Automated Theory Interpretation

by

Immanuel Normann

Submitted as PhD Thesis in Computer Science
19 November 2008

Jacobs University Bremen
School of Engineering and Science

Date of Defense: 8 December 2008

Dissertation Committee

Prof. Dr. Michael Kohlhase, Jacobs University Bremen (Supervisor)
Prof. Dr. Herbert Jaeger, Jacobs University Bremen
PD Dr. Till Mossakowski, German Research Center for Artificial Intelligence, Bremen

Abstract

Automated reasoning is a computer aided technology to support human reasoning based on formalized content. The main benefits expected from automated reasoning are reliability regarding its correctness and reasoning speed. Speed matters if reasoning is applied on large formalized knowledge bases and reliability matters if critical decisions depend on reasoning or if reliability of reasoning is per se desirable as in mathematics.

This thesis presents novel methods from automated reasoning to find reusable knowledge and knowledge overlaps in large formalized knowledge bases. The focus is on formalized mathematics, but the methods are applicable on any kind of formalized content as long as the application of classical logic inference rules on this content is admissible.

Formalized as well as traditional mathematics is advantageously organized in theories as its biggest knowledge units. As history of mathematics has shown, appearently unrelated theories happen to share a common core (i.e. knowledge overlap) once the concepts of each theories are translated appropriately. If all valid sentences of a theory A also hold in another theory B with an appropriate translation T, then A is said to be included in B – or A can be interpreted in B. Since all sentences of a theory are derivable from a few of it, namely from its axioms, we know that A is included in B as soon as we have shown that the axioms of A hold in B. In this case all derivable sentences in A automatically hold in B too, i.e. the knowledge from A can be reused in B.

In this thesis an algorithm for automated theory interpretation search is presented. It is based on semantic formula matching by means of normalization taken from term rewriting and a novel standardization technique for associative and commutative terms. Moreover, a practical notion of theory intersection is introduced and an algorithm for the construction of such intersections is presented.

Both algorithms are implemented in a prototype system and experiments are conducted on the largest library of formalized mathematics – the Mizar Mathematical Library – which demonstrates the scalability of the algorithms and also reveal thousands of theory interpretations.

Parts of this thesis are based on material previously published: [41, 42].

Acknowledgement

The first time I learned about automated theorem proving was about 10 years ago – when I finished my studies in physics. During this period I developed a particular interest in formalized mathematical content. I read many articles about this field – in particular about Michaels chief work: OMDoc – the markup language for Open Mathematical Documents. At the same time however, I thought it was too late in my career to make a new start in research. First, because I already had a job in industry and second, I was neither a mathematician nor a computer scientist. I am most grateful to Michael, first of all, because he gave me this second chance to follow my passion for research in formal methods. His ideas always inspired me and his patient support during critical phases of my research made it possible to turn my former vague research dreams into a concrete scientific work.

All the programming aspects of my research were supervised by Till, whom I thank for his always responsive support, not just in programming issues but also concerning theo- retical questions. Due to discussions with Herbert Jaeger I felt encouraged not to forget a philosophical view on mathematics.

As one of the earliest members of the working group "KWARC (Knowledge Adaptation and Reasoning for Content)" I was pleased to see it growing. I enjoyed particularly enthusiastic discussions with Normen Müller about various research ideas in endless Fritz sessions and Lakatos nights. But I also want to thank all other members of the KWARC group: Christoph Lange for his support in programming issues and the review of my thesis, Florian Rabe for theoretical insights related to category theory, Christine Müller for constructive critiques.

As member of John Bateman's working group, I want to thank John and Alexander Garcia who released me from extra work load during the final phase of my thesis, and Oliver Kutz for many late night discussions. Last but not least I am deeply indebted to my family: Maren, Kasimir and Lorenz. You endured so much, but – "change has come" – as someone recently said – and that's for sure in our case.

Table of contents

Chapter 1

Introduction

This thesis intends to contribute to the research area of formal methods. More specifically, it presents new formal techniques to employ to the advantage of three important application fields: 1) formal specification of software or hardware systems, 2) formalized mathematics, and 3) formal ontologies. Our techniques are not specific for one of these three application fields, but they rather rely on formal logics – the common ground for all of them.

Depending on the application area the first central notion is called *formal specification* for software and hardware systems, *theory* in formal mathematics, and *ontology* in the field of formal ontologies (and it might have another name in another discipline making use of formal logics). From the perspective of our techniques these notions are identical. We will treat them in this respect as synonyms (except for those chapters where we make some historical notes). Since mathematics is the first discipline involving formal logics, we will prefer the term *theory* over *formal specification* or *ontology*. Another reason for this preference is that most experimental results in this thesis are taken from formal mathematics.

Based on the notion of *theory*, our second central notion is a relation between two theories called *theory interpretation*. Essentially, a *source theory* can be *interpreted* in a *target theory* if there is an appropriate translation so that all translated statements of the source theory are valid in the target theory. These two notions, *theory* and *theory interpretation*, are extensively investigated in the history of mathematics with the rise of formal logics – chapter 2 gives a short account of that development. With the rise of the computer tools and machine readable formal languages tools have been developed to manage formalized theories and their interpretations. A corresponding overview of the state of the art of *theory management* is given in chapter 3. The fundamental purpose of these tools is *knowledge reuse* in at least two ways: 1) new theories are often extensions of already existing theories. Instead of writing a new theory from scratch we can import these existing theories – we call this *theory import*. Since theories are constituted by its axioms, we can consider theory imports as *reuse of axioms*. 2) sometimes new theories turn out to be partially old in the sense that the old theory can be interpreted in the new theory. But this means that all theorems of the old theory are also theorems in the new theory and we do not have to prove them again – that is *theory interpretation* facilitates the *reuse of theorems*. We refer to *theory management* as the sound operation on the *theory graph* (also called *development graph*) whose nodes are *theories* and whose edges are *theory imports* or *theory interpretations*. Elementary operations on a graph are adding and removing nodes and edges. This thesis focuses on adding edges. More precisely, it investigates means to **find automatically new theory interpretations** for

theory nodes in a given theory graph and thereby facilitating automated theorem reuse. Such a functionality is not supported by any existing theory management tool. The formal background of theory interpretation and an abstract description of the search algorithm is presented in chapter 4.

