))
         | (q @ X))
   )
   ).



thf(c,conjecture,
   ( ? [Q : $i > $o] :
      ( (Q != (^ [X : $i] :
         ( p @ (f @ X) )
      ))
```

```
        & ( ! [X : $i] :
           ( Q @ X)
        )
     )
  )
  ).
```

where the line `Q != (^ [X : $i] : ( p @ (f @ X))` indicates that we don't want this unifier.

As before, attempting to run this program times out after 60 seconds. The reason, ultimately, we argue, being that the reliance on the standard higher-order unification algorithm limits the search for unifiers.

As with Isabelle, it would be very sensible to extend Leo III's approach to use a different unification approach that produced all unifiers, but this is essentially the same challenge as the one this thesis does tackle, and I considered that using Leo III as basis would not offer any significant advantage for this particular task.

5.4.1.3 Lambda Prolog

Lambda Prolog[5] [Miller, 2021, Nadathur and Miller, 1988] (§3.3.1.1) is a logic programming language that extends traditional Prolog with higher-order capabilities. As any other programming language, it is fundamentally a query language, and the production of witnesses for proofs as output and the full exploration of the search space are considered more explicitly.

However, Lambda Prolog also has other important limitations that make it unsuitable to solve our problems. We explore them (mostly in the scope of the most prominent implementation Teyjus[6]) in this section. These limitations come in the shape of three constraints on what can be expressed in the Lambda Prolog language to begin with. The reasons for these constraints are precisely computational feasibility, and relate (and to a degree reinforce my arguments) that the fundamental limitations of the higher-order unification algorithm [Huet, 1975] are at the core of the unsuitability of these approaches. Lambda Prolog disallows certain structures because the unification

[5] http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/
[6] http://teyjus.cs.umn.edu/

algorithm is unfit for handling them properly. This means that we cannot express most of our patterns in Lambda Prolog. The following are the relevant expressivity constraints:

- **Horn clauses only** - A Horn clause (§3.3.1) is a clause with only one positive literal (*goal* or *head*). Most or all logic programming languages are limited to Horn clauses (in higher-order logic extended to the notion of *hereditary Harrop formulas*, which allows quantifiers to appear), as this severely restricts the search properties of the algorithm (in particular, the growth of goal sizes, see §3.2.1.2). However, a lot of our patterns cannot be expressed via Horn clauses (as an example, you may consider §A.2.7).

- **No uninstantiated variables as heads** - Teyjus has a specific check that outputs an error when a rule or a goal have uninstantiated variables in their head. For example, the following rule cannot be expressed in Lambda Prolog, where P is a second-order variable:

$$P(c_1) \wedge p(x) \implies P(x) \tag{5.3}$$

  The main reasons for disallowing this are also very clear: the logic programming approach and resolution search means that rules, facts and goals are matched on their head, and unification is used to propagate this matching unto the body of the rules. A rule or goal with a variable head would in principle match with any or many different goals, producing very different unrelated unifiers (again, the fundamental aspect of the higher-order unification algorithm we have been talking about), which would likely make the search space explode and the program to not terminate easily. Thus, Lambda Prolog restricts it. The same example referred to above utilizes variable heads (§A.2.7).

- **Queries with body** - In most or all logic programming languages, *queries* are referred to as *goals*, and consist of a clause with a single, positive literal. A query with a *body* would be equivalent to a clause with negative literals as a goal. This is disallowed in any logic programming language for the same reason that the language is limited to Horn clauses: the combination of these two limitations severely limits the search space and allows the search to be conducted linearly in a regular and predictable manner. Allowing queries with body in a logic program

would have similar effects to allowing non-Horn clauses in the program. Once again, the same example referred to above utilizes queries with body (§A.2.7).

These limitations are too severe to allow Lambda Prolog to constitute an acceptable basic approach to our problem. The same reasons why these limitations exist are those that we attempt to overcome by leveraging other, different, limitations in our particular problem (that Lambda Prolog does not assume). Thus, the two problems are too different in application to be worth it to consider one as a particular case of the other.

5.4.2 Decidable subsets of second-order logic

In §3.2.2.2 we described a series of subsets of second-order logic that are known to be decidable and have specific algorithms for their resolution which are computationally more attractive than the general higher-order unification algorithm. These are *linear second-order unification*, *bounded second-order unification* and *monadic second-order unification*. It is relevant to consider whether our problem might fall within any of these subproblems and thus can benefit from these existing more attractive algorithms. The situation is not in general attractive, though we do identify a couple of directions of potential investigation that might yield useful results. In this section we detail the reasons for this and the extent of this issue.

5.4.2.1 Linear and bounded second-order unification

Linear second-order unification [Levy, 1996] restricts the set of unifiers that can be produced to those that instantiate second-order variables to *linear terms*; a linear term being a term for which bound variables in lambda abstractions occur exactly once in the body. Bounded second-order unification [Schmidt-Schauß, 2004] can be considered to be an extension of linear second-order unification to the more general case of where the number of occurrences is bounded at the beginning of the problem.

It is in fact true that all of our test cases (as described in chapter §9) have target instantiations associated with them that fall within the restrictions of linear second-order unification. One could argue that in general we would want other instantiations, but at a first glance utilizing the algorithm associated with linear and bounded second-order unification could be promising, at least to produce the simpler solutions to the problem, even if not all.

The main issue with these algorithms is that they are fundamentally similar to the brute force search approach described in §5.2, with limits for the depth of this search (this finite aspect of the search is not relevant for the approach followed in this thesis, as we are prepared to deal with infinite enumerations of solutions in general). There are other aspects of the algorithms described in [Levy, 1996, Schmidt-Schauß, 2004] that could be attractive for our problem, but as a matter of fact, [Schmidt-Schauß, 2004] provides a formal proof of the NP-hardness of their algorithm, which relates to the computational issues we described in §5.2.

In other words, the attractive algorithmical properties of these algorithms come in the flavour of limits to the search that make the problem *decidable* but not *efficient*. I must, however, acknowledge, that I had not become aware of the applicability of these algorithms until the post-viva modifications to this thesis, which is ultimately the reason I didn't explore these avenues further. The argument provided makes me confident that this was definitely not a trivial perfectly viable solution that I ignored, but I have to admit that investigating this in more detail could be productive. However, at the time of this writing, I could not find any actual automated theorem provers which utilize these algorithms to try some practical problems on. I believe considerations about decidability of subsets of second-order logic tend to be theoretical and concerned with the decidability of the problems, rather than pragmatic. Further investigation would be useful, but almost certainly not leading to a direct solution to the problem, and I do not have the time availability at the time of this writing to conduct it in full extent.

5.4.2.2 Monadic second-order unification

Monadic second-order unification [Farmer, 1988] is second-order unification in a language that contains no function symbols with aritty greater than or equal to 2. The algorithm associated with it is in fact considerably more efficient than standard higher-order unification. However, in this case the expressivity limitation is too much for most of our patterns (too much to be worth considering).

Most or all of the patterns and ontologies that we would want to detect patterns in, are non-monadic. In particular, any OWL ontology that contains any *properties* (see §3.1) is automatically non-monadic, as properties in OWL are binary function symbols in first/second-order logic terms. As a particular example (just one of many) from our patterns, consider §A.2.4.

5.4.3 Constraint programming

5.4.3.1 Satisfiability Modulo Theories (SMT)

The application of an SMT-based approach (§3.2.3) to the automatic detection of ESQ logic queries in ontologies would require the separation of this detection problem in two parts: A propositional part, and a part that can be implemented as either an eager transformation of a ESQ problem into a propositional formula, or a lazy element which can be queried during the resolution of a query using a SAT algorithm. The difficulty of this task comes from the fundamental notion that we are trying to express full first-order logic in propositional logic. We note that ESQ logic strictly contains all of first-order logic. In particular, a ESQ query with no second-order variables is exactly a first-order theorem proving problem, and every first-order theorem proving problem can be presented this way. The task of detaching the propositional logic aspect of first-order logic from the strictly first-order elements has no evident solution. We can look to the literature [Bongio et al., 2008, Barrett et al., 2002] for existing research in this area.

On one hand, work like [Barrett et al., 2002] encodes quantifier-free first-order logic iteratively as a propositional problem, reducing the size of the grounding of formulas in the naive transformation of quantifier-free first-order logic to propositional logic. However, quantifier-free first-order logic is by far not expressive enough for our purposes. In particular, nearly all of the patterns in our catalogue (appendix A) have first-order quantifiers. In general, we focus a lot of our practical applications on ontology languages like OWL, whose bread and butter include subsumptions of classes and domain axioms, both of which are fundamentally quantified first-order statements about the instances of classes and properties. This branch of the research does not concern itself with quantified variables, the reason being the fundamentally non-propositional nature of these that impedes their iterative grounding approach.

On the other hand, work like [Bongio et al., 2008] focuses on the technical proof objects of first-order logic and when these can be easily encoded in an SMT problem. In particular, [Bongio et al., 2008] uses SMT to solve *rigid* first-order theorem proving problems by using connection Tableaux and encoding their technical aspects as an SMT solver. We note that the results in this work are mixed, and are positive mostly when subjected to two conditions:

1. **Rigidness** - Rigid satisfiability of a first-order formula is the problem of checking whether a ground instance of the formula is satisfiable. Thus, checking non-rigid unsatisfiability involves checking rigid unsatisfiability of every possible instantiation of a formula. This is semi-decidable in general, and computationally problematic. The work in [Bongio et al., 2008] note this difference and perform their encoding for both rigid and non-rigid, utilizing an enumeration procedure similar to the one mentioned. Their positive results are almost exclusively limited to the rigid case.

2. **Horn clauses** - As discussed in §5.4.1.3, limiting our problem to Horn clauses would severely limit the extent of the applicability of our problem. For example, the pattern in §A.2.7 cannot be expressed as a Horn clause. The approach in [Bongio et al., 2008] does extend to non-horn clauses, but they report negative results having to do with computational issues in these cases.

All of the above discussion does not even consider the fact that ESQ logic is strictly more expressive than first-order logic: first-order logic being comfortably solvable using SMT approaches is a *necessary*, but not *sufficient* condition for automatic detection of ESQ queries using SMT to be feasible. Since this condition is already only partly and conditionally held, I considered it sensible not to explore this avenue of work. This is not to say that the notion of extending SMT to solve ESQ queries (or a subset of these) is absurd, because it is not. But developing this approach would involve a large amount of research in a different direction than the one I chose for this thesis.

5.4.3.2 Answer Set Programming (ASP)

The idea of utilizing Answer Set Programming (§3.3.2) to solve ESQ queries encounters very similar challenges to those that utilizing SMT did. ASP is mainly aimed at finite spaces, with quantified formulas being fundamentally problematic to represent and solve within it. Approaches to this normally come in the shape of enumerating or limiting the extent to which the formulas are quantified, similar to the rigidness issue described in §5.4.3.1. Indeed, in [Baget et al., 2018], one of the main pieces of literature that I was able to find in this topic, section 2.4 explicitly describes the intrinsic limitations of existential rules when utilizing an ASP approach to solve them.

We remind the reader that this describes the limitations of utilizing this approach to solve *restricted versions of first-order logic*, with ESQ logic being an *extension* of first-order logic. The expressivity is simply too small. As with SMT, this avenue of

research is not absurd. Indeed, extensions of ASP approaches to the particularities of ESQ (or restricted versions of it) could end up being productive. However, this would involve a significant amount of research that is not part of what I have decided to spend my time on in this thesis.

5.4.3.3 Constraint logic programming

Constraint logic programming (§3.3.1.1) refers to the extension of the syntax, philosophy and algorithmics of Prolog to more general constraint-solving problems. It does not constitute, at its core, a particular algorithm for solving constraint problems, but rather a framework for expressing them and tackling the different steps that constitute it.

We do embrace the basic ideas of constraint logic programming in the way that ESQ logic is expressed itself, and the computational semantics attached to itt. In particular, ESQ queries are expressed as compositions of smaller queries, solved in a particular order, the results of some are used as inputs to following queries. Solving each query involves finding solutions to a particular kind of constraint problem. Thus, ESQ logic takes heavy inspiration from constraint logic programming. But this does not provide us with an algorithm for finding solutions to atomic ESQ queries.

As an example of how a ESQ query can be seen as a constraint logic programming goal, consider the following query, taken from the pattern in §A.2.3:

$$\begin{aligned}&((X) \vDash \neg\exists x.X(x)) \bowtie \\ &\bowtie ((X) \vDash_M \mathit{primitive}(X))\end{aligned} \tag{5.4}$$

This could easily be seen as a constraint logic program looking something like this:

```
unsatisfiable_class(X) :- primitive(X), entailed(not(exists(y,X(y)))).
```

The algorithmical difficulty, however, comes from the resolution of the goal `entailed(not(exists(y,X(y))))`. The formal particularities of ESQ logic are discussed in more detail in §6.2.

5.5 Minimal commitment resolution for ESQ logic

Minimal commitment resolution for ESQ logic is the algorithm that we have developed

to find instantiations of ESQ queries. The fundamental difference between this approach and the brute force approach described earlier is the *minimal commitment* principle: we only instantiate variables **when** it is necessary to continue with a proof, and we only instantiate them **as much** as necessary to continue with a proof. This principle is applied in a few different ways throughout the description of the algorithm. It should be noted that minimal commitment is also what underlies standard first-order unification, among a lot of other algorithms and approaches in the field; we are extending it to our particular problem in a particular way.

The general approach of the algorithm can be described as follows:

Algorithm 5.5.1 (Minimal commitment resolution for ESQ logic (outline))**.**

1. Use first-order resolution as a basis. See §3.2.1.
2. On one hand, non-deterministically instantiate second-order predicate variables to composite predicates with logical connectives. On the other, assume second-order predicate variables to be atoms and apply resolution normally to them. Details of this aspect are discussed in §5.5.1.
3. Instead of solving unification problems whenever resolution is applied, generate *declarative equations* representing these unifications.
4. Once a proof is found with an associated set of equations, apply *dependency graph for unification equations* to find instantiations of second-order variables that satisfy the equations. Details of this aspect are discussed in §5.5.3.

Starting with part 1, the aspects of first-order resolution (§3.2.1) that we preserve are:

- We use *conjunctive normal form* to express the formulas that we are working to prove
- We use a *refutation* approach, by conjunctively joining the conjunction of the whole theory with the negation of the conjectured theorem.
- We apply the resolution rule between two sets of literals in two clauses in the CNF. However, this is applied a bit differently in minimal commitment resolution for ESQ logic. See §5.5.1 for details.

- A proof is finished if the empty clause is found.

5.5.1 Maximal CNFs

Maximal CNFs are an extension of first-order logic Conjunctive Normal Form formulas (CNFs) to include unifier variables and second-order variables, while keeping the same basic structure. The reason we call these *maximal* is that they could stop being in conjunctive normal form depending on the instantiations of second-order variables and the values of unifier variables, but structurally they are as close to a conjunctive normal form as they can be without having more information about these instantiations. In this section we describe this concept in detail.

Consider a *maximal conjunctive normal form* formula with second-order variables:

$$(\delta_{1,1}\alpha_{1,1} \vee \delta_{1,2}\alpha_{1,2} \vee \ldots \vee ..\delta_{1,m_1}\alpha_{1,m_1}) \wedge (\delta_{2,1}\alpha_{2,1} \vee \ldots \vee \delta_{2,m_2}\alpha_{2,m_2}) \wedge \ldots \wedge (\delta_{n,1}\alpha_{n,1} \vee \ldots \vee \delta_{n,m_n}\alpha_{n,m_n}) \tag{5.5}$$

or, more succinctly:

$$\bigwedge_{i \in 1..n} \Big(\bigvee_{j \in 1..m_i} (\delta_{i,j}\alpha_{i,j}) \Big) \tag{5.6}$$

where:

- n and each m_i can be zero (an empty CNF is never unsatisfiable, whereas an empty clause is always unsatisfiable).
- Each $\delta_{i,j}$ is either a negation or nothing (negative or positive literal).
- Each $\alpha_{i,j}$ is what we call a *meta-atom*. The formal details of meta-atoms can be found under the definition of *unifier expressions* (definitions 6.1.29, 6.1.19).

  Conceptually, meta-atoms are like first-order atoms except they may contain:

  - Second-order variables.
  - Unifier variables. The meaning and usage of these is explained in more detail in §5.5.2, and the technical details are presented in §6.1.3.

Second-order variables and unifier variables both represent a level of *indeterminacy* in the formula. A second-order variable is to be *instantiated* to a predicate or function

(depending on whether it is a predicate or function variable), and a unifier variable is to be replaced by a first-order substitution (a first-order unifier, in the usual sense). For example, consider the maximal CNF:

$$(p(x) \vee \neg P(x)) \wedge (\neg \sigma_1 p(f(y))) \wedge (q(z))$$

If we apply the instantiation $P := q \circ f$ and the substitution $\sigma_1 := \{y \to f(w)\}$ to it, the result would be the first-order (no longer has second-order variables or unifier variables) CNF formula:

$$(p(x) \vee \neg q(f(x))) \wedge (\neg p(f(f(w)))) \wedge (q(z))$$

However, if we apply the same substitution, but instead use the *composite* instantiation $P := (\lambda a.p(a) \wedge q(a))$, it would result in the formula:

$$(p(x) \vee \neg (p(x) \wedge q(x))) \wedge (\neg p(f(f(w)))) \wedge (q(z))$$

which **no longer is in conjunctive normal form**.

We can consider every possible instantiation of the second-order variables while being able to treat a maximal CNF as an actual CNF for resolution purposes by performing *inductive instantiation* of the maximal CNF. This is essentially a systematic way to return the formula to CNF with low cost and ensuring productivity of the search:

At each step, if there are second-order predicate variables in a maximal CNF, produce the following non-deterministic branches in our search:

- Assume every second-order predicate variable can only be instantiated to atoms and proceed with steps 5.5.3 and 4 of algorithm 5.5.1 (see §5.5.2 and §5.5.3).
- Pick one second-order variable[7] P with arity k and do one step of partial instantiation, producing branches:
  - Instantiate P to $Q \wedge R$, for fresh predicate variables Q and R.
  - Instantiate P to $\neg Q$, for fresh variable Q.
  - Instantiate P to $(\lambda x_1, ..., x_k. \forall y. Q(y, x_1, ..., x_k)$.

  After each of the above steps, transform the formula into a conjunctive normal form again.

[7] How to make this choice is a heuristic decision that we do not discuss here.

This process is relatively technical. A more precise definition and theoretical results about it can be found on §6.3. Moreover, as discussed in chapter 8, in our implementation we only pursue the first branch (assuming variables are instantiated to atoms). This does sacrifice completeness, and reasons for this choice are explained in chapter 8. Theoretical results are however produced on the complete search space.

At a conceptual level, what we are doing here is to divide the set of instantiations of the maximal CNF in a way that allows us to productively explore branches that we can assume correspond to actual CNFs, and apply resolution to them. Resolution cannot be applied to formulas unless we are certain of the fact that they are in CNF.

The remainder of this section works with formulas in which we assume that each meta-atom can only be instantiated to an actual atom, and thus that the formula is going to be a CNF after instantiation.

5.5.2 Resolution and implicit unification

The next thing that we have to do, once we have a maximal CNF in which we assume all meta-atoms can only be instantiated to atoms, is to apply the resolution rule. To remind what was discussed in §3.2.1, in first-order logic, at this step we apply unification to the two clauses being resolved, and apply the resulting unifier to the resolvent. For example, if we resolved the following two clauses on the literals marked in bold:

$$C_1 \equiv (\mathbf{p}(\mathbf{x}) \vee q(f(x)))$$
$$C_2 \equiv (r(y) \vee \neg\mathbf{p}(\mathbf{g}(\mathbf{y})))$$

this would have the unifier $x \sim g(y)$ and result in the following resolvent:

$$R \equiv (q(f(g(y))) \vee r(y))$$

At this point, we will bring attention to a fundamental aspect of resolution called *factoring* of literals. Any resolution refutation procedure must implement this to be correct [Robinson and Voronkov, 2001], but there are multiple different ways to do so.

One way of *factoring* consists in, given a single clause, if it has multiple literals with the same sign (positive or negative) that are unifiable, attempt to unify them, producing a new clause.

An alternative way of implementing *factoring*, which we inherit in our approach, is, when applying the resolution rule, rather than unifying a single literal from each clause,

unify multiple literals from each clause (same sign in each of the clauses, and all are unified at the same time). This is usually called *general resolution*.

Factoring is necessary for resolution to be correct. Otherwise, a CNF formula with two or more literals in each clause would **never** produce the empty clause, even when there is a proof[8].

Factoring implemented as part of the resolution rule would, for example, take the following two clauses and resolve on the literals marked in bold:

$$C_1 \equiv (\mathbf{p}(\mathbf{x},\mathbf{f}(\mathbf{z})) \vee \mathbf{p}(\mathbf{z},\mathbf{f}(\mathbf{z})) \vee q(f(x),z))$$
$$C_2 \equiv (r(y,w) \vee \neg\mathbf{p}(\mathbf{g}(\mathbf{y}),\mathbf{w}))$$

with the resulting unifier: $\{x \sim z \sim g(y), w \sim f(g(y))\}$, producing the resolvent:

$$R \equiv (q(f(g(y)),g(y)) \vee r(y,f(g(y))))$$

Not surprisingly, factoring becomes more tricky when considering second-order variables. To see this, consider a variation on the previous example with some second-order variables substituted in:

$$C_1 \equiv (\mathbf{p}(\mathbf{x},\mathbf{F}(\mathbf{z})) \vee \mathbf{P}(\mathbf{z},\mathbf{f}(\mathbf{z})) \vee q(f(x),z))$$
$$C_2 \equiv (r(y,w) \vee \neg\mathbf{p}(\mathbf{g}(\mathbf{y}),\mathbf{w}))$$

(note the predicate second-order variable P and the function second-order variable F).

Function variables do not alter the principle of the resolution rule or factoring itself, and instead are entirely dealt with in the unification step 4, explained in §5.5.3. Predicate variables, however, do.

Are $p(x,F(z))$ and $P(z,f(z))$ unifiable? That depends on the instantiation of P. The resolution could be applied regardless, since $p(x,F(z))$ and $\neg p(g(y),w)$ are unifiable in any case; the doubt comes as to whether we should factor in $P(z,f(z))$ or not.

The solution is yet again to introduce non-determinism (in a similar way that first-order resolution already does by allowing multiple search orders in the application of the resolution rule). We consider two possible resolutions: one that includes $P(z,f(z))$ and one that does not. Each of these could lead to different instantiations of

[8]This is because each resolution application would produce a clause of equal or larger size than the previously existing ones, and the size of the clauses would never be reduced.

the second-order variables.

With factoring out of the way, we still have the question: how do we unify $p(x,F(z))$, $P(z,f(z))$ and $\neg p(g(y),w)$? The result of first-order unification would depend (drastically) on the instantiations of P and F. We could, of course, consider possible instantiations at this point, but this would be close to the brute force approach described in §5.2. Instead, we do *implicit unification*, producing *unification equations* that we solve later (see §5.5.3). This is what *unifier variables* are used for.

In the example above:

$$C_1 \equiv (\mathbf{p}(\mathbf{x},\mathbf{F}(\mathbf{z})) \vee \mathbf{P}(\mathbf{z},\mathbf{f}(\mathbf{z})) \vee q(f(x),z))$$
$$C_2 \equiv (r(y,w) \vee \neg\mathbf{p}(\mathbf{g}(\mathbf{y}),\mathbf{w}))$$

instead of unifying the selected literals, we use a fresh unifier variable (σ_1) to indicate what this unifier may end up being, without finding its actual value; and produce two things: the resolvent, and a unification equation to be solved later. The latter is to be seen as a constraint that needs to hold for the proof to be correct, and which summarizes what the instantiations of the second-order variables can and cannot be.

The unification equations produced would be:

$$\sigma_1 p(x,F(z)) \approx \sigma_1 P(z,f(z)) \approx \sigma_1 p(g(y),w)$$

while the resolvent would be:

$$(\sigma_1 q(f(x),z) \vee \sigma_1 r(y,w))$$

This allows us to produce a full resolution proof without having to worry about unification, generating equations that express the unification problems we would have to solve to have a correct proof. Only once we find the empty clause, do we worry with the *unification equation system* produced in the process. This is explained in the next section.

5.5.3 Dependency graph unification

At this point, we have reduced our problem to **solving systems of unification equations**, where each unification equation is of the form:

$$\delta_1 \approx \delta_2$$

where δ_1 and δ_2 are *unifier expressions* (formally, see definition 6.1.19).

We shall make it clear what a *solution* to a system of unification equations is. Technically, we call this a *unification solution* (definition 6.1.22). It consists of two things:

- A substitution (definition 6.1.10) associated to each unifier variable present in the equations.
- A single instantiation (definition 6.1.14) of the second-order variables present in the equations.

For example, consider the following set of unification equations:

$$\sigma_1 f(x) \approx \sigma_1 F(y)$$
$$\sigma_2 \sigma_1 g(F(y), w) \approx \sigma_2 g(z, f(z))$$

where we note that we have one second-order variable (F) and two unifier variables (σ_1, σ_2).

One possible unification solution U to this system is the following[9]:

$$U(F) = \pi_1^1$$
$$\sigma_1^U = \{y \to f(x)\}$$
$$\sigma_2^U = \{z \to f(x), w \to f(f(x))\}$$

which, when applied to the original equations, make the equality syntactically explicit:

$$U(\sigma_1 f(x)) = f(x) = U(\sigma_1 F(y))$$
$$U(\sigma_2 \sigma_1 g(F(y), w)) = U(\sigma_2 g(f(x), w)) = g(f(x), f(f(x))) = U(\sigma_2 g(z, f(z)))$$

In this section we describe how the set of unification solutions of a unification equation system can be extracted from the system, at an informal / semiformal level. This is the most technically advanced aspect of the work in this thesis, and chapter 7 contains a lot of detail about it.

The fundamental idea is that we represent systems of unification equations as *dependency graphs* where:

[9] We note that π_1^1 is the unary projection on the first argument (in other words, the identity function)

- *Nodes* represent *dependants*: atomic elements on which to define the substitutions and instantiation at the smallest level of granularity. For example, second-order variables F or function symbols f are second-order dependants, while unifier variables applied to first-order variables, like $\sigma_1 x$ or $\sigma_3\sigma_1 y$, are first-order dependants.

- *Edges* represent *dependencies* between these dependants. These are not technically edges in the usual sense, since not only are they directed, but they also have multiple (or zero) sources (whose order matters) and a head. Details are explained in chapter 7. For example, an equation $\sigma_1 y \approx f(\sigma_1 x)$ would normally be expressed as an edge with source $\sigma_1 x$, head f and target $\sigma_1 y$.

We note that the directionality on the edges represents the flow of information. **Targets depend on the head and the sources**, and not the other way around, which may be used sometimes in different contexts to represent dependencies. That is, informally, edges represent equations of the form Target = Head(Sources)

Encoded this way, a dependency graph represents all the relations between the atomic elements as defined by the unification equations. This representation allows us to identify independent elements, codependent elements and the most suitable parts of the problem to focus on to simplify it, changing the representation and producing different dependency graphs encoding the same sets of solutions in a more straightforward way.

This fundamental aspect of the *minimal commitment* approach is similar to existing ways to represent first-order unification as term graphs [Robinson and Voronkov, 2001]. However, there are some fundamental differences, such as the much more explicit way in which substitutions / instantiations are represented in our approach, the second-order aspect and perhaps most importantly, the set of rewrite rules on the graphs that we will introduce shortly. The minimal commitment and targetted reduction strategy can be compared to equational reasoning in arithmetic. For example, if we have equations:

$$\frac{3y^2+x}{x+1} = 2yx - 3y$$
$$2x = 6$$

then we may wish to solve the second equation first, to obtain $x = 3$, and then substitute it in the first one, obtaining $\frac{3y^2+3}{3+1} = 2 \cdot 3y - 3y$, which now only contains one variable

and can be solved using the formula for obtaining roots of second degree polynomials.

Our approach is similar at a conceptual level, except that we use dependency graphs and the relations between atomic elements in the unification solutions to find the best parts to solve first and propagate the solutions.

In a bit more detail, we have developed an algorithm based on *non-deterministic rewrite rules on dependency graphs*[10] that:

1. Begins with a dependency graph that encodes the system of unification equations in a direct way.
2. Applies rewrite rules to it.
3. Reaches a certain *normalization* level for the resulting graphs. In those normalization levels, explicit unification solutions may be extracted or a given solution can be checked against it.

The technical details of this approach, along with theoretical results about it, can be found in chapter 7. Here is a summary of the most important results:

- There is a systematic way to build a dependency graph from a system of unification equations with the same unification solutions. See algorithm 7.2.5.
- All rewrite rules on dependency graphs are *solution preserving* (they do not alter the set of unification solutions of the graph, definition 7.4.1).
- Explicit unification solutions may be extracted from a *normal* dependency graph systematically. See definition 7.6.4 and theorem 7.6.5.
- Every *quasinormal* dependency graph has at least one solution. This allows us to check whether a given unification solution is a solution to a given set of equations. See definition 7.6.16, theorem 7.6.9, algorithm 7.6.17 and theorem 7.6.10.
- There is an algorithm[11] that enumerates all unification solutions to a dependency graph in a *sound* and *fair* way. See theorem 7.6.8.

[10] Non-deterministic here means that some rules rewrite a single dependency graph into a set of dependency graphs, such that the solution of the original graph is the union of the solutions of all the graphs produced.

[11] Under a condition of *no acyclicity* in the process of solving the graph. In practice, this condition is nearly always met.

Together with the maximal CNF search approach and the adapted resolution rule with implicit unification, this allows us to produce a sound and fair enumeration of all the instantiations that make a given formula entailed by a given theory in ESQ logic, with a minimal commitment approach that greatly reduces the search as compared to the brute force approach.

5.6 Summary

In chapter 4, we introduced ESQ logic as a way to express fault patterns. In this chapter we have described an algorithmical approach to finding solutions to ESQ queries. We call this approach *minimal commitment resolution for ESQ logic*, and is an extension of first-order logic resolution in which we deal with second-order variables by only instantiating them whenever necessary. In particular, we deal with unification by representing it implicitly as equations during a proof, and solving these sets of equations once a proof has been found to find the instantiations of the second-order variables that satisfy them. This unification step is done via a novel approach that uses *dependency graphs* and a set of non-deterministic rewrite rules on dependency graphs to tackle the most productive parts of the problem first and reduce the size of the instantiation search space as much as possible. We have shown strong theoretical properties about this approach.

Chapter 6

Minimal commitment resolution for ESQ logic: Theoretical results

6.1 Basic pieces

In this section we provide some standard and non-standard definitions for the elemental concepts sustaining our resolution and unification algorithm. The core concepts of our developed algorithm are introduced in later sections in this and the next chapter. While some of these concepts are well known and used in the literature, formalizations may vary in approach. Since they are important elements of our theory, we introduce our specific definitions / useful results in the way that best suits our goals. In each case we indicate whether this concept is just our formalization of a preexisting concept, or a novel idea developed as part of this thesis.

6.1.1 Terms

In this subsection we introduce the basic notions of *first and second-order terms* (see §3.2.1 and §3.2.2), as well as definitions and results having to do mainly with their equality and normalization. Each first-order and second-order term has a unique normal form, and equality corresponds to equality of normal forms. These results are important for many of the considerations in the more novel sections of this chapter, as they enable us to normalize terms before comparing or decomposing them, ensuring certain useful propertties for the algorithms.

All ideas in this subsection are preexisting and appear extensively in the literature.

Some places to start would be [Robinson and Voronkov, 2001, Bundy, 1983].

Definition 6.1.1 (Signature)**.** *A second-order* signature *consists of:*

- *A countable (possibly empty, and typically finite) set of* function symbols*. Each function symbol has an associated* arity*: a natural number (including zero) indicating the number of arguments it may take. We normally use lowercase letters $f, g, h...$ to represent constant function symbols.*

- *A countable set of* predicate symbols*. Each predicate symbol has an associated* arity*: a natural number (including zero) indicating the number of arguments it may take. We normally use lowercase letters $p, q, r...$ to represent constant predicate symbols.*

- *A countable set of* first-order variables*. We normally use uppercase letters $X, Y, Z...$ to represent first-order variables.*

- *A countable set of* second-order variables*. Each second-order variable has an associated* arity*. We normally use uppercase letters $F, G, H...$ to represent second-order variables.*

- *A countable set of* meta-predicate symbols*. Each meta-predicate symbol has an associated* arity*. We normally use lowercase greek letters $\delta, \varepsilon...$ to represente meta-predicates.*

We note that, in conventional second-order logic, a first-order variable and a second-order variable of arity 0 have equivalent semantics. Indeed, a second-order variable of arity 0 is a variable that stands for a 0-ary function, which is a constant; so it's a variable that stands for a constant, which is a first-order variable. Thus[1], one could consider that having first-order variables is redundant. This is absolutely not true in our case.

Our second order variables are introduced at a separate layer from first-order variables, and the whole purpose of patterns and our algorithm is to find specific instantiations of second-order variables in our formulas that make the rest of the formula *entailed*, *satisfied* or a similar notion of provability. Thus, in a solution, a second-order variable is instantiated *before* the formula is considered to be entailed or satisfied, whereas a first-order variable is part of the semantics of whether the formula is entailed or satisfied. Thus, they are different. Clearly, this signals that our second-order

[1] and as noted by the principal examiner of this Thesis, Paul Jackson

variables are in some sense meta-variables (which we discuss in a little more detail on chapter 4), which they are in the way they are solved, but in terms of what they can be instantiated to, they behave exactly like second-order variables.

Throughout a lot of this section we will not explicitly mention predicate symbols. Syntactically, they behave very similarly to function symbols, except that they do not produce terms that are then usable to build newer terms; instead making *atoms*. However, the usage of atoms is more clearly detailed in the detailed theory in this chapter and the previous one. In other words, every time a function symbol or a second-order function variable appears, it will also be applicable to predicate symbols and second-order predicate variables, with the only difference being the result will not be a term but instead an atom. The syntax works the same way.

We define second-order terms first, since first-order terms may contain them.

Definition 6.1.2 (Second-order term)**.** *A* second-order term *has an associated* arity *(a natural number, including zero) and is either:*

- *A function symbol, with its associated arity.*
- *A projection*[2] π_i^n*, where* $n \geq i$*, with arity* n*.*
- *A second-order variable, with its associated arity.*
- *A* composition*, formed by a* head *(another second-order term) and a sequence of* arguments *(other second-order terms). The number of arguments must be equal to the arity of the head, and the arities of all arguments must be equal*[3]*. We write*[4] $\phi_0\{\phi_1, ..., \phi_n\}$ *to represent the composition of head* ϕ_0 *with arguments* $\phi_1, ..., \phi_n$*. Its arity is equal to the arity of its arguments.*

Note that because we allow compositions of zero arguments and arities must be well defined, the arity of a composition of zero arguments could, in principle, not be well defined. That is, $f\{\}$ could have any arity, since all its arguments (none) share the same arity, and any such arity works. To prevent this, we consider the arity of a

[2]We do not define semantics for these yet, but projections represent the selection of an argument.

[3]The *arguments* in a composition are the inner-most functions, each of them ultimately applied to the same set of actual first-order arguments, which is why they must have the same arity.

[4]The notation $\phi_0(\alpha_1, ..., \alpha_n)$ is reserved for application of a second-order term to a set of first-order terms. Composition in mathematics is normally represented with $\circ$, but this only works naturally for composition of unary functions. Thus the use of the curly brackets $\{\}$.

composition to be defined at the composition itself, and a composition is only well defined if it matches the arity of all its arguments. Therefore, we will write $f\{\}^n$ to indicate that the arity of the composition is n, and therefore it is different from $f\{\}^m$, if $n \neq m$.

In all other cases, the arity will be well defined by the context, but whenever we may want to be explicit about the arity of a second-order term, we will use superscripts as well to indicate it.

While we have not introduced semantics yet, readers familiar with higher-order functions may wonder whether abstractions are or should be allowed in the syntax of second-order terms (specially when considering instantiations for second-order variables). For example, replacing second-order variable F with $(\lambda X.f(g(),X))$. Our term system is simple enough (purely syntactic) that every second-order term/function that could be achieved via abstraction can be achieved just with composition. For example, $(\lambda X.f(g(),X)) \equiv f\{g\{\}^1,\pi^1_1\}$. Keep in mind that we are restricted strictly to second-order, with no third- or higher-order functions. Therefore, abstractions do not need to be considered in any way other than at this point of the document, but every result presented here includes anything that could be produced by adding abstractions to our definition of second-order terms.

We write $\phi \equiv \psi$ to denote *syntactic equality* of ϕ and ψ. That is, the two second-order terms are formed in exactly the same way and are indistinguishable as per the definition of second-order term.

Definition 6.1.3 (Ground second-order term)**.** *A second-order term is* ground *if it contains no second-order variables. That is, it is not a second-order variable and, if it is a composition, both its head and its arguments are inductively ground.*

Definition 6.1.4 (Second-order normal form)**.** *A second-order term is in* normal form *when it is either:*

- *Not a composition.*
- *It is a composition, but then all of the following are true:*
  - *Its head is not a composition nor a projection*
  - *All its arguments are inductively in normal form*

  - *One of the following holds:*
    * *There is an i for which the i-th argument is not* π_i^m
    * *The arity of the head is different from the arity of the arguments.*

In essence, second-order normal form is an inductive definition that states that normal second-order terms can only be compositions when that is the only way to express them. Specifically:

- A composition with a composition as head can always be simplified.
- A composition with a projection as head can always be simplified.
- A composition whose arguments are all ordered projections of the same arity as the head is equivalent to the head itself, and thus can be simplified.

These notions are expressed formally in the rewrite system below. For now, here are some examples of **not normal** second-order terms:

- $\phi_0^2\{\phi_1^3, \phi_2^3\}\{\psi_1^n, \psi_2^n, \psi_3^n\}$ is not normal because its head ($\phi_0^2\{\phi_1^3, \phi_2^3\}$) is a composition, which means one can *dump* the head's arguments into the main arguments:

$$\phi_0^2\{\phi_1^3, \phi_2^3\}\{\psi_1^n, \psi_2^n, \psi_3^n\} \xrightarrow{*} \phi_0^2\{\phi_1^3\{\psi_1^n, \psi_2^n, \psi_3^n\}, \phi_2^3\{\psi_1^n, \psi_2^n, \psi_3^n\}\}$$

- $\pi_1^2\{\phi_1^n, \phi_2^n\}$ is not normal because we can simply apply the projection:

$$\pi_2^2\{\phi_1^n, \phi_2^n\} \xrightarrow{*} \phi_2^n$$

- $\phi^3\{\pi_1^3, \pi_2^3, \pi_3^3\}$ is not in normal form because its arguments are all ordered projections, and thus we can reduce it to its head.

$$\phi^3\{\pi_1^3, \pi_2^3, \pi_3^3\} \xrightarrow{*} \phi^3$$

- $\phi_0^2\{\phi_1^2\{\pi_1^2, \pi_2^2\}, \phi_2^2\}$ is not in normal form because one of its arguments ($\phi_1^2\{\pi_1^2, \pi_2^2\}$) is not inductively in normal form and thus we can inductively reduce it:

$$\phi_0^2\{\phi_1^2\{\pi_1^2, \pi_2^2\}, \phi_2^2\} \xrightarrow{*} \phi_0^2\{\phi_1^2, \phi_2^2\}$$

and here are some examples of **normal** second-order terms:

- ϕ^2 and π_1^2 are in normal form because they are not compositions.

- $\phi_0^2\{\phi_1^2, \pi_1^2\}$ is in normal form because its head is not a composition, all its arguments are in normal form and not all its arguments are ordered projections
- $\phi^2\{\pi_1^3, \pi_2^3\}$ is in normal form because, while all its arguments are ordered projections, they have different arity. This means that $\phi^2\{\pi_1^3, \pi_2^3\}$ has arity 3 (takes 3 arguments), whereas its head, ϕ^2, has arity 2 (takes 2 arguments), and thus one cannot reduce one to the other (this would make arity change with reduction or not be well defined).

All of this becomes formally defined with the notion of *second-order term equivalence*.

Definition 6.1.5 (Second-order term equivalence)**.** *We first define the following rewrite rules (see §3.4) of* directly reducible *second-order terms, written* $\to$*:*

- $\phi_0\{\pi_1^n, ..., \pi_n^n\} \to \phi_0$. *We call this rule* head simplification.
- $\pi_i^n\{\phi_1, ..., \phi_n\} \to \phi_i$. *We call this rule* projection simplification.
- $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}\{\psi_1, ..., \psi_m\} \to \phi_0^n\{\phi_1^m\{\psi_1, ..., \psi_m\}, ..., \phi_n^m\{\psi_1, ..., \psi_m\}\}$. *We call this rule* function dumping.

We then define the reducibility relation*, written* $\overset{*}{\to}$*, between second-order terms to be the closure of direct reducibility under the following properties:*

- *Reflexivity* $\forall t.t \overset{*}{\to} t$.
- *Transitivity* $\forall r, s, t.(r \overset{*}{\to} s \wedge s \overset{*}{\to} t) \implies r \overset{*}{\to} t$
- *Algebraic closure over the structure second-order term terms. That is, for every production rule for this algebraic structures that is formed from smaller elements, if each of the smaller elements reduce to something else, then the larger term reduces to the result of substituting these for their corresponding reduced forms.*

The equivalence relation*, written* $\cong$*, between second-order terms is the symmetric closure of* $\overset{*}{\to}$*, and is a congruence relation [Wikipedia contributors, 2022a] by definition.*

We write $\overset{i}{\to}$ *when we wish to indicate direct, reflexive or inductive, but not transitive, reduction*[5].

[5] We do allow the argument or head being reduced to be reduced transitively in this step, but the outer-most reduction is only inductive with no transitivity involved.

The set of equivalence classes of second-order terms defines an algebraic structure typically called an *abstract clone* [Kerkhoff et al., 2014], defined precisely by the equivalence induced by the rewrite rules that we have used to define second-order term equivalence ($\cong$).

Theorem 6.1.1 (Normalization of second-order terms)**.** *Every second-order term ϕ is equivalent to a unique normal second-order term $\mathcal{N}(\phi)$.*

Proof. We use standard techniques for rewriting systems. This proof is long and very detailed, but does not offer any major insights and is quite straightforward from an intuitive point of view. Thus, the proof can be found in theorem B.0.1 in the appendix. □

Corollary 6.1.1. *Two second-order terms ϕ and ψ are equivalent if and only if they have the same normal form.*

Proof. By the theorem, we know there is a unique equivalent normal form for $\phi \cong \mathcal{N}(\phi)$, and a unique equivalent normal form for $\psi \cong \mathcal{N}(\psi)$.

So, if they have the same normal form, $\mathcal{N}(\phi) \equiv \mathcal{N}(\psi)$, and therefore $\phi \cong \psi$. Conversely, if $\phi \cong \psi$ then $\mathcal{N}(\phi) \cong \mathcal{N}(\psi)$, but then $\phi \cong \mathcal{N}(\psi)$ and $\psi \cong \mathcal{N}(\phi)$, and we know that normal forms are unique. So it must be $\mathcal{N}(\phi) \equiv \mathcal{N}(\psi)$. □

Definition 6.1.6 (First-order term)**.** *A* first-order term *is either:*

- *A first-order variable.*
- *An* application*, formed by a* head *(a second-order term) and a sequence of* arguments *(other first-order terms). The number of arguments must be equal to the arity of the head. We write $\phi(\alpha_1, ..., \alpha_n)$ to represent the application of head ϕ to arguments $\alpha_1, ..., \alpha_n$.*

We write $\alpha \equiv \beta$ to denote *syntactic equality* of α and β. That is, the two first-order terms are formed in exactly the same way and are indistinguishable as per the definition of first-order term.

Definition 6.1.7 (Ground first-order term)**.** *A first-order term is* ground *if it contains no first-order variables. That is, if it is not a first-order variable and, if it is an application, both its head and its arguments are recursively ground.*

Definition 6.1.8 (First-order normal form)**.** *A first-order term is in* normal form *if it is either a first-order variable, or its head is a function symbol or a second-order variable; and all its arguments are recursively in normal form.*

Definition 6.1.9 (First-order equivalence)**.** *We first define the following rewrite rules (see §3.4) of* directly reducible *first-order terms, written* $\rightarrow$*:*

- $\pi_i^n(\alpha_1, ..., \alpha_n) \rightarrow \alpha_i$. *We call this rule* projection simplification.
- $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}(\alpha_1, ..., \alpha_m) \rightarrow \phi_0^n(\phi_1^m(\alpha_1, ..., \alpha_m), ..., \phi_n^m(\alpha_1, ..., \alpha_m))$. *We call this rule* function dumping.

We then define the reducibility relation*, written* $\xrightarrow{*}$*, between first-order terms to be the closure of direct reducibility under the following properties:*

- *Reflexivity* $\forall t.t \xrightarrow{*} t$.
- *Transitivity* $\forall r, s, t.(r \xrightarrow{*} s \wedge s \xrightarrow{*} t) \implies r \xrightarrow{*} t$
- *Algebraic closure over the structure second-order and first-order terms. That is, for every production rule for these algebraic structures that is formed from smaller elements, if each of the smaller elements reduce to something else, then the larger term reduces to the result of substituting these for their corresponding reduced forms.*

The equivalence relation*, written* $\cong$*, between first-order terms is the symmetric closure of* $\xrightarrow{*}$*, and is a congruence relation [Wikipedia contributors, 2022a] by definition.*

We write $\xrightarrow{i}$ *when we wish to indicate direct, reflexive or inductive, but not transitive, reduction*[6].

Similarly to second-order equivalence, the set of equivalence classes of first-order terms is a clone [Kerkhoff et al., 2014] over the abstract clone of second-order terms.

Theorem 6.1.2 (Normalization of first-order terms)**.** *Every first-order term* α *is equivalent to a unique normal first-order term* $\mathcal{N}(\alpha)$.

[6] We do allow the argument or head being reduced to be reduced transitively in this step, but the outer-most reduction is only inductive with no transitivity involved.

Proof. We use standard techniques for rewriting systems. This proof is long and very detailed, but does not offer any major insights and is quite straightforward from an intuitive point of view. Thus, the proof can be found in theorem B.0.2 in the appendix. □

Corollary 6.1.2. *Two first-order terms* α *and* β *are equivalent if and only if they have the same normal form.*

Proof. By the theorem, we know there is a unique equivalent normal form for $\alpha \cong \mathcal{N}(\alpha)$, and a unique equivalent normal form for $\beta \cong \mathcal{N}(\beta)$.

So, if they have the same normal form, $\mathcal{N}(\alpha) \equiv \mathcal{N}(\beta)$, and therefore $\alpha \cong \beta$. Conversely, if $\alpha \cong \beta$ then $\mathcal{N}(\alpha) \cong \mathcal{N}(\beta)$, but then $\alpha \cong \mathcal{N}(\beta)$ and $\beta \cong \mathcal{N}(\alpha)$, and we know that normal forms are unique. So it must be $\mathcal{N}(\alpha) \equiv \mathcal{N}(\beta)$. □

Theorem 6.1.3 (Extensionality of second-order terms)**.** *Two second-order terms* ϕ_1 *and* ϕ_2*, both with arity m, are equivalent if and only for all sequences of first-order terms* $\alpha_1, ..., \alpha_m$*,* $\phi_1(\alpha_1, ..., \alpha_m)$ *and* $\phi_2(\alpha_1, ..., \alpha_m)$ *are equivalent.*

Proof. This proof is relatively long, standard and does not provide relevant intuitive insights and thus it can be found in theorem B.0.3 in the appendix.

□

6.1.2 Substitution / instantiation

We use the word *substitution* for replacement of first-order terms, and the word *instantiation* for replacement of second-order terms. The reason for this is that the way in which we treat second-order and first-order variables are fundamentally different. We do not produce second-order unifiers, but rather multiple first-order unifier solutions, each of which is associated to second-order instantiations. So, in a sense, we consider unifiers to act **after** instantiation. This is the way in which we approach the problem, and the way in which we define things. It does not mean it is the only mathematical way to see it.

The ideas in this subsection are particular variations / implementations of common ideas specifically suited for our problem. The fundamental notion of a substitution or instantiation (also sometimes called unifier) is explicitly and implicitly omnipresent in the literature. A standard place to start would be [Robinson and Voronkov, 2001, Bundy, 1983].

Definition 6.1.10 (Substitution)**.** *A* substitution *is an assignment of first-order terms to first-order variables. If not indicated explicitly, a substitution leaves a first-order variable as is (replaces it with itself).*

Note that substitutions with cyclic references[7] *are considered invalid.*

A substitution may be applied to a first-order term, replacing each appearance of first-order variables in them with their assignments under the substitution. We write $\sigma_i(\alpha)$ *to mean the application of substitution* σ_i *to first-order term* α*.*

Definition 6.1.11 (Ground substitution)**.** *A substitution is* ground *(with respect to a context) if it replaces all first-order variables present in the context (a formula, or an entire signature) with ground terms.*

Definition 6.1.12 (Composition of substitutions)**.** *Since substitutions can be applied to first-order terms, substitutions can be* composed*, producing new substitutions. We write the usual* $\sigma_i \circ \sigma_j$ *to express the substitution result of applying* σ_j *first, and then* σ_i*.*

Definition 6.1.13 (Finer substitutions)**.** *We say that a substitution* $\sigma_i \circ \sigma_j$ *is* finer *than* σ_j*, written* $\sigma_i \circ \sigma_j \preceq \sigma_j$*, for any* σ_i *and* σ_j*.*

We say that σ_i *is* strictly finer *than* σ_j*, written* $\sigma_i \prec \sigma_j$*, if* σ_i *is finer than* σ_j *and* σ_j *is not finer than* σ_i*.*

Definition 6.1.14 (Instantiation)**.** *An* instantiation *is an assignment of second-order terms to second-order variables, analogously to substitutions for first-order terms, with the additional constraint that the arity of the second-order term each variable is assigned to must be equal to the arity of the second-order variable.*

Similarly to substitutions, instantiations with cyclic references are not allowed. Also similarly, instantiations can be applied to second-order terms, replacing each appearance of second-order variables in them with their assignments in the instantiation. Moreover, instantiations can be applied to first-order terms, replacing each appearance of second-order variables in them with their assignments. We write $I(\phi)$ *to mean the application of instantiation I to second-order term* ϕ*, and similarly* $I(\alpha)$ *to mean the application of instantiation I to first-order term* α*.*

[7] A substitution σ with cyclic references can be equivalently defined (without a need to consider alpha-equivalence) as a substitution such that for all natural numbers n, $\sigma^{n+1} \neq \sigma^n$.

Definition 6.1.15 (Ground instantiation)**.** *An instantiation is* ground *(with respect to a context) if it replaces all second-order variables in the context (a formula or an entire signature) with ground second-order terms.*

Definition 6.1.16 (Composition of instantiations)**.** *Since instantiations can be applied to first and second-order terms, instantiations can be* composed*, producing new instantiations. We write the usual $I_1 \circ I_2$ to express the instantiation result of applying I_2 first, and then I_1.*

Definition 6.1.17 (Finer instantiations)**.** *We say that an instantiation $I_1 \circ I_2$ is* finer *than I_2, written $I_1 \circ I_2 \preceq I_2$, for any I_1 and I_2.*

We say that I_1 is strictly finer *than I_2, written $I_1 \prec I_2$, if I_1 is finer than I_2 and I_2 is not finer than I_1.*

6.1.3 Unifier expressions

An important aspect of our unification algorithm is that it deals with *symbolic unifiers*. To do this, it utilizes *unifier variables* that can be added to first-order terms, extending their language. In this subsection we formalize these notions.

While the general idea of utilizing variables to stand for unifiers / substitutions is natural and has precedents in the literature, our particular approach is novel and has important aspects that are critical to the dependency graph unification algorithm, such as the ordering of unifier variables in expressions. The most prominent field in which unifier variables (often called *binders*) appears is *nominal unification* [Urban et al., 2004, Schmidt-Schauß et al., 2019, Levy and Villaret, 2010, Calvès, 2013]. The definitions presented here clearly fall within this general notion and most of the theory in the literature in this regard has a direct effect here. However, our problem is a much simpler particular case for which we identify no significant benefit in utilizing all the explicit theoretical firepower of nominal unification literature: ultimately, our problem is only a slight extension of first-order unification in this regard. Moreover, and in a transversal direction, the presence of existential second-order variables in our problem complicates things in a different way that standard nominal unification is only prepared to tackle with full higher-order expressivity, which we are trying to avoid (see chapter 5). In chapter 10 we discuss the relation between nominal unification and our work in a bit more detail.

Definition 6.1.18 (Unifier variable)**.** *We may augment a second-order signature to a* second-order unification signature *by adding a sequence of* unifier variables *to it. We usually write* $\sigma_1, \sigma_2, \ldots$ *to represent unifier variables.*

We use the same notation σ for unifier variables as we do for substitutions because unifier variables are meant to be replaced by substitutions. This can arguably be slightly confusing at times, but in every situation it should be clear whether we are talking about a specific substitution or we are simply using a unifier variable that has not been replaced by a substitution yet.

Note that unifier variables represent unification steps in, for example, a resolution proof (§3.2.1.2). Therefore, each unifier variable is to be replaced by a substitution, and moreover, the indices associated with the unifier variables indicate the order in which these substitutions are applied (and these have a strict ordering). Also note that, in the context of resolution proofs, after each unifier we may have to introduce new first-order variables (a process called *standardization* in resolution proofs (§3.2.1.2)). This accounts for the universally quantified nature of first-order variables and the assumption (true in resolution) that the two terms being unified come from conjunctively joined sub-formulas which therefore can be thought of as having separate variable sets for full generality of the subsequent unifications.

The formalization of the notions explained here, while present in this thesis, do not come until later on (§7.2). The intuitive explanation is included here to help the reader approach the following sections from the conceptual point of view that it is intended.

Unifier variables do not include an instantiation of second-order variables. This is a consequence of the way in which we use second-order variables: they are existentially quantified outside the scope of the universal quantifiers for first-order variables, and therefore it is meant to be thought that an instantiation includes an instantiation of the unifier variables themselves: for each valid instantiation of second-order variables, there is an associated set of substitutions for the unifier variables, that ultimately apply substitutions to the first-order variables. This is one of the fundamental simplifications that make our problem a proper sub-case of general second-order unification (§3.2.2), and which we exploit intensely in the algorithm presented.

Next we define the notion of *unifier expressions*, which combine unifier variables with first-order terms to represent a first-order term, but including also the dependence

on the substitutions with which unifier variables are replaced.

For example, $\sigma_2\sigma_1\alpha$ represents *the first-order term that will result after applying substitutions* σ_1 *and then* σ_2 *to the first-order term* α, where σ_1 and σ_2 are variables and not replaced by specific substitutions yet.

Note that our algorithm considers unifier expressions explicitly, so they are not just a theoretical tool but rather an embodied concept in our approach.

Definition 6.1.19 (Unifier expression)**.** *We define the algebraic structure of a* unifier expression *at the same time that we define its associated* unifier level *(a natural number, or* $\bot$ *for certain expressions that have no unifier level), and an ordering (*$>$*) between these unifier levels. We write* $\#_\sigma(\varepsilon)$ *to refer to the unifier level of expression* ε*, where* $\#_\sigma(\varepsilon) \in \{\bot, 0\} \cup \mathbb{N}$*. We also define* $i > 0 > \bot$*, for every unifier level* $i \in \mathbb{N}$*. A* unifier first-order expression *is either:*

1. *A first-order variable. In this case its unifier level is* 0*.*
2. *An application formed by a second-order term* ϕ*, with arity m, applied to a set of other unifier expressions* $\varepsilon_1, ..., \varepsilon_m$*:* $\phi(\varepsilon_1, ..., \varepsilon_m)$*, where each* ε_i *that has a unifier level (i.e. not* $\bot$*) has the same unifier level. Its unifier level is the same as that of the* ε_i *that have unifier level, or* $\bot$ *if no arguments have a unifier level. For consistency, we will call* ϕ *the* head *and the* ε_i *the* arguments *of the expression.*
3. *A substitution, formed by a unifier variable* σ_i *applied to a unifier expression* ε*:* $\sigma_i\varepsilon$*, where* $i > \#_\sigma(\varepsilon)$ *(we consider every i to be* $i > \bot$*). Its unifier level is i. We call* σ_i *the* unifier variable *of* $\sigma_i\varepsilon$ *and* ε *its* sub-expression*.*

Some examples of valid unifier expressions include:

$$\begin{array}{lll} \varepsilon_1 \equiv X & \varepsilon_2 \equiv \sigma_1 X & \varepsilon_3 \equiv \sigma_3\sigma_1 X \\ \varepsilon_4 \equiv f(X,Y) & \varepsilon_5 \equiv \sigma_3\sigma_2 f(X,Y) & \varepsilon_6 \equiv f(\sigma_3\sigma_2 X, \sigma_3\sigma_2 Y) \\ \varepsilon_7 \equiv f() & \varepsilon_8 \equiv f(g(), f(g())) & \varepsilon_9 \equiv f(g(), f(\sigma_2 X, \sigma_2 X)) \end{array}$$

where

$$\begin{array}{lll} \#_\sigma(\varepsilon_1) = 0 & \#_\sigma(\varepsilon_2) = 1 & \#_\sigma(\varepsilon_3) = 3 \\ \#_\sigma(\varepsilon_4) = 0 & \#_\sigma(\varepsilon_5) = 3 & \#_\sigma(\varepsilon_6) = 3 \\ \#_\sigma(\varepsilon_7) = \bot & \#_\sigma(\varepsilon_8) = \bot & \#_\sigma(\varepsilon_9) = 2 \end{array}$$

Note that in this definition, we rely on an ordering of the unifier variables $\{\sigma_i\}$, associated to its indices $\{i\}$. The fundamental formal properties here are that unifier

variables have a specific increasing ordering that denotes which ones may appear to the left of which other ones in unifier expressions, and that we have two special unifier levels: 0 to denote expressions with no unifier variables, and $\perp$ to denote expressions with no unifier variables **and** no first-order variables (not affected by unifier variables), i.e. only first-order functions and constants. We use the indices $\{i\}$ to denote this, such that for any expression of the form $\sigma_j\sigma_i\alpha$, $j > i$.

Definition 6.1.20 (Equivalence of unifier expressions)**.** *We define the following rewrite rules (see §3.4) of* directly reducible *unifier expressions, written $\rightarrow$:*

- $\pi_i^n(\varepsilon_1,...,\varepsilon_n) \rightarrow \varepsilon_i$. *We call this rule* projection simplification.
- $\phi_0^n\{\phi_1^m,...,\phi_n^m\}(\varepsilon_1,...,\varepsilon_n) \rightarrow \phi_0^n(\phi_1^m(\varepsilon_1,...,\varepsilon_m),...,\phi_n^m(\varepsilon_1,...,\varepsilon_m))$. *We call this rule* function dumping.
- $\sigma_i\phi(\varepsilon_1,...,\varepsilon_m) \rightarrow \phi(\sigma_i\varepsilon_1,...,\sigma_i\varepsilon_m)$. *We call this rule* unifier variable dumping.

where

- π_i^n *is the i-th projection of arity n, as defined in definition 6.1.2.*
- $\varepsilon_1,...,\varepsilon_n$ *are any n unifier expressions, as defined in definition 6.1.19.*
- ϕ_0^n *is any second-order term of arity n, and* $\phi_1^m,...,\phi_n^m$ *are any n second-order terms of arity m, as defined in definition 6.1.2.*
- σ_i *is the i-th unifier variable, as defined in 6.1.19.*

We then define the reducibility relation*, written $\overset{*}{\rightarrow}$, between unifier expressions to be the closure of direct reducibility under the following properties:*

- *Reflexivity* $\forall t.t \overset{*}{\rightarrow} t$.
- *Transitivity* $\forall r,s,t.(r \overset{*}{\rightarrow} s \wedge s \overset{*}{\rightarrow} t) \implies r \overset{*}{\rightarrow} t$
- *Algebraic closure over the structure of unifier expressions, first-order terms and second-order term heads. That is, for every production rule for these algebraic structures that is formed from smaller elements, if each of the smaller elements reduce to something else, then the larger term reduces to the result of substituting these for their corresponding reduced forms.*

The equivalence relation*, written* $\cong$*, between unifier expressions is the symmetric closure of* $\xrightarrow{*}$*, and is a congruence relation [Wikipedia contributors, 2022a] by definition.*

We write $\xrightarrow{i}$ *when we wish to indicate direct, reflexive or algebraic, but not transitive, reduction.*

Lemma 6.1.1. *Let* $\varepsilon_1 \xrightarrow{*} \varepsilon_2$*. Then, at least one of the following must be true:*

- $\#_\sigma(\varepsilon_2) = \bot$.
- $\#_\sigma(\varepsilon_1) = \#_\sigma(\varepsilon_2)$.

Proof. We need to consider reflexivity, transitivity, inductive reduction on subexpressions and direct reduction rules.

- **Reflexivity** - Clearly, an expression has the same unifier level as itself.
- **Transitivity** - $\varepsilon_1 \xrightarrow{*} \varepsilon_3$ and $\varepsilon_3 \xrightarrow{*} \varepsilon_2$. We may inductively assume that the lemma holds for both of these reductions. Then, one of the following holds:
  - $\#_\sigma(\varepsilon_2) = \bot$. Then the transitive case is proven.
  - $\#_\sigma(\varepsilon_3) = \#_\sigma(\varepsilon_2)$ and $\#_\sigma(\varepsilon_3) = \bot$. Then $\#_\sigma(\varepsilon_2) = \bot$ and the transitive case is proven.
  - $\#_\sigma(\varepsilon_3) = \#_\sigma(\varepsilon_2)$ and $\#_\sigma(\varepsilon_1) = \#_\sigma(\varepsilon_3)$, in which case $\#_\sigma(\varepsilon_1) = \#_\sigma(\varepsilon_2)$ and the transitive case is proven.
- **Inductive reduction** - Consider the possible shapes of ε_1:
  - ε_1 is a first-order variable. No inductive reduction is possible here.
  - $\varepsilon_1 \equiv \phi(\delta_1, ..., \delta_n)$, where $\delta_i \xrightarrow{*} \delta_i^2$ and $\varepsilon_2 \equiv \phi(\delta_1, ..., \delta_i^2, ..., \delta_n)$. Consider the possibilities:
    - $\#_\sigma(\delta_i) = \bot$. Then, inductively, $\#_\sigma(\delta_i^2) = \bot$, which means the unifier level of ε_2 is the same as ε_1.
    - $\#_\sigma(\delta_i^2) = \bot$ and $\#_\sigma(\varepsilon_1) \neq \bot$, either there was another δ_j with $\#_\sigma(\delta_j) \neq \bot$, and so $\#_\sigma(\varepsilon_2) = \#_\sigma(\varepsilon_1) = \#_\sigma(\delta_j)$; or $\#_\sigma(\varepsilon_2) = \bot$. The lemma is true in all cases.
    - It is not possible that $\#_\sigma(\delta_i) \neq \bot$ and $\#_\sigma(\varepsilon_1) = \bot$.

  - $\#_\sigma(\delta_i^2) = \#_\sigma(\delta_i)$. In such case, $\#_\sigma(\varepsilon_2) = \#_\sigma(\varepsilon_1)$.
- $\varepsilon_1 = \sigma_i\delta_1$, with $\delta_1 \stackrel{*}{\to} \delta_2$, and $\varepsilon_2 = \sigma_i\delta_2$. Regardless of δ_1 and δ_2, by definition $\#_\sigma(\varepsilon_1) = i = \#_\sigma(\varepsilon_2)$.

- **Direct reduction** - Consider the three direct reduction rules:

  - Projection simplification. The arguments of the expression, by definition, have unifier level either $\perp$ or the same unifier level as the whole expression, so reducing the whole expression to one of its arguments preserves the unifier level or reduces it to $\perp$.
  - Function dumping. $\phi_0^n\{\phi_1^m,...,\phi_n^m\}(\varepsilon_1,...,\varepsilon_m)$ has as unifier level the unifier level of the ε_i that are not $\perp$. Similarly, the reduced $\phi_0^n(\phi_1^m(\varepsilon_1,...,\varepsilon_m),...,\phi_n^m(\varepsilon_1,...,\varepsilon_m))$ has as unifier level the unifier level of the $\phi_j^m(\varepsilon_1,...,\varepsilon_m)$ that are not $\perp$, each of which has as unifier level the unifier level of the ε_i that are not $\perp$, so the unifier level is preserved. If $n = 0$, then the unifier level of $\varepsilon_2 \equiv \phi_0^0()$ is $\perp$. If $m = 0$, then the unifier level of $\varepsilon_2 \equiv \phi_0^n(\phi_1^0(),...,\phi_n^0())$ is the unifier level of each $\phi_i^0()$, which is $\perp$, and so the unifier level of ε_2 is $\perp$.
  - $\sigma_i\phi(\varepsilon_1,...,\varepsilon_m)$ has as unifier level i. Similarly, $\phi(\sigma_i\varepsilon_1,...,\sigma_i\varepsilon_m)$ has as unifier level the unifier level of the $\sigma_i\varepsilon_j$ that are not $\perp$, but they are all i. The only exception case is when $m = 0$, in which case $\sigma_i\phi() \stackrel{*}{\to} \phi()$, so the unifier level is reduced from i to $\perp$.

□

Lemma 6.1.2. *If a unifier expression* ε *has unifier level* $\#_\sigma(\varepsilon) = \perp$*, then for any* $i_1, i_2, ..., i_n > 0$, $\varepsilon \cong \sigma_{i_n}...\sigma_{i_2}\sigma_{i_1}\varepsilon$.

Proof. The way we prove this is by showing that $\sigma_i\varepsilon \stackrel{*}{\to} \varepsilon$, for all $i > 0$, if $\#_\sigma(\varepsilon) = \perp$. Thus, we can reduce $\sigma_{i_1}\varepsilon \stackrel{*}{\to} \varepsilon$, and so we can reduce $\sigma_{i_2}\sigma_{i_1}\varepsilon \stackrel{*}{\to} \sigma_{i_2}\varepsilon \stackrel{*}{\to} \varepsilon$; and so on for all $i_1, i_2, ..., i_n$.

To prove that $\sigma_i\varepsilon \stackrel{*}{\to} \varepsilon$, consider what ε might be. If $\#_\sigma(\varepsilon) = \perp$, then necessarily ε is of the shape $\phi_0^m(\varepsilon_1,...,\varepsilon_m)$, where each ε_j has $\#_\sigma(\varepsilon_j) = \perp$. We proceed inductively on the ε_j. There are two possibilities:

- $m = 0$. Then $\varepsilon \equiv \phi_0^0()$, and then $\sigma_i\varepsilon \equiv \sigma_i\phi_0^0() \stackrel{*}{\to} \phi_0^0() \equiv \varepsilon$.

- $m \neq 0$. Then $\sigma_i\varepsilon \equiv \sigma_i\phi_0^m(\varepsilon_1, ..., \varepsilon_m) \xrightarrow{*} \phi_0^m(\sigma_i\varepsilon_1, ..., \sigma_i\varepsilon_m)$. But we may inductively assume that each $\sigma_i\varepsilon_j \xrightarrow{*} \varepsilon_j$, and so $\sigma_i\varepsilon \xrightarrow{*} \phi_0^m(\varepsilon_1, ..., \varepsilon_m) \equiv \varepsilon$.

□

Lemma 6.1.3. *If a unifier expression* ε *has as unifier level* $i > 0$*, then it is equivalent to an expression of the form* $\sigma_i\delta$*.*

Proof.

- If ε were a first-order variable, then it would have unifier level 0, so ε cannot be a first-order term.
- If ε is of the form $\sigma_i\delta$, then it already is of the desired form.
- The only case left is if ε is of the form $\phi(\varepsilon_1, ..., \varepsilon_n)$.

  By definition, ε has the same unifier level as the ε_j whose unifier level is not $\perp$. Therefore, we can recursively assume that there is an ε_j whose unifier level is not $\perp$ and that it is equivalent to $\sigma_i\delta_j$ for some δ_j. For those ε_j such that $\#_\sigma(\varepsilon_j) = \perp$, by lemma 6.1.2, we have $\varepsilon_j \cong \sigma_i\varepsilon_j$. Define Δ_j to be δ_j for those ε_j whose level is not $\perp$, and $\Delta_j = \varepsilon_j$ for those whose level is $\perp$. By inductive equivalence, ε is then equivalent to $\phi(\sigma_i\Delta_1, ..., \sigma_i\Delta_n)$, which in turn is directly equivalent to $\sigma_i\phi(\Delta_1, ..., \Delta_n)$; and we have proven the lemma.

□

Definition 6.1.21 (Unifier expression normal form)**.** *Unifier expression normal form is defined as an extension of first-order normal form to unifier expressions.*

To do so, we first define the utility notion of function free unifier expression*. A first-order term with no unifier variables is function free if it is a first-order variable. A unifier expression of the form* $\sigma_i\varepsilon$ *is function free if* ε *is function free. In any other case, it is not function free.*

We then say a unifier expression is normal if it is function free or if it is an application whose head is a function symbol or a second-order variable; and all its arguments are recursively in normal form.

This definition is very similar to the definition of normal first-order terms (definition 6.1.8), where any unifier variables have been pushed inwards as much as possible.

For example:

- X is normal because it is function free.
- $\sigma_2\sigma_1 X$ is normal because it is function free.
- $f(X)$ is normal because it is an application, its head (f) is a function symbol, and its argument (X) is recursively normal.
- $F(X)$ is normal because it is an application, its head (F) is a second-order variable, and its argument (X) is recursively normal.
- $f(\sigma_2\sigma_1 X)$ is normal because it is an application, its head (f) is a function symbol, and its argument ($\sigma_2\sigma_1 X$) is recursively normal.
- $f(g(X))$ is normal because it is an application, its head (f) is a function symbol, and its argument ($g(X)$) is recursively normal.
- $\pi_1^2(X,Y)$ is not normal because its head (π_1^2) is a projection.
- $f\{g\}(X)$ is not normal because its head ($f\{g\}$) is a composition.
- $\sigma_1 f(X)$ is not normal because it is not function free and it is not an application.

Theorem 6.1.4 (Normalization of unifier expressions)**.** *Every unifier expression* ε *is equivalent to a unique normal unifier expression* $\mathcal{N}(\varepsilon)$.

Proof. We use standard techniques for rewriting systems. This proof is long and very detailed, but does not offer any major insights and is quite straightforward from an intuitive point of view. Thus, the proof can be found on theorem B.0.4 in the appendix. □

Corollary 6.1.3. *Two unifier expressions* ε *and* δ *are equivalent if and only if they have the same normal form.*

Proof. By the theorem, we know there is a unique equivalent normal form $\varepsilon \cong \mathcal{N}(\varepsilon)$, and a unique equivalent normal form $\delta \cong \mathcal{N}(\delta)$.

So, if they have the same normal form, $\mathcal{N}(\varepsilon) \equiv \mathcal{N}(\delta)$, and therefore $\varepsilon \cong \delta$. Conversely, if $\varepsilon \cong \delta$ then $\mathcal{N}(\varepsilon) \cong \mathcal{N}(\delta)$, but then $\varepsilon \cong \mathcal{N}(\delta)$ and $\delta \cong \mathcal{N}(\varepsilon)$, and we know that normal forms are unique. So it must be $\mathcal{N}(\varepsilon) \equiv \mathcal{N}(\delta)$. □

We will note that while second-order unifier expressions make conceptual sense, second-order variables are not affected by unifier variables, and so second-order unifier expressions are exactly second-order terms.

6.1.4 Unification solutions and equations

In this subsection we define unification equations that express the conditions of a unification problem, and the unification solutions that satisfy them, as well as the connections between them.

While the particular approach followed here is likely to not have been presented elsewhere, due to its particular adaptation to our problem's conditions, the basic notion of representing unification problems with equations and their associated solutions is core to the unification literature, and therefore not novel. A standard place to begin would be [Robinson and Voronkov, 2001, Bundy, 1983].

Definition 6.1.22 (Unification solution)**.** *Given a signature, including a sequence of unifier variables σ_i, we define a* unification solution *U to consist of a single instantiation and a substitution associated with each unifier variable. We write I^U for the instantiation of U and σ_i^U for the substitution associated with unifier variable σ_i of U.*

We say a unification solution is ground *if its instantiation is ground.*

Definition 6.1.23 (Finer solution)**.** *We say a unification solution U_1 is* finer *than another unification solution U_2, written $U_1 \preceq U_2$, if $I^{U_1} \preceq I^{U_2}$ and for every unifier variable σ_i, $\sigma_i^{U_1} \preceq \sigma_i^{U_2}$.*

Similarly, we say U_1 is strictly finer *than U_2, written $U_1 \prec U_2$, if it is finer and at least one substitution or the instantiation is strictly finer.*

Definition 6.1.24 (Evaluation of a unifier expression)**.** *Consider a unifier expression ε and a unification solution U. We define the* evaluation *of ε over U, written $U(\varepsilon)$, to be a unifier expression in the following way:*

- *If ε is a first-order variable, then $U(\varepsilon) \equiv \varepsilon$.*
- $U(\phi(\varepsilon_1, ..., \varepsilon_m)) \equiv I^U(\phi)(U(\varepsilon_1), ..., U(\varepsilon_n))$.
- $U(\sigma_i\delta) \equiv \sigma_i^U(U(\delta))$

The reader may wonder why does the evaluation of a first-order variable correspond to the variable, and whether evaluation should not replace it with a term. The key notion to understand here is that every first-order unification effect of the unification solution

U is encoded in the substitutions σ_i^U; and similarly, every way in which the unification solution could affect a unifier expression is explicitly indicated by the unifier variables σ_i. In other words, $U(X) \equiv X$ for every unification solution X and every first-order variable X, but $U(\sigma_1 X)$ depends on the specific substitution σ_1^U that U contains. This dependence is included explicitly in the syntax of unifier expressions.

Lemma 6.1.4. *For every unification solution U, $U(\sigma_i\sigma_j\varepsilon) \equiv (\sigma_i^U \circ \sigma_j^U)(U(\varepsilon))$.*

Proof. By definition,

$$U(\sigma_i\sigma_j\varepsilon) \equiv \sigma_i^U(U(\sigma_j\varepsilon)) \equiv \sigma_i^U(\sigma_j^U(U(\varepsilon)))$$

But then $\sigma_i^U(\sigma_j^U(U(\varepsilon)) \equiv (\sigma_i^U \circ \sigma_j^U)(U(\varepsilon))$. □

Definition 6.1.25 (Equality in a solution)**.** *Given a unification solution U and two unifier expressions ε_1 and ε_2, we say ε_1 and ε_2 are* equal in U*, written $\varepsilon_1 \approx_U \varepsilon_2$, if $U(\varepsilon_1) \cong U(\varepsilon_2)$.*

Similarly, we say two second-order terms ϕ_1 and ϕ_2 are equal in U*, written $\phi_1 \approx_U \phi_2$ if $I^U(\phi_1) \cong I^U(\phi_2)$.*

Definition 6.1.26 (Unification equation)**.** *A* first-order unification equation *is an equation of the form $\varepsilon_1 \approx \varepsilon_2$, where ε_1 and ε_2 are unifier expressions.*

A unification solution U is a solution to *a first-order unification equation $\varepsilon_1 \approx \varepsilon_2$ if $\varepsilon_1 \approx_U \varepsilon_2$.*

A second-order unification equation *is an equation of the form $\phi_1 \approx \phi_2$, where ϕ_1 and ϕ_2 are second-order terms.*

A unification solution U is a solution to *a second-order unification equation $\phi_1 \approx \phi_2$ if $\phi_1 \approx_U \phi_2$.*

We call first and second-order unification equations jointly unification equations*.*

Definition 6.1.27 (Unification system)**.** *A* system of unification equations *is a set of unification equations.*

A unification solution U is a solution to *a system of unification equations if it is a solution to every equation in it.*

Given a unification equation system $\mathcal{E}$, we write $\mathcal{U}(\mathcal{E})$ to describe the set of unification solutions to $\mathcal{E}$.