It should be noted that it is not always possible to link two theories with a theory interpretation, since a theory interpretation is an assertion: all statements of the translated source theory must hold in the target theory (theory imports in contrast always hold by definition). Automated theory interpretation search can be thus considered as a field of automated theorem proving (ATP) – i.e. proving automatically that assertions can be derived from given hypothesis. It is different in that 1) in ATP hypothesis and assertions are from the same theory and consequently 2) all involved formulae are from a common signature (i.e. they share common symbols). The challenge of automated theory interpretation search is to find an appropriate translation between the signatures of two theories such that the translated formulae of the source theory can be proved inside the target theory.

Undecidability, though, is their common inevitable problem: what can be proved in finite time is always only a fragment of what is provable in principle (unless we restrict ourselves to decidable logics – but this thesis does not investigate dedicated methods for theory management in decidable logics). The size of the provable fragment depends monotonically on the process time the theorem prover consumes. State of the art theorem provers are highly optimized for speed, but (usually) restricting their provable fragment in favour of faster results is not considered as option – time has not the first priority. At least in formalized system specification or even more in mathematics, it is not relevant whether a proof takes a few milliseconds or minutes. If the theorem is involved and important for a certain community, a process time of hours or even days is considered acceptable.

For the methods presented in this thesis, time has priority over the provable fragment – i.e. we do not try to prove automatically at any time costs, but limit our proof automation to proofs that are guaranteed to be fast. The envisioned application scenario of our methods is not to let a prover run minutes and potentially hours in order to find out whether one source theory can be interpreted inside one target theory (which may involve tricky proof tactics). Instead the methods are optimized to scan masses of theories for possible theory interpretations in a time frame of a few seconds. Consequently, we sacrifice for the benefit of responsiveness some theory interpretations that could be found by powerful theorem provers[1.1] (within unpredictable time). The optimization goal is thus inverse to automated theorem proving: instead of minimizing computation time for a fixed maximal provable fragment, we try to maximize the provable fragment for a computation time limited to a few seconds.

In summary, the algorithms for theory interpretation search in this thesis consists of three steps: two prepossessing steps, namely formula standardization and formula normalization, and the actual search process by syntactic matching. Given a theory graph, all its theories, more precisely, their formulae, are standardized and normalized before any search. This indexing process is conducted only once. Depending on the numbers of theories it might take hours, but not at cost of search time. For the actual search, only

1.1. It should be noted that in fact there are no automated theorem provers at all that could be used for theory interpretation search out of the box. As mentioned above, they are designed to prove theorems from axioms within the same theory. It would be another interesting research effort to adapt them for theory interpretation search.

the source theory for which we are seeking theory interpretations to one or several theories in the theory base need to go through the prepossessing step. For a single theory this step typically takes less then a second. After that, theory interpretations can be searched very efficiently with a purely syntactic search.

Chapter 6 introduces a very general formal language using λ-notation suitable for representing a wide range of formal languages used for first- or higher-order logics. It also specifies the kind of formula translations supported, namely induced by symbol renaming as signature mappings (symbol to term mappings are not considered). This restriction allows for specific optimizations for the syntactic matching – as presented in chapter 7. In particular the idea to use the formula skeleton, i.e. the formula with all its symbols replaced by a placeholder, as a very efficient (almost constant time) search filter is presented that reduces the search space significantly. Beyond this purely syntactic matching the concept of a skeleton modulo an equivalence relation is introduced, defined as the least element of the induced equivalence class for an arbitrary but fixed term ordering. The most basic equivalence class is defined by formula equivalence modulo renaming of the non-logical symbols. All other standardization and normalization steps make use of it.

An algorithm for formula standardization modulo associativity and commutativity (AC) is found and presented also in chapter 7. Thus it is possible to compute a skeleton for formula AC matching, i.e. a constant time search filter with a significant search space reduction for an inherently complex matching problem.

Whereas we define standardization on a term ordering, normalization is the result of convergent term rewriting. Chapter 8 starts with all the background of term rewriting theory needed for the remainder of this chapter. In essence, a convergent term rewriting system is a set of rewrite rules whose application always terminates and whose final result does not depend on the order the rules are applied from the initial to the final formula. Thus the normal form is also a unique representative of an equivalence class induced by a convergent rewrite rule system. The goal of applying rewrite rules is to find unique representatives for logically equivalent formulae. Many of the rewrite rules we make use of are already known, like negation normal form and prenex form. Others that are used in automated theorem proving, like disjunctive and conjunctive normal form, are not used, since their rewrite rules are not confluent. Instead, the Boolean ring normalization[1.2] is used which is confluent (modulo AC). The innovation is how these normalization steps are modified slightly and put together in particular with AC standardization.

All the standardization, normalization, and matching techniques can be used not just for theory interpretation search, but also to find approximations of theory intersections. The concept of theory intersection is introduced in chapter 5. Here, the task is to find formula translations such that the set of translated formulae valid in two given theories is maximized. This set of formulae is itself a theory that can be interpreted in the intersecting theories. Again, this task is undecidable for the same reasons as in theory interpretation search. The presented algorithm finds an optimal solution w.r.t. the explicitly given formulae of the two involved theories and w.r.t. to the given equivalence class determined by the rewrite rules in the normalization step. The basic idea of the algorithm is to try to match each formula of one theory to as many formulae as possible to the other formulae and combine as many as possible of the corresponding symbol

1.2. Surprisingly, Boolean ring normal forms have not gained much attention in automated theorem proving in spite of their nice properties.

renamings to a single consistent renaming. It turns out that this problem can be reduced to the maximum clique problem from graph theory.