Definition 6.1.28 (Equivalent equation systems)**.** *Given two unification equation systems $\mathcal{E}_1$ and $\mathcal{E}_2$, we say $\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent, written $\mathcal{E}_1 \simeq \mathcal{E}_2$, if $\mathcal{U}(\mathcal{E}_1) = \mathcal{U}(\mathcal{E}_2)$.*

The main use that we will make of equivalent unification equation systems will be to treat unification equation systems the same when they are equivalent after considering the transitivity of equations. For example, equation system $\mathcal{E}_1$:

$$\varepsilon_1 \approx \varepsilon_2$$
$$\varepsilon_2 \approx \varepsilon_3$$

is equivalent to equation system $\mathcal{E}_2$:

$$\varepsilon_1 \approx \varepsilon_3$$
$$\varepsilon_2 \approx \varepsilon_3$$

Another way that you could see this is that we consider unification equation systems "modulo their semantics". That is, we usually treat equivalent equation systems as the same system, using the most convenient representation each time.

6.1.5 Meta-CNF formulas

The concepts in this subsection extend the notion of terms and unifier expressions to consider formulas in a resolution setting, containing unifier variables. However, these extensions do not appear during the unification algorithm described in the next chapter, and instead are kept exclusively at the resolution step.

Once again, the particular ideas described here are likely to be exclusive to this work, but the basic notion is simple: Take the well known and standard concepts of atom, literal, clause and conjunctive normal form (see §3.2.1), and extend them to our particular circumstance where we have second-order variables and unification variables. It is likely similar ideas exist elsewhere in the literature, but we have not taken direct inspiration from any of them.

Definition 6.1.29 (Meta-atom)**.** *A* meta-atom *is a unifier expression (definition 6.1.19) that fills the role of an atom. Precisely, a unifier expression whose head is a predicate symbol or a predicate second-order variable.*

Syntactically, it behaves exactly like any other unifier expression, except that it may never appear as argument to another expression.

A meta-atom must always contain a composition, but it may also have unifier variables. First-order variables or unifier variables applied to first-order variables are not valid meta-atoms.

Examples of meta-atoms include $p(x), q(), P(x,y), P(F(f(x),y), f(F(y,y)))$, etc.

Definition 6.1.30 (Truth value of a meta-atom)**.** *Given a ground unification solution U that applies to all second-order variables and unifier variables in a meta-atom A, we can apply U to A to obtain a first-order atom $U(A)$.*

Then, given an interpretation I (see §3.2.1.1) of the signature, $U(A)$ will have a truth value (true or false). We call this the truth value of A under the solution U and interpretation I.

For example, a unification solution U that instantiated P to q and F to $f\{g\}$, and an interpretation of the first-order signature that made $q(f(g()))$ "true", would give a truth value of "true" to the meta-atom $P(F())$.

Definition 6.1.31 (Meta-literal)**.** *A* meta-literal *is either a meta-atom A (called a* positive meta-literal*) or a negated meta-atom $\neg A$ (called a* negative meta-literal*).*

Given a positive meta-literal A, we call $\neg A$ its negation*. Given a negative meta-literal $\neg A$, we call A its* negation*.*

Definition 6.1.32 (Truth value of a meta-literal)**.** *Given a ground unification solution U, an interpretation I, and a meta-literal L, we define the* truth-value of L under U and I*:*

- *Equal to the truth value of A under U and I if $L \equiv A$ for meta-atom A.*
- *The opposite of the truth value of A under U and I if $L \equiv \neg A$ for meta-atom A.*

Definition 6.1.33 (Meta-clause)**.** *A* meta-clause *is a disjunction of a finite number of meta-literals $L_1 \vee L_2 \vee \ldots \vee L_n$. n can be zero, in which case we call the meta-clause the* empty clause*, which can be written as $\top$.*

Definition 6.1.34 (Truth value of a meta-clause)**.** *Given a ground unification solution U, an interpretation I, and a meta-clause C, we define the* truth-value of C under U and I *to be true if there is at least one meta-literal in C which is true under U and I, and false otherwise.*

It follows that the empty clause is false for any solution and interpretation.

Definition 6.1.35 (Meta-CNF)**.** *A* meta-CNF *is a conjunction of a finite number of meta-clauses $C_1 \wedge C_2 \wedge ... \wedge C_n$. n can be zero, in which case we call the meta-CNF the* empty CNF*, which can be written as $\bot$.*

Definition 6.1.36 (Truth value of a meta-CNF)**.** *Given a ground unification solution U, an interpretation I, and a meta-CNF N, we define the* truth-value of *N* under *U* and *I to be true if every meta-clause in N is true under U and I, and false otherwise.*

It follows that the empty CNF is true for any solution and interpretation.

6.1.6 Formulas with meta-predicates

The concepts in this section formally are a simplified version of first-order logic (see §3.2.1), which however we define separately because we use them in conjunction, but separated, from actual first-order statements. In particular, we define statements that contain meta-predicates instead of predicates, first-order function and predicate symbols as first-order constants, and second-order variables as first-order variables. This is clearly not novel, neither technically (as it is a subet of first-order logic), nor as a simplified way to implement meta-predicates; and we provide it here for completeness of definitions.

We note that these formulas are a small use case particularly related with contextual knowledge (see §4.3), and despite their name, are not directly related in any way with Meta-CNFs defined above (definition 6.1.35). Meta-CNFs are more complex and contain more aspects and semantics that we explicitly use as a core part of our solving algorithm. Formulas with meta-predicates are a simple syntactic tool to define and utilise contextual knowledge.

Definition 6.1.37 (Formula with meta-predicates)**.** *A* formula with meta-predicates *is a formula of the form*

$$\delta(\phi_1, \phi_2, ..., \phi_n) \tag{6.1}$$

where δ is a meta-predicate symbol with arity n, and the ϕ_i are either first-order predicate or function symbols, or second-order variables.

The semantics is strictly that of its first-order structure. In particular, the additional semantics of first-order predicate and function symbols in the *actual first-order* structure of the logic is not relevant for formulas with meta-predicates at all.

We note that, as a first-order structure, it is simplified. In particular, **there are no function applications nor quantifiers**.

More precisely, we use formulas with meta-predicates exclusively for simple first-order unification as an aspect of queries in ESQ logic (§6.2). We add formulas with meta-predicates not containing second-order variables to the first-order theory, and then may have queries that check for the presence of a formula with meta-predicates.

For example, consider that we add the following formulas with meta-predicates to a theory:

$$\begin{array}{l} primitive(icecream) \\ primitive(pizza) \end{array} \tag{6.2}$$

and then we use a query to find instantiations of the formula with meta-predicates containing second-order variables:

$$primitive(P) \tag{6.3}$$

Then, a simplified first-order unification algorithm applied to each of the formulas with meta-predicates will yield the results $P \equiv icecream$ and $P \equiv pizza$.

We describe this in more detail in §6.2.

6.2 Existential second-order query logic

In this section we describe existential second-order query logic (ESQ logic) formally, as initially introduced in §4.3.

The formalization is used mostly in the description of the evaluation test cases as presented in chapter 9 and completed at (`https://tinyurl.com/y67nsebs`) and the pattern catalogue presented in chapter 4. In practice, they express the intuitive notions of the patterns as described in chapters 4 and 5.

All of the concepts in this section are entirely novel to this work, though obviously they build on the pieces described so far and the cumulative knowledge on automated theorem proving. In particular, the notions in this section are heavily related to constraint logic programming (§3.3.1.1). But the definitions here are new and not standard or trivial variations, to the best of our knowledge.

We will first describe a formalism for the **denotational** semantics for queries. That is,

for what their solutions should be. We then discuss some **computational** aspects about *solving* queries, though we do not formalize these here.

6.2.1 Denotational semantics

Queries represent semantic conditions having to do with the provability and satisfiability of first-order formulas with existentially quantified second-order variables in them, as well as direct unification constraints; and have an associated *solution*: The (possibly infinite) set of instantiations (definition 6.1.14) of second-order variables in the logic signature that fulfil the conditions of the query.

Definition 6.2.1 (Instantiation set)**.** *An* instantiation set *is a set of tuples (including unary) of second-order terms (definition 6.1.2).*

All the tuples must have the same arity, and each second-order term in the same position in the tuple in the same instantiation set must have the same arity.

Note, however, that because we use curly brackets $\{\}$ to indicate function composition and it can get confusing with the curly brackets used for set notation, any curly brackets inside a tuple (parenthesis) are function composition, and any curly bracket outside a tuple (parenthesis) indicates set notation. For example, $\{(f),(f\{g,h\})\}$ is an instantiation set with unary tuples.

We will be talking about *solutions* to queries given theories. In order to formalize this, we will define what a *full ground solution* to a query is on a case by case basis. Full ground solutions are always *ground instantiations* (definition 6.1.15). We can then extend this definition in general (for all types of queries) to the definition of a ground instantiation being a *ground solution* to a query, and then extending this definition to *all instantiations* (ground or not). We do this here first, in general:

Definition 6.2.2 (Ground solution of a query)**.** *We say that a ground instantiation I_S of second-order variables in S is a* ground solution *of a query Q with select clause S, given theory T, if I_S is the restriction of a full ground solution I of Q given T, only to the variables in S.*

Definition 6.2.3 (Solution of a query)**.** *We say that an instantiation I of second-order variables in S is a* solution *of a query Q with select clause S, given theory T, if, for every ground instantiation I_G that is finer than I, I_G is a ground solution to Q given T.*

We will write $\mathbb{S}_T(Q)$ to describe the set of all solutions to a query given a theory T. We say that a subset $\bar{\mathbb{S}}_T(Q) \subset \mathbb{S}_T(Q)$ is a *complete* set of solutions if for every ground solution I_G of Q given T, there is an instantiation $I \in \bar{\mathbb{S}}_T(Q)$ such that I_G is finer than I. The objective of our algorithm is to compute a complete subset of solutions $\bar{\mathbb{S}}_T(Q)$. We write $\mathbb{G}_T(Q) \subset \mathbb{S}_T(Q)$ for the set of *ground solutions* of a query only.

There are two classes of ESQ queries: *Basic queries* and *Composite queries*. We will explain basic queries first.

6.2.1.1 Basic queries

All basic queries have two fundamental elements:

- A *body*, indicating the constraint that defines the instantiation set.
- A *select clause*, indicating the instantiation of which second-order variables we wish to capture.

The *select clause* is a simple tuple of second-order variables contained in the body. When we write a query, we always write this clause first.

There are five types of basic queries. We use standard theorem proving notions such as entailment and satisfiability in these definitions. See §3.2.1 for definitions.

Definition 6.2.4 (Entailment query)**.** *An* entailment query *is written $S \models F$, where F is a meta-CNF (definition §6.1.35) and S is a subset of the second-order variables in F.*

Given a theory T and a ground instantiation I of all second-order variables in F, we say that I is a full ground solution *of $S \models F$ if, when substituting the values of I in F, the resulting first-order formula F_I is* entailed *by T.*

For example, consider the theory T:

$$\begin{aligned} &\forall x.p(x) \\ &\forall y.p(y) \implies q(f(y)) \end{aligned} \tag{6.4}$$

and query $Q \equiv (P) \models P(a)$. Then, $(P) \equiv (p)$ and $(P) \equiv (q\{f\})$ are ground solutions of Q given T; and $\{(p\{F\}), (q\{f\{F\}\})\}$ is a complete set of solutions of Q given T,

because any composition whose head is p and any composition whose head is $q\{f\}$, when applied to a, produces a provable formula; and every provable formula of this shape belongs to one of these two sets. F is a free second-order function variable that indicates the partial instantiation, including an infinite set of ground instantiations.

Definition 6.2.5 (Satisfiability query)**.** *A* satisfiability query *is written $S \models^* F$, where F is a meta-CNF (definition §6.1.35) and S is a subset of the second-order variables in F.*

Given a theory T and a ground instantiation I of all second-order variables in F, we say that I is a full ground solution *of $S \models^* F$ if, when substituting the values of I in F, the resulting first-order formula F_I is* satisfiable *in T.*

For example, consider the theory T:

$$\begin{aligned} &\forall x.\neg p(f(x)) \\ &\forall y.q(f(y)) \implies p(y) \end{aligned} \tag{6.5}$$

and query $Q \equiv (P) \models^* P(a)$. Then, $(P) \equiv (p)$, $(P) \equiv (q)$ and $(P) \equiv (q\{f\})$ are ground solutions of Q given T. However, $(P) \equiv (p\{f\})$ and $(P) \equiv (q\{f\{f\}\})$ are not solutions, because they are unsatisfiable in T (adding them to the theory would make it inconsistent). It is in general not easy (and we have not found a way) to find finite complete sets of solutions for satisfiability queries.

We note that in patterns we may sometimes informally write entailment and satisfiability queries where we use freeform formulas containing second-order variables instead of strictly meta-CNFs. The meaning of this is clear, as even when containing second-order variables, the translation of a freeform formula to a meta-CNF is unique and clearly defined. That is, we merely syntactically restructure the formulas for clarity when reading the patterns, without introducing any additional meaning.

Definition 6.2.6 (Unification query)**.** *A* unification query *is written $S : T_1 \simeq T_2$, where T_1 and T_2 are both second-order terms and S is a subset of the second-order variables in T_1 and T_2.*

We say that a ground instantiation I of all second-order variables in T_1 and T_2 is a full ground solution *of $S : T_1 \simeq T_2$ if, when substituting the values of I in T_1 and T_2, the resulting first-order terms T_{1I} and T_{2I} are* unifiable *(as first-order terms).*

Note that unification queries (as disunification queries below) are independent of the theory T.

For example, the query $Q \equiv (P) : P(f(x)) \simeq q(f(y))$ has, among its ground solutions, $(P) \equiv (q)$ and $(P) \equiv (q\{f\})$. However, $(P) \equiv (p)$ and $(P) \equiv (q\{g\})$ would not be solutions as that would make the two terms non-unifiable. It is not trivial to find a finite complete set of solutions for this query either, and we are unaware if one exists.

Definition 6.2.7 (Disunification query)**.** *A* disunification query *is written $S : T_1 \neq T_2$, where T_1 and T_2 are both second-order terms and S is a subset of the second-order variables in T_1 and T_2.*

We say that a ground instantiation I of all second-order variables T_1 and T_2 is a full ground solution *of $S : T_1 \neq T_2$ if, when substituting the values of I in T_1 and T_2, the resulting first-order terms T_{1I} and T_{2I} are* not unifiable *(as first-order terms).*

For example, the query $Q \equiv (P) : P(f(x)) \neq q(f(y))$ has, among its ground solutions, $(P) \equiv (p)$ and $(P) \equiv (q\{g\}))$. However, $(P) \equiv (q)$ and $(P) \equiv (q\{f\})$ would not be solutions as that would make the two terms unifiable. Similarly to before, it is possible there is no finite complete set of solutions for disunification queries like this one.

Definition 6.2.8 (Meta-predicate query)**.** *A* meta-predicate query *is written $S \models_M F$, where F is a formula with meta-predicates (definition §6.1.37) and S is a subset of the second-order variables in F.*

Given a theory T and a ground instantiation I of all second-order variables in F, we say that I is a full ground solution *of $S \models_M F$ if, when substituting the values of I in F, the resulting formula with meta-predicates (but no second-order variables) F_I appears (explicitly) in T.*

Meta-predicate queries are the simplest to solve, as they essentially consist in a simple first-order matching exercise between the query body and the formulas with meta-predicates in T.

For example, consider the theory T with the following formulas with meta-predicates:

$$\begin{array}{l} \delta(p,q) \\ \delta(r,q) \end{array} \tag{6.6}$$

and query $Q \equiv (P) \models_M \delta(P,Q)$. Then $(P,Q) \equiv (p,q)$ and $(P,Q) \equiv (r,q)$ are all the solutions of Q given T.

6.2.1.2 Composite queries

Composite queries combine two or more other queries to produce a new query whose set of solutions is related to those of the smaller queries, recursively over the basic queries. We will define the sets of solutions of composite queries similar to how we did for basic queries: we will define the set of full ground solutions, and then utilize definitions 6.2.2 and 6.2.3 to extend it to the definition of ground solutions and solutions in general.

The main type of composition query is *join queries*. We have chosen this name because of its semantic similarity to traditional relational join queries (see any introductory book to databases. For example [Lake and Crowther, 2013]). A join query takes two queries and a binding between the second-order variables in both queries, and has as solutions the union of solutions from both queries where the instantiations for bound variables match.

We can present this normally. However, similar to relational queries, it is easier to define a *renaming operation* first, that allows us to change second-order variables in queries to enforce or prevent matching.

Definition 6.2.9 (Variable renaming in queries)**.** *Let Q be a ESQ query. Let X and Y be two second-order variables. Then, define the query $Q_{Y/X}$ to be the query resulting from taking Q and replacing every instance of X for Y.*

We will use the shorthand notation $Q_{Y_1/X_1,Y_2/X_2,\ldots,Y_n/X_n} \equiv (((Q_{Y_1/X_1})_{Y_2/X_2})_{\ldots})_{Y_n/X_n}$.

Lemma 6.2.1 (Alpha-equivalence of queries)**.** *Alpha-equivalent queries have alpha-equivalent solutions. That is, consider any theory T. If a query Q has set of solutions $\mathbb{S}_T(Q)$ and second-order variable Y does not appear in Q, then for any second-order variable X:*

$$\mathbb{S}_T(Q) = \mathbb{S}_T(Q_{Y/X}) \tag{6.7}$$

(Note that the second-order variables do not explicitly appear in the set of solutions, but what happens is that the instantiations of variable X gets replaced with the instantiations of variable Y).

Proof. It's trivial. The second-order variables' connection to the set of solutions is only that they are replaced in the query. If the variable Y did not previously appear in the query, then renaming another variable to it causes no change in the semantics of the query. □

We are now in position to define and use *join queries*.

Definition 6.2.10 (Join query)**.** *Let Q_1 and Q_2 be two ESQ queries. Then, write $Q_1 \bowtie Q_2$ for the* join *of these two queries. We define the* select clause *S of $Q_1 \bowtie Q_2$ to be the union of the select clauses of Q_1 and Q_2.*

Given a theory T and a ground instantiation I of all second-order variables in Q_1 and Q_2, we say that I is a full ground solution *of $Q_1 \bowtie Q_2$ if it is a full ground solution to both Q_1 and Q_2.*

The definition of the solution set may lead to thinking that *intersection* would be a better name for this type of query. However, closer inspection reveals that *joins* are more general than intersections (they also are in relational queries). Depending on which variables are in common between Q_1 and Q_2, a join might be an intersection of solutions, a full cartesian product of solutions or somewhere inbetween.

For example, if Q_1 and Q_2 have the exact same second-order variables, then clearly the full ground solutions of $Q_1 \bowtie Q_2$ are the intersection of those of Q_1 and Q_2. On the other extreme, if Q_1 and Q_2 share no second-order variables whatsoever, then any combination of a full ground solution of Q_1 and a full ground solution of Q_2 will be a full ground solution of $Q_1 \bowtie Q_2$. When there is a proper intersection, the result is more complex. The rename operation (definition 6.2.9) allows us to fine-tune how exactly we want these sets to intersect.

For example of join query, consider the following theory T:

$$\begin{array}{c} p(a) \\ q(b) \end{array} \tag{6.8}$$

and query $Q \equiv ((P,Q) \vDash P(a) \implies Q(a)) \bowtie ((R,Q) \vDash R(b) \implies Q(b))$.

$(P,Q,R) \equiv (q,p,p)$ is a solution because $q(a) \implies p(a)$ is entailed by T (as $p(a)$ is an axiom), and so is $p(b) \implies p(b)$ (tautology).

$(P,Q,R) \equiv (q,q,p)$ is also a solution because $q(a) \implies q(a)$ is entailed (tautology) and so is $p(b) \implies q(b)$ ($q(b)$ is an axiom).

However, $(P,Q,R) \equiv (q,p,q)$ is not a solution because while $q(a) \implies p(a)$ is entailed by T, $q(b) \implies p(b)$ is not entailed (it is *satisfiable*, so potentially true in some interpretations, but not necessarily true in all of them). However, $(P,Q) \equiv (q,p)$ is a solution to the left query.

Similarly, $(P,Q,R) \equiv (p,q,p)$ is not a solution either because while $p(b) \implies q(b)$ is entailed by T, $p(a) \implies q(a)$ is not. However, $(Q,R) \equiv (q,p)$ is a solution to the right query.

Note that the full ground solutions of join queries are defined in terms of full ground solutions of the inner queries, and not based on their select clauses. This means that the join combines **all** variables present in the queries, not just those that would be output by the select clause.

We can trivially define n-ary joins (for finite n) as the sequential joins of more than 2 queries. To do this, it is important to establish that joins are **associative** and **commutative**. This is trivial from the definition.

The other type of composite query is what we call *forall queries*. These queries solutions are the intersection of the solutions of one of the queries that appear for *every possible solution* of the other query.

Let's define this formally:

Definition 6.2.11 (Forall query)**.** *Let Q_S and Q_C be two ESQ queries. Consider the set V_S of second-order variables appearing in Q_S and the set V_C of second-order variables appearing in Q_C. Then, write $S : \forall Q_C.Q_S$ for the* forall intersection *of Q_S over Q_C, where S is a subset of $V_S - V_C$.*

Given a theory T and a ground instantiation I of all second-order variables in $V_S - V_C$, we say that I is a full ground solution *of $S : \forall Q_C.Q_S$ if, for each full ground solution I_C of Q_C, there is at least one full ground solution I_S^C of Q_S such that I_S^C matches I on $V_S - V_C$ and I_S^C matches I_C on $V_S \cap V_C$.*

Essentially, forall queries are quantifying over the variables in the intersection of Q_S and Q_C, and returning the variables in Q_S that do not appear in Q_C that are present as a solution of Q_S for every solution of Q_C.

6.2.2 Computational aspects

The purpose of this section is mainly to explicitly acknowledge questions that have not been answered in this section, and to give some basic answers for some, point to the part of the thesis that provides them in others, and simply delineate the gap in others.

6.2.2.1 Satisfiability and disunification

One of the core computational difficulties of solving ESQ queries has to do with a fundamental theoretical limitation in the decidability of first-order logic itself (even without second-order variables). Indeed, we presented here *satisfiability* queries, but satisfiability of a first-order formula is known to be, in general, *co-semi-decidable* [Robinson and Voronkov, 2001]. This means that while an algorithm can be produced that will always terminate when a formula is not satisfiable (in fact, the resolution algorithm presented in §3.2.1.2 is one such algorithm), if a formula is satisfiable, for any algorithm conceivable, it may be the case that the algorithm never terminates.

Similarly, disunification formulas present a difficulty. While first-order disunification is in general decidable [Comon, 1990], it is a much more complex problem than first-order unification, and requires an embedding of term equality in first-order logic itself. Instead, in our implementation, and because of the much less relevant nature of disunification queries, we utilize a similar approach to disunification queries as we do for satisfiability queries, which is to conceive them as the *negation of an entailment / unification query*.

In particular, we note that the set of solutions of a satisfiability query is exactly the complement of that of an associated entailment query; and similarly for disunification and unification queries. We can present these as two lemmas:

Lemma 6.2.2 (Satisfiability is the opposite of entailment)**.** *Consider a theory T, and a satisfiability query $S \models^* F$, and its associated set of ground solutions $\mathbb{G}_T(S \models^* F)$.*

Consider the entailment query $S \models \neg F$. Then, a ground solution G has $G \in \mathbb{G}_T(S \models^* F)$ if and only if $G \notin \mathbb{G}_T(S \models \neg F)$.

Proof. G is a ground solution to $S \models^* F$ if and only if, when applying the instantiations of G to F, for the resulting first-order formula F_G, there is an interpretation I that satisfies all axioms in T and also satisfies F_G. But then, in the interpretation I, $\neg F_G$ is not satisfied (since interpretations are always complete). Therefore, $\neg F_G$ is not entailed by T, and therefore $G \notin \mathbb{G}_T(S \models \neg F)$.

Conversely, if $G \notin \mathbb{G}_T(S \models \neg F)$, then that means $\neg F_G$ is not entailed by T, and thus there must be an interpretation I in which $\neg F_G$ is not satisfied, and by completeness, F_G must be satisfied. Therefore, F_G is satisfiable in T, and thus $G \in \mathbb{G}_T(S \models^* F)$. □

Lemma 6.2.3 (Disunification is the opposite of unification)**.** *Consider a theory T, and a disunification query $S : T_1 \neq T_2$, and its associated set of ground solutions $\mathbb{G}_T(S : T_1 \neq T_2)$.*

Consider the unification query $S : T_1 \simeq T_2$. Then, a ground solution G has $G \in \mathbb{G}_T(S : T_1 \neq T_2)$ if and only if $G \notin \mathbb{G}_T(S : T_1 \simeq T_2)$.

Proof. It's the same idea as before but even more obvious due to the definition of disunification.

G is a ground solution to $S : T_1 \neq T_2$ if and only if, when applying the instantiations of G to T_1 and T_2, the resulting terms $(T_1)_G$ and $(T_2)_G$ are not unifiable. But this is equivalent to G not being a ground solution to $S : T_1 \simeq T_2$, by definition of unification query solutions. □

What we do instead with both of these query types is approximate them, and utilize the same underlying mechanism for entailment/unification (unification dependency graphs, described in chapter 7). Semantically, when refining a set of solutions (see §6.2.2.2) with a satisfiability or disunification query, we could produce the associated entailment/unification dependency graph, and check / attempt to combine the previous solutions with the current solutions, and output only those that **do not** appear in the associated graph. However, due to the infinite nature (discussed in more detail in §6.2.2.4) both of the solution sets and the dependency graph unification algorithm execution, this would be a non-terminating task in general. We can approximate this by limiting the depth / extent to which we execut the algorithm, and assuming if no result is found up to that point, noone will. This is exactly our approach.

6.2.2.2 Combining queries

We introduced two types of composite queries: Joins and Forall queries. Both of these combine the sets of solutions of underlying queries in different ways. However, there are two important computational aspects to consider here. First, in many cases we can solve the combined query a lot more efficiently than solving each query independently and then combining the solution sets. Second, there are nuances with how we combine infinite sets computationally speaking. The first topic we discuss here, the second is discussed in §6.2.2.4.

Forall queries do not benefit from the first aspect: in our implementation, every time we solve a forall query, we do so by solving each invidual query and then combining them (how explained in §6.2.2.4). Join queries, however, can greatly benefit from the first aspect in most cases.

In particular, consider the join of two entailment queries: $(S_1 \vDash F_1) \bowtie (S_2 \vDash F_2)$. Semantically, the set of solutions of this query (given a theory) is the set of tuples of instantiations all variables in S_1 and S_2 such that when replacing the instantiations in F_1 and F_2, both are entailed. But note that the select clause of this query is $S_1 \cup S_2$, and that F_1 and F_2 are both entailed if and only if $F_1 \wedge F_2$ is entailed by the theory. Thus, we have the following lemma:

Lemma 6.2.4 (Join of entailment queries)**.** *For any theory T, sets of variables S_1 and S_2 and formulas F_1 and F_2, the following holds:*

$$\mathbb{S}_T((S_1 \vDash F_1) \bowtie (S_2 \vDash F_2)) = \mathbb{S}_T((S_1 \cup S_2) \vDash (F_1 \wedge F_2)).$$

Proof. The select clause is by definition of join query. $F_1 \wedge F_2$ is entailed if and only if both F_1 and F_2 are entailed. □

More generally, in terms of unification dependency graphs, we can *merge* dependency graphs to generate a combined graph that has all the *constraints* of both graphs. When we join any two queries completely defined by a single dependency graph, merging the graphs is always enough. This includes entailment queries and unification queries. Even when joining queries with approximations / multiple graphs, merging graphs is still a useful tool to generate a more efficient result other than calculating the sets of solutions independently and then combining them.

6.2.2.3 Query order

In the previous section we have described how combined queries can often be merged in one way or another, but sometimes they may not. For example, the described approximation for satisfiability and disunification queries (§6.2.2.1) requires starting from a base dependency graph (product of one query) and producing a set of dependency graphs that approximate the solution in some way from it. In these circumstances, *query order matters*. This, however, is a minor concern, as it does not change things drastically and it only is required with certain composite queries, but not with all.

We do not have a standard query order depending on query types, but we do acknowledge the relevance of it. In the implementation, the query order is determined by the order the query is written. In this sense, we would lose the associativity and commutativity of certain composite queries.

6.2.2.4 Enumeration of solutions

In general, solution sets of queries are infinite. In the case of entailment queries, our algorithm (chapter 7) produces a sound and *fair* (i.e. complete) enumeration of the set, where every solution is guaranteed to be output after a finite amount of time (under certain conditions). This is good, as it allows us to produce solutions iteratively without fear that we may be completely losing important families of them. In practice, this is done through a lazy evaluation algorithm, explained in more detail in chapter 8.

In general, we can extend this approach to all queries, through the approximations described above, but also through a generic approach to lazy evaluation and enumeration-based processing of data. This is done through specific data structures designed to this effect and combination functions for them that pervasively care about this property. Chapter 8 contains more specific details about this.

6.3 Maximal CNFs and inductive instantiation

In this section we describe how we adapt the conventional resolution algorithm (see §3.2.1.2) to existential second-order query logic, by applying minimal commitment, building what we call *maximal CNFs* through *inductive instantiation*.

The theory in this section is not very extensive and it is a direct extension of standard resolution. However, the extent to which it differs from standard resolution is completely novel to this thesis.

We will not prove here[8], but will use, the fact that the usual algorithm for transforming a first-order formula into CNF can be used to transform a formula with second-order variables into a meta-CNF.

Theorem 6.3.1 (Refutation procedure for ESQ logic)**.** *Consider a theory (a finite set of formulas) T and a conjecture F, each of which possibly contains second-order variables. Consider the meta-CNF result of conjunctively joining $T \wedge (\neg F)$ and transforming it into a meta-CNF. Write N_F^T for this meta-CNF.*

For each ground unification solution U, consider the first-order formula $U(F)$ result of applying U to F, and correspondingly $U(T)$ and $U(N_F^T)$.

Then, $U(T) \models U(F)$ if and only if $U(N_F^T)$ is unsatisfiable.

That is, the set of unification solutions that make F entailed by T are the same that make N_F^T unsatisfiable.

There is a standard (and simple) proof that a refutation procedure is correct for first-order formulas. We will not repeat this here. See [Robinson and Voronkov, 2001, Bundy, 1983].

For each ground unification solution U, $U(F)$ and $U(N_F^T)$ are both first-order formulas and thus the standard proof applies. The rest of the theorem is just a restatement of this fact.

Definition 6.3.1 (Inductive instantiation of a meta-CNF)**.** *Consider a meta-CNF N that contains one second-order predicate variable P with arity n. Consider the second-order signature that we are using S (which contains the second-order variables as well). Moreover, consider a partial instantiation I of the second-order variables in S. Assume, without loss of generality, that*

$$N \equiv ((\delta_P^1 P(\varepsilon_1^1, \varepsilon_2^1, ..., \varepsilon_n^1) \vee A) \wedge B)$$

where δ_P^1 is a negation or empty, each ε_i^1 is a unifier expression, A is a disjunction of meta-literals and B is a conjunction of meta-clauses. P may appear in more than one such meta-literal, in which case we will consider $\delta_P^2, \delta_P^3, ...$ and $\varepsilon_i^2, \varepsilon_i^3, ...$

Then, define the inductive instantiation *of N over P to be the set of the following meta-CNFs, each with its own updated signature and instantiation:*

[8] It is conceptually trivial, but tedious

- **Negated instantiation**:
  - *Add a fresh second-order variable $P^{\neg}$ with arity n to the signature S.*
  - *Add $P := \neg P^{\neg}$ to the partial instantiation I.*
  - *For each j, if δ_P^j is empty, replace it with a negation; if δ_P^j is a negation, replace it with empty.*
  - *Replace each appearance of P in N with $P^{\neg}$.*
- **Disjunctive instantiation**:
  - *Add two fresh second-order variables P_1 and P_2 with arity n to the signature S.*
  - *Add $P := P_1 \vee P_2$ to the partital instantiation I.*
  - *For each j, if δ_P^j is empty, replace $P(\varepsilon_1,...,\varepsilon_n)$ with $P_1(\varepsilon_1,...,\varepsilon_n) \vee P_2(\varepsilon_1,...,\varepsilon_n)$.*
  - *For each j, if δ_P^j is a negation, replace $(\neg P(\varepsilon_1,...,\varepsilon_n) \vee A)$ with $(\neg P_1(\varepsilon_1,...,\varepsilon_n) \vee A) \wedge (\neg P_2(\varepsilon_1,...,\varepsilon_n) \vee A)$*
- **Universal instantiation**:
  - *Add a fresh second-order variable Q with arity $n+1$ to the signature S.*
  - *Add $P := \lambda\varepsilon_1...\varepsilon_n.\forall x.Q(\varepsilon_1,...,\varepsilon_n,x)$ to the partial instantiation I.*
  - *For each j, if δ_P^j is empty:*
    - *Add a fresh first-order variable x to the signature S (use the same variable for every j that is positive).*
    - *Replace $P(\varepsilon_1,...,\varepsilon_n)$ with $Q(\varepsilon_1,...,\varepsilon_n,x)$.*
  - *For each j, if δ_P^j is a negation:*
    - *Add a new function symbol f[9] with arity n to the signature S (use the same function for every j that is negative).*
    - *Replace $\neg P(\varepsilon_1,...,\varepsilon_n)$ with $\neg Q(\varepsilon_1,...,\varepsilon_n,f(\varepsilon_1,...,\varepsilon_n))$.*

[9] A Skolem function.

Let's show an example. Consider the following meta-CNF:

$$(P(x,y) \vee q(x)) \wedge (R(z) \vee \neg P(f(z),z))$$

Then, the inductive instantiation over P consists in the following formulas:

- **Negated instantiation** - $(\neg Q(x,y) \vee q(x)) \wedge (R(z) \vee Q(f(z),z))$
- **Disjunctive instantiation** - $(P_1(x,y) \vee P_2(x,y) \vee q(x)) \wedge (R(z) \vee \neg P_1(f(z),z)) \wedge (R(z) \vee \neg P_2(f(z),z))$
- **Universal instantiation** - $(Q(x,y,w) \vee q(x)) \wedge (R(z) \vee \neg Q(f(z),z,g(f(z),z)))$, where w is a fresh first-order variable and g is a new function symbol (Skolem function).

Theorem 6.3.2 (Soundness and completeness of the inductive instantiation)**.** *Let N be a meta-CNF that contains a second-order predicate P. Let S be the second-order signature we are using, and I a partial instantiation of the second-order variables in S.*

Then, for any ground unification solution U, U makes N unsatisfiable if and only if U makes one of the meta-CNFs in the inductive instantiation unsatisfiable or $U(P)$ is a predicate (i.e. does not contain logical connectives)[10].

Proof. The basis of the proof is that every logical formula can be described as a combination of negation, disjunction and universal quantification of predicates. This is a well known standard fact that we will not prove again here.

We can use this to prove the theorem by noticing that for any solution U, the instantiation of P is a logical formula function of its arguments, and thus the theorem mentioned above means that every instantiation's logical connectives are a combination of the inductive instantiation steps.

The only thing left is the case when the unification solution instantiates P to a predicate, but this is specifically covered by the theorem. □

The usefulness of the inductive instantiation and the theorem is that we can use inductive instantiation to forget about the logical connectives in the instantiation and, at each step, apply the resolution refutation procedure assuming that the instantiation is to a predicate, and thus, **the meta-CNF will be instantiated to an actual CNF**.

We can write this down as an algorithm.

[10] It may, however, contain compositions, so it shall not be necessarily exactly a predicate symbol.

Algorithm 6.3.2 (Inductive instantiation resolution)**.** Consider a meta-CNF N, a signature S and a partial instantiation I. Assume we have a procedure *resolve* that takes a meta-CNF and finds all the instantiations of the second-order variables that appear in the meta-CNF in which the second-order variables are instantiated to predicates.

Then, produce the following algorithm to find all instantiations of N in the signature S with the initial partial instantiation I:

1. Apply *resolve* to N to find predicate instantiations and output them.
2. If N has no second-order predicate variables, finish the algorithm.
3. Pick a second-order predicate variable P in N. Consider the formulas in the inductive instantiation, and for each of them, non-deterministically run the algorithm from step 1.

We note that this algorithm is in general non-terminating and produces an infinite output. But it is a fair enumeration procedure as described in §8.5.

6.4 Summary

In this chapter we describe the two high-level pieces of our technical approach to automatically detecting faults in ontologies. On one hand, the formalism that allows us to express ESQ queries and their semantics as patterns that allow us to find instantiations of second-order variables that relate to entailment in a logical theory. On the other hand, the notion of *inductive instantiation* of a meta-CNF formula, that is needed for the complete exploration of the set of instantiations of predicate second-order variables. We show basic results about the soundness and completeness of inductive instantiation.

Chapter 7

Dependency graph unification for ESQ logic: Theoretical results

7.1 Basic pieces

In this section we extend some of the basic concepts introduced in the previous chapter with some additional concepts exclusive to the internal workings of the unification dependency graph algorithm. Everything appearing in this chapter is fundamentally novel to this work, despite building up on all the concepts introduced in chapters 3 and 6.

7.1.1 Dependants

Definition 7.1.1 (First-order dependant)**.** *A first-order* dependant *is either:*

- *A first-order variable.*
- *A unifier variable (definition 6.1.18) applied to another dependant.*

Definition 7.1.2 (Second-order dependant)**.** *A second-order* dependant *is either:*

- *A function symbol.*
- *A projection (see definition 6.1.2).*
- *A second-order variable.*

First-order dependants are, therefore, a subset of unifier expressions; while second-order dependants are a subset of second-order terms. This will be more relevant later

when we define graphs and the translation of expressions into graphs. The reader may wonder why we call these *dependants* (what do they depend on?). They will be the nodes of dependency graphs and the edges in said graphs will express their dependencies. Therefore, dependants depend on other dependants, and the dependency graph expresses how they do so. First-order dependants are function free, and functional dependencies are expressed through edges.

7.1.2 Equational reasoning

The following lemma is useful to reason more easily about non-deterministic rules in dependency graphs later on. This is basic set theory but we want to make it explicit because the handling of solutions, graphs and equations can get tricky and easily confusing. This allows us to have better grounding on the meaning of non-deterministic arguments.

Lemma 7.1.1 (Non-deterministic equational reasoning)**.** *Consider a system of unification equations (definition 6.1.26) $\mathcal{E}$, and consider its set of solutions, $\mathcal{U}(\mathcal{E})$. If we can show that there exist a (possibly infinite) set of systems of unification equations $\{\mathcal{E}_i\}$ such that for every unification solution $U \in \mathcal{U}(\mathcal{E})$ there is at least one $\mathcal{E}_i$ such that $U \in \mathcal{U}(\mathcal{E}_i)$; and also that for every i and for every $U_i \in \mathcal{U}(\mathcal{E}_i)$, $U_i \in \mathcal{U}(\mathcal{E})$, then it is true that $\mathcal{E}$ has the same set of solutions as the union of the sets of solutions of the $\mathcal{E}_i$. Formally,*

$$\mathcal{U}(\mathcal{E}) = \bigcup_i \mathcal{U}(\mathcal{E}_i)$$

This is basic set theory and logic, so we will not spell out the proof.

7.2 Unification dependency graphs

All concepts introduced from this section onwards are novel to this work. Of course there is inspiration and building on existing concepts, but they are not a variation of a previously existing approach (to the best of the author's knowledge). However, some particular approaches to these definitions so heavily reminisce pre-existing concepts (even if not being exactly the same), that we use similar terminology. We will explicitly note these.

It might be worth noting that, technically, unification dependency graphs are a particular case of *term graph rewrite systems* (see §2.3.3, §3.4.1, [Plump, 2002, Habel and Plump, 1995, Plump, 1999, Barendregt et al., 1987, Dwork et al., 1984, Plump, 2005]). However, the author of this thesis was not aware of this relation until after the PhD viva. Clearly, the notion of using graphs to represent unification problems is very natural. Work such as the one cited above provides general abstract results on theoretical aspects of term graph rewriting in general. Most of these requires grounding in an associated string rewriting system, which we do not have in our case. It is arguable, however, that embracing the general abstract results could have simplified some of the most difficult theorems and lemmas in this chapter. We discuss this further in chapter 10. But the reality is that the work produced here was not directly based in this and merely related in the sense of being a natural idea in the context of unification. Nonetheless, the theory in the literature would probably help understand and extend the work produced here. We did not have time to do this in this thesis, but some thoughts are shared in chapter 10.

A unification dependency graph is, for all intents and purposes, a second-order unifier. It is a representation of a set of instantiations and associated substitutions. It differs from conventional unifiers in that a dependency graph represents several independent unification steps (unifiers) at once, instead of just one. We use unification dependency graphs as an implicit and compact representation of the set of all ground instantiations and associated substitutions that make a given set of unification steps work; in the same way that a conventional first-order unifier is an implicit and compact representation of the set of all ground substitutions of first-order variables that makes the unified terms equal.

An important detail is that unifier variables, which are meant to be replaced by first-order substitutions, do not affect the instantiation of second-order variables. This is a consequence of the way in which we use second-order variables. We consider instantiation of second-order variables to happen before considering the replacement of unifier variables for substitutions. For each instantiation of second-order variables, there is an associated set of substitutions for the unifier variables, that ultimately are applied only to the first-order variables. This is one of the fundamental differences between our problem and general second-order unification.

For example, consider the following system of unification equations:

$$\sigma_1 f(X) \approx \sigma_1 F(X) \tag{7.1}$$

$$\sigma_2 \sigma_1 F(Y) \approx \sigma_2 F(X) \tag{7.2}$$

Every instantiation for F that would make equation 7.1 hold would have $F \approx f$ (and this is independent of σ_1, by definition). Then, σ_1 can be any substitution (for example $\sigma_1 X \approx g()$, but also $\sigma_1 X \approx f(g())$ or $\sigma_1 X \approx X$, etc.). This makes it so that equation 7.2 is equivalent, in this case, to $\sigma_2 \sigma_1 f(Y) \approx \sigma_2 f(X)$ (because $F \approx f$ independently of σ_1 and σ_2). This equation then tells us that σ_2 will bind Y to $\sigma_1 X$, whatever $\sigma_1 X$ is. This is part of the reason why unifier variables have an ordering: later unifications may not conceptually condition the result of earlier unifications.

In the following, we describe a dependency graph informally. A formal definition requires a lot of layers and is provided later (definition 7.2.1).

For example, a dependency graph for the given set of equations (see figure 7.1) has four labelled first-order nodes: $\sigma_1 X$, $\sigma_1 Y$, $\sigma_2 \sigma_1 Y$, $\sigma_2 X$. It has a *vertical edge* from $\sigma_1 Y$ to $\sigma_2 \sigma_1 Y$, indicating that the latter contains the former as an expression (a simple type of dependency that we call *vertical* because it relates different *unifier variables* (σ_1 and σ_2)). It has two second-order labelled nodes: f and F.

It then, more importantly, has an *anonymous* first-order node (no dependants), with two incoming *horizontal edges*. Horizontal edges can be first-order or second-order and, by definition, have a *head* (which is always a second-order node, representing the function that relates the sources with the target), a sequence of *sources* (first- or second-order nodes, respectively for first- and second-order edges), and a single *target* (a first- or second-order node, respectively). A first-order horizontal edge represents a *functional dependency* between unifier expressions via function *application*, such as $\sigma_1 Y \approx f(\sigma_1 X)$. A second-order edge represents a *functional dependency* between second-order terms via function *composition*, such as $F \approx f\{g\}$. Note that we do not explicitly indicate the ordering of sources in horizontal edges, but usually these are presented from left to right in the diagrams. The diagrams are only a simplified graphical representation of the more abstract mathematical concept of the dependency graph, in which the sources are ordered.

In our particular example, the anonymous first-order node represents the expression $f(\sigma_1 X)$ (but it's anonymous because we only use *dependants* (i.e. function free) as first-order labels in our graph), and has an incoming horizontal edge with single source

$\sigma_1 X$ and head f, representing the functional dependency of this unifier expression on the dependant $\sigma_1 X$. More importantly, it has a second incoming horizontal edge, with single source $\sigma_1 X$ and head F, standing for the right side of the equation. These two edges have the same node as target precisely because the first equation tells us that the two terms $f(\sigma_1 X)$ and $F(\sigma_1 X)$ must be equal, once instantiations and substitutions have applied. Similarly, the second equation is represented by an anonymous node with two incoming edges, one with head F from source $\sigma_2\sigma_1 Y$ and the other with head F from source $\sigma_2 X$.

Note that the naming of horizontal and vertical edges is related to the notion of unifier variables representing different *levels*: Horizontal edges are edges that connect nodes within the same unifier level, whereas vertical edges are edges that connect nodes on different unifier levels. Unfortunately, for better visualization in our diagrams, horizontal edges will often be represented vertically and vertical edges horizontally. We will use the convention that vertical edges are dotted, curved lines and horizontal edges are solid, straight lines. Moreover, vertical edges only have a single source and no head, whereas horizontal edges often have several sources and always have a head. This should be enough to distinguish them in diagrams.

Let us formalize this. First, we specify the syntax of what a dependency graph contains, and later we relate this to the semantics of the solutions it represents.

Definition 7.2.1 (Unification dependency graph)**.** *A* unification dependency graph *consists of:*

- *A set of* second-order nodes*. Each second-order node may contain any number (including zero) of second-order dependants.*
- *A set of* first-order nodes*. Each first-order node may contain any number (including zero) of first-order dependants, all of which must have the same unifier level (this applies to no unifier level ($\bot$) too).*
- Second-order horizontal edges*, each of which consists of a* head *(a second-order node), a sequence (possibly empty) of* sources *(second-order nodes) and a single* target *(a second-order node). Each horizontal edge may be marked as* redundant*.*

  While redundancy is formally just a toggle on an edge that we manipulate directly, conceptually a redundant *horizontal edge is used to indicate that there are other elements in the graph that represent the same information as this edge in ways*

Figure 7.1: A dependency graph.

The graph contains 6 first-order nodes (dark blue), 4 of which containing one dependant each, and 2 of them containing no dependants (represented by a dot).

It also contains 2 second-order nodes (red), though one of them (containing F) is drawn three times to simplify edges. It is formally the same node.

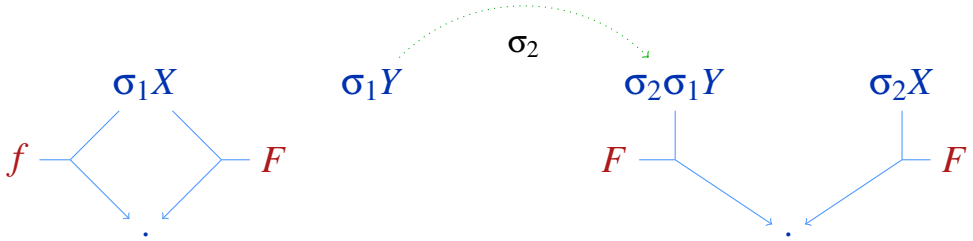
It contains 4 first-order horizontal edges (light blue, angled arrows), each of which has 1 source (first-order node), its head (second-order node) and its target (first-order node).

Second-order horizontal edges would have second-order nodes as sources and target.

Horizontal edges can have any number of sources, not necessarily 1, but they always have a single head and a single target.

The graph contains 1 vertical edge (green). Vertical edges always have a single first-order node as source and a single first-order node as target.

Colors are only used in this example graph, they will not be used in following graphs.



that we prefer. We keep these edges in the graph because we may wish to check whether this information already exists in the graph, and for that purpose the edges are still useful.

When representing graphs with diagrams in this text, redundant edges will never be graphically represented. Horizontal edges are never actually removed from the graph through any of the rewrite rules (§7.4), and marking as redundant will graphically be represented as removing the edge.

- First-order horizontal edges*, each of which consists of a* head *(a second-order node), a sequence (possibly empty) of* sources *(first-order nodes) and a single* target *(a first-order node). Each horizontal edge may be marked as* redundant.

- Vertical edges*, each of which has a single source, a single target, both first-order nodes, and an associated unifier variable* σ_i*. We sometimes interchangeably refer to the* unifier level *i of a vertical edge as an equivalent of its unifier variable* σ_i*. See definition 6.1.19 if you want to make sure that this makes sense.*

An essential restriction to the above on dependency graphs is that there must never be different nodes containing the same first-order or second-order dependant. Another way to think about this restriction is that if two nodes contained the same dependant, then they must be considered the same node and treated as a single, merged entity in the dependency graph.

We also note that horizontal edges are considered indistinguishable if they have the same sources, head and target. While in some rules we will indicate we want to remove all but one of the edges which have the same sources, head and target, this is, from a theoretical point of view, unnecessary and meaningless, since each edge (each set of sources, head and target being related) either is or is not in the graph. Multiple edges connecting exactly the same nodes have absolutely no effect on the graph.

This is different from explicitly redundant *horizontal edges. These are edges that are not repeated but for which we have verified that their exclusion from the graph would not change the graph's semantics. We keep them for easy verification of certain aspects, but exclude them from other rules that would involve loops if applied to redundant edges.*

We will now define the semantics of dependency graphs. For this, let us first produce a useful tool.

Definition 7.2.2. *We extend unification equations to allow special symbols called first and second-order* proxies. *We will usually write κ_N and χ_M to represent a first-order proxy associated with N and a second-order proxy associated with M, respectively.*

Given a system of unification equations with proxies, for each unification solution U, U is defined to be a solution to *the system if there is a way to choose a first-order term for each first-order proxy and a second-order term for each second-order proxy, such that U is a solution of the unification system result of replacing each proxy for its corresponding term (U is a solution as long as there is such a choice, whichever the choice may need to be).*

That is, proxies have no meaning other than being a way to link different equations, stating that, whatever the term is that we replace for the proxy (which will be different for each specific solution), it will satisfy the equations if replaced the same way everywhere. The usual way in which we will use proxies when dealing with dependency graphs is to use them to refer to nodes in the graph and produce transitive equivalence between any expressions associated with that node, in a succinct way.

For example, we may express the equation system:

$$\sigma_1 X \approx \sigma_1 Y$$
$$\sigma_1 Y \approx \sigma_1 Z$$

as

$$\sigma_1 X \approx \kappa_N$$
$$\sigma_1 Y \approx \kappa_N$$
$$\sigma_1 Z \approx \kappa_N$$

Note that, by definition of the unification solutions, for any unification equation system with proxies, there is a unification equation system without proxies with the same unification solutions.

Definition 7.2.3 (Dependency graph equations)**.** *A unification dependency graph $\mathcal{G}$ has an associated unification equation system $\mathcal{E}(\mathcal{G})$. The way to build the set of equations associated with a dependency graph is as follows.*

For each first- and second-order node in the graph, we produce a number of unifier expressions or second-order terms, respectively, in the following way. The resulting unification system has equations to establish that all expressions associated with the

same node are equivalent after unification. These equations are produced by using proxies.

- *For each first-order node N in the graph, produce the following equations:*
  - *For each dependant* ε *contained in the node, produce an equation* $\kappa_N \approx \varepsilon$.
  - *For each non-redundant incoming horizontal edge to the node, with head node H and source nodes* $S_1, ..., S_n$*, produce an equation* $\kappa_N \approx \chi_H(\kappa_{S_1}, ..., \kappa_{S_n})$
- *For each second-order node M in the graph, produce the following equations:*
  - *For each dependant* ϕ *contained in the node, produce an equation* $\chi_M \approx \phi$.
  - *For each non-redundant incoming horizontal edge to the node, with head node H and source nodes* $S_1, ..., S_n$*, produce an equation* $\chi_M \approx \chi_H\{\chi_{S_1}, ..., \chi_{S_n}\}$.
- *For each vertical edge V in the graph, with source node S, target node T and unifier variable* σ_i*, produce an equation* $\kappa_T \approx \sigma_i \kappa_S$.

For example, the dependency graph in figure 7.2 has the following associated unification equation system:

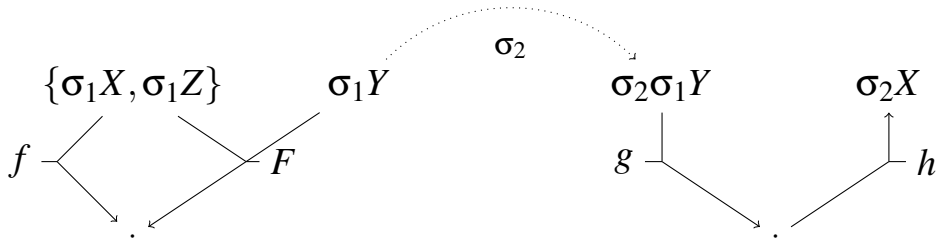
$$\begin{array}{lllll} \chi_1 \approx f & \chi_2 \approx F & \chi_3 \approx g & \chi_4 \approx h & \kappa_1 \approx \sigma_1 X \\ \kappa_1 \approx \sigma_1 Z & \kappa_2 \approx \sigma_1 Y & \kappa_3 \approx \chi_1(\kappa_1) & \kappa_3 \approx \chi_2(\kappa_1, \kappa_2) & \kappa_4 \approx \sigma_2\sigma_1 Y \\ \kappa_5 \approx \sigma_2 X & \kappa_6 \approx \chi_3(\kappa_4) & \kappa_6 \approx \chi_4(\kappa_5) & \kappa_4 \approx \sigma_2\kappa_2 & \end{array}$$

Reading equations produced this way can be hard due to the sheer number of proxies, however. We can simplify this system, by considering the transitivity of $\approx$ to produce the following equivalent system (same unification solutions):

$$\sigma_1 X \approx \sigma_1 Z \quad f(\sigma_1 X) \approx F(\sigma_1 X, \sigma_1 Y) \quad g(\sigma_2\sigma_1 Y) \approx h(\sigma_2 X)$$

Equations with proxies are useful in a formal way because they systematize proofs and algorithms, but humans generally find it more convenient to process fewer equations. We defined and use proxies in the definitions and proofs, but whenever possible we will avoid them in examples meant to be read by humans.

Figure 7.2: A dependency graph.
Note that for simplicity and reducing clutter, we use $\sigma_1 Y$ instead of $\{\sigma_1 Y\}$ to refer to a node that contains a singleton set of dependants (just one dependant). In general, we will omit the curly braces for singleton sets of dependants where it is obvious that the label is a singleton set of dependants instead of a node label.



Definition 7.2.4 (Dependency graph solutions)**.** *A unification solution U is a* solution to *a dependency graph $\mathcal{G}$ if U is a solution to $\mathcal{E}(\mathcal{G})$.*

We similarly write $\mathcal{U}(\mathcal{G})$ as shorthand notation for $\mathcal{U}(\mathcal{E}(\mathcal{G}))$.

We will write $\varepsilon_1 \approx_{\mathcal{G}} \varepsilon_2$ to mean that for every $U \in \mathcal{U}(\mathcal{G})$, $\varepsilon_1 \approx_U \varepsilon_2$.

We should make a note about vertical edges. The equations produced by vertical edges are fundamentally redundant. If the graph is produced properly[1], the equations produced by vertical edges could be removed and it would produce the same set of solutions. In that sense, we could have perhaps omitted vertical edges in this theoretical description of dependency graphs. Vertical edges are used as an implementation mechanism with two objectives. First, they are used in certain rewrite rules (§7.4) for dependency graphs to check when there is the kind of dependency that vertical edges represent in an easy way. Instead of checking whether nodes containing certain dependants are present in the graph, we can just check whether any vertical edges are present, being able to describe all rules in a more local and topological way. Second, this local and topological identification of relevant nodes in a graph allows for the implementation to be more efficient, since the search during the execution of the dependency graph solving algorithm is performed using this local mechanism. We include them here because they are relevant for the correctness of the **algorithm**, since it uses these edges as an implementation tool. But an algorithm could be produced

[1] What this means is not the point of this discussion, which is an informal discussion and only aimed at helping the reader understand.

without anything resembling vertical edges that would work. It would just have a more complicated description and less efficient performance.

In order for this to work, we need to ensure that all the relevant vertical edges in the graph are present. We do this only once per relevant node, and use the vertical edges as a persistent witness of this search process that we can reuse over and over. We call this process *vertical alignment*. It is a simple operation, that involves ensuring that for each dependant of the form $\sigma_i\alpha$ in a node in the graph, the dependant α is also in the graph, and there is a vertical edge between them. We may, however, omit some of these nodes in the diagrams we use to represent dependency graphs in this text, for clarity. When we do so, it is because they have no other edges or are particularly relevant. In a formal dependency graph during a dependency graph solving algorithm the way we present it in this thesis, however, it is necessary to keep these nodes in the graph.

The following produces, from a unification equation system $\mathcal{E}$, a dependency graph $\mathcal{G}$ such that $\mathcal{U}(\mathcal{E}) = \mathcal{U}(\mathcal{G})$:

Algorithm 7.2.5 (Dependency graph from equations)**.**

- When we say that we *grab* the node associated with a dependant from a graph, we mean that, if a node exists containing that dependant, we use that node. If no node exists containing that dependant, we create a new node containing it and use it. We also must ensure the vertical alignment of the newly created dependant, by adequately grabbing dependants the node vertically depends on and producing vertical edges between them.
- When we say that we *merge* two nodes, we mean that we transform the dependency graph to consider those two nodes equivalent for all intents and purposes. In practice, this means merging the sets of dependants they contain into a single node, and combining all edges related to those nodes so that they all relate to the resulting single node (both incoming, outgoing and those that the node is a head for).
- Given a unifier expression or a second-order term, we can *grab* a node associated with it in the following way:
  - If the expression is a dependant, then grab the node associated with that dependant.

  - If it is an expression containing function symbols, then first normalize it so that it is a second-order term applied to / composed with a set of sub-expressions. Then, create an anonymous node (no dependants), recursively grab the nodes associated with each sub-expression and with the head, and create an edge with the sub-expression nodes as sources, the second-order term as head and the anonymous node as target. The anonymous node is the resulting node.

- Given an equation, we introduce it into a dependency graph by grabbing the nodes associated with the expressions on each side of the equation and merging them.

- Introduce all equations in the system sequentially (order is irrelevant) to produce a dependency graph with the same set of solutions as the unification equation system.

Theorem 7.2.1. *Algorithm 7.2.5 correctly produces a dependency graph $\mathcal{G}$ from a unification equation system $\mathcal{E}$ such that $\mathcal{U}(\mathcal{G}) = \mathcal{U}(\mathcal{E})$.*

Proof. First, we note that the fact that dependants only appear once in the graph and that only dependants appear in the graph, combined with vertical alignment, guarantees that all relations between unifier expressions that differ only in application of unifier variables are preserved by connections to the same node in the dependency graph. For example, the relation between $\sigma_2\sigma_1 X$ and $\sigma_1 X$ is represented between the adequate nodes because of exhaustive application of vertical alignment.

Second, we note that, by definition of the *grab* process and of the unification equation system associated to a graph (definition 7.2.3), it is always the case that, for any given expression ε, the node $N(\varepsilon)$ that we obtain by grabbing it in the graph is such that $\kappa_{N(\varepsilon)} \approx \varepsilon$ is an equation in the unification system associated to the graph (or equivalently for second-order expressions).

Finally, merging nodes N and M in the graph has the consequence, when producing the unification system associated with the graph, of producing equations transitively equivalent to $\kappa_N \approx \kappa_M$ (or equivalently for second-order expressions). Transitivity of equations ensures that this is semantically the only relevant consequence that it has (i.e. the graph may produce more, less or different equations than the original system, but they will only differ in terms of transitivity of equivalence, which holds in the

semantics of substitution/instantiation solutions).

Therefore, each equation of the form $\varepsilon_1 \approx \varepsilon_2$ is translated into elements in the graph ultimately producing equations equivalent to $\varepsilon_1 \approx \kappa_N$ and $\varepsilon_2 \approx \kappa_N$ (or equivalently for second-order expressions). Thus, the unification system associated with the produced graph has as set of solutions exactly those that equate expressions that were equated by the original unification equation system, and so their sets of solutions are equal.

□

7.3 Normalization of dependency graphs

In this section we will define six normalization levels for dependency graphs (most of which are strictly more constrained than the previous ones). Their purpose is to strike a balance between the complexity / non-deterministic branching factor of transforming a dependency graph into such normal forms, and the simplicity with which we can relate the graph with its set of solutions. Each specific normalization level fills a particular purpose in this spectrum.

However, *acyclicity* should be regarded as being at a different level than the remaining normalization levels. Transforming a cyclic graph into a set of acyclic graphs may incur non-determinism (see §7.4.4), but transforming an acyclic graph into an equivalent factorizable or seminormal graph never incurs non-determinism *unless a cyclic graph is produced along the way*. These situations are also considerably rare in practice. Therefore, acyclicity is regarded as a formal condition that is required for theoretical results but which is rare and algorithmically costly to have to enforce. Not surprisingly, cycles are related to the usual *occurs check* in unification theory, and thus why this is the name chosen for the rule that breaks cycles in dependency graphs.

Clearly, the notion of *normalization* or its utility to describe subsequent reductions in the form of a formal representation, and the properties they have to allow us to perform certain operations with them, is not novel to this work. However, as unification dependency graphs themselves in the way presented here are completely novel to this work, so are the particularities of their normalization levels as described in this section.

7.3.1 Prenormal form

Prenormalization is the least restrictive normalization level. Prenormalization and factorizability (§7.3.3) are closely related but need to be divided due to acyclicity. Where factorizability describes the situation in which the graph has been simplified as much as possible without *factorizing*[2] any edges; prenormalization describes all conditions that must be enforced to allow acyclicity to be reachable.

In other words, we normally take a dependency graph, apply rules until it is prenormal, then apply the occurs check[3], which deals with cycles, then apply prenormalizing rules again, and keep doing this until no cycles remain. At this point, we can finally focus on trying to reach factorizability by applying the remaining prefactorizing rules that require the graph to be acyclic.

Definition 7.3.1 (Prenormal dependency graph)**.** *A dependency graph is* prenormal *if the following all hold:*

1. *There is no node in the graph with two outgoing vertical edges with the same unifier variable.*
2. *For any node in the graph containing a dependant of the form $\sigma_i\varepsilon$, there is an incoming vertical edge whose source contains dependant ε.*
3. *For each vertical edge in the graph with source S, target T and unifier variable σ_i, and each dependant ε in S, dependant $\sigma_i\varepsilon$ is in T.*
4. *For each vertical edge V in the graph with source S and target T, and each horizontal edge H whose source or target is S, the edge $H[V]$[4] is in the graph.*
5. *There are no two non-redundant*[5] *horizontal edges in the graph with the same source sequence and the same head.*
6. *The head of every non-redundant horizontal edge in the graph contains only second-order variables or function symbols.*

[2] *Factorization* is formally defined later, but the definition of factorizability or prenormalization does not depend on it, although it is related to it.

[3] §7.4.4

[4] Definition 7.4.4. The reason we define this later is because of its relation to the *vertical monotony* reduction rule, defined later. These definitions are, though, clearly not circular. Informally, $H[V]$ is another horizontal edge representing the propagation of the horizontal edge H to higher unifier levels through the vertical edge V.

[5] We will remind the reader here that *redundancy* is a formal tag of horizontal edges explicitly represented in the implementation of the algorithm.

We will show examples once we have introduced *factorizability*, due to the strong connection between these two.

7.3.2 Acyclic form

As the name suggests, *acyclicity* relates to the presence of *directed cycles* in the dependency graph. By the definition of vertical edges, cycles can only appear on horizontal edges, and we consider cycles both from sources to target and from head to target. Cycles express circular dependencies which usually do not have solutions in finite terms. However, in §7.4.4 we explain in more detail how in certain situations cycles may be only apparent and could be broken by adequate instantiations of second-order variables. Cycles also prevent several of the manipulations that we do with graphs from properly terminating and may produce infinite loops if not dealt with.

Definition 7.3.2 (Acyclic dependency graph)**.** *A dependency graph is* cyclic *if there is a set of nodes (either all first-order nodes or all second-order nodes)* $\{N_i\}$*, horizontal edges* $\{E_i\}$ *and indices* j_i *(we call them* source indices*); for* $1 \leq i \leq n$ *in the graph such that:*

- *For each* $i < n$*:*
  - *If* $j_i = 0$*, then* N_i *is the head of* E_i*.*
  - *If* $j_i \neq 0$*, then* N_i *is the* j_i*-th source of* E_i*.*
  - N_{i+1} *is the target of* E_i*.*
- *The same applies for* $i = n$*, except instead of* N_{i+1}*,* N_1 *is the target of* E_n*.*

A graph is acyclic *if it is not cyclic.*

We recommend that readers make sure that they are convinced that this definition is equivalent to the usual notion of the graph having no directed cycles, in any way or shape.

More details on how we break cycles or invalidate graphs with cycles that cannot be broken, and the meaning of all of this in the semantics of solutions can be found on §7.4.4.

7.3.3 Factorizable form

Factorizability describes the situation in which the graph has been simplified as much as possible without *factorizing* any edges, and furthermore some situations that represent the graph has no solutions have been validated.

Definition 7.3.3. *In an acyclic graph, define the* recursive arity *of a second-order node in the following way:*

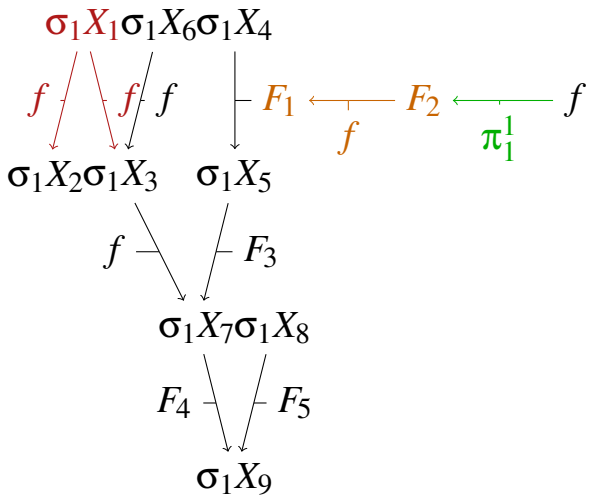
- *If the node has no incoming horizontal edges and contains one or more second-order dependants all of which have the same arity, then it corresponds to the arity of said dependants.*
- *If the node has no incoming horizontal edges and contains no second-order dependants, then it is defined as zero*[6].
- *If the node has no incoming horizontal edges and contains second-order dependants with different arities, then it is undefined.*
- *If the node has incoming horizontal edges and the recursive arities of all sources of all incoming horizontal edges are the same, and it is also the same as the arity of all dependants in the node, then it corresponds to that arity.*
- *If the node has incoming horizontal edges but there are sources of incoming horizontal edges with different recursive arities, or it is different from the arities of second-order dependants in the node, then it is undefined.*

Definition 7.3.4 (Factorizable dependency graph)**.** *A dependency graph is* factorizable *if the following all hold:*

1. *It is prenormal.*
2. *It is acyclic.*
3. *The head of every non-redundant horizontal edge in the graph has no incoming horizontal edges.*
4. *For each second-order node, all non-variable dependants it contains are equivalent.*

[6]This is an extreme case that should not appear in real cases and for which we do not really care of the arity. However, we consider undefined recursive arities incorrect, and this case is not technically incorrect, so we just define it as zero in this case.

Figure 7.3: An unnormalized graph $\mathcal{G}_0$.
Note that we repeat heads for clarity, but heads with the same dependants are formally the same node.
The graph is not prenormal nor factorizable because there are edges with same sources and head (red), an edge with a projection as head (green), and an edge head with an incoming horizontal edge (orange).



5. *For each second-order node, its recursive arity (definition 7.3.3) is well defined.*

6. *For each non-redundant horizontal edge, the recursive arity of its head is equal to the number of sources of the edge.*

For example, consider the graph $\mathcal{G}_0$ in figure 7.3, assuming a signature where the only function symbol is f. It is acyclic but it is not prenormal nor factorizable, because it fails conditions 5 and 6 of definition 7.3.1, and condition 3 of definition 7.3.4. However, graph $\mathcal{G}_1$ in figure 7.4 is factorizable and has the same solutions as $\mathcal{G}_0$.

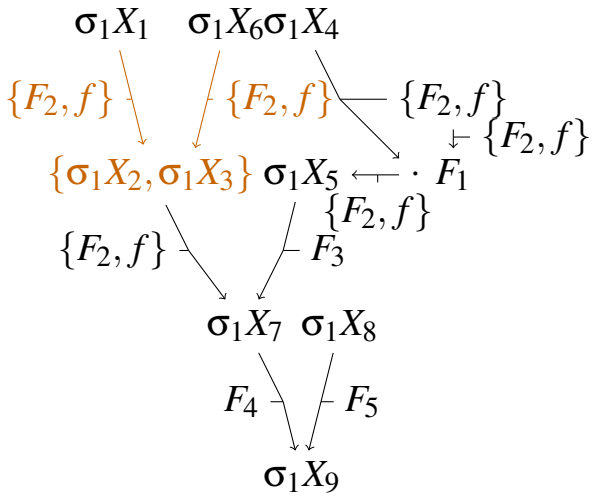
7.3.4 Seminormal form

Seminormalization is the most simplified normalization level that can be achieved for every graph (so long as the graph never becomes cyclic) without incurring any non-determinism[7]. Conceptually, a seminormal graph is a graph that is both factorizable and cannot be zero factorized[8].

Definition 7.3.5. *A dependency graph is* seminormal *if:*

[7] This claim is not strictly formal, but holds for our algorithm and normalization levels.
[8] The simplest kind of factorization, defined in §7.4.7.1.

Figure 7.4: A factorizable graph $\mathcal{G}_1$ equivalent to $\mathcal{G}_0$.
Note that we repeat heads for clarity, but heads with the same dependants are formally the same node.
The graph is not seminormal because there is a node with multiple incoming horizontal edges, none of which has a variable head (orange).



1. *It is factorizable.*

2. *For any node N with more than one non-redundant incoming horizontal edge (edges E_i), there is at least one E_{i_0} whose head only contains variable dependants.*

The graph $\mathcal{G}_1$ in figure 7.4 is not seminormal because there is a node with multiple incoming horizontal edges, and all of the heads of the edges contain non-variable dependants. In contrast, the graph $\mathcal{G}_2$ in figure 7.5 is seminormal and has the same solutions as $\mathcal{G}_1$.

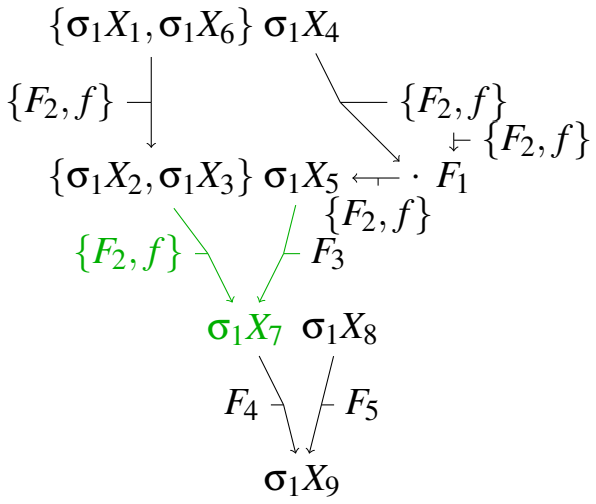
7.3.5 Quasinormal form

Quasinormalization may require non-determinism to reach (there may be a single dependency graph for which there does not exist any single quasinormal graph with the same set of solutions; and instead a set of quasinormal graphs are required to express the full set of solutions). It is relevant because a consistent quasinormal graph is guaranteed to have a non-empty set of solutions (theorem 7.6.9).

Definition 7.3.6. *A dependency graph is* quasinormal *if:*

1. *It is seminormal.*

Figure 7.5: A seminormal graph $\mathcal{G}_2$ equivalent to $\mathcal{G}_1$.
Note that we repeat heads for clarity, but heads with the same dependants are formally the same node.
The graph is not quasinormal because there is a node with multiple incoming horizontal edges, and some of them do not have variable heads (green).



2. *For any node N with more than one non-redundant incoming horizontal edge, all the heads of its non-redundant incoming horizontal edges contain only variable dependants.*

3. *Any second-order node with non-redundant incoming horizontal edges contains only variable dependants.*

The graph $\mathcal{G}_2$ in figure 7.5 is not quasinormal because there is a node with multiple incoming horizontal edges, and some of the heads of the edges contain non-variable dependants. In contrast, the graphs $\mathcal{G}_{3A}$ and $\mathcal{G}_{3B}$ in figure 7.6 are quasinormal and each include a subset of the solutions of $\mathcal{G}_2$. If we fully unfolded this into all the possible quasinormal graphs, these would combine to include all of the solutions of $\mathcal{G}_2$.

7.3.6 Normal form

Normalization is the maximum level of simplicity that a dependency graph can reach[9]. It is relevant because unification solutions can be extracted directly from normal graphs. Formally:

Definition 7.3.7. *A dependency graph is* normal *if:*

[9]This is entirely an informal statement.

Figure 7.6: Two quasinormal graphs $\mathcal{G}_{3A}$ and $\mathcal{G}_{3B}$, such that $\mathcal{U}(\mathcal{G}_2) \supseteq \mathcal{U}(\mathcal{G}_{3A}) \cup \mathcal{U}(\mathcal{G}_{3B})$.

The algorithm would unfold it into further non-deterministic quasinormal graphs but we only include 2 here to avoid overcrowding the diagram.

Note that we repeat heads for clarity, but heads with the same dependants are formally the same node.

Neither $\mathcal{G}_{3A}$ nor $\mathcal{G}_{3B}$ are normal because they both have a node with multiple incoming horizontal edges (red).