Chapter 9 is about theory completion – another continuation of the theory interpretation idea: since once a theory interpretation is found all the translated theorems can be reused in the target theory. But this means that the set of explicitly known formulae of the target theory is increased, which may in turn facilitate further theorem reuse: suppose a former theory interpretation search on this target theory but with a different source theory has failed. Now that the set of known theorems in this target theory has increased it might be the case that it contains now a theorem that fits to an axiom of the source theory so that it finally could be interpreted automatically in this target theory. Applying this procedure of reusing theorems and retrying interpretation search exhaustively will be called theory completion: as many theorems as possible are collected from other theories via repetitive theory interpretation search. It is sketched how this procedure can be mapped to a Prolog-like forward chaining procedure.

All the theoretical investigations are finally implemented in a running prototype system implemented in Haskell. Chapter 10 starts with a short introduction to Haskell sufficient to understand the code fragments in the remainder of that chapter. Its main part is the detailed explanation of a "didactic system" that can be considered as a lite version of the actual system implementation of the system for theory interpretation and intersection search. In fact, the code presented there is compilable and it needs only a few additional lines to built a basic running system. The subsequent section is about desirable features for a mature theory interpretation and intersection search system. And the final section discusses what has actually been implemented with all the lessons learned.

Experiments have been conducted on the world's largest knowledge base of formalized mathematics in order to evaluate the presented theory and its implementation. Chapter 11 gives an introduction to that knowledge base – the Mizar Mathematical library (MML) – and to its translation into a format readable by the implemented prototype. The huge size of this (translated) library (>40000 theories and >4 million formulae) made several memory and search optimizations necessary that are also sketched in this chapter. Finally, several statistics are provided describing the experimental search results in particular for theory interpretation search and a few for theory intersection. The number of found interpretations and the search speed demonstrate the scalability of the search techniques implemented in the prototype system.

Figure 1.1 should give an impression of the size of this library and moreover a motivation why improvements in the research on theory management is really desirable. Even more so when we take into account that there are indeed several libraries of formalized mathematics of comparable size (e.g. from the CoQ or the NuPRL system).

Much more formalized content, however, is produced by the semantic-web community. Yet, it must be mentioned that most of those ontologies are simple taxonomies (i.e. concept hierarchies) where theory interpretations do not make much sense.

Finally chapter 12 summarizes all the theoretical and experimental findings of this thesis and provides an outlook of mainly two research aspects: 1) To evaluate with the authors of the Mizar library the found theory interpretations. The interesting question here is: which of the found interpretations are actually valuable for the working mathematician? 2) To develop the system from the prototype system to a robust and mature system – this means in particular to provide a user friendly GUI.