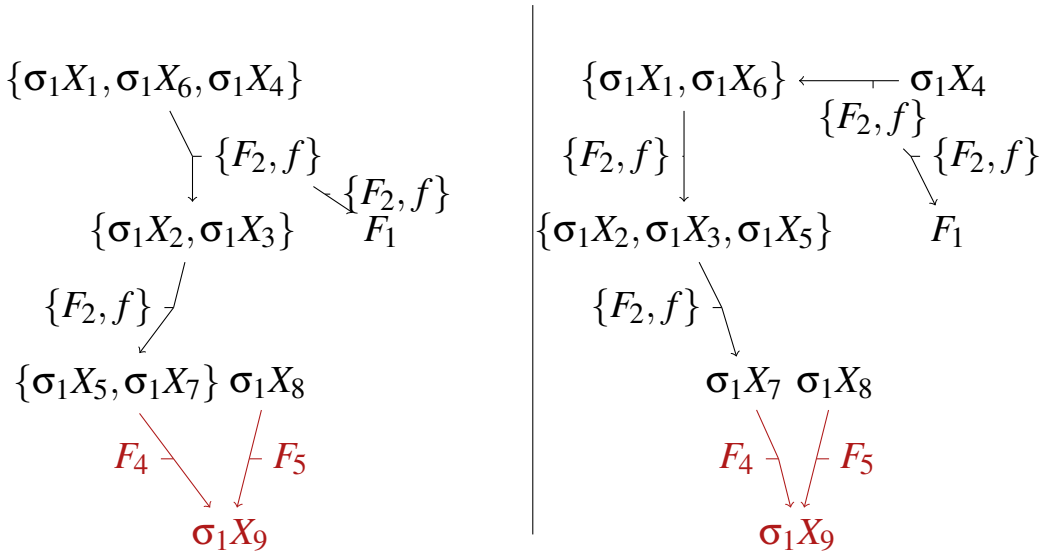
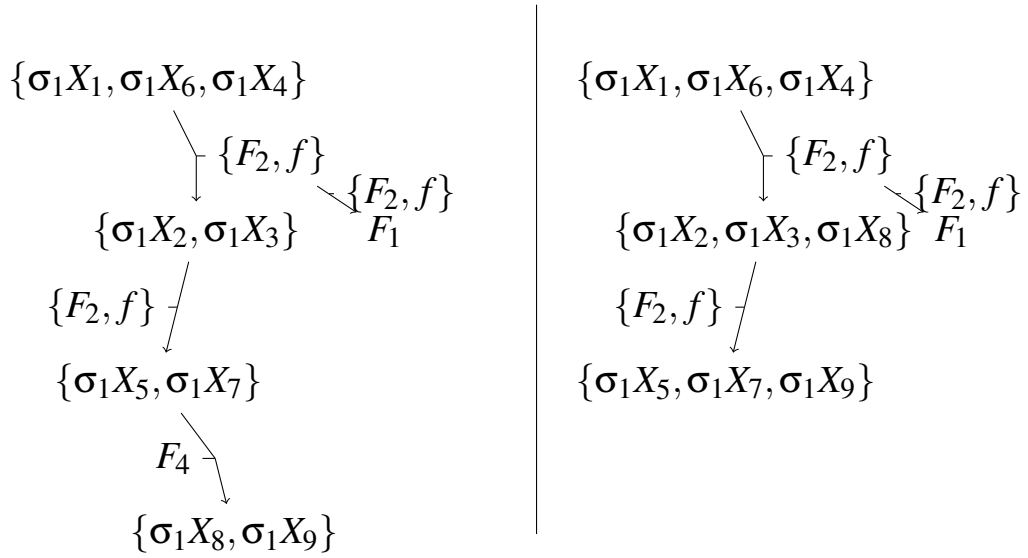


Figure 7.7: Two normal graphs $\mathcal{G}_{4A}$ and $\mathcal{G}_{4B}$, such that $\mathcal{U}(\mathcal{G}_{3A}) \supseteq \mathcal{U}(\mathcal{G}_{4A}) \cup \mathcal{U}(\mathcal{G}_{4B})$. The algorithm would unfold it into further non-deterministic normal graphs but we only include 2 here to avoid overcrowding the diagram.
Note that we repeat heads for clarity, but heads with the same dependants are formally the same node.
In both $\mathcal{G}_{4A}$ and $\mathcal{G}_{4B}$ each node has at most one incoming horizontal edge, which means every equation derived from the graph is either an equality or can be taken as a definition. This means that we can enumerate solutions in a straightforward way from the graph. See definition 7.6.4.



1. *It is quasinormal.*

2. *There is no node with more than one non-redundant incoming horizontal edge.*

The graph $\mathcal{G}_{3A}$ in figure 7.6 is not normal because there is a node with multiple incoming horizontal edges. In contrast, the graphs $\mathcal{G}_{4A}$ and $\mathcal{G}_{4B}$ in figure 7.7 are normal and each include a subset of the solutions of $\mathcal{G}_{3A}$. If we fully unfolded this into all the possible normal graphs, these would combine to include all of the solutions of $\mathcal{G}_{3A}$. The attractive of normal graphs is that each node has at most one incoming horizontal edge. This means that we can enumerate solutions in a straightforward way from the graph (see definition 7.6.4).

7.4 Rewrite rules for dependency graphs

The way in which we use a dependency graph can be described at a high level as follows:

1. Translate a system of unification equations into a dependency graph.
2. Apply a series of *rewrite rules* (see §3.4) to the graph. Each of these rewrite rules preserves the set of unification solutions of the graph. In order for this to be possible, these rewrite rules may be non-deterministic, meaning that from a single dependency graph they may produce a set of dependency graphs, such that the set of solutions for the original dependency graph is equal to the union of the sets of solutions for the resulting dependency graphs. In other words, the rewrite rules are applied to sets of dependency graphs to produce new sets of dependency graphs (possibly infinite sets).
3. Reach a certain *normalization level* for the dependency graph. At that normalization level, enumeration of unification solutions and/or verification of whether a given unification solution is represented by the graph is relatively straightforward.

In this section we will describe what the set of rewrite rules is, including the preconditions that the graph must fulfill for each rule to be applicable. Later on we will combine these rewrite rules to produce normalizing algorithms.

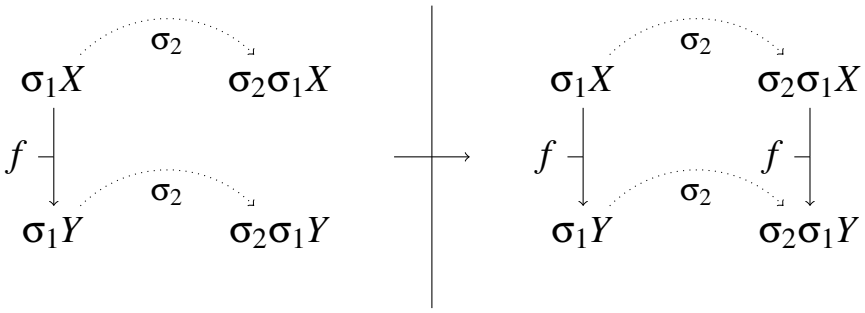
The intention of each rewrite rule is to change the *presentation* of the graph, while preserving its semantics (its set of solutions).

The basics of rewrite systems are described in §3.4 and are not novel to this work. However, our application to unification dependency graphs is naturally unique to our approach, and has multiple important complexities and elements that are particular to it, discussed in this and following sections.

From now on, we use the shorthand notation $\mathcal{U}(\mathbb{G})$, where $\mathbb{G}$ is a set of dependency graphs, defined as $\mathcal{U}(\mathbb{G}) = \bigcup_{\mathcal{G}\in\mathbb{G}} \mathcal{U}(\mathcal{G})$.

Definition 7.4.1 (Solution preserving rule)**.** *Given a rewrite rule R that transforms a dependency graph $\mathcal{G}$ into a countable set of dependency graphs $R(\mathcal{G})$ (non-deterministic transformation of a dependency graph), we say R* is solution preserving *if for all $\mathcal{G}$, $\mathcal{U}(R(\mathcal{G})) = \mathcal{U}(\mathcal{G})$.*

Figure 7.8: Vertical monotony.



We now describe the rewrite rules of our system.

7.4.1 Vertical monotony

Propagate dependencies encoded by equivalences (multiple dependants in a node) and first-order horizontal edges through vertical edges to explicitly represent the consequences of a horizontal edge on all unifier levels after that. For example, if $\sigma_1 Y = f(\sigma_1 X)$, then it must also be true that $\sigma_2\sigma_1 Y = f(\sigma_2\sigma_1 X)$. Graphically, the two graphs in figure 7.8 are equivalent.

Vertical monotony unfolds into three separate rules. The first two have to do with equivalences, and the third one with more complex dependencies. The first one deals with equivalences that are already explicit in the graph structure, and representing them in a better way, while the second one deals with syntactic equivalences that are not explicit in the graph structure.

Rewrite rule 1 (Vertical monotony of explicit equivalences)**.** Vertical monotony of explicit equivalences is applicable when

- There are two vertical edges V_1 and V_2 with the same source node S and unifier variable σ_i.

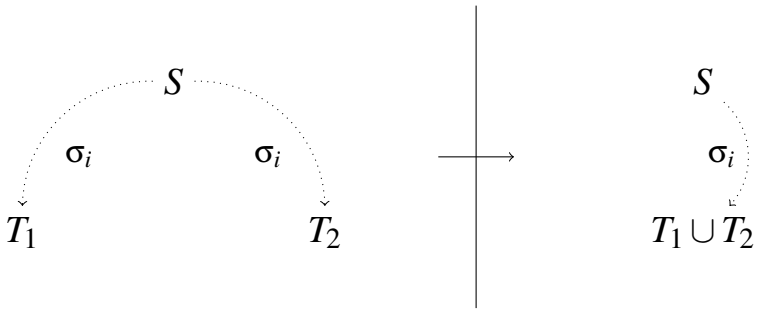
and does the following:

- Merge T_1 and T_2, the targets of V_1 and V_2 (respectively).
- Remove V_1 or V_2 (at this point they are absolutely equivalent) from the graph.

You can see a graphical depiction of this in 7.9.

Lemma 7.4.1. *Vertical monotony of explicit equivalences is a solution preserving rule.*

Figure 7.9: Vertical monotony of explicit equivalences. General case.



Proof. We begin with a dependency graph $\mathcal{G}_1$ and end with a dependency graph $\mathcal{G}_2$ equal to $\mathcal{G}_1$ except that T_1 and T_2 are merged and V_2 (w.l.o.g.) has been removed.

By definition of merging nodes and of the equation system associated with a dependency graph, the equation system of $\mathcal{G}_2$ is equivalent to that of $\mathcal{G}_1$, with the added equation $\kappa_{T_1} \approx \kappa_{T_2}$, and the equation $\kappa_{T_2} \approx \sigma_i \kappa_{S_2}$ removed.

But in $\mathcal{G}_1$, due to the presence of V_1, there is an equation $\kappa_{T_1} \approx_{\mathcal{G}_1} \sigma_i \kappa_S$ and because of V_2 we have $\kappa_{T_2} \approx_{\mathcal{G}_1} \sigma_i \kappa_S$. Combining these through transitivity of $\approx$, we have $\kappa_{T_1} \approx_{\mathcal{G}_1} \sigma_i \kappa_S \approx_{\mathcal{G}_1} \kappa_{T_2}$.

Similarly, we have $\kappa_{T_2} \approx_{\mathcal{G}_2} \kappa_{T_1} \approx_{\mathcal{G}_2} \sigma_i \kappa_S$, so the removing of V_2 does not change the associated solutions either.

Therefore, the set of solutions of $\mathcal{G}_1$ and the set of solutions of $\mathcal{G}_2$ are the same.

□

Rewrite rule 2 (Vertical monotony of syntactic equivalences)**.** Vertical monotony of syntactic equivalences is applicable when

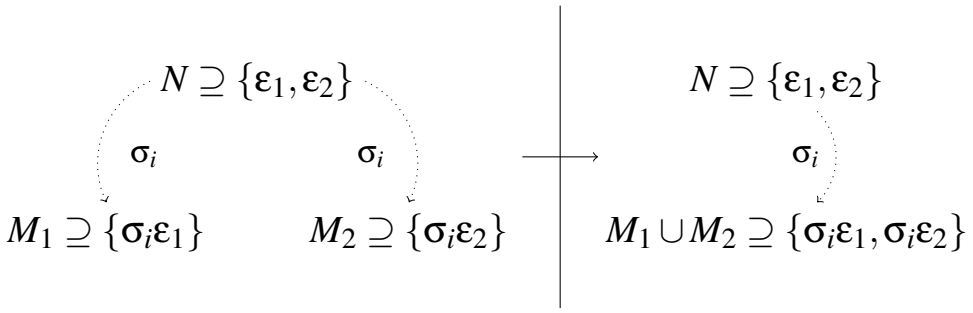
- There is a vertical edge V with source node N and unifier variable σ_i.
- N contains dependants ε_1 and ε_2.
- Dependants $\sigma_i \varepsilon_1$ and $\sigma_i \varepsilon_2$ are not in the same node.

and does the following:

- Merge the nodes result of grabbing $\sigma_i \varepsilon_1$ and $\sigma_i \varepsilon_2$.

You can see a graphical depiction of this in 7.10.

Figure 7.10: Vertical monotony of syntactic equivalences. General case.



As a note on this diagram and some of the subsequent ones, we would clarify the notation $\supseteq$, in particular in the form $N \supseteq \{\varepsilon_1, \varepsilon_2\}$. This is just a way to represent a *single* node, which we *label* as N (since the operation does something with it and we need a way to reference it) while at the same time adding a constraint to the rule stating that the node N **must contain** dependants ε_1 and ε_2.

Lemma 7.4.2. *Vertical monotony of syntactic equivalences is a solution preserving rule.*

Proof. We begin with a dependency graph $\mathcal{G}_1$ and end with a dependency graph $\mathcal{G}_2$ equal to $\mathcal{G}_1$ except that the nodes containing $\sigma_i\varepsilon_1$ and $\sigma_i\varepsilon_2$ are merged.

By definition of merging nodes and of the equation system associated with a dependency graph, the equation system of $\mathcal{G}_2$ is equivalent to that of $\mathcal{G}_1$, with the added equation $\sigma_i\varepsilon_1 \approx \sigma_i\varepsilon_2$. But in $\mathcal{G}_1$ we already have equation $\varepsilon_1 \approx \varepsilon_2$, and by congruence of $\approx$, then we also have $\sigma_i\varepsilon_1 \approx_{\mathcal{G}} \sigma_i\varepsilon_2$. Therefore, the set of solutions of $\mathcal{G}_1$ and the set of solutions of $\mathcal{G}_2$ are the same.

□

To formally define the vertical monotony of edges in the graph, first we provide a series of definitions that help us describe it precisely.

Definition 7.4.2 (Lifting of a dependant)**.** *Given a unifier variable σ_i and a dependant ε, we call the node result of grabbing the dependant $\sigma_i\varepsilon$ in the dependency graph the* lifting *of ε to σ_i.*

Definition 7.4.3 (Lifting of a first-order node)**.** *Given a first-order node N and a vertical edge V with unifier variable σ_i, we define the* lifting *of N through V, written $N[V]$, to be a first-order node*[10] *that is the target of the vertical edge with source N and unifier*

[10]If vertical monotony of explicit equivalences has been applied exhaustively then this node is unique.

variable σ_i. If N has no outgoing vertical edges with unifier variable σ_i, then $N[V]$ is added to the graph as an anonymous node alongside the vertical edge that joins them.

We will also sometimes use the notation $N[\sigma_i] \equiv N[V]$.

Definition 7.4.4 (Lifting of a horizontal edge)**.** *Given a vertical edge V and a horizontal edge H, the lifting of H through V, written $H[V]$, is a horizontal edge such that:*

- *The head of $H[V]$ is the same as the head of H.*
- *For each source S_i of H, $H[V]$ has $S_i[V]$ as a source.*
- *If the target of H is T, then the target of $H[V]$ is $T[V]$.*

Rewrite rule 3 (Vertical monotony of horizontal edges)**.** Vertical monotony of horizontal edges is applicable when:

- Vertical monotony of explicit equivalences is not applicable.
- Vertical monotony of syntactic equivalences is not applicable.
- There is a vertical edge V with source node N.
- There is a horizontal edge H with either source or target node N.
- $H[V]$ is not in the graph.

and does the following:

- Add $H[V]$ to the graph.

You can see a graphical depiction of this in figure 7.11.

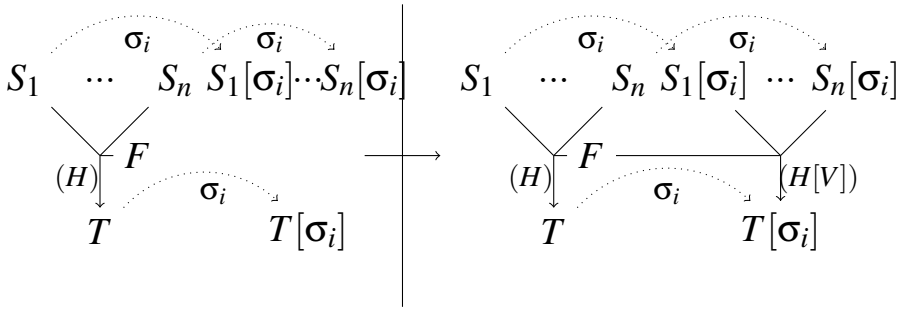
Lemma 7.4.3. *Vertical monotony of horizontal edges is a solution preserving rule.*

Proof. We begin with a dependency graph $\mathcal{G}_1$ and end with a dependency graph $\mathcal{G}_2$ that is equal to $\mathcal{G}_1$ except that the edge $H[V]$ has been added to the graph.

Let i be the unifier level of V. Let T be the target of H, with F its head and $\{S_j\}$ its sources. Then, in $\mathcal{G}_1$ the equation $\kappa_T \approx \chi_F(\kappa_{S_1}, ..., \kappa_{S_n})$ holds.

The target of $H[V]$ is $T[V]$, its sources are $\{S_j[V]\}$ and its head is F. Therefore, $\mathcal{G}_2$ has an equivalent equation system to $\mathcal{G}_1$ but with the added equation $\kappa_{T[V]} \approx \chi_F(\kappa_{S_1[V]}, ..., \kappa_{S_n[V]})$. Note, however, that for any node N, $\kappa_{N[V]} \approx \sigma_i \kappa_N$ holds in $\mathcal{G}_1$

Figure 7.11: Vertical monotony of horizontal edges. General case.
The structure of the edge is extended monotonously from the base unifier level to the unifier level i, using the vertical edges.
Note that the N in the definition could be any of $\{S_1, ..., S_n, T\}$ and V any of the vertical (dotted) edges (the corresponding one).



(and $\mathcal{G}_2$). This is trivially true if the graph contains dependants, and also true if the graph is anonymous because $\kappa_{N[V]}$ is defined through its incoming horizontal edges, all of which will be produced through vertical monotony.

Therefore, the following equations are all true in $\mathcal{G}_1$:

$$\begin{aligned}
\kappa_T &\approx \chi_F(\kappa_{S_1}, ..., \kappa_{S_n}) \\
\sigma_i\kappa_T &\approx \sigma_i\chi_F(\kappa_{S_1}, ..., \kappa_{S_n}) \\
\sigma_i\kappa_T &\approx \chi_F(\sigma_i\kappa_{S_1}, ..., \sigma_i\kappa_{S_n}) \\
\kappa_{T[V]} &\approx \chi_F(\kappa_{S_1[V]}, ..., \kappa_{S_n[V]})
\end{aligned}$$

and so the solutions of $\mathcal{G}_2$ are the same as those of $\mathcal{G}_1$.

□

7.4.2 Edge zipping

This rule can be applied to first or second-order edges. It represents the notion that if there are two edges with the same sources and head, then the targets must be equivalent. Merge the nodes to represent this. For example, if $F \cong g\{f\}$ and $G \cong g\{f\}$, then it must be true that $F \cong G$, as depicted in figure 7.12.

Rewrite rule 4 (Edge zipping)**.** Edge zipping is applicable when:

- There are two non-redundant horizontal edges E_1 and E_2 with the same head and the same sequence of sources.

Figure 7.12: Second-order edge zipping.

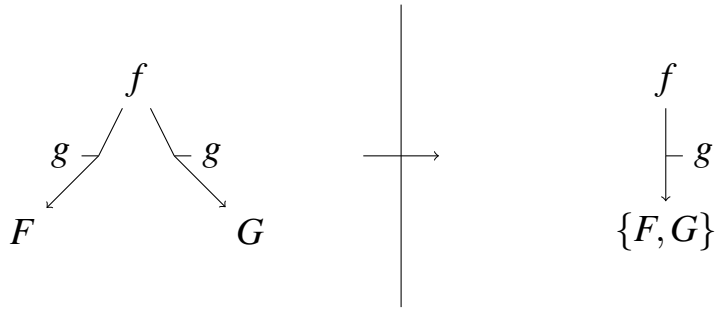
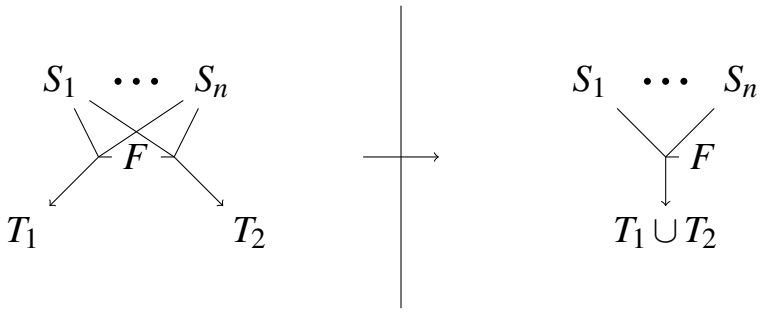


Figure 7.13: Edge zipping. General case.
Two distinct nodes with equivalent incoming edges must be equivalent.



- The targets T_1 and T_2 of E_1 and E_2 are not the same.

and does the following:

- Merge T_1 and T_2.
- Remove either E_1 or E_2 from the graph (at this point they are indistinguishable).

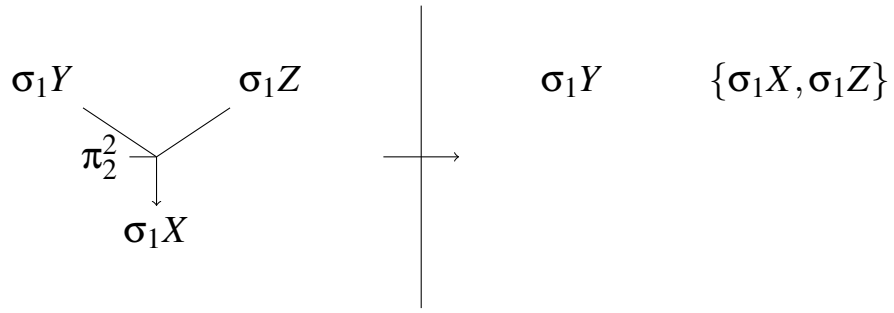
You can see a graphical depiction of this in figure 7.13.

Lemma 7.4.4. *Edge zipping is a solution preserving rule.*

Proof. We will produce parallel proofs for first and second-order zipping, by considering application or composition of functions in each case. The proofs are analogous otherwise so we consider both of them at the same time.

We begin with a dependency graph $\mathcal{G}_1$ and end with a dependency graph $\mathcal{G}_2$ equal to $\mathcal{G}_1$ except that the nodes T_1 and T_2 have been merged. Therefore, the associated equation system of $\mathcal{G}_2$ will be equivalent to that of $\mathcal{G}_1$ with added equation $\kappa_{T_1} \approx \kappa_{T_2}$.

Figure 7.14: Projection simplification of first-order edges.



Let F be the head node of E_1 and E_2, and $\{S_i\}$ their sources (which are equal for both edges by definition). Then, the edge E_1 produces the equation $\kappa_{T_1} \approx \chi_F(\kappa_{S_1}, ..., \kappa_{S_n})$ (if first-order; or with second-order instead of first-order proxies and composition instead of application if second-order nodes). Similarly edge E_2 produces equations $\kappa_{T_2} \approx \chi_F(\kappa_{S_1}, ..., \kappa_{S_n})$. In either case, it is true by transitivity of equivalence that $\kappa_{T_1} \approx \kappa_{T_2}$ holds in $\mathcal{G}_1$ (or with second-order proxies).

Therefore, $\mathcal{G}_1$ and $\mathcal{G}_2$ have the same solutions.

□

7.4.3 Projection simplification

When an edge has a projection as head, then we can remove the edge and merge the corresponding source with the target. For example, we can replace the equation $\sigma_1 X \approx \pi_2(\sigma_1 Y, \sigma_1 Z)$, with a simple $\sigma_1 X \approx \sigma_1 Z$, as depicted in figure 7.14.

The name corresponds to the name of the unifier expression rewrite rule that produces the same semantic result.

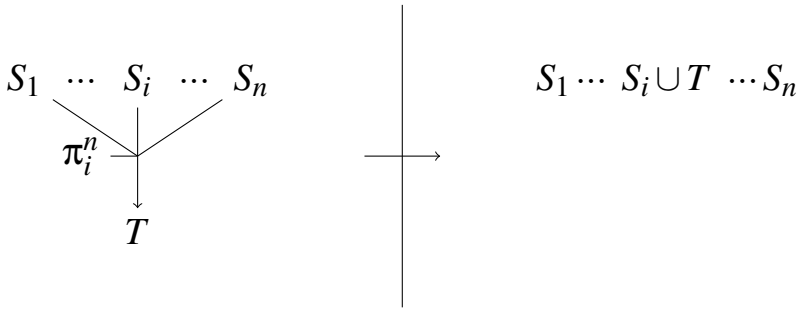
Rewrite rule 5 (Projection simplification)**.** Projection simplification is applicable when:

- There is a non-redundant horizontal edge E whose head node H, contains a projection π_i^n.
- The target T of E is not the same as the i-th source of E, S_i.

and does the following:

- Merge S_i and T.

Figure 7.15: Projection simplification. General case.
Merge the i-th target with the source of an edge with π_i^n as head.



- Mark E as redundant.

You can see a graphical depiction of this in figure 7.15.

Lemma 7.4.5. *Projection simplification is a solution preserving rule.*

Proof. We will produce parallel proofs for first and second-order zipping, by considering application or composition of functions in each case. The proofs are analogous otherwise so we consider both of them at the same time.

Let $\{S_j\}$ be the sources of the edge E. We begin with a dependency graph $\mathcal{G}_1$ and end with a dependency graph $\mathcal{G}_2$ equivalent to $\mathcal{G}_1$ except that the nodes S_i and T have been merged, and the edge E has been marked as redundant. This last part is equivalent to the removal of the edge for semantic purposes, since only non-redundant edges are considered in the semantics of the graph.

The merging of S_i and T produces the equation $\kappa_{S_i} \approx \kappa_T$. The removal of E removes the pre-existing equation $\kappa_T \approx \pi_i^n(\kappa_{S_1}, ..., \kappa_{S_i}, ..., \kappa_{S_n})$ for first-order (replacing first for second-order proxies and application for composition for second-order nodes). These equations, however, can be reduced, via the projection simplification rewrite rule for unifier expressions, to $\kappa_T \approx \kappa_{S_i}$ in both cases.

Therefore, the solutions of $\mathcal{G}_1$ and $\mathcal{G}_2$ are the same.

□

7.4.4 Occurs check

In usual unification theory, the *occurs check* refers to the verification that the substitution of a term does not properly (i.e. not through equality) contain itself. There is no finite substitution that can satisfy this. For example, first-order unification would write $x \sim f(x)$ or, using our unifier variable notation, $\sigma_i x \approx \sigma_i f(x)$, which implies $\sigma_i x \approx f(\sigma_i x)$.

In terms of graphs, this notion is equivalent to that of *directed cycles* in the graph expressing the dependencies between terms. That is, the occurs check in first-order unification establishes that **there may not be non-trivial (i.e. not equality) dependency cycles**.

The occurs check is also one of the more algorithmically complex parts of first-order unification, and while efficient solutions exist, the fundamental issue is that it *cannot be checked locally* and instead requires inspecting the entire unification problem (equivalently, the whole dependency graph).

The situation in our case is slightly more complicated, due to the presence of second-order variables, and specifically two ways in which these can be instantiated that affect the semantics of the occurs check:

- Second-order variables can be instantiated into projections, effectively making them trivial dependencies. For example, $\sigma_1 x \approx F(\sigma_1 x)$ can have solutions when $F \approx_U \pi^1_1$.
- Second-order variables with multiple arguments can be instantiated into second-order terms that do not actually depend on all of their arguments, effectively breaking the cycle. For example, $\sigma_1 x \approx F(\sigma_1 x, \sigma_1 y)$ can have solutions when $F \approx_U f\{\pi^2_2, \pi^2_2\}$.

You can see graphical examples of these situations in figures 7.16 and 7.17.

We can apply minimal commitment reasoning to incorporate these two cases with the occurs check into a rewrite rule that, when it finds cycles, either invalidates the graph or instantiates second-order variables in ways that will remove the cycles in the graph.

This still has the difficulty, that no other rewrite rule in this text has, of being a **global rule** that works on entire cycles in the graph of unbounded length, rather than

Figure 7.16: Directed cycles in a dependency graph that imply the graph has no solutions.
Any solution U to the left graph would have $\sigma_1 X \approx_U f(\sigma_1 X)$, which is a syntactic impossibility for any substitution σ_1.
The right graph has a similar issue with $\sigma_1 X \approx_U g(\sigma_1 X, \sigma_1 Y)$.

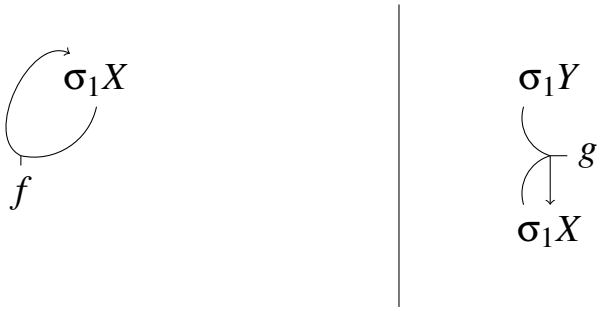
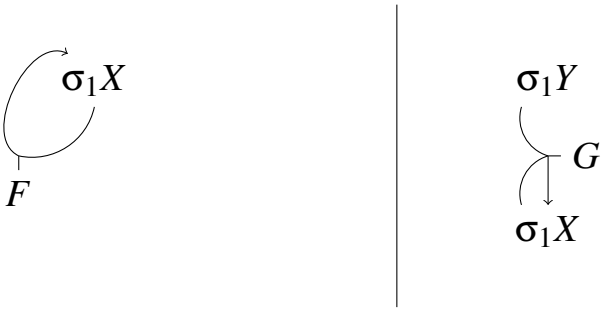


Figure 7.17: Directed cycles in a dependency graph that could have solutions depending on second-order variable instantiations.
For example, if $F \approx_U \pi_1^1$ or $G \approx_U g\{\pi_2^2, \pi_2^2\}$.



on limited parts of the graph.

Note, as well, that when considering cycles formed by second-order nodes, the heads of the edges are, for all intents and purposes, also sources with respect to graph cycles.

Rewrite rule 6 (Occurs check)**.** The *occurs check* is applicable whenever the graph is *prenormal* and *cyclic*. We reproduce definition 7.3.2 here, since the details are relevant to define the behaviour of the *occurs check* rule. A graph is cyclic if there is a set of nodes (either all first-order nodes or all second-order nodes) $\{N_i\}$, horizontal edges $\{E_i\}$ and indices j_i (we call them *source indices*); for $1 \leq i \leq n$ in the graph such that:

- For each $i < n$:
  - If $j_i = 0$, then N_i is the head of E_i.
  - If $j_i \neq 0$, then N_i is the j_i-th source of E_i.
  - N_{i+1} is the target of E_i.
- The same applies for $i = n$, except instead of N_{i+1}, N_1 is the target of E_n.

The occurs check rule then does the following (we will consider first and second-order cases separately here):

- **First-order cycle** - Produce the following non-deterministic branches:
  1. **Trivial (equality) dependency** - Consider the head H_i of each E_i. If any of the H_i contains any dependant that is not a second-order variable, do not produce any branch from this case.

     Otherwise, note that in first-order cases, the source index may not be zero because first-order nodes are not heads of edges. Then, produce a single branch in which we merge each H_i with $\pi_{j_i}^{m_i}$, where m_i is the number of sources of E_i.

  2. **Eliminate the dependency** - For each $1 \leq i \leq n$, produce a branch in the following way:

  - Consider the sources $\{S_{i,k}\}$, head H_i and target T_i of E_i. Let m_i be the number of sources of E_i.
  - Add a fresh second-order variable V_i, with arity $m_i - 1$.
  - Add a second-order edge E_i^2 with head V_i, sources $\pi_k^{m_i}$ for $k \neq j_i$, and target H_i
  - Add a first-order edge E_i' with head V_i, sources $\{S_{i,k}\}$, for $k \neq j_i$ (in the same order), and target T_i.
  - Mark E_i as redundant.

- **Second-order cycle** - It depends on whether some of the source indices are not head indices (0). There are two possibilities:

  1. **Trivial dependency / Eliminate dependency** - There is some $j_i \neq 0$. Then, produce the same two families of branches as above (cases 1 and 2), but apply everything only to those i for which $j_i \neq 0$ (and leave the others as is). As long as there is one $j_i \neq 0$, this will break the cycle.
  2. **Permutation** - If all the $j_i = 0$, then the above cases will not break the cycle. That is, each node is the head of the edge that moves to the next one. The only solutions to this situation are those in which the solutions to each node in the chain are almost equivalent to each other, but possibly with permutations of their arguments[11].

     Then, for each E_i, consider the number of sources of E_i, m_i. Consider all the permutations of $\{\pi_1^{m_i}, ..., \pi_{m_i}^{m_i}\}$. Given one such permutation P, write P_j to represent the j-th projection in that permutation. Consider as well the sources $S_{i,k}$ for $k \in 1..m_i$ of the E_i.

     For each combination of choices of n permutations $\{P^i\}$, for $1 \leq i \leq n$, consider the sequence of compositions of projections:

     $$\mathcal{P}^* = \{P_1^1\{P_1^2\{...\{P_1^n\{...\}, ...\}...\}, ...\}, P_2^1\{...\}, ..., P_{m_1}^1\{...\}\}$$

     For each combination of permutations for which $\mathcal{P}^*$ is equivalent (second-order term equivalence) to $\{\pi_1^{m_1}, ..., \pi_{m_1}^{m_1}\}$, and only for those (we show that for this to be true, all the m_i need to be equal); produce a branch in which the following is done:

[11] This is proven on lemma -pendingref-.

- Merge $S_{i,k}$ with P^i_k, for all $1 \leq i \leq n$ and for all $1 \leq k \leq m_i$.
- Mark E_n as redundant.

We will now show that *occurs check* is a solution preserving rule. But for that proof, it is useful to first show a smaller lemma about the *permutation* case.

Lemma 7.4.6. *Let $\{N_i\}$ and $\{E_i\}$ be a second-order cycle in a dependency graph with every $j_i = 0$. That is, each N_i is the head of each E_i.*

Then, for every solution U of the graph, and for every two $1 \leq i_1 \leq i_2 \leq n$, it is true that:

- *N_{i_1} and N_{i_2} have the same arity m.*
- $$\chi_{N_{i_1}} \approx_U \chi_{N_{i_2}}\{P_1,...,P_m\}$$
  $$\chi_{N_{i_2}} \approx_U \chi_{N_{i_1}}\{P_1,...,P_m\}$$
  where $\{P_1,...,P_m\}$ is some permutation of $\{\pi^m_1,...,\pi^m_m\}$.

In other words, all the N_i are the same function, with the only possible difference of changing the order of the arguments.

Proof. By virtue of the cycle and the fact that all $j_i = 0$, we have that for every i_1, all the following equations are true:

$$\begin{aligned}
&\chi_{N_{i_1}} \approx \\
&\chi_{N_{i_1-1}}\{...\} \approx \\
&\chi_{N_{i_1-2}}\{...\}\{...\} \approx \\
&... \\
&\chi_{N_1}\{...\}...\{...\} \approx \\
&\chi_{N_n}\{...\}...\{...\} \approx \\
&... \\
&\chi_{N_{i_1}}\{...\}...\{...\} \approx \\
&\chi_{N_{i_1}}\{\pi^{m_{i_1}}_1,...,\pi^{m_{i_1}}_{m_{i_1}}\}
\end{aligned}$$

Which means that the sources of each of the edges in the cycle must be such that when compounding them with each other, they produce the natural arguments $\{\pi^{m_{i_1}}_1,...,\pi^{m_{i_1}}_{m_{i_1}}\}$. The important part is that these natural arguments:

- Lose no information. They strictly depend on every argument.

- Have no compositions with function symbols.

In the solution U, any solution to the nodes $\{N_i\}$ that would contain function symbols or ignore some arguments would therefore not fulfill these conditions. If the N_i had different arities, then the smaller arity nodes would inevitably lose some information with respect to the larger arity ones.

Thus, all the N_i must have the same arity and all of the sources of these edges must be permutations of the natural arguments, since these are the only sets of arguments that contain no function symbols and strictly depend on all of the arguments.

□

Lemma 7.4.7. *Occurs check is a solution preserving rule*

Proof. We begin with a dependency graph $\mathcal{G}_1$. If the occurs check is applicable, then that means $\mathcal{G}_1$ is prenormal and cyclic. We end with a set of dependency graphs $\mathbb{G}_2$.

We will first show that for each graph $\mathcal{G}_2 \in \mathbb{G}_2$, $\mathcal{U}(\mathcal{G}_2) \subseteq \mathcal{U}(\mathcal{G}_1)$. Consider the possibilities for the occurs check rule:

- **First-order cycle**

  1. **Trivial (equality) dependency** - In this case, the result dependency graph $\mathcal{G}_2$ is the same as $\mathcal{G}_1$ except we have merged H_i with $\pi_{j_i}^{m_i}$; but this means we have **added** an equation $\chi_{H_i} \approx \pi_{j_i}^{m_i}$, and so the solutions of $\mathcal{G}_2$ are strictly contained in those in $\mathcal{G}_1$.
  2. **Eliminating the dependency** - We have n branches (for each $1 \leq i \leq n$) with a dependency graph $\mathcal{G}_2$ with the new edges E_i^2 and E_i' and the edge E_i removed. The introduced edges add equations, which restrict solutions, so we are not concerned with those now. Instead, we will look at what equations the marking of E_i as redundant may have removed.

     The edge E_i in $\mathcal{G}_1$ produces the equation:

     $$\kappa_{T_i} \approx_{\mathcal{G}_1} \chi_{H_i}(\kappa_{S_{i,1}}, ..., \kappa_{S_{i,m_i}}) \tag{7.3}$$

     But the edge E_i^2 in $\mathcal{G}_2$ produces the equation:

     $$\chi_{H_i} \approx_{\mathcal{G}_2} \chi_{V_i}\{\pi_1^{m_i}, ..., \pi_{j_i-1}^{m_i}, \pi_{j_i+1}^{m_i}, ..., \pi_{m_i}^{m_i}\} \tag{7.4}$$

and the edge E_i' produces the equation:

$$\kappa_{T_i} \approx_{\mathcal{G}_2} \chi_{V_i}(\kappa_{S_{i,1}}, ..., \kappa_{S_{i,j_i-1}}, \kappa_{S_{i,j_i+1}}, ..., \kappa_{S_{i,m_i}}) \tag{7.5}$$

which is equivalent to:

$$\kappa_{T_i} \approx_{\mathcal{G}_2} \chi_{V_i}\{\pi_1^{m_i}, ..., \pi_{j_i-1}^{m_i}, \pi_{j_i+1}^{m_i}, ..., \pi_{m_i}^{m_i}\}(\kappa_{S_{i,1}}, ..., \kappa_{S_{i,m_i}})$$

which, combined with equation 7.4, is equivalent to:

$$\kappa_{T_i} \approx_{\mathcal{G}_2} \chi_{H_i}(\kappa_{S_{i,1}}, ..., \kappa_{S_{i,m_i}})$$

which is equation 7.3, and so $\mathcal{U}(\mathcal{G}_2) \subseteq \mathcal{U}(\mathcal{G}_1)$.

- **Second-order cycle**

  1. The proofs for the *trivial dependency* and *eliminate dependency* cases is analogous to the first-order case.
  2. **Permutation** - In this case, the only information that might be lost is the one contained in E_n. This edge produces the equation

  $$\chi_{N_1} \approx_{\mathcal{G}_1} \chi_{N_n}\{P_1^n, ..., P_m^n\} \tag{7.6}$$

  From lemma 7.4.6, we know that in every solution, the arguments of this equation must be a permutation of the natural arguments (projections).

  Moreover, the following equation, derived from combining all the E_i except E_n, holds both in $\mathcal{G}_1$ and $\mathcal{G}_2$:

  $$\chi_{N_n} \approx \chi_{N_1}\{P_1^{-n}, ..., P_m^{-n}\} \tag{7.7}$$

  where the arguments is a combination of permutations, and therefore a permutation itself. But every permutation $\{P_1, ..., P_m\}$ has a unique inverse permutation $\{P_1^{-1}, ..., P_m^{-1}\}$ such that $\{P_1, ..., P_m\}\{P_1^{-1}, ..., P_m^{-1}\} = \{P_1^{-1}, ..., P_m^{-1}\}\{P_1, ..., P_m\} = \{\pi_1^m, ..., \pi_m^m\}$.
  Moreover, in $\mathcal{G}_1$ we have that the permutation in equation 7.6 and the one in 7.7 combine to produce the equations:

$$\chi_{N_1} \approx \chi_{N_1}\{P_1^{-n},...,P_m^{-n}\}\{P_1^n,...,P_m^n\}$$
$$\chi_{N_n} \approx \chi_{N_n}\{P_1^n,...,P_m^n\}\{P_1^{-n},...,P_m^{-n}\}$$

so $\{P_1^n,...,P_m^n\}$ and $\{P_1^{-n},...,P_m^{-n}\}$ (we will write P^n and P^{-n} for short) are inverses.

This means that, following equation 7.7, in $\mathcal{G}_2$, the following holds:

$$\chi_{N_n}\{P^n\} \approx_{\mathcal{G}_2} \chi_{N_1}\{P^{-n}\}\{P^n\} \approx \chi_{N_1}$$

which is equivalent to equation 7.6, and so $\mathcal{G}_1$ and $\mathcal{G}_2$ have the same solutions.

For the second half of the proof, we show that every solution $U \in \mathcal{U}(\mathcal{G}_1)$ is a solution to one $\mathcal{G}_2 \in \mathbb{G}_2$.

- **First-order cycle** - Imagine that in U, there is one i_0 for which the solution to H_{i_0} does not depend on N_i. But then we can rewrite the equation for E_i into another equation that does not depend on N_i. The dependency graph result of the **eliminate dependency** branch for i expresses the most general case possible for this situation, and this branch only differs from $\mathcal{G}_1$ in this particular aspect, and so U is a solution to the graph in this branch.

Therefore, we can assume from now on, without loss of generality, that each H_i strictly depends on all of its arguments. Then, we can show that it is not possible that for any of the i, $\chi_{H_i} \approx_U f$, for any function symbol f. Assume on the contrary that $\chi_{H_i} \approx_U f$. Then, the edges $\{E_1,...,E_n\}$ produce the following equation that holds in U:

$$\kappa_{N_1} \approx_U \kappa_{N_1}(...\chi_{H_i}(...)...) \approx_U \kappa_{N_1}(...f(...)...)$$

which means that κ_{N_1} depends properly on itself, which is a syntactic impossibility.

And so, each H_i cannot be equivalent, in U, to a function symbol f. But in a ground solution U, each second-order node has to be instantiated to either a

function symbol or a projection, and so each H_i must, in U, be equivalent to a projection.

Moreover, because E_i strictly depends on N_i, which is its j_i-th argument, we then know that $H_i \approx_U \pi_{j_i}^{m_i}$. The *trivial dependency* branch covers this case.

- **Second-order cycle** - We first note that for those i for which $j_i = 0$, then N_i is the head of E_i. Also note that it is not possible to eliminate the dependency of an edge on its head, so we can reproduce the argument for the eliminate dependency case for first-order cycles, but only for those i for which $j_i \neq 0$. The others necessarily depend on N_i.

  We can also, similarly to the first-order case, prove that, as long as all the E_i depend on N_i, then none of the H_i for which $j_i \neq 0$ can be equivalent, in a solution, to a function symbol, and must thus be projections. The eliminate dependency case represents exactly this case, with no further added equations, and thus the solutions that fall under this case are represented by this.

We thus only have left the case where all the $j_i = 0$. Lemma 7.4.6 shows that every solution U of $\mathcal{G}_1$ must be such that one of the sets of permutations described holds, and so for every solution, one of the permutation branches must cover this solution.

□

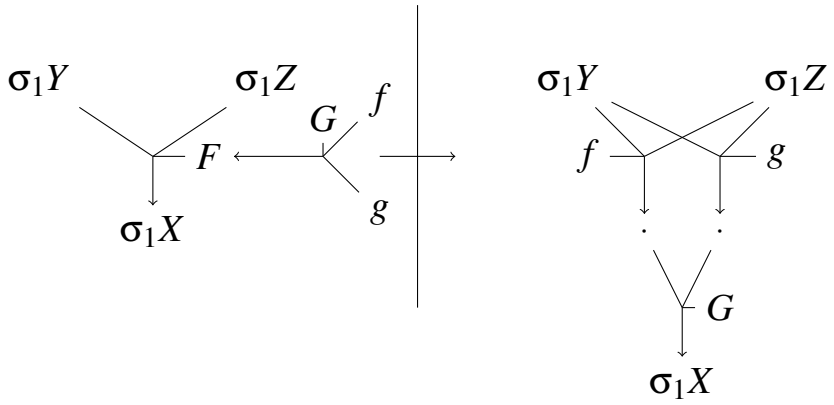
7.4.5 Function dumping

When an edge has a head that has incoming second-order edges, we can apply the instantiation represented by the second-order edge to the edge for which it is a head. For example, if $\sigma_1 X \approx F(\sigma_1 Y, \sigma_1 Z)$ and $F \approx G\{f, g\}$, then we can replace the first edge with edges and nodes corresponding to the equation $\sigma_1 X \approx G(f(\sigma_1 Y, \sigma_1 Z), g(\sigma_1 Y, \sigma_1 Z))$, as depicted in figure 7.18.

The name corresponds to the name of the unifier expression rewrite rule that produces the same semantic result.

Rewrite rule 7 (Function dumping)**.** Function dumping is applicable when:

Figure 7.18: Function dumping on a first-order edge.
The second-order edge targetting the node containing F remains in the graph, but we omit it on the right to simplify. We do present it in the general case for full representation.



- There is a non-redundant (first or second-order) horizontal edge E with head H, target T and sources S_i.
- There is a non-redundant second-order horizontal edge E^2 with target H, head H^2 and sources S_j^2.
- H does not contain a projection.

and does the following:

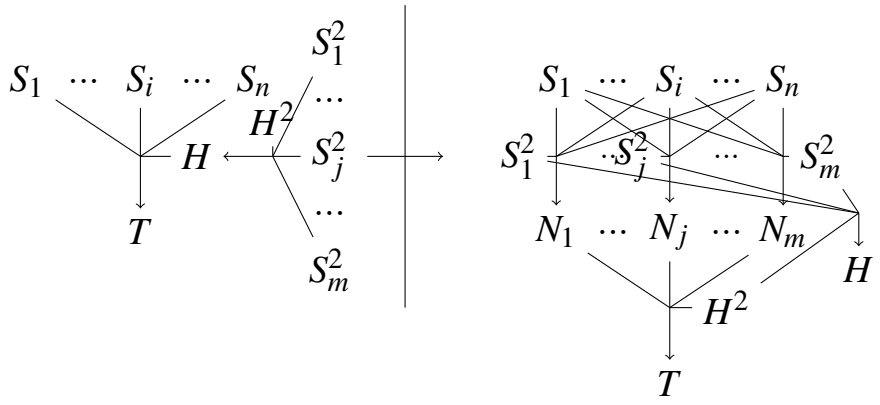
- Creates an anonymous node N_j associated with each S_j^2.
- For each N_j, creates a horizontal edge with sources $\{S_i\}$ (all of them, in the same order), target N_j and head S_j^2.
- Creates a horizontal edge with sources N_j (all of them, in the same order), target T and head H^2.
- Marks E as redundant.

You can see a graphical depiction of this in figure 7.19.

Lemma 7.4.8. *Function dumping is a solution preserving rule.*

Proof. We will produce parallel proofs for first and second-order function dumping, by considering application or composition of functions in each case. The proofs are analogous otherwise so we consider both of them at the same time.

Figure 7.19: Function dumping. General case.
Dump the second-order structure of the head H onto the first-order structure of the target T.



Call $\{E_j\}$ the edges created with the $\{N_j\}$ nodes as target and E_T the new edge with T as target. We begin with a dependency graph $\mathcal{G}_1$ and end with a dependency graph $\mathcal{G}_2$ equivalent to $\mathcal{G}_1$ except that the edge E has been marked as redundant (which is semantically equivalent to removing the edge from the graph), the nodes $\{N_j\}$ have been added, the edges $\{E_j\}$ have been added and the edge E_T has been added.

The creation of the nodes $\{N_j\}$ per se does not change the associated equation system in any way. The removal of the edge E removes the equation $\kappa_T \approx \chi_H(\kappa_{S_1}, ..., \kappa_{S_n})$ (or replacing first for second-order proxies and application for composition for second-order nodes). Note, however, that the edge E^2 (present both in $\mathcal{G}_1$ and $\mathcal{G}_2$) produces the equation $\chi_H \approx \chi_{H^2}\{\chi_{S_1^2}, ..., \chi_{S_m^2}\}$, and so the equation for E is equivalent to

$$\kappa_T \approx \chi_{H^2}\{\chi_{S_1^2}, ..., \chi_{S_m^2}\}(\kappa_{S_1}, ..., \kappa_{S_n}) \approx \chi_{H^2}(\chi_{S_1^2}(\kappa_{S_1}, ..., \kappa_{S_n}), ..., \chi_{S_m^2}(\kappa_{S_1}, ..., \kappa_{S_n}))$$

or the equivalent version with second-order proxies and composition for second-order nodes.

On the other hand, consider node N_j. It has exactly one associated equation, corresponding to the edge E_j, of the form $\kappa_{N_j} \approx \chi_{S_j^2}(\kappa_{S_1}, ..., \kappa_{S_n})$ (or the equivalent with second-order proxies and composition for second-order nodes). The edge E_T has the corresponding associated equation $\kappa_T \approx \chi_{H^2}(\kappa_{N_1}, ..., \kappa_{N_m})$ (or the respective second-order version). Replacing the expressions, we get

$$\kappa_T \approx \chi_{H^2}(\chi_{S_1^2}(\kappa_{S_1}, ..., \kappa_{S_n}), ..., \chi_{S_m^2}(\kappa_{S_1}, ..., \kappa_{S_n}))$$

or the equivalent version with second-order proxies and composition for second-order nodes.

Thus, the newly created edges produce the exact same equation that was removed, and so the solutions of $\mathcal{G}_1$ are the same as the solutions of $\mathcal{G}_2$.

□

7.4.6 Validate consistency

This rule verifies that certain types of inconsistency (equations that prevent any solutions from satisfying them) are not present in the dependency graph. For example, $f \approx g$, for constant function symbols f and g would be inconsistent. This rule does not change the graph itself. Instead, it either validates it or invalidates it. In terms of sets of dependency graphs, it produces an empty set from a singleton set. Note, however, that this rule *is solution preserving*, because it only invalidates a graph when it is verified to have no solutions at all.

Rewrite rule 8 (Validate consistency)**.** Validating consistency is applicable whenever the graph is acyclic and prenormal and checks that the following all hold in the dependency graph:

- For each second-order node, all non-variable dependants it contains are equivalent.
- For each second-order node, its recursive arity (definition 7.3.3) is well defined.
- For each non-redundant horizontal edge, the recursive arity of its head is equal to the number of sources of the edge.

failing (producing an empty set of graphs) if any of them do not hold.

Note that so while this definition is technically declarative, these checks are straightforward to implement into an actual algorithm.

Lemma 7.4.9. *For any dependency graph $\mathcal{G}$, if the validate consistency rule on $\mathcal{G}$ fails, then the dependency graph $\mathcal{G}$ has no solutions.*

Proof. We will enumerate the potential reasons for which the validate consistency rule may fail, and show that each of them indicates a situation producing equations that no unification solution may satisfy.

- There is a second-order node, N, with two non-variable dependants ϕ_1 and ϕ_2 that are not equivalent.

  Since ϕ_1 and ϕ_2 are non-variable, they must be of the form f_1 or $\pi^n_{i_1}$ and f_2 or $\pi^m_{i_2}$, for function symbols f_1 and f_2. In either case, all of these second-order terms are independent of instantiations. Because both dependants are in N, then $\mathcal{G}$ must satisfy an equation of the form $\phi_1 \approx \phi_2$, but these terms are not equivalent regardless of instantiation, and therefore no instantiation may satisfy this equation.

- There is a second-order node whose recursive arity is undefined. This may be because it contains dependants with different arities or incoming horizontal edges with sources with different recursive arities. The recursive arity of node N corresponds to the arity of second-order terms ϕ with which equations $\chi_N \approx \phi$ may appear in the equation system associated with the graph. So, by definition 7.2.3, undefined recursive arity would translate into equations of the form $\phi_1 \approx \phi_2$, where the arity of ϕ_1 and ϕ_2 are different. Equivalence between second-order terms with different arities may never be satisfied because all rewrite rules **for second-order terms** preserve arity, which is what defines the relation $\approx$.

- There is a horizontal edge E, with head H, such that E has n sources and the recursive arity of H is m, where $m \neq n$.

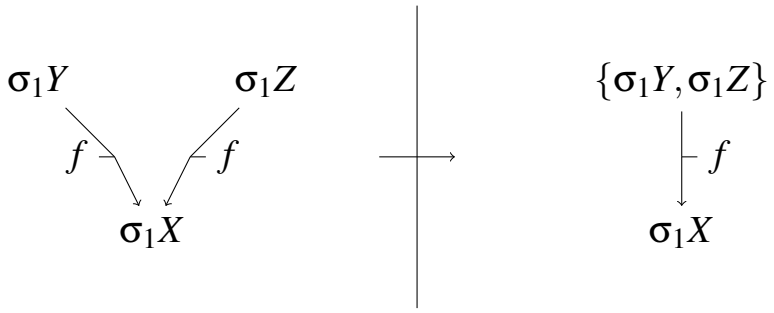
□

It is very important to note that while we just proved the *soundness* of the validate consistency rule (it will never fail for satisfiable graphs), it is not in general a *complete* rule: **there may be dependency graphs which pass the validate consistency rule, despite having no solutions**.

Corollary 7.4.1. *Validate consistency is a solution preserving rule.*

Proof. Validate consistency either leaves a graph as is, or invalidates it.

Figure 7.20: Zero factorization over a first-order node.



If it leaves it as is, then the set of solutions are trivially preserved.

If it invalidates it, by the lemma, we know that the original graph had no solutions. Thus, the set of solutions was empty and remains empty afterwards.

Therefore, the rule is solution preserving. □

7.4.7 Factorization

This is the most complex and most important family of rewrite rules. They are the only ones that introduce non-determinism[12], and normalization levels are defined mostly to differentiate between which of the different types of factorization are still applicable.

Factorization fundamentally consists in looking at nodes with more than one incoming edge and extracting conclusions about those edges' heads from the equivalence that this situation represents. We distinguish four types of factorization depending on the variable / non-variable nature of the heads of those edges. This is very closely related to flexible / rigid critical pairs in higher-order unification, and is meant mostly to distinguish situations with very different branching factors in its sets of solutions.

7.4.7.1 Zero factorization

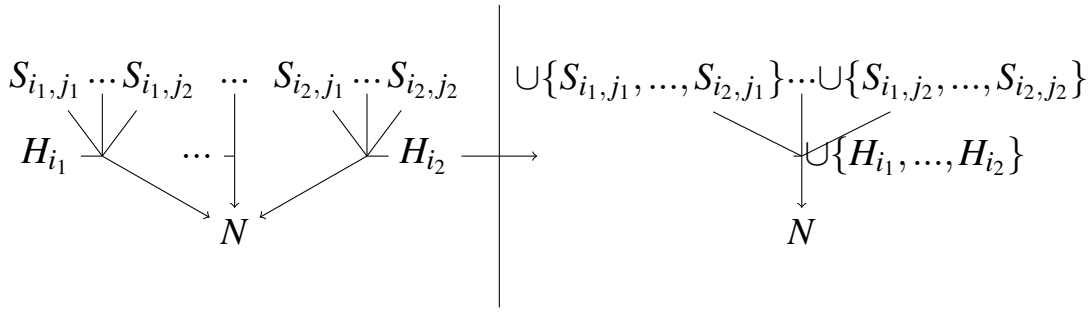
Zero factorization is applicable when all incoming edges to a node have function symbols as heads. They must be equal and their sources must be equal, respectively. For example, if we have edges indicating that $\sigma_1 X \approx f(\sigma_1 Y)$ and $\sigma_1 X \approx f(\sigma_1 Z)$, then we can conclude that $\sigma_1 Y \approx \sigma_1 Z$. This is depicted in figure 7.20.

Rewrite rule 9 (Zero factorization)**.** Zero factorization is applicable when:

[12] Except zero factorization, which is still conceptually a factorization rule as per the informal definition below.

Figure 7.21: Zero factorization. General case.

Merge the heads and sources of incoming edges as long as all heads are non-variable.



- The dependency graph is factorizable.
- A node N has several non-redundant incoming horizontal edges $\{E_i\}$.
- Each head H_i of E_i contains at least one non-variable dependant.

and does the following:

- Checks that all the non-variable dependants contained in each H_i are equivalent. If they are not, the dependency graph is marked as invalid (i.e. an empty set of graphs is produced as a result of applying the rule).

  Due to the preconditions for the rule, this is the same as checking that all the H_i are the same node (all the H_i contain equivalent non-variable dependants, but non-variable dependants which are equivalent must necessarily be equal, and therefore by definition of dependency graph they are contained in the same node).

- Consider the sources $S_{i,j}$ of edges E_i ($S_{i,j}$ is the j-th source of the i-th incoming edge). For each j, merge all $S_{i,j}$[13].
- Remove all but one E_i from the graph (at this point they are indistinguishable).

You can see a graphical depiction of this in figure 7.21.

Lemma 7.4.10. *If zero factorization invalidates a dependency graph $\mathcal{G}$, then $\mathcal{G}$ had no solutions.*

[13]Because the graph is factorizable (definition 7.3.4), then the recursive arity of all nodes is well defined, and so the number of sources of each E_i must be the same.

Proof. We will produce parallel proofs for first and second-order zero factorization, by considering application or composition of functions in each case. The proofs are analogous otherwise so we consider both of them at the same time.

We will proceed by reductio ad absurdum. Assume zero factorization invalidates $\mathcal{G}$ but $\mathcal{G}$ had solutions. If zero factorization invalidates $\mathcal{G}$, then $\mathcal{G}$ has a node N with at least two non-redundant incoming horizontal edges E_{i_1}, E_{i_2}, with heads H_{i_1}, H_{i_2} which contain non-variable dependants $\phi_{H_{i_1}}$, $\phi_{H_{i_2}}$, and such that they are not equal, with equations $\chi_{H_{i_1}} \approx \phi_{H_{i_1}}$ and $\chi_{H_{i_2}} \approx \phi_{H_{i_2}}$. Consider the sources $S_{i_1,j}$ of E_{i_1} and $S_{i_2,j}$ of E_{i_2}. The edges E_{i_1} and E_{i_2} have the same target node N, and therefore they imply the equation $\chi_{H_{i_1}}(\kappa_{S_{i_1,1}}, ..., \kappa_{S_{i_1,n}}) \approx \chi_{H_{i_2}}(\kappa_{S_{i_2,1}}, ..., \kappa_{S_{i_2,n}})$ (or the equivalent version replacing first for second-order proxies and application for composition if they are second-order nodes). This is equivalent to

$$\phi_{H_{i_1}}(\kappa_{S_{i_1,1}}, ..., \kappa_{S_{i_1,n}}) \approx \phi_{H_{i_2}}(\kappa_{S_{i_2,1}}, ..., \kappa_{S_{i_2,n}})$$

which must be true in every solution of $\mathcal{G}$.

Since $\mathcal{G}$ must be factorizable, then the non-variable heads $\phi_{H_{i_1}}$ and $\phi_{H_{i_2}}$ must be function symbols. This allows us to reconsider the equation above. The heads of both sides of the equation are function symbols, so the only rewrite rules (for second-order terms) that may reduce them (to make them have the same normal form) are recursive reductions on the arguments. Either way, their equivalence means that it must be $\phi_{H_{i_1}} \cong \phi_{H_{i_2}}$, which, in the case of function symbols, also implies $\phi_{H_{i_1}} \equiv \phi_{H_{i_2}}$. This is a contradiction.

□

Lemma 7.4.11. *Zero factorization is a solution preserving rule.*

Proof. We will produce parallel proofs for first and second-order zero factorization, by considering application or composition of functions in each case. The proofs are analogous otherwise so we consider both of them at the same time.

Zero factorization begins with a graph $\mathcal{G}_1$ and produces a dependency graph $\mathcal{G}_2$ or none, meaning no solutions.

By lemma 7.4.10, whenever the rule invalidates the graph, $\mathcal{G}_1$ had no solutions, and so the set of solutions is preserved.

If $\mathcal{G}_2$ is non-empty, it means that all the H_i were equal. The equation system associated with $\mathcal{G}_2$ is equivalent to the one of $\mathcal{G}_1$ except for the result of merging the sources $S_{i,j}$ and the marking of all but one of the E_i edges as redundant (which is semantically equivalent to removing them from the graph).

In $\mathcal{G}_1$, each incoming E_i to N produced an equation $\kappa_N \approx \chi_{H_i}(\kappa_{S_{i,1}}, ..., \kappa_{S_{i,n}})$ (or equivalently with second-order proxies and composition for second-order nodes). But by assumption we know that all the χ_{H_i} are equivalent, so we will use χ_H to refer to any of them. Therefore, for each two different i_1 and i_2, we have equation $\chi_H(\kappa_{S_{i_1,1}}, ..., \kappa_{S_{i_2,n}}) \approx \chi_H(\kappa_{S_{i_2,1}}, ..., \kappa_{S_{i_2,n}})$. Because $\mathcal{G}_1$ is factorizable, χ_H must be a second-order variable or a function symbol. Therefore, the only rewrite rule that may apply to these two expressions is recursive reduction of their arguments. Thus, we can conclude that, for all j, $\kappa_{S_{i_1,j}} \approx_{\mathcal{G}_1} \kappa_{S_{i_2,j}}$.

Therefore, merging the nodes $S_{i,j}$ does not change the solutions to the graph. Similarly, the E_i are equal since all their associated equations are equivalent, and therefore removing all except one of them does not change the solutions.

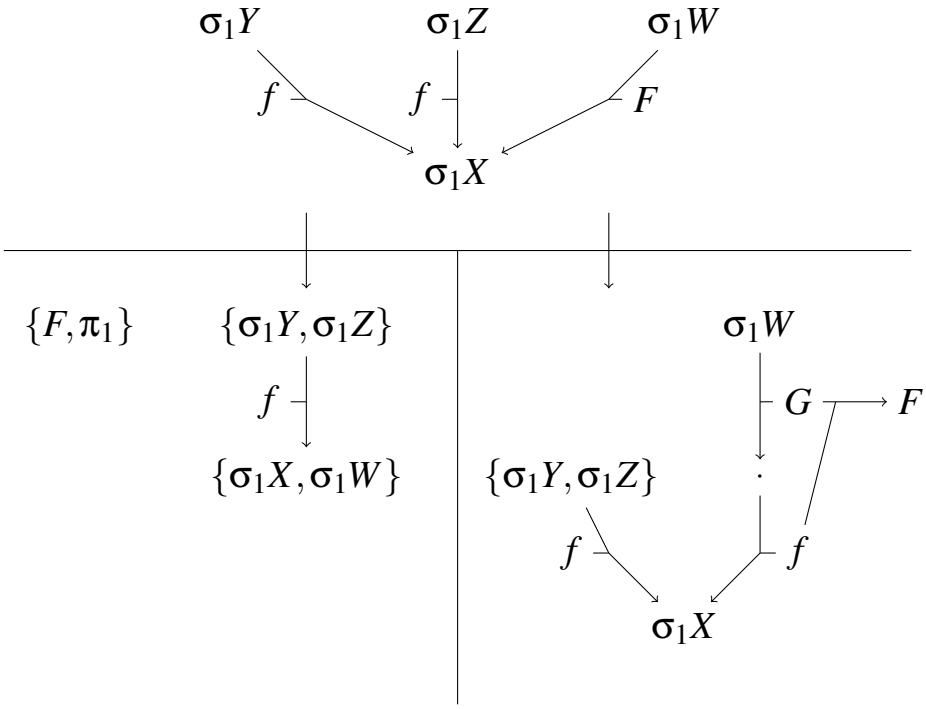
□

7.4.7.2 Single factorization

Single factorization is applicable when there are both constant and variable functions as heads of incoming edges to a node. All constant function symbols must be equal, and the variable functions may be instantiated in associated solutions to either projections or compositions whose heads are the constant function. We produce associated graphs for these options non-deterministically, and potentially some of these may be incompatible with other elements of the graph. For example, if we have equations stating that $\sigma_1 X \approx f(\sigma_1 Y)$, $\sigma_1 X \approx f(\sigma_1 Z)$ and $\sigma_1 X \approx F(\sigma_1 W)$, then we know that in every solution $\sigma_1 Y \approx \sigma_1 Z$ and then, either $F \approx \pi_1$ and $\sigma_1 X \approx \sigma_1 W$, or $F \approx f\{G\}$, for some other second-order variable G. Note that both options have a large number of consequences, but all of these consequences will come after making the graph factorizable again. See figure 7.22 for a graphical depiction of this.

Rewrite rule 10 (Single factorization)**.** Single factorization is applicable when:

Figure 7.22: Single factorization over a first-order node. The results in this diagram include some basic propagation after the actual factorization. Specifically, single factorization will always be followed by projection simplification or function dumping.



- The dependency graph is factorizable.
- A node N has several non-redundant incoming horizontal edges $\{E_i\}$.
- There is at least one head of an E_i that contains a non-variable dependant. Call $\{H_i^c\}$ the heads that contain non-variable dependants, and their respective edges $\{E_i^c\}$.
- There is at least one head of an E_i that only contains variable dependants (and contains at least one). Call $\{H_i^v\}$ the heads that contain only variable dependants, and their respective edges $\{E_i^v\}$.

and does the following first (this part is equivalent to zero factorization over only the constant-headed edges):

- Checks that all non-variable dependants contained in all H_i^c are equivalent. If they are not, invalidate the dependency graph (produce an empty set of dependency graphs as result). Otherwise, write H^c for one such dependant.

- Consider the sources $\{S^c_{i,j}\}$ of edges E^c_i ($S^c_{i,j}$ is the j-th source of the i-th incoming edge). Note again that the number of sources of all E^c_i must be the same because of the definition of factorizable graph (definition 7.3.4). For each j, merge all $S^c_{i,j}$.
- Remove all but one E^c_i from the graph (at this point they are indistinguishable).

and then non-deterministically produces different dependency graphs. Pick one head H^v from H^v_i (arbitrarily[14]), and its associated edge E^v. Let n be the number of sources of E^v. Produce the following branches:

1. Merge H^v with the node result of grabbing π^n_k, for each k from 1 to n (one non-deterministic branch for each k), where n is the recursive arity of H^v.
2. An additional branch with the following steps applied:
   - Add m fresh second-order variables to the graph. Call their nodes $\{V_l\}$, for l from 1 to m, where m is the recursive arity of H^c.
   - Add a second-order edge with H^c as head, all the V_l as sources and H^v as target.

You can see a graphical depiction of this in figure 7.23.

Lemma 7.4.12. *Single factorization is a solution preserving rule*

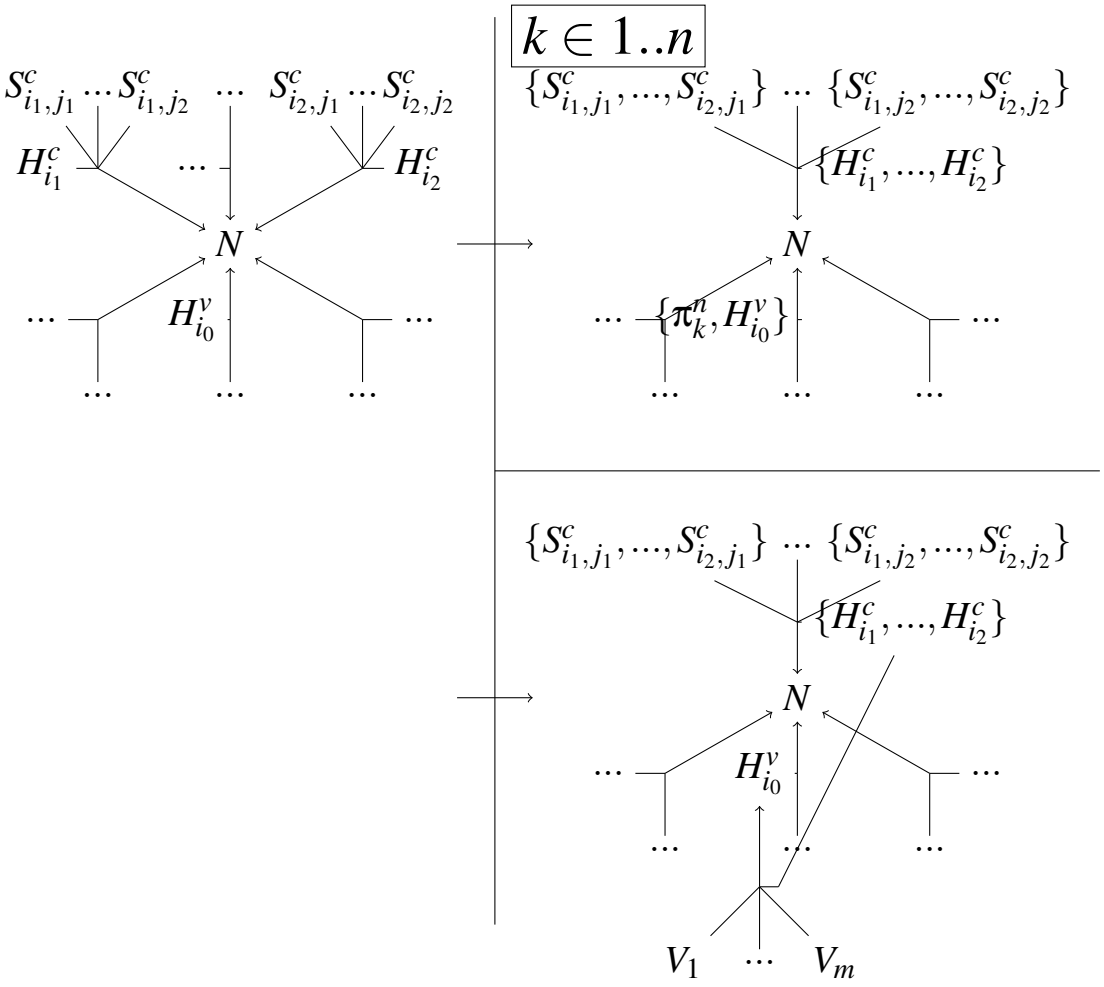
Proof. We will produce parallel proofs for first and second-order single factorization, by considering application or composition of functions in each case. The proofs are analogous otherwise so we consider both of them at the same time.

Single factorization departs from a graph $\mathcal{G}_1$ and produces a (possibly empty) set of dependency graphs $\mathbb{G}_2$.

If $\mathbb{G}_2$ is empty, it means that there were two $H^c_{i_1}$ and $H^c_{i_2}$ that were not equivalent. This is absolutely analogous to the case of zero factorization and so lemma 7.4.10 applies: $\mathcal{G}_1$ had no solutions, and therefore the rule is solution preserving in this case.

[14]This rule is instantiating one second-order variable to everything it can possibly be via non-determinism. The other second-order variables in the same situation will be dealt with eventually by this or other rules in each of those non-deterministic branches, after the graph has been made factorizable again.

Figure 7.23: Single factorization. General case.
Each variable head among incoming horizontal edges will necessarily be instantiated to either a projection or a composition with the non-variable head as head.



Otherwise, all the H_i^c were equal. Single factorization is a two step rule. First, a common part (zero factorization) is applied, producing a single dependency graph; and afterwards, the graph is branched out on different parts. The first step makes changes equivalent to those produced in zero factorization. Therefore, the arguments provided in the proof of lemma 7.4.11 are applicable here in exactly the same way. So we can focus our attention exclusively on the branching aspect. Call the intermediate graph after producing this first step $\mathcal{G}_{1.5}$

Consider each graph result of merging H^v with π_k^n in $\mathcal{G}_{1.5}$. Call such graph $\mathcal{G}_2^{\pi_k^n}$. Similarly, consider the single graph result of adding the variable nodes $\{V_l\}$ to the graph. Call this graph $\mathcal{G}_2^V$. By definition of single factorization, $\mathbb{G}_2 = \mathcal{G}_2^V \cup \bigcup_{k \in 1..n} \mathcal{G}_2^{\pi_k^n}$. Each of these graphs has an associated equation system, and we need to show that their union is equivalent (has the same solutions) as $\mathcal{G}_{1.5}$. We will use lemma 7.1.1. Note that each of the graphs in $\mathbb{G}_2$ has strictly more equations than $\mathcal{G}_{1.5}$, and so the implication in that direction is trivial, we need only show the implication in the other direction: each solution of $\mathcal{G}_{1.5}$ is a solution of a graph in $\mathbb{G}_2$.

The edge E^v produces the equation $\kappa_N \approx \chi_{H^v}(\kappa_{S_1^v}, ..., \kappa_{S_n^v})$, both in $\mathcal{G}_{1.5}$ and in every graph in $\mathbb{G}_2$. On the other hand, the edge E_i^c (for the i that were not removed in the first step of the rule) produces the equation $\kappa_N \approx \chi_{H_j^c}(\kappa_{S_{j,1}^c}, ..., \kappa_{S_{j,m}^c})$, for all j. By transitivity, we have

$$\chi_{H^v}(\kappa_{S_1^v}, ..., \kappa_{S_n^v}) \approx \chi_{H_j^c}(\kappa_{S_{j,1}^c}, ..., \kappa_{S_{j,m}^c})$$

We also know that H^v contains only variable dependants. Consider one such variable F. We also know that H_j^c contains at least one non-variable dependant. We also know that $\mathcal{G}_1$ was factorizable, and so this dependant may not be a projection. Thus, it is a function symbol. Call this function symbol f. We therefore have

$$F(\kappa_{S_1^v}, ..., \kappa_{S_n^v}) \approx f(\kappa_{S_{j,1}^c}, ..., \kappa_{S_{j,m}^c})$$

In each solution of the equation, the evaluations of both sides of the equation have the same normal form. But, for any unification solution, the only rewrite rule applicable to the right side of this equation is recursive reduction on its arguments, and so the normal form associated with it must have f as head. Consider the potential instantiations for F in a unification solution:

- F is instantiated to a function symbol g. Then, the normal form of the left

side will, for the same reasons, have head g, and so it must be $g \equiv f$. But, the instantiation that instantiates V_l to π_l^n is a solution to the dependency graph $\mathcal{G}_2^V$, which implies the equation $F \approx f\{V_1, ..., V_m\}$, and therefore the instantiation that instantiates F to $g \equiv f$ corresponds to the solution of $\mathcal{G}_2^V$ that instantiates the V_l to projections. $\mathcal{G}_2^V$ covers this case.