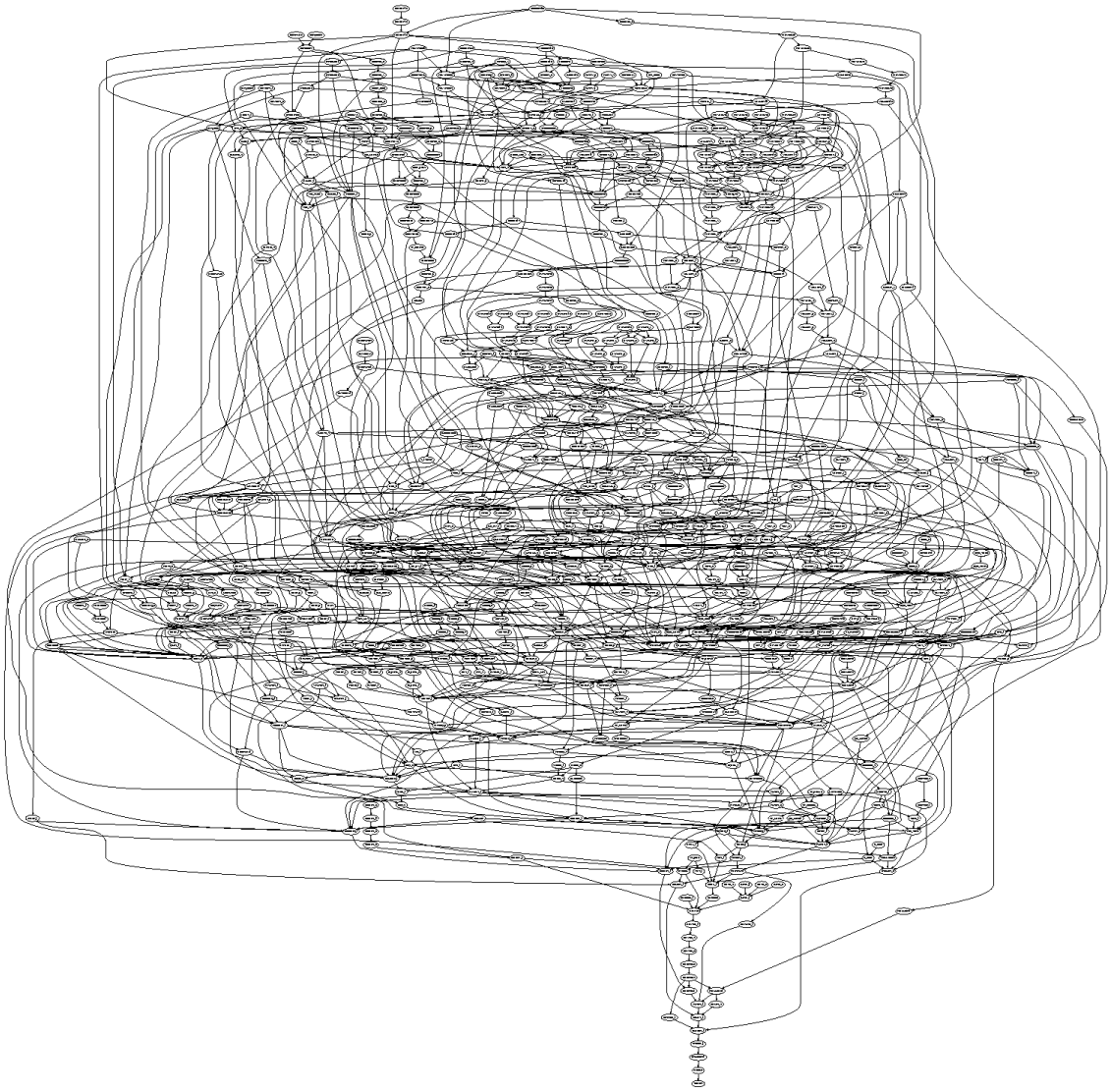

Figure 1.1. The theory graph of the Mizar Mathematical library. Every node represents a theory and every edge a theory import.

Chapter 2
Logical Reasoning

Logic has a long tradition of reflecting upon itself. In mathematics two fundamental disciplines have been evolved to study logic: model theory and proof theory. Both theories investigate a large variety of *logics*, but have not yet formally defined on a widely accepted basis what a *logic* in general is. A rather new approach to accomplish this goal on a most abstract level came from category theory with Joseph Goguen's et.al. pioneering work on **institutions** [24]. In essence it is an category theoretical abstraction of the fundamental model theoretical notions originated from Tarski's notion of semantics. A more recent article on the question "What is a logic" based on institutions is [50].

Jose Meseguer introduced with his notion of **entailment system** in [38] more proof theoretic aspects. There he defines a logic[2.1] as the combination of an entailment system and an institution. We will take his axiomatic definition of entailment as our basis, but with a slight adaption. The first adaption to mention here is the term "entailment": this term is somewhat dangerous as it is occupied by some proof theorists as well as by some model theorists, but with their own distinct meaning. To avoid misconceptions we will use the term "**derivability**" where Meseguer uses the term "entailment".

For now let us return to model and proof theory. Both theories operate on formal languages – which in general consist of logical and non-logical symbols. In this section we do not make further grammatical distinctions within a formal language. For convenience we call a formal language just **language** and its syntactic expressions simply **formulae**.

Model theory defines semantics of a language by mapping the non-logical symbols to mathematical objects. These mappings are called interpretations. For a given formula φ and a certain interpretation $\mathcal{I}$ we say that φ has a model iff $\mathcal{I}(\varphi)$ is a true mathematical statement. In model theory this is commonly denoted by $\mathcal{I} \models \varphi$, where $\models$ is called the **satisfaction relation**. A formula that holds for every interpretation is called **valid**. A formula φ is called a consequence of a set of formulae Γ, denoted by $\Gamma \models \varphi$, iff $\mathcal{I} \models \Gamma$ implies $\mathcal{I} \models \varphi$ for any interpretation $\mathcal{I}$. The $\models$ is therefore also called **consequence relation**. This is the central relation in the theory of institutions.

Proof theory does not give semantics to formulae in that sense, but rather investigates syntactical transformations of formulae. These transformation rules – called **inference rules** - constitute a **logical calculus**. For a given calculus we write $\Gamma \vdash \varphi$ iff the formula φ is **derivable** from the set of formulae Γ via the inference rules of that calculus. Γ might be the empty set, in that case we call φ a valid formula in this calculus. The derivability relation corresponds to Meseguer's entailment relation in [38].

Model theory and proof theory are connected by the two fundamental notions **correctness** and **completeness**: a calculus is called correct iff $\Gamma \vdash \varphi$ implies $\Gamma \models \varphi$ and it is called complete iff $\Gamma \models \varphi$ implies $\Gamma \vdash \varphi$ for any set of formulae Γ and any formula φ.

2.1. More precisely: a *logical system*.

In formalized knowledge bases theorems are typically statements which have been proved by (semi-) automated theorem provers which implement a certain logical calculus. Hence such knowledge bases have a proof theoretic nature. Since this work is targeted to such knowledge bases we will adopt a proof theoretic perspective on the logical relation between formulae through this thesis. Actually, we even do not care about the method how a proof of a theorem was accomplished: the involved theorem prover might be flawed or its calculus incorrect. We even accept intuition or belief as proof as long as the properties of derivability – i.e. entailment system in Mesgeuer's words – are respected.

2.1 Derivability

The basic motivation for the definition of an derivability system is to describe the commonalty of all reasonable calculi instead of defining a specific calculus:

> "...we shall make abstraction of the particular [inference] rules used to generate the relation $\vdash$ and concentrate on the relation itself. Indeed, the entailment relation plays a more central role, since it remains the same across the many different proof calculi ...[p. 282]. ...[the] conditions [of the entailment system] should not be confused with particular rules of a specific calculus....[p. 283][38]"

In this sense our following definition of derivability has to be considered independent of concrete calculi. Moreover, we neglect any model theoretic issues. Strictly speaking the formal system we are going to investigate is not a logic in the sense of [38] or [50]. Nevertheless, this formal system becomes a logical system if the user's derivations are logical, i.e. obey a correct calculus. However, this is not a our concern, but completely up to the user.

Let us now informally describe these commonalty of all reasonable calculi as identified by Meseguer:

1. *reflexivity*: we can always prove a formula if we can assume it,
2. *monotonicity*: we can always prove with more assumptions what we can prove with fewer,
3. *transitivity*: using as an additional assumption something already proved should not give us more conclusions than those already entailed by our original assumptions, and
4. *translation*: we may change our signature which should neither affect the semantics of our formula nor the validity of our proofs.