- F is instantiated to a projection π_k^n. But this is equivalent to the equation $\chi_{H^v} \approx \pi_k^n$, which is covered by the graph $\mathcal{G}_2^{\pi_k^n}$.

- F is instantiated to a composition $\phi_0\{\phi_1, ..., \phi_p\}$, but then the left side is reducible to

$$F(\kappa_{S_1^v}, ..., \kappa_{S_n^v}) \xrightarrow{*} \phi_0(\phi_1(\kappa_{S_1^v}, ..., \kappa_{S_n^v}), ..., \phi_p(\kappa_{S_1^v}, ..., \kappa_{S_n^v}))$$

  If ϕ_0 is a projection, then the instantiation of F is reducible to ϕ_j, and we can recursively consider what the instantiation ϕ_j may be. Thus, we may assume without loss of generality, not only that ϕ_0 is not a projection symbol but also that recursively it is not any number of compositions with a projection as head on the last level. And therefore, the head on the last level of compositions must be a function symbol, and it will be the head of its normal form, and therefore must be f. And thus, because solutions are equivalent up to equivalence of expressions / second-order terms, we can assume, without loss of generality, that $\phi_0 \equiv f$. But then the instantiation that instantiates V_l to ϕ_l corresponds to this solution in the dependency graph $\mathcal{G}_2^V$, and so $\mathcal{G}_2^V$ covers this case.

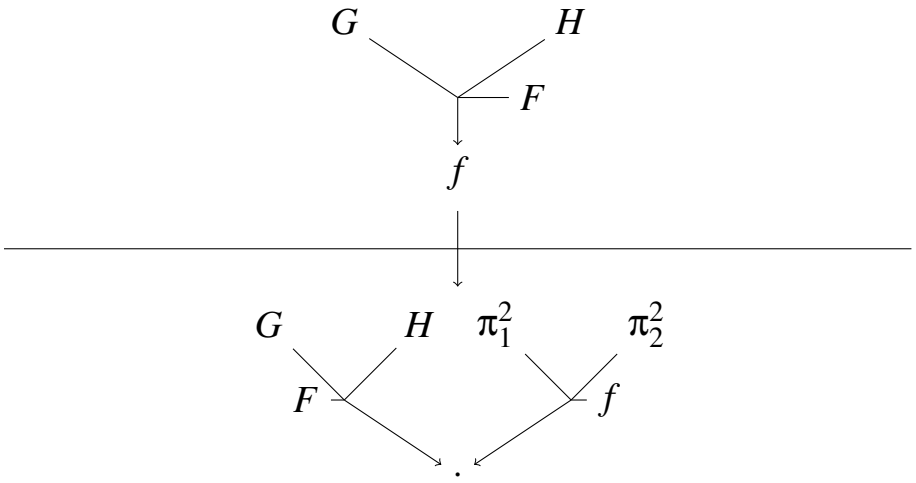
Therefore, the set of solutions of $\mathcal{G}_{1.5}$ is equal to the union of the sets of solutions of graphs in $\mathbb{G}_2$, and so single factorization is a solution preserving rule.

□

7.4.7.3 Half factorization of function symbols

Half factorization is a degenerate case conceptually related to single factorization but not easily covered by the graph conditions for single factorization, and instead better dealt with separately. It happens when, in extreme cases, a dependant with a function symbol may have incoming second-order edges and still be a consistent graph. Normally, incoming edges indicate a composition, which can never be syntactically equal to a function symbol. For this to be consistent, either the source or the head of said edge

Figure 7.24: Half factorization of the function symbol f.



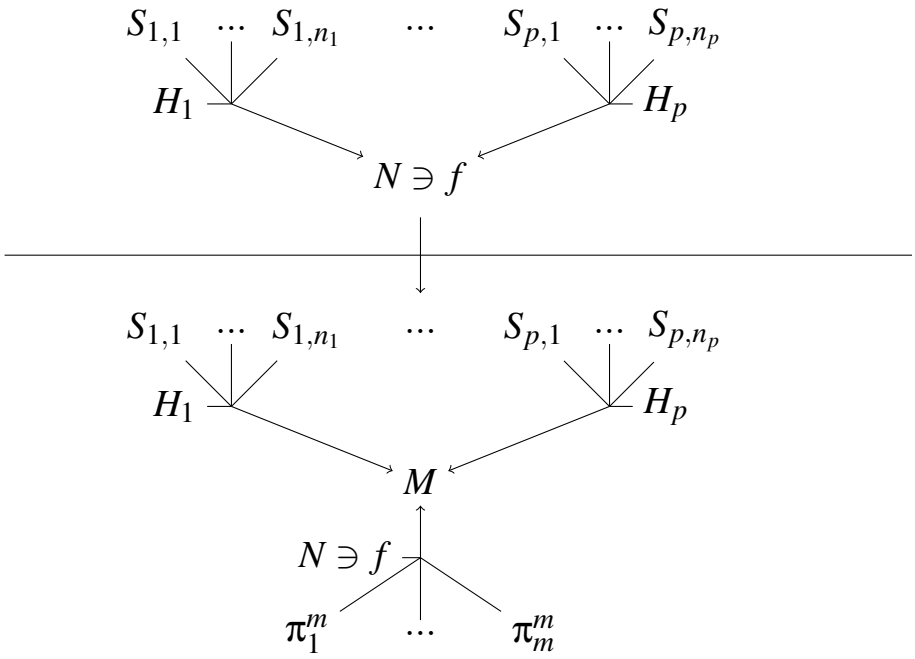
must be equivalent to either a variable or the function symbol itself, albeit with possibly some argument permutation. The way in which we deal with this in graphs is to translate the graph into an equivalent, very similar one that single factorization itself can deal with and which does not have function symbols with incoming edges. For example, if $f \approx F\{G,H\}$, then in each solution U one of the following must be true:

- $F \approx_U \pi_1^2$ and $G \approx_U f$.
- $F \approx_U \pi_2^2$ and $H \approx_U f$
- $F \approx_U f$, $G \approx_U \pi_1^m$ and $H \approx_U \pi_2^m$.
- $F \approx_U f\{P\}$, where P is some permutation of projections, and so are G and H.

However, instead of enumerating all of these cases individually, half factorization of function symbols simply presents the equation $f \approx F\{G,H\}$ in a way that single factorization can deal with: through multiple incoming edges to the same node. It is single factorization that produces the branching later on. The way it does this is to produce an anonymous second-order node with an incoming edge with f as head and moves all edges from the node containing f to that one.
See figure 7.24 for a graphical depiction of this.

Rewrite rule 11 (Half factorization of function symbols)**.** Half factorization of function symbols is applicable when:

Figure 7.25: Half factorization of function symbols. General case.
If a function symbol has incoming horizontal edges, factorize the function symbol out to let single or zero factorization deal with it.



- The dependency graph is factorizable.
- A second-order node N contains a dependant of the form f, for function symbol f. Let m be the arity of f.
- N has non-redundant incoming horizontal edges $\{E_i\}$, each with head H_i and n_i sources $\{S_{i,j}\}$, for j from 1 to n_i.

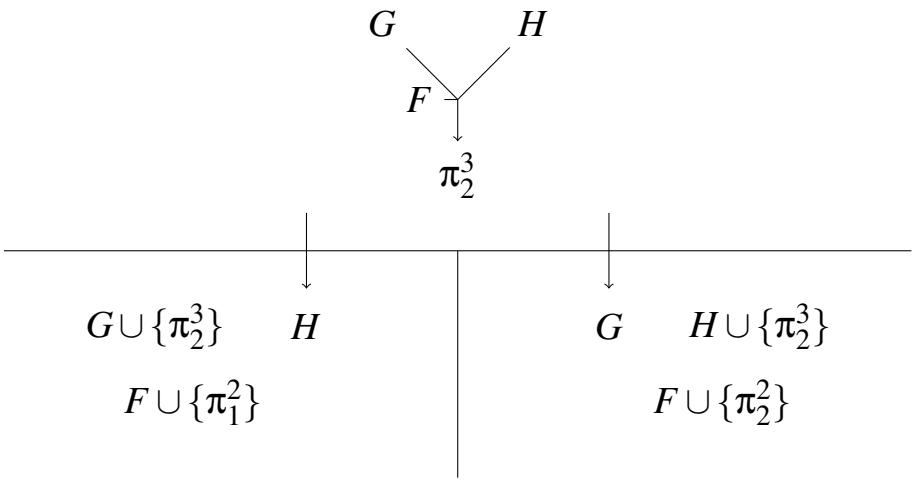
and does the following on the graph:

- Create an anonymous second-order node. Call it M.
- Create an edge E_M with head N, sources π_k^m, for all k from 1 to m, in the natural order, and target M.
- Mark each edge E_i as redundant, and add a new edge E_i^M with the same heads and sources as E_i, but target M instead of N.

You can see a graphical depiction of this in figure 7.25.

Lemma 7.4.13. *Half factorization of function symbols is a solution preserving rule*

Figure 7.26: Half factorization of a projection.



Proof. Half factorization begins with a graph $\mathcal{G}_1$ and produces a dependency graph $\mathcal{G}_2$ in which all equations for node N now apply to node M. All the second-order terms ϕ_i associated to the edges E_i such that $\chi_N \approx_{\mathcal{G}_1} \phi_i$ are such that $\chi_M \approx_{\mathcal{G}_2} \phi_i$ trivially because we have moved all E_i so that M is their target instead of N. However, expressions ψ such that $\chi_N \approx_{\mathcal{G}_1} \psi$ because ψ corresponds to a dependant in N do not have these equations trivially. However, we note that the new edge E_M produces the equation $\chi_M \approx_{\mathcal{G}_2} \chi_N\{\pi_1^m, ..., \pi_m^m\} \cong \chi_N$, and therefore the equation systems of both graphs are equivalent.

□

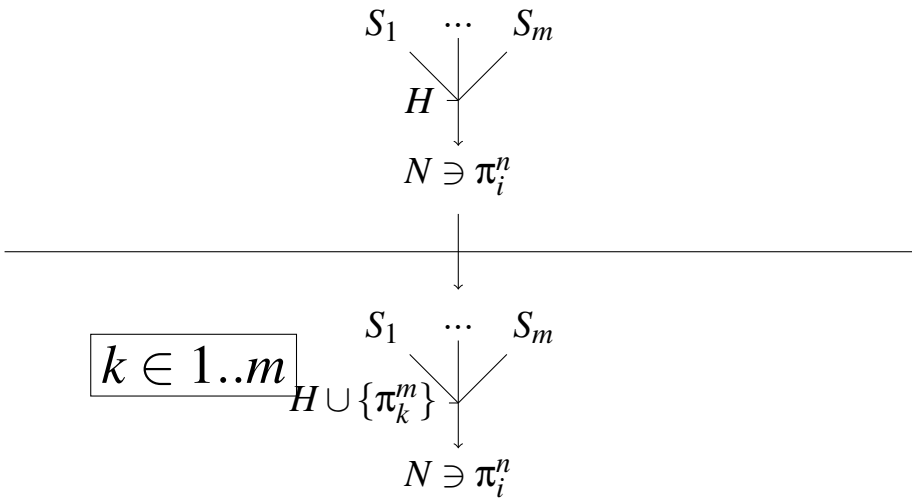
7.4.7.4 Half factorization of projections

This is a complementary rule to half factorization of function symbols that allows us to get rid of the base case in which projections have incoming edges, reducing the edge count of the graph. The underlying principle for this rule is that compositions can only reduce to projections when both the head and every source in the composition are equivalent to projections. For example, if $\pi_2^3 \approx F\{G, H\}$ then there are two possibilities for unification solutions satisfying this equation:

- $F \approx \pi_1^2$ and $G \approx \pi_2^3$.
- $F \approx \pi_2^2$ and $H \approx \pi_2^3$.

See figure 7.26 for a graphical depiction of this.

Figure 7.27: Half factorization of projections. General case.
If a projection has incoming horizontal edges, the head must necessarily be a projection.



Formally, it is enough to consider each projection that F may correspond to, since the merging of the sources with the target will occur naturally through projection simplification.

Rewrite rule 12 (Half factorization of projections)**.** Half factorization of projections is applicable when:

- The dependency graph is factorizable.
- A second-order node N contains a dependant of the form π_i^n.
- N has a non-redundant incoming horizontal edge E with head H, with arity m, and sources S_j, for j from 1 to m.

and does the following:

- For each k from 1 to m, produce a non-deterministic branch where H is merged with the result of grabbing the dependant π_k^m.

You can see a graphical depiction of this in figure 7.27.

Lemma 7.4.14. *Half factorization of projections is a solution preserving rule*

Proof. Half factorization of projections begins with a graph $\mathcal{G}_1$ and produces a set of dependency graphs $\mathbb{G}_2$.

Each graph in $\mathbb{G}_2$ corresponds to the merging of H with a projection π_k^m. Call this graph $\mathcal{G}_2^k$. We will use lemma 7.1.1. The equations for $\mathcal{G}_1$ are the same as those for each $\mathcal{G}_2^k$, except each $\mathcal{G}_2^k$ includes the additional equation $\chi_H \approx \pi_k^m$. Therefore, it is true that every solution to a graph in $\mathbb{G}_2$ is a solution to $\mathcal{G}_1$. We will show that for each solution U in $\mathcal{U}(\mathcal{G}_1)$, there is a k for which $\chi_H \approx_U \pi_k^m$.

The edge E in $\mathcal{G}_1$ produces the equation $\pi_i^n \approx \chi_H\{\chi_{S_1}, ..., \chi_{S_m}\}$. But π_i^n is a normal form, and therefore this means that in the solution U, the right side of this equation reduces to π_i^n. It is therefore enough to show that if a composition reduces to a projection, the head must be equivalent to a projection. Since we are considering all k from 1 to m, which projection the head reduces to is irrelevant.

Consider the expression

$$U(\chi_H\{\chi_{S_1}, ..., \chi_{S_m}\}) \cong U(\chi_H)\{U(\chi_{S_1}), ..., U(\chi_{S_m})\}$$

Because the rewrite system for second-order terms is confluent, we can conclude that it is true that

$$U(\chi_H)\{U(\chi_{S_1}), ..., U(\chi_{S_m})\} \xrightarrow{*} \pi_i^n$$

Consider the rewrite rules applicable to the left side of this equation:

- Head simplification. Then the head $U(\chi_H) \xrightarrow{*} \pi_i^n$, which is a projection and we have finished.
- Projection simplification. By definition the head $U(\chi_H)$ is a projection and we have finished.
- Function dumping. Then the head $U(\chi_H) \equiv \phi_0\{\phi_1, ..., \phi_p\}$ and we have that

  $$\begin{aligned} &U(\chi_H)\{U(\chi_{S_1}), ..., U(\chi_{S_m})\} \cong \\ &\phi_0\{\phi_1, ..., \phi_p\}\{U(\chi_{S_1}), ..., U(\chi_{S_m})\} \xrightarrow{*} \\ &\phi_0\{\phi_1\{U(\chi_{S_1}), ..., U(\chi_{S_m})\}, ..., \phi_p\{U(\chi_{S_1}), ..., U(\chi_{S_m})\}\} \end{aligned}$$

  But then we can recursively assume that ϕ_0 reduces to a projection, and then by projection simplification

  $$\begin{aligned} &\phi_0\{\phi_1\{U(\chi_{S_1}), ..., U(\chi_{S_m})\}, ..., \phi_p\{U(\chi_{S_1}), ..., U(\chi_{S_m})\}\} \xrightarrow{*} \\ &\phi_l\{U(\chi_{S_1}), ..., U(\chi_{S_m})\} \end{aligned}$$

But again we can recursively assume that ϕ_l reduces to a projection by the same reasons, and then we have that $U(\chi_H) \equiv \phi_0\{\phi_1, ..., \phi_p\} \xrightarrow{*} \phi_l$, which is a projection and we have finished.

- Recursive reduction on the head or the arguments do not change the composition structure of the expression and therefore we can assume, without loss of generality, that the reduced head will reduce to a projection, which means that $U(\chi_H)$ reduces to a projection.

Therefore, in every solution of $\mathcal{G}_1$, χ_H must be equivalent to a projection, and therefore the solution is a solution to a graph in $\mathbb{G}_2$.

Thus, the solutions of $\mathcal{G}_1$ are the same as those of $\mathbb{G}_2$ and therefore half factorization of projections is a solution preserving rule. □

7.4.7.5 Multiple factorization

Multiple factorization is applicable when all incoming edges to a node have (only) variables as heads. This is analogous to a flex-flex pair in higher-order unification. The possibilities are so many that the way we proceed is to non-deterministically partially instantiate one of the variables to projections or compositions with constant heads. This introduces an exponential amount of non-determinism with base dependent on the size of the signature, and therefore we try to avoid it as much as possible. For example, if $F \approx G\{g\}$ and $F \approx H\{h\}$, then one potential branch is doing $G \approx f\{G_1\}$ and proceeding. See figure 7.28 for a graphical depicition of this.

This rule has multiple critical differences with other rules.

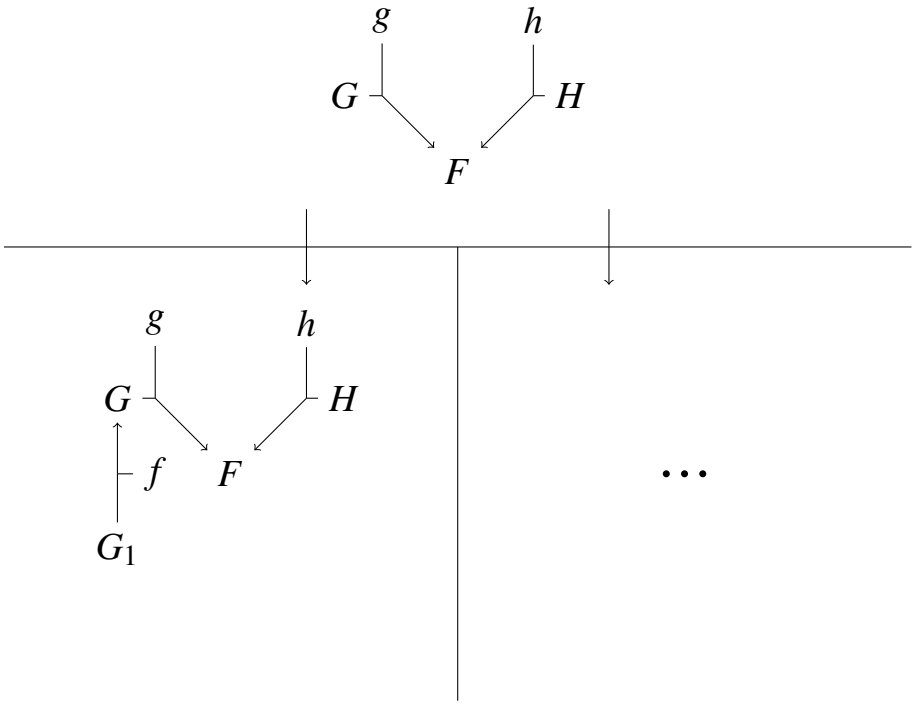
- It depends heavily on the signature, in a sense not being purely diagramatic. It depends on what potential function symbols the second-order variables may be replaced with, irrespective of the context of the variable.

- It produces a very large amount of non-deterministic branching. The process can always be applied again after applying it once, and the branching factor increases with the number of variables and the number of functions in the signature.

Rewrite rule 13 (Multiple factorization)**.** Multiple factorization is applicable when:

- The dependency graph is factorizable.

Figure 7.28: Example of multiple factorization.



- A node N has several non-redundant incoming horizontal edges $\{E_i\}$.
- Each head H_i of E_i contains only variable dependants.

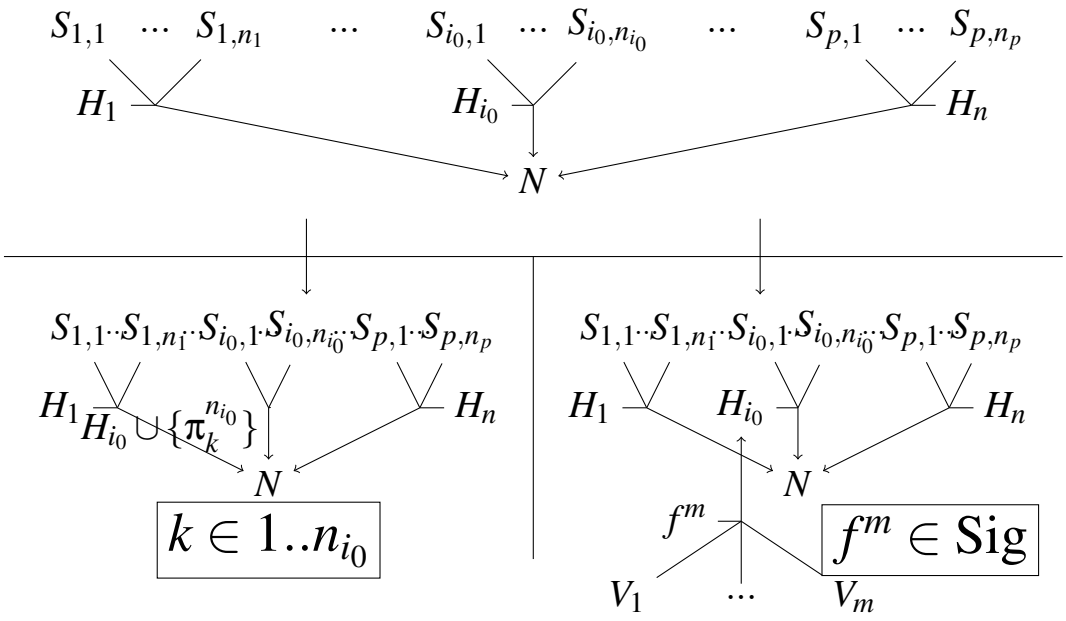
and does the following:

- Pick an arbitrary H_{i_0}.
- Let n_{i_0} be the arity of H_{i_0}. Produce the following non-deterministic branches:
  - For each k from 1 to n_{i_0}, produce a branch in which we merge node H_{i_0} with the result of grabbing $\pi_k^{n_{i_0}}$.
  - For each function symbol f^m in the signature (with arity m), add m new second-order variables V_1 to V_m, each with arity n_{i_0}, add them to the graph, and add a second-order edge with head f^m, sources $\{V_1, ..., V_m\}$ and target H_i.

You can see a graphical depiction of this in figure 7.29.

Lemma 7.4.15. *Multiple factorization is a solution preserving rule.*

Figure 7.29: Multiple factorization. General case.
The variable head may be instantiated to either a projection or a composition with a function symbol as head.



Proof. We begin with a graph $\mathcal{G}_1$ and produce a set of dependency graphs $\mathbb{G}_2$. We will use lemma 7.1.1. Call the graph result of merging H_{i_0} with $\pi_k^{n_{i_0}}$, $\mathcal{G}_2^k$; and the result of partially instantiating H_{i_0} to a composition with head f^m, $\mathcal{G}_2^{f^m}$. These conform the entirety of $\mathbb{G}_2$.

Note that each graph in $\mathbb{G}_2$ is strictly more constrained than $\mathcal{G}_1$, since they only add new equations to their unification systems, and therefore every solution of $\mathbb{G}_2$ is trivially a solution to $\mathcal{G}_1$.

In the other direction, consider a solution $U \in \mathcal{U}(\mathcal{G}_1)$. We will show that there is at least one $\mathcal{G}_2 \in \mathbb{G}_2$ for which $U \in \mathcal{U}(\mathcal{G}_2)$. Consider the second-order term $U(\chi_{H_{i_0}})$. It has an equivalent normal form $\mathcal{N}(U(\chi_{H_{i_0}}))$. By definition of second-order normal form, this normal form may be either a projection, a function symbol or a composition with a function symbol as head. Consider each of these cases:

- $U(\chi_{H_{i_0}}) \cong f^m$ for function symbol f^m with arity m. Consider the graph $\mathcal{G}_2^{f^m}$. In this graph, the only equation added with respect to $\mathcal{G}_1$ is the equation $\chi_{H_{i_0}} \approx f^m\{V_1, ..., V_m\}$. But the second-order variables V_j have been just added to the graph and therefore have no additional equations associated with them, and they

were not present in $\mathcal{G}_1$. Therefore, we may, without loss of generality, assume U to have $U(V_j) \equiv \pi_j^m$, and then $U(\chi_{H_{i_0}}) \cong U(f^m\{V_1,...,V_m\}) \cong f^m\{\pi_1^m,...,\pi_m^m\} \cong f^m$. Therefore, U is a solution to $\mathcal{G}_2^{f^m}$.

- $U(\chi_{H_{i_0}}) \cong \pi_k^{n_{i_0}}$, for some k. But the graph $\mathcal{G}_2^k$ corresponds exactly to the dependency graph $\mathcal{G}_1$ with added equation $\chi_{H_{i_0}} \approx \pi_k^{n_{i_0}}$, and therefore this graph covers this solution.

- $U(\chi_{H_{i_0}}) \cong f^m\{\phi_1,...,\phi_m\}$ for function symbol f^m with arity m. Consider the graph $\mathcal{G}_2^{f^m}$. Analogously to the constant function symbol case, the only equation added in this graph is the equation $\chi_{H_{i_0}} \approx f^m\{V_1,...,V_m\}$, where the second-order variables V_j have no additional equations and were not present in $\mathcal{G}_1$. Thus, we can, without loss of generality, assume U to have $U(V_j) \equiv \phi_j$, and then $U(\chi_{H_{i_0}}) \approx U(f^m\{V_1,...,V_m\}) \cong f^m\{\phi_1,...,\phi_m\}$. Therefore, U is a solution to $\mathcal{G}_2^{f^m}$.

Therefore, every solution to $\mathcal{G}_1$ is a solution to $\mathbb{G}_2$, and so the rule is solution preserving.

□

7.5 Normalization and rewrite rules

The results related to solution preservation were presented individually for each rewrite rule, because they can be phrased and proven in a local way, for each individual rule independently of the others. We now wish to show results that are global and are associated with sets of rules rather than individual rules. The normalization levels introduced before will help us describe these groups and their properties. We will group the rules in six groups, related to the normalization levels that they move the graph towards. Note, however, that normally after the application of a rule in a latter group, we have to re-apply rules from previous groups to go back to the normalization level we were at. In other words, some of these groups contain each other. In §7.6 we show results relating to termination and fairness of this process.

Definition 7.5.1. *We call the set of rules on dependency graphs formed by vertical monotony (of explicit equivalences (rule 1), of syntactic equivalences (rule 2) and of horizontal edges (rule 3)), edge zipping (rule 4) and projection simplification (rule 5), the set of* prenormalizing rules*. We write $\mathcal{R}^0$ to represent this set of rules.*

Lemma 7.5.1. *A dependency graph $\mathcal{G}$ is prenormal (definition 7.3.1) if and only if no prenormalizing rule is applicable to it.*

Proof. We will first prove that if it is prenormal, then no prenormalizing rule is applicable.

- If vertical monotony of explicit equivalences were applicable, then there are two vertical edges V_1 and V_2 with the same source node and unifier variable σ_i. But this directly violates the definition of prenormal graph.
- If vertical monotony of syntactic equivalences were applicable, then there are two dependants $\sigma_i\varepsilon_1$ and $\sigma_i\varepsilon_2$ not in the same node. Dependants ε_1 and ε_2 are in node N. Because of vertical alignment (which is applied whenever a new dependant is added to the node and is not a rewrite rule), we know that there is a vertical edge from N to a target node T, whose unifier variable is σ_i. But since the graph is prenormal, then we know that dependants $\sigma_i\varepsilon_1$ and $\sigma_i\varepsilon_2$ are in T. So the preconditions do not hold.
- If vertical monotony of horizontal edges were applicable, then there would be a vertical edge V with source N, and a horizontal edge H with source or target N, and $H[V]$ is not in the graph. But by definition of prenormal graph, this situation may explicitly not happen. So vertical monotony of horizontal edges is not applicable.
- If edge zipping were applicable, then there would be two non-redundant horizontal edges E_1 and E_2 with the same head and sources, but by definition of prenormal dependency graph this may not happen.
- If projection simplification were applicable, then there would be a non-redundant horizontal edge with a head that contains a projection, but if the graph is prenormal then the head of every non-redundant horizontal edge must contain only second-order variables or function symbols, so projection simplification is not applicable.

We will now show that if the graph is not prenormal, then there must be an applicable prenormalizing rule. Consider the potential reasons for which the graph may not be prenormal:

- There is a node with two outgoing vertical edges with the same unifier variable. But these are exactly the preconditions of the vertical monotony of explicit equivalences rule.

- There is a node containing a dependant of the form $\sigma_i\varepsilon$ but there is no incoming vertical edge whose source contains ε. But this violates vertical alignment, which is effected every time a dependant is added to the graph, so it is not possible.
- There is a vertical edge V with source S, target T and unifier variable σ_i, a dependant ε in S, but dependant $\sigma_i\varepsilon$ is not in T. But if V has been added to the graph, it means there must be another dependant ε_2 in S such that $\sigma_i\varepsilon_2$ is in T. But then vertical monotony of syntactic equivalences is applicable.
- There is a vertical edge V with source S, target T and a horizontal edge H whose source or target is S, but $H[V]$ is not in the graph. Also, assume, without loss of generality, that vertical monotony of explicit equivalences and vertical monotony of syntactic equivalences are not applicable. Consider the head F of H. If it contains a projection, projection dumping is applicable. Otherwise, if it has incoming horizontal edges, then function dumping is applicable on H. Otherwise, vertical monotony of horizontal edges is applicable.
- There are two non-redundant horizontal edges with the same sequence of sources and head. But then edge zipping is applicable.
- There is a non-redundant horizontal edge with a head that does not contain only second-order variables or function symbols. Since second-order nodes may only contain second-order dependants, then it must contain a projection. But then projection simplification is applicable.

Thus, prenormalization is equivalent to non-applicability of prenormalizing rules.

□

Definition 7.5.2. *We call the singleton set of rules on dependency graphs consisting only of the occurs check (rule 6), the set of* decycling rules. *We write $\mathcal{R}^1$ to represent this set of rules.*

Note that $\mathcal{R}^1 \not\supseteq \mathcal{R}^0$. This is an exception because of acyclicity being a different kind of normalization level that can be defined independently of prenormalization.

Lemma 7.5.2. *A prenormal dependency graph $\mathcal{G}$ is acyclic (definition 7.3.2) if and only if no decycling rule is applicable to it.*

Proof. The occurs check is defined precisely to be applicable whenever the graph is prenormal and cyclic. Thus, if the graph is prenormal and acyclic, the occurs check is

not applicable to it; and conversely, if the graph is prenormal and the occurs check is not applicable to it, then the graph cannot be cyclic. □

Definition 7.5.3. *We call the set of rules on dependency graphs formed by prenormalizing rules, decycling rules, function dumping (rule 7) and the validate consistency rule (rule 8), the set of* prefactorizing rules. *We write $\mathcal{R}^2$ to represent this set of rules.*

Lemma 7.5.3. *A dependency graph $\mathcal{G}$ is factorizable (definition 7.3.4) if and only if no prefactorizing rule is applicable to it.*

Proof. We first show that if the graph is factorizable, then no prefactorizing rule is applicable to it. By definition, if the graph is factorizable, then it is prenormal and acyclic, and so by lemmas 7.5.1 and 7.5.2, no prenormalizing or decycling rules are applicable, so we only have to prove that function dumping and the validate consistency rule are not applicable.

If function dumping were applicable, then there is a non-redundant second-order horizontal edge E^2 with target H and a non-redundant (first or second-order) horizontal edge E with head H. But if the graph is factorizable, then the head of every non-redundant horizontal edge has no incoming horizontal edges. H is the head of a horizontal edge with incoming horizontal edges, and so it violates this rule. So function dumping is not applicable.

If validate consistency were applicable, then there would either:

- Be a second-order node with different non-variable dependants.
- Be a second-order node whose recursive arity is not well defined.
- Be a non-redundant horizontal edge for which the recursive arity of itts head is not equal to the number of sources of the edge.

But all of these are explicitly conditions of factorizability, so the rule is not applicable.

Now for the other direction, assume no prefactorizing rules are applicable, and show that the graph must be factorizable. First, by lemmas 7.5.1 and 7.5.2, if no prenormalizing or decycling rules are applicable, then the graph must be prenormal and acyclic.

Consider each of the four other conditions of factorizability that may fail:

- There is a non-redundant horizontal edge whose head has incoming horizontal edges. But the graph is prenormal, and so the head of all horizontal edges do not contain projections. Thus, all preconditions of the function dumping rule hold and so function dumping is applicable.
- There is a second-order node that contains two different non-variable dependants. But then the validate consistency rule would be applicable.
- There is a second-order node whose recursive arity is not well defined. But then the validate consistency rule would be applicable.
- There is a non-redundant horizontal edge whose head's recursive arity is not equal to the number of sources of the edge. But then the validate consistency rule would be applicable.

And thus, the graph must be factorizable.

□

Definition 7.5.4. *Define the set of* seminormalizing rules *to be the set of prefactorizing rules and zero factorization (rule 9). Write* $\mathcal{R}^3$ *to represent this set of rules.*

The occurs check is a problematic rule. This has been discussed in §7.3, §7.4.4, and will be further discussed in §7.6.1. It needs to be applied to ensure acyclicity before a graph can be considered factorizable (and thus seminormal), but it is the only seminormalizing rule that does not fulfill certain key properties. Thus, we will define a special terminology to refer to all seminormalizing rules other than the occurs check.

Definition 7.5.5. *Define the set of* acyclic seminormalizing rules *to be the set of seminormalizing rules,* **except the occurs check**. *Write* $\mathcal{R}^{3^*}$ *to represent this set of rules.*

Lemma 7.5.4. *A dependency graph* $\mathcal{G}$ *is seminormal (definition 7.3.5) if and only if no seminormalizing rule is applicable to it.*

Proof. We will first prove that if it is seminormal, then no seminormalizing rule is applicable to it. First note that by definition of seminormal, $\mathcal{G}$ is factorizable, and therefore, by lemma 7.5.3, no prefactorizing rule is applicable to it. Thus, we only need to show that zero factorization is not applicable to it either.

But by definition of seminormal, for every node that contains more than one non-redundant incoming horizontal edge, there is one such edge whose head only

contains variable dependants, so zero factorization is not applicable to any of those nodes.

In the other direction, if no seminormalizing rule is applicable to it, then in particular, no prefactorizing rule is applicable to it. Thus, by lemma 7.5.3, the graph must be factorizable. If the graph were not seminormal, it would be because there is a node N with more than one non-redundant incoming horizontal edge, such that each head of such edges contains at least one non-variable dependant. But that is exactly the pre-condition for zero factorization to be applicable, and thus the graph must be seminormal.

Therefore, seminormality is equivalent to non-applicability of seminormalizing rules.

□

Definition 7.5.6. *Define the set of* quasinormalizing rules *to be the set of seminormalizing rules, single factorization (rule 10), half factorization of function symbols (rule 11) and half factorization of projections (rule 12). Write $\mathcal{R}^4$ to represent this set of rules.*

Lemma 7.5.5. *A dependency graph $\mathcal{G}$ is quasinormal (definition 7.3.6) if and only if no quasinormalizing rule is applicable to it.*

Proof. We will first show that if $\mathcal{G}$ is quasinormal then no quasinormalizing rule is applicable to it. First note that by definition of quasinormal, the graph is seminormal. Thus, by lemma 7.5.4, no seminormalizing rule is applicable. Therefore, we only have left to prove that neither single factorization, half factorization of function symbols or half factorization of projections is applicable.

If the graph is quasinormal, if a node has more than one non-redundant incoming horizontal edge, all their heads must contain only variable dependants. But for single factorization to be applicable, we would need to have a node with at least two incoming horizontal edges, one of which contains at least one non-variable dependant. So single factorization is not applicable.

If the graph is quasinormal, then all second-order nodes with non-redundant incoming horizontal edges contain only variables. But then half factorization of function symbols is not applicable because there must be a second-order node with non-redundant incoming horizontal edges and containing a function symbol. Similarly, half factorization of projections is not applicable because there would need to be a second-order node with non-redundant incoming horizontal edges containing a projection.

Thus, if $\mathcal{G}$ is quasinormal then no quasinormalizing rule is applicable to it.

In the other direction, assume no quasinormalizing rule is applicable. But, in particular, no seminormalizing rule is applicable. Thus, by lemma 7.5.4, the graph is seminormal. Consider the reasons for which the graph may not be quasinormal:

- There is a node N with more than one non-redundant incoming horizontal edge, and one of such heads contains at least one non-variable dependant. Because the graph has already been shown to be seminormal, we know there is at least one other of said edges whose head contains only variable dependants. But then the conditions of single factorization are all met and thus the rule is applicable, which is a contradiction. So this condition of quasinormalization must be met in $\mathcal{G}$.
- There is a second-order node with non-redundant incoming horizontal edges containing a function symbol. But the graph has already been shown to be seminormal (and thus factorizable), so the conditions of half factorization of function symbols are all met, so thus the rule is applicable, which is a contradiction. So this condition of quasinormalization must be met in $\mathcal{G}$.
- There is a second-order node with non-redundant incoming horizontal edges containing a projection. But the graph has already been shown to be seminormal (and thus factorizable), so the conditions of half factorization of projections are all met, so thus the rule is applicable, which is a contradiction. So this condition of quasinormalization must be met in $\mathcal{G}$.

So the graph is quasinormal.

Therefore, quasinormality is equivalent to non-applicability of quasinormalizing rules.

□

Definition 7.5.7. *Define the set of* normalizing rules *to be the set of quasinormalizing rules and multiple factorization (rule 13). Write $\mathcal{R}^5$ to represent this set of rules.*

We note that normalizing rules are all the rules that we have defined on dependency graphs.

Lemma 7.5.6. *A dependency graph $\mathcal{G}$ is normal (definition 7.3.7) if and only if no normalizing rule is applicable to it.*

Proof. We first prove that if the graph is normal then no normalizing rule is applicable to it. By definition of normality, the graph is also quasinormal, so by lemma 7.5.5, no quasinormalizing rule is applicable. Therefore, we only need to show that multiple factorization is not applicable. But if the graph is normal, then there is no node with more than one non-redundant incoming horizontal edge, which makes multiple factorization not applicable.

In the other direction, if no normalizing rule is applicable, in particular, no quasinormalizing rule is applicable, and therefore, by lemma 7.5.5, the graph is quasinormal. We only need to show that there may be no node with multiple non-redundant incoming horizontal edges. But for each node with multiple non-redundant incoming horizontal edges, there may only be three possibilities:

- Each head of such edges contains one non-variable dependant. Then, zero factorization is applicable.
- There is at least one head that contains one non-variable dependant and at least one head that contains only variable dependants. Then, single factorization is applicable.
- All heads contain only variable dependants. Then, multiple factorization is applicable.

Thus, the graph must be normal.

And therefore we have shown that normality is equivalent to non-applicability of normalizing rules.

□

7.6 Termination, productivity, fairness and solution shape verification

In this section we present results related to the running time of algorithms that apply the rewrite rules on dependency graphs, including but not limited to termination properties. These properties will be different depending on the set of rewrite rules we consider, and we will need some auxiliary definitions.

These results are all novel, as unification dependency graphs are. However, the concepts of termination, enumeration, productivity and fairness used here are the standard ones in computability theory (see §3.5).

First we need an upper boundary to the set of dependants that may be relevant in a dependency graph. Rules do not increase the number of unifier levels appearing in a dependency graph.

Theorem 7.6.1 (First-order dependant boundary on dependency graphs)**.** *Consider any dependency graph $\mathcal{G}$. Consider the maximum unifier level (definition 6.1.19) explicitly appearing on first-order dependants in nodes in the graph, i_M. For any possible application of a rule $R \in \mathcal{R}^5$, let $\mathcal{G}_2$ be the result of applying the rule to $\mathcal{G}$.*

Then, $\mathcal{G}_2$ contains no first-order dependants with a unifier level $j > i_M$.

Proof. We will prove it rule by rule:

- **Vertical monotony of explicit equivalences, vertical monotony of syntactic equivalences, edge zipping, projection simplification, function dumping, validate consistency, zero factorization, single factorization** - None of these rules can add new dependants and therefore the result is trivial.
- **Vertical monotony of horizontal edges** - In order to introduce the edge $H[V]$ we may need to grab new dependants, but all of them are with unifier level i, where there was a vertical edge V with unifier level i, and therefore the target of V already had level i. Therefore, $i_M \geq i$ and thus no new dependant with unifier level $j > i_M$ is added.
- **Occurs check, half factorization of function symbols, half factorization of projections, multiple factorization** - These rules may only add second-order dependants to the graph, and therefore the result is trivial.

□

Corollary 7.6.1. *Let $\mathcal{G}_1$ be a dependency graph. There is a* **finite** *set of first-order dependants $\mathcal{D}^1(\mathcal{G}_1)$ such that after any number of applications of rewrite rules in $\mathcal{R}^3$ on $\mathcal{G}_1$, every first-order dependant in the resulting graph $\mathcal{G}_2$ is in $\mathcal{D}^1(\mathcal{G}_1)$.*

Proof. By the theorem, no new unifier levels may be introduced. No first-order variables are ever introduced by rules in $\mathcal{R}^3$. We will recursively show that the number of first-order dependants with a certain unifier level is finite. Let i be a unifier level. Consider the possible dependants with that unifier level in a fixed signature:

- If $i = \perp$, the only unifier expressions in this level do not contain unifier variables or first-order variables. But dependants, by definition, always contain at least one first-order variable. That is, there can be no dependants on this unifier level in the graph (the unifier expressions in this level are expressed entirely through edges between anonymous nodes).
- If $i = 0$, then the dependants must be first-order variables. Because the set of first-order variables in the signature is finite and invariant, this number is finite.
- For $i > 0$, the dependants must be of the form $\sigma_i\delta$, where δ is a dependant with unifier level $j < i$. We may recursively assume that there is thus a finite number of possible dependants δ, and therefore there is also a finite numberof possible dependants $\sigma_i\delta$ for unifier level i.

Therefore, for each unifier level the number of dependants that may exist at that level is finite, and rewrite rules may not introduce new unifier levels. Thus, the total set of dependants that may ever appear in the graph after any number of applications of rewrite rules is bounded from above by a finite set.

□

Theorem 7.6.2. *Let $\mathcal{G}_1$ be a dependency graph. There is a* **finite** *set of second-order dependants $\mathcal{D}^2(\mathcal{G}_1)$ such that after any number of applications of rewrite rules in $\mathcal{R}^{3^*}$, other than the occurs check, on $\mathcal{G}_1$, every second-order dependant in the resulting graph $\mathcal{G}_2$ is in $\mathcal{D}^2(\mathcal{G}_1)$.*

Proof. First, we note that while the occurs check and multiple factorization may introduce new second-order variables, these rules are not in $\mathcal{R}^{3^*}$. Thus, the set of second-order variables in the signature remains invariant when applying rules in $\mathcal{R}^{3^*}$ other than the occurs check.

Second, consider the maximum arity amongst function symbols and second-order variables in the signature. Because this is a finite set, there is a maximum arity m amongst these. Second-order dependants may be function symbols, which are finite as defined in the signature, second-order variables, which are finite as defined and invariant through rules in $\mathcal{R}^{3^*}$, and projections. Projections are in principle infinite, but any rule in $\mathcal{R}^{3^*}$ that introduces projections into the graph introduces projections with arity equal to the arity of a node previously existing in the graph. Thus, the maximum arity amongst projections remains less than or equal to m, and consequently, the set of projections that a rule in $\mathcal{R}^{3^*}$ may introduce into the graph after any number of applications is finite.

□

7.6.1 The issue with cycles

In §7.3, we already discussed the inconvenience of both cyclic graphs and the method to make them acyclic. To rehearse, cycles must be eliminated from the graph before function dumping and validate consistency can be applied, but in order to do so, we need to apply the *occurs check* rule (§7.4.4), which has two large issues:

- It is of intrinsically *global* nature, potentially requiring checking large portions of the graph.
- It may introduce non-determinism.

There is another added potential issue with the occurs check rule, that is however related to the non-determinism issue: **the occurs check rule may potentially cause reduction chains that never terminate**. We conjecture that, in fact, this does not happen. We conjecture that the occurs check rule may only be applied a finite number of times to any given dependency graph, but we have not been able to prove this. We have not been able to find any instance of a dependency graph in which the occurs check rule may be applied an infinite number of times, but we also have not been able to prove that one does not exist.

Pragmatically, however, because the occurs check is already problematic even if it terminates, the veracity of this conjecture does not fundamentally change the fact that cycles are an issue.

Thus, we will prove termination results for prenormal graphs, and then we will extend these to seminormal graphs, **under the condition that no cyclic graph is ever produced while applying seminormalizing rewrite rules**. In other words, we will show that we can extend termination results to acyclic seminormalizing rules.

We first begin by strictly proving the result for prenormal graphs.

7.6.2 Prenormalizing rules: Termination

Let us consider the set of prenormalizing rules, $\mathcal{R}^0$. The application of these rules is *terminating*: for any dependency graph, and regardless of the order in which they are

applied, only a finite number of these rules will be applicable. To show this, we will define measures[15] on dependency graphs, such that each rule strictly decreases one of them, and we can combine them into a single positive measure that every rule strictly decreases.

Definition 7.6.1 (Implicit equivalent dependants measure)**.** *Define the measure $\mu^{\approx}$, called* implicit equivalent dependants measure*, on a dependency graph $\mathcal{G}$, to be the number of pairs of different first-order dependants $(\varepsilon_1, \varepsilon_2) \in \mathcal{D}^1(\mathcal{G})^2$ such that $\varepsilon_1 \approx_{\mathcal{G}} \varepsilon_2$ but ε_1 and ε_2 are on different nodes in the graph; plus the number of pairs of different second-order dependants $(\phi_1, \phi_2) \in \mathcal{D}^2(\mathcal{G})^2$ such that $\phi_1 \approx_{\mathcal{G}} \phi_2$ but ϕ_1 and ϕ_2 are on different nodes in the graph.*

This measure is positive and finite because $\mathcal{D}^1(\mathcal{G})$ and $\mathcal{D}^2(\mathcal{G})$ are finite.

Lemma 7.6.1. *Let $\mathcal{G}_1$ be a dependency graph and consider a rule $R \in \mathcal{R}^{3^*}$ applicable to $\mathcal{G}_1$. Write $\mathcal{G}_2$ for the graph result of the application of the rule.*

Then, $\mu^{\approx}(\mathcal{G}_1) \geq \mu^{\approx}(\mathcal{G}_2)$. Furthermore, for vertical monotony of explicit equivalences, vertical monotony of syntactic equivalences, edge zipping, projection simplification and zero factorization, this inequality is strict.

Proof. First, we note that because every rule in $\mathcal{R}^{3^*}$ is solution preserving, then for any two first-order dependants ε_1 and ε_2, $\varepsilon_1 \approx_{\mathcal{G}_1} \varepsilon_2$ if and only if $\varepsilon_1 \approx_{\mathcal{G}_2} \varepsilon_2$, and equivalently for second-order dependants.

Moreover, the measure $\mu^{\approx}$ is defined with respect to $\mathcal{D}^1(\mathcal{G})$ and $\mathcal{D}^2(\mathcal{G})$, which, by theorems 7.6.1 and 7.6.2, do not change when applying rewrite rules in $\mathcal{R}^{3^*}$. Thus, grabbing new dependants does not change the measure, only merging or splitting nodes would. But no rule ever splits nodes (separate dependants in the same node), so the number of equivalent pairs of first-order dependants that are not in the same node may never increase. This is enough to conclude the proof of the non-strict inequality.

Furthermore, any rule that merges nodes that were not previously the same node will strictly reduce this value. This is explicitly the case for vertical monotony of explicit equivalences, vertical monotony of syntactic equivalences, edge zipping and projection simplification. In the case of zero factorization, note that if the rule is applied and the resulting graph passes the validity check of this rule, the heads of the edges were

[15] These are not measures in the sense of measure theory, but rather, functions that measure some properties of graphs with a finite, positive number.

necessarily the same. If each source of each edge were the same nodes, then the graph would not have been prenormal and therefore the preconditions of the rule would not have applied. Thus, there is at least one source index for which the source nodes of two different incoming edges were not the same, and they have been merged, and thus $\mu^{\approx}$ is strictly reduced in this case as well.

□

Definition 7.6.2 (Implicit related first-order dependants measure)**.** *Define the measure $\mu^{\rightarrow}$, called* implicit related first-order dependants measure*, on a dependency graph $\mathcal{G}$, to be the number of combinations of a sequence of first-order dependants $\varepsilon_1^S, ...\varepsilon_n^S \in \mathcal{D}^1(\mathcal{G})$, a target dependant $\varepsilon^T \in \mathcal{D}^1(\mathcal{G})$ and a second-order dependant $\phi \in \mathcal{D}^2(\mathcal{G})$ for which $\varepsilon^T \approx_{\mathcal{G}} \phi(\varepsilon_1^S, ..., \varepsilon_n^S)$ but there is no horizontal edge with sources $\varepsilon_1^S, ..., \varepsilon_n^S$ (the result of grabbing their nodes), target ε^T and head ϕ, and it is not the case that ε^T is in the same node as one of the ε_i^S*

This measure is finite because $\mathcal{D}^1(\mathcal{G})$ and $\mathcal{D}^2(\mathcal{G})$ are finite.

Lemma 7.6.2. *Let $\mathcal{G}_1$ be a dependency graph and consider a rule $R \in \mathcal{R}^{3^*}$ applicable to $\mathcal{G}_1$. Write $\mathcal{G}_2$ for the graph result of the application of the rule.*

Then, $\mu^{\rightarrow}(\mathcal{G}_1) \geq \mu^{\rightarrow}(\mathcal{G}_2)$. Furthermore, for vertical monotony of horizontal edges, this inequality is strict.

Proof. First and foremost, note that the definition of $\mu^{\rightarrow}$ does **not** exclude redundant horizontal edges. This is, to put it bluntly, the whole reason we introduced redundant horizontal edges. Redundant edges are a way to have a witness in the graph that we have done something and we wish to not do it again, so that this measure can never increase. In particular, we use them as a witness that vertical monotony of horizontal edges was already applied on a node, when we wish to semantically remove the edge from the graph due to, mostly, function dumping (because another, semantically more precise edge was added in exchange). That is why they have no semantics and are only used in the preconditions of rules, so that loops in rule application in $\mathcal{R}^{3^*}$ are not possible.

We note that because every rule in $\mathcal{R}^{3^*}$ is solution preserving, then for any combination of first-order dependants $\varepsilon_1^S, ..., \varepsilon_n^S$, target first-order dependant ε^T and second-order dependant ϕ, $\varepsilon^T \approx_{\mathcal{G}_1} \phi(\varepsilon_1^S, ..., \varepsilon_n^S)$ if and only if $\varepsilon^T \approx_{\mathcal{G}_2} \phi(\varepsilon_1^S, ..., \varepsilon_n^S)$.

Moreover, because the measure $\mu^{\rightarrow}$ is defined with respect to $\mathcal{D}^1(\mathcal{G})$ and $\mathcal{D}^2(\mathcal{G})$, which do not change when applying rewrite rules, grabbing new dependants

does not change the measure, only adding or removing edges, or merging nodes would. But adding edges or merging nodes would decrease the measure, and removing edges only happens whenever there is an exactly equal edge (so formally the graph is entirely unchanged in that respect). Therefore, this measure never increases.

Furthermore, vertical monotony of equivalences explicitly adds a horizontal edge that was previously not in the graph, and therefore $\mu^{\rightarrow}$ strictly decreases.

□

Theorem 7.6.3 ($\mathcal{R}^0$ is terminating)**.** *Consider an arbitrary dependency graph $\mathcal{G}$. Then, the application of rules in $\mathcal{R}^0$ to $\mathcal{G}$ is terminating, regardless of the order of their application.*

Proof. Consider the lexicographic ordering among dependency graphs which orders first over the value of $\mu^{\approx}$ and over the value of $\mu^{\rightarrow}$ when this is equal. Write μ^* to describe this ordering. The measures $\mu^{\approx}$ and $\mu^{\rightarrow}$ are both finite and non-negative and so they have a minimum value, and thus μ^* has a minimum value as well.

If we can show that every rewrite rule in $\mathcal{R}^0$ strictly decreases μ^*, then this means that only a finite number of applications may be possible before this reaches a minimum value, and the proof is done. Consider each rewrite rule in $\mathcal{R}^0$:

- **Vertical monotony of explicit equivalences, vertical monotony of syntactic equivalences, edge zipping, projection simplification** - By lemma 7.6.1, $\mu^{\approx}$ strictly decreases and thus so does μ^*.
- **Vertical monotony of horizontal edges** - By lemma 7.6.1, $\mu^{\approx}$ does not increase. Moreover, by lemma 7.6.2, $\mu^{\rightarrow}$ strictly decreases and thus so does μ^*.

□

7.6.3 Seminormalizing rules: Termination under acyclicity

We now extend the results in the previous section to all acyclic seminormalizing rules.

Definition 7.6.3 (Composite heads measure)**.** *Define the measure $\mu^{\{\}}$, called* composite heads measure*, recursively on each second-order node that is the head of a non-redundant horizontal edge. For second-order node N which is the head of a horizontal*

edge, if N has no non-redundant incoming horizontal edges, then $\mu^{\{\}}(N) = 0$. If N has non-redundant incoming horizontal edges, define $\mu^{\{\}}(N)$ to be 1 plus the sum over all its non-redundant incoming horizontal edges, of $\mu^{\{\}}(H_i)$, where H_i is the head of each of such edges.

Then, define $\mu^{\{\}}$ on the graph to be the sum over all non-redundant horizontal edges, of the value of $\mu^{\{\}}$ on their heads.

This measure is well defined and finite as long as the graph is acyclic, since the number of edges in a dependency graph is always finite and the recursive process always terminates if no loops are present.

This measure is explicitly introduced to describe the way in which function dumping reduces some sense of complexity in the graph. Therefore, we are only interested in the effect function dumping has in it.

Lemma 7.6.3. *Let $\mathcal{G}_1$ be a dependency graph to which function dumping is applicable. Write $\mathcal{G}_2$ for the graph result of one application of function dumping.*

Then, $\mu^{\{\}}(\mathcal{G}_1) > \mu^{\{\}}(\mathcal{G}_2)$.

Proof. In function dumping, a horizontal edge is added with target T and head H^2, but another horizontal edge not previously redundant is marked as redundant. The edge being marked as redundant had as head H, who had a non-redundant incoming horizontal edge with head H^2, and thus its value of $\mu^{\{\}}$ was strictly larger than that of H^2, which is the head of the new edge. Therefore, the value of $\mu^{\{\}}$ has been overall strictly reduced in the graph.

□

Theorem 7.6.4 ($\mathcal{R}^{3^*}$ is terminating)**.** *Consider an arbitrary dependency graph $\mathcal{G}$. Then, the application of rules in $\mathcal{R}^{3^*}$ to $\mathcal{G}$ is terminating, regardless of the order of their application.*

Proof. Consider the lexicographic ordering among dependency graphs which orders first over the value of $\mu^{\approx}$, over the value of $\mu^{\rightarrow}$ when this is equal, and over the value of $\mu^{\{\}}$ when this is equal. Write μ^* to describe this ordering. The measures $\mu^{\approx}$, $\mu^{\rightarrow}$ and $\mu^{\{\}}$ are all finite and non-negative and so they have a minimum value, and thus μ^* has a minimum value as well.

We will first say that the validate consistency rule can, by definition, only be applied once to a graph, to invalidate it. Therefore, the application of this rule may never be

a problem for non-termination. If we can show that every other rewrite rule in $\mathcal{R}^{3^*}$ strictly decreases μ^*, then this means that only a finite number of applications may be possible before this reaches a minimum value, and the proof is done. Consider each rewrite rule in $\mathcal{R}^{3^*}$:

- **Validate consistency** - As described, this rule can at most be applied once to a graph, to invalidate it (so no more rules would be applicable), so it may not cause non-terminating reduction.
- **Vertical monotony of explicit equivalences, vertical monotony of syntactic equivalences, edge zipping, projection simplification, zero factorization** - By lemma 7.6.1, $\mu^{\approx}$ strictly decreases and thus so does μ^*.
- **Vertical monotony of horizontal edges** - By lemma 7.6.1, $\mu^{\approx}$ does not increase. Moreover, by lemma 7.6.2, $\mu^{\rightarrow}$ strictly decreases and thus so does μ^*.
- **Function dumping** - By lemma 7.6.3, $\mu^{\approx}$ does not increase. Moreover, by lemma 7.6.2, $\mu^{\rightarrow}$ does not increase either. Finally, by lemma 7.6.3, $\mu^{\{\}}$ strictly decreases and thus so does μ^*.

□

It is important to understand the scope of this result: it does not produce any guarantees on when cycles may be introduced into a graph (most seminormalizing rules have the potential ability to introduce them). It establishes that, if we are lucky enough that cycles are never produced, then the process is terminating. The moment a cycle is introduced, the occurs check rule needs to be applied, which may potentially undo the work that the other rules do, and potentially produce non-terminating reduction.

7.6.4 Normalizing rules: Fairness

Unfortunately, it is provably not the case that $\mathcal{R}^4$ or $\mathcal{R}^5$ are terminating as $\mathcal{R}^{3^*}$ is. In fact, this would be absurd since for some dependency graphs there is no finite set of unification solutions that are, in some sense, complete. The last statement makes more sense when we consider the fact, that will be shown shortly, that normal dependency graphs are almost directly related to unification solutions. This further makes it absurd to consider the *confluence* of the set of rules $\mathcal{R}^5$.

A weaker result to termination is *productivity*. This establishes that, as long as not all solutions have been found, a new solution can be produced by applying the

rewrite rules a finite number of steps. However, this is not strong enough for our purposes, and furthermore we can prove a stronger result on $\mathcal{R}^5$: *fairness*. Fairness implies productivity, but further establishes that for every solution to the dependency graph, it will be produced by applying the rewrite rules some finite number of times. In some sense, this says that the non-terminating properties of $\mathcal{R}^5$ are associated with the intrinsic infinite nature of the set of solutions, rather than with a problem in the search method itself.

However, in order to properly define any of these results, we first need to formally establish a constructive connection between dependency graphs and solutions. So far, that connection has been existential through the notion of the unification equation system associated to the graph and the set of solutions of the equaiton system. But the purpose of the rewrite rules is precisely to be able to produce these solutions constructively.

Definition 7.6.4 (General solution of a normal graph)**.** *Consider a normal dependency graph $\mathcal{G}$. Then, constructively produce the* general unification solution *of $\mathcal{G}$, written $U_0(\mathcal{G})$ in the following way:*

- *First, we use a technical tool to make sure that we have enough available first-order variables to express the solution.*

  Consider the set of first-order variables $\mathcal{V}_0$ appearing in $\mathcal{G}$. Then, for each unifier level i in $\mathcal{G}$, define the following sets:

  - $\bar{\mathcal{V}}_0 = \mathcal{V}_0$
  - *$\mathcal{V}_i$ is formed by one fresh first-order variable for each variable in $\bar{\mathcal{V}}_{i-1}$*
  - $\bar{\mathcal{V}}_i = \bar{\mathcal{V}}_{i-1} \cup \mathcal{V}_i$.

  For each first-order variable $X \in \bar{\mathcal{V}}_{i-1}$, write $\mathcal{V}_i(X)$ to be the fresh first-order variable associated with it. That is, at most, for each unifier variable we may require one new first-order variable in its "image" (before applying the substitution associated with the unifier variable) for each first-order variable in its "domain" (after applying it).

- *If a node contains no dependants and has no incoming non-redundant horizontal edges, ignore it*[16]

[16] Note that such a node has no effect whatsoever on the associated equation system to the graph and in fact is never produced through any of our rewrite rules.

- *For each other second-order node N, define the second-order term $U_0(N)$ in the following way:*

  - *If N has no non-redundant incoming horizontal edges and it contains non-variable dependants, then because the graph is factorizable, they must all be equivalent. Pick an arbitrary one ϕ and define $U_0(N) = \phi$.*
  - *If N has no non-redundant incoming horizontal edges and all dependants it contains are variable, then pick an arbitrary such variable, F. Define $U_0(N) = F$.*
  - *If N has non-redundant incoming horizontal edges, then because the graph is normal it must only have one such edge, call it E. Write H for the head of E and S_i for its sources. Then, $U_0(N) = U_0(H)\{U_0(S_i), ..., U_0(S_n)\}$. Note that because the graph is acyclic, this recursion is well defined.*

- *For each other first-order node N in $\mathcal{G}$, define the first-order term $U_0(N)$ in the following way:*

  - *If N has no non-redundant incoming horizontal nor vertical edges, then it must contain dependants. Each of those dependants ε must have either $\#_\sigma(\varepsilon) = \bot$ or $\#_\sigma(\varepsilon) = 0$. But in lemma 7.6.1 we showed that there are no dependants on unifier level $\bot$, so in this case it must be $\#_\sigma(\varepsilon) = 0$. Thus, N must contain a dependant of the form X for a first-order variable X. Then, $U_0(N) = X$, for one arbitrary variable in N.*
  - *If N has no non-redundant incoming horizontal edges but has incoming vertical edges, then it must contain at least one dependant of the form $\sigma_i\delta$ where δ has unifier level $j < i$, because vertical edges are only introduced on nodes who contain these dependants or which have incoming horizontal edges. Consider the node N_δ in which the dependant δ is in the graph. Since $\mathcal{G}$ is prenormal, δ must also not have any non-redundant incoming horizontal edges. Therefore, $U_0(N_\delta)$ can be assumed to recursively be a first-order variable in $\bar{\mathcal{V}}_j$. Then, define $U_0(N) = \mathcal{V}_i(U_0(N_\delta))$*
  - *If N has non-redundant incoming horizontal edges, then because the graph is normal it must have only one of such, call it E. Write H for the head of E and S_i for the sources of E. Then, $U_0(N) = U_0(H)(U_0(S_1), ..., U_0(S_n))$. Note that because the graph is acyclic, this recursion is well defined.*

- *For each second-order variable F appearing in the graph, let N_F be the node in which F appears. Then, define the instantiation I_0 to be such that $I_0(F) = U_0(N_F)$ as defined above.*

- *For each unifier level i and each first-order variable $X \in \mathcal{V}_0$, consider the dependant $\sigma_i X$ and the node $N_{\sigma_i X}$ in which it appears in the graph. Then, define the substitution $\sigma_i^{U_0}$ to be such that $\sigma_i^{U_0} X = U_0(N_{\sigma_i X})$ as defined above, or $\sigma_i^{U_0} X = \mathcal{V}_i(X)$ if $\sigma_i X$ is not in the graph.*

- *Finally, define the unification solution $U_0(\mathcal{G})$ to consist of the instantiation I_0 and the substitution $\sigma_i^{U_0}$ for each unifier variable σ_i.*

In short, the solution is defined as the most general solution for root nodes in the graph and the dependants they represent, and then uses the dependencies to propagate these through the graph to other dependants. This is why normal graphs are useful for defining solutions: A single incoming non-redundant horizontal edge per node uniquely defines the solution to that node.

Consider, for example, the normal graph in figure 7.30. If we apply the procedure above to extract a general solution from it, we will obtain the following (write $X_{\sigma_1} = \mathcal{V}_1(X)$ to represent the fresh variable associated with X at unifier level 1, and similarly for $Y_{\sigma_1} = \mathcal{V}_1(Y)$, $X_{\sigma_2} = \mathcal{V}_2(X)$, $X_{\sigma_2\sigma_1} = \mathcal{V}_2(X_{\sigma_1})$, $Y_{\sigma_2} = \mathcal{V}_2(Y)$, $Y_{\sigma_2\sigma_1} = \mathcal{V}_2(Y_{\sigma_1})$):

$$
\begin{aligned}
&U_0(F_1) = f\{g, \pi_1^2\} \\
&U_0(F_2) = g
\end{aligned}
$$

$$
\begin{aligned}
&\sigma_1^{U_0} X = X_{\sigma_1} \\
&\sigma_1^{U_0} Y = g(X_{\sigma_1}, h())
\end{aligned}
$$

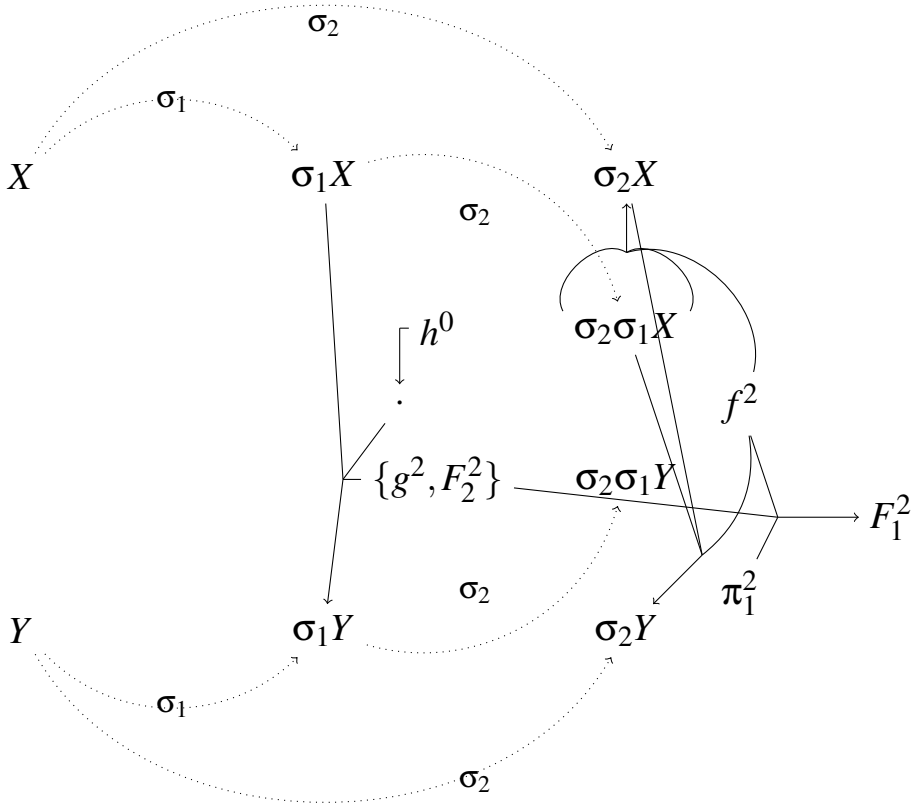
$$
\begin{aligned}
&\sigma_2^{U_0} X = f(X_{\sigma_2\sigma_1}, X_{\sigma_2\sigma_1}) \\
&\sigma_2^{U_0} Y = f(X_{\sigma_2\sigma_1}, f(X_{\sigma_2\sigma_1}, X_{\sigma_2\sigma_1}))
\end{aligned}
$$

Understood as a solution to a unification problem (with two unifiers/substitutions σ_1 and σ_2) and using conventional first-order unification notation, what this means is that:

- It provides ground instantiations of F_1 and F_2, the only ones which would satisfy the graph.

- The substitution σ_1 must be such that $Y \sim_{\sigma_1} g(X, h())$.

Figure 7.30: A normal dependency graph, from which we can extract a solution.



- The substitution σ_2 must be such that $X \sim_{\sigma_2} f(X_{\sigma_2\sigma_1}, X_{\sigma_2\sigma_1})$ and $Y \sim_{\sigma_2} f(X_{\sigma_2\sigma_1}, X)$.

Theorem 7.6.5 (Generality of the general solution of a normal graph)**.** *Let $\mathcal{G}$ be a normal dependency graph whose consistency has been validated. Let U be any unification solution.*

Then, U is a unification solution to $\mathcal{G}$ if and only if U is finer (definition 6.1.23) than $U_0(\mathcal{G})$.

Proof. Note that U is finer than $U_0(\mathcal{G})$ by definition if we can compose the instantiation and each substitution of $U_0(\mathcal{G})$ on the left to produce U. Also note that U is a solution to $\mathcal{G}$ by definition if it satisfies every equation in the unification equation system associated to $\mathcal{G}$.

First, let us show that for every unification solution U to $\mathcal{G}$, U is finer than $U_0(\mathcal{G})$. The following holds true for every solution U to $\mathcal{G}$:

- For each second-order variable F, let N_F be the node in which the dependant F is. Then, because the graph is normal, exactly one of the following possibilities hold:

  - N_F has no non-redundant incoming horizontal edges and contains no non-variable dependants. Then $U_0(N_F) = G$, for G a second-order variable in N_F. But for any solution U, the equation $F \approx_U G$ holds, and thus whichever the instantiation of U is, it must be the same for F and G, and this instantiation is finer than U_0, which merely implies the equality of these two variables.
  - N_F has no non-redundant incoming horizontal edges but contains exactly one non-variable dependant (it may not contain more than one because it is a prenormal graph) ϕ. Then $U_0(N_F) = \phi$. But for any solution U, the equation $F \approx_U \phi$ holds, and thus whichever the instantiation of U is, it must be such that $U(F) = \phi$. Thus, this instantiation is finer than U_0.
  - N_F has an incoming horizontal edge E with head H and sources S_i. Then $U_0(N_F) = U_0(H)\{U_0(S_1), ..., U_0(S_n)\}$. Because the graph is acyclic, we may recursively assume that for nodes H and S_i, the instantiation of U is finer than U_0. Moreover, the graph implies the equation $F \approx_U \chi_H\{\chi_{S_1}, ..., \chi_{S_n}\}$. These two facts together imply that, for any U, its instantiation is finer than the instantiation $U_0(F) = U_0(\chi_H)\{U_0(\chi_{S_1}), ..., U_0(\chi_{S_n})\}$.

- For each unifier variable σ_i and each first-order variable X, let $N_{\sigma_i X}$ be the node in which the dependant $\sigma_i X$ is. Then, because the graph is normal, exactly one of the following possibilities hold:

  - $N_{\sigma_i X}$ has no non-redundant incoming horizontal edges. It does, however, contain at least one incoming vertical edge from a node without incoming horizontal edges. Thus, $U_0(N_{\sigma_i X}) = Y$, where Y is a fresh first-order variable thus not assigned to any other node in the graph. This literally means that the substitution σ_i is not constrained at all for its value on X. Note as well that for every other dependant δ in $N_{\sigma_i X}$, the equation $\delta \approx_U \sigma_i X$ holds, and therefore, whatever the substitution σ_i^U is, it will be finer than U_0 for $\sigma_i X$.
  - $N_{\sigma_i X}$ has exactly one non-redundant incoming horizontal edge E with head H and sources S_i. Then, $U_0(N_{\sigma_i X}) = U_0(H)(U_0(S_1), ..., U_0(S_n))$. Because the graph is acyclic, we may recursively assume that for nodes H and

S_i, the substitutions and instantiation of U is finer than U_0. Moreover, the graph implies the equation $\sigma_i X \approx_U \chi_H(\kappa_{S_1}, ..., \kappa_{S_n})$. These two facts together imply that, for any U, its instantiation is finer than the instantiation $U_0(\sigma_i X) = U_0(\chi_H)(U_0(\kappa_{S_1}), ..., U_0(\kappa_{S_n}))$.

Now in the other direction, consider a solution U finer than U_0 and show that it must be a solution to $\mathcal{G}$.

- Consider each second-order variable F on second-order node N_F and each equation associated with it in the equation system associated to $\mathcal{G}$. U is a solution to $\mathcal{G}$ if it satisfies every equation. There are the following possibilities for N_F:

  - There are no non-redundant incoming horizontal edges to N_F and it contains only variable dependants. Then $U_0(N_F) = G$ for a second-order variable G in N_F. But also, the only equations associated with N_F are establishing the equivalence of all second-order variables in N_F, including $F \approx_{\mathcal{G}} G$. Thus, any solution U finer than U_0 will satisfy these equations.
  - There are no non-redundant incoming horizontal edges to N_F and exactly one non-variable dependant ϕ in N_F (there can be no more than one because the graph is prenormal). Then, $U_0(N_F) = \phi$. The equations associated with N_F are establishing the equivalence of all dependants in N_F, and thus their equivalence with ϕ. But ϕ is a non-variable dependant, and therefore any solution U finer than U_0 must be such that $U(F) = \phi$, which thus satisfies the equations.
  - There is exactly one non-redundant incoming horizontal edge E to N_F, with head H and sources S_i. Then, $U_0(N_F) = U_0(H)\{U_0(S_1), ..., U_0(S_n)\}$, and in particular this is the value for F. Note that because the graph is quasinormal, N_F must contain only variable dependants. Therefore, all the equations associated with N_F establish that all these variables are instantiated to equivalent terms, and the equation associated to the edge: $\chi_{N_F} \approx \chi_H\{\chi_{S_1}, ..., \chi_{S_n}\}$. But because instantiations are defined on second-order variables, any finer instantiation U to $U_0(N_F) = U_0(H)\{U_0(S_1), ..., U_0(S_n)\}$ must be of the form $U(N_F) = U(H)\{U(S_1), ..., U(S_n)\}$, and thus the equation is satisfied.

- Consider each unifier variable σ_i and each first-order variable X. Consider the node $N_{\sigma_i X}$ containing the dependant $\sigma_i X$ and each equation associated with it in

the equation system associated with $\mathcal{G}$. There are the following possibilities for $N_{\sigma_i X}$:

- $N_{\sigma_i X}$ has no non-redundant incoming horizontal edges. It does, however, contain at least one incoming vertical edge from a node without incoming horizontal edges. Thus, $U_0(N_{\sigma_i X}) = Y$, where Y is a first-order variable not assigned to any other node in the graph. But then the only equations associated with $N_{\sigma_i X}$ establish the equality of all dependants in the node. By definition of U_0, these equations are already satisfied in U_0, and so they must be in any finer solution U.
- $N_{\sigma_i X}$ has exactly one non-redundant incoming horizontal edge E with head H and sources S_i. Then $U_0(N_{\sigma_i x}) = U_0(H)(U_0(S_1),...,U_0(S_n))$. All the equations associated with $N_{\sigma_i X}$ establish that all the dependants contained in it are substituted by equivalent terms, and the equation associated to the edge: $\kappa_{N_{\sigma_i X}} \approx \chi_H(\kappa_{S_1},...,\kappa_{S_n})$. But because substitutions are defined on first-order variables, any finer substitution to $U_0(N_{\sigma_i X}) = U_0(H)(U(S_1),...,U(S_n))$ must be of the form $U(N_F) = U(H)(U(S_1),...,U(S_n))$, and thus the equation is satisfied.

□

The above result is to be read the following way: *A normal dependency graph is equivalent to an explicit solution to the unification equation system it represents, and thus we can consider to have* **solved** *the graph if we reduce it to a normal one.*

In the following we will deal directly and formally with non-determinism. In order for that process to be smooth, we provide a few definitions and general results that apply beyond dependency graphs and unification systems.