A formal definition of derivability systems will be provided in chapter 4. There we will extend Meseguer axiomatization of derivability system by an axiom that takes the inductive construction of expressions out of signatures into account. Since the author is not aware of any formal language that does not construct inductively its expressions from its signature, this additional axiom is not considered as severe restriction of the natural concept of derivability. But even a weaker constraint will be stated in the additional axiom: if signature morphisms are equivalent on their common domain then the induced formula translations should be equal too. We will elaborate the rational behind this in chapter 4.

2.2 Theories

Mathematics, of course, provides more means for knowledge structuring than only derivations of conclusions from assumptions. For mathematicians formulae are not just objects for syntactic manipulation, but a formalization of mental concepts like geometry, probability, arithmetic, etc. Even in the most abstract field of mathematics formulae are intended to formalize (abstract) concepts. A formula like $e(t) < e(s)$ can be true or false depending on what the involved symbols mean. It is in the center of model theory to analyze possible interpretations that make formulae true. In the given example the interpretation "t for triangle, r for rectangle, $e(.)$ for number of edges, and $<$ for the less than relation" would make the formula true. Formalization can be regarded as the inverse mental operation of interpretation: $e(t) < e(s)$ is a formalization of the (true) sentence "a triangle has fewer edges than a rectangle" assuming the inverse association of symbols and concepts from the above interpretation.

Explaining the general relation between formulae and their meaning as above is of course an unacceptable oversimplification. An in depth investigation of this topic would soon lead us to philosophy of mathematics which is out of our scope. Nevertheless, our simplified view on formulae and their meanings points to the following unquestionable observations: whether a formula has a meaning or not depends on the context. A formula has a meaning if all it symbols are understood. Further on, every meaningful formula has either a known or unknown truth value. In any context the truth value of every meaningful formula is either self-evident or provable from the self-evident formulae. Context and self evidence of formulae are mutually dependent.

In "Proof and Refutations" [37] Imre Lakatos has convincingly shown how context and self evidence involve each other and how they have evolved together in history. One of the major conclusions of his philosophical as well as historical investigation is that self evidence is not absolute. Since the longing for certainty has always been a driving force for doing mathematics the observed instability of self evidence has become a growing disconcertment for many mathematicians.

Eventually at the end of the 19th century mathematics has experienced a remarkable effort towards foundations of its disciplines with its well known results in arithmetic (Peano), set theory (Cantor), theory of real numbers (Dedekind), etc. Most of them became part of the standard curriculum in mathematics. Many mathematical contexts have been consolidated thereby to theories based on a fixed set of axioms.

The intention was to find for each context a minimal, sufficient and necessary set of evident statements from which all other evident statements can be proven. The context is then called **theory** and the evident statements **axioms**.

Once fixed, the axiomatization of theories obtain a normative character: Reasoning in a theory assumes accepting its axioms as evident. Modern mathematical logic eventually defines a **theory** as being **constituted by** a set of axioms. Due to this turn towards the primacy of axioms, theories can be defined by arbitrary axioms without reference to any mental concepts. With the invention of **formal languages** reasoning on theories was investigated as a theory on its own. From our perspective even more interesting: reasoning became a syntactical operation that can be performed not only by humans, but also by machines. Formal theories do not rely on meaningful descriptors, but they can have their own set of symbols – i.e. the **signature** of a theory. More precisely, a theory is determined by a signature and a set formulae built from this signature called axioms. All formulae derivable from these axioms are called theorems. This is of course a strong reduction of all the variants of sentences that actually can be found in

mathematical textbooks and also at most knowledge bases of formalized mathematics where we find:

- *axioms* and *definitions*; the latter in variants such as: *simple*, *inductive*, *recursive*,...
- *theorem, lemma, corollary, proposition, property*, ...
- *conjecture, thesis*,...

From a proof theoretic perspective, however, all sentences fall in one of the three categories: the **given**, those **derivable**/refutable from the given, and those not derivable/refutable from the given. Further sub categorization, as those listed above, does not have a proof theoretic meaning, although they are quite relevant for mathematicians, e.g. to express their importance.

It is clear which of the above listed terms fall into which category – except for the *conjecture* type, whose status is unclear, but from a proof theoretical view they are derivable or not, we as humans just do not know whether they are. Our concept of *theory* is that of positive knowledge, i.e. a container of given sentences, which we call axioms, and sentences derivable from them, which we call **theorems**. Since the latter are implicitly determined by the axioms, it is sufficient to represent a theory just by the signature and its axioms.

One of the properties of the derivability relation is its invariance with respect to a **translation** of the signature. Signature morphisms translate signatures, and induce a sentence translation. So we can apply them on theories – more precisely we apply a signature morphism on the signature of the theory and the induced sentence translation on the axioms as well as on the theorems. Theory translation is something very common in mathematics. One of the most prominent examples from Algebra is probably that of a group where we typically find both presentations in textbooks, the *additive* and the *multiplicative*:

$\mathcal{G} = \langle G, +, -, 0\rangle$	$\mathcal{G} = \langle G, *, .^{-1}, 1\rangle$
$\forall a, b, c.\, a + (b + c) = (a + b) + c$	$\forall a, b, c.\, a * (b * c) = (a * b) * c$
$\forall a.\, a + 0 = a$	$\forall a.\, a * 1 = a$
$\forall a.\, a + (-a) = 0$	$\forall a.\, a * a^{-1} = 1$

Table 2.1. Group axiom in additive and multiplicative notation.

Theory translation, however, is not only intended to serve presentational adaption to a mathematicians preference. The deeper purpose is to find theory interpretations which basically makes use of the *translation* axiom of derivability systems. A formal definition of *theory* and *theory interpretation* will be provided in chapter 4.