Definition 7.6.5. *Let $\mathcal{R}$ be a set of non-deterministic rewrite rules, each of which $R \in \mathcal{R}$, when applicable, produces, from an element $\mathcal{G}$ of a certain kind, a (possibly empty) set $R(\mathcal{G})$ of the same kind.*

Given an initial element $\mathcal{G}_0$, we define a search tree *with root $\mathcal{G}_0$ to be any directed tree containing elements of the kind of $\mathcal{G}_0$, whose root is $\mathcal{G}_0$, and such that for each non-leaf node $\mathcal{G}_i$, it is such that its set of children is the result $R(\mathcal{G}_i)$ of applying an applicable rewrite rule R to $\mathcal{G}_i$.*

We say that a search tree $\mathcal{T}$ is complete *with respect to $\mathcal{R}$ if for each leaf $\mathcal{G}_i$ in $\mathcal{T}$, no rule in $R \in \mathcal{R}$ is applicable to $\mathcal{G}_i$ (every leaf is irreducible). We write $\mathcal{L}(\mathcal{T})$ to describe the set of leaves of $\mathcal{T}$.*

There are several things to unpack in this definition that are worth describing explicitly, specially when it comes to cardinalities:

- For the same initial element $\mathcal{G}_0$, there may be several and very different search trees, and in general an infinite amount of them. This is because at each level we choose **only one rule** to explore. Different trees would choose different rules at each level, from the set of applicable ones.

- If $\mathcal{R}$ is countable (and the number of potential ways in which each is applicable to any given graph is countable), then the number of **nodes** (and thus the number of **leaves**) of any search tree must also be countable. This can be seen by considering that nodes correspond to finite sequences of natural numbers (which branch to take at each division until we reach the node).

- However, it is **not** in general true that the number of **paths** in a search tree with countable $\mathcal{R}$ is countable. Infinitely deep, infinitely branching branches may occur that produce uncountable numbers of infinitely deep branches. This can be seen by considering that this would correspond to infinite sequences of natural numbers (and thus equipotent to irrational numbers).

Search trees allow us to consider non-deterministic spaces declaratively, without thinking explicitly about the order in which we explore them. We are, however, interested in building algorithms and procedures that explore these search trees. Fortunately, we can provide *generic results* that establish that any search tree with countable $\mathcal{R}$ can be explored in a *fair* way: every leaf is eventually explored.

Definition 7.6.6 (Fair exploration of search trees)**.** *Let $\mathcal{R}$ be a countable set of rewrite rules. Let* `children` *be a function that, for an arbitrary search tree $\mathcal{T}$, and given a node $\mathcal{G} \in \mathcal{T}$, produces the (potentially infinite) list of its children and* `isleaf` *a function that, given a node $\mathcal{G} \in \mathcal{T}$ identifies if $\mathcal{G}$ is a leaf in $\mathcal{T}$. Then, an algorithm $\mathbb{D}$ is a* fair exploration *algorithm if, using the* `children` *and* `isleaf` *functions, it produces, given the root $\mathcal{G}_0$ of an arbitrary $\mathcal{T}$, a (potentially infinite) list $\mathbb{D}(\mathcal{G}_0)$ such that every leaf $L \in \mathcal{L}(\mathcal{T})$ is in $\mathbb{D}(\mathcal{G}_0)$ (with finite index, of course).*

It is very important to note the order of the quantifiers in the statement of the fairness problem. A fair exploration algorithm **is not** one algorithm for each search tree: such a solution is trivial to obtain. Rather, there must be a **single algorithm** $\mathbb{D}$ that, when applied to an arbitrary search tree, correctly produces a fair exploration of its leaves.

The following is the pseudocode for $\mathbb{D}$ (`diagonalize`), an algorithm that solves the fairness problem. The way that we work with infinite lists in this algorithm is the way that lazy languages like Haskell do so: function calls are to be understood as assignment of expressions that are only evaluated when the expression is pattern matched against. This allows infinite recursion expressions that are productive (generate some partial results over the recursion process) to correctly correspond to infinite lists whose every element has a finite index. Note that we use an additional element `(..)` to represent a computation step that does not produce any result but is finite. We will call elements that are not `(..)` *actual* elements. This is crucial to the fairness of the algorithm, as it prevents a child with no leaves underneath it from blocking other children producing leaves.

Algorithm 7.6.7 (Diagonalization of a search tree)**.** .

```
fun diagonalize(G):
   return collapse(diagonalize_uncollapsed(G))

fun diagonalize_uncollapsed(G):
   if isleaf(G):
      return [G]
   else:
      cs <- children(G)
      return diagonalize_apply(diagonalize_uncollapsed,cs)

-- Takes a function that produces a (potentially infinite) list from an
    element
-- and a (potentially infinite) list of elements, and combines
-- the result of applying the function to each element
-- together into a single, fair, (potentially infinite) list.
-- The result list may, however, contain newly introduced (..) elements
```

```
    to ensure
-- the fairness of the search.
fun diagonalize_apply(F,L):
   case L of
      []: return []
      -- We apply the function to the first element to produce a single
          (potentially infinite) list.
      -- And also recursively diagonalize the remainder of the list.
      -- We interleave these two results for fairness.
      -- However, this is not enough,
      -- because diagonalize_apply may call itself recursively to find
          the first element
      -- in a sub-tree with no leaves, never finding an inexistent
      -- first element, blocking the rest of the tree from being explored.
      -- We use (..) to let the parent function call know that we did
          some work,
      -- even if we did not produce any leaf yet.
      (x:xs): return interleave(F(x),diagonalize_apply(F,xs))

end fun

-- Combines two (potentially infinite) lists into a single, fair,
    infinite list,
-- by interleaving the two lists.
fun interleave(L1,L2):
   case L1 of
      []: return (..):L2
      -- We swap the order of the lists at each step to ensure the
          fairness.
      (x:xs): return (x:interleave(L2,xs))

-- Removes any (..) elements from an infinite list.
-- Of course, if the list has no more elements, this function produces a
    non-terminating list.
-- But if between any two elements there is a finite amount of (..),
    then it removes them without issues.
fun collapse(L):
```

```
case L of:
  []: return []
  ((..):xs): return collapse(xs)
  (x:xs): return x:collapse(xs)
```

Theorem 7.6.6 (Fairness of diagonalization)**.** $\mathbb{D}$ *is a fair exploration algorithm.*

Proof. To properly produce this proof, and due to how the algorithm relies on lazy evaluation, it is important to differentiate between the following three things, given a list in the program:

- The list's *thunk* itself. This is the exact expression with which the list is defined. For example, for `collapse([(..),1,(..),2])`, the thunk is **exactly** `collapse([(..),1,(..),2])`.
- The *one-step pattern match* of the thunk, after matching the thunk with a **cons** structure `head:tail`. This may incur evaluation of functions and expressions **just as much as necessary** to obtain the head and tail of the thunk, as unevaluated thunks themselves.

  For example, for `collapse([(..),1,(..),2])`, the one-step pattern match evaluates the collapse function, which evaluates to the thunk `collapse([1,(..),2])`, which must still be evaluated, producing the thunk `x:collapse([(..),2])`, which is in the desired shape and is thus the *one-step pattern match* of `collapse([(..),1,(..),2])`.
- The *abstract semantic sequence* that the list represents. This is a mathematical element result of fully evaluating (including an infinite number of steps) the thunk, and is better understood as a sequence that produces elements rather than a static list. In order to reason about the abstract semantic list, it is not enough to consider inductive definitions and individual steps, but instead we will need to consider universally quantified reasoning and infinitely many steps at once.

  For example, the abstract semantic sequence of `collapse([(..),1,(..),2])` is the finite sequence `[1,2]`. However, the abstract semantic sequence result of applying `collapse` to a list consisting exclusively of an infinite number of `(..)` is an a non-terminating sequence that never produces any elements.

  We say that a list is *well defined* if every element in its abstract semantic sequence has a finite index. In other words, there is only a finite number of elements that

come before it in the list.

For example, the list result of naively concatenating the list of all natural numbers with itself is not well defined, because some elements are hidden behind infinite numbers of elements. However, interleaving the list of natural numbers with itself is a well defined list.

We say that a list `L` is *fair* if its abstract semantic sequence is productive for as long as it has not terminated, and for every element in `L`, there is only a finite number of elements that come before it in `L`.

First, we note that if a non-empty list `L` is fair, then producing its one-step pattern match is a terminating computation. If the list is not empty, then it has elements in it. Thus, and because it is fair, it must be productive. Which by definition means that we can compute the one-step pattern match.

Next, we note that if `L` is a well defined fair list, then `collapse(L)` is also well defined and fair, and produces exactly the actual elements of `L` in the same order. This is clear as `collapse` leaves the list as-is except it removes `(..)` one at a time, and every actual element will by definition appear after a finite amount of `(..)`.

Second, we claim that if `L1` and `L2` are well defined fair lists, then `interleave(L1,L2)` is a well defined fair list whose actual elements correspond to the union of all the actual elements in `L1` and `L2`. To see this, consider that `interleave` never throws away any elements in `L1` or `L2`: every time it extracts one from them, it includes them in the result; nor does it introduce any actual elements not in `L1` or `L2`. `L1` is fair and thus either it is empty or its one-step pattern match can be computed. Furthermore, for every element in `L1` or `L2`, because the lists are well defined, they will appear in the abstract semantic sequences of those lists after a finite number of steps. Thus, because `interleave` explores both lists at the same time one step at a time, there is only a finite number of computations (and thus also a finite number of elements) before each element on each of the two lists in the abstract semantic sequence of `interleave(L1,L2)`, and thus it is well defined and fair.

At the centre of this proof is the proof that if `F` terminates for every element in `L`, `L` is a well defined fair list and `F(x)` is a well defined fair list for every `x` in `L`,

then `diagonalize_apply(F,L)` is a well defined fair list containing the union of all `F(x)` for `x` in `L`. To see this, first consider that if `L` is non-empty and fair, then its one-step pattern match `x:xs` can be computed in finite time. Thus the evaluation of `diagonalize_apply(F,L)` will terminate in finite time and, if `L` is non-empty, return `interleave(F(x),diagonalize_apply(F,xs))`. Next, note that `interlave(L1,L2)` will never be an empty list. It may contain no actual elements, but it will always contain at least one `(..)`. Thus, and since `interleave`'s first element will be either `F(x)`'s first element if it has one or `(..)`, we will always be able to produce the one-step pattern match of `interleave(F(x),diagonalize_apply(F,xs))`, and thus we will always be able to produce the one-step pattern match of `diagonalize_apply(F,L)` (and we have proven this without relying on the well definedness or fairness of `diagonalize_apply(F,L)`).

Therefore, the first depth of the execution of `interleave(F(x),diagonalize_apply(F,xs)` will terminate and produce either the first element of `F(x)` or `(..)`. The second depth will terminate and produce either the first element of `diagonalize_apply(F,xs)` or `(..)`. But we already knew that, unless `xs` is empty, the first element of `diagonalize_apply(F,xs)` will be either the first element of `F(x2)` (where `x2` is the first element of `xs`) or `(..)`. By recursion, this shows that the first element of every `F(x)`, for `x` in `L` is produced. Similarly, the third depth of `interleave(F(x),diagonalize_apply(F,xs)` will terminate and produce either the first element of the tail of `F(x)` (which is the second element of `F(x)`) or `(..)` if there is no such element. By recursion, we can see this produces all the elements in every `F(x)` after a finite number of steps. Thus, `diagonalize_apply(F,L)` is a well defined fair list containing the union of all `F(x)` for `x` in `L`.

Now show that `diagonalize_uncollapsed(G)` is a well defined fair list containing all leaves under `G` in (T). Here we **can** use induction over the structure of (T), because we are only concerned about **leaves**, which thus appear in the search tree at a finite depth. If `G` is a leaf, then `diagonalize_uncollapsed(G)` trivially returns `[L]`, which is a fair list containing all leaves under `G`. If `G` is not a leaf, then it has children. By induction, we may assume that `diagonalize_uncollapsed(c)` is a well defined fair list containing all leaves under `c` for each child `c` of `G`. By definition, the leaves under `G` are the union of all the leaves under each `c`. Thus, and because we proved the well definedness and fairness of `diagonalize_apply`, we know that `diagonalize_apply(diagonalize_uncollapsed,cs)` will be a well defined fair list containing the union of all leaves under each child `c` of `G`, which is the same as a well

defined fair list containing the union of all leaves under `G`, and we are finished.

Finally, we already showed that if `diagonalize_uncollapsed(G)` is a well defined fair list, then `collapse(diagonalize_uncollapsed(G))` will be a well defined fair list containing exactly the same actual elements, and thus it is a well defined fair list containing exactly the union of all leaves under `G`.

□

As a side note, the reader should convince themselves that the collapsing of the lists must happen at the end of the algorithm: the separation between `diagonalize` and `diagonalize_uncollapsed` is necessary. If we diagonalized over the collapsed list, then we would not be able to ensure productivity of each sub-list, which could make sub-trees with no leaves kidnap the algorithm's execution and prevent exploring other sub-trees. Collapsing in the end is fine, because then the only thing that may happen is that if there are no leaves under `G` then `diagonalize(G)` may not terminate or produce any results: **this is acceptable for the general tree, but not for each sub-tree**.

Corollary 7.6.2. *Let $\mathcal{R}$ be a countable set of non-deterministic rewrite rules. Assume that for each rule $R \in \mathcal{R}$ there is an algorithm that tells us whether R is applicable to a given element $\mathcal{G}$, and if it is, it produces a well defined fair list of the results of applying R to $\mathcal{G}$.*

Then, for any arbitrary choice of rewrite rule application order, there is an algorithm that, given an element $\mathcal{G}$, produces a fair list of all the irreducible elements result of applying the rules in said order to $\mathcal{G}$.

Proof. The assumptions give us the `children` and `isleaf` functions. The arbitrary choice of rewrite rule application order gives us a search tree $\mathcal{T}$. Then, we need only to apply `diagonalize` to the root $\mathcal{G}$ to obtain the fair list.

□

Going back to $\mathcal{R}^5$ and dependency graphs, we can use this result to show that $\mathcal{R}^5$ is fair regardless of application order. For that, though, we still need to show that every solution to a dependency graph will be represented by a leaf in any valid search order. That is a task of its own, but it can be reduced to defining an appropriate measure on dependency graphs.

The result that we eventually want to prove is that for any search tree $\mathcal{T}$ complete with respect to $\mathcal{R}^5$ and root $\mathcal{G}$, all solutions $U \in \mathcal{U}(\mathcal{G})$, are represented by a leaf $\mathcal{G}^N_U \in \mathcal{L}(\mathcal{T})$. But note that by definition of complete search tree, $\mathcal{G}^N_U$ is a leaf in $\mathcal{T}$ if and only if $\mathcal{G}^N_U$ is irreducible in $\mathcal{R}^5$, and by lemma 7.5.6, this is equivalent to $\mathcal{G}^N_U$ being a normal graph. Moreover, note that because all rules in $\mathcal{R}^5$ are solution preserving, we know that for every non-leaf node in $\mathcal{T}$, its set of solutions is equal to the set of solutions of its children. This implies that, if $\mathcal{T}$ were finite, then every branch would eventually end in a leaf, and the result would be trivially true. However, this is not, in general, the case: infinitely deep branches (that never produce leaves) may occur in $\mathcal{T}$. Note as well that it would **not** be enough to show that every branch has a leaf underneath it, since there could be a core set of solutions that permanently remained in the infinitely deep branches of the graph with no leaves underneath it. But we can give a twist to this approach to make it work. What we will do instead is to show that there is a measure μ^N on dependency graphs that **increases** as we move further down the graph, such that for every solution U, there is a maximum ${}^M\mu^N_U$ amongst all dependency graphs for which U is a solution. This requires further precision.

The measure μ^N need not strictly increase for every rule in $\mathcal{R}^5$. In theorem 7.6.4 we showed that for every graph $\mathcal{G}$, there may only be a finite number of rules in $\mathcal{R}^{3^*}$ applied to it, regardless of order. Thus, if we show that μ^N strictly increases for rules in $\mathcal{R}^5 - \mathcal{R}^3$ (that is, single factorization, half factorization of function symbols, half factorization of projections and multiple factorization) and it does not decrease for any rule in $\mathcal{R}^{3^*}$, then between any two applications of rules in $\mathcal{R}^5 - \mathcal{R}^{3^*}$, there may only be a finite number of applications of rules in $\mathcal{R}^{3^*}$, and the measure increases for each application of rules in $\mathcal{R}^5 - \mathcal{R}^{3^*}$, and therefore for any number n, there is a depth d_n such that all dependency graphs in the search tree $\mathcal{T}$ with depth greater than d_n have $\mu^N > n$. In the previous conclusion we are using the fact that, for any given tree $\mathcal{T}$, and for each specific value of μ^N, the number of finite steps before no more acyclic seminormalizing rules can be applied is bounded among branches. This is a consequence of the fact that all rules in $\mathcal{R}^5$ produce only a finite number of dependency graphs.

This means that as we move down the depth of the search tree, and as long as cycles are not produced in the graph, we inexorably increase μ^N. Because at any depth, the set of solutions of all graphs at that depth must be equal to the set of solutions of the root (except for solutions accounted for in leaves at smaller depths), and all graphs for which each solution is a solution have bounded μ^N, then every solution of the root graph must

eventually be accounted for by a leaf, and thus a normal dependency graph, in $\mathcal{T}$.

Therefore, and as a conclusion to this initial part of the proof, we have shown that the eventual result we wish to show can be shown (as long as no cycles are produced in the graph) by producing a measure μ^N such that:

- μ^N is finite for every dependency graph.
- μ^N does not decrease when applying any rule in $\mathcal{R}^{3^*}$
- μ^N strictly increases when applying any rule in $\mathcal{R}^5 - \mathcal{R}^3$.
- For every ground unification solution U, there is a maximum value ${}^M\mu^N_U$ such that U can never be a solution to a dependency graph $\mathcal{G}_>$ for which $\mu^N(\mathcal{G}_>) > {}^M\mu^N_U$.
- μ^N is not *dense* with respect to the rewrite rules: from an initial graph $\mathcal{G}$ and for any given value of μ^N, only a finite number of rules may be applied before a value of μ^N greater than or equal is reached.

To define μ^N, we will, once again, combine several individual measures into one.

Definition 7.6.8 (Depth of instantiation measure)**.** *For each second-order node N in an acyclic graph, define $\mu^I(N)$, the* depth of instantiation measure *to be* 0 *if the node has no incoming horizontal edges, or* 1 *plus the maximum of the recursive $\mu^I(N)$ on the sources of all incoming horizontal edges if it has incoming horizontal edges.*

For each second-order variable F, define $\mu^I(F)$ to be $\mu^I(N_F)$, where N_F is the node in which F appears in the graph.

Then, define the measure μ^I on the graph to be the sum of $\mu^I(F)$ for all second-order variables F in the graph.

Lemma 7.6.4. *Let R be any rewrite rule in $\mathcal{R}^5$, and $\mathcal{G}_1$ a dependency graph to which R is applicable. Let $\mathbb{G}_2$ be the set of dependency graphs after applying the rule. Then, for every $\mathcal{G}_2 \in \mathbb{G}_2$, $\mu^I(\mathcal{G}_2) \leq \mu^I(\mathcal{G}_1)$.*

Moreover, for the partial head instantiation branches of single factorization and multiple factorization, μ^I strictly increases.

Proof. Since horizontal edges are never removed in the graph, μ^I may only change if horizontal edges are added to the graph or nodes are merged. But both of these actions may only increase μ^I, so no rule may decrease it.

The partial head instantiation branch of single factorization takes the node H^v that contains only second-order variables, which does not have incoming horizontal edges because it is the head of a horizontal edge and the graph is factorizable, and adds an incoming horizontal edge to it. Thus, μ^I strictly increases for all the second-order variables of it, and therefore it strictly increases globally in the graph.

Similarly, the head instantiation branch of multiple factorization takes the node H_{i_0} that contains only variable dependants, which does not have incoming horizontal edges because it is the head of a horizontal edge and the graph is factorizable, and adds an incoming horizontal edge to it. Thus, μ^I strictly increases for all the second-order variables of it, and therefore it strictly increases globally in the graph.

□

Definition 7.6.9 (Instantiated variables measure)**.** *Define the* instantiated variables measure *on a dependency graph, written $\mu^{I\approx}$, to be the number of second-order variables F in the dependency graph for which the dependant F is in the same node as a non-variable dependant.*

Lemma 7.6.5. *Let R be any rewrite rule in $\mathcal{R}^5$, and $\mathcal{G}_1$ a dependency graph to which R is applicable. Write $\mathbb{G}_2$ to be the set of dependency graphs that R produces when applied to $\mathcal{G}_1$. Then, for each $\mathcal{G}_2 \in \mathbb{G}_2$, $\mu^{I\approx}(\mathcal{G}_1) \leq \mu^{I\approx}(\mathcal{G}_2)$.*

Moreover, for the projection merging branches of single factorization and multiple factorization, the inequality is strict.

Proof. No rule removes variables from the graph or splits nodes that are together. This alone makes it impossible for the measure to ever decrease.

The projection merging branches of single factorization and multiple factorization explicitly merge a node containing only second-order variables with a projection (non-variable), and thus the measure is strictly increased.

□

We will use the measure $\mu^{\approx}$ defined in definition 7.6.1.

Lemma 7.6.6. *$\mu^{\approx}$ does not increase when applying half factorization of function symbols or half factorization of projections.*

Proof. An important part of the definition of $\mu^{\approx}$ is that it is dependent on $\mathcal{D}^1(\mathcal{G})$ and $\mathcal{D}^2(\mathcal{G})$. We showed that rules in $\mathcal{R}^{3^*}$ did not change these sets. Neither do they change when applying half factorization rules, since they do not add second-order variables. This is important for the correctness of this proof.

Since half factorization of function symbols is solution preserving and deterministic, it produces one graph with exactly the same set of solutions as the original graph, and so $\mu^{\approx}$ may only change when nodes are merged, decreasing it.

On the other hand, half factorization of projections is non-deterministic, but each result graph is absolutely equal to the original graph, except the node H has been merged with a projection π_k^m. The only difference there may therefore be in the sets of solutions between the original graph and each individual result graph is that the equation $\pi_k^m \approx \chi_H$ holds afterwards and possibly did not before. But this equation is explicitly represented by the dependants being in the same node in the graph, so $\mu^{\approx}$ does not increase for half factorization of projections either.

□

Definition 7.6.10 (Inverse implicit equivalent dependants measure)**.** *Define the measure $\mu^{-\approx}$, called the* inverse *of the implicit equivalent dependants measure, as the number of elements in $\mathcal{D}^1(\mathcal{G}) \cup \mathcal{D}^2(\mathcal{G})$ minus the value of $\mu^{\approx}$.*

Lemma 7.6.7 (Increasing inverse implicit equivalent dependants measure)**.** *Let R be any rewrite rule that does not change $\mathcal{D}^1(\mathcal{G})$ or $\mathcal{D}^2(\mathcal{G})$, then, $\mu^{-\approx}$ is strictly increasing or non-decreasing respectively if $\mu^{\approx}$ is strictly decreasing or non-increasing.*

Proof. It is straighforward from the definition of the inverse measure and the fact that the boundary sets do not change.

□

Definition 7.6.11 (Target function symbols measure)**.** *Define the* target function symbols measure*, written $\mu^{\rightarrow^c}$ on a dependency graph $\mathcal{G}$ to be the number of function symbols f, each on node N_f, for which N_f has non-redundant incoming horizontal edges.*

Lemma 7.6.8. *Let $\mathcal{G}_1$ be a dependency graph for which half factorization of function symbols is applicable. Let $\mathcal{G}_2$ be the result of applying the rule to it. Then,*

$\mu^{\rightarrow^c}(\mathcal{G}_1) > \mu^{\rightarrow^c}(\mathcal{G}_2)$.

Also, let $\mathcal{G}_1$ be a dependency graph for which vertical monotony of horizontal edges is applicable. Let $\mathcal{G}_2$ be the result of applying the rule to it. Then, $\mu^{\rightarrow^c}(\mathcal{G}_1) \geq \mu^{\rightarrow^c}(\mathcal{G}_2)$.

Finally, let $\mathcal{G}_1$ be a dependency graph for which function dumping is applicable. Let $\mathcal{G}_2$ be the result of applying the rule to it. Then, $\mu^{\rightarrow^c}(\mathcal{G}_1) \geq \mu^{\rightarrow^c}(\mathcal{G}_2)$.

Proof. Half factorization of function symbols explicitly marks as redundant every incoming horizontal edge to a node with a function symbol in it, and thus $\mu^{\rightarrow^c}$ strictly decreases.

Vertical monotony of horizontal edges does not change anything related to second-order nodes or edges except making them the head of edges, and thus it does not change $\mu^{\rightarrow^c}$.

Finally, function dumping marks a horizontal edge as redundant, but adds a new one with the same target in its place. Therefore, the number of function symbols with incoming horizontal edges does not change.

□

Definition 7.6.12 (Inverse target function symbols measure)**.** *Define the* inverse target function symbols measure *on a dependency graph $\mathcal{G}$, written $\mu^{-\rightarrow^c}$, to be such that $\mu^{-\rightarrow^c}(\mathcal{G}) = \mathcal{D}^2(\mathcal{G}) - \mu^{\rightarrow^c}(\mathcal{G})$.*

Lemma 7.6.9. *Let R be a rewrite rule that does not change $\mathcal{D}^2(\mathcal{G})$ and for which $\mu^{\rightarrow^c}$ is non-increasing or strictly decreasing. Then, $\mu^{-\rightarrow^c}$ is non-decreasing or strictly increasing respectively for this rule.*

Proof. It is a direct consequence of the definition of $\mu^{-\rightarrow^c}$ and the fact that $\mathcal{D}^2(\mathcal{G})$ does not change.

□

Definition 7.6.13 (Target projections measure)**.** *Define the* target projections measure*, written $\mu^{\rightarrow^\pi}$ on a dependency graph $\mathcal{G}$ to be the number of non-redundant second-order horizontal edges whose head does not contain a projection but whose target does.*

Lemma 7.6.10. *Let $\mathcal{G}_1$ be a dependency graph to which half factorization of projections is applicable. Let $\mathbb{G}_2$ be the set of result dependency graphs after applying the rule. Then, for any $\mathcal{G}_2 \in \mathbb{G}_2$, $\mu^{\rightarrow^\pi}(\mathcal{G}_1) > \mu^{\rightarrow^\pi}(\mathcal{G}_2)$.*

Also, let $\mathcal{G}_1$ be a dependency graph to which half factorization of function symbols is applicable. Let $\mathcal{G}_2$ be the result of applying the rule to it. Then, $\mu^{\rightarrow^\pi}(\mathcal{G}_1) \geq \mu^{\rightarrow^\pi}(\mathcal{G}_2)$.

Moreover, let $\mathcal{G}_1$ be a dependency graph to which vertical monotony of horizontal edges is applicable. Let $\mathcal{G}_2$ be the result of applying the rule to it. Then, $\mu^{\rightarrow^\pi}(\mathcal{G}_1) \geq \mu^{\rightarrow^\pi}(\mathcal{G}_2)$.

Finally, let $\mathcal{G}_1$ be a dependency graph to which function dumping is applicable. Let $\mathcal{G}_2$ be the result of applying the rule to it. Then, $\mu^{\rightarrow^\pi}(\mathcal{G}_1) \geq \mu^{\rightarrow^\pi}(\mathcal{G}_2)$.

Proof. Half factorization of projections precisely merges the head of an incoming edge to a projection that did not previously contain a projection with a projection, therefore strictly decreasing $\mu^{\rightarrow^\pi}$.

On the other hand, if half factorization of function symbols is applicable, then the graph is factorizable. This means that the node N containing a function symbol cannot contain projections. The rule marks as redundant edges incoming to this node and which therefore cannot be relevant for $\mu^{\rightarrow^\pi}$ since their target does not contain a projection, and adds new edges whose target is a new anonymous second-order node, which therefore does not contain projections either. Thus, $\mu^{\rightarrow^\pi}$ may not increase.

Vertical monotony of horizontal edges does not change anything related to second-order nodes or edges except making them the head of first-order edges, and thus it does not change $\mu^{\rightarrow^\pi}$, because no edges with target a projection may be created or marked as redundant.

Finally, function dumping marks a horizontal edge as redundant, but adds a new one with the same target in its place. Furthermore, the edge being marked as redundant did not contain a projection by definition of function dumping. Therefore, if the added edge has as target a node with a projection, then so did the edge being marked as redundant, and therefore $\mu^{\rightarrow^\pi}$ does not increase.

□

Definition 7.6.14 (Inverse target projections measure)**.** *Define the* inverse target projections measure *on a dependency graph $\mathcal{G}$, written $\mu^{-\rightarrow^{\pi}}$, to be such that $\mu^{-\rightarrow^{\pi}}(\mathcal{G}) = \mathcal{D}^2(\mathcal{G}) - \mu^{\rightarrow^{\pi}}(\mathcal{G})$.*

Lemma 7.6.11. *Let R be a rewrite rule that does not change $\mathcal{D}^2(\mathcal{G})$ and for which $\mu^{\rightarrow^{\pi}}$ is non-increasing or strictly decreasing. Then $\mu^{\rightarrow^{\pi}}$ is non-decreasing or strictly increasing respectively for this rule.*

Proof. It is a direct consequence of the definition of $\mu^{\rightarrow^{\pi}}$ and the fact that $\mathcal{D}^2(\mathcal{G})$ does not change.

□

Definition 7.6.15 (Normalizing measure on dependency graphs)**.** *Define the ordering μ^N on dependency graph $\mathcal{G}$ to be the lexicographic ordering which considers the ordering induced by the following measures (giving most priority to the first one and least to the last one):*

- μ^I
- $\mu^{I\approx}$
- $\mu^{-\approx}$
- $\mu^{-\rightarrow^{\pi}}$
- $\mu^{-\rightarrow^{c}}$

Lemma 7.6.12 (Non-decreasing normalizing measure)**.** *Let R be any rule in $\mathcal{R}^{3^*}$ and $\mathcal{G}_1$ a dependency graph to which R is applicable. Write $\mathbb{G}_2$ for the set of dependency graphs result of applying the rule.*

Then, for every graph $\mathcal{G}_2 \in \mathbb{G}_2$, $\mu^N(\mathcal{G}_1) \leq \mu^N(\mathcal{G}_2)$.

Proof. Consider each rule in $\mathcal{R}^{3^*}$:

- **Validate consistency** - This rule does not change the graph, other than to invalidate it, so the measure does not change unless a branch is terminated.

- **Vertical monotony of equivalences, edge zipping, projection simplification, zero factorization** - By lemma 7.6.4, μ^I does not decrease. By lemma 7.6.5, $\mu^{I\approx}$ does not decrease either. By lemma 7.6.1, $\mu^{\approx}$ strictly decreases, and since this rule does not modify $\mathcal{D}^1(\mathcal{G})$ or $\mathcal{D}^2(\mathcal{G})$, by lemma 7.6.7, $\mu^{-\approx}$ strictly increases, and thus so does μ^N.

- **Vertical monotony of horizontal edges, function dumping** - By lemma 7.6.4, μ^I does not decrease. By lemma 7.6.5, $\mu^{I\approx}$ does not decrease either. By lemma 7.6.1, $\mu^{\approx}$ does not increase, and since this rule does not modify $\mathcal{D}^1(\mathcal{G})$ or $\mathcal{D}^2(\mathcal{G})$, by lemma 7.6.7, $\mu^{-\approx}$ does not decrease either. By lemma 7.6.10, $\mu^{\rightarrow^\pi}$ does not increase and then by lemma 7.6.11, $\mu^{-\rightarrow^\pi}$ does not decrease either. Finally, by lemma 7.6.8, $\mu^{\rightarrow^c}$ does not increase and then by lemma 7.6.9, $\mu^{-\rightarrow^c}$ does not decrease either. Therefore, μ^N does not decrease.

□

Lemma 7.6.13 (Increasing normalizing measure)**.** *Let R be any rule in $\mathcal{R}^5 - \mathcal{R}^3$ and $\mathcal{G}_1$ a dependency graph to which R is applicable. Write $\mathbb{G}_2$ for the set of dependency graphs result of applying the rule.*

Then, for every graph $\mathcal{G}_2 \in \mathbb{G}_2$, $\mu^N(\mathcal{G}_1) < \mu^N(\mathcal{G}_2)$.

Proof. Consider each rule in $\mathcal{R}^5 - \mathcal{R}^3$:

- **Single factorization** - By lemma 7.6.4, μ^I does not decrease. By lemma 7.6.5, $\mu^{I\approx}$ does not decrease either. Moreover, for the partial instantiation branch, lemma 7.6.4 establishes that μ^I strictly increases. For the projection merging branches, 7.6.5 establishes that $\mu^{I\approx}$ strictly increases. These are all the branches that single factorization produces, and therefore μ^N strictly increases in every case.

- **Half factorization of function symbols** - By lemma 7.6.4, μ^I does not decrease. By lemma 7.6.5, $\mu^{I\approx}$ does not decrease either. By lemma 7.6.6, $\mu^{\approx}$ does not increase, and since this rule does not modify $\mathcal{D}^1(\mathcal{G})$ or $\mathcal{D}^2(\mathcal{G})$, lemma 7.6.7 implies that $\mu^{-\approx}$ does not decrease. By lemma 7.6.10, $\mu^{\rightarrow^\pi}$ does not increase, and thus by lemma 7.6.11, $\mu^{-\rightarrow^\pi}$ does not decrease. Finally, by lemma 7.6.8, $\mu^{\rightarrow^c}$ strictly decreases, and thus by lemma 7.6.9, $\mu^{-\rightarrow^c}$ strictly increases, and thus so does μ^N.

- **Half factorization of projections** - By lemma 7.6.4, μ^I does not decrease. By lemma 7.6.5, $\mu^{I\approx}$ does not decrease either. By lemma 7.6.6, $\mu^{\approx}$ does not increase,

and since this rule does not modify $\mathcal{D}^1(\mathcal{G})$ or $\mathcal{D}^2(\mathcal{G})$, lemma 7.6.7 implies that $\mu^{-\approx}$ does not decrease. Finally, by lemma 7.6.10, $\mu^{\rightarrow^\pi}$ strictly decreases, and thus by lemma 7.6.11, $\mu^{-\rightarrow^\pi}$ strictly increases, and thus so does μ^N.

- **Multiple factorization** - By lemma 7.6.4, μ^I does not decrease. By lemma 7.6.5, $\mu^{I\approx}$ does not decrease either. Moreover, for the partial instantiation branch, lemma 7.6.4 establishes that μ^I strictly increases. For the projection merging branches, 7.6.5 establishes that $\mu^{I\approx}$ strictly increases. These are all the branches that multiple factorization produces, and therefore μ^N strictly increases in every case.

□

We are only missing now the association with the solutions.

Lemma 7.6.14. *Let U be a ground unification solution. Then, there is a maximum value ${}^M\mu^N_U$ such that for every dependency graph $\mathcal{G}$ for which U is a solution, $\mu^N(\mathcal{G}) \leq {}^M\mu^N_U$.*

Proof. Consider the second-order variables in U. Assume, without loss of generality, that the instantiations are normal. For each of them, consider the depth of its instantiation d^I, defined recursively as 0 if they are instantiated to a function symbol or projection, and 1 plus the maximum amongst the depth of instantiation of its arguments if it is instantiated to a composition. Consider the sum $\sum_F d^I(F)$. If $\mathcal{G}$ has $\mu^I(\mathcal{G}) > \sum_F d^I(F)$, then U cannot possibly be a solution to $\mathcal{G}$, because the total depth of instantiation of all its variables would need to be greater than $\sum_F d^I(F)$.

Thus, there is a maximum value for μ^N such that any dependency graphs with greater value for μ^N cannot possibly have U as solution.

□

Lemma 7.6.15. *Given an initial dependency graph $\mathcal{G}$, and a specific value μ^N_0 of μ^N, there is a maximum number of applications of rewrite rules (other than the occurs check) such that after that number of applications, all dependency graphs that are in the result set have μ^N greater than μ^N_0.*

Proof. First note that since μ^I is the governing measure in μ^N and μ^I is a natural number (and therefore not dense), if we show that for any graph with a certain value of μ^I, there is a maximum number of applications of rules before all result graphs have μ^I greater than the original, this may only happen a finite number of times and the lemma would

be implied. So we can focus only on the remaining measures.

But if μ^I does not increase when applying rewrite rules, then none of those rules are multiple factorization. All the rules in $\mathcal{R}^5$ except the occurs check and multiple factorization preserve $\mathcal{D}^1(\mathcal{G})$ and $\mathcal{D}^2(\mathcal{G})$, and in particular the number of second-order variables in the graph. Thus, $\mu^{I\approx}$ will have a maximum value amongst those graphs (the total number of second-order variables), and similarly will the measures $\mu^{-\approx}$, $\mu^{-\to^\pi}$ and $\mu^{-\to^c}$ which are always between zero and a finite number dependent on $\mathcal{D}^1(\mathcal{G})$ and $\mathcal{D}^2(\mathcal{G})$. All of these measures are natural numbers and therefore there can only be a finite number of them before they reach the maximum value.

By lemma 7.6.13, every rule in $\mathcal{R}^5 - \mathcal{R}^3$ strictly increases μ^N. Also, by lemma 7.6.12, no rule in $\mathcal{R}^{3^*}$ decreases it, and by theorem 7.6.4, only a finite number of these rules may be applied to any dependency graph. Therefore, between every two applications of a rule in $\mathcal{R}^5 - \mathcal{R}^3$, there may only be a finite number of applications of a rule in $\mathcal{R}^{3^*}$, and thus, for every value of μ^N, it will be reached after a finite number of applications of these rewrite rules.

□

Theorem 7.6.7 (Eventual non-deterministic confluence of $\mathcal{R}^5$ under acyclicity)**.** *Let $\mathcal{T}$ be any search tree that is complete with respect to $\mathcal{R}^5$, that contains no cyclic graphs, and has root $\mathcal{G}$. Let $U \in \mathcal{U}(\mathcal{G})$ be a solution to $\mathcal{G}$.*

Then, there is a leaf $G_U^N \in \mathcal{L}(\mathcal{T})$ such that U is finer than the general solution $U_0(G_U^N)$ of the normal graph G_U^N.

Proof. By lemma 7.6.15, at any given depth of the search tree, there is a minimum value of μ^N that every dependency graph at that level must have. But by lemma 7.6.14, there is a maximum ${}^M\mu_U^N$ of μ^N amongst graphs that have U as solution. Since every rule is solution preserving, this must mean that U must be a solution to a leaf G_U^N at depth at most ${}^M\mu_U^N$. G_U^N is normal because of lemma 7.5.6 and the definition of complete search tree. But then, by theorem 7.6.5, U is finer than $U_0(G_U^N)$.

□

Theorem 7.6.8 (Solution finding algorithm for dependency graphs)**.** *For every way to choose rewrite rules amongst applicable ones to dependency graphs, there exists an algorithm that, given an arbitrary dependency graph $\mathcal{G}$, produces a well defined fair list of unification solutions U_i^N such that every unification solution $U \in \mathcal{U}(\mathcal{G})$ is finer than one of the U_i^N.*

Proof. It is a direct consequence of theorem 7.6.7 and corollary 7.6.2.

□

7.6.5 Quasinormalizing rules: Solution shape verification

Theorem 7.6.8 is the principal result about our rewrite rule system. However, as we described when we introduced the rule, multiple factorization is a very undesirable rule to apply because it introduces a large amount of non-determinism, largely increasing the size of the search space. Moreover, it may make sense at times to want to *verify* if there exists a solution to a set of equations with a certain shape, rather than to obtain the explicit solutions. It is important to understand the difference between this and the solution being an actual solution to the set of equations.

Definition 7.6.16 (Solution shape verification)**.** *Let E be a unification equation system. Let U be a unification solution. We say that E has a solution* with the shape of *U if there is a unification solution $U_R \preceq U$ (finer than U) such that $U_R \in \mathcal{U}(E)$.*

Of course, if U is a solution to E, then E has a solution with the shape of U. But the reciprocal is not always true.

The important result in this section is the following: a quasinormal graph always has at least one unification solution. And the relevant application of this is the following: we can verify if a graph $\mathcal{G}$ has a solution with the shape of a solution U by replacing the values of U in the graph and then checking if $\mathcal{G}$ can be quasinormalized while leaving one consistent result graph. The remainder of this section details this.

Theorem 7.6.9 (Quasinormal graphs have solutions)**.** *Let $\mathcal{G}$ be a quasinormal dependency graph. Then, $\mathcal{U}(\mathcal{G})$ is not empty.*

Proof. We begin this proof by noting that theorem 7.6.5 implies that this is true for **normal** graphs: the general solution is one solution and therefore $\mathcal{U}(\mathcal{G})$ is not empty.

What we will do is to show that, given a quasinormal graph $\mathcal{G}$, it can be modified into a normal graph $\mathcal{G}_N$ such that every solution of $\mathcal{G}_N$ is a solution of $\mathcal{G}$. Then, since $\mathcal{U}(\mathcal{G}_N)$ is not empty, then neither is it $\mathcal{U}(\mathcal{G})$.

The difference between quasinormality and normality is that while normality prevents any presence of multiple non-redundant incoming horizontal edges to a graph,

quasinormaly admits these as long as the heads of all of these edges are all second-order variables. Furthermore, a quasinormal graph is factorizable, and thus these heads have no incoming horizontal edges. This is important since it allows these variables to be instantiated to anything in solutions. Even further, any second-order node with non-redundant incoming horizontal edges contain only variable dependants. This gives total freedom to these variables, and only relative dependency between them. Let's use this formally.

Consider all nodes V_i in $\mathcal{G}$ that contain only second-order variables and whose nodes have no non-redundant incoming horizontal edges, and each second-order variable F in them. Write E_j^V to describe all the edges in the graph whose heads contain only second-order variables. Choose an arbitrary function symbol f with arity 0 in the signature[17]. Note that, just like a function symbol of arity n can have its arity artificially increased by composing it with projections (for example $g^1\{\pi_1^3\}$ has arity 3), the same can be done with function symbols of arity 0, except that they do not have explicit projections composed. That is, we can consider a second-order term of arbitrary arity which ignores all of its arguments and returns a constant. Form the graph $\mathcal{G}_N$ to be the graph $\mathcal{G}$ with the following changes:

- For each V_i, let n be the arity of V_i. If n is zero, merge V_i with f. Otherwise, add an incoming horizontal edge to it with f as head and no sources (to adapt f to the arity of V_i).
- Remove all edges E_j^V
- For each node N that had at least one incoming edge E_j^V, add an incoming edge to N with head f and no sources.

We note that $\mathcal{G}_N$ is normal. Because the V_i had no non-redundant incoming horizontal edges and contained only variables, neither the new incoming horizontal edges nor the merging of nodes can violate factorizability. Because $\mathcal{G}$ was quasinormal, the node containing f could not have non-redundant incoming horizontal edges, and thus the new edges replacing the removed ones do not violate factorizability either. Furthermore, all the nodes with more than one non-redundant incoming horizontal edge had all such edges be one of the E_j^V, all of which have been removed and replaced

[17]We do assume one exists in the signature. This assumption is often made in Herbrand model reasoning and it does not alter the fundamental notion of solution, rather only ensuring the syntactic expressivity is there to sustain it.

by an individual edge per node, and thus the normality condition is now met (and in particular quasinormality and seminormality).

Because $\mathcal{G}_N$ is normal, it has, by theorem 7.6.5, at least one solution $U \in \mathcal{U}(\mathcal{G}_N)$. It will satisfy all the equations in the equation system associated with $\mathcal{G}_N$, and thus automatically all the equations in the equation system of $\mathcal{G}$, except those that were removed when producing $\mathcal{G}_N$. These must thus be associated with the edges E_j^V removed. Note that since U is a solution to $\mathcal{G}_N$, $V_i \approx_U f$ for every V_i. For each node N in $\mathcal{G}$ with incoming edges E_j^V, consider whether N is a first or second-order node:

- If it is a first-order node, then the equations removed from $\mathcal{G}$ are of the form $\kappa_N \approx V_i(\kappa_{S_1}, ..., \kappa_{S_n})$. But, in U, each V_i is the constant function f which ignores its arguments, and thus these equations are all of the form $\kappa_N \approx f()$. These equations are held trivially since they only relate to each other, and each node N is only defined by these equations. It is important to note that vertical monotony ensures that any constraints at lower unifier levels that may extend to this level would come in the shape of incoming horizontal edges to the node that would have propagated through vertical monotony.

- If it is a second-order node, then the equations removed from $\mathcal{G}$ are of the form $\chi_N \approx V_i\{\chi_{S_1}, ..., \chi_{S_n}\}$. But, in U, each V_i is the constant function f which ignores its arguments, and thus these equations are all of the form $\chi_N \approx f$. These equations may not come in conflict with dependants in N since $\mathcal{G}$ was quasinormal and therefore the node N, with non-redundant incoming horizontal edges, could only contain variable dependants.

Thus, U is a solution to $\mathcal{G}$ and the theorem is proven.

□

Definition 7.6.17 (Solution shape verification algorithm)**.** *Assume we have an algorithm that chooses rewrite rules amongst applicable ones in $\mathcal{R}^4$ to dependency graphs. Then, define the* solution shape verification algorithm *$\mathcal{V}$ that takes as input a dependency graph and, as long as no cyclic graphs are produced, when it terminates, returns with true or false:*

- *Diagonally apply (see corollary 7.6.2) the rewrite rules in $\mathcal{R}^4$, except the occurs check.*

- *If an irreducible graph is produced, return true.*
- *If all branches end in failure, return false.*

Of course, the attractiveness of $\mathcal{V}$ is that it need not apply multiple factorization, thus considerably reducing its complexity.

Theorem 7.6.10 (Correctness and semidecidability of the solution shape verification algorithm under acyclicity)**.** *Let $\mathcal{G}$ be a dependency graph. If $\mathcal{V}$ terminates when applied to $\mathcal{G}$, then its result correctly indicates whether the graph has solutions. Furthermore, if $\mathcal{G}$ has solutions, then $\mathcal{V}$ will terminate as long as no cyclic graphs are produced.*

Proof. The correctness comes from theorem 7.6.9. Each rule in $\mathcal{R}^4$ is solution preserving, and an irreducible graph with respect to $\mathcal{R}^4$ is quasinormal by lemma 7.5.5. Thus if an irreducible graph is produced, then said graph has solutions and thus those solutions are also solutions of the root graph $\mathcal{G}$. Furthermore, if all branches end in failure, by the solution preserving properties of the rules, this means that $\mathcal{G}$ had no solutions.

The second part of the theorem is a consequence of the proof of theorem 7.6.7. If $\mathcal{G}$ has solutions, then by applying rules in $\mathcal{R}^5$, there would be leaves (irreducible graphs) in the search tree. But the same would trivially be true from rules in $\mathcal{R}^4$, because an irreducible graph with respect to $\mathcal{R}^5$ is also irreducible to $\mathcal{R}^4$, and we imposed no ordering in the application of rules in theorem 7.6.7. Thus, if $\mathcal{G}$ has solutions, then the complete search tree with respect to $\mathcal{R}^4$ has leaves, which can thus be enumerated fairly by corollary 7.6.2. Since we are only looking for one such solution, the algorithm will terminate in this case.

□

Note that if the graph has no solutions it is possible for the algorithm to continue indefinitely applying rules but never quite invalidating every branch in the graph. There is no solution to this particular situation. The problem is semidecidable.

7.7 Relation to standard higher-order unification

In this section, we discuss the inspiration, similarities and differences, at a technical level, between the algorithm described and standard higher-order unification and other closely related algorithms (see §3.2.2 or [Huet, 1975, Dowek, 2001]).

Generally speaking, the approach is very similar to any other unification algorithm: we express the unification problem systematically and structurally proceed through it, instantiating variables when necessary and propagating information. One notable difference between our algorithm and standard higher-order logic is that we use a graph, as opposed to equations, to express the unification problem. In §7.7.1 we explain this in more detail, but fundamentally the aim of this is to allow us to propagate information and choose the next part of the problem to tackle more effectively. More precisely, our algorithm proceeds by:

- Normalizing terms to enable effective comparison between them.
- Instantiate variables based on the constraints presented in them when necessary. This produces non-determinism.
- Propagating information generated through instantiation in parts of the problem to every part of the problem that is affected by it.
- Checking for situations that indicate the problem has no solutions to stop the search.

All of these are achieved through the rewrite rules. We explain how in more detail in §7.7.2. We note that normalization of terms is in general assumed / enforced throughout in standard higher-order unification. In that case, it is also easier than in our algorithm because the scope of the standard algorithm is a single equation / unification problem, whereas in our algorithm we solve multiple unification equations simultaneously. We describe this aspect more in detail in §7.7.1. This also relates to the value of propagating information through different parts of the problem, also explained in §7.7.1. Standard higher-order unification does not check for complex situations that indicate the problem has no solutions (e.g. occurs check). This is primarily due to the goal of checking for *unifiability* rather than for an *enumeration of all unifiers* that allows this and other algorithms to be more clever about the exploration of the search space, and makes verification of some of these situations unnecessary / unhelpful. We talk about this in §7.7.2 and §7.7.4. We do note that, in the implementation of the algorithm, we decided to forgo some of these checks for practical reasons as well. This is explained in more detail in chapter 8. Finally, both our algorithm and standard higher-order unification algorithms incur non-determinism. We explore this aspect in §7.7.3.

7.7.1 Why graphs

Dependency graphs express relations between terms and other elements; the same type of relations that unification equations do. Indeed, definition 7.2.3 and algorithm 7.2.5 establish this connection. However, a dependency graph allows more explicit and consequential relations between elements. As a particularly relevant case, they allow the representation of multiple unification equations at once. Multiple representation is not, per se, a fundamental difference, but combined with the more direct representation of all the connections and relations between multiple elements in the system (for example, a variable appearing in multiple parts of the equation system), it also enables us to propagate information about instantiations throughout the graph in a more targetted manner, and to choose the next aspects of the problem to focus on in a more informed way (in particular, when multiple valid non-deterministic options are possible). This is what [Dowek, 2001] calls *don't care* non-determinism.

In practice, this (among other things) is embodied by the application of prefactorizing rules (definition 7.5.3) to exhaustion (factorizable graph, definition 7.3.4) before applying any factorization rules, and the priority in the application between multiple factorization rules. This closely relates to what in higher-order unification are called *rigid-rigid*, *flexible-rigid* and *flexible-flexible pairs*. A *rigid* head is a head of a function application that contains no variables, whereas a *flexible* head is one that contains variables. Unifying two terms with rigid heads is deterministic and straightforward, and therefore efficient (and is exactly what zero factorization (§7.4.7.1) does). Unifying a flexible and a rigid term incurs non-determinism but in a limited manner, and also allows us to constrain the instantiation of the variables, so it is also desirable (this is what single factorization (§7.4.7.2) does). Unifying multiple flexible heads incurs a lot of non-determinism, essentially equivalent to enumerating potential instantiations of a variable (this is what multiple factorization (§7.4.7.5) does).

A graph being factorizable, when representing multiple unification equations possibly containing the same second-order variables, means that we can safely ensure we will solve all rigid-rigid (zero factorization) and flexible-rigid (single factorization) pairs (tuples, in fact, since these rules are defined for more than 2 incoming edges to a node) before solving flexible-flexible tuples, throughout the entire system. An important fact that underlies this property about factorizable graphs is that unifiers do not directly affect second-order variables (see chapter 5 for a clear description of why this is the

case). Being able to solve head pairs in this order is more efficient, as is well known in the higher-order unification literature.

7.7.2 Rewrite rules and their roles

When comparing dependency graph unification with higher-order unification, it is useful to consider what aspect each rewrite rule is related to. We describe this explicitly in this subsection.

- *Vertical monotony* and *vertical alignment* rules relate to the **propagation** of information between multiple unification problems. It does not have an equivalent in standard higher-order unification because that algorithm traditionally solves single unification problems.

- All *prefactorizing rules* except the occurs check and validating consistency, and half-factorization rules (§7.4.7.3, §7.4.7.4), relate to the *normalization of terms* to ensure that comparison between them is adequate and effective.

- The *occurs check* and *validate consistency* rule aim at **stopping the search** in situations that will yield no solutions. In particular, at avoiding infinite unproductive search. This is not necessary / ineffective in the standard higher-order unification, and the reasons behind this relate closely to the topics described in §7.7.4.

- Zero *factorization*, single factorization and multiple factorization are the actual unification rules that **produce instantiations** of second-order variables, and relate to rigid-rigid, flexible-rigid and flexible-flexible pairs in higher-order unification, as explained above.

7.7.3 Non-determinism

Higher-order and second-order unification, as well as the majority of their variations, are fundamentally *non-deterministic*. Unlike first-order unification, there is not a single *most general unifier*. An important classification of non-determinism (sometimes, like in [Dowek, 2001], called *don't care* and *don't know* non-determinism) is between non-determinism where the order in which the search space is explored does not change its results (don't care), and non-determinism where we have to explore multiple possible options independently (don't know).

In our algorithm, don't care non-determinism is reduced to the choice of which rule to apply to which nodes and edges within the priorities defined by the algorithm. Don't know non-determinism is embodied by explicit non-determinism in the result of rules. This is the most costly kind of non-determinism, and that's why in our algorithtm it is reduced primarily to single factorization and multiple factorization, which are applied at the lowest priority.

One family of optimizations of higher-order unification and second-order unification has to do with utilizing *unification schemata* instead of individual unifiers, to represent families of unifiers with a common structure. For example, in [Zaionc, 1987, Farmer, 1988]. We have not explored this or how it could affect dependency graph unification.

7.7.4 Unifiability versus explicit unifiers

One of the most important differences in scope, introduced already in chapter 5, between our algorithm and its application, and standard higher-order logic applications, is our need to produce *explicit and exhaustive instantiations* of second-order variables, as opposed to simply checking for unifiability. This is a major difference that conditions a lot of our decisions and the approach of the algorithm. Most or all of these have to do with Huet's lemma, establishing that a unification problem where all unification pairs are flexible-flexible always has at least one solution (in standard higher-order logic). See section 4.2.3 of [Dowek, 2001] for more details. When only unifiability is a concern, a flexible-flexible pair means acceptance of the problem as unifiable and stopping the search. However, because we are looking for all possible instantiations, we need to continue the search through this search space with a huge branching factor.

It is also important to understand that focusing on unifiability rather than finding explicit unifiers is a systematic feature of higher-order unification algorithms, rather than a choice that can be easily avoided. For example, when one instantiation does not work in our approach, it is not feasible to simply produce *a different one*: the higher-order unification algorithms work is conditioned to assume that this is not relevant and is exploited to guide the search itself, and cannot be easily modified with an "exception".

Some of the ways in which this aspect conditions our approach include:

- One of the reasons we choose to solve multiple unification problems at once is to

reduce the search space as much as possible in the case of flexible-flexible pairs.

- The propagation of information as much as possible before committing to instantiations, which is enabled by the graph approach, is primarily aimed at reducing or eliminating the need to explore flexible-flexible pairs.
- The prioritization of rules is also closely related to this last aspect. This would not be possible in a straightforward way without the graph data structure and the rewrite rule approach. See §7.4.7.
- Our particular concern and systematic approach to non-determinism is necessary due to the large amount of non-determinism generated by flexible-flexible pairs.
- The Validate Consistency, and specially the Occurs Check rules, are necessary / useful in our algorithm, and not present in standard higher-order unification, mainly due to the necessity to reduce unproductive search while preserving productive search.

7.8 Summary

This chapter formalizes dependency graphs as a tool to solve unification equation systems and the basic pieces needed to do so. It introduces a set of non-deterministic *solution preserving* rewrite rules on these dependency graphs, that change their representation while keeping the set of solutions they represent. We introduce a set of *normalization levels* for dependency graphs and use adapted versions of standard techniques in rewrite systems to show that the set of rewrite rules are, under certain conditions, confluent to the specified normalization levels, and that we can use these normalized graphs to produce or verify explicit solutions in a simple way. All of this comes together into an algorithm for producing sound and fair enumerations of the set of unification solutions of a unification equation system.

In the last section of the chapter, we describe the similarities and differences between our algorithm and standard higher-order unification, and how this relates to the nature of our problem and the specific decisions in the way our algorithm is designed.

Chapter 8

Implementation

I have produced an implementation in Haskell of most of the ideas presented in this thesis. This is a proof of concept implementation rather than a general tool ready for industrial use, and it includes a representation of ESQ logic, an implementation of all the patterns in the catalogue in appendixA, and most relevantly, an implementation of minimal commitment resolution for ESQ logic as described in chapter 5.

The entirety of this (large) implementation can be found at `https://github.com/Undeceiver/metaunif`. In this chapter we provide a very general overview of the pieces it consists of, while stopping to describe a few relevant details about it. An empirical evaluation of the implementation can be found on chapter 9.

8.1 Terms and unifier expressions

One of the most basic pieces in the program is the definition of the term and unifier expression structures. It is, however, a relatively simple piece with few complexities compared to the rest of the program.

One of the important aspects about the implementation of this part is that it is a very general and modular approach, in which many different pieces are implemented more or less independently and combined. For example, we define the general notion of a type being a "term structure" meaning that it can be built and unbuilt into a head and set of arguments. Both first-order terms, second-order terms and second-order atoms implement their own versions of this structure. Similarly, we have type classes indicating what we require of a type to be considered a "variable" in terms of what can be done with them that is relevant for the algo-

rithm. We then have four different implementations of this type class: first-order variables, second-order function variables, second-order predicate variables and unifier variables. Each has their own properties, but they share some structure which is used commonly in many places. There are many other such instances of this approach.

One of these common aspects is the *normalization* of terms and unifier expressions. Apart from having a common structural definition, we implement this individually for first-order terms, second-order terms, second-order atoms and unifier expressions. These correspond to the definitions of normal elements in these types defined in §6.1. The implementation is relatively easy and follows almost directly the formal definition of the rewrite rules.

8.2 ESQ logic

At the other side of the spectrum, at the highest level, is the implementation of ESQ logic, which expresses how to define ESQ queries in the program and translates them into a set of minimal commitment resolution for ESQ logic operations to run to find the solutions.

This layer is actually quite simple, most of the complexity being present in the lower layers of resolution and unification. The main concern is having the ability to properly express and differentiate the queries, and how each of them translates into combinations of applications of the refutation procedure.

One relatively tricky aspect of this layer is the difficulty in representing implicit combinations of queries and the corresponding applications of the refutation procedure while only explicitly running when it is necessary. This is closely related to the topics discussed in §8.5.

8.3 Resolution

In this layer we implement the process of producing and maintaining maximal CNFs as described in §5.5.1, as well as the resolution rule with implicit unification and generation of unification equations, as described in §5.5.2.

This mostly involves using a suitable data structure for CNFs that not only keeps track of current clauses and literals, but also which ones can be unified (possible unification steps), including groups of literals to do with factoring, as described in §5.5.2.

In relation to this, a relevant implementation aspect of automated theorem proving based on resolution for first-order logic is the choice of an *heuristic* for deciding which resolutions to apply at each moment, since it cannot be known beforehand which will lead to a proof. Good heuristics often reduce the running times of this process dramatically.

While the heuristics we have adopted in our implementation are rather simple (mainly, choosing the resolution step that will produce the clause with the least size), we have made sure to leave a clear space for this to be changed easily if necessary. Some references to literature on more up-to-date first-order automated theorem proving techniques can be found on §2.3.

A critical aspect of our particular implementation of the handling of maximal CNFs is that we have decided to completely forgo *inductive instantiation* of second-order variables, as described in §5.5.1. This means that the set of instantiations that our implementation will output is limited to atomic instantiations and no composite instantiations like conjunctions, disjunctions or negations. This does mean the resulting procedure is not complete in the sense of instantiations of second-order variables containing logical connectives. Atomic instantiations do, however, remain complete. There are three main reasons for this decision:

- Inductive instantiation greatly increases the size of the search space, and thus the running time, of the program.
- We argue that in most fault detection cases, the most interesting instantiations, or in fact sometimes the only ones that are relevant, are atomic ones. Generally speaking, whenever a pattern is interested in a certain structure, it will explicitly present it in the associated query, using separate atomic second-order variables. Inductive instantiation is still theoretically fundamental to present the completeness of the approach and consider alternative approaches, but it makes sense to avoid it.
- The real time required for me to develop this part of the program would be

relevant and given the time constraints of this PhD, I decided it was best to instead spend that time in other more core tasks.

8.4 Unification

This is without a doubt the largest, most complex and most core element of the implementation. It consists of several modules working together to implement the functionality of dependency graphs for solving systems of unification equations, as described in §5.5.3.

Apart from the underlying basic modules describing term structures as described in §8.1, and the handling of non-determinism and fair enumerations described in §8.5, there are mainly two modules that implement the unification algorithm itself: one concerned with dependency graphs as an abstract concept in terms of equivalence of elements, operations and correctness of the semantics of dependency graphs themselves; and another concerned with applying this abstract structure to the specific problem of finding instantiations of second-order variables that satisfy a set of unification equations.

The first one is a relatively standard (though still intricate) exercise in data structures, whereas the second one is more specific and implements things like the rewrite rule system described in chapter 7.

There are a couple of important aspects of this second module that are worth discussing, other than the transversal aspects discussed in §8.1, §8.5 and §8.7.

First, in chapters 5 and 7, when describing the rewrite rule system, we heavily rely on the notion of *levels of normalization* for dependency graphs, that help us understand the issue and show some of its most important properties. In practice, however, there are reasons to avoid explicitly describing these normalization levels in the implementation, for performance and simplicity reasons. Instead, what we do is we give a precise priority order to the application of rewrite rules, ensure that they are applied exactly when they are, in fact, applicable (as described formally in chapter 7), and use a queue to handle the set of rewrite rules that need applying. As we describe in chapter 9, this part turned out to be one of the most problematic in terms of performance, due to the unavoidable need to constantly check a lot of preconditions for rules.

This means that rewrite rules will only be applied to dependency graphs which

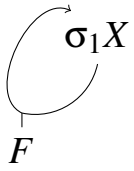
fulfill all preconditions for them, including their normalization level, even if this is not explicit. The correctness of this approach is embodied by theorems 7.5.1, 7.5.3, 7.5.4, 7.5.5 and 7.5.6, that connect the normal levels with the applicability of each of the families of rewrite rules.

Second, in §7.6.1 we discuss the issue that cycles in dependency graphs produce with the algorithm and why the occurs check is a theoretically complete solution to this, but also a practically unsatisfactory one[1]. As a consequence of this, and also the complexity of the occurs check rule both in development time and computational time, we decided to not implement it in full. Instead, we replace it with a check for cycles that gets flagged up in the algorithm, but not solved (the search is halted when a cycle is found). This allows us to notice the prevalence of this situation in our cases, to be able to take measures to correct it if necessary, without dealing with the aforementioned complexities. Clearly, this once again will make the algorithm fail in certain situations where theoretically it should not. We have, however, given reasons why this is not a big theoretical problem, and our unit tests show that it is almost never problematic in practice.

There is an important exception to this, however. A very specific and simple case of cycles in the graph is produced systematically by other rules. In particular, factorization rules work in a recursive way by instantiating second-order variables to partial compositions and introducing new variables. One of the fundamental base cases for this is when the composition cannot possibly have any more depth, which in practice means that a second-order variable needs to be instantiated to a projection, ending the recursion. The way this appears in the graph is by having an edge which has its target be the same as one of its sources (a cycle) and the head be the second-order variable. See figure 8.1. These cycles appear even when the initial graph is completely acyclic. Thus, in the implementation, we check for this exact specific case and solve it by instantiating the second-order variable to an adequate projection. Apart from this only working for unary cycles, this approach is also limited in that another possible instantiation would be to remove the dependency on some of the sources. Nonetheless, in our unit tests (see §8.7), every single case that did not have a fundamental cycle to begin with could be solved with this exception, while over half of them needed this exception to avoid failure.

[1] But we have shown why it cannot be relevantly improved upon.

Figure 8.1: A unary cycle produced by the application of factorization rules.



8.5 Non-determinism

Non-determinism is used extensively in both our theoretical description and our implementation of the algorithm. This is a transversal aspect, and our usage of it is among the most complex that can be conceived, for several important reasons:

- Most importantly, our search spaces are, in general, infinite. Moreover, the search trees that some aspects of our algorithm describes are not just infinite in depth, but also in width; meaning that from a single state an infinite number of different new states can be reached in just one step.
- We are very concerned with the fairness of the search procedure. Obviously, an infinite search space cannot be explored in finite time. However, enumerable search spaces can be attempted to be explored in a fair way, so that for each element within it, it can be reached within finite time.
- The order of the search and in general the width and depth of the different branches in the search tree can greatly affect the computational complexity (running time) of different parts of the algorithm.
- An overarching topic in our contributions in this thesis is the usage of *implicit representations* of large sets of solutions that we can do operations with, rather than explicitly finding solutions that we will never use. This needs to be explicitly implemented in the program, in ways that allow us to operate with implicit solutions and use them semi-transparently as if they were explicit solutions, while only explicitly evaluating them when necessary.

One big ally in this process is the fact that Haskell is a *lazily evaluated language*. This allows even simple list data structures in Haskell to be infinite and allowing computations with them. Lazy evaluation falls far from the topic of this thesis, but the fundamental idea is precisely that computations are represented implicitly by default

and only evaluated when the shape of the result is required to make decisions about the rest of the program (pattern matching).

However, both basic Haskell data structures (like lists) and existing libraries fall short of some of the behaviours and capabilities that we needed. Thus, we implemented three different modules to deal with three very closely related aspects of this problem:

- **Enumeration procedures** - This is a replacement data structure for lists that presents one additional capability and enhanced operations with much more attractive properties for our purposes.

  The additional capability is the ability to represent within the data structure itself the notion of a computation step that did not produce any result. This is fundamental in ensuring fairness, because it allows us to "quantize" infinite computations that never produce results into an infinite set of terminating steps that do not produce results. This prevents these unproductive search branches (that cannot, however, be distinguished beforehand), from holding the computation capabilities of the programs hostage and preventing other productive search branches from producing their results.

  Enhanced operations generally involve variations of usual list operations, and some additional operations, all of which explicitly ensure the fairness of the resulting list: each element in the result can be produced in finite time.

  One key operation in this module is the *diagonalization* of an enumeration procedure over a function. This is an explicit implementation of definition 7.6.7 and theorem 7.6.6. It also happens to be a *monadic* structure over the type of enumeration procedures that preserves fairness.

- **Non-deterministic algorithms** - This module builds on enumeration procedures to allow the implicit representation of non-deterministic computations, with plenty of fundamental and useful operations like composing them, combining them, currying and uncurrying functions, providing provenance for elements, etc.

  One key aspect of this module is the separation of the *definition* of the non-deterministic algorithm in terms of its *search space*, and the chosen *search order*. To be precise, an **algorithm** in this module represents a search space which only makes assumptions about the search order that are explicitly indicated when

physically building the algorithm data structure itself (and are not necessary at all). When trying to produce the set of results of an algorithm (which will be a fair enumeration procedure like expressed above), the chosen search order must be indicated, producing different enumeration procedures depending on it. Examples of search orders include depth-first search, breadth-first search, iterative deepening, diagonalization and other variations of these.

- **Answer sets** - In this module, we provide infrastructure to be able to handle implicit and explicit representations of sets of solutions (of any kind) opaquely. At a basic level, an answer set represents a conceptual set of solutions, but it may be expressed explicitly or implicitly (or partial combinations of these two), while providing operations for applying functions to these, combining them, enumerating them and building non-deterministic algorithms over them, that allow us to only make them explicit when it is absolutely necessary.

8.6 Dependency graph data structure

An aspect of the implementation that has turned out to be relevant for some of the considerations and limitations of this approach has been the way in which the dependency graph data structure is implemented. There are a series of aspects of what this data structure needs to represent and how it is used that play important factors in this:

- **Cyclicity** - Apart from the thoroughly discussed directed cycles, even on directed acyclic graphs (which is what is relevant for the abstract algorithm's purpose), we may have undirected cycles. For example, two edges coming out of a node to different targets, and then two edges coming out of these two different nodes and targetting another, same, final node.

  In Haskell specifically, there are two properties of the language that make these structures problematic to deal with naturally. First, *immutability* of data structures. Data structures can be efficiently pattern matched and reproduced, but one cannot modify a Haskell data structure in-place. Second, all Haskell data structures are fully acyclic by construction. Pointers or references to data structures are not something that Haskell provides to the user in a usable way. This is by design, and has to do with the aforementioned immutability of data structures.

  This means that in order to represent cyclic dependency graphs in Haskell, we

need to explicitly create a pointer/reference system and maintain it, while copying entire data structures often (instead of changing internal aspects of them).

- **Node merging** - One of the most important and common operations in our rewrite systems is *merging* two nodes, combining both the dependants they contain and the edges that go in and out from them. It is a challenge to make both the process of merging the nodes and the process of traversing the graph from nodes to edges to nodes to edges efficient. Our implementation is based on the usual union/find algorithm for representing disjoint sets [Tarjan, 1975], with particular details aimed at our particular problem.

- **Edge redundancy/deletion** - While dependants and nodes are, by definition, never removed from the graph (in fact, every dependant is implicitly in the graph from the beginning, but only explicitly added when we need to use it), edges can be marked as redundant. We still need to keep them in the graph, but we need to treat them differently. This is not a huge challenge, but it complicates thing a little more, and also adds to the algorithmic complexity of operations on the graph that require looking at edges.

- **Accessing nodes "by name"** - In many rewrite rules, and also when encoding unification equation systems into a dependency graph, we need to access nodes not based on their topological situation in the graph (which other edges/nodes they may be connected to), but rather based on the **dependants** they contain. For example, we often will want to find things like "the node that contains the dependant $\sigma_2\sigma_1 X$".

Our implementation of the dependency graph data structure was done with some of these abstract ideas in mind, but not specifically optimized for the particular ways in which the algorithm uses it. As such, it consists in about ten dictionaries with different types of keys and values. For example, a dictionary mapping first-order dependants to the first-order nodes in which they originally were. Another dictionary mapping first-order nodes to first-order nodes they were merged with, implementing the union/find aspect of the data structure. Another dictionary mapping each first-order node to a list of all the first-order dependants contained in it. Etc. Some local traversal exists, namely, each node contains direct references to all incoming and outgoing edges it has, and edges have direct references to all nodes involved in it. Keeping these local data structures updated, specially in an immutable language like Haskell, also proves

problematic.

This approach means that no individual operation will be particularly problematic, but it also means (as we discovered when attempting to evaluate the algorithm) that over large sequences of operations, a very large amount of lookups on dictionaries are necessary. Moreover, the immutability of data structures in Haskell means that dictionaries are less efficient to use than in other languages.

At the latter stages of this PhD, a profiling study of the performance of the program made me aware of the scale of this problem, consuming upwards of 70% of the program's running time over problematic test cases of between 10 and 30 seconds (that did not produce any results). I tried some optimizations, like using a better dictionary data structure or optimizing pointer chains every so often to reduce to more direct links, and while some improvement was noticed, none proved enough.

Later on (after the first submission of this thesis), and as we discuss in chapter 9, a more profound analysis of this problem and its relevance was carried out, and it turned out to be one of the fundamental problems with the computational performance of the implementation. The fact that it is related to Haskell's limitations gives some cause for hope that a re-implementation in a different language that allowed more efficient data structures could improve the computational properties of the algorithm (though there are no guarantees of this).

I discuss this further in chapter 9, but overall, it is a possibility that an implementation of the dependency graph data structure and its basic operations that is more carefully tailored to the particular ways in which the rewrite rule system handles them, making operations more local and streamlined and avoiding "global" searches over the graph so often; and using a different language that does not make dealing with mutable and cyclic data structures so hard, could greatly boost the performance of the program. I also imagine there are other performance-deteriorating factors in the implementation that I did not get the chance to observe due to this being the largest one.

This is an important and challenging problem with the implementation and the approach, and it is unclear whether it could be sufficiently improved upon so as to change the conclusions of the evaluation in this thesis.

8.7 Unit tests

It should be noted that, while not all, most aspects of this program have extensive sets of unit tests that we have used to verify correctness throughout changes in the way the program works. More relevantly, the two main modules of the unification aspect have seen the most changes and variations, and sets of unit tests for these, while not entirely complete, have proven extremely useful.

The following is a summary of the extent and purpose of the unit tests contained in `https://github.com/Undeceiver/metaunif`:

Test file	**Module under test**	**# Tests**	**Summary of tests**
AnswerSetTest.hs	Implicit/explicit answer sets	2	Simple check that implicitly and explicitly defined answer sets can be accessed in the same manner.
CesqLogicTests.hs	ESQ logic queries	29	Tests of simple and composite queries returning the adequate instantiations on simpler and more complex theories.
DependencyGraphTests.hs	Dependency graph data structure	6	Very high-level simple tests to check that building and modifying dependency graphs produces adequate data structures.

EnumProcTests.hs	Infinite lists / enumeration procedures with lazy evaluation that can be combined preserving lazy evaluation, with void steps for possibly non-terminating computations.	6	Simple tests verifying that this data structrure can be used to combine multiple operations with potential infinite, non-productive computation processes that can be then explored lazily without running into non-productive infinite rabbit holes.
ESUnificationTests.hs	Unification dependency graph operations	428	Exhaustive tests on each individual operation type of unification graphs in multiple different circumstances of varying complexities.
ESUnificationExtraTests.hs	Unification dependeny graph operations	2	Some additional high-level tests for the complete operation of unification dependency graph logic.
HeuristicsTest.hs	Heuristics for search procedures	4	Basic checkes that the heuristics defined are being followed in searches utilizing the defined types.
MetaLogicTests.hs	Highest level of term definitions	38	Verification that parsing and combination of terms defined using the module work properly.
ObjectLogicTests.hs	Intermediate level of term definitions	87	Verification that parsing and unification of terms defined using the module work properly.

SOResolutionTests.hs	Resolution from second-order equations to instantiations using dependency graphs	50	Exhaustive verification that the correct instantiations are produced in different systems of equations of varying complexities and particularities.

8.8 Summary

I have produced a Haskell implementation of the ideas included in this thesis. This follows quite accurately the theoretical descriptions included in this text. Some variations have been introduced from the theory, mostly to do with performance optimizations or reasonable simplifications of the theory. Some of these do affect the results produced by the algorithm but I have taken measures to place adequate boundaries around these.

The core and most detailed modules of the implementation have to do with the unification procedure via dependency graphs and minimal commitment; but also with the non-determinism present throughout the entirety of the ideas in this thesis.

Important issues with performance arose, which are discussed further in chapter 9. I discovered some of these have to do with the implementation of the dependency graph data structure and the properties of the Haskell programming language. I was not able to overcome these, and they present a difficult challenge for the feasibility of the algorithm that is unclear if it could be overcome.

Chapter 9

Evaluation

In this chapter I first describe the evaluation methodology followed and then present the actual results produced. While not entirely independent, the methodology is presented relatively independently of the results, so as to describe not only the actual results that I obtained, but also those that I did not, those that I would have wanted to, and a few details about the story of how I encountered these during the development of the program and the PhD.

There are two main (related) research hypotheses that we evaluate in this thesis:

Hypothesis 1. minimal commitment resolution for existential second-order query logic *(the algorithm/procedure described in chapter 5 and more formally defined in chapters 6 and 7) is a sound, complete and computationally feasible implementation of existential second-order query logic.*

Some clarifications:

- By *sound* we mean that all solutions output by the algorithm/procedure are members of the answer set of the input query (for all valid queries and theories).
- By *complete* (also sometimes referred to as *fair*) we mean that all elements of the answer set of the input query are output as solutions by the algorithm/procedure after finite running time (for all valid queries and theories).
- By *computationally feasible* we mean that the running times and memory consumption numbers for the algorithm are such that it outputs relevant solutions to small inputs based on application cases in a reasonable time.