2.3 Notions of Theory in Logic and Practice

We have to distinguish two notions of "theory": in the pure logical context a theory comprises an infinite set formulae composed of a finite set of axioms and the infinite set of theorems entailed by these axioms. In practice, however, our explicit knowledge about is always limited – every mathematical text book or formal library necessarily contains only a limited number of theorems. To give it a handy name we want to call a theory in pure logic context an just **theory** and in the practical context a **explored theory** (sometimes we will omit the adjective "explored" when it is clear from the context what notion is meant). A explored theory is hence always only an approximation of a theory – it reflects our current knowledge state about a theory.

2.4 The Role of Theory Interpretation in Mathematics

In modern history of mathematics one of the most esteemed project was the foundation of mathematics as a whole. In particular the rise of modern set theory seemed to provide the appropriate means to achieve the best results. Probably the most prominent foundational set theories are known under the names *Zermelo-Fraenkel* and *von Neumann-Bernays-Gödel*. Many decades later the *Tarski-Grothendieck* set theory became interesting in particular in the regime of mechanized reasoning. Common to all of them is the intention to find a foundational mathematical theory such that all other mathematical theories can be defined in terms of that. Thus all fields of mathematics which have been conceived in early ages as completely unrelated could be unified in one theory. However, soon it was shown that none of such foundational theories can be proved to be consistent. On the other hand, no inconsistency has been found in any of these foundational theories and most mathematicians nowadays believe that they are consistent.

As for the mathematicians' practice these foundations apparently have not had a substantial effect: mathematicians still work all in their very particular theories – and the diversity of mathematical theories increases rapidly. Only a very little minority of mathematicians actually care how their theory can be founded on one of those set theories. It is simply out of the interest of, e.g. a specialist in differential geometry, to define the concept of diffeomorphism in terms of the notion "set" and its element relation. He may assume that this is always possible, but more likely he even does not care about that at all. In fact the dominant mathematical practice continues to organize mathematical knowledge in an extensive variety of theories. With the foundation of mathematics, its theoretical self-conception as a unified science may have consolidated – e.g. number theory and geometry are considered both as parts of mathematics whereas in earlier days they were regarded as different sciences – but practically mathematicians are focused on their own theories all having their own axioms. This axiomatic program was substantially promoted by Hilbert – the introduction to *The Foundations of Geometry* reflects as paradigm this attitude:

> The following investigation is a new attempt to choose for geometry a simple and complete set of independent axioms and to deduce from these the most important geometrical theorems in such a manner as to bring out as clearly as possible the significance of the different groups of axioms and the scope of the conclusions to be derived from the individual axioms.

As the title already says it is a foundation, not of the mathematics as a whole, but of a particular mathematical theory. We can say the axiomatic program does not investigate the whole mathematics as one **big theory**, but rather divides the whole mathematics into many **little theories**[2.2]. Another very important aspect of the axiomatic program is that the objects of a theory are completely described by the axioms, i.e. they do not bear any implicit meaning just by their given names. About the objects in the foundation of geometry Hilbert used to note that the terms *point, line, plane*, and others, could be substituted by *tables, chairs, glasses of beer* and other such terms. This renaming would not change the correctness of the derived theorems. We only give those objects very specific names to support our intuition about them, but we are always free

2.2. It should be mentioned that the *Foundations of Geometry* were written before the earliest set theoretic foundations of mathematics, but the axiomatic spirit of little theories has been kept vivid until today in spite of those foundation efforts towards one *big theory*.

to translate them to other names. In fact this freedom of **theory translation** is the essential power of the axiomatic program since it allows to investigate a logical relation between two theories which were unrelated before just because their objects had different names. A theory translations which translates axioms of one theory into theorems of the other is called **theory interpretation**. These are the translations mathematicians are most interested in since they allow to **reuse theorems** of the source theory inside the target theory. In §15 and §17 of *The Foundations of Geometry* one can find a simple example of this theory inclusion and theorem reuse respectively, where certain line segments are shown to form an ordered field with appropriately chosen operations.

In the logical literature theory interpretations have been used at least since the 1950's. In the classic work of Tarski, Mostowski, and Robinson [1], for example, theory interpretations are used as a fundamental means to prove undecidability of certain theories. In general logicians have used this technique since then to prove metamathematical properties about theories, particularly decidability, undecidability, consistency, relative consistency, and logical independence[2.3]. These properties are interesting to investigate not just for *little theories*, but in particular for the *big theories* – to give some prominent examples:

- Gödel's undecidability theorems on first-order logic and Peano arithmetic.
- Neumann-Bernay-Gödel set theory remains consistent[2.4] when the axiom of choice and the continuum hypothesis are added to the axioms.
- The independence of the parallel axiom from the other Euclidean axioms.

The last example points to another important aspect of the axiomatic program – which is also referred to in *The Foundations of Geometry*: the attempt to always find a simple and complete set of independent axioms for the theory under consideration. Obviously this maxim to have independent axioms has a very long tradition as the more than 2000 year enduring debate about that parallel postulate has illustrated. One can think of different motivations for that maxim: before the rise of Hilbert's modern axiomatic program there was an unease with the parallel postulate as it is not as self-evident as the other Euclidean axioms. In the modern axiomatic program this aspect does not matter in the first place, since it abstracts from every implicit meaning terms of the theory may have in our mind, but only take into account the meaning determined by the theory's axioms. Here the requirement to keep an axiom system simple and independent has at least one other motivation, notably to ease proofs whether the axioms of the given theory are theorems in another theory or not: 1) The simpler the axioms of a theory the easier to prove or refute their validity in another theory; 2) If an axiom is derivable from some other axioms (i.e. not independent) then it is sufficient to prove only the other axioms of the source theory inside the target theory in order to reuse the theorems of the source theory there. A further reason to keep axiom systems small is that the more axioms there are the more dangerous it becomes that they form an inconsistent theory.

2.3. See for instance [46, 11].

2.4. But it is not known wether this set theory is consistent. Hence, the additions are only relatively consistent.

Chapter 3

State of the Art in Theory Management

With the advent of mechanized reasoning the principles of the axiomatic program have been formalized and steadily refined in order to benefit from these principles in mechanized reasoning very much like the mathematicians already. The idea of little theories were probably first introduced by Rod Burstall and Joseph Goguen [8] to equip specification languages with a formal semantics. In fact the works of Burstall and, in particular, of Goguen have a long-standing impact on formal methods, i.e. on specification languages, theorem provers, and most recently to logical frameworks too.

3.1 Little Theories in Formal Methods

A **specification language** is a formal language used in computer science to formally describe a system on a high level. In contrast to programming languages they are typically not executable and they are not intended to describe implementation details, but rather to model programs as algebraic structures. Basically a system specification is an axiomatic system. The merits of Burstall and Goguen in this respect is the formal semantics they developed for their specification languages. Clear [7] is the earliest specification language designed by Burstall and Goguen with such a formal semantics. Another early and popular specification language influenced by their work is Larch [28]. A contemporary specification language with an active research community is CASL [9, 10]. It was designed by a group of experts as a general-purpose language and includes carefully selected features from many previous specification languages, as well as some novel features that allow specifications to be written much more concisely. It may ultimately replace most of the previous languages.

Once a system is formally described in a specification language a **theorem prover** can be used to reason about that specification. There exist fully- and semi-automated theorem provers – the former are usually used to prove rather mechanical proofs, whereas the latter are taken for more sophisticated proofs where the machine needs help from the experienced user. Concerning the little theory idea, Burstall and Sannella integrated rudimentary aspects of it in an extended version of the LCF theorem prover [49], and Goguen in OBJ [25]. Also the Larch Prover [23] has some restricted support of little theories. Other theorem provers have implemented little theory related mechanisms ad hoc, e.g. HOL [56]. The first interactive theorem prover that has been designed from start to support little theories is IMPS, an Interactive Mathematical Proof System [19] developed by William Farmer and Joshua Guttman. Their article, with the title "Little Theories" [16] is actually the origin of the notion "little theories" as it is used in this

thesis. Again the impact of Burstall and Goguen's work is visible in that paper. As IMPS is probably the theorem prover with the best integration of the little theory concept, we will discuss it in more depth below.

Influenced by IMPS, systems have been developed exclusively geared towards managing little theories: whenever little theories are added, removed, or modified this action may cause new proof obligations. Such **theory management systems** try to reuse old proofs to free the user from discharging redundant proof obligations by hand. For that they use an elaborated internal data structure for little theories based on a concept called **development graph**. The first system implementing the development graph was MAYA [3] and a later improved system is HETS [40]. Below we provide an overview of these systems as they are most closely related to the work of this thesis.

The semantics of development graphs relies on the notion of **institution** introduced by Goguen and Burstall [26, 24] and **entailment systems** as introduced by Meseguer [38]. We have already introduced entailment systems, but not yet institutions. In fact Meseguer tries to answer in [38] the question "what is a logic?" and his work has goals in full agreement with Goguen and Burstall's theory of institutions. He, however, addresses proof-theoretic aspects not addressed by institutions which is rather about model theory. Institutions can be viewed as the model-theoretic component of Meseguer's notion of general logics, whereas the entailment system represents the proof-theoretic aspect. The consequence relation $\models$ is characterized in the theory of institutions and the derivability relation $\vdash$ in the entailment system. A logic in Meseguer's sense basically combines institutions with entailment systems with the soundness constraint that derviability must be a subrelation of the consequence relation. As already mentioned in the previous section, we will only care about derviability explicitly, but keep its connection to model theory in mind.

Logical frameworks (LF) are a general, purely proof-theoretic approach to logics: an LF is a meta language for the specification of deductive systems, i.e. systems defined by axioms and rules of inference. Historically, the first logical framework was AUTOMATH developed by de Bruijn [12] whose goal was to provide a tool for the formalization of mathematics without foundational previous knowledge. Many of the ideas from AUTOMATH influenced modern systems significantly. Another impact on modern logical frameworks came from Martin Löf's constructive type theory, which influenced LF [30] and the modern variant TWELF [45]. Concurrent with the development of LF, frameworks based on higher-order logic and resolution were designed in the form of generic theorem provers. ISABELLE is one of them which we will discuss below as it is probably the one with the most elaborated support of little theories.

3.2 IMPS

IMPS [19] is a system for a rigorous development of mathematics. The intention of IMPS is to serve students as a general mathematics laboratory as well as an assistant tool in mathematics research. It contains an extensible database for mathematical objects and an interactive theorem prover[3.1]. The underlying logic, called LUTINS [17], is a version of simple type theory with partial functions and subtypes which allows for natural formalization of wide range of mathematics.

3.1. In fact the theorem prover might be considered as the core of IMPS, but we dwell on that only to the extent the concept of theories is involved.