Note that we talk about *computational feasibility* but not about *complexity* in a theoretical sense. While in principle it could be considered, the complexity of theorem proving in first-order logic (which is a much simpler problem than the one presented here) is known to be at least exponential. Therefore, theoretical complexity results for this particular problem are likely to be of little relevance for real world applications. We could offer empirical complexity approximations that would be more relevant for the scope of this algorithm, but we have several reasons to forgo a systematic approach to this as well:

- There are no other algorithms or procedures that solve similar problems to compare to.
- The implementation and running context are still experimental and unoptimized, so these results would likely not be reliable.
- The performance will depend strongly on the specific examples, and thus empirical performance or complexity would be measured with respect to a standard benchmark, which does not currently exist, and the production of which is not part of this PhD.
- The results of most inputs to the algorithm are infinite sets of solutions that are output sequentially. Thus, performance or complexity measurements should account for the dependency on how many and which solutions are output. This increases the uncertainty of the measurements further.

This contrasts with the properties of *soundness* and *completeness* in this hypothesis, which are used in the usual and formal sense. We prove these theoretically in chapters 6 and 7, but we also evaluate them with respect to the actual implementation of the algorithm in the current chapter. It is worth stating that we have not formally verified the implementation and therefore it is possible that implementation errors have made it incorrect, even if the theoretical algorithm has been proven correct.

Definition 9.0.1. Meta-ontology fault detection *is the framework by which we encode abstract and common sense patterns (originating either from independent ontology development research, agreed upon good and bad practices or intuition) formally (including a precise definition of what is and what is not an instance of a pattern) and use* automated *methods to detect instances of these patterns.*

That is, the framework described in chapter 4.

Hypothesis 2. *Meta-ontology fault detection, used by encoding patterns in* existential second-order query logic*, has the potential to be an effective and feasible approach to detecting common faults in ontologies formalized in first-order logic.*

A few clarifications are due:

- By *effective* we mean both that the method *detects the faults that it was designed to detect*, and in particular that it has *high precision* and *high recall*. We define these terms more precisely below.
- By *high precision* we mean that a large proportion of the initial[1] potential fault instances output by the system are considered to indeed be faulty by a trusted oracle[2], and that this proportion may drop as we continue into the latter potential faults output by the system, but will do so at a progressive and not excessively fast pace.
- By *high recall* we mean that a large proportion of the fault instances provided by a trusted oracle will be eventually detected by the system, and that the distribution of these detected faults leans noticeably towards the initial potential fault instances output by the system, with progressively smaller proportions being found in latter outputs.
- By *feasible* we mean that the running times, memory consumption numbers and the *specificity* of fault patterns provided remain low, and within the limits of what is technically viable for realistic results in the current time and age.
- By *specificity* of fault patterns, we refer to the degree to which a pattern is able only to detect specific faults in specific ontologies, or by contrast, is able to detect multiple faults in different ontologies (low specificity).

There are a few still largely subjective terms used here, such as “large proportion”, “trusted oracle”, “progressive and not excessively fast”, “leans noticeably towards”, “progressively smaller proportions”, “low”, “within the limits of what is technically viable and realistic”, etc. This is done on purpose, and rather than trying to provide objective (but more likely misleading or confusing, and definitely arbitrary) measures

[1] i.e. the ones output first by the system

[2] In our case, this will consist in labeling performed by myself and collaborators *before* running the evaluation tests (based on the original research sources of the encoded patterns), but also on a posteriori qualitative evaluation of the outputs to reflect on their actual faultiness.

for these, we acknowledge their subjectivity here and make a point of keeping the qualitative and quantitative aspects of the evaluation as clearly separate as possible.

In other words, the evaluation of this hypothesis will be unavoidably qualitative in a large proportion. This is due to several factors, the most important of which are:

- The lack of previous work with the same scope, and standard benchmarks to evaluate it, or even evaluation sets that could be applied to this work.

- The still experimental and initial nature of this work. We are not providing a tool, nor prescribing a specific methodology. We are investigating the feasibility of a conceptual approach. Limitations and issues of this first attempt do not necessarily indicate fundamental limitations and issues of the approach, and a qualitative approach will work better to identify directions for future work and differentiate them from intrinsic properties.

Moreover, the *effectiveness* of the meta-ontology fault detection approach is closely connected with the pattern catalogue as described in chapter 4. This means that as part of this evaluation we will evaluate the framework in general, the pattern catalogue and the specific implementation as much independently as working together. We describe this in more detail in §9.1.

Finally, a word about the reasons for dividing the research hypothesis into two separate (and possibly at a glance similar) hypotheses. The main difference between the two hypotheses is that hypothesis 1 is about the **technical** properties of an **algorithm**, whereas hypothesis 2 is about the **pragmatic** properties of a **framework**. Their connection is that the algorithm is the way that we have developed to implement the framework. As such, some properties of the framework will inevitably depend on the properties of the algorithm (e.g. computational feasibility), specially when evaluated empirically. However, the algorithm is evaluated mostly for its *correctness*, whereas the framework is evaluated mostly for its *usefulness*. Moreover, there are other potential uses of the algorithm and parts of it, and other potential implementations of the framework and parts of it (such as the pattern catalogue). This is why we have chosen to evaluate them separately.

9.1 Evaluation methodology

At this point, we need to look ahead a bit and explain some aspects of the actual results of the evaluation to properly understand the evaluation methodology and how it relates both to the hypotheses and the results.

My original and ideal intention was to perform an evaluation based on a moderate number of clearly defined test cases with different patterns and ontologies, run by the program, and focus on different aspects of these results to evaluate the different aspects of the hypotheses as described before.

Indeed, in preparation for this, I produced a large document (large enough not to include it even as an appendix of this thesis) containing detailed and technical descriptions of these test cases (`https://tinyurl.com/y67nsebs`). Moreover, I partially implemented all of these test cases as automated tests in Haskell.

However, as it turned out, the current implementation is still unfortunately mostly *computationally infeasible*. Only a very small proportion of the test cases in this test suite actually produced any results at all within reasonable running times (in the order of hours or days). The details of this are described in detail in §9.3, §9.4 and chapter 10, but at this point it is relevant in two ways:

1. While we will present and describe this evaluation test set, it has not been used in the way that we anticipated.

2. Due to the difficulties with using this test set, we have produced other alternative ways of empirically evaluating the *effectiveness* of the meta-ontology fault detection framework. These are described in the following.

3. The *soundness* and *completeness* of minimal commitment resolution for existential second-order query logic has been successfully demonstrated in several ways. First, through the theoretical proofs in chapters 6 and 7. Second, even though the larger test set could not be systematically used, smaller unit tests have been used throughout the entire development (as described in chapter 8; the full tests can be found on the GitHub repository linked in that chapter), that have shown the correctness of the implementation in producing the expected results in each case; as well as a small proportion of the large test set that was indeed computationally feasible and produced exactly the expected results (discussed further in §9.3 and §9.4).

9.1.1 Extensive pattern test cases

We now describe how we would have evaluated each of the aspects of the two hypotheses **empirically**[3], should we not have had the computational feasibility challenges discussed, or how we would, should we be able to, in the future, produce a better version of the approach that overcomes these challenges:

- Hypothesis 1:
  - **Soundness** - All the solutions output by the algorithm within the time given to it to run will be manually verified to be actual solutions to the formal query presented to the algorithm.
  - **Completeness** - The expected solutions presented pre-evaluation in this document will be verified to be within the solutions output within the time given to the algorithm to run.
  - **Computational feasibility** - This will be evaluated in multiple ways **for each test case**.
    * The expected solution is output within the time given to the algorithm to run (yes/no).
    * We will take measurements of the times it takes the algorithm to present each sequential solution, and provide summaries of these, but perform no systematic analysis of this.
- Hypothesis 2. To evaluate this hypothesis, we will not only run each of the test cases, but also do crossover runs of patterns from some test cases on the theories of other test cases. This will allow us to more accurately evaluate the *precision*, *recall* and, more importantly, *specificity* of these patterns when used outwith the theory they were designed for:
  - **Precision** - For each of the solutions output for each of the runs of the algorithm, including crossover runs, we will manually check (and judge) which can be considered to be actual faults in the ontology, as intended to be targetted by the pattern.

[3] As described, some of them are also proven theoretically in chapters 6 and 7

- **Recall** - The expected solutions presented pre-evaluation in this document will be verified to be within the solutions output within the time given to the algorithm to run.
- **Computational feasibility** - This will be measured the same way that computational feasibility is measured for hypothesis 1, but it will be applied to the crossover runs as well.
- **Specificity** - This will be evaluated by comparing the precision, recall and computational performance values of the intended theory / pattern pair with those of the crossover runs. Similar results would indicate lower specificity (which is desirable), whereas different results would indicate higher specificity.

In the large test set, we present individual test cases with different expected results. We will, however, try to provide multiple test cases for the same pattern with different situations that would likely give rise to different potential problems and concerns for the algorithm. This is knowingly done based on relatively arbitrary criteria, i.e. my own judgement; due to the lack of better, more standard criteria to prepare these tests. This is a known and long standing problem in the field where this works stands, and the approach followed is the usual way to deal with it.

Therefore, even with a computationally feasible evaluation, statistic measurements on the tests would be largely uninformative and definitely inconclusive. Anyone potentially reutilizing this benchmark may choose to do statistics, but should be aware of the source of the data and the challenge that it presents to the validity of statistic measurements on these tests. Instead, we will focus on qualitative analysis of the results and extraction of rational explanations of the behaviour observed.

For each test case we will present the following pre-evaluation information:

- Background theory in which the pattern will be detected (a set of first-order logic axioms).
- Additional contextual information. While this will also be part of the theory, it is worth separating it as this will often contain second-order functions and predicates, and the source of this information in a practical implementation would be an external (simple) analyzer of the theory and the way it is expressed, separate from the theory itself.

- Pattern to be detected (am ESQ[4] query).
- A set (often just a single element) of instances of the pattern that an algorithm should detect.
- Additionally, we may present comments on the source of the test, its relevance, importance and whatnot.

As already mentioned, note that we do not include in this thesis nor in its appendices the entirety of the test suite, due to its length. Therefore, it should be understood that the set of test cases itself is not a contribution that this thesis should be assessed on. Instead, the actual contributions in this section are, on the one hand, the approach followed for the preparation of the patterns based on the existing research on good ontology practices and, on the other, the evaluation results.

We will present here a couple of example test cases to showcase the approach and the format, with the remnant being available at `https://tinyurl.com/y67nsebs`.

9.1.2 Pattern completeness and specificity against original research

An alternative way to qualitatively evaluate the *effectiveness* of the meta-ontology fault detection framework is to take the patterns from the pattern catalogue and go back to the original research which motivated them, looking at the examples presented there, and evaluating:

1. Whether the pattern would correctly identify those faults in the original ontologies they come from. This is to be understood qualitatively and not automatically. That is, I will provide informal, semi-formal and formal (e.g. manually produced examples of algorithm runs) arguments, using the original research's language as much as possible, to justify why the framework and its implementation would correctly identify those faults should the computational feasibility challenge be overcome.
2. The specificity of the produced patterns. I will do this by comparing fault families and types in different pieces of research and showing how some of our patterns would work for all of them, or would in fact improve on many of them.

[4]Existential second-order query logic

Clearly, the validity of this evaluation method is less than ideal; due to its qualitative and manually produced nature. However, the arguments produced here can be adequately **reproduced** and discussed by other researchers, including the proposers of the original fault detection approaches. Moreover, in the face of the computational feasibility challenge to our current implementation, these arguments provide other researchers a way to understand the potential capabilities of the approach should this challenge be overcome. In other words, while less than ideal, we consider this evaluation method useful and relevant, which is the reason we have performed it and included it in this thesis.

9.2 Pattern test case examples

In order to illustrate the nature of these test cases, we include here a couple of examples. However, for reasons of space, and as explained, we only include two example test cases, and one of its corresponding crossover tests, the rest (26 base tests and 52 crossover tests) being available at `https://tinyurl.com/y67nsebs`.

9.2.1 SpicyTopping pattern test case

In this case, the class spicyTopping should not be primitive, but defined as any topping that has spiciness in it. This is meant to be detected by using the pattern that there should not be a primitive class that is independently subsumed by two different primitive classes.

- **Theory** -

$$\begin{aligned}
&\forall X.spicyTopping(X) \implies (pizzaTopping(X) \wedge (\exists Y.hasSpiciness(X,Y) \wedge spicy(Y))) \\
&\forall X.meatTopping(X) \implies pizzaTopping(X) \\
&\forall X.spicyBeefTopping(X) \implies (meatTopping(X) \wedge spicyTopping(X) \\
&\forall X,Y.hasSpiciness(X,Y) \implies (pizzaTopping(X) \wedge spiciness(Y)) \\
&\forall X,Y.hasTopping(X,Y) \implies (pizza(X) \wedge pizzaTopping(Y))
\end{aligned} \tag{9.1}$$

- **Contextual knowledge** -

$$
\begin{aligned}
&primitive(spicyTopping)\\
&primitive(meatTopping)\\
&primitive(spicyBeefTopping)\\
&primitive(pizza)\\
&primitive(spiciness)\\
&explicit\_property(hasSpiciness)\\
&explicit\_property(hasTopping)
\end{aligned}
\tag{9.2}
$$

- **Full theory (CNF)** -

$$
\begin{aligned}
&\neg spicyTopping(X) \vee pizzaTopping(X)\\
&\neg spicyTopping(X) \vee hasSpiciness(X, x(X))\\
&\neg spicyTopping(X) \vee spicy(x(X))\\
&\neg meatTopping(X) \vee pizzaTopping(X)\\
&\neg spicyBeefTopping(X) \vee meatTopping(X)\\
&\neg spicyBeefTopping(X) \vee spicyTopping(X)\\
&\neg hasSpiciness(X,Y) \vee pizzaTopping(X)\\
&\neg hasSpiciness(X,Y) \vee spiciness(Y)\\
&\neg hasTopping(X,Y) \vee pizza(X)\\
&\neg hasTopping(X,Y) \vee pizzaTopping(Y)\\
&primitive(spicyTopping)\\
&primitive(meatTopping)\\
&primitive(spicyBeefTopping)\\
&primitive(pizza)\\
&primitive(spiciness)\\
&explicit\_property(hasSpiciness)\\
&explicit\_property(hasTopping)
\end{aligned}
\tag{9.3}
$$

- **Pattern** -

$$\begin{aligned}&((P,Q,R) \models^{*} (\exists X.P(X) \wedge \neg Q(X)) \wedge (\exists X.Q(X) \wedge \neg P(X))) \bowtie \\ &\bowtie ((P,Q,R) \models (\forall X.R(X) \implies P(X)) \wedge (\forall X.R(X) \implies Q(X)) \bowtie \\ &\bowtie ((P,Q,R) \models_{M} primitive(P) \wedge primitive(Q) \wedge primitive(R))\end{aligned} \tag{9.4}$$

- **Target instantiations** -

  1.

$$\begin{aligned}P &= spicyTopping \\ Q &= meatTopping \\ R &= spicyBeefTopping\end{aligned} \tag{9.5}$$

9.2.2 ProteinLoversPizza pattern test case

In this example, the proteinLoversPizza class is inferred as subsumed by vegetarianPizza. This is due to a faulty definition of proteinLoversPizza, in which the logical "and" is used instead of the logical "or", resulting in the entailment that proteinLoversPizzas must not have any toppings. This is detected by using the pattern that subsumptions with universal property restrictions (all toppings in a vegetarian pizza must be vegetables) that are only satisfied trivially (proteinLoversPizza satisfies this because it cannot have any toppings) are generally faulty.

A universal property restriction establishes that whenever a property (binary relation) between two elements happens, and the first element is of a certain class, then the second element must be of another certain class. In this instance, whenever the relation *hasTopping* happens between a vegetarian pizza and a topping, then that topping has to be of the class $(\neg meatTopping \wedge \neg cheeseTopping \wedge \neg seafoodTopping)$, representing any toppings that are not meat nor cheese nor seafood. In first-order logic, this would look like:

$$\begin{aligned}&\forall x,y.(vegetarianPizza(x) \wedge hasTopping(x,y)) \implies \\ &\quad (\neg meatTopping(y) \wedge \neg cheeseTopping(y) \wedge \neg seafoodTopping(y))\end{aligned} \tag{9.6}$$

- **Theory** -

$$\begin{aligned}
&\forall X.proteinLoversPizza(X) \iff (pizza(X) \wedge (\forall Y.hasTopping(X,Y) \implies \\
&\quad (meatTopping(Y) \wedge cheeseTopping(Y) \wedge seafoodTopping(Y)))) \\
&\forall X.vegetarianPizza(X) \iff (pizza(X) \wedge (\forall Y.hasTopping(X,Y) \implies \\
&\quad (\neg meatTopping(Y) \wedge \neg cheeseTopping(Y) \wedge \neg seafoodTopping(Y)))) \\
&\forall X.meatTopping(X) \implies \neg cheeseTopping(X) \\
&\forall X.meatTopping(X) \implies \neg seafoodTopping(X) \\
&\forall X.cheeseTopping(X) \implies \neg seafoodTopping(X) \\
&\forall X,Y.hasTopping(X,Y) \implies (pizza(X) \wedge pizzaTopping(Y)) \\
&\forall X.meatTopping(X) \implies pizzaTopping(X) \\
&\forall X.cheeseTopping(X) \implies pizzaTopping(X) \\
&\forall X.seafoodTopping(X) \implies pizzaTopping(X)
\end{aligned} \tag{9.7}$$

- **Contextual knowledge** -

$$\begin{aligned}
&univ_class_prop_restriction(vegetarianPizza, hasTopping, \\
&\quad (\neg meatTopping \wedge \neg cheeseTopping \wedge \neg seafoodTopping)) \\
&univ_class_prop_restriction(proteinLoversPizza, hasTopping, \\
&\quad (meatTopping \wedge cheeseTopping \wedge seafoodTopping)) \\
&primitive(pizza) \\
&primitive(meatTopping) \\
&primitive(cheeseTopping) \\
&primitive(seafoodTopping) \\
&primitive(pizzaTopping) \\
&explicit_property(hasTopping)
\end{aligned} \tag{9.8}$$

- **Full theory (CNF)** -

$$
\begin{aligned}
&\neg proteinLoversPizza(X) \vee pizza(X)\\
&\neg proteinLoversPizza(X) \vee \neg hasTopping(X,Y) \vee meatTopping(Y)\\
&\neg proteinLoversPizza(X) \vee \neg hasTopping(X,Y) \vee cheeseTopping(Y)\\
&\neg proteinLoversPizza(X) \vee \neg hasTopping(X,Y) \vee seafoodTopping(Y)\\
&\neg pizza(X) \vee proteinLoversPizza(X) \vee hasTopping(X,y(X))\\
&\neg pizza(X) \vee proteinLoversPizza(X) \vee\\
&\quad \neg meatTopping(y(X)) \vee \neg cheeseTopping(y(X)) \vee \neg seafoodTopping(y(X))\\
&\neg vegetarianPizza(X) \vee pizza(X)\\
&\neg vegetarianPizza(X) \vee \neg hasTopping(X,Y) \vee \neg meatTopping(Y)\\
&\neg vegetarianPizza(X) \vee \neg hasTopping(X,Y) \vee \neg cheeseTopping(Y)\\
&\neg vegetarianPizza(X) \vee \neg hasTopping(X,Y) \vee \neg seafoodTopping(Y)\\
&\neg pizza(X) \vee vegetarianPizza(X) \vee hasTopping(X,y_2(X))\\
&\neg pizza(X) \vee vegetarianPizza(X) \vee\\
&\quad meatTopping(y_2(X)) \vee cheeseTopping(y_2(X)) \vee seafoodTopping(y_2(X))\\
&\neg meatTopping(X) \vee \neg cheeseTopping(X)\\
&\neg meatTopping(X) \vee \neg seafoodTopping(X)\\
&\neg cheeseTopping(X) \vee \neg seafoodTopping(X)\\
&\neg hasTopping(X,Y) \vee pizza(X)\\
&\neg hasTopping(X,Y) \vee pizzaTopping(Y)\\
&\neg meatTopping(X) \vee pizzaTopping(X)\\
&\neg cheeseTopping(X) \vee pizzaTopping(X)\\
&\neg seafoodTopping(X) \vee pizzaTopping(X)\\
&univ\_class\_prop\_restriction(vegetarianPizza, hasTopping,\\
&\quad (\neg meatTopping \wedge \neg cheeseTopping \wedge \neg seafoodTopping))\\
&univ\_class\_prop\_restriction(proteinLoversPizza, hasTopping,\\
&\quad (meatTopping \wedge cheeseTopping \wedge seafoodTopping))\\
&primitive(pizza)\\
&primitive(meatTopping)\\
&primitive(cheeseTopping)\\
&primitive(seafoodTopping)\\
&primitive(pizzaTopping)\\
&explicit\_property(hasTopping)
\end{aligned}
\tag{9.9}
$$

- **Pattern** -

$$
\begin{aligned}
&((P,Q,R) \vDash \\
&\quad (\forall X.P(X) \implies Q(X)) \wedge \\
&\quad (\forall X.P(X) \implies \neg \exists Y.R(X,Y))) \bowtie \\
&\bowtie ((Q,R) \vDash \\
&\quad \exists X,Y.Q(X) \wedge R(X,Y))
\end{aligned}
\tag{9.10}
$$

- **Target instantiations** -

1.

$$
\begin{aligned}
P &= proteinLoversPizza \\
Q &= vegetarianPizza \\
R &= hasTopping
\end{aligned}
\tag{9.11}
$$

9.2.3 ProteinLoversPizza / Primitive subsumption cycles pattern test case

This is a crossover test from the two tests above, using the pattern from the first test case in the theory of the second test case. The intention here is to see how patterns perform in ontologies they are not designed for.

- **Theory** -

$$\begin{aligned}
&\forall X.proteinLoversPizza(X) \iff (pizza(X) \wedge (\forall Y.hasTopping(X,Y) \implies \\
&\quad (meatTopping(Y) \wedge cheeseTopping(Y) \wedge seafoodTopping(Y)))) \\
&\forall X.vegetarianPizza(X) \iff (pizza(X) \wedge (\forall Y.hasTopping(X,Y) \implies \\
&\quad (\neg meatTopping(Y) \wedge \neg cheeseTopping(Y) \wedge \neg seafoodTopping(Y)))) \\
&\forall X.meatTopping(X) \implies \neg cheeseTopping(X) \\
&\forall X.meatTopping(X) \implies \neg seafoodTopping(X) \\
&\forall X.cheeseTopping(X) \implies \neg seafoodTopping(X) \\
&\forall X,Y.hasTopping(X,Y) \implies (pizza(X) \wedge pizzaTopping(Y)) \\
&\forall X.meatTopping(X) \implies pizzaTopping(X) \\
&\forall X.cheeseTopping(X) \implies pizzaTopping(X) \\
&\forall X.seafoodTopping(X) \implies pizzaTopping(X)
\end{aligned} \tag{9.12}$$

- **Contextual knowledge** -

$$\begin{aligned}
&univ_class_prop_restriction(vegetarianPizza, hasTopping, \\
&\quad (\neg meatTopping \wedge \neg cheeseTopping \wedge \neg seafoodTopping)) \\
&univ_class_prop_restriction(proteinLoversPizza, hasTopping, \\
&\quad (meatTopping \wedge cheeseTopping \wedge seafoodTopping)) \\
&primitive(pizza) \\
&primitive(meatTopping) \\
&primitive(cheeseTopping) \\
&primitive(seafoodTopping) \\
&primitive(pizzaTopping) \\
&explicit_property(hasTopping)
\end{aligned} \tag{9.13}$$

- **Full theory (CNF)** -

$$
\begin{aligned}
&\neg proteinLoversPizza(X) \vee pizza(X)\\
&\neg proteinLoversPizza(X) \vee \neg hasTopping(X,Y) \vee meatTopping(Y)\\
&\neg proteinLoversPizza(X) \vee \neg hasTopping(X,Y) \vee cheeseTopping(Y)\\
&\neg proteinLoversPizza(X) \vee \neg hasTopping(X,Y) \vee seafoodTopping(Y)\\
&\neg pizza(X) \vee proteinLoversPizza(X) \vee hasTopping(X,y(X))\\
&\neg pizza(X) \vee proteinLoversPizza(X) \vee\\
&\quad \neg meatTopping(y(X)) \vee \neg cheeseTopping(y(X)) \vee \neg seafoodTopping(y(X))\\
&\neg vegetarianPizza(X) \vee pizza(X)\\
&\neg vegetarianPizza(X) \vee \neg hasTopping(X,Y) \vee \neg meatTopping(Y)\\
&\neg vegetarianPizza(X) \vee \neg hasTopping(X,Y) \vee \neg cheeseTopping(Y)\\
&\neg vegetarianPizza(X) \vee \neg hasTopping(X,Y) \vee \neg seafoodTopping(Y)\\
&\neg pizza(X) \vee vegetarianPizza(X) \vee hasTopping(X,y_2(X))\\
&\neg pizza(X) \vee vegetarianPizza(X) \vee\\
&\quad meatTopping(y_2(X)) \vee cheeseTopping(y_2(X)) \vee seafoodTopping(y_2(X))\\
&\neg meatTopping(X) \vee \neg cheeseTopping(X)\\
&\neg meatTopping(X) \vee \neg seafoodTopping(X)\\
&\neg cheeseTopping(X) \vee \neg seafoodTopping(X)\\
&\neg hasTopping(X,Y) \vee pizza(X)\\
&\neg hasTopping(X,Y) \vee pizzaTopping(Y)\\
&\neg meatTopping(X) \vee pizzaTopping(X)\\
&\neg cheeseTopping(X) \vee pizzaTopping(X)\\
&\neg seafoodTopping(X) \vee pizzaTopping(X)\\
&univ\_class\_prop\_restriction(vegetarianPizza, hasTopping,\\
&\quad (\neg meatTopping \wedge \neg cheeseTopping \wedge \neg seafoodTopping))\\
&univ\_class\_prop\_restriction(proteinLoversPizza, hasTopping,\\
&\quad (meatTopping \wedge cheeseTopping \wedge seafoodTopping))\\
&primitive(pizza)\\
&primitive(meatTopping)\\
&primitive(cheeseTopping)\\
&primitive(seafoodTopping)\\
&primitive(pizzaTopping)\\
&explicit\_property(hasTopping)
\end{aligned}
\tag{9.14}
$$

- **Pattern** -

$$\begin{aligned}&((P,Q,R) \vDash^{*} (\exists X.P(X) \wedge \neg Q(X)) \wedge (\exists X.Q(X) \wedge \neg P(X))) \bowtie \\ &\bowtie ((P,Q,R) \vDash (\forall X.R(X) \implies P(X)) \wedge (\forall X.R(X) \implies Q(X)) \bowtie \\ &\bowtie ((P,Q,R) \vDash_{M} primitive(P) \wedge primitive(Q) \wedge primitive(R))\end{aligned} \tag{9.15}$$

9.3 Results

In this section we lay out the results of running the evaluation methodologies described in the previous sections. We will provide some context and a basic way of understanding the results, but the analysis of the meaning and validity of these and their relation with the research hypotheses is discussed in §9.4.

9.3.1 Pattern automated test cases

Unfortunately, one of the primary results of this part of the evaluation is that, in its current implementation, the algorithm for **minimal commitment resolution for existential second-order query logic is computationally infeasible**. This is clear from the fact that out of the 26 base test cases described, only 1 of them finished with results within reasonable time. While we did not try all 26 test cases individually, and we did not try any of the 52 crossover test cases at all (since we would expect even worse running times for these), we tried the ones that were potentially simplest, and only test number 20 (discussed in the following) produced results (in fact, it produces the first result extremely quickly, in the neighbourhood of 1 second on my computer).

The following is the test case that produced correct results quickly:

9.3.1.1

- **Theory** -

$$\forall X.falls(X) \iff waterfall(X) \tag{9.16}$$

- **Contextual knowledge** -

$$\begin{aligned} &primitive(falls) \\ &primitive(waterfall) \\ &equivalent\_classes(falls, waterfall) \end{aligned} \tag{9.17}$$

- **Full theory (CNF)** -

$$\begin{aligned} &\neg falls(X) \vee waterfall(X) \\ &\neg waterfall(X) \vee falls(X) \\ &primitive(falls) \\ &primitive(waterfall) \\ &equivalent\_classes(falls, waterfall) \end{aligned} \tag{9.18}$$

- **Pattern** -

$$\begin{aligned} &((P,Q) \models primitive(P) \wedge primitive(Q)) \bowtie \\ &\bowtie ((P,Q) \models_M equivalent\_classes(P,Q)) \end{aligned} \tag{9.19}$$

- **Target instantiations** -

1.

$$\begin{aligned} P &= falls \\ Q &= waterfall \end{aligned} \tag{9.20}$$

As it can be noticed, this test case is particularly simple. It utilizes three very straightforward axioms in the theory, and uses **only contextual information** (as we shall explain later in this section, this has turned out to be the fundamental factor why this test case is so fast), with no transitive inference. Nonetheless, it involves second-order variable instantiation and multiple chained queries, which indicates that the fundamental approach can be adequate in at least some cases.

Another relevant (though, as it turns out, not critical) factor in this query is that it only uses entailment queries (translated into unsatisfiability queries), rather than satisfiability queries, which are computationally semi-decidable and approximated in a much more computationally costly way. The issues with satisfiability queries are explained in more detail in chapter 5 and more formally in §6.2.

By contrast, consider what we consider to be the simplest test case that we could not get the program to run properly in reasonable time:

9.3.1.2

- **Theory** -

$$
\begin{aligned}
&\forall X.(\exists Y.hasTopping(X,Y) \wedge edamTopping(Y)) \implies \\
&\quad \neg(\exists Y.hasTopping(X,Y) \wedge mozzarellaTopping(Y)) \\
&\forall X.(\exists Y.hasTopping(X,Y) \wedge mozzarellaTopping(Y)) \implies \\
&\quad \neg(\exists Y.hasTopping(X,Y) \wedge edamTopping(Y)) \\
&\forall X.fourCheesePizza(X) \implies (pizza(X) \wedge \\
&\quad (\exists Y.hasTopping(X,Y) \wedge mozzarellaTopping(Y)) \wedge \\
&\quad (\exists Y.hasTopping(X,Y) \wedge edamTopping(Y)) \wedge \\
&\quad (\exists Y.hasTopping(X,Y) \wedge cheddarTopping(Y)) \wedge \\
&\quad (\exists Y.hasTopping(X,Y) \wedge parmezanTopping(Y)))
\end{aligned}
\tag{9.21}
$$

Note that the first two axioms are equivalent, and therefore redundant. This is not obvious at first glance, it was not to me until I converted them both to CNF and realized they were equal. Thus why I leave them both in the non-CNF version of the theory. I only include it once in the final CNF version.

- **Contextual knowledge** -

$$
\begin{aligned}
&primitive(edamTopping) \\
&primitive(mozzarellaTopping) \\
&primitive(cheddarTopping) \\
&primitive(parmezanTopping)
\end{aligned}
\tag{9.22}
$$

- **Full theory (CNF)** -

$$\begin{aligned}
&\neg hasTopping(X,Y) \vee \neg edamTopping(Y) \vee \neg hasTopping(X,Z) \vee \neg mozzarellaTopping(Z)\\
&\neg fourCheesePizza(X) \vee pizza(X)\\
&\neg fourCheesePizza(X) \vee hasTopping(X, y(X))\\
&\neg fourCheesePizza(X) \vee mozzarellaTopping(y(X))\\
&\neg fourCheesePizza(X) \vee hasTopping(X, y_2(X))\\
&\neg fourCheesePizza(X) \vee edamTopping(y_2(X))\\
&\neg fourCheesePizza(X) \vee hasTopping(X, y_3(X))\\
&\neg fourCheesePizza(X) \vee cheddarTopping(y_3(X))\\
&\neg fourCheesePizza(X) \vee hasTopping(X, y_4(X))\\
&\neg fourCheesePizza(X) \vee parmezanTopping(y_4(X))\\
&primitive(edamTopping)\\
&primitive(mozzarellaTopping)\\
&primitive(cheddarTopping)\\
&primitive(parmezanTopping)
\end{aligned} \tag{9.23}$$

- **Pattern** -

$$\begin{aligned}
&((P) \models \neg\exists X.P(X)) \bowtie\\
&\bowtie ((P) \models_M primitive(P))
\end{aligned} \tag{9.24}$$

- **Target instantiations** -

1.

$$P = fourCheesePizza \tag{9.25}$$

This test case also has a very simple pattern, with no satisfiability queries. However, it has a slightly more complicated theory, and more importantly, **includes a (simple) query with a second-order variable in the form of a first-order formula**. In the following I discuss what the relevant differences between these two test cases are, and where the line between tractable and intractable seems to lie for examples of my algorithm. I also discuss some issues on the design of the algorithm that are particularly difficul to deal with, that have to do with the computational feasibility problems.

9.3.2 Detailed profiling and debugging of an intermediate test case

In an attempt to better understand the computational feasibility issues of the program, I identified a variation of the example in §9.3.1.1 that would still produce results, but takes a relevant amount of time to produce the first. This example is still extremely simple but presents a fundamental difference from the original (§9.3.1.1): it includes first-order formulas.

Here is the description of this example (we focus on the CNF form exclusively due to the technical focus of this section). I note this example contains *no second-order variables*, and thus the task consists merely in proving the specified formula using my algorithm:

9.3.2.1

- **Full theory (CNF)** -

$$\begin{aligned} &\neg falls(X) \vee waterfall(X) \\ &\neg waterfall(X) \vee falls(X) \\ &primitive(falls) \\ &primitive(waterfall) \\ &equivalent\_classes(falls, waterfall) \end{aligned} \tag{9.26}$$

- **Pattern (CNF)** -

$$((\,) \vDash falls(X) \wedge \neg waterfall(X)) \tag{9.27}$$

The intended proof is very simple, and consists in resolving the pattern with the first clause in the theory twice, each of them eliminating one of the literals, to obtain the empty clause.

I should note that my algorithm does produce a correct solution for this case in approximately 11 seconds on my computer. This runtime is extremely high for such a simple case, and my exploration of it shows interesting data that we believe explain the problems with the current implementation of the algorithm.

Before going into the details, however, let me note that the debugging task in itself was not only challenging and time consuming, but also problematic. The main

reason behind this is related to my decision to use Haskell as programming language to implement the program. As I explained in chapter 8, there were two main attractive properties of Haskell behind this decision:

1. Abstract structures like terms and unifier expressions are easy to encode and work with.

2. *Lazy evaluation* helps implement non-determinism in effective ways.

One well known downside of Haskell as a programming language, though, (see, for example, [Ennals and Jones, 2003, Schilling, 2011]) is that it is incredibly difficult to debug. This is a fundamental issue with the approach of the language and has no singular and simple solution. Haskell is difficult to debug mainly because of its laziness and because its computations have no side effects (which makes introducing observers / manipulators in the middle of computations extremely hard). I shall note that it is **not** as simple as using monads (intended to encode side effects in computations in a way that respects the philosophy of the language) to bypass this difficulty. Monads require declaration in the data types of all functions that utilize them, and while tools like monad transformers intend to ease up this process, they by no means remove it or provide a systematic solution. This is well known in the Haskell community. In other words, simple debugging approaches that in other languages may involve adding a line of code, in Haskell may involve changing the type signature of your entire program and introducing high-level instructions to ensure that the computation is produced in the order that we want / expect it to happen (so as to avoid lazy evaluation producing non-representative results). This is particularly hard as well when our program, by design, requires lazy evaluation to function properly, and the computations in our program involve many layered functions of different type with more and less generic types that weave with each other.

The relevance of these debugging issues to the results presented here are that the results are not as complete and not as conclusive as I would like. However, I believe they show clear patterns of issues that I can relate to the implementation of the program and that explain the performance issues.

The main target of this exercise is to locate what the main sources of performance bottlenecks are. To this end, we measure the running time (in real time) of both the whole program up until the first solution is produced, and exclusively the unification

part (that means, the dependency graph algorithm, excluding the resolution search aspect of the proof (we explain why this is relevant later.)). We also include traces that tell us how many times a non-deterministic branch happens, and how many times each of the dependency graph rules are run (and how many of those are aborted (meaning explained later)). Here is the data:

- 10.988 seconds total runtime.
- 5.691 seconds runtime exclusively unification.
- 578 non-deterministic branches.
- At least 497 branches happen exclusively during resolution (**before** any unification takes place).
- 4 sets of unification equations that would solve the problem are found.
- 1583 applications of the vertical monotony of explicit equivalences rule (number of abortions not tracked).
- 1583 applications of the vertical monotony of syntactic equivalences rule (number of abortions not tracked).
- 1583 applications of the vertical monotony of horizontal edges rule (number of abortions not tracked).
- 6504 applications of the vertical alignment rule (number of abortions not tracked).
- 279 applications of the second-order zip rule (number of abortions not tracked).
- 2718 applications of the first-order zip rule (of which 2457 were aborted).
- 182 applications of the projection simplification for first-order edges rule (number of abortions not tracked).
- 1760 applications of the projection simplification for second-order edges rule (number of abortions not tracked).
- 1051 applications of the (simplified) occurs check rule (number of abortions not tracked).
- 182 applications of the second-order function dumping rule (number of abortions not tracked).

- 1760 applications of the first-order function dumping rule (all 1760 of which were aborted due to finding no edges incoming to the edge's head).
- 2 applications of the zero factorization rule.

This information says a lot when processed adequately, as I will try to do in the following.

First, note the distinction between the resolution and unification parts of the program. This is explained in detail in chapter 5. In principle, it would have been sensible to think that the performance issues have to do with bad resolution heuristics that are leading to unproductive search and applications of the resolution rule. However, the data shows that at least half of the running time is spent in the unification aspect of the program, which should be the simplest in this example. Therefore, even if resolution search was optimized, the performance issues would not be reduced in a relevant way[5]. **Thus, we can conclude the problem is not primarily related with the resolution search and instead lies in dependency graph unification.**

Second, another sensible initial idea for potential performance issues is unnecessary non-deterministic branching that would incur too much parallel search without reaching deep enough solutions. This would have already been strange given the simplicity of the example, but it was an important aspect to consider. However, out of the 578 branches produced in the example, at least 497 of them happen before any dependency graph unification takes place, and entirely in the resolution search part of the program. This, combined with the knowledge explained in the previous paragraph stating that most of the delay does not happen in the resolution search, says that **the problem is not primarily related with non-deterministic branching.**

Having discarded those two principal suspects, we look at the data to find what it can tell us about the real culprit. One thing that stands out is that the number of rule applications is considerably large for such a simple example. One of the main reasons behind this can be found when looking at the number of abortions of the two rules for which I tracked abortion rates[6] (first-order zip, of which 2457 out of 2718 (90.4%) were

[5]Note that the rest of the data shows that the scaling of the unification part of the program has issues, so it is not reasonable to expect that this would remain constant or near constant in larger examples.

[6]These were the two rules with most surprising numbers when considering the properties of the particular example and when rules should be run.

aborted; and first-order function dumping, of which all 1760 (100%) were aborted). I should explain what an *abortion* means in this case.

The rewrite rules for dependency graphs as described in chapter 7 are defined in terms of a series of pre-conditions on a local part of the graph. The termination, productivity and fairness properties of the algorithm are proven using measures on the graph that are reduced / increased with each rule, ensuring certain convergence properties. However, at no point in the theoretical description of the algorithm did we specify an exact moment in which to decide to run each specific rule. Thus, in the implementation, we achieve complete and correct application of the rules through two mechanisms (briefly described in chapter 8):

1. A *priority queue* of rules to be run on the graph, from which rules are *pulled* one at a time.

2. To *push* rules into the queue, I use a *local* but *complete* approach: after each successful rule application, I add to the priority queue every possible rule that could have not been applicable before but may be applicable now after the changes in the graph. It is only when the rule is pulled to be run that the complete set of pre-conditions for its applicability are checked. I note that these pre-conditions need to be explicitly checked and cannot be simply assumed from the local context, since a lot of them might have to do with parts of the graph that are unrelated to the rule that just finished application. Thus, checking for these pre-conditions **before** pushing into the priority queue would simply be bringing forward this computation, and not actually avoiding any.

   When a rule is pulled from the priority queue and its pre-conditions **fail**, the rule is *aborted*.

Thus, the large number of *abortions* in the rules measured (which are the most relevant ones in this case, and can be safely assumed by construction to apply to most other rules), indicates not that a lot of rules *need* to be run in the graph, but rather that a large number of *potential rule applications* happen, and the program spends a lot of time *checking for their applicability*.

It is important to note at this point that a large number of rule applications does not necessarily imply a large amount of runtime spent, specially when it's in the order of thousands, which is still not, in this day and age, a relevantly large number of computations, if they are small. Are they small?

The answer to this question comes when we dig a bit deeper into profiling the program. While Haskell debugging is difficult and problematic, it does have an automatic profiler. This profiler can also be problematic (in this instance, for example, the generated file has over 50000 lines of different *cost centres*), but with patience and clever investigation, some insights can be extracted. The following is an extract of the information produced by the profiler for this example, indicating the top 5 *cost centres* of the program, in terms of runtime spent in them:

Cost centre	**% runtime**
ghashWithSalt:40:5-36	6.7%
ghashWithSalt:36:5-62	6.1%
gtraceM:(34,1)-(35,25)	4.5%
hashWithSalt:227:5-38	3.7%
runESUStateTOp:(862,1)-(872,244)	3.3%

I note that these cost centres amount to 24.3% of the total runtime of the program; and the top 20 cost centres amount to 54.2% of the total runtime of the program.

As described, Haskell's debugging issues make this information hard to process and understand, but let's try.

One first conclusion coming from the distribution of runtime cost of the program is that there is no single fundamental part of the program that is consuming most of the time. This generally means that the most basic and fundamental optimization of performance have already been done and it is more abstract and subtle optimizations that need to be done, or total reworks of the approach.

From the five top cost centres, number four is `gtraceM`, which is the debugging function we used to count the number of rule applications. Thus, we can ignore this line since while relevant, it is not fundamentally altering the performance of the program, and it's not present when not debugging.

The other 4 are functions run at the low levels of the dependency graph data structure handling. It is hard to check, but I tried to get an accurate approximation of the number of times the first cost centre is called, and the number for the execution of this example lies somewhere between 500,000 and 2,000,000.

Why so many? When each of the rewrite rules for the graph are run (even if aborted! the checking of pre-conditions also incurs a lot (most) of this), we use operations on the dependency graph data structure that involve looking up hash maps multiple times to

locate the particular nodes/edges that we are talking about. This can build up due to the merging and renaming of nodes and edges that may make finding a specific node/edge that we are looking for involve quite a few look ups.

In other words, the data presented here together shows that it is very likely that the main reason for the performance issues of the program is that the graph data structure:

1. Is accessed too many times even for simple cases, due to the need to constantly check pre-conditions for rules even when they will not be applied.
2. Each time it is accessed, and due to its design, the complexity of the algorithm and some difficulties having to do with Haskell's inability to use actual pointers and instead having to rely on hash maps, we incur a considerable amount of hash map lookups.

These two factors combine to produce extremely high numbers of basic operations being run even for extremely simple examples, which, as the profiling shows, make up a large proportion of the program's runtime. Note that most of the other cost centres with relevant runtime costs, not presented here, are also related to this data structure access.

As a conclusion thus far, we can say that **it is the design of the dependency graph data structure and the properties of how the algorithm needs to access it that are responsible for the largest proportion of the performance issues of the program**. In section §9.4 and chapter 10 we discuss in larger depth what this means in terms of the value of the algorithm and what potential solutions or paliatives could and could not be implemented.

9.3.3 Qualitative evaluation of patterns on original research examples

The full results of applying the pattern catalogue to the set of examples present in the original research from which we produced the pattern catalogue can be found in appendix C. Here we summarize this information.

All the patterns would be able to detect the specific examples which motivated them. Moreover, the majority of the examples appearing in the literature used to produce the catalogue would be covered by at least one fault pattern. There is, however, a relevant

portion of examples that were not covered by any. These are generally related to one of two situations:

- The fault is fundamentally about a mismatch between the preferred model of the author or authors of the ontology and the actual model the ontology represents. The model the ontology represents is perfectly coherent in all senses, but just is not what the author had in mind. These cannot be detected with pure semantic methods like the ones we propose here.
- The fault has to do with naming conventions and systems, or other purely formal elements such as the particular order or structure in which elements are defined. We actually cover some of these faults with some of our patterns, by using *contextual information* to detect it, but specially when it comes to naming conventions, our approach is completely unprepared to tackle this, essentially by design.

Moreover, a lot of the examples from the literature of these two kinds could be argued to not be faults by other authors (including myself). These often boil down to a subjective opinion on what is the best way to define ontologies, rather than to actually incorrect ontologies.

When it comes to the overlap, generality and specificity of the patterns, we note that there are some patterns that cover plenty of examples (for example, unsatisfiable classes, or subsumption cycles). There are also families of patterns that have a very similar nature but apply to slightly different cases of related faults. This could signal a lack of generality of the approach, but it could also signal that a better, more general pattern could exist that we have not found that would encompass all of these sub-patterns.

9.4 Analysis

In this section we focus on utilizing the results described in §9.3 to reach a conclusion on the research hypotheses presented.

Hypothesis 1. minimal commitment resolution for existential second-order query logic *(the algorithm/procedure described in chapter 5 and more formally defined in chapters 6 and 7) is a sound, complete and computationally feasible implementation of existential second-order query logic.*

- **Soundness** - First, we note that in chapter 7 we provide a *formal proof* of this. Moreover, in all the *unit tests*, the specific *evaluation test case* that the algorithm produced proper results for, and in every case that did not produce results, this property is respected. In other words, no instance of unsoundness was found in the implementation.

  Thus, we conclude that **Minimal commitment resolution for existential second-order query logic is a sound algorithm**.

- **Completeness** - We provide some *formal proof* of certain slightly restricted versions of this in chapter 7. Empirical evaluation did not show clear issues with this, but failed to produce sufficient evidence, due to the computational issues.

  Thus, we conclude that **minimal commitment resolution for existential second-order query logic is probably a complete algorithm, under certain restrictions**, but not enough evidence has been produced to adequately ascertain the practical implications of said restrictions.

- **Computational feasibility** - The current implementation of the algorithm is *clearly not computationally feasible*. While some ideas for re-implementations of the algorithm or changes in approach are presented in this thesis that could improve this situation, it is still a reasonable conclusion that the fundamental idea of the algorithm is inherently computationally problematic.

  Thus, we produce the following two conclusions:

  – The **current implementation of minimal commitment resolution for existential second-order query logic is computationally infeasible**.

  – Some **relevant challenges remain to be overcome for any potential implementation of minimal commitment resolution for existential second-order query logic**, and it is **unclear whether these challenges can be overcome or not**.

Hypothesis 2. *Meta-ontology fault detection, used by encoding patterns in* existential second-order query logic*, has the potential to be an effective and feasible approach to detecting common faults in ontologies formalized in first-order logic.*

- **Effectiveness - High precision** - In all of our empirical tests and all examples qualitatively considered, the issues have mostly been with not detecting enough faults, rather than with false positives. Precision has not been a problem at any point.

- **Effectiveness - High recall** - While the results are generally favourable in this regard, empirical evidence of the program is lacking due to the computational feasibility issues. The qualitative analysis suggests good but definitely improvable and limited recall, meaning that this approach would either not be complete or would need much better patterns to be complete.

- **Effectiveness** - Thus, we conclude that **meta-ontology fault detection is a moderately effective approach to detecting common faults in ontologies**, with open avenues for relatively easy improvement to be explored.

- **Feasibility - Computational** - In the current implementation, the approach is clearly computationally infeasible. We are not currently aware of alternative algorithms to implement the meta-ontology fault detection framework that would be adequate. Similarly, while we have identified the main challenges in the current algorithm's computational infeasibility, as we discussed the potential solutions for these require further investigation and it is unclear whether they could be overcome or not.

- **Feasibility - Specificity** - The current pattern catalogue and the qualitative evaluation of their application to the original research suggests a moderate degree of generality, which is positive and is one of the main attractive aspects of this approach as opposed to the approaches suggested by the original research. Room for improvement remains, however, with some patterns being very similar but applying to different instances, and some patterns seeming quite specific to some instances.

- **Feasibility** - Thus, we conclude that **meta-ontology fault detection is a moderately general approach to detecting common faults in ontologies, is more general than previously existing approaches, and has room for easy improvement in the generality aspect**. However, the fundamental challenge to this approach is that **meta-ontology fault detection currently does not have an algorithmic implementation that has been shown to be computationally feasible**.

We further analyze and discuss the implications of these results in chapter 10.

9.5 Summary

Our two research hypotheses evaluate the power and the feasibility of both the minimal commitment resolution for existential second-order query logic algorithm, and the more general meta-ontology fault detection approach. The former, from a purely **technical** point of view. The latter, from a **pragmatic** point of view.

The main way in which we aimed to evaluate these research hypotheses was with a moderately sized test suite of faults in small ontologies to be run through the algorithm with our pattern catalogue. Unfortunately, the current implementation of the algorithm has proven to be computationally unable to produce results for most of these. All the results produced were correct and expected, but not enough results were produced.

An alternative qualitative way to evaluate the pattern catalogue and the pragmatic aspects of the meta-ontology fault detection framework was to analyze every example present in the original research that motivated the patterns in the catalogue, and discuss whether the catalogue would or would not be able to detect the faults described in them, should the algorithm be computationally feasible. We did that, with moderately good results, but with some room for improvement.

As a result of all of this, on top of the theoretical results already formally proven about the algorithm, we conclude that the algorithm is sound and complete under certain restrictions, but has important computational feasibility challenges; and it is unclear whether they could be overcome.

Similarly, we conclude that the meta-ontology fault detection framework is moderately effective and practically feasible, except for the computational feasibility challenges related to our current implementation of the algorithm supporting it.

Chapter 10

Conclusions

I started this PhD with the goal of understanding the state of the research in ontology debugging, and in particular automated approaches to the detection and repair of faults in logical ontologies, and exploring new and ideally more general approaches, mechanisms and solutions in this field.

Ontology engineering is still not a large mainstream field, and researchers and practitioners typically agree in some of the most relevant obstacles for a more extensive adoption. Finding effective methodologies, techniques and tools for *ontology debugging*, both in terms of preventing, detecting or correcting human error, but also in terms of providing support for other computational processes like ontology merging and ontology alignment, is one of these obstacles. The difficulties are large and too many to even properly summarize at once, let alone overcome.

Thus, almost all of the existing research in this field focuses on smaller sub-problems or simplified versions, and/or remains at a purely explorative level of qualitatively and informally understanding some aspects of these difficulties.

Once I understood this context, my main direction of work became to try to provide some generality, systematicity and a more unified approach to the field. Not so much to solve the entire problem (which I knew would be impossible within the constraints of a PhD), but rather to hopefully contribute to a slowly growing basic foundation to the particular subfield of ontology debugging.

To this end, one of the most useful revelations that I had was that different pieces of research on ontology debugging did not even use the same language or formal methods at all. Mutual awareness and communication is there, but there is no foundation, other than the foundations of the larger field of ontology engineering itself (which are, in fact,

very solid themselves at this point, but fail to capture all the intricacies of the ontology debugging subfield).

This is what motivated me to analyze different pieces of existing research in ontology debugging from a more abstract and theoretical point of view, and to try to extract some common themes, approaches and ideas in a way that could be expressed systematically and formally. The hope was that this could also lead to algorithmical and/or methodological tools that had a greater scope than the previously existing research.

Looking back, and even though the results have been mixed and often disappointing, I have no doubt that this was an avenue of work that was worth exploring. I believe (and I believe I have given good arguments for others to believe) that some of the results contained in this thesis are strong, useful and worth building on. Among these I include:

- The formalism of existential second-order query logic.
- The fundamental ideas behind the meta-ontology fault detection framework.
- The pattern catalogue as an initial and partial attempt at an explicit general approach to ontology debugging.
- The theoretical principles behind the dependency graph unification for ESQ logic algorithm.
- The theoretical results about the algorithm proven from the aforementioned theoretical principles.
- The systematic and well-grounded approach to the handling of infinite and infinitely branching search spaces and the implementation of an independent Haskell library to handle this.

However, I must acknowledge that other results have been disappointing and signal at the very least important challenges with some of the ideas proposed, if not outright presenting a fundamental issue with them. Specifically:

- The generality of the meta-ontology fault detection approach and its technical implementation through ESQ logic is attractive and moderately effective, but does not fully or most adequately capture every relevant aspect of ontology debugging. *It is not entirely general.*

- There are important and difficult challenges to finding a computationally feasible implementation of meta-ontology fault detection. I do not feel confident producing a prediction as to whether these challenges may be overcome or not. I have provided specific ideas and directions for exploring this, but it also may end up being an insurmountable problem. *Automation is currently computationally infeasible and it is not clear whether it would ever be feasible.*

Nonetheless, these two conclusions were not evident before this PhD, and I believe the identification of these challenges to be a relevant positive contribution of the PhD.

This dual result of positive contributions but important challenges is best expressed through our analysis of the two research hypotheses, as provided in chapter 9. Some aspects of the research hypotheses have a clear and rotund positive answer, some have a moderate but promising balance, and some are blocked by large and difficult problems.

The work carried out was always meant to be mostly exploratory. The implementation was never meant to end up as a readily available tool for practitioners, the pattern catalogue was not meant to be complete, etc. In §10.2 we discuss this in more detail, but we feel there are three main types of directions for future work from this PhD, each of them with abundant topics in them:

- Attempting to overcome the difficult challenges to the approaches and methods produced, such as the computational properties of the implementation of minimal commitment resolution for ESQ logic.

- Further developing and consolidating moderately successful results, such as extending the pattern catalogue or refining ESQ logic as a formalism to express fault patterns.

- Applying some of the successful results to other fields. For example, extending dependency graph unification to full higher-order logic and comparing it with classical higher-order unification as described by Huet (see chapter 3).

That there would be such open ended results was known since the beginning of the PhD, and I believe that it is in identifying them and giving useful and solid notions on how to continue pursuing them that the contributions of this PhD mainly lie in.

It it thus most appropriate, and in relation to the notion, expressed before, of contributing to a foundation and systematic methods for ontology debugging, that I

clearly describe in the following sections the relation of the work done in this PhD and the conclusions extracted from it to both existing work by other researchers, and directions for future work.

10.1 Related work

I first provide context on related work, how it differs from the work presented in this thesis and how the ideas in either of them can inform the other in relation to future work.

10.1.1 Meta-ontology fault detection

The most obvious body of work to compare this PhD to is the original research from which we extracted the fault patterns [Poveda-Villalón et al., 2010, Poveda-Villalón et al., 2012, Rector et al., 2004, Prince Sales and Guizzardi, 2017], as well as similar work in ontology debugging [Kindermann et al., 2019, Balaban et al., 2015, Blomqvist, 2010, Copeland et al., 2013, Gkaniatsou et al., 2012, Guarino and Welty, 2009, Hammar and Presutti, 2017, Haverty, 2013, Lambrix and Liu, 2013, Markakis, 2013, Mikroyannidi et al., 2012, Pinto et al., 2009, Poveda Villalón, 2016]. I have already discussed this quite thoroughly throughout the thesis, but I can conclude this comparison here by incorporating the evaluation results.

Our approach is, by design, more general than those bodies of work. To the best of our understanding, none of the cited works aim or attempt to introduce a generic approach for detecting ontology faults, and rather focus on particular families of faults, sometimes offering detection methods, some of which are automated. Our approach has a unified formalism to describe the patterns (ESQ logic), and a single algorithmic approach to obtain results from them. Moreover, an important pro of our more general approach is the ability to understand the theoretical and abstract properties of the specific patterns in a less context-dependent and isolated setting.

However, it is clear both that our approach did not fully capture everything that those approaches were designed to capture (the ones that we had time to implement as patterns), and more importantly, that our approach is currently computationally infeasible, whereas those other approaches are, in one way or another, practically applicable (even if with limitations in many cases). Thus, I would argue that, at least in part, our approach is more interesting in the grand scheme of things and worthy of

further work, but in short term applications is insufficient and the previously existing approaches remain superior.

The fact that most previous approaches were less general, and the challenges that we found in ours are not independent facts. The previously existing approaches were more constrained by design, because the authors were aware of the existence of some of the difficulties that I have faced, and the necessity to understand some particular aspects about the nature of ontology faults before pursuing such a general approach. My choice was to pursue the more general avenue because I felt it would be a valuable contribution at this point, knowing well that it would involve some added difficulties and challenges.

As an exception to this, the work in [Bundy and Mitrovic, 2016, Urbonas, 2019, Urbonas et al., 2020] (aimed towards repair in the face of an already detected fault), all of which is performed within my research group (and the last two of which I am a co-supervisor/co-author in) do offer a general and systematic approach. However, while this work is closely related to mine, mainly to the overlap of people working in it, it focuses on a different aspect of the problem (repair instead of detection), so comparisons would not make much sense: this work works from a *preferred structure* that states the desired statements that the ontology should infer, and its scope is limited to notions of *incompatibility* (false facts that are inferred) and *insufficiency* (true facts that are not inferred), with little room for more flexible definitions of fault.

At a more conceptual level, another field with obvious similarities is belief revision (see §2.2.1). However, there are important diferences. First, work in belief revision is done mostly under the assumption that maintaining consistency is fundamental. Our approach considers more general notions of faultiness and how to detect them. Second, and more importantly, belief revision is concerned with the process of *updating* the knowledge base *when new information is found*, and with the properties of these update operations. In meta-ontology fault detection, we work within the ontology to find errors in it, relying on the fault patterns and with no newly introduced information to work from.

Another field of work to compare the meta-ontology fault detection framework to, mentioned before in chapter 2, is automated bug detection and *bad smells* in software engineering. The situation in that subfield is similar to the one I described for ontology debugging, at least when it comes to automated methods. Software engineering is a larger and more established field than ontology engineering (which could be said to be

a subfield of software engineering in practice), and debugging in software engineering is to the date mostly handled through both methodological approaches and systematic ways to enhance those methodological approaches. For example, the usage of type safe programming languages, software modeling methodologies, programming techniques like inversion of control or the ever present notion of modularity are just some examples of this. These notions are also applied in ontology engineering, but mainly due to the far reaching inference mechanisms of logical ontologies and how these interact with bugs, automated methods seem more attractive. Automated methods to bug detection and bad smell detection in software engineering are very similar to the ones used in ontology debugging, being incomplete and/or ad hoc for the most part, with no or few well established general approaches.

Moreover, the application of the principles of meta-ontology fault detection to automated detection of bugs or bad smells is not absurd, since programs can be interpreted as logical ontologies containing facts. However, we have not explored the details of this at all, so I cannot offer any relevant insights on this avenue of work, other than it seems as unexplored as its application to ontology debugging.

10.1.2 Minimal commitment resolution for ESQ logic

The application of a pattern-based and inference-backed approach to automated detection of faults in ontologies is novel to the best of our knowledge. While a relevant proportion of the research discussed in the previous section uses the notion of pattern or similarly conceptualized terms in it, these are detected either manually or in a **purely morphological manner**: they do not exploit the inference properties of the ontology itself and the embedding of patterns within it to find these patterns. In other words, our framework detects *inferred* patterns, and not just explicit or semi-explicit ones. This was part of the design of our approach and one of the strengths I believe it has. However, it is also in large measure related to the reasons of the computational difficulties we have faced.

The relation between the technical problem of automatically detecting instantiations of ESQ queries and existing technical approaches is discussed at great length in §5.4. There we concluded there was no obvious path of application of those approaches to our problem, though we identified potential avenues of future research in adapting

some of that work to our problem, including higher-order theorem provers, SMT and ASP solvers with restricted versions of ESQ logic.

Moreover, technically speaking, minimal commitment resolution for ESQ logic is closely related and to a degree inspired by higher-order unification [Huet, 1975]. The main differences between them (described in much more depth in §5.3, §5.4.1 and §7.7) being:

- The ability to find all instantiations of patterns.
- The usage of dependency graphs as a way to prioritize and guide the search for unifiers.

One of the most promising directions for future work, that we will discuss in more detail in §10.2, is precisely the transportation of the two added capabilities of our algorithm to usual higher-order unification, whether by utilizing dependency graphs if the fundamental limitations to them can be overcome, or by translating the ideas to the usual ways in which higher-order unification is implemented.

One of the extensions of first-order unification that our dependency graph unification algorithm introduces is that of *unifier variables*. This concept is fundamentally the same as that of *binders* in the *nominal unification* literature [Urban et al., 2004, Schmidt-Schauß et al., 2019, Levy and Villaret, 2010, Calvès, 2013]. Nominal unification is usually concerned with unifying equations containing binders, as a general, higher-order problem. As such, it concerns itself with complex concepts such as *freshness constraints* that explicitly indicate disequality relationships between regular variables in a unification problem. Effectively, this makes the scope of binders in terms of variables and the way in which the general unification problem relates to the binders explicit. Ultimately, the primary problems of conventional unification that nominal unification concerns itself with are preserving α-equivalence when introducing explicit binders in the language and the related issues of variable *capture* and variable *renaming*. However, while these issues do appear in our problem, their semantics are very much grounded in those of first-order unification: minimal commitment resolution for existential second-order logic is explicitly defined as an extension of first-order resolution (and thus first-order unification) in which we **delay** unification by making it implicit and solving it using the dependency graph unification algorithm; but

every solution to one of such unification problems results in an *ordered sequence of first-order unifiers*, and thus not only is the scope of unifier variables (binders) clearly defined, but it is also simple, sequential, and our algorithm heavily relies and utilizes these semantics in its definitions. In contrast, the primary applications of full power nominal unification are for programming languages in which binding operations are an explicit and core aspect of the language, rather than a byproduct of the unification algorithm itself, and thus have much more general semantics. It is possible, though, that there may be something to learn from a more thorough grounding of unifier variables in dependency graph unification in nominal unification terms. Some results may be simplified and some aspects of our algorithm streamlined. Direct application of general nominal unification algorithms to our problem is unlikely to be effective, however.

Another important connection is to bounded second-order unification[Schmidt-Schauß, 2004]. Late in this PhD (in fact, after the viva), we discovered that in principle this algorithm could be applied to our problem. However, as described in §5.4.2.1, it is likely that this would have computational feasibility issues of its own, with the bounded second-order unification algorithm being provably NP-hard. Nonetheless, it is an avenue worth exploring.

Finally, the technical approach followed in the implementation of our dependency graph unification algorithtm is a particular case of *term graph rewriting* (§2.3.3, §3.4.1). The exploration of this field happened mostly in the post-viva modifications of this thesis, and it seems like most results in it have to do with establishing intuitive results such as the soundness, completeness and termination/confluence properties of the algorithms, with no big surprises. We have proven these results for our particular algorithm. It is possible, however, that a more thorough analysis and reformulation of the unification dependency graph algorithm and the theorems proven in chapter 7, from the point of view of the term graph rewriting literature, could yield some useful insights into the algorithm and its properties, though it is very unlikely it would change the fundamental results that we have discussed.

10.2 Future work

Having laid out the positive and negative results of the work in this PhD and their relation to existing approaches, I can now formulate the main ways in which I think it

can be further explored in the future.

10.2.1 Minimal commitment resolution for ESQ logic

We will first discuss the most obvious line of future work: improving the algorithm and attempting to tackle some of the computational challenges. While some direct further exploration into this, in the form of more profiling, better data structure design, heuristics and across-the-board optimization of the implementation would be warranted[1], at this point I believe that perhaps a more effective approach would be to take the principles and theoretical algorithm described in this thesis and embedding them into an existing theorem prover that already has a lot of the optimization work done, perhaps iteratively with small and clearly separated layers building on top of each other. This would not only save a lot of the optimization work to be done in the current implementation, but also help debug and identify a lot better the issues with the approach, by reducing the number of possible sources of complexity problems. There were two main reasons for which the current implementation was made independently rather than embedded into an existing theorem prover:

- Learning the ins and outs of the internal code of an existing theorem prover would have required copious amounts of time and the permission and collaboration of the people responsible for them.
- More importantly, the current state of our understanding of minimal commitment resolution for ESQ logic is the product of this PhD, not something that we had beforehand; and the flexibility of having an independent implementation that is not constrained by particular rationales and idiosyncrasies of an existing theorem prover has been key to enabling and speeding up this process of understanding.

In other words, I believe that it is now, that we have a formal description, theoretical real estate and insights into the limitations and difficulties of the approach, that the task of combining it with the pre-existing complexities of a theorem prover should be tackled. Having attempted this before would have made the task perhaps even more overwhelming than it already has.

Another potentially useful way to explore how to make the algorithm more practically applicable would be to reduce the expressivity of the underlying logic in

[1] The only reason I did not do this during the PhD was a lack of time.

which we are trying to find instantiations of the patterns. This makes particular sense in the context of ontology engineering. I chose first-order logic as the underlying object-level formalism for ESQ logic because, from a theoretical point of view, it is simpler and easier to understand. However, in practice, ontologies typically use less expressive formalisms such as OWL (see §3.1) or Datalog, precisely for performance reasons. Even first-order theorem proving approaches often limit the theories to Horn clauses (such as the Prolog programming language, see §3.2.1). Choosing one or several of these limited expressivity logics and simplifying the algorithm to leverage the simplifications that these offer could prove to be a relatively simple and effective way of making the algorithm computationally feasible. This would also allow the exploration of some of the existing approaches with limited expressivity discussed in §5.4, such as SMT, ASP or Lambda Prolog. Specific, justified choices of expressivity limitations would need to be made and adequately connected to existing approaches, and the particular implementations built and understood in detail.

Another aspect not pursued during this PhD that would be of benefit for both points above and for other potential future applications of the ideas here, would be to implement the theoretical concepts presented here within a formal proof system. To be clear, we are not discussing about extending a formal proof systems with these ideas, but rather, and not related to the fact that the ideas here are related to formal proofs, to use a formal proof system to check and streamline the proofs in chapters 6 and 7. Potentially also to verify that the patterns described in the pattern catalogue (appendix A) are in fact capable of detecting the faults they were designed to detect to begin with, without the search and general aspect that the algorithm implemented in this PhD carries out (verification rather than automation). Formalizing the theory and patterns in this thesis to this level of detail might provide additional insights and signal issues not identified thus far.

10.2.2 Pattern catalogue

Another clear direction for future work is the extension of the pattern catalogue, using ESQ logic (or a refinement of it) to express more patterns from the literature.

This is promising, at least from a conceptual point of view as a way to discuss the abstract and general implications of ontology debugging, which was one of the goals of this PhD.

However, the current state of the algorithm's implementation would mean that this would not, right now, translate into any practical capabilities of automatically detecting faults in ontologies. Moreover, we have discussed how there are some other limitations (albeit not as blocking) to the semantic systematic approach of ESQ logic's ability to express certain patterns that have a more lexical (i.e. naming) or methodological nature (see chapter 9 and appendix C).

I believe it would still be valuable work, even if perhaps it would be more effective to further refine the approach first.

Another step in relation to the pattern catalogue that would be wise to take should the ideas in this thesis be developed further, would be to formalize not just the patterns themselves (which has already been done), but rather the full second-order problem associated with the examples they are based on. This would give a more objective, formal goal for our algorithm and any other competing algorithms to solve. We have not done this during this PhD for the reasons listed above relating to the issues with the implementation, and a lack of time.

10.2.3 Higher-order unification

As mentioned before, one of the most interesting extensions of the work in this PhD to problems outwith ontology debugging is the possibility of some of the theoretical principles and approaches, and in particular dependency graph unification, to improving existing higher-order unification [Huet, 1975] algorithms.

While it is clearly not certain that the exploration of this avenue would yield direct results, I strongly believe there is, at the very least, a lot to learn about higher-order unification from this direction of work; and there may be some improvement to be had in its capabilities from it. In order to do this, the first step would be to generalize the dependency graph approach through a graph rewrite system to full higher-order logic rather than just ESQ logic. While this is no small task, there are no reasons to believe it would have any fundamental challenges, since the basic ideas of dependency graph unification are core to the definition of unification rather than to the specificities of ESQ logic. After this has been achieved, we could use the dependency graph point of view to try to find:

- Heuristic or pseudo-heuristic ways in which looking at the dependency graph

could allow us to optimize the non-determinism and thus search inherent in higher-order unification (which is the single source of computational challenges to it).

- Theoretical results that could be extracted from the dependency graph point of view that would allow us to better understand the details of higher-order unification and potentially find other improvements to it, using dependency graphs only as a theoretical tool and not an implementation tool.
- An implementation that would allow us to find instantiations of queries rather than individual proofs in higher-order logic, as efficiently as possible.

It is worth noting that higher-order theorem proving is an incredibly generic problem with enormous applications ranging from formal verification to type checking in programming languages, going through ontology engineering itself. Two of the main reasons its use is not more widespread are precisely its technical complexity and performance challenges. A potential enhancement to the performance of higher-order theorem proving algorithms could have important long-term consequences in a variety of fields.

10.2.4 Other applications of ESQ logic

There are at least two problems outwith ontology debugging to which I believe ESQ logic patterns and an automated way to find their instantiations could be of potential use, although we imagine some more may exist. These are obviously avenues of future work that we have not explored in much detail so we are not certain of how far we could go.

The first of them is for type inference in functional-style programming languages (like Haskell). Type inference in strictly typed functional languages is and is treated, for all intents and purposes, as a theorem proving problem. This is based on the well known Curry-Howard isomorphism [Sørensen and Urzyczyn, 2006]. I think it is possible that ESQ logic, and specifically the pattern instantiation aspect novelty in it, when applied to type inference, may enable certain abilities that current type checkers do not have. In particular, the ability to auto-complete inferrable types or automatic implementation of multiple differently typed versions of the same function from its type definition.

Similarly, in usual first-order theorem proving, finding intermediate lemmas for inductive proofs has long been a known source of problems. Approaches like [Bundy et al., 1993] use similar but less powerful ideas of pattern matching to potentially find such lemmas. It is conceivable that an adaptation of ESQ logic would allow us to generate patterns for these intermediate lemmas and find them automatically.

10.2.5 Enumeration procedures and infinite search spaces

Our systematic and computationally sound treatment of infinite and infinitely branching search spaces, as discussed in chapter 8, is based on well-known theoretical principles in mathematics and theoretical computer science, like cardinality, diagonalization and enumeration procedures. However, existing implementations, at least in Haskell, of such approaches, do not offer all the guarantees and properties that our approach does. For example, we found that the most advanced Haskell library available for this sort of approach does not have the capability to **fairly** interleave two search processes: One that never produces any results, and another one that does. Instead, the unproductive search process will take the processing thread hostage and never return it, not allowing the productive process from producing any results. This gets further complicated when you consider infinitely many processes, each of which could potentially be unproductive, that need to be interleaved in a fair way. Our implementation tackles all of these problems and guarantees fairness in all cases.

It is my opinion that the reason why existing approaches do not tackle all of these issues is that problems which involve traversing an infinite search space are often dismissed as computationally intractable, or replaced by different versions of the problem that have a finite search space or that can find a single solution within finite time. Situations where the set of solutions is infinite and the search space is infinitely branching are not typically tackled directly.

However, producing standard libraries to this end, and perhaps more importantly, properly describing what traditional mathematical theory says and does not say about the productivity and usefulness of traversing infinite search spaces, from the point of view of actual implementation rather than pure computational theory, could have the potential of opening up a new way to tackle certain problems with large and complex search spaces like ours, that would not help their performance, but could enable them to be implemented in the first place.

Thus, properly and independently presenting the enumeration procedure approach

we have followed, its properties and providing a clean and reusable implementation of it in Haskell could be a productive direction of work.

10.3 Summary

This PhD had as a principal goal providing more general and abstract common languages, formalisms and techniques in the field of ontology debugging and in particular automatic detection of faults in ontologies. To this end, we inspected the existing literature, produced a novel formalism that could capture a large proportion of the notions in it in a unified way, and designed a novel algorithm that could automatically detect instantiations of patterns in this formalism.

In practice, this algorithm has proven to be computationally infeasible in its current implementation, with important challenges that are not clear whether they could be overcome. While this is disappointing, the formalism itself and the partial pattern catalogue developed by expressing notions in the literature within it are by themselves valuable contributions that move the field towards the goal of having a common foundation. Moreover, identifying the sources of the computational challenges and some of the strengths of the algorithm offers us insights into the general problem. Finally, it is still possible that subsequent attempts at implementing algorithms based on the one presented here could end up proving computationally adequate, providing a general and entirely new way to detect patterns in ontologies.

Due to its nature as exploratory work, there are few definitive conclusions from this work. Ideas have been explored, insights have been gathered, results have been produced, challenges have been found and new ideas have been laid down. Ontology debugging remains a challenging field and an important limitation on the capabilities of ontology engineering as a larger field, but I hope to have produced some useful knowledge in how it could be conquered or how its fundamental limits may eventually be understood.

Appendix A

Pattern catalogue

A.1 Fault information

For each example, the following information is given:

- Description of the example.
- Reasoning behind considering it a fault.
- Formalism in which the example is originally represented. For example, OWL, first-order logic, etc.
- Conceptual source of the fault. For example, an inadequate blend of ontologies, an imprecision in a natural language processing technique or a misconception by an ontology designer.
- Specific source of the example. A reference to an existing ontology where it was found, a paper where it was mentioned, etc.
- Detection strategy
- (Optional) Repair suggestions. These are always quite informal and non-systematic, but they may still be quite useful in many cases. We do not explore repair mechanisms in this thesis, but it is an obvious avenue of future work and when compiling the pattern catalogue it made sense to make some notes about it.
- Formal fault pattern. Expressed using ESQ logic (see chapter 4).
- Additional contextual information that is used at the meta-level for defining the patterns.

As an additional note regarding formulation: take into account that whenever the present tense is used to make a statement (for example, “a chocolate ice-cream is not a pizza”), we mean that *in the preferred model*[1] a chocolate ice-cream is not a pizza, whereas the faulty ontology might entail so.

A.2 Fault patterns

A.2.1 OWL: Primitive versus defined classes (Spicy topping)

In OWL, class definitions may be made partial or complete. A partial definition gives rise to what is called a primitive class. In other words, it only has necessary conditions but not sufficient conditions. On the other hand, a complete definition gives rise to a defined class, having both necessary and sufficient conditions.

A common error in OWL is to make primitive classes which should be defined, defined classes which should be primitive or make subsumptions between them which are awkward and can even, for example, make certain classes unsatisfiable.

The following definition of SpicyTopping, SpicyBeefTopping and MeatTopping, the three of them as primitive classes, is faulty.

class(SpicyTopping partial PizzaTopping restriction(hasSpiciness someValuesFrom {“Spicy”}))
class(MeatTopping partial PizzaTopping)
class(SpicyBeefTopping partial MeatTopping SpicyTopping)

A.2.1.0.1 Why is it a fault

A topping is a spicy topping *whenever* it is spicy. SpicyBeefTopping is a SpicyTopping *because* it is spicy, not the other way around. While it is correct that SpicyBeefTopping

[1] In some sense, what we consider *to be really true.*

is both a MeatTopping and a SpicyTopping, other toppings which were spicy could fail to be subsumed by SpicyTopping because of SpicyTopping being a primitive class when it should be defined.

A.2.1.0.2 Formalism

OWL.

A.2.1.0.3 Conceptual source of the fault

A misconception or a mistake from the ontology designer. They have failed to correctly use the language constructs.

A.2.1.0.4 Specific source of the example

[Rector et al., 2004]

A.2.1.0.5 Detection strategy

As mentioned in [Rector et al., 2004], a good way to detect these kinds of faults is to generally assume that

1. Primitive classes do not inherit from multiple other primitive classes.
2. Defined classes do not subsume primitive classes

Robert Stevens, external examiner of this thesis, and co-author of [Rector et al., 2004], on which this pattern is based, has noted that their intention in said paper was to apply this pattern only to *asserted* subsumptions rather than all *inferred* subsumptions, which is the way in which we apply it here. I have, however, chosen to keep this pattern applied to all asserted classes for multiple reasons:

- It exhibits the capabilities of the algorithm much better.
- This particular example is not asserted multiple inheritance, so that pattern would fail to detect this fault.
- Changing it in our pattern catalogue and all test cases would involve a large time consumption at the late stages of this PhD.

- The specific semantics of the patterns is not a primary contribution or particularly relevant aspect of this thesis.

However, it made sense to add this clarification here for correctness and completeness of the arguments.

In the pizza ontology, this would succeed to detect this fault.

A.2.1.0.6 Repair suggestions

Essentially, any way of breaking any of the subsumptions that produce the fault seems like a sensible repair suggestion. When a primitive class inherits from multiple other primitive classes, making one of them inherit the other will also avoid the faulty situation. We call this *ordering* the classes.

A.2.1.0.7 Fault pattern

Due to multiple inheritances over arbitrary classes (first-order expressions) being too broad, and limiting ourselves to primitive classes (which are finite in the ontology), the search can start with the finite enumeration of primitive classes and then uses this to constrain further search. In order to check for multiple inheritance, we not only have to check for proven subsumption but for *unproven* subsumption, which is equivalent to satisfiability checks, which are, in general, not computable. In order to deal with this, we leave this part of the query the latest (leftmost), so that it can be implemented last and using approximate methods. Formally:

$$\begin{aligned}&((X,Y,Z) \models^* (\exists x.X(x) \wedge \neg Y(x)) \wedge (\exists x.Y(x) \wedge \neg X(x))) \bowtie \\ &\bowtie ((X,Y,Z) \models (\forall x.Z(x) \implies X(x)) \wedge (\forall x.Z(x) \implies Y(x))) \bowtie \\ &\bowtie ((X,Y,Z) \models_M primitive(X) \wedge primitive(Y) \wedge primitive(Z))\end{aligned} \tag{A.1}$$

This is to be read this way:

1. Find classes X, Y and Z such that Z is subsumed by both X and Y...
2. ... but remaining limited to primitive classes...
3. ... and for each of them check whether it is the case that neither X subsumes Y nor viceversa.

For the second pattern (defined classes that subsume primitive classes), it is even easier to express:

$$\begin{aligned}&((X,Y) \vDash \forall x.Y(x) \implies X(x)) \bowtie \\ &\bowtie ((X,Y) \vDash_M \mathit{defined}(X) \wedge \mathit{primitive}(Y))\end{aligned} \tag{A.2}$$

A.2.1.0.8 Related contextual information

The predicates $\mathit{primitive}(X)$ and $\mathit{defined}(X)$ are contextual information, as it is related to the *partial* and *complete* constructs of OWL. While a class definition in OWL is simply a labelling of a logical condition, the fault pattern relies on *how* that condition is expressed (which is contextual information). A primitive class definition indicates that any object that falls under that label must satisfy its definition. A defined class definition indicates that it is equivalent to its definition.

A.2.2 Missing necessary conditions (Margherita pizza)

This example, drawn from [Rector et al., 2004], and within the commonly used Pizza ontology in OWL, is faulty because it fails to indicate that Margherita pizza necessarily has only mozzarella and tomato toppings.

class(MargheritaPizza partial
Pizza
restriction(hasTopping someValuesFrom Mozzarella)
restriction(hasTopping someValuesFrom Tomato))

A.2.2.0.1 Why is it a fault

A Margherita pizza cannot have any other toppings other than mozzarella and tomato. More abstractly, the definition of a Margherita pizza not only says which toppings it needs to have, also those which it cannot have. If I ask for a Margherita pizza, then there is no doubt about what toppings it should have; whereas, if I asked for a vegetarian pizza, I would only be stating some conditions over the toppings of the pizza that I want. This is related to the notion of defined and primitive classes.

A.2.2.0.2 Formalism

OWL.

A.2.2.0.3 Conceptual source of the fault

A misconception or a mistake from the ontology designer. They have failed to state all the axioms for a Margherita pizza.

A.2.2.0.4 Specific source of the example

[Rector et al., 2004]

A.2.2.0.5 Detection strategy

It is related to the defined versus primitive problem, but is not exactly so. In a sense, with respect to the hasTopping property, Margherita pizza should be defined, but it is primitive.

However, we have been unable to identify any characteristic situation in this example which would allow us to detect it as faulty, and which does not at the same time mistakenly detect faults in correct examples. In a way, it is a *purely semantic* fault that is happening here, where something is simply incorrect but has no particular shape that would allow us to detect it without having some additional knowledge. At least, it is not possibly by only looking at the definition of the Margherita pizza; in a more complete ontology where the Margherita pizza is related to other parts of the ontology, this situation might become more apparent.

A.2.3 Incorrect subclass axioms (Four cheese pizza)

In general, it is odd to use global subclass axioms in OWL, as subclass axioms are generally specified locally in the subsumed class. This generally points out a fault.

As explained in [Rector et al., 2004], the reasons for introducing global subclass axioms might be related to their meaning of implication being misunderstood.

For example, we might want to state that Mozzarella and Edam are necessarily different kinds of cheese, so that Mozzarella is not Edam and Edam is not Mozzarella. The correct way to state this would be to make MozzarellaTopping and EdamTopping disjoint classes. This would be equivalent to the following (correct) subclass axioms:

EdamTopping SubClassOf not (MozzarellaTopping)
MozzarellaTopping SubClassOf not (EdamTopping)

However, because cheese toppings are always conceived in the context of pizzas, it is possible for an ontology designer to (incorrectly) think that this incompatibility is in fact related to pizzas and their toppings, and so instead use the following subclass axioms

(hasTopping someValuesFrom EdamTopping) SubClassOf (not (hasTopping someValuesFrom MozzarellaTopping))
(hasTopping someValuesFrom MozzarellaTopping) SubClassOf (not (hasTopping someValuesFrom EdamTopping))

If this is combined with the definition of a four-cheese pizza

class(FourCheesePizza partial
Pizza
restriction(hasTopping someValuesFrom MozzarellaTopping)
restriction(hasTopping someValuesFrom EdamTopping))
restriction(hasTopping someValuesFrom CheddarTopping))
restriction(hasTopping someValuesFrom ParmezanTopping))

The result is that FourCheesePizza is unsatisfiable.

A.2.3.0.1 Why is it a fault

FourCheesePizza is unsatisfiable. The sub-class axioms are stating the incorrect notion that no pizza can have both Mozzarella and Edam, when what they were introduced for was to state that Edam and Mozzarella are two different kinds of cheese.

A.2.3.0.2 Formalism

OWL.

A.2.3.0.3 Conceptual source of the fault

The ontology designer used global subclass axioms incorrectly. In general, as indicated in [Rector et al., 2004], global class axioms are rarely a correct construct to use.

A.2.3.0.4 Specific source of the example

While I personally invented this example, the motivation for this kind of example (incorrect usage of subclass axioms) comes from [Rector et al., 2004].

A.2.3.0.5 Detection strategy

In [Rector et al., 2004], it suggests that any usage of global subclass axioms is prone to be faulty. Moreover, in this particular example, the presence of the unsatisfiable class FourCheesePizza certainly points out to an error.

A.2.3.0.6 Repair suggestions

If the source of the fault is indeed considered to be the global subclass axiom, then removing it would remove the fault, but it likely would also remove a relevant chunk of semantics that we do not wish to remove. Moreover, if we use the approach of considering the general case of unsatisfiable classes, then it is near impossible to give a generic repair suggestion.

A.2.3.0.7 Fault pattern

It is easy to express that unsatisfiable primitive classes are likely faulty.

$$\begin{aligned}&((X) \models \neg\exists x.A(x)) \bowtie \\ &\bowtie ((X) \models_M primitive(X))\end{aligned} \tag{A.3}$$

A.2.3.0.8 Related contextual information

The *primitive* predicate was already discussed in a previous example.

A.2.4 Incoherent domain axioms (Chocolate ice-cream)

As described in [Rector et al., 2004], domain and range axioms which are incoherent[2] with other parts of an ontology in OWL may produce very unexpected results.

If we have the following domain axiom for the hasTopping property in our Pizzas ontology:

hasTopping domain Pizza

[2]Note here that incoherent is intentionally used as an informal and imprecise word describing some lack of regularity or relation between different parts of the same whole.

But then we (incoherently) use the hasTopping property to talk about ice-cream toppings, the ontology would use reasoning to conclude faulty facts: that a chocolate ice-cream is a pizza.

class(ChocolateIceCream partial
IceCream
restriction(hasTopping someValuesFrom ChocolateTopping))

A.2.4.0.1 Why is it a fault

A chocolate ice-cream is not a pizza.

A.2.4.0.2 Formalism

OWL

A.2.4.0.3 Conceptual source of the fault

An incoherence in the use of the hasTopping predicate. On one hand, it is stated that it only applies to pizzas, but later it is also used for ice-creams.

A.2.4.0.4 Specific source of the example

[Rector et al., 2004]

A.2.4.0.5 Detection strategy

In this particular fault related to domain axioms, the result is that ChocolateIceCream is a primitive class which inherits both IceCream and Pizza. Thus, it violates the principle, mentioned before, that primitive classes should form a tree. Meta-ontology fault detection using this pattern would detect this fault. Another way to detect this fault would be if we further indicated that IceCream and Pizza are disjoint classes. This would make ChocolateIceCream an unsatisfiable class, which would, in turn, also be detected as a fault.

A.2.4.0.6 Repair suggestions

The repair suggestions included in the previous example of non-tree primitive classes would also work perfectly for this example.

A.2.5 Assuming universal quantification implies existential quantification (Empty pepper pizza)

In [Rector et al., 2004], it is mentioned that one of the most common sources of errors for ontology designers unfamiliar with OWL is to assume that universal quantifiers imply existential quantifiers. A class which only has a universal quantifier as an axiom for a property may have that axiom satisfied trivially by having an empty range for that property.

As an example, consider the following definition of a PepperPizza:

```
class(PepperPizza complete
Pizza
restriction(hasTopping allValuesFrom PepperTopping))
```

And now consider the following definition of MargheritaPizza:

```
class(MargheritaPizza complete
Pizza
restriction(hasTopping allValuesFrom Nothing))
```

The ontology then entails that a margherita pizza is a pepper pizza, which is not conceptually what we refer to when we talk about pepper pizzas.

A.2.5.0.1 Why is it a fault

A margherita pizza is not a pepper pizza. A pepper pizza needs to have **some** pepper in it.

A.2.5.0.2 Formalism

OWL

A.2.5.0.3 Conceptual source of the fault

A failure, by the ontology designer, to express all conditions that define a PepperPizza. An assumption that universal quantification entails existential quantification.

A.2.5.0.4 Specific source of the example

I have concocted this example based on the ideas expressed in [Rector et al., 2004].

A.2.5.0.5 Detection strategy

A good approach is to check if there is any class subsumption (PepperPizza subsumes MargheritaPizza) which is *only* enabled by a trivial satisfaction of a universal quantifier in the definition of PepperPizza. While there may be some legitimate cases where this kind of subsumption is correct (for example, a MargheritaPizza is arguably a VegetarianPizza), it is a good start to signal these cases.

An important note should be made here. If tomato and mozzarella were considered toppings in the ontology, then this fault would be harder to detect, at least with the approach described here. While having no toppings is an extreme case that seems easier to flag, having no toppings except mozzarella or tomato (or any other kind of composite predicate like that) seems too broad to signal as a fault. For example, if we had a concept PepperPizza which could have green, red or yellow peppers and then a RedPepperPizza that could have only red peppers, it would be foolish to suggest that it is a fault that the subsumption between RedPepperPizza and PepperPizza is spurious: it is not. There is no structural difference, without lexical knowledge about what pepper pizzas are, that differentiates this from the Margherita pizza being subsumed by Pepper pizza.

A.2.5.0.6 Repair suggestions

A sensible suggestion is to explicitly remove the trivial case from the definition of the class. This is easily done by adding a condition that there are at least some toppings in pepper pizzas.

A.2.5.0.7 Fault pattern

To put the idea described previously more precisely, we are looking for classes A (Pizza), B (MargheritaPizza) and C (PepperPizza), property R (hasTopping) and class P (PepperTopping) such that:

1. C subsumes B. We can use the previously defined macro $subsumes(C,B)$.
2. C has a universal property restriction to it. We require contextual information to identify this as being something explicitly indicated in the ontology[3].

[3] We could find any entailed universal property restriction for a class, but that would make the purpose of the pattern moot as we are looking for definitions that are not used, in some sense, not for entailments that are not used.

class_property_restriction(C,R,P). There is a class property restriction indicating that for every x of class C (every pepper pizza), it is related through relation R only to elements y of class P (it only has pepper toppings).

3. All instances of B have that universal property restriction fulfilled trivially. $\forall x.B(x) \implies \neg\exists y.R(x,y)$, which is read as, for every element of class B, there is no element to which it is related through property R.

A key difference with this case, and one that showcases the usefulness of the instantiation set approach as compared to more ad-hoc procedures, is that the subsumed class (X) is not constrained by the domain constraint, and so any class may work there. However, we can still leverage the computational properties of the rest of the pattern to ideally find instances quick, using equational reasoning while keeping account of what we are limited to.

The resulting pattern would be:

$$\begin{aligned}&((X,Y,R) \vDash (\forall x.X(x) \implies \neg\exists y.R(x,y)) \wedge (\forall x.X(x) \implies Y(x))) \bowtie \\ &\bowtie ((Y,R) \vDash_M \textit{class\_property\_restriction}(Y,R,P))\end{aligned} \tag{A.4}$$

A.2.5.0.8 Related contextual information

We have introduced the new contextual predicate *class_property_restriction* to express that something is expressed as a class property restriction explicitly in the OWL ontology. It is fundamental for the intuition behind this fault pattern.

A.2.6 Incorrect usage of logical constraints (ProteinLoversPizza)

Another example of trivial satisfaction of universal quantifiers, but due to a different underlying reason. It is worth including as an example in order to showcase the different applications that a single rule (looking for only trivially satisfiable classes) may have.

class(ProteinLoversPizza complete Pizza restriction(hasTopping allValuesFrom (MeatTopping and CheeseTopping and Seafood-Topping)))

Since MeatTopping, CheeseTopping and SeafoodTopping have been indicated as disjoint classes, the result is that ProteinLoversPizza necessarily has no toppings, and consequently, is a VegetarianPizza.

A.2.6.0.1 Why is it a fault

The intended meaning is that a ProteinLoversPizza only has either meat or cheese or seafood, not that it only has the intersection of them (which is empty).

A.2.6.0.2 Formalism

OWL

A.2.6.0.3 Conceptual source of the fault

A misuse of the logical construct “and”, instead of “or” by the ontology designer.

A.2.6.0.4 Specific source of the example

[Rector et al., 2004]

A.2.6.0.5 Detection strategy

This fault can also be detected by looking for universal quantifications which are only trivially satisfiable. That is, it should be detected by the same pattern specified in the previous example. ProteinLoversPizza is subsumed by VegetarianPizza, but only through a trivial satisfaction of its class property restrictions.

Another potential approach to detect this fault is to consider *unsatisfiable relative properties*. That is, classes and properties (that are meant to be related, a fact that we can derive from classes the class is subsumed by) such that it is entailed by the ontology that there is no possible instance of that relation.

A.2.6.0.6 Repair suggestions

Unlike the previous case, however, the repair suggestion is no longer sensible in this case. The problem is not that VegetarianPizzas need to have some toppings (they don’t), it is that ProteinLoversPizza needs to have some. This could be dealt with by extending the repair suggestion in the previous case with another option, being a fix to the lack of

toppings on ProteinLoversPizza.

If considering the unsatisfiable relative property approach, then it is hard to provide a particular suggestion as to how to make the subsumed class have some elements it is related to, similarly to how it was difficult in the unsatisfiable class case to provide a generic repair suggestion.

A.2.6.0.7 Fault pattern

The trivial satisfaction of universal quantifications is discussed in a previous example. The formalization of a pattern to detect unsatisfiable relative properties would be:

$$\begin{aligned}&((X,Y,R) \vDash (\forall x.Y(x) \implies X(x)) \wedge (\forall x.Y(x) \implies (\neg\exists y.R(x,y)))) \bowtie \\ &\bowtie ((X,R) \vDash \exists x,y.X(x) \wedge R(x,y))\end{aligned} \tag{A.5}$$

As usual, this does not necessarily indicate a fault, but it is likely a sufficiently uncommon situation so as to offer relevant potential faults. It could also be improved and extended by including conditions on the elements it is related to or considering only direct subsumptions, making it applicable to fewer cases.

A.2.7 Heterogeneus collective: Technical administrative group

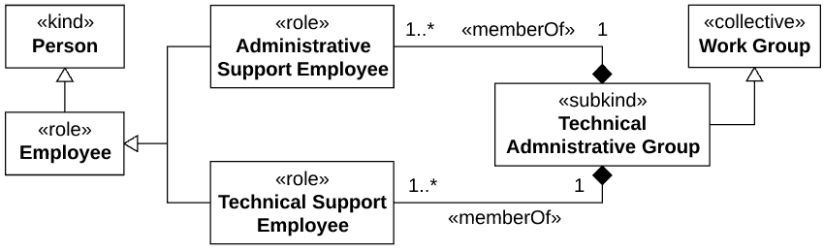
This example, coming from [Prince Sales and Guizzardi, 2017], is presented in said work as an example of an *heterogeneous collection*. The *anti-patterns* presented here are fully defined in the context of the Unified Foundational Ontology (UFO)[4].

In this ontology, there are four kinds of parthood relations: *subQuantityOf*, *subCollectiveOf*, *memberOf* and *componentOf*, each with their semantics. A *memberOf* relation has amongst its semantics the notion that different members of the same collective are *undistinguishable* w.r.t. the relation. A *heterogeneous collective* happens when the ontology implies that there is a way to distinguish the members of the collective.

An example of this situation is given in [Prince Sales and Guizzardi, 2017], as included in figure A.1.

[4] https://ontouml.org/ufo/

Figure A.1: Technical administrative group: a heterogeneous collective



The issue with this example is that a Technical Administrative Group is formed by employees, which can be Administrative Support Employees or Technical Support Employees, but if Technical Administrative Group is regarded as a collective, then there should be no way to distinguish between these from the point of view of the collective, and the ontology implies so, by using different classes both related to the Group class through the *memberOf* relation.

A.2.7.0.1 Why is it a fault

If Technical Administrative Group is a collective, then from its point of view, its members should not be distinguishable. They are all members of the group.

A.2.7.0.2 Formalism

OntoUML

A.2.7.0.3 Conceptual source of the fault

A misunderstanding by the ontology developer of the notion of collective and what it truly means. They should either change the type of parthood relation or use an intermediate Employee class.

A.2.7.0.4 Specific source of the example

[Prince Sales and Guizzardi, 2017].

A.2.7.0.5 Detection strategy

The paper gives a very specific anti-pattern to detect this particular type of fault.

A.2.7.0.6 Repair suggestion

The paper is also quite explicit about this. It is either not a collective, or the relationships used are not correct. Potentially, the distinguishing property between the two classes is not correct. So either change it to a functional complex, change at least one of the relationships to a different one or get rid of the distinguishing property. This last case is very generic and seemingly hard to make more concrete.

A.2.7.0.7 Fault pattern

The ideas expressed in the paper can be quite easily translated into fault detecting axioms in the context of meta-ontology fault detection, if we assume the ontology is linked with the UFO. A way to explain the anti-pattern described in [Prince Sales and Guizzardi, 2017] is the following:

1. There is a "whole" class which is stereotyped as one of *collective*, *subkind*, *phase*, *role*, *category*, *roleMixin* or *mixin*.
2. If it is stereotyped as *subkind*, *phase* or *role*, then it should be a subclass of a *collective*
3. If it is stereotyped as *mixin*, *category* or *roleMixin*, then all its subclasses should be *collective* or meet the previous condition.
4. There is two or more "part" classes which are stereotyped as one of the same stereotypes as the whole or the *kind* stereotype.
5. Each of these part classes are related to the whole through a *memberOf* relation.

We can express this easily as fault detecting axioms if we assume the existence of the corresponding classes and relations.

- When working with inferred subsumptions, we can summarize the first three conditions as... "any subclass of the whole class is a subclass of collective, and the class is either a subclass of collective itself or category, roleMixin or mixin". Note that "any subclass" here refers to any explicit sub-class.

  We can consider this a domain constraint that relies on the universal quantification over a finite set for S. Formally, we first define a domain constraint for explicit S that are collectives:

$$
\begin{aligned}
&D_S = \\
&((S) \vDash \forall x.S(x) \implies \mathit{collective}(x)) \bowtie \\
&\bowtie ((S) \vDash_M \mathit{explicit}(S))
\end{aligned}
\tag{A.6}
$$

On the side, we produce a domain constraint for X's, which involves search but should yield a fairly reduced answer set:

$$
\begin{aligned}
&D_X = \\
&((X) \vDash (\forall x.X(x) \implies \mathit{collective}(x)) \vee (\forall x.X(x) \implies \mathit{category}(x)) \vee \\
&(\forall x.X(x) \implies \mathit{roleMixin}(x)) \vee (\forall x.X(x) \implies \mathit{mixin}(x)))
\end{aligned}
\tag{A.7}
$$

We then combine these two to produce X's that satisfy the domain constraint for all S that subsume them:

$$
\begin{aligned}
&Q_X = \\
&(X,S) : \forall(((X,S) \vDash \forall x.S(x) \implies X(x)) \bowtie D_S).D_X
\end{aligned}
\tag{A.8}
$$

This is to be read in the following way: Consider all explicit classes S that are subsumed by *collective*. For *each of those*, find all X's in the answer set of D_X which subsume it. From those, keep only those X's that fulfill that condition for all S's, and those are the X's that we will execute the rest of the query against. In other words, we use a forall query to find this query dependent on another query.

We will use this same domain constraint on each of the three variables X, Y and Z in the final query.

- Same thing for classes Y and Z, and add that $\forall y.Y(y) \iff Z(y)$ is not entailed (its negation is satisfiable) to indicate that they are different classes.
- $\forall y.Y(y) \implies (\exists x.X(x) \wedge \mathit{memberOf}(y,x))$ and $\forall z.Z(z) \implies (\exists x.X(x) \wedge \mathit{memberOf}(z,x))$, to indicate the relations.

An important thing to note is that the pattern to indicate that Y and Z are different classes would, in principle, be a satisfiability pattern. However, it is syntactic difference that we are looking between Y and Z, not semantic one (or, at least, it would be more than enough for our purposes). We can do this via a check $Y \neq Z$ that indicates *syntactic*

distinction between their instantiations. We do this in the end, however, because doing it before could possibly hinder the possibility of performing the complex query over implicit instantiation sets, and instead force us to do it over large explicit instantiation sets. We do not want to generally do the *simplest* query first, but rather the one that produces the *most simply expressed* instantiation set.

The final pattern would then be:

$$\begin{aligned}&((X,Y,Z) \vDash Y \neq Z) \bowtie \\ &\bowtie ((X,Y,Z) \vDash (\forall y.(Y(y) \implies (\exists x.X(x) \wedge memberOf(y,x)))) \wedge \\ &\wedge (\forall z.(Z(z) \implies (\exists x.X(x) \wedge memberOf(z,x))))) \bowtie \\ &\bowtie (Q_X \bowtie Q_{XY/X} \times Q_{XZ/X})\end{aligned} \tag{A.9}$$

It is worth noting that a difference (which I consider an advantage) of using meta-ontology fault detection to detect this anti-pattern is that meta-ontology fault detection will detect any such anti-pattern in the *inferred* ontology, not only in the explicit definition of the ontology, which brings up potentially a lot harder to detect cases.

However, a very different (and simpler) approach can be pursued in meta-ontology fault detection by appealing to inference and elements instead of classes, stating that if two distinguishable members of a collective exist, then there is a fault:

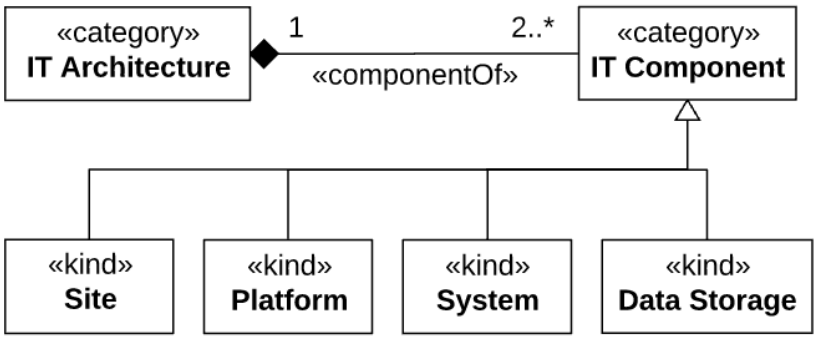
$$\begin{aligned}&((X) \vDash \exists x,y,z.X(z) \wedge memberOf(x,z) \wedge memberOf(y,z) \wedge P(x) \wedge \neg P(y)) \bowtie \\ &\bowtie ((X) \vDash \forall x.X(x) \implies collective(x))\end{aligned} \tag{A.10}$$

This showcases the usefulness of being able to have meta-variable P that ranges over any subformulas and not just actual predicates in the signature.

A.2.7.0.8 Related contextual information

We have introduced the predicate *explicit* to indicate that a class is epxlicitly defined in the ontology.

Figure A.2: IT component: a homogeneous functional complex



A.2.8 Homogeneous functional complex: IT component

Another example from [Prince Sales and Guizzardi, 2017], called a *homogeneous functional complex*. This situation happens when the ontology implies that all the components of a functional complex are indistinguishable.

An example of this situation is given in [Prince Sales and Guizzardi, 2017], as included in figure A.2.

A functional complex has several, *different*, parts. This example implies that, from the point of view of the relation, all components of the architecture are equal, which means that it is not truly a functional complex.

A.2.8.0.1 Why is it a fault

If IT Architecture is a functional complex, then from its point of view, it should have different components.

A.2.8.0.2 Formalism

OntoUML

A.2.8.0.3 Conceptual source of the fault

A misunderstanding by the ontology developer of the notion of functional complex and what it truly means. They should either change the type of parthood relation or remove the intermediate class.

A.2.8.0.4 Specific source of the example

[Prince Sales and Guizzardi, 2017].

A.2.8.0.5 Detection strategy

The paper gives a very specific anti-pattern to detect this particular type of fault.

A.2.8.0.6 Repair suggestion

The paper is also quite explicit about this. It is either not a functional complex, or the relationships used are not correct. So either change it to a collective, or change at least one of the relationships to a different one.

A.2.8.0.7 Fault pattern

Once again in a very similar way to the previous anti-pattern, the ideas expressed in the paper can be quite easily translated into fault detecting axioms in the context of meta-ontology fault detection, if we assume the ontology is linked with the UFO:

1. There is a "whole" class which is stereotyped as one of *kind*, *subkind*, *phase*, *role*, *category*, *roleMixin* or *mixin*.
2. If it is stereotyped as *subkind*, *phase* or *role*, then it should be a subclass of a *kind*
3. If it is stereotyped as *mixin*, *category* or *roleMixin*, then all its subclasses should be *kind* or meet the previous condition.
4. There is exactly one "part" class which is stereotyped as one of the same stereotypes as the whole.
5. The two classes are related through a *componentOf* relation.

We can express this easily as fault detecting axioms if we assume the existence of the corresponding classes and relations.

$$
\begin{aligned}
&D_S = \\
&((S) \vDash \forall x.S(x) \implies kind(x)) \bowtie \\
&\bowtie ((S) \vDash_M explicit(S))
\end{aligned}
\tag{A.11}
$$

Produce a similar domain constraint for the actual classes to check (X):

$$
\begin{aligned}
&D_X = \\
&((X) \vDash (\forall x.X(x) \implies kind(x)) \vee (\forall x.X(x) \implies category(x)) \vee \\
&(\forall x.X(x) \implies roleMixin(x)) \vee (\forall x.X(x) \implies mixin(x)))
\end{aligned}
\tag{A.12}
$$

and combine them to produce the X's that satisfy the domain constraint for all S that subsume them:

$$
\begin{aligned}
&Q_X = \\
&(X,S) : \forall(((X,S) \vDash \forall x.S(x) \implies X(x)) \bowtie D_S).D_X
\end{aligned}
\tag{A.13}
$$

Finally, the global pattern would look like this:

$$
\begin{aligned}
&((X,Y) \vDash (\forall x,y.(X(x) \wedge componentOf(y,x)) \implies Y(y)) \\
&\wedge (\forall y.Y(y) \implies (\exists x.X(x) \wedge componentOf(y,x)))) \bowtie \\
&\bowtie Q_X \bowtie Q_{XY/X})
\end{aligned}
\tag{A.14}
$$

Similarly to the previous example, a very different approach is to consider that if a functional complex has no undistinguishable elements, then something is incorrect. However, note that in this case, this is not exactly the same condition, as the elements themselves might be distinguishable, but maybe not directly through the direct relation they have with the functional complex. It is arguable, however, that this is an advantage and not a disadvantage.

As it turns out, in this case it requires a universal quantifier over a second-order variable (two elements are undistinguishable if *all predicates* are equal over them, so this cannot be expressed in ESQ logic the way it currently is.

A.2.9 Creating synonyms as classes

Pitfall P2 on [Poveda-Villalón et al., 2010, Poveda Villalón, 2016], defining several classes which are merely synonyms, while not directly harmful, is confusing and bad practice, and particularly dangerous as the ontology evolves.

An example would be to define two classes, Falls and Waterfall, indicating an equivalence between them:

EquivalentClasses(Falls Waterfall)

A.2.9.0.1 Why is it a fault

It provides unnecessary redundancy that can be considerably dangerous as the ontology evolves.

A.2.9.0.2 Formalism

Mostly OWL, but it is applicable to other formalisms.

A.2.9.0.3 Conceptual source of the fault

It depends. It could be due to two initially different concepts evolving to become the same, or due to miscoordination between different ontology developers.

A.2.9.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

A.2.9.0.5 Detection strategy

It is fairly simple. Any equivalent class axiom in which both classes are primitive classes is most likely faulty in the misrepresentation sense explained.

Note that, at least in the way it is approached in [Poveda-Villalón et al., 2010], when the classes are equivalent but this is *inferred* and not an equivalent classes axiom, it is not likely to be faulty. It still presents a fairly strange situation, but we will not include it here.

A.2.9.0.6 Repair suggestions

Either remove the axiom, or make one of the classes defined.

A.2.9.0.7 Fault pattern

We can easily translate what's explained above into a pattern:

$$\begin{aligned}&((X,Y) \models_M primitive(X) \wedge primitive(Y)) \bowtie \\ &\bowtie ((X,Y) \models_M equivalent\_classes(X,Y))\end{aligned} \tag{A.15}$$

A.2.9.0.8 Related contextual information

Aside from the primitive class definition that we use throughout, we are using contextual information *equivalent_classes* that indicates that there is an equivalent classes axiom between *A* and *B*.

A.2.10 Subsumption cycles

Pitfall P6 in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016], having (primitive) classes *A* and *B* transitively subsume each other (through explicit subsumption definitions) but not be explicitly equivalent indicates that one of such subsumptions is likely faulty.

For example, consider that an ontology correctly indicates that Professor is a subclass of Person, and Person is a subclass of Individual, but then incorrectly indicates that Individual is a subclass of Professor:

class(Professor partial Person)
Person SubClassOf Individual
Individual SubClassOf Professor

A.2.10.0.1 Why is it a fault

It generates a pletora of incorrect semantics. The original one is that not all individuals are professors.

A.2.10.0.2 Formalism

OWL

A.2.10.0.3 Conceptual source of the fault

A double understanding of the meaning of the class Individual, or a mistake in the ordering of the classes when defining the subclass axiom.

A.2.10.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

A.2.10.0.5 Detection strategy

In essence this example is completing the one presented in the previous example, by stating that *inferred* equivalence of primitive classes is also faulty.

A.2.10.0.6 Repair suggestions

Removing one of the subsumptions from the chain of subsumptions seems like the most obvious repair. However, if this subsumptions are inferred, how to do this exactly could be complicated and very dependent on the particular case.

A.2.10.0.7 Fault pattern

As mentioned, this example is completing the one presented in the previous example, by stating that *inferred* equivalence of primitive classes is also faulty. Therefore, both of them could be detected with a single pattern:

$$\begin{aligned}&((X,Y) \vDash \forall x.X(x) \iff Y(x)) \bowtie \\ &\bowtie ((X,Y) \vDash_M primitive(X) \wedge primitive(Y))\end{aligned} \tag{A.16}$$

A.2.10.0.8 Related contextual information

Once again, we use the primitive class definition. We also introduced the notion of explicit subsumption (a subsumption that is directly expressed in the ontology, either in the definition of the subsumed class or through a SubClassOf axiom).

A.2.11 Missing domain or range properties

Pitfall P11 in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016], properties which have no domain or range constraints are regarded as prone to faults.

For example, consider the following ontology:

class(Writer)
class(LiteraryWork)
ObjectProperty(writesLiteraryWork)

A.2.11.0.1 Why is it a fault

Without domain or range constraints, *writesLiteraryWork* could in theory apply to any object in the semantics. This has many unwanted consequences, such as the inability to infer useful theorems about the property because its wide domain and range mean that strange situations can appear, the potential for the property itself to interfere with the inference of other parts of the ontology for the same reason, and all the secondary consequences these may have.

A.2.11.0.2 Formalism

OWL

A.2.11.0.3 Conceptual source of the fault

Most likely the author forgot to include the domain and range constraints.

A.2.11.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

A.2.11.0.5 Detection strategy

It is in principle easy to detect. Look for explicit properties in the ontology that have no domain or range constraints.

It is very interesting that in [Poveda-Villalón et al., 2012], the authors acknowledge that their detection mechanism is limited in the sense that it will not count inherited domain or range constraints for these purpose. Our approach in principle goes even further to consider any inferred domain or range constraints as valid, enabling the author of the ontology to rely on reasoning to provide these, and meaning that our automated detection mechanism is even more useful for working with these situations while still having a way to check whether we forgot to provide them.

However, because we are looking for properties that *do not have* range or domain constraints, we are not finding an instantiation of these domain or range constraints that makes them provable, but rather, that are satisfiable.

A.2.11.0.6 Repair suggestions

It is very hard to provide useful repair suggestions for this fault pattern, as that would imply having knowledge about what the domain or range of the property should be.

A.2.11.0.7 Fault pattern

$$((P) \models^* \forall x.\exists y.P(x,y)) \bowtie ((P) \models_M explicit\_property(P))$$
$$((P) \models^* \forall x.\exists y.P(y,x)) \bowtie ((P) \models_M explicit\_property(P)) \tag{A.17}$$

A.2.11.0.8 Related contextual information

We have introduced the predicate *explicit_property* to indicate that a property is explicitly defined in the ontology.

A.2.12 Missing inverse properties

This relates to pitfall P13 in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016], although we consider it as two separate (albeit related) faults with separate detection methods.

Firstly, the definition of the pitfall in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016] presumes that any property without an inverse property is likely faulty. This is arguable, but the pattern can be included whenever we work under such assumptions. Yet again, the detection method in [Poveda-Villalón et al., 2012] only finds properties without *explicit* inverse properties. However, we do not see a way to extend this to inferred inverse properties. Any property necessarily has an inferred property that is its inverse: its inverse, by definition (formally, swapping the order of the arguments).

A sub-part of the problem (explicitly considered in [Poveda-Villalón et al., 2012]) is to have two explicit properties such that the explicit domain of one is the explicit range of the other and which are not explicit inverses. We can extend this to (still explicit) properties such that the *inferred domain* of one is the *inferred range* of the other, and which are not *inferred inverses*. This provides, yet again, potentially a larger and more interesting fault detection space. The reason to stick to explicit properties is, as mentioned above, the fact that the inverse property is always definable as an inferred

property.

An example of a situation that includes both cases, taken from [Poveda Villalón, 2016] is the following:

class(AdministrativeArea)
class(Language)
ObjectProperty(hasOfficialLanguage domain(AdministrativeArea) range(Language))
ObjectProperty(isOfficialLanguageOf domain(Language) range(AdministrativeArea))

A.2.12.0.1 Why is it a fault

The no explicit inverse property case assumes that it is always relevant to have access to the inverse property, and that lacking it is faulty in that the ontology is incomplete. In the case where both properties are present but they are not related, it is faulty because the relationship between the two properties is neither explicit nor inferred. This is a clearer faulty situation.

A.2.12.0.2 Formalism

OWL

A.2.12.0.3 Conceptual source of the fault

Forgetting to include the inverse relationships between the properties.

A.2.12.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

A.2.12.0.5 Detection strategy

The first case is very easy to detect because it relies exclusively on explicitly included information.

The second case can be detected by looking for properties such that the domain of one is the range of the other, but they are not inverse.

A.2.12.0.6 Repair suggestions

In the first case, the repair suggestion would be to add a new property that is the inverse of the existing one. In the second case, to explicitly indicate that they are inverses.

A.2.12.0.7 Fault pattern

The first case is an explicit check, which however requires a forall query to be able to check for all second-order variables.

$$\begin{aligned}&((P) : \forall(Q) \vDash \mathit{explicit\_property}(Q).\neg \mathit{explicit\_inverse}(P,Q)) \bowtie \\ &\bowtie ((P) \vDash_M \mathit{explicit\_property}(P))\end{aligned} \tag{A.18}$$

As for the second case, we can find it with satisfiability checking:

$$\begin{aligned}&((Q,P) \vDash^* (\exists x,y.\neg(P(x,y) \iff Q(y,x)))) \bowtie \\ &\bowtie ((Q,X,Y) \vDash \forall x,y.Q(x,y) \implies (Y(x) \wedge X(y))) \bowtie \\ &\bowtie ((Q) \vDash_M \mathit{explicit\_property}(Q)) \bowtie \\ &\bowtie ((X,Y,P) \vDash \forall x,y.P(x,y) \implies (X(x) \wedge Y(y))) \bowtie \\ &\bowtie ((P) \vDash_M \mathit{explicit\_property}(P))\end{aligned} \tag{A.19}$$

A.2.12.0.8 Related contextual information

The *explicit_property* predicate previously explained, and a *explicit_inverse* relation that relates properties with their explicit inverses.

A.2.13 Inkless books

Pitfall P14 in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016]. It is closely related to the fault described in §A.2.5, although the pattern used there cannot be directly applied to detect this fault. A book is (mistakenly) defined to be anything produced by a writer that *only uses paper*, instead of anything produced by a writer that *uses at least paper*. This is a particular case of the issue with intuition making users think that universal quantification implies existential quantification, or simply confusing the two of them.

An example ontology:

class(Book partial)
EquivalentClasses(Book ((producedBy someValuesFrom Writer) and (uses allValuesFrom Paper)))

Thus, a book cannot use ink.

A.2.13.0.1 Why is it a fault

Books use other things apart from paper, such as ink. It is an incorrect definition for the Book class.

A.2.13.0.2 Formalism

OWL

A.2.13.0.3 Conceptual source of the fault

A misuse of universal quantification where existential quantification should have been used.

A.2.13.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

A.2.13.0.5 Detection strategy

Related to, but not the same as, the pattern in §A.2.5, we can indicate that, generally, a universal property restriction (potentially inferred instead of explicit) for a primitive class lacking an associated existential property restriction is likely to indicate a fault.

In fact, this pattern would also detect the fault in §A.2.5, without the need for the subsumed class VegetarianPizza. It just would also trigger more false positives.

Note that [Poveda-Villalón et al., 2012] provides no automated detection for this.

A.2.13.0.6 Repair suggestions

Replacing the universal quantification with an existential quantification, or adding an existential quantification constraint, depending on the case. In the particular book case, replacing it would be the most adequate repair.

A.2.13.0.7 Fault pattern

As explained, finding any primitive class with a universal property restriction (over an explicit property) that lacks an associated existential property restriction should do the trick. Note that we do not require the class it is associated to to be primitive, as that would fail to detect a lot of fault cases (for example, say we indicated that a Book only used paper *or* ink, it would still be faulty, as it also uses covers). Formally:

$$\begin{aligned}&((X,Y,P) \models^* \exists x.X(x) \wedge \neg(\exists y.(P(x,y) \wedge Y(y)))) \bowtie \\ &\bowtie ((X,Y,P) \models \forall x.X(x) \implies \forall y.(P(x,y) \implies Y(y))) \bowtie \\ &\bowtie ((X) \models_M \mathit{primitive}(X)) \bowtie \\ &\bowtie ((P) \models_M \mathit{explicit\_property}(P))\end{aligned} \tag{A.20}$$

A.2.13.0.8 Related contextual information

primitive and *explicit_property*, which had previously been defined.

A.2.14 Vegetarian pizzas with meat

Pitfall P15 in [Poveda-Villalón et al., 2010, Poveda Villalón, 2016], it is related to the fault in §A.2.6 and also to §A.2.5, and in [Poveda-Villalón et al., 2010], even [Rector et al., 2004] is cited in relation to it. However, it is not exactly the same thing.

This fault occurs when the negation of existential property restrictions is misused, resulting in expressing that a vegetarian pizza is a pizza with *some topping that is not meat* instead of a pizza with *no toppings that are meat*. Formally, the following ontology would be faulty:

class(VegetarianPizza complete restriction(hasTopping someValuesFrom (not MeatTopping)))

A.2.14.0.1 Why is it a fault

Pizzas with meat and vegetables would be considered vegetarian, and it is not.

A.2.14.0.2 Formalism

OWL

A.2.14.0.3 Conceptual source of the fault

The interchange of the existential property restriction and the negation.

A.2.14.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

A.2.14.0.5 Detection strategy

Once again, it is hard to provide a pattern that would be general, would detect this fault and would not trigger a lot of false positives. Our approach is generally to not worry too much about the amount of false positives, specially if these happen at larger instantiations of the patterns (which will therefore be detected later). That is, our fault patterns aim to trigger potential faults, rather than signal almost definitive faults.

In that sense, we can say that a defined class that has an existential property restriction whose body is the negation of an explicit class in the ontology is likely to indicate a fault, as it is a very small restriction for a defined class. This pattern would, however, depend on this negation being directly on an explicit class. A more general approach would not be easily achieved, as any class is the negation of its negation, and thus a pattern like $\neg P(x)$, where P is non-restricted, is non-restricted itself.

A.2.14.0.6 Repair suggestions

Swapping the negation from within the existential quantification to outside it.

A.2.14.0.7 Fault pattern

We simply make explicit what we described in the detection strategy:

$$\begin{aligned}
&((X,P,Z) \models_M \mathit{class_property_restriction}(X,P,Z)) \bowtie \\
&\bowtie ((Y,Z) \models (\forall x.Y(x) \iff \neg Z(x))) \bowtie \\
&\bowtie ((X) \models_M \mathit{defined}(X)) \bowtie \\
&\bowtie ((P) \models_M \mathit{explicit_property}(P)) \bowtie \\
&\bowtie ((Y) \models_M \mathit{explicit}(Y))
\end{aligned} \tag{A.21}$$

A.2.14.0.8 Related contextual information

We use *defined*, *class_property_restriction*, *explicit* and *explicit_property*. These were all defined before in this document.

A.2.15 Unsatisfiable domains or ranges

It is related to pitfall P19 of [Poveda-Villalón et al., 2010, Poveda Villalón, 2016], but it is at the same time more general and more specific than it, in several senses. In the original source, the pitfall is described as "swapping intersection and union", which is more general than the one specified here. However, the description of the pitfall focuses on the case where this happens in the domain or range of properties. The example presented there implies defining the intersection of two classes as the range of a property instead of their union, resulting in an unsatisfiable range.

This fault really is a generalization to binary predicates (properties) of the fault specified in §A.2.3.

A concrete example, taken from [Poveda Villalón, 2016], of this fault is the following:

class(Event)
class(City)
class(Nation)
ObjectProperty(takesPlaceIn domain(Event) range(City and Nation))

This ontology, per se, does not imply an unsatisfiable property, it implies that the range of the property is in the intersection of cities and nations. However, it is to be expected that other axioms imply that cities and nations are disjoint (possibly due to classes they inherit from that are disjoint).

If we add the disjointness axiom to the ontology, the fault follows:

DisjointClasses(Nation City)

A.2.15.0.1 Why is it a fault

Technically, the property takesPlaceIn is useless because there will be no instances of it. Conceptually, the ontology claims that events take place in places that are *both* cities and nations. In repair suggestions we discuss what exactly is semantically wrong with this, but it certainly is not correct.

A.2.15.0.2 Formalism

OWL

A.2.15.0.3 Conceptual source of the fault

A misuse of an intersection (conjunction) when it should have been a union (disjunction).

A.2.15.0.4 Specific source of the example

[Poveda-Villalón et al., 2010, Poveda Villalón, 2016]

A.2.15.0.5 Detection strategy

As mentioned, we will flag as likely faulty (and in this case it is almost certain it is), any property for which it is entailed that there may be no instance of it.

A.2.15.0.6 Repair suggestions

Changing the intersection to a union is the most obvious solution. However, another possibility is to separate the property into two, one which indicates in what *city* the event takes place, and the other in what *country* the event takes place. However, this is probably less adequate. More importantly, a takes place property is probably transitive through geographic inclusions. That is, if an event takes place in a city, then it also takes place in the country that city is in, so a disjunction would be able to express this adequately.

A.2.15.0.7 Fault pattern

A pattern describing that unsatisfiable explicit properties are faulty would be:

$$\begin{aligned}&((P) \models \neg\exists x, y.P(x,y)) \bowtie \\ &\bowtie ((P) \models_M explicit\_property(P))\end{aligned} \tag{A.22}$$

A.2.15.0.8 Related contextual information

The *explicit_property* that we have used throughout.

Appendix B

Additional theoretical results and proofs

Theorem B.0.1 (Normalization of second-order terms)**.** *Every second-order term* ϕ *is equivalent to a unique normal second-order term* $\mathcal{N}(\phi)$.

This theorem was originally presented in theorem 6.1.1.

Proof. We use standard techniques for rewriting systems. In order to do this, we first must show that our notion of normality (definition 6.1.4) corresponds to normality over second-order reducibility. That is, we show that a second-order term is normal if and only if it is not reducible.

Reducibility may be produced through one of the three direct reducibility rules, transitively or through structural induction over the term structure. Assume ϕ is a normal second-order term. Then, it is either not a composition or, if it is a composition, its head is not a composition nor a projection, all its arguments are recursively normal and either there is an i for which the i-th argument is not π_i^m or the arity of the arguments is different from the arity of the head. If it is not a composition, then neither of the three direct reducibility rules apply, and it is not an inductively defined second-order term, so it is not reducible. Head simplification cannot be applied because there is an i for which the i-th argument is not π_i^m or the arity of the head is different from the arity of the arguments. Projection simplification cannot be applied because the head is not a projection. Similarly, function dumping cannot be applied because the head is not a composition. Finally, inductive reduction cannot be applied because the head is not a composition (and therefore cannot be reduced) and the arguments are recursively

assumed to be normal.

On the other direction, we will show that if a second-order term is not normal, then it can be reduced. If ϕ is not normal, then it must be a composition; whose head is a composition or a projection or its arguments are not normal or all its arguments are π_i^m, and the head has arity m. If any of its arguments are not normal, then we may recursively assume we may reduce those through inductive reduction. If all its arguments are π_i^m, where m is the arity of the head, then we may apply the head simplification rule. Therefore, the only remaining case is that ϕ is a composition whose head is a composition or a projection. If its head is a projection, then the projection simplification rule may be applied. If its head is a composition, then the function dumping rule may be applied. In either case, it is reducible.

We will now show that the rewrite system $\xrightarrow{*}$ is confluent. In order to do this, we will show it is locally confluent and terminating, and apply the diamond lemma. This, together with the irreducibility of normal forms, implies the theorem.

We first show termination. Define a measure d on second-order terms indicating the depth of compositions, defined as 0 for non-compositions and $d(\phi_0) + \max\{d(\phi_1), ..., d(\phi_n)\} + 1$ for $\phi_0\{\phi_1, ..., \phi_n\}$. Next, define the ordering $\bar{d}$ to be the lexicographic ordering that orders over the value of d first, over the value of d on the composition's head when d is equal, and recursively over the value of $\bar{d}$ on the composition's arguments (lexicographically on the ordering of the arguments) in other cases. Since d is greater than or equal to 0 in every second-order term, and non-compositions do not have a head or arguments, there is a minimum, corresponding to non-compositions. We will now show that every rewrite rule reduces it.

The head simplification rule $\phi_0\{\pi_1^n, ..., \pi_n^n\} \to \phi_0$ reduces d from $d(\phi_0) + 0 + 1$ (since the π_i^n all have $d = 0$) to $d(\phi_0)$. The projection simplification rule $\pi_i^n\{\phi_1, ..., \phi_n\} \to \phi_i$ reduces d from $0 + \max\{d(\phi_1), ..., d(\phi_n)\} + 1$ to $d(\phi_i) \leq \max\{d(\phi_1), ..., d(\phi_n)\}$.

The function dumping rule preserves the value of d, going from

$$(d(\phi_0) + \max\{d(\phi_1), ..., d(\phi_n)\} + 1) + \max\{d(\psi_1), ..., d(\psi_m)\} + 1$$

to

$$d(\phi_0) + \max\{(d(\phi_1) + \max\{d(\psi_1), ..., d(\psi_m)\} + 1), ..., (d(\phi_n) + \max\{d(\psi_1), ..., d(\psi_m)\} + 1)\} + 1$$

Since the $\max\{d(\psi_1),...,d(\psi_m)\}$ is common among all elements over which the maximum is calculated in the last version, we can see these two sums are equal in all circumstances. However, the measure d on the head of the composition goes from $d(\phi_0)+\max\{d(\psi_1),...,d(\psi_m)\}+1$ to $d(\phi_0)$, so in the lexicographic ordering we have gone down.

Finally, we may recursively assume that when applying the inductive rule, the measure has been strictly reduced for the head or one of the arguments. d itself may not increase in this case, because it never increases under any reduction rule, recursively. It may, however, stay the same. In such case, if $\bar{d}$ has been reduced on the head, then $\bar{d}$ has been reduced on the composition by the lexicographic ordering. Similarly, when the head remained the same, arguments that have not been changed remain with the same value of $\bar{d}$, but there is at least one of them for which $\bar{d}$ has been reduced, and so their lexicographic ordering must have been reduced, so the overall value of $\bar{d}$ has been strictly reduced.

This concludes the proof of termination.

We will now show local confluence of the set of rules. There are four rules for reducibility other than reflexivity and transitivity: head simplification, projection simplification, function dumping and inductive reduction over the head or an argument. For each pair of these that can stem from the same source, we will show there is a common term they can both be reduced to. We write ϕ for the original second-order term.

If ϕ matches the conditions for head simplification and projection simplification, then it must be of the form $\pi_i^n\{\pi_1^n,...,\pi_n^n\}$. Head simplification reduces it to π_i^n, and projection simplification reduces it to π_i^n similarly, so they are locally confluent.

If ϕ matches head simplification and function dumping, then it must be of the form $\phi_0^n\{\phi_1^m,...,\phi_n^m\}\{\pi_1^m,...,\pi_m^m\}$. Head simplification reduces it to $\phi^1\equiv\phi_0^n\{\phi_1^m,...,\phi_n^m\}$ and function dumping to $\phi^2\equiv\phi_0^n\{\phi_1^m\{\pi_1^m,...,\pi_m^m\},...,\phi_n^m\{\pi_1^m,...,\pi_m^m\}\}$. But we may apply head simplification on each of the arguments of ϕ^2, via induction, to obtain $\phi^2\xrightarrow{*}\phi_0^n\{\phi_1^m,...,\phi_n^m\}\equiv\phi^1$, and so they are locally confluent.

If ϕ matches head simplification and inductive reduction, then it must be of the form $\phi_0^n\{\pi_1^n,...,\pi_n^n\}$, and there is a π_i^n that can be recursively reduced. But this is impossible since no rule's conditions matches π_i^n.

If ϕ matches projection simplification and function dumping, then it must be of the form $\pi_i^n\{\phi_1,...,\phi_n\}$ but also of the form $\phi_0^n\{\phi_1^m,...,\phi_n^m\}\{\psi_1,...,\psi_m\}$. But this is impossible since the head of this term must be π_i^n and also a composition $\phi_0^n\{\phi_1^m,...,\phi_n^m\}$.

If ϕ matches projection simplification and inductive reduction, then it must be of the form $\pi_i^n\{\phi_1, ..., \phi_n\}$, where there is a $\phi_j \xrightarrow{*} \phi_j^2$, and so we must conflate the result of applying projection simplification $\phi^1 \equiv \phi_i$ with the result of inductively reducing ϕ_j: $\phi^2 \equiv \pi_i^n\{\phi_1, ..., \phi_j^2, ..., \phi_n\}$. If $j \neq i$ then trivially $\phi^2 \to \phi_i \equiv \phi^1$ through projection simplification. If $j = i$, then $\phi^2 \to \phi_j^2$ through projection simplification, but then $\phi_i \equiv \phi_j$ and therefore $\phi^1 \equiv \phi_i \xrightarrow{*} \phi_j^2$, so they are locally confluent.

If ϕ matches function dumping and inductive reduction on the head, then it must be of the form

$$\phi_0^n\{\phi_1^m, ..., \phi_n^m\}\{\psi_1, ..., \psi_m\}$$

where $\phi_0^n\{\phi_1^m, ..., \phi_n^m\} \xrightarrow{*} \psi_0^m$, and we must conflate

$$\phi^1 \equiv \phi_0^n\{\phi_1^m\{\psi_1, ..., \psi_m\}, ..., \phi_n^m\{\psi_1, ..., \psi_m\}\}$$

with

$$\phi^2 \equiv \psi_0^m\{\psi_1, ..., \psi_m\}$$

For local confluence, we can consider only direct reduction steps on the head itself and use recursion, so we consider the cases for that:

- Head simplification.

  $$\phi \equiv \phi_0^n\{\pi_1^n, ..., \pi_n^n\}\{\psi_1, ..., \psi_m\}$$

  It must then be $n = m$ and so

  $$\psi_0^m \equiv \phi_0^n$$

  and

  $$\phi^1 \equiv \phi_0^n\{\pi_1^n\{\psi_1, ..., \psi_n\}, ..., \pi_n^n\{\psi_1, ..., \psi_n\}\}$$

  which by inductively applying projection simplification on each argument reduces to

  $$\phi^1 \xrightarrow{*} \phi_0^n\{\psi_1, ..., \psi_n\}$$

On the other hand

$$\phi^2 \equiv \psi_0^m\{\psi_1, ..., \psi_m\} \equiv \phi_0^n\{\psi_1, ..., \psi_n\}$$

so they are locally confluent.

- Projection simplification.

$$\phi \equiv \pi_i^n\{\phi_1^m, ..., \phi_n^m\}\{\psi_1, ..., \psi_m\}$$

and so

$$\psi_0^m \equiv \phi_i^m$$

and

$$\phi^1 \equiv \pi_i^n\{\phi_1^m\{\psi_1, ..., \psi_m\}, ..., \phi_n^m\{\psi_1, ..., \psi_m\}\}$$

which after applying projection simplification reduces to

$$\phi^1 \xrightarrow{*} \phi_i^m\{\psi_1, ..., \psi_m\}$$

On the other hand

$$\phi^2 \equiv \psi_0^m\{\psi_1, ..., \psi_m\} \equiv \phi_i^m\{\psi_1, ..., \psi_m\}$$

so they are locally confluent.

- Function dumping.

$$\phi \equiv \phi_{0,0}^p\{\phi_1^n, ..., \phi_p^n\}\{\phi_1^m, ..., \phi_n^m\}\{\psi_1, ..., \psi_m\}$$

and so

$$\psi_0^m \equiv \phi_{0,0}^p\{\phi_1^n\{\phi_1^m, ..., \phi_n^m\}, ..., \phi_p^n\{\phi_1^m, ..., \phi_n^m\}\}$$

and

$$\phi^1 \equiv \phi^p_{0,0}\{\phi^n_1,...,\phi^n_p\}\{\phi^m_1\{\psi_1,...,\psi_m\},...,\phi^m_n\{\psi_1,...,\psi_m\}\}$$

which by applying function dumping reduces to

$$\phi^1 \xrightarrow{*} \phi^p_{0,0}\{\phi^n_1\{\phi^m_1\{\psi_1,...,\psi_m\},...,\phi^m_n\{\psi_1,...,\psi_m\}\},...,\phi^n_p\{\phi^m_1\{\psi_1,...,\psi_m\},...,\phi^m_n\{\psi_1,...,\psi_m\}\}\}$$

On the other hand

$$\phi^2 \equiv \psi^m_0\{\psi_1,...,\psi_m\} \equiv \phi^p_{0,0}\{\phi^n_1\{\phi^m_1,...,\phi^m_n\},...,\phi^n_p\{\phi^m_1,...,\phi^m_n\}\}\{\psi_1,...,\psi_m\}$$

which by applying function dumping, and then function dumping inductively on each argument, reduces to

$$\phi^2 \xrightarrow{*} \phi^p_{0,0}\{\phi^n_1\{\phi^m_1\{\psi_1,...,\psi_m\},...,\phi^m_n\{\psi_1,...,\psi_m\}\},...,\phi^n_p\{\phi^m_1\{\psi_1,...,\psi_m\},...,\phi^m_n\{\psi_1,...,\psi_m\}\}\}$$

so they are locally confluent.

- Inductive reduction on the head. So there is a $\psi^n_{0,0}$ such that

$$\phi^n_0 \xrightarrow{*} \psi^n_{0,0}$$

and

$$\phi^m_0 \equiv \psi^n_{0,0}\{\phi^m_1,...,\phi^m_n\}$$

and so

$$\phi^1 \equiv \phi^n_0\{\phi^m_1\{\psi_1,...,\psi_m\},...,\phi^m_n\{\psi_1,...,\psi_m\}\}$$

which by inductive reduction on the head reduces to

$$\phi^1 \xrightarrow{*} \psi^n_{0,0}\{\phi^m_1\{\psi_1,...,\psi_m\},...,\phi^m_n\{\psi_1,...,\psi_m\}\}$$

On the other hand

$$\phi^2 \equiv \psi_0^m\{\psi_1,...,\psi_n\} \equiv \psi_{0,0}^n\{\phi_1^m,...,\phi_n^m\}\{\psi_1,...,\psi_m\}$$

which by function dumping reduces to

$$\phi^2 \xrightarrow{*} \psi_{0,0}^n\{\phi_1^m\{\psi_1,...,\psi_m\},...,\phi_n^m\{\psi_1,...,\psi_m\}\}$$

so they are locally confluent.

- Inductive reduction on an argument. So there is a ψ_i^m such that

$$\phi_i^m \xrightarrow{*} \psi_i^m$$

and

$$\phi_0^m \equiv \phi_0^n\{\phi_1^m,...,\psi_i^m,...,\phi_n^m\}$$

and so

$$\phi^1 \equiv \phi_0^n\{\phi_1^m\{\psi_1,...,\psi_m\},...,\phi_n^m\{\psi_1,...,\psi_m\}\}$$

which by inductive reduction reduces to

$$\phi^1 \xrightarrow{*} \phi_0^n\{\phi_1^m\{\psi_1,...,\psi_m\},...,\psi_i^m\{\psi_1,...,\psi_m\},...,\phi_n^m\{\psi_1,...,\psi_m\}\}$$

On the other hand

$$\phi^2 \equiv \phi_0^m\{\psi_1,...,\psi_m\} \equiv \phi_0^n\{\phi_1^m,...,\psi_i^m,...,\phi_n^m\}\{\psi_1,...,\psi_m\}$$

which by funciton dumping reduces to

$$\phi^2 \xrightarrow{*} \phi_0^n\{\phi_1^m\{\psi_1,...,\psi_m\},...,\psi_i^m\{\psi_1,...,\psi_m\},...,\phi_n^m\{\psi_1,...,\psi_m\}\}$$

so they are locally confluent.

If ϕ matches function dumping and inductive reduction on an argument, then it must be of the form $\phi_0^n\{\phi_1^m,...,\phi_n^m\}\{\psi_1,...,\psi_m\}$, where there is a $\psi_j \xrightarrow{*} \psi_j^2$, and so we must conflate the result of applying function dumping

$$\phi^1 \equiv \phi_0^n\{\phi_1^m\{\psi_1,...,\psi_m\},...,\phi_n^m\{\psi_1,...,\psi_m\}\}$$

with the result of inductively reducing ψ_j:

$$\phi^2 \equiv \phi_0^n\{\phi_1^m,...,\phi_n^m\}\{\psi_1,...,\psi_j^2,...,\psi_m\}$$

But we may apply inductive reduction over $\psi_j \xrightarrow{*} \psi_j^2$ on each argument of ϕ^1 to obtain

$$\phi^1 \xrightarrow{*} \phi_0^n\{\phi_1^m\{\psi_1,...,\psi_j^2,...,\psi_m\},...,\phi_n^m\{\psi_1,...,\psi_j^2,...,\psi_m\}\}$$

and apply function dumping on ϕ^2 to obtain

$$\phi^2 \xrightarrow{*} \phi_0^n\{\phi_1^m\{\psi_1,...,\psi_j^2,...,\psi_m\},...,\phi_n^m\{\psi_1,...,\psi_j^2,...,\psi_m\}\}$$

so they are locally confluent.

Finally, if ϕ matches inductive reduction on the head and on an argument, then it must be of the form $\phi_0^m\{\psi_1,...,\psi_m\}$, and there is a ψ_0^m such that $\phi_0^m \xrightarrow{*} \psi_0^m$ and a ψ_j^2 such that $\psi_j \xrightarrow{*} \psi_j^2$, and so we must conflate

$$\phi^1 \equiv \psi_0^m\{\psi_1,...,\psi_m\}$$

with

$$\phi^2 \equiv \phi_0^m\{\psi_1,...,\psi_j^2,...,\psi_m\}$$

But we may apply inductive reduction on the argument ψ_j to reduce

$$\phi^1 \xrightarrow{*} \psi_0^m\{\psi_1,...,\psi_j^2,...,\psi_m\}$$

and similarly apply inductive reduction on the head to reduce

$$\phi^2 \xrightarrow{*} \psi_0^m\{\psi_1,...,\psi_j^2,...,\psi_m\}$$

so they are locally confluent.

Thus, the rewrite system is locally confluent and terminating, and therefore, by the diamond lemma, it is confluent. Since we have proven that second-order normal forms

correspond exactly to the irreducible terms in this rewrite system, we have shown that the rewrite system reduces every second-order term to a unique equivalent second-order normal form.

□

Theorem B.0.2 (Normalization of first-order terms)**.** *Every first-order term* α *is equivalent to a unique normal first-order term* $\mathcal{N}(\alpha)$.

This theorem was originally presented in theorem 6.1.2.

Proof. The proof is analogous to the one for second-order terms. We begin by showing that our notion of normality (definition 6.1.8) corresponds to normality over first-order reducibility. That is, we show that a first-order term is normal if and only if it is not reducible.

Reducibility may be produced through one of two direct reducibility rules, transitively or through structural induction over the term structure. Assume α is a normal first-order term. Then, it is either a first-order variable or, if it is an application, its head is not a composition nor a projection, and all its arguments are recursively normal. If it is a first-order variable, then neither of the direct reducibility rules apply, and it is not an inductively defined first-order term, so it is not reducible. Projection simplification cannot be applied because the head is not a projection, and function dumping cannot be applied because its head is not a composition. Finally, inductive reduction cannot be applied because the head is not a composition (and therefore cannot be reduced) and the arguments are recursively assumed to be normal.

In the other direction, we will show that if a first-order term is not normal, then it can be reduced. If α is not normal, then it must be an application, whose head is a composition or a projection, or one of its arguments is not normal. If any of its arguments are not normal, then we may recursively assume we may reduce those through inductive reduction. Otherwise, α must be an application, and its head a composition or a projection. If its head is a projection, then the projection simplification rule may be applied. If its head is a composition, then the function dumping rule may be applied. In either case, it is reducible.

We will now show that the rewrite system $\xrightarrow{*}$ is confluent. In order to do this, we will show it is locally confluent and terminating, and apply the diamond lemma. This, together with the irreducibility of normal forms, implies the theorem.

We first show termination. Define a measure d on second-order terms indicating the depth of compositions, defined as 0 for non-compositions and $d(\phi_0) + \max\{d(\phi_1), ..., d(\phi_n)\} + 1$ for $\phi_0\{\phi_1, ..., \phi_n\}$. Next, define the measure d^* on first-order terms to be 0 for first-order variables and $d(\phi) + \max\{d^*(\alpha_1), ..., d^*(\alpha_n)\} + 1$ for the application $\phi(\alpha_1, ..., \alpha_n)$. Finally, define the ordering $\bar{d}$ on first-order terms to be the lexicographic ordering that orders over the value of d^* first, over the value of d on the application's head when d^* is equal, and recursively over $\bar{d}$ on the arguments of the application in other cases. Since d^* is greater than or equal to 0 in every first-order term, and non-applications do not have arguments, there is a minimum to $\bar{d}$, corresponding to non-applications. We will now show that every rewrite rule strictly reduces $\bar{d}$.

The projection simplification rule $\pi_i^n(\alpha_1, ..., \alpha_n) \to \alpha_i$ reduces d^* from $0 + \max\{d(\alpha_1), ..., d(\alpha_n))\} + 1$ to $d(\alpha_i) \leq \max\{d(\alpha_1), ..., d(\alpha_n)\}$.

The function dumping rule preserves the value of d^*, going from

$$(d(\phi_0) + \max\{d(\phi_1), ..., d(\phi_n)\} + 1) + \max\{d^*(\alpha_1), ..., d^*(\alpha_m)\} + 1$$

to

$$d(\phi_0) + \max\{(d(\phi_1) + \max\{d^*(\alpha_1), ..., d^*(\alpha_m)\} + 1), ..., (d(\phi_n) + \max\{d^*(\alpha_1), ..., d^*(\alpha_m)\} + 1)\} + 1$$

Since the $\max\{d^*(\alpha_1), ..., d^*(\alpha_m)\}$ is common among all elements over which the maximum is calculated in the last version, we can see these two sums are equal in all circumstances. However, the measure d on the head of the application goes from $d(\phi_0) + \max\{d(\alpha_1), ..., d(\alpha_m)\} + 1$ to $d(\phi_0)$, so in the lexicographic ordering we have gone down.

Finally, we may recursively assume when applying the inductive rule, that either d has been strictly reduced for the head, or d^* has been strictly reduced for one of the arguments. d^* itself may not increase in this case, because it never increases under any reduction rule, recursively. It may, however, stay the same. In such case, if d has been reduced on the head, then $\bar{d}$ has been reduced on the application by the lexicographic ordering. Similarly, when the head remained the same, arguments that have not changed remain with the same value of $\bar{d}$, but there is at least one of them for which $\bar{d}$ has been reduced, and so their lexicographic ordering must have been reduced, so the overall value of $\bar{d}$ has been strictly reduced.

This concludes the proof of termination.

We will now show local confluence of the set of rules. There are four rules for reducibility other than reflexivity and transitivity: projection simplification, function dumping and inductive reduction over the head or an argument. For each pair of these that can stem from the same source, we will show there is a common term they can both be reduced to. We write α for the original first-order term.

If α matches the conditions for projection simplification and function dumping, then it must be of the form $\pi_i^m(\alpha_1, ..., \alpha_m)$ but also of the form $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}(\alpha_1, ..., \alpha_m)$. But this is impossible since the head of this term must be π_i^m and also a composition $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}$.

If α matches projection simplification and inductive reduction, then it must be of the form $\pi_i^n(\alpha_1, ..., \alpha_n)$, where there is a $\alpha_j \xrightarrow{*} \alpha_j^2$ (because inductive reduction of the head cannot possibly match), and so we must conflate the result of applying projection simplification $\alpha^1 \equiv \alpha_i$ with the result of inductively reducing α_j: $\alpha^2 \equiv \pi_i^n(\alpha_1, ..., \alpha_j^2, ..., \alpha_n)$. If $j \neq i$ then trivially $\alpha^2 \to \alpha_i \equiv \alpha^1$. If $j = i$, then $\alpha^2 \to \alpha_j^2$ through projection simplification, but then $\alpha_i \equiv \alpha_j$ and therefore $\alpha^1 \equiv \alpha_i \xrightarrow{*} \alpha_j^2$, so they are locally confluent.

If α matches function dumping and inductive simplification of the head, then it must be of the form $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}(\alpha_1, ..., \alpha_m)$, where $\phi_0^n\{\phi_1^m, ..., \phi_n^m\} \to \psi_0^m$, and so we must conflate $\alpha^1 \equiv \phi_0^n(\phi_1^m(\alpha_1, ..., \alpha_m), ..., \phi_n^m(\alpha_1, ..., \alpha_m))$ with $\alpha^2 \equiv \psi_0^m(\alpha_1, ..., \alpha_m)$. For local confluence, we can consider only direct reduction steps on the head itself and use recursion, so we consider the cases for that:

- Head simplification.

  $$\alpha \equiv \phi_0^n\{\pi_1^n, ..., \pi_n^n\}(\alpha_1, ..., \alpha_m)$$

  It must then be $n = m$ and so

  $$\psi_0^m \equiv \phi_0^n$$

  and

  $$\alpha^1 \equiv \phi_0^n(\pi_1^n(\alpha_1, ..., \alpha_n), ..., \pi_n^n(\alpha_1, ..., \alpha_n))$$

  which by inductively applying projection simplification on each argument reduces to

$$\alpha^1 \xrightarrow{*} \phi_0^n(\alpha_1, ..., \alpha_n)$$

In the other hand

$$\alpha^2 \equiv \psi_0^m(\alpha_1, ..., \alpha_m) \equiv \phi_0^n(\alpha_1, ..., \alpha_n)$$

so they are locally confluent.

- Projection simplification.

$$\alpha \equiv \pi_i^n\{\phi_1^m, ..., \phi_n^m\}(\alpha_1, ..., \alpha_m)$$

and so

$$\psi_0^m \equiv \phi_i^m$$

and

$$\alpha^1 \equiv \pi_i^n(\phi_1^m(\alpha_1, ..., \alpha_m), ..., \phi_n^m(\alpha_1, ..., \alpha_m))$$

which after applying projection simplification reduces to

$$\alpha^1 \xrightarrow{*} \phi_i^m(\alpha_1, ..., \alpha_m)$$

On the other hand

$$\alpha^2 \equiv \psi_0^m(\alpha_1, ..., \alpha_m) \equiv \phi_i^m(\alpha_1, ..., \alpha_m)$$

so they are locally confluent.

- Function dumping.

$$\alpha \equiv \phi_{0,0}^p\{\phi_1^n, ..., \phi_p^n\}\{\phi_1^m, ..., \phi_n^m\}(\alpha_1, ..., \alpha_m)$$

and so

$$\psi_0^m \equiv \phi_{0,0}^p\{\phi_1^n\{\phi_1^m, ..., \phi_n^m\}, ..., \phi_p^n\{\phi_1^m, ..., \phi_n^m\}\}$$

and

$$\alpha^1 \equiv \phi^p_{0,0}\{\phi^n_1,...,\phi^n_p\}(\phi^m_1(\alpha_1,...,\alpha_m),...,\phi^m_n(\alpha_1,...,\alpha_m))$$

which by applying function dumping reduces to

$$\alpha^1 \xrightarrow{*} \phi^p_{0,0}(\phi^n_1(\phi^m_1(\alpha_1,...,\alpha_m),...,\phi^m_n(\alpha_1,...,\alpha_m)),...,\phi^n_p(\phi^m_1(\alpha_1,...,\alpha_m),...,\phi^m_n(\alpha_1,...,\alpha_m)))$$

On the other hand

$$\alpha^2 \equiv \psi^m_0(\alpha_1,...,\alpha_m) \equiv \phi^p_{0,0}\{\phi^n_1\{\phi^m_1,...,\phi^m_n\},...,\phi^n_p\{\phi^m_1,...,\phi^m_n\}\}(\alpha_1,...,\alpha_m)$$

which by applying function dumping, and then function dumping inductively on each argument, reduces to

$$\alpha^2 \xrightarrow{*} \phi^p_{0,0}(\phi^n_1(\phi^m_1(\alpha_1,...,\alpha_m),...,\phi^m_n(\alpha_1,...,\alpha_m)),...,\phi^n_p(\phi^m_1(\alpha_1,...,\alpha_m),...,\phi^m_n(\alpha_1,...,\alpha_m)))$$

so they are locally confluent.

- Inductive reduction on the head. So there is a $\psi^n_{0,0}$ such that

$$\phi^n_0 \xrightarrow{*} \psi^n_{0,0}$$

and

$$\phi^m_0 \equiv \psi^n_{0,0}\{\phi^m_1,...,\phi^m_n\}$$

and so

$$\alpha^1 \equiv \phi^n_0(\phi^m_1(\alpha_1,...,\alpha_m),...,\phi^m_n(\alpha_1,...,\alpha_m))$$

which by inductive reduction on the head reduces to

$$\alpha^1 \xrightarrow{*} \psi^n_{0,0}(\phi^m_1(\alpha_1,...,\alpha_m),...,\phi^m_n(\alpha_1,...,\alpha_m))$$

On the other hand

$$\alpha^2 \equiv \psi_0^m(\alpha_1, ..., \alpha_m) \equiv \psi_{0,0}^n\{\phi_1^m, ..., \phi_n^m\}(\alpha_1, ..., \alpha_m)$$

which by function dumping reduces to

$$\alpha^2 \xrightarrow{*} \psi_{0,0}^n(\phi_1^m(\alpha_1, ..., \alpha_m), ..., \phi_n^m(\alpha_1, ..., \alpha_m))$$

so they are locally confluent.

- Inductive reduction on an argument. So there is a ψ_i^m such that

$$\phi_i^m \xrightarrow{*} \psi_i^m$$

and

$$\phi_0^m \equiv \phi_0^n\{\phi_1^m, ..., \psi_i^m, ..., \phi_n^m\}$$

and so

$$\alpha^1 \equiv \phi_0^n(\phi_1^m(\alpha_1, ..., \alpha_m), ..., \phi_n^m(\alpha_1, ..., \alpha_m))$$

which by inductive reduction reduces to

$$\alpha^1 \xrightarrow{*} \phi_0^n(\phi_1^m(\alpha_1, ..., \alpha_m), ..., \psi_i^m(\alpha_1, ..., \alpha_m), ..., \phi_n^m(\alpha_1, ..., \alpha_m))$$

On the other hand

$$\alpha^2 \equiv \phi_0^m(\alpha_1, ..., \alpha_m) \equiv \phi_0^n\{\phi_1^m, ..., \psi_i^m, ..., \phi_n^m\}(\alpha_1, ..., \alpha_m)$$

which by function dumping reduces to

$$\alpha^2 \xrightarrow{*} \phi_0^n(\phi_1^m(\alpha_1, ..., \alpha_m), ..., \psi_i^m(\alpha_1, ..., \alpha_m), ..., \phi_n^m(\alpha_1, ..., \alpha_m))$$

so they are locally confluent.

If α matches function dumping and inductive reduction on an argument, then it must be of the form $\phi_0^n\{\phi_1^m,...,\phi_n^m\}(\alpha_1,...,\alpha_m)$, where there is a $\alpha_j \xrightarrow{*} \alpha_j^2$, and so we must conflate the result of applying function dumping

$$\alpha^1 \equiv \phi_0^n(\phi_1^m(\alpha_1,...,\alpha_m),...,\phi_n^m(\alpha_1,...,\alpha_m))$$

with the result of inductively reducing α_j:

$$\alpha^2 \equiv \phi_0^n\{\phi_1^m,...,\phi_n^m\}(\alpha_1,...,\alpha_j^2,...,\alpha_m)$$

But we may apply inductive reduction over $\alpha_j \xrightarrow{*} \alpha_j^2$ on each argument of α^1 to obtain

$$\alpha^1 \xrightarrow{*} \phi_0^n(\phi_0^m(\alpha_1,...,\alpha_j^2,...,\alpha_m),...,\phi_n^m(\alpha_1,...,\alpha_j^2,...,\alpha_m))$$

and apply function dumping on α^2 to obtain

$$\alpha^2 \xrightarrow{*} \phi_0^n(\phi_1^m(\alpha_1,...,\alpha_j^2,...,\alpha_m),...,\phi_n^m(\alpha_1,...,\alpha_j^2,...,\alpha_m))$$

so they are locally confluent.

Finally, if α matches inductive reduction on the head and on an argument, then it must be of the form $\phi_0^m(\alpha_1,...,\alpha_m)$, and there is a ψ_0^m such that $\phi_0^m \xrightarrow{*} \psi_0^m$ and a α_j^2 such that $\alpha_j \xrightarrow{*} \alpha_j^2$, and so we must conflate

$$\alpha^1 \equiv \psi_0^m(\alpha_1,...,\alpha_m)$$

with

$$\alpha^2 \equiv \phi_0^m(\alpha_1,...,\alpha_j^2,...,\alpha_m)$$

But we may apply inductive reduction on the argument α_j to reduce

$$\alpha^1 \xrightarrow{*} \psi_0^m(\alpha_1,...,\alpha_j^2,...,\alpha_m)$$

and similarly apply inductive reduction on the head to reduce

$$\alpha^2 \xrightarrow{*} \psi_0^m(\alpha_1,...,\alpha_j^2,...,\alpha_m)$$

so they are locally confluent.

Thus, the rewrite system is locally confluent and terminating, and therefore, by the diamond lemma, it is confluent. Since we have proven that second-order normal forms correspond exactly to the irreducible terms in this rewrite system, we have shown that the rewrite system reduces every first-order term to a unique equivalent first-order normal form.

□

Theorem B.0.3 (Extensionality of second-order terms)**.** *Two second-order terms* ϕ_1 *and* ϕ_2*, both with arity m, are equivalent if and only for all sequences of first-order terms* $\alpha_1, ..., \alpha_m$*,* $\phi_1(\alpha_1, ..., \alpha_m)$ *and* $\phi_2(\alpha_1, ..., \alpha_m)$ *are equivalent.*

This theorem was originally presented in theorem 6.1.3.

Proof. First, assume that ϕ_1 and ϕ_2 are equivalent, and take an arbitrary sequence of first-order terms $\alpha_1, ..., \alpha_m$. By corollary 6.1.1, ϕ_1 and ϕ_2 have the same normal form $\mathcal{N}(\phi)$, such that $\phi_1 \xrightarrow{*} \mathcal{N}(\phi)$ and $\phi_2 \xrightarrow{*} \mathcal{N}(\phi)$. By definition of first-order reduction, this implies that $\phi_1(\alpha_1, ..., \alpha_m) \xrightarrow{*} \mathcal{N}(\phi)(\alpha_1, ..., \alpha_m)$ and $\phi_2(\alpha_1, ..., \alpha_m) \xrightarrow{*} \mathcal{N}(\phi)(\alpha_1, ..., \alpha_m)$. Therefore, $\phi_1(\alpha_1, ..., \alpha_m) \cong \mathcal{N}(\phi)(\alpha_1, ..., \alpha_m) \cong \phi_2(\alpha_1, ..., \alpha_m)$.

Now assume that for all sequences of first-order terms $\alpha_1, ..., \alpha_m$, $\alpha^1 \equiv \phi_1(\alpha_1, ..., \alpha_m) \cong \phi_2(\alpha_1, ..., \alpha_m) \equiv \alpha^2$. In particular, let $\alpha_i \equiv X_i$ for distinct first-order variables X_i. ϕ_1 and ϕ_2 have their respective normal forms $\phi_1 \cong \mathcal{N}(\phi_1)$ and $\phi_2 \cong \mathcal{N}(\phi_2)$. By inductive reduction, this implies that $\phi_1(X_1, ..., X_m) \cong \mathcal{N}(\phi_1)(X_1, ..., X_m)$ and $\phi_2(X_1, ..., X_m) \cong \mathcal{N}(\phi_2)(X_1, ..., X_m)$. Consider the potential cases for each of $\mathcal{N}(\phi_1)$ and $\mathcal{N}(\phi_2)$, and for each of them, calculate the normal form of α^1 and α^2:

- $\mathcal{N}(\phi_k) \equiv f$ for function symbol f. Then, $\mathcal{N}(\alpha^k) \equiv f(X_1, ..., X_m)$.
- $\mathcal{N}(\phi_k) \equiv F$ for second-order variable F. Then, $\mathcal{N}(\alpha^k) \equiv F(X_1, ..., X_m)$.
- $\mathcal{N}(\phi_k) \equiv \pi_i^m$. Then, $\alpha^k \xrightarrow{*} \pi_i^m(X_1, ..., X_m) \to X_i \equiv \mathcal{N}(\alpha^k)$.
- $\mathcal{N}(\phi_k) \equiv f^n\{\psi_1^k, ...\psi_n^k\}$ for function symbol f with arity n, where each ψ_j^k is normal.

  Then, $\alpha^k \xrightarrow{*} f^n\{\psi_1^k, ..., \psi_n^k\}(X_1, ..., X_m) \xrightarrow{*} f^n(\psi_1^k(X_1, ..., X_m), ..., \psi_n^k(X_1, ..., X_m))$. Since each ψ_j^k is normal, then either this is the normal form, or (if any of them are projections) some (though not all) of the $\psi_1^k(X_1, ..., X_m)$ may reduce to X_j, or (if any of them are normal compositions) some may reduce to higher depth applications.

- $\mathcal{N}(\phi_k) \equiv F^n\{\psi_1^k, ..., \psi_n^k\}$ for second-order variable F with arity n, where each ψ_j^k is normal.

  Similarly to the previous case, $\alpha^k \xrightarrow{*} F^n(\psi_1^k(X_1, ..., X_m), ..., \psi_n^k(X_1, ..., X_m))$, where some, though not all, of the arguments may reduce to X_j and some may reduce to higher depth applications.

From the previous list, and by looking at whether the head is a function symbol or a second-order variable (or the term is a first-order variable) and the maximum depth of applications present in the first-order term, we gather that if $\alpha^1 \cong \alpha^2$ then, by corollary 6.1.1, we know that $\mathcal{N}(\alpha^1) \equiv \mathcal{N}(\alpha^2)$, and this may only happen if both ϕ_1 and ϕ_2 fall under the same category of the five kinds of normal forms indicated above, because each of them produce different heads and/or maximum application depths for $\mathcal{N}(\alpha^k)$. Moreover, within each category, it is only possible that $\mathcal{N}(\alpha^1) \equiv \mathcal{N}(\alpha^2)$ if $\mathcal{N}(\phi_1) \equiv \mathcal{N}(\phi_2)$, and so $\phi_1 \cong \phi_2$.

□

Theorem B.0.4 (Normalization of unifier expressions)**.** *Every unifier expression* ε *is equivalent to a unique normal unifier expression* $\mathcal{N}(\varepsilon)$.

This theorem was originally presented in theorem 6.1.4.

Proof. The proof is analogous to the one for first-order terms. We begin by showing that our notion of normality (definition 6.1.21) corresponds to normality over unifier expression reducibility. That is, we show that a unifier expression is normal if and only if it is not reducible.

Reducibility may be produced through one of three direct reducibility rules, transitively or through structural induction over the expression structure. Assume ε is a normal unifier expression. Then, it is function free or its head is a function symbol or a second-order variable; and all its arguments are recursively in normal form. If it is function free then it either is a first-order variable or $\sigma_i\delta$, where δ is a recursively function free expression. If it is a first-order variable then it does not match any of the reduction rule heads or has any inductive structure over which to reduce, so it is not reducible. If it is of the form $\sigma_i\delta$, where δ is function free, then we can recursively assume δ is not reducible, and moreover, we know that δ is not an application. Projection simplification and function dumping do not match, and neither does unifier variable dumping because δ is not an application. And since δ is not reducible, then ε cannot be inductively reduced either. So ε is not reducible. The only case left is if $\varepsilon \equiv \phi(\varepsilon_1, ..., \varepsilon_n)$,

where ϕ is a function symbol or a second-order variable, and all its arguments are recursively in normal form. Projection simplification does not match because ϕ is not a projection. Neither does function dumping because ϕ is not a composition. Unifier variable dumping does not match either because it has no unifier variable. Moreover, because ϕ is a function symbol or second-order variable, it is normal and therefore irreducible. Similarly, the ε_i are normal by assumption and therefore we can recursively assume them to be irreducible. Thus, inductive reduction does not apply either. So ε is irreducible.

On the other direction, we will show that if ε is not normal, then it can be reduced. If ε is not normal, then it must not be function free, and it must be a substitution or an application whose head is a composition or a projection. If ε is a substitution, it is of the form $\varepsilon \equiv \sigma_i\delta$. But δ may not be function free, because ε is not function free, and so δ cannot be normal unless it is an application. If δ is an application, then $\varepsilon \equiv \sigma_i\phi(\varepsilon_1, ..., \varepsilon_n)$ and so the unifier variable dumping rule applies. If δ is not normal, then we may recursively assume it is reducible and therefore ε is inductively reducible. The only case left is when ε is an application whose head is a composition or a projection. But if ε is an application whose head is a composition then the function dumping rule applies. Finally, if ε is an application whose head is a projection then the projection simplification rule applies. In any case, ε is reducible.

We will now show that the rewrite system $\xrightarrow{*}$ is confluent. In order to do this, we will show it is locally confluent and terminating, and apply the diamond lemma. This, together with the irreducibility of normal forms, implies the theorem.

We first show termination. First, define a measure d on second-order terms indicating the depth of compositions, defined as 0 for non-compositions and $d(\phi_0) + \max\{d(\phi_1), ..., d(\phi_n)\} + 1$ for $\phi_0\{\phi_1, ..., \phi_n\}$. Next, define the measure d^* on unifier expressions to be 0 for first-order variables, $d(\phi) + \max\{d^*(\varepsilon_1), ..., d^*(\varepsilon_n)\} + 1$ for the application $\phi(\varepsilon_1, ..., \varepsilon_n)$ and $d^*(\delta)$ for the substitution $\sigma_i\delta$. Finally, define the ordering $\bar{d}$ on unifier expressions to be the ordering that orders over the value of d^*, considers substitutions to be higher in the ordering than non-substitutions when d^* is equal, orders over the lexicographic ordering of the value of d on the application's head and the recursive value of $\bar{d}$ on the arguments when comparing two applications with the same value of d^*, and orders over the recursive value of $\bar{d}$ of the sub-expression when comparing two substitutions with the same value of $\bar{d}$. Since d^* is greater than or equal

to 0 in every unifier expression, and there is no recursive ordering in non-inductive expressions, there is a minimum to $\bar{d}$, corresponding to non-inductive expressions. We will now show that every rewrite rule strictly reduces $\bar{d}$.

The projection simplification rule $\pi_i^n(\varepsilon_1,...,\varepsilon_n) \to \varepsilon_i$ reduces d^* from $0 + \max\{d(\varepsilon_1),...,d(\varepsilon_n)\}+1$ to $d(\varepsilon_i) \leq \max\{d(\varepsilon_1),...,d(\varepsilon_n)\}$.

The function dumping rule preserves the value of d^*, going from

$$(d(\phi_0)+\max\{d(\phi_1),...,d(\phi_n)\}+1)+\max\{d^*(\varepsilon_1),...d^*(\varepsilon_m)\}+1$$

to

$$d(\phi_0)+\max\{(d(\phi_1)+\max\{d^*(\varepsilon_1),...,d^*(\varepsilon_m)\}+1),...,(d(\phi_n)+\max\{d^*(\varepsilon_1),...,d^*(\varepsilon_m)\}+1)\}+1$$

Since the $\max\{d^*(\varepsilon_1),...,d^*(\varepsilon_m)\}$ is common among all elements over which the maximum is calculated in the last version, we can see these two sums are equal in all circumstances. However, the measure d on the head of the application goes from $d(\phi_0)+\max\{d(\varepsilon_1),...,d(\varepsilon_m)\}+1$ to $d(\phi_0)$, so in the lexicographic ordering we have gone down.

The unifier variable dumping rule preserves the value of d^* as well, since substitutions do not change the value of d^* on either side of the rule, and they are equal otherwise.

Finally, we may recursively assume when applying the inductive rule, that either d has been strictly reduced for the head, or d^* has been strictly reduced for one of the arguments, or d^* has been strictly reduced for the sub-expression. d^* itself may not increase in this case, because it never increases under any reduction rule, recursively. It may, however, stay the same. In such case, if d has been reduced on the head, then $\bar{d}$ has been reduced on the application by the lexicographic ordering. Similarly, when the head remained the same, arguments that have not changed remain with the same value of $\bar{d}$, but there is at least one of them for which $\bar{d}$ has been reduced, and so their lexicographic ordering must have been reduced, so the overall value of $\bar{d}$ has been strictly reduced. Similarly, if it is a substitution, then the value of $\bar{d}$ must have been strictly reduced on the sub-expression, and therefore so has it on the substitution.

This concludes the proof of termination.

We will now show local confluence of the set of rules. There are six rules for reducibility other than reflexivity and transitivity: projection simplification, function

dumping, unifier variable dumping and inductive reduction over the head, an argument, or the sub-expression. For each pair of these that can stem from the same source, we will show there is a common term they can both be reduced to. We write ε for the original unifier expression.

If ε matches the conditions for projection simplification and function dumping, then it must be of the form $\pi_i^m(\varepsilon_1, ..., \varepsilon_m)$ but also of the form $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}(\varepsilon_1, ..., \varepsilon_m)$. But this is impossible since the head of this term must be π_i^m and also a composition $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}$.

ε may not match projection simplification and unifier variable dumping either, since then it would have to be a substitution and an application at the same time.

If ε matches projection simplification and inductive reduction, then it must be of the form $\pi_i^n(\varepsilon_1, ..., \varepsilon_n)$, where there is a $\varepsilon_j \xrightarrow{*} \varepsilon_j^2$ (because inductive reduction of the head cannot possibly match), and so we must conflate the result of applying projection simplification $\varepsilon^1 \equiv \varepsilon_i$ with the result of inductively reducing ε_j: $\varepsilon^2 \equiv \pi_i^n(\varepsilon_1, ..., \varepsilon_j^2, ..., \varepsilon_n)$. If $j \neq i$ then trivially $\varepsilon^2 \to \varepsilon_i \equiv \varepsilon^1$. If $j = i$, then $\varepsilon^2 \to \varepsilon_j^2$ through projection simplification, but then $\varepsilon_i \equiv \varepsilon_j$ and therefore $\varepsilon^1 \equiv \varepsilon_i \xrightarrow{*} \varepsilon_j^2$, so they are locally confluent.

ε may not match function dumping and unifier variable dumping, since then it would have to be a substitution and an application at the same time.

If ε matches function dumping and inductive simplification of the head, then it must be of the form $\phi_0^n\{\phi_1^m, ..., \phi_n^m\}(\varepsilon_1, ..., \varepsilon_m)$, where $\phi_0^n\{\phi_1^m, ..., \phi_n^m\} \to \psi_0^m$, and so we must conflate $\varepsilon^1 \equiv \phi_0^n(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m))$ with $\varepsilon^2 \equiv \psi_0^m(\varepsilon_1, ..., \varepsilon_m)$. For local confluence, we can consider only direct reduction steps on the head itself and use recursion, so we consider the cases for that:

- Head simplification.

$$\varepsilon \equiv \phi_0^n\{\pi_1^m, ..., \pi_n^n\}(\varepsilon_1, ..., \varepsilon_m)$$

  It must then be $n = m$ and so

$$\psi_0^m \equiv \phi_0^n$$

  and

$$\varepsilon^1 \equiv \phi_0^n(\pi_1^n(\varepsilon_1, ..., \varepsilon_n), ..., \pi_n^n(\varepsilon_1, ..., \varepsilon_n))$$

which by inductively applying projection simplification on each argument reduces to

$$\varepsilon^1 \xrightarrow{*} \phi_0^n(\varepsilon_1, ..., \varepsilon_n)$$

On the other hand

$$\varepsilon^2 \xrightarrow{*} \psi_0^m(\varepsilon_1, ..., \varepsilon_m) \equiv \phi_0^n(\varepsilon_1, ..., \varepsilon_n)$$

so they are locally confluent.

- Projection simplification.

$$\varepsilon \equiv \pi_i^n\{\phi_1^m, ..., \phi_n^m\}(\varepsilon_1, ..., \varepsilon_m)$$

and so

$$\psi_0^m \equiv \phi_i^m$$

and

$$\varepsilon^1 \equiv \pi_i^n(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m))$$

which after applying projection simplification reduces to

$$\varepsilon^1 \xrightarrow{*} \phi_i^m(\varepsilon_1, ..., \varepsilon_m)$$

On the other hand

$$\varepsilon^2 \equiv \psi_0^m(\varepsilon_1, ..., \varepsilon_m) \equiv \phi_i^m(\varepsilon_1, ..., \varepsilon_m)$$

so they are locally confluent.

- Function dumping.

$$\varepsilon \equiv \phi_{0,0}^p\{\phi_1^n, ..., \phi_p^n\}\{\phi_1^m, ..., \phi_n^m\}(\varepsilon_1, ..., \varepsilon_m)$$

and so

$$\psi_0^m \equiv \phi_{0,0}^p\{\phi_1^n\{\phi_1^m, ..., \phi_n^m\}, ..., \phi_p^n\{\phi_1^m, ..., \phi_n^m\}\}$$

and

$$\varepsilon^1 \equiv \phi_{0,0}^p\{\phi_1^n, ..., \phi_p^n\}(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m))$$

which by applying function dumping reduces to

$$\varepsilon^1 \xrightarrow{*} \phi_{0,0}^p(\phi_1^n(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m))), ..., \phi_p^n(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m)))$$

On the other hand

$$\varepsilon^2 \equiv \psi_0^m(\varepsilon_1, ..., \varepsilon_m) \equiv \phi_{0,0}^p\{\phi_1^n\{\phi_1^m, ..., \phi_n^m\}, ..., \phi_p^n\{\phi_1^m, ..., \phi_n^m\}\}(\varepsilon_1, ..., \varepsilon_m)$$

which by applying function dumping, and then function dumping inductively on each argument, reduces to

$$\varepsilon^2 \xrightarrow{*} \phi_{0,0}^p(\phi_1^n(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m))), ..., \phi_p^n(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m)))$$

so they are locally confluent.

- Inductive reduction on the head. So there is a $\psi_{0,0}^n$ such that

$$\phi_0^n \xrightarrow{*} \psi_{0,0}^n$$

and

$$\phi_0^m \equiv \psi_{0,0}^n\{\phi_1^m, ..., \phi_n^m\}$$

and so

$$\varepsilon^1 \equiv \phi_0^n(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_n^m(\varepsilon_1, ..., \varepsilon_m))$$

which by inductive reduction on the head reduces to

$$\varepsilon^1 \xrightarrow{*} \psi^n_{0,0}(\phi^m_1(\varepsilon_1,...,\varepsilon_m),...,\phi^m_n(\varepsilon_1,...,\varepsilon_m))$$

On the other hand

$$\varepsilon^2 \equiv \psi^m_0(\varepsilon_1,...,\varepsilon_m) \equiv \psi^n_{0,0}\{\phi^m_1,...,\phi^m_n\}(\varepsilon_1,...,\varepsilon_m)$$

which by function dumping reduces to

$$\varepsilon^2 \xrightarrow{*} \psi^n_{0,0}(\phi^m_1(\varepsilon_1,...,\varepsilon_m),...,\phi^m_n(\varepsilon_1,...,\varepsilon_m))$$

so they are locally confluent.

- Inductive reduction on an argument. So there is a ψ^m_i such that

$$\phi^m_i \xrightarrow{*} \psi^m_i$$

and

$$\phi^m_0 \equiv \phi^n_0\{\phi^m_1,...,\psi^m_i,...,\phi^m_n\}$$

and so

$$\varepsilon^1 \equiv \phi^n_0(\phi^m_1(\varepsilon_1,...,\varepsilon_m),...,\phi^m_n(\varepsilon_1,...,\varepsilon_m))$$

which by inductive reduction reduces to

$$\varepsilon^1 \xrightarrow{*} \phi^n_0(\phi^m_1(\varepsilon_1,...,\varepsilon_m),...,\psi^m_i(\varepsilon_1,...,\varepsilon_m),...,\phi^m_n(\varepsilon_1,...,\varepsilon_m))$$

On the other hand

$$\varepsilon^2 \equiv \phi^m_0(\varepsilon_1,...,\varepsilon_m) \equiv \phi^n_0\{\phi^m_1,...,\psi^m_i,...,\phi^m_n\}(\varepsilon_1,...,\varepsilon_m)$$

which by function dumping reduces to

$$\varepsilon^2 \xrightarrow{*} \phi^n_0(\phi^m_1(\varepsilon_1,...,\varepsilon_m),...,\psi^m_i(\varepsilon_1,...,\varepsilon_m),...,\phi^m_n(\varepsilon_1,...,\varepsilon_m))$$

so they are locally confluent.

If ε matches function dumping and inductive reduction on an argument, then it must be of the form $\phi_0^n\{\phi_1^m,...,\phi_n^m\}(\varepsilon_1,...,\varepsilon_m)$, where there is a $\varepsilon_j \xrightarrow{*} \varepsilon_j^2$, and so we must conflate the result of applying function dumping

$$\varepsilon^1 \equiv \phi_0^n(\phi_1^m(\varepsilon_1,...,\varepsilon_m),...,\phi_n^m(\varepsilon_1,...,\varepsilon_m))$$

with the result of inductively reducing ε_j:

$$\varepsilon^2 \equiv \phi_0^n\{\phi_1^m,...,\phi_n^m\}(\varepsilon_1,...,\varepsilon_j^2,...,\varepsilon_m)$$

But we may apply inductive reduction over $\varepsilon_j \xrightarrow{*} \varepsilon_j^2$ on each argument of ε^1 to obtain

$$\varepsilon^1 \xrightarrow{*} \phi_0^n(\phi_0^m(\varepsilon_1,...,\varepsilon_j^2,...,\varepsilon_m),...,\phi_n^m(\varepsilon_1,...,\varepsilon_j^2,...,\varepsilon_m))$$

and apply function dumping on ε^2 to obtain

$$\varepsilon^2 \xrightarrow{*} \phi_0^n(\phi_1^m(\varepsilon_1,...,\varepsilon_j^2,...,\varepsilon_m),...,\phi_n^m(\varepsilon_1,...,\varepsilon_j^2,...,\varepsilon_m))$$

so they are locally confluent.

If ε matches function dumping and inductive reduction on the sub-expression, then it must be of the form $\phi_0^n\{\phi_1^m,...,\phi_n^m\}(\varepsilon_1,...,\varepsilon_m)$ and also of the form $\sigma_k\delta$, but this is impossible.

If ε matches unifier variable dumping and inductive reduction on the head or an argument, then it must be of the form $\sigma_k\delta$ and also $\phi_0^m(\varepsilon_1,...,\varepsilon_m)$, but this is impossible.

If ε matches unifier variable dumping and inductive reduction on the sub-expression, then it must be of the form $\sigma_k\phi_0^m(\varepsilon_1,...,\varepsilon_m)$, where $\phi_0^m(\varepsilon_1,...,\varepsilon_m) \xrightarrow{*} \delta$, and so we must conflate the result of applying unifier variable dumping

$$\varepsilon^1 \equiv \phi_0^m(\sigma_k\varepsilon_1,...,\sigma_k\varepsilon_m)$$

with the result of applying inductive reduction on the sub-expression:

$$\varepsilon^2 \equiv \sigma_k\delta$$

For local confluence, we can consider only direct reduction steps on the sub-expression itself and use recursion, so we consider the cases for that:

- Projection simplification.

$$\varepsilon \equiv \sigma_k\pi_i^m(\varepsilon_1,...,\varepsilon_m)$$

and so

$$\varepsilon^1 \equiv \pi_i^m(\sigma_k\varepsilon_1, ..., \sigma_k\varepsilon_m)$$

and

$$\varepsilon^2 \equiv \sigma_k\varepsilon_i$$

But we can apply projection simplification on ε^1 to obtain

$$\varepsilon^1 \to \sigma_k\varepsilon_i$$

so they are locally confluent.

- Function dumping.

$$\varepsilon \equiv \sigma_k\phi_{0,0}^p\{\phi_1^m, ..., \phi_p^m\}(\varepsilon_1, ..., \varepsilon_m)$$

and so

$$\varepsilon^1 \equiv \phi_{0,0}^p\{\phi_1^m, ..., \phi_p^m\}(\sigma_k\varepsilon_1, ..., \sigma_k\varepsilon_m)$$

and

$$\varepsilon^2 \equiv \sigma_k\phi_{0,0}^p(\phi_1^m(\varepsilon_1, ..., \varepsilon_m), ..., \phi_p^m(\varepsilon_1, ..., \varepsilon_m))$$

But we can apply function dumping on ε^1 to obtain

$$\varepsilon^1 \xrightarrow{*} \phi_{0,0}^p(\phi_1^m(\sigma_k\varepsilon_1, ..., \sigma_k\varepsilon_m), ..., \phi_p^m(\sigma_k\varepsilon_1, ..., \sigma_k\varepsilon_m))$$

and we can apply unifier variable dumping and then inductive unifier variable dumping on the arguments on ε^2 to obtain

$$\varepsilon^2 \xrightarrow{*} \phi_{0,0}^p(\phi_1^m(\sigma_k\varepsilon_1, ..., \sigma_k\varepsilon_m), ..., \phi_p^m(\sigma_k\varepsilon_1, ..., \sigma_k\varepsilon_m))$$

so they are locally confluent.

- Unifier variable dumping. This is not possible because ε would have to be of the form $\sigma_k \phi_0^m(\varepsilon_1, ..., \varepsilon_m)$ and also $\sigma_k \sigma_l \delta$, which is impossible.
- Inductive reduction on the head. Then there is a ψ_0^m such that $\phi_0^m \xrightarrow{*} \psi_0^m$ and

$$\varepsilon^1 \equiv \phi_0^m(\sigma_k \varepsilon_1, ..., \sigma_k \varepsilon_m)$$

and

$$\varepsilon^2 \equiv \sigma_k \psi_0^m(\varepsilon_1, ..., \varepsilon_m)$$

But we may apply inductive reduction on the head of ε^1 to obtain

$$\varepsilon^1 \xrightarrow{*} \psi_0^m(\sigma_k \varepsilon_1, ..., \sigma_k \varepsilon_m)$$

and also apply unifier variable dumping on ε^2 to obtain

$$\varepsilon^2 \xrightarrow{*} \psi_0^m(\sigma_k \varepsilon_1, ..., \sigma_k \varepsilon_m)$$

so they are locally confluent.

- Inductive reduction on the arguments. Then there is a ε_i such that $\varepsilon_i \xrightarrow{*} \varepsilon_i^2$ and

$$\varepsilon^1 \equiv \phi_0^m(\sigma_k \varepsilon_1, ..., \sigma_k \varepsilon_m)$$

and

$$\varepsilon^2 \equiv \sigma_k \phi_0^m(\varepsilon_1, ..., \varepsilon_i^2, ..., \varepsilon_m)$$

But we may apply inductive reduction on $\sigma_k \varepsilon_i$ to reduce it to $\sigma_k \varepsilon_i^2$, and apply this reduction inductively on ε^1 to obtain

$$\varepsilon^1 \xrightarrow{*} \phi_0^m(\sigma_k \varepsilon_1, ..., \sigma_k \varepsilon_i^2, ..., \sigma_k \varepsilon_m)$$

and also apply unifier variable dumping on ε^2 to obtain

$$\varepsilon^2 \xrightarrow{*} \phi_0^m(\sigma_k \varepsilon_1, ..., \sigma_k \varepsilon_i^2, ..., \sigma_k \varepsilon_m)$$

so they are locally confluent.

- Inductive reduction on the sub-expression. This is not possible because ε would have to be of the form $\sigma_k \phi_0^m(\varepsilon_1, ..., \varepsilon_m)$ and also $\sigma_k \sigma_l \delta$, which is impossible.

If ε matches inductive reduction on the head and on an argument, then it must be of the form $\phi_0^m(\varepsilon_1, ..., \varepsilon_m)$ and there is a ψ_0^m such that $\phi_0^m \xrightarrow{*} \psi_0^m$ and a ε_j^2 such that $\varepsilon_j \xrightarrow{*} \varepsilon_j^2$, and so we must conflate

$$\varepsilon^1 \equiv \psi_0^m(\varepsilon_1, ..., \varepsilon_m)$$

with

$$\varepsilon^2 \equiv \phi_0^m(\varepsilon_1, ..., \varepsilon_j^2, ..., \varepsilon_m)$$

But we may apply inductive reduction on the argument ε_j to reduce

$$\varepsilon^1 \xrightarrow{*} \psi_0^m(\varepsilon_1, ..., \varepsilon_j^2, ..., \varepsilon_m)$$

and similarly apply inductive reduction on the head to reduce

$$\varepsilon^2 \xrightarrow{*} \psi_0^m(\varepsilon_1, ..., \varepsilon_j^2, ..., \varepsilon_m)$$

so they are locally confluent.

Finally, if ε matches inductive reduction on the head or on an argument and also on the sub-expression, then it must be of the form $\phi_0^m(\varepsilon_1, ..., \varepsilon_m)$ and also of the form $\sigma_k \delta$, but this is impossible.

Thus, the rewrite system is locally confluent and terminating, and therefore, by the diamond lemma, it is confluent. Since we have proven that unifier expression normal forms correspond exactly to the irreducible unifier expressions in this rewrite system, we have shown that the rewrite system reduces every unifier expression to a unique equivalent normal unifier expression.

□

Appendix C

Qualitative evaluation of fault pattern coverage

In this chapter, we produce all the necessary information to *reproduce* the qualitative evaluation of the completeness and specificity of the pattern catalogue against the original research that motivated the patterns.

This consists in what we consider to be a complete list of all the **examples** of faults as described in the original research, each of which with an explanation for why one of the patterns in our catalogue would adequately identify it.

Specifically, each example consists of the following information:

- **Source** - Reference and specific page in which it is described.
- **Description** - Brief, just enough to follow the example.
- **Applicable patterns** - References to patterns in chapter A that would detect it.
- **Explanation** - An explanation of why the pattern would successfully detect the fault.
- **Result** - One of three: **success**, **failure** or **partial**, indicating the degree by which we qualitatively see the patterns to have succeeded in detecting the fault.

C.1 Examples

C.1.1 MeatyVegetable

- **Source** - [Rector et al., 2004], page 3.
- **Description** - Meat and Vegetable have a common subclass MeatyVegetable, which should have no instances.
- **Applicable patterns** - §A.2.1, §A.2.3
- **Explanation** - As described in [Rector et al., 2004], this fault is internally fully consistent until a disjointness axiom is addded between meat and vegetable. However, assuming MeatyVegetable was a primitive class, pattern §A.2.1 would detect it, since MeatyVegetable would be a primitive class that is subsumed by two different primitive classes, Meat and Vegetable, which do not have any subsumption relation between them.

  Moreover, once the disjointness axiom is added, pattern §A.2.3 would also detect it. Meat and Vegetable being disjoint means there is no element that is an instance of both classes. But an instance of MeatyVegetable would be an instance of both. Thus, MeatyVegetable would be a primitive class that can have no instances, and this is exactly what pattern §A.2.3 detects.
- **Result** - Success.

C.1.2 Margherita pizzas with unwanted toppings

- **Source** - [Rector et al., 2004], page 4.
- **Description** - A margherita pizza defined only as having some tomato and some mozzarella could end up having all sorts of extra ingredients that would not be appropriate for a margherita pizza.
- **Applicable patterns** - §A.2.2.
- **Explanation** - We explicitly consider this situation on §A.2.2. However, we were unable to find a pattern (the authors of [Rector et al., 2004] also failed to do so) that would detect this.
- **Result** - Failure.

C.1.3 Pizzas with cheese that are not cheesy

- **Source** - [Rector et al., 2004], page 5.
- **Description** - If the definition of cheesy pizza is not made *complete* (*defined class*), then there would exist a lot of pizzas with cheese that would not be cheesy pizzas by the definition.
- **Applicable patterns** - §A.2.1.
- **Explanation** - The argument, extracted from [Rector et al., 2004], and distilled into a formal pattern, that primitive classes should not form subsumption cycles, could potentially detect this error, since CheesyPizza would be incorrectly be indicated as primitive. However, this would depend on the contextual ontology.
- **Result** - Partial.

C.1.4 Non-vegetarian margherita pizzas

- **Source** - [Rector et al., 2004], page 6.
- **Description** - This is really just an extension of the problem in §C.1.2 which becomes clear when defining a VegetarianPizza class.
- **Applicable patterns** - §A.2.2.
- **Explanation** - As described in §C.1.2, we currently have no patternt that would detect this situation.
- **Result** - Failure.

C.1.5 A chocolate ice-cream that is a pizza

- **Source** - [Rector et al., 2004], page 8.
- **Description** - The property hasTopping is used for both ice-creams and pizzas, but a domain constraint is indicated on hasTopping that says only Pizzas can have toppings. This makes the chocolate ice-cream class either subsumed by pizza or an unsatisfiable class, depending on other factors.
- **Applicable patterns** - §A.2.4, §A.2.1, §A.2.3.

- **Explanation** - In §A.2.4 we examine this particular situation and show how the pattern already described in §A.2.1, stating that primitive classes should not form subsumption cycles, already would detect this situation as a potential fault. Alternatively, the pattern in §A.2.3 would detect this if IceCream and Pizza were indicated to be disjoint.

- **Result** - Success.

C.1.6 Empty pizzas

- **Source** - [Rector et al., 2004], page 10.

- **Description** - An empty pizza fulfills the definition of a vegetarian pizza, because we only used universal restrictions of the definition of vegetarian pizza.

- **Applicable patterns** - §A.2.5, §A.2.13.

- **Explanation** - While §A.2.5 is a slightly different example, it follows exactly the same pattern as this example: universal restrictions that are satisfied only trivially. This is exactly what the pattern described in §A.2.5 detects.

  The pattern in §A.2.13 is more generic and would also detect this fault.

- **Result** - Success.

C.1.7 Vegetarian protein lovers pizza

- **Source** - [Rector et al., 2004], page 10.

- **Description** - An incorrect usage of an "and" operator instead of an "or" operator on disjoint classes makes a universal property restriction trivially satisfiable, producing unexpected subsumptions, like a ProteinLoversPizza being a vegetarian pizza, besides the intention of defining it as containing only fish and meat.

- **Applicable patterns** - §A.2.6, §A.2.5, §A.2.13, §A.2.14, §A.2.15.

- **Explanation** - Our pattern catalogue explicitly considers this example, and offers two completely different ways to detect it. First, the one discussed above about universal property restrictions that are satisfied only trivially (§A.2.5, and the more general §A.2.13). But moreover, in this case we can detect another fault related to the presence of what we describe as an "unsatisfiable property". This is

described in more detail in §A.2.6, and is also described in a slightly different way in §A.2.15.

This is related to the pattern in §A.2.14, in that it describes very similar situations, although the latter pattern would not detect this particular example.

- **Result** - Success.

C.1.8 Protein lovers pizzas do not exist

- **Source** - [Rector et al., 2004], page 12.
- **Description** - The incorrect definition of protein lovers pizza by using "and" instead of "or", combined with one sensible definition of a pizza, makes ProteinLoversPizza unsatisfiable.
- **Applicable patterns** - §A.2.3.
- **Explanation** - The issue is that an unsatisfiable class is produced. This is dealt with specifically via the pattern in §A.2.3.
- **Result** - Success.

C.1.9 Untangling of spicy toppings

- **Source** - [Rector et al., 2004], page 13.
- **Description** - SpicyBeefTopping is both a spicy topping and a meat topping. This can often be described incorrectly, specially when defined and primitive classes are used inadequately. Making SpicyTopping a primitive class makes certain toppings not be subsumed by it when they should.
- **Applicable patterns** - §A.2.1.
- **Explanation** - This situation can be detected by both patterns described in §A.2.1, detecting subsumption cycles on primitive classes or defined classes that subsume primitive classes.
- **Result** - Success.

C.1.10 Heterogeneous technical administrative group

- **Source** - [Prince Sales and Guizzardi, 2017], page 10.
- **Description** - A technical administrative group is characterized as a *collective* (see [Prince Sales and Guizzardi, 2017] for a precise definition), meaning its members should be indistinguishable from the point of view of the administrative group. However, it contains two families of employees that are clearly distinguishable within it.
- **Applicable patterns** - §A.2.7.
- **Explanation** - The multiple pattern alternatives specified in §A.2.7 are all designed specifically around this particular example. One does so by using the specific structural elements of the UFO ontology, and the other by using a more generic and flexible approach based on finding **any** property that allows us to distinguish elements of the collective. Both of those patterns apply directly to this specific example.
- **Result** - Success.

C.1.11 Homogeneous IT architecture

- **Source** - [Prince Sales and Guizzardi, 2017], page 11.
- **Description** - An IT architecture is characterized as a *functional complex* (see [Prince Sales and Guizzardi, 2017] for a precise definition), meaning its members should have different roles. However, it is formed exclusively by IT components which are homogeneous from the point of view of the IT architecture.
- **Applicable patterns** - §A.2.8.
- **Explanation** - The multiple pattern alternatives specified in §A.2.8 are all designed specifically around this particular example. One does so by using the specific structural elements of the UFO ontology, and the other by using a more generic and flexible approach based on finding whether there is **any** property at all that would allow us to distinguish ellements of the functional complex. Both of those patterns apply directly to this specific example, though in the second case other contextual aspects of the ontology could potentially prevent it from being detected.

- **Result** - Success.

C.1.12 Theatre in a theatre

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - A class Theatre is used to represent multiple elements, such as the artistic discipline and the family of buildings in which the plays are performed at the same time.
- **Applicable patterns** - None.
- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. We note that the original authors do not propose any systematic way to detect this either.
- **Result** - Failure.

C.1.13 Cars, motorcars and automobiles

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - Multiple classes are created for synonyms that represent exactly the same element (and then they are made equivalent). For example, cars, motorcars and automobiles.
- **Applicable patterns** - §A.2.9.
- **Explanation** - The pattern in §A.2.9 is simple, straightforward and designed to deal specifically with this kind of situation. It is very easy to automatically detect explicitly equivalent primitive classes. Moreover, we could change our pattern to detect any equivalent classes at all.
- **Result** - Success.

C.1.14 An actor “does is” a man

- **Source** - [Poveda-Villalón et al., 2010], page 6.

- **Description** - An "is" property / relation is created instead of using class subsumption.
- **Applicable patterns** - None.
- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Perhaps in large ontologies using such a property, some pattern of the semantics of such a property would emerge, but we are not aware of it.
- **Result** - Failure.

C.1.15 Members of non-existent teams

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - The property memberOfTeam is created in complete disconnection from the rest of the ontology. There is not even a Team class to relate it to.
- **Applicable patterns** - None.
- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. While a pattern checking whether there exists any other element of the ontology that has any relation of some kind with this element could be conceived, it would be extremely complicated, probably incomplete and create lots of performance problems. This seems to be a perfect example of a situation where more rudimentary methods might be more effective than the semantic approach of this thesis.
- **Result** - Failure.

C.1.16 The item sells the buyer

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - The ontology defines the properties sells and buys as inverses, when they are not. The inverse of sells is isSold, and the inverse of buys is isBought.

- **Applicable patterns** - None.
- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Again, the fault is a fault because of the differences between the actual model and the preferred model of the person representing it, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.
- **Result** - Failure.

C.1.17 All persons are professors

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - A subsumption cycle is created between primitive classes, in which all professors are persons, and all persons are professors.
- **Applicable patterns** - §A.2.10, §A.2.1.
- **Explanation** - Pattern §A.2.10 is designed specifically to detect this kind of situation, and is quite simple. While pattern §A.2.1 would not directly apply to the most basic version of this fault, in practice it would trigger in a large proportion of cases in which the fault was present, since the fault combined with usual definitions would produce cycles in the sense of pattern §A.2.1.
- **Result** - Success.

C.1.18 Style and period

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - A class StyleAndPeriod is created to represent multiple concepts at the same time (an artistic style in a specific period of history).
- **Applicable patterns** - None.
- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Again, the fault is a fault because of the differences between

the actual model and the preferred model of the person representing it, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.

- **Result** - Failure.

C.1.19 Product or service

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - A class ProductOrService is created to represent multiple concepts at the same time (a product or a service, possibly to represent things that can be sold), rather than using a more adequate name.
- **Applicable patterns** - None.
- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Again, the fault is a fault because of the differences between the actual model and the preferred model of the person representing it, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.

  Moreover, we note that in this case, the fault is not really one of semantic definition, but rather of naming. The concept of something that can be sold by a company is a single concept, but there is no adequate single word that represents this concept. If such a word were found, changing the name of the class would get rid of the entire problem.
- **Result** - Partial.

C.1.20 Routes that start but do not end

- **Source** - [Poveda-Villalón et al., 2010], page 6.
- **Description** - A property startsIn is created to denote where routes start, but no property endsIn is created to denote where routes end.
- **Applicable patterns** - None.

- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Again, the fault is a fault because of the differences between the actual model and the preferred model of the person representing it, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.

- **Result** - Failure.

C.1.21 Followed but not preceded

- **Source** - [Poveda-Villalón et al., 2010], page 6.

- **Description** - A property follows is created to denote a sequence of elements, but no property precedes is created to traverse them in the opposite order.

- **Applicable patterns** - §A.2.12.

- **Explanation** - While we did not design any pattern to deal with this fault specifically, or in the context in which it was introduced in [Poveda-Villalón et al., 2010], either of the versions of the pattern in §A.2.12 would in fact detect this situation, as precedes is the inverse property of follows.

- **Result** - Success.

C.1.22 Numbers that are both odd and even

- **Source** - [Poveda-Villalón et al., 2010], page 7.

- **Description** - Classes Odd and Even are not defined as disjoint, allowing the possibility of numbers that are both odd and even.

- **Applicable patterns** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Again, the fault is a fault because of the differences between the actual model and the preferred model of the person representing it, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.

- **Result** - Failure.

C.1.23 Numbers that are both prime and composite

- **Source** - [Poveda-Villalón et al., 2010], page 7.

- **Description** - Classes Prime and Composite are not defined as disjoint, allowing the possibility of numbers that are both prime and composite.

- **Applicable patterns** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Again, the fault is a fault because of the differences between the actual model and the preferred model of the person representing it, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.

- **Result** - Failure.

C.1.24 Objects writing emotions

- **Source** - [Poveda-Villalón et al., 2010], page 7.

- **Description** - A property hasWritten is created, but no domain or range constraints indicating that only Writers write, and they write LiteraryWork is not described.

- **Applicable patterns** - §A.2.11.

- **Explanation** - The pattern in §A.2.11 was designed specifically to detect this kind of situation. It also does so semantically, rather than checking explicit definitions of domain or range. Classes that are defined explicitly but which have no restrictions whatsoever on its domain and/or range are flagged as potentially faulty.

- **Result** - Success.

C.1.25 My city is not a CITY

- **Source** - [Poveda-Villalón et al., 2010], page 7.
- **Description** - Two classes which clearly represent the same conceptual element from different ontologies that are combined are not defined to be equivalent.
- **Applicable patterns** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. The fault has to do with naming conventions and the conceptualization of multiple authors, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.
- **Result** - Failure.

C.1.26 Referees being referees in matches

- **Source** - [Poveda-Villalón et al., 2010], page 7.
- **Description** - Properties hasReferee and isRefereeOf are defined in the ontology, but they are not indicated as being inverse properties.
- **Applicable patterns** - §A.2.12.
- **Explanation** - The patterns in §A.2.12 were designed explicitly to detect the situation in which properties do not have inverses or inverses are not indicated as such. These two situations are dealt with separately with two different patterns, one of which finds that there is no explicit inverse property, the other of which checks whether two properties have all the semantic conditions required to be potential inverses.
- **Result** - Success.

C.1.27 Inkless books

- **Source** - [Poveda-Villalón et al., 2010], page 7.
- **Description** - A Book is defined to be something produced by a Writer that **only** uses Paper, so if it uses Ink, then it cannot be a Book.

- **Applicable patterns** - §A.2.13, §A.2.5, §A.2.14.
- **Explanation** - The pattern in §A.2.13 is designed specifically to deal with this type of situation. It detects when a universal property restriction is present without an existential one. This is related to the pattern in §A.2.5, although the latter would not detect this partitcular example without additional context.

  The pattern in §A.2.14 is very closely related to this situation, but this pattern would not detect this particular example based on the way this example defines the situation.
- **Result** - Success.

C.1.28 Vegetarian pizzas with *some* vegetables

- **Source** - [Poveda-Villalón et al., 2010], page 7.
- **Description** - An existential restriction over a negated property is used instead of a negated existential restriction. That is, a VegetarianPizza is defined as a Pizza with *some* toppings that are not meat, rather than stating that it is a Pizza which does not have some toppings that are meat.
- **Applicable patterns** - §A.2.14, §A.2.6, §A.2.5.
- **Explanation** - The pattern described in §A.2.14 detects any class which is defined with a property restriction as general and unlikely to be of any use as an existential restriction over a negated property. This would find this example, although it would have some likelihood of triggering false positives.

  Once again, this example is very related to the patterns in §A.2.6 and §A.2.5, but the particular situations are slightly different, and these patterns would not detect the fault in this particular example.
- **Result** - Success.

C.1.29 Many Madrids and many Barcelonas

- **Source** - [Poveda-Villalón et al., 2010], page 8.
- **Description** - Individuals, such as Madrid and Barcelona, are created as classes subsumed by the class City, rather than as individuals of the class.

- **Applicable patterns** - §A.2.3.
- **Explanation** - In most cases, a definition such as this would make these classes unsatisfiable. This would be detected by the pattern in §A.2.3. However, the definition of individuals as classes itself does not necessarily make them unsatisfiable. Once again, we are in a case of preferred model issues that are undetectable in a purely semantic way.
- **Result** - Partial.

C.1.30 Only cities have an official language

- **Source** - [Poveda-Villalón et al., 2010], page 8.
- **Description** - The range of the property isOfficialLanguage is restricted to the class City, rather than considering that more general notions may also have an official language.
- **Applicable patterns** - None.
- **Explanation** - We did not address this pitfall from [Poveda-Villalón et al., 2010] precisely because we could not find a general notion that could be characterized generically about it. Again, the fault is a fault because of the differences between the actual model and the preferred model of the person representing it, but cannot be inferred purely semantically, at least in general. We note that the original authors do not propose methods to detect these faults either.
- **Result** - Failure.

C.1.31 Olympics happen in city-nations

- **Source** - [Poveda-Villalón et al., 2010], page 8.
- **Description** - A property takesPlaceIn is defined, whose domain is OlympicGames, and whose range is the **intersection** of City and Nation, meaning that only cities that are also nations can have olympic games take place in them. The **union** should have been used instead.
- **Applicable patterns** - §A.2.15, §A.2.6.

- **Explanation** - The pattern in §A.2.15 is specifically designed to detect this situation. This is very closely related to the pattern to detect unsatisfiable relative properties described in §A.2.6, but the latter is more specific and would not apply in this case.

- **Result** - Success.

C.1.32 Incorrectly labelled crossroads

- **Source** - [Poveda-Villalón et al., 2010], page 8.

- **Description** - The label and comment properties of a class are used in the opposite way that they should.

- **Applicable patterns** - None.

- **Explanation** - This is almost exactly the opposite of a semantic fault, one exclusively related with meta-properties of the ontology, which our patterns currently do not analyze at all. It is not completely impossible that some patterns in our framework would be able to deal with this, but currently they certainly do not.

- **Result** - Failure.

C.1.33 Other river element

- **Source** - [Poveda-Villalón et al., 2010], page 8.

- **Description** - A miscellaneous sub-class (presumedly primitive) is created under another class whose only purpose is to contain instances of the super-class that do not belong to other sub-classes. OtherRiverElement is simply any HydrographicalResource that is not classified as some other particular sub-class of HydrographicalResource.

- **Applicable patterns** - None.

- **Explanation** - Once again, this does not seem to be a semantically detectable fault, since it requires identifying OtherRiverElement as a different class from all the other sub-classes. No current pattern is implemented that would detect

this. The authors of the original research suggest at most lexical (i.e. based on the naming) ways to detect this sort of issue.

- **Result** - Failure.

C.1.34 animalorigin

- **Source** - [Poveda-Villalón et al., 2010], page 8.
- **Description** - No consistent naming convention is used, a class is named "animalorigin", as a sub-class of "Ingredient".
- **Applicable patterns** - None.
- **Explanation** - Purely lexical fault, cannot be detected through semantic methods like the approach we follow without further context.
- **Result** - Failure.

C.1.35 Yes and No as instances

- **Source** - [Poveda-Villalón et al., 2010], page 8.
- **Description** - Instead of using a boolean attribute isEcological for the class Car, a property is used with the instances Yes and No as range.
- **Applicable patterns** - None.
- **Explanation** - I am actually surprised I did not include a pattern for this in my pattern catalogue. Detecting a class that only has two instances and suggesting that should possibly be a datatype attribute instead is something our approach could do, but I overlooked it when designing the catalogue. It will likely be included in future versions (if they are produced).
- **Result** - Partial.

C.1.36 hasFork if and only if it hasFork

- **Source** - [Poveda-Villalón et al., 2010], page 9.
- **Description** - A recursive definition in which the range of the hasFork property is defined as exactly those elements that have an incoming hasFork relationship.

- **Applicable patterns** - §A.2.11.
- **Explanation** - Our extension of the original suggested pattern by the authors of [Poveda-Villalón et al., 2010, Poveda-Villalón et al., 2012] that detects not only failing to provide **explicit** domain or range constraints, but also in general failing to provide them, would actually detect this fault as well. The range definition in this example results in a tautology that means the range is everything. The purely semantic pattern in §A.2.11 would detect this.
- **Result** - Success.

Bibliography

[Alchourrón et al., 1985] Alchourrón, C. E., Gärdenfors, P., and Makinson, D. (1985). On the logic of theory change: Partial meet contraction and revision functions. *The journal of symbolic logic*, 50(2):510–530.

[Andrews, 2010] Andrews, P. B. (2010). *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer Publishing Company, Incorporated, 2nd edition.

[Angele et al., 2009] Angele, J., Kifer, M., and Lausen, G. (2009). Ontologies in F-Logic (Handbook on ontologies (Second edition)).

[Armando et al., 1999] Armando, A., Castellini, C., and Giunchiglia, E. (1999). Sat-based procedures for temporal reasoning. In *European Conference on Planning*, pages 97–108. Springer.

[Armando et al., 2004] Armando, A., Castellini, C., Giunchiglia, E., and Maratea, M. (2004). A sat-based decision procedure for the boolean combination of difference constraints. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 16–29. Springer.

[Baader and Ghilardi, 2011] Baader, F. and Ghilardi, S. (2011). Unification in modal and description logics. *Logic Journal of IGPL*, 19(6):705–730.

[Baader et al., 2009] Baader, F., Horrocks, I., and Sattler, U. (2009). Description Logics (Handbook on ontologies (Second edition)).

[Bafandeh Mayvan et al., 2017] Bafandeh Mayvan, B., Rasoolzadegan, A., and Ghavidel Yazdi, Z. (2017). The state of the art on design patterns: A systematic mapping of the literature. *Journal of Systems and Software*, 125:93–118.

[Baget et al., 2018] Baget, J.-F., Garcia, L., Garreau, F., Lefèvre, C., Rocher, S., and Stéphan, I. (2018). Bringing existential variables in answer set programming and

bringing non-monotony in existential rules: two sides of the same coin. *Annals of mathematics and artificial intelligence*, 82(1):3–41.

[Balaban et al., 2015] Balaban, M., Maraee, A., Sturm, A., and Jelnov, P. (2015). A pattern-based approach for improving model quality. *Software & Systems Modeling*, 14(4):1527–1555.

[Balduccini, 2009] Balduccini, M. (2009). Representing constraint satisfaction problems in answer set programming. In *Proceedings of ICLP*, volume 9, pages 61–70. Citeseer.

[Barendregt, 1992] Barendregt, H. P. (1992). Lambda calculi with types.

[Barendregt et al., 1987] Barendregt, H. P., van Eekelen, M. C., Glauert, J. R., Kennaway, J. R., Plasmeijer, M. J., and Sleep, M. R. (1987). Term graph rewriting. In *International conference on parallel architectures and languages Europe*, pages 141–158. Springer.

[Barrett et al., 2002] Barrett, C. W., Dill, D. L., and Stump, A. (2002). Checking satisfiability of first-order formulas by incremental translation to sat. In *International Conference on Computer Aided Verification*, pages 236–249. Springer.

[Beck, 1987] Beck, K. (1987). Using pattern languages for object-oriented programs. *http://c2. com/doc/oopsla87. html*.

[Beth, 1955] Beth, E. W. (1955). Semantic entailment and formal derivability.

[Blomqvist, 2010] Blomqvist, E. (2010). Ontology patterns: Typology and experiences from design pattern development. In *The Swedish AI Society Workshop May 20-21; 2010; Uppsala University*, number 48, pages 55–64. Linköping University Electronic Press; Linköpings universitet.

[Bongio et al., 2008] Bongio, J., Katrak, C., Lin, H., Lynch, C., and McGregor, R. E. (2008). Encoding first order proofs in smt. *Electronic Notes in Theoretical Computer Science*, 198(2):71–84.

[Bozzano et al., 2005] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Rossum, P. v., Schulz, S., and Sebastiani, R. (2005). An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 317–333. Springer.

[Brachman and Schmolze, 1989] Brachman, R. J. and Schmolze, J. G. (1989). An overview of the kl-one knowledge representation system. In Mylopolous, J. and Brodie, M., editors, *Readings in Artificial Intelligence and Databases*, pages 207 – 230. Morgan Kaufmann, San Francisco (CA).

[Brewka et al., 2011] Brewka, G., Eiter, T., and Truszczynski, M. (2011). ASP at a glance. *Communications of the ACM*.

[Brickely and Guha, 2014] Brickely, D. and Guha, R. (2014). Rdf schema 1.1. [Online (https://www.w3.org/TR/rdf-schema/); accessed 19-December-2016].

[Brummayer and Biere, 2009] Brummayer, R. and Biere, A. (2009). Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer.

[Bundy, 1983] Bundy, A. (1983). *The computer modelling of mathematical reasoning*, volume 10. Academic Press London.

[Bundy and Mitrovic, 2016] Bundy, A. and Mitrovic, B. (2016). Reformation: a domain-independent algorithm for theory repair. [Online (http://www.research.ed.ac.uk/portal/en/publications/reformation-a-domainindependent-algorithm-for-theory-repair(cac700ba-6e37-4609-9b4f-22b166e831cf).html); accessed 10-September-2017].

[Bundy et al., 1993] Bundy, A., Stevens, A., Van Harmelen, F., Ireland, A., and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial intelligence*, 62(2):185–253.

[Calvès, 2013] Calvès, C. (2013). Unifying nominal unification. In *RTA 2013-24th International Conference on Rewriting Techniques and Applications*, volume 21, pages 143–157. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Casanova, 2017] Casanova, J. (2017). Meta-ontology fault detection (master of science by research). [Online (https://tinyurl.com/yb72r3ch); accessed 3-October-2017].

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

[Church, 1940] Church, A. (1940). A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68.

[Comon, 1990] Comon, H. (1990). Equational formulas in order-sorted algebras. In *International Colloquium on Automata, Languages, and Programming*, pages 674–688. Springer.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158.

[Copeland et al., 2013] Copeland, M., Gonçalves, R. S., Parsia, B., Sattler, U., and Stevens, R. (2013). *Finding Fault: Detecting Issues in a Versioned Ontology*, pages 113–124. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Cox and Pietrzykowski, 1986] Cox, P. T. and Pietrzykowski, T. (1986). Causes for events: Their computation and applications. In Siekmann, J. H., editor, *8th International Conference on Automated Deduction*, pages 608–621, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

[Dionne et al., 1992] Dionne, R., Mays, E., and Oles, F. J. (1992). A non-well-founded approach to terminological cycles. In *AAAI*, volume 92, pages 761–766.

[Dionne et al., 1993] Dionne, R., Mays, E., and Oles, F. J. (1993). The equivalence of model-theoretic and structural subsumption in description logics. In *IJCAI*, pages 710–717.

[Dowek, 2001] Dowek, G. (2001). Higher-order unification and matching. *Handbook of automated reasoning*, 2:1009.

[Dwork et al., 1984] Dwork, C., Kanellakis, P. C., and Mitchell, J. C. (1984). On the sequential nature of unification. *The Journal of Logic Programming*, 1(1):35–50.

[Emerson, 1991] Emerson, E. A. (1991). Temporal and modal logic, handbook of theoretical computer science (vol. b): formal models and semantics.

[Ennals and Jones, 2003] Ennals, R. and Jones, S. P. (2003). Hsdebug: debugging lazy programs by not being lazy. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 84–87.

[Farmer, 1988] Farmer, W. M. (1988). A unification algorithm for second-order monadic terms. *Annals of Pure and applied Logic*, 39(2):131–174.

[Farmer, 1991] Farmer, W. M. (1991). Simple second-order languages for which unification is undecidable. *Theoretical Computer Science*, 87(1):25–41.

[Ferreirós, 2001] Ferreirós, J. (2001). The road to modern logic—an interpretation. *Bulletin of Symbolic Logic*, 7(4):441–484.

[Fontana et al., 2012] Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1.

[Fowler, 1999] Fowler, M. (1999). *Bad smells in code*, pages 63–73. Addison-Wesley Professional.

[Gärdenfors, 1992] Gärdenfors, P. (1992). *Belief Revision*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

[Gärdenfors, 2003] Gärdenfors, P. (2003). *Belief revision*, volume 29. Cambridge University Press.

[Girard, 1987] Girard, J.-Y. (1987). Linear logic. *Theoretical computer science*, 50(1):1–101.

[Gkaniatsou et al., 2012] Gkaniatsou, A., Bundy, A., and Mcneill, F. (2012). Towards the automatic detection and correction of errors in automatically constructed ontologies. In *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems*, pages 860–867.

[Gomes et al., 2008] Gomes, C. P., Kautz, H., Sabharwal, A., and Selman, B. (2008). Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134.

[Guarino et al., 2009] Guarino, N., Oberle, D., and Staab, S. (2009). What Is an Ontology? (Handbook on ontologies (Second edition)).

[Guarino and Welty, 2009] Guarino, N. and Welty, C. A. (2009). An Overview of OntoClean (Handbook on ontologies (Second edition)).

[Habel and Plump, 1995] Habel, A. and Plump, D. (1995). Unification, rewriting, and narrowing on term graphs. *Electronic Notes in Theoretical Computer Science*, 2:110–117.

[Hammar and Presutti, 2017] Hammar, K. and Presutti, V. (2017). Template-based content odp instantiation. In *Advances in Ontology Design and Patterns :*, number 32 in Studies on the Semantic Web.

[Haverty, 2013] Haverty, T. (2013). Automated error classification in the KnowItAll ontology. Unpublished (Master of Science dissertation).

[Herbrand, 1930] Herbrand, J. (1930). *Recherches sur la théorie de la démonstration*. PhD thesis, Université de Paris.

[Heyting, 1966] Heyting, A. (1966). *Intuitionism: an introduction*, volume 41. Elsevier.

[Horridge, 2011] Horridge, M. (2011). *Justification based explanation in ontologies*. PhD thesis, University of Manchester.

[Horridge et al., 2009] Horridge, M., Jupp, S., Moulton, G., Rector, A., Stevens, R., and Wroe, C. (2009). A practical guide to building owl ontologies using protégé 4 and co-ode tools edition1. 2. *The university of Manchester*, 107.

[Huet, 1975] Huet, G. (1975). A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27 – 57.

[Huet and Oppen, 1980] Huet, G. and Oppen, D. C. (1980). Equations and rewrite rules: A survey. In BOOK, R. V., editor, *Formal Language Theory*, pages 349–405. Academic Press.

[Jaffar and Lassez, 1987] Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119.

[Jaśkowski, 1934] Jaśkowski, S. (1934). On the rules of suppositions in formal logic.

[Kerkhoff et al., 2014] Kerkhoff, S., Pöschel, R., and Schneider, F. M. (2014). A short introduction to clones. *Electronic Notes in Theoretical Computer Science*, 303:107–120.

[Kindermann et al., 2019] Kindermann, C., Parsia, B., and Sattler, U. (2019). Detecting influences of ontology design patterns in biomedical ontologies. In *International Semantic Web Conference*, pages 311–328. Springer.

[Kowalski, 2014] Kowalski, R. (2014). *Logic for Problem Solving, Revisited*. BoD–Books on Demand.

[Kripke, 2007] Kripke, S. (2007). Semantical considerations of the modal logic.

[Kripke, 1959] Kripke, S. A. (1959). A completeness theorem in modal logic. *The journal of symbolic logic*, 24(1):1–14.

[Kurian et al., 2013] Kurian, M. et al. (2013). A survey on tools essential for semantic web research. *Int. J. Comput. Appl*, 62(9):26–29.

[Lake and Crowther, 2013] Lake, P. and Crowther, P. (2013). *Concise Guide to Databases: A Practical Introduction*. Undergraduate Topics in Computer Science. Springer London, London, 2013 edition.

[Lambrix and Liu, 2013] Lambrix, P. and Liu, Q. (2013). Debugging the missing is-a structure within taxonomies networked by partial reference alignments. *Data and Knowledge Engineering*, 86:179 – 205.

[Lassila and Swick, 1999] Lassila, O. and Swick, R. R. (1999). Resource description framework(rdf) model and syntax specification. [Online (https://www.w3.org/TR/PR-rdf-syntax/); accessed 19-December-2016].

[Levy, 1996] Levy, J. (1996). Linear second-order unification. In *International Conference on Rewriting Techniques and Applications*, pages 332–346. Springer.

[Levy, 1998] Levy, J. (1998). Decidable and undecidable second-order unification problems. In *International Conference on Rewriting Techniques and Applications*, pages 47–60. Springer.

[Levy and Veanes, 2000] Levy, J. and Veanes, M. (2000). On the undecidability of second-order unification. *Information and Computation*, 159(1-2):125–150.

[Levy and Villaret, 2010] Levy, J. and Villaret, M. (2010). An efficient nominal unification algorithm.

[Li et al., 2018] Li, X., Bundy, A., and Smaill, A. (2018). Abc repair system for datalog-like theories. In *10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, volume 2, pages 335–342. SCITEPRESS.

[Mall, 2018] Mall, R. (2018). *Fundamentals of software engineering*. PHI Learning Pvt. Ltd.

[Markakis, 2013] Markakis, Z. (2013). Repairing the KnowItAll Ontology. Unpublished (Master of Science dissertation).

[Mikroyannidi et al., 2011] Mikroyannidi, E., Iannone, L., Stevens, R., and Rector, A. (2011). Inspecting regularities in ontology design using clustering. In Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., and Blomqvist, E., editors, *The Semantic Web – ISWC 2011*, pages 438–453, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Mikroyannidi et al., 2012] Mikroyannidi, E., Stevens, R., Iannone, L., and Rector, A. (2012). Analysing syntactic regularities and irregularities in SNOMED-CT. *Journal of Biomedical Semantics*, 3(1):8.

[Miller, 2021] Miller, D. (2021). Lambda-prolog: Logic programming in higher-order logic. [Online (http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/); accessed 30-June-2021].

[Motik et al., 2012] Motik, B., Patel-Schneider, P. F., and Parsia, B. (2012). Owl 2 web ontology language. structural specification and functional-style syntax. [Online (https://www.w3.org/TR/owl2-syntax/); accessed 19-December-2016].

[Nadathur and Miller, 1988] Nadathur, G. and Miller, D. (1988). An overview of lambda-prolog.

[Newman, 1942] Newman, M. H. A. (1942). On theories with a combinatorial definition of" equivalence". *Annals of mathematics*, pages 223–243.

[Nieuwenhuis et al., 2006] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. (2006). Solving sat and sat modulo theories: from an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977.

[Paulson, 1989] Paulson, L. C. (1989). The foundation of a generic theorem prover. *CoRR*, cs.LO/9301105.

[Peirce, 1901] Peirce, C. S. (1901). On the logic of drawing history from ancient documents, especially from testimonies. *The Essential Peirce, 1893-1913*, 2:75–114.

[Peirce, 1906] Peirce, C. S. (1906). Prolegomena to an apology for pragmatism. *The Monist*, 16(4):492–546.

[Pinto et al., 2009] Pinto, H. S., Tempich, C., and Staab, S. (2009). Ontolgy Engineering and Evolution in a Distributed World Using DILIGENT (Handbook on ontologies (Second edition)).

[Plump, 1999] Plump, D. (1999). Term graph rewriting. *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools*, pages 3–61.

[Plump, 2002] Plump, D. (2002). Essentials of term graph rewriting. *Electronic Notes in Theoretical Computer Science*, 51:277–289.

[Plump, 2005] Plump, D. (2005). Confluence of graph transformation revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, pages 280–308. Springer.

[Poveda-Villalón et al., 2010] Poveda-Villalón, M., Suárez-Figueroa, M. C., and Gómez-Pérez, A. (2010). A double classification of common pitfalls in ontologies. *Workshop on Ontology Quality*.

[Poveda Villalón, 2016] Poveda Villalón, M. (2016). *Ontology Evaluation: a pitfall-based approach to ontology diagnosis*. PhD thesis, ETSI_Informatica.

[Poveda-Villalón et al., 2012] Poveda-Villalón, M., Suárez-Figueroa, M. C., and Gómez-Pérez, A. (2012). *Validating Ontologies with OOPS!*, pages 267–281. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Prince Sales and Guizzardi, 2017] Prince Sales, T. and Guizzardi, G. (2017). "is it a fleet or a collection of ships?": Ontological anti-patterns in the modeling of part-whole relations.

[Prior, 1962] Prior, A. N. (1962). Tense-logic and the continuity of time. *Studia Logica*, 13(1):133–148.

[Rector et al., 2004] Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). *OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns*, pages 63–81. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Robinson and Voronkov, 2001] Robinson, A. J. and Voronkov, A. (2001). *Handbook of automated reasoning*, volume 1. Gulf Professional Publishing.

[Robinson et al., 1965] Robinson, J. A. et al. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41.

[Schilling, 2011] Schilling, T. (2011). Challenges for a trace-based just-in-time compiler for haskell. In *International Symposium on Implementation and Application of Functional Languages*, pages 51–68. Springer.

[Schmidt-Schauß, 2004] Schmidt-Schauß, M. (2004). Decidability of bounded second order unification. *Information and Computation*, 188(2):143–178.

[Schmidt-Schauß et al., 2019] Schmidt-Schauß, M., Sabel, D., and Kutz, Y. D. (2019). Nominal unification with atom-variables. *Journal of Symbolic Computation*, 90:42–64.

[Sipser, 1996] Sipser, M. (1996). Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29.

[Smith, 1987] Smith, R. (1987). Panel on design methodology. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 91–95.

[Sørensen and Urzyczyn, 2006] Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard isomorphism*. Elsevier.

[Steen, 2020] Steen, A. (2020). Extensional paramodulation for higher-order logic and its effective implementation leo-iii. *KI-Künstliche Intelligenz*, 34(1):105–108.

[Steen and Benzmüller, 2018] Steen, A. and Benzmüller, C. (2018). The higher-order prover leo-iii. In Galmiche, D., Schulz, S., and Sebastiani, R., editors, *Automated Reasoning*, pages 108–116, Cham. Springer International Publishing.

[Sterling and Shapiro, 1994a] Sterling, L. and Shapiro, E. (1994a). *Meta-Interpreters*, pages 303–330. MIT Press, Cambridge, MA, USA.

[Sterling and Shapiro, 1994b] Sterling, L. and Shapiro, E. Y. (1994b). *The art of Prolog: advanced programming techniques*. MIT press.

[Stumme, 2009] Stumme, G. (2009). Formal Concept Analysis (Handbook on ontologies (Second edition)).

[Sure et al., 2009] Sure, Y., Staab, S., and Studer, R. (2009). Ontology Engineering Methodology (Handbook on ontologies (Second edition)).

[Šváb-Zamazal and Svátek, 2008] Šváb-Zamazal, O. and Svátek, V. (2008). Analysing ontological structures through name pattern tracking. In Gangemi, A. and Euzenat, J., editors, *Knowledge Engineering: Practice and Patterns*, pages 213–228, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Tarjan, 1975] Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225.

[Tsarkov and Horrocks, 2006] Tsarkov, D. and Horrocks, I. (2006). Fact++ description logic reasoner: System description. In *International joint conference on automated reasoning*, pages 292–297. Springer.

[Turing, 1937] Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265.

[Urban et al., 2004] Urban, C., Pitts, A. M., and Gabbay, M. J. (2004). Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497.

[Urbonas, 2019] Urbonas, M. (2019). A Heuristic Approach for Guiding Automated Theory Repair for the ABC Theory Repair System. Unpublished (Undergraduate fourth year dissertation).

[Urbonas et al., 2020] Urbonas, M., Bundy, A., Casanova, J., and Li, X. (2020). The use of max-sat for optimal choice of automated theory repairs. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 49–63. Springer.

[Van Dalen, 1986] Van Dalen, D. (1986). Intuitionistic logic. In *Handbook of philosophical logic*, pages 225–339. Springer.

[Wikipedia contributors, 2022a] Wikipedia contributors (2022a). Congruence relation — Wikipedia, the free encyclopedia. [Online; accessed 5-September-2022].

[Wikipedia contributors, 2022b] Wikipedia contributors (2022b). Horn clause — Wikipedia, the free encyclopedia. [Online; accessed 12-May-2022].

[Wikipedia contributors, 2022c] Wikipedia contributors (2022c). Np-completeness — Wikipedia, the free encyclopedia. [Online; accessed 23-May-2022].

[Zaionc, 1987] Zaionc, M. (1987). The regular expression descriptions of unifier set in the typed λ-calculus. *Fundamenta Informaticae*, 10(3):309–322.

Index of concepts and definitions

Index of notation

Here we try to capture some of the least standard technical notations that are used throughout the thesis whose meaning may not be entirely clear, and reference the sections in which they are defined, so that they can be more easily found. While we do include some standard notation here, this is not exhaustive, and in general for standard notations, consider looking at chapter 3 and the standard literature cited there.

Note that, while in general avoided, in some occasions the same symbol might mean different things in different contexts. Normally this only happens when the context is absolutely clear.

General mathematical symbols

$\equiv$ - *Equivalence* - Used to indicate that the two elements are syntactically equivalent at the most abstract level.

$:=$ - *Assignment* - Used to indicate that we define the left element to be the second element.

First-order logic standard syntax

$\neg$ - *Negation* - Definition 3.2.4.

$\wedge$ - *Conjunction* - Definition 3.2.4.

$\vee$ - *Disjunction* - Definition 3.2.4.

$\Longrightarrow$ - *Implication* - Definition 3.2.4.

$\forall$ - *Universal quantification* - Definition 3.2.5.

$\exists$ - *Existential quantification* - Definition 3.2.6.

$\models$ - *Entailment* - Definition 3.2.14.

$\vdash$ - *Provability* - Definition 3.2.16.

$\top$ - *True/Empty CNF* - §3.2.3.

$\bot$ - *False/Empty clause* - §3.2.3.

$X, Y...$ - Typically used to reference *First-order variables* - Definition 6.1.1.

$f, g...$ - Typically used to reference *Function symbols* - Definition 6.1.1.

π_i - *Projection* function on the i-th argument - Definition 6.1.2.

$p, q...$ - Typically used to reference *Predicate symbols* - Definition 6.1.1.

$\mathcal{N}(*)$ - *Normal form* - Theorems 6.1.1, 6.1.2 and 6.1.4.

Higher-order logic standard syntax

λ - *Lambda abstraction* - §3.2.2.

ϕ, ψ - Typically used to reference *Second-order terms* - Definition 6.1.2.

$F, G...$ - Typically used to reference *Second-order function variables* - Definition 6.1.1.

$P, Q...$ - Typoically used to reference *Second-order predicate variables* - Definition 6.1.1.

Unification

$\sim$ - *Unification* - §3.2.1.2.

$\rightarrow$ - *Variable substitution* - §3.2.1.2.

$\approx$ - *Unification equation* - Definition 6.1.26.

Rewrite systems

$\rightarrow$ - *Direct reduction* - §3.4.

$\xrightarrow{*}$ - *Reduction* - Definitions 6.1.5, 6.1.9.

$\xrightarrow{i}$ - *Non-transitive reduction* - Definition 6.1.5.

Term syntax

$\circ$ - *Unary function composition* - Definition 6.1.2.

$*\{*,*,...\}$ - *General function composition* - Definition 6.1.2.

$\cong$ - *Term equivalence* - Definitions 6.1.5, 6.1.9.

Existential second-order query logic

$\models$ - *Entailment query* - Definition 6.2.4.

$\models^*$ - *Satisfiability query* - Definition 6.2.5.

$\simeq$ - *Unification query* - Definition 6.2.6.

Dependency graph unification