As usual translations of formulae – and thus of whole theories – are inductively defined over **signature morphisms**, i.e. mappings of the primitive symbols (atomic sorts, variables, constants, function, and relation symbols) of the source theory to objects of the target theory. The signature morphisms supported by LUTINS are quite powerful: atomic sorts can be associated with (atomic or compound) sorts and closed unary predicates; variables are associated with variables; and constants are associated with closed expression of appropriate sort. A simple example should illustrate the expressiveness of translations in LUTINS: assume a signature morphism where the type α is associated with the unary predicate $\lambda x_\alpha.x < 0$ and the relation symbol $\geqslant$ with the expression function $\lambda x_\alpha,\ y_\alpha.|x| > y$. This determines a translation τ translating, for example, the formula $\forall x_a.x \geqslant x$ as follows:

$$\begin{aligned}\tau(\forall x_\alpha.x \geqslant x) &= \forall x_\alpha.\tau(\alpha)(x_\alpha) \Rightarrow \tau(\geqslant)(x\,,x)\\ &= \forall x_\alpha.(\lambda x_\alpha.x<0)(x_\alpha) \Rightarrow (\lambda x_\alpha, y_\alpha.|x|>y)(x\,,x)\\ &= \forall x_\alpha.x<0 \Rightarrow |x|>x\end{aligned}$$

Particular features of LUTINS exhibited by this example are

- how a formula is translated to a more complex formula by associating symbols to expression functions: $x \geqslant x$ becomes $|x| > x$; and
- how a formula can be relativised by associating types to unary predicates: $|x| > x$ where x requires $x < 0$.

Those translation features are beyond those found in the pure theory management systems MAYA and HETS (s. below).

As mentioned above, a translation τ from source theory S to a target theory T is called interpretation of S in T if $\tau(\varphi)$ is a theorem in T whenever φ is a theorem in S. In [17] Farmer lists sufficient conditions when a theory translation in LUTINS is a theory interpretation: at first all axioms of S must be theorems in T and moreover sorts must not be empty, all constants must be defined for their sort, and some sort inclusion property must hold.

The mathematical database of IMPS is a file based database [52]. Working on this database thus means inserting, editing, or removing text fragments representing mathematical objects in so called theory files. All top level items are declared in a fixed set of so called def-forms (with meaningful names like def-theorem, def-constant, etc.). These are labeled tree structures. Below the most prominent objects together with their most important child nodes are listed (optional child nodes are listed in rectangular brackets):

- def-section: id, section, file-names
- def-theory: id, parent-theories, language, axioms
- def-translation: id, source-theory, target-theory, sort-, constant-pairs.
- def-theorem: id, formula, target-theory, [source-theory, translation, proof]

def-section is used to conveniently load packages of knowledge into working space. Def-theory allows multiple theory inheritance. As IMPS supports only one logic the notion "language" can be understood as "signature". The language of a theory is the union of the explicitly named language and the languages of the parent theories. Thus theories can be built up in a hierarchy such that the whole mathematics of the IMPS database is based on some foundational theory (e.g. Zermelo-Fraenkel set theory). However, the user is not forced to commit to one particular hierarchy, but allows for relating theories of different languages via translations.

For every inserted translation IMPS tries as much as possible to verify that the translation is an interpretation. If successful it marks the translation as interpretation otherwise it prompts the user to prove the outstanding obligations. This is done by def-theorem where the formula is the obligation in the source theory which is translated then to the target theory and finally verified by an explicit proof provided by the user.

Recent theoretical development based on IMPS A successor framework of IMPS is the *biform theory* [20] which is simultaneously an axiomatic theory and an algorithmic theory; i.e. it provides a formal context for both deduction and computation. Moreover it is intended to be used with several background logics simultaneously.

3.3 MAYA and the Development Graph

The **development graph** [3] is a theory management framework implemented in the MAYA [3] system. It was designed to maintain and utilize the structured mechanisms of various specification languages along the evolution and verification of industrial-size software. In this setting, software systems as well as their requirement specifications are formalized in a specification language like CASL or VSE-SL. The MAYA system provides a generic interface to plug in additional parsers for the support of other specification languages. Moreover, MAYA allows the integration of different theorem provers to deal with the actual proof obligations arising from the specification. MAYA, unlike IMPS, does not come with its own theorem prover.

Fundamental for the understanding of MAYA are the notions **verification in-the-large** and **verification in-the-small**. Basically verification in-the-large means the verification of postulated theory[3.2] interpretations whereas verification in-the-small is about verification of postulated assertions within a theory.

MAYA parses and transforms the structured organization of theories written in a specification language as text files into the structure of a development graph whose nodes represent theories and whose edges represent logical relations between those theories. The development graph is the central data structure for a uniform mechanism of verification in-the-large in the evolutionary modification of specifications. Any local modification in one theory may trigger new proof obligations to verify the relations between the connected theories. And the other way round the failure to prove a proof obligation usually gives rise to modify the specification. MAYA supports this evolutionary process as it calculates minimal changes to the logical representation readjusting it to a modified specification while preserving as much verification work as possible. To accomplish this MAYA uses a **development graph calculus** and conducts **difference analysis** between two states of the development graph.

MAYA abstracts from any specific logic by allowing any logic that satisfies the entailment properties: reflexivity, monotonicity, transitivity, and signature morphism invariance of the deduction relation as described in chapter 2. This logical abstraction is based on theoretical framework of entailment and institutions [38, 26, 15].

The development graph is a graph whose nodes represent theories and whose links represent logical relations between those theories. There are four types of edges:

local definition links	**global definition links**
local theorem links	**global theorem links**

3.2. We use the notions theory and specification synonymously.

Global links are transitive relations, i.e. if theories T_1, T_2 and T_3 are on a path of global links then T_1 is implicitly related with T_3. Local links in contrast do not have this transitive semantics. All links are labeled with signature morphisms (related to translations in Imps). The axiomatization of each theory is split into a local part which is attached to the node as a set of formulae and into a global parts, denoted by ingoing definition links, which import the axiomatization of other nodes via the signature morphisms attached to the links. While a local link imports only the local part of the axiomatization of the source node of links, global links are used to import the entire axiomatization of the source node (including all the imported axiomatization of other source nodes). In the same way local and global theorem links are used to postulate relations between nodes, namely theory interpretations. The distinction between local and global is a specific feature of development graphs.

The development graph calculus provides inference rules to decompose the global links into local links preserving the semantics of the logical relations between theory nodes: all theorem links between theory nodes hold in the original graph iff all local relations between those nodes hold after the decomposition. In the course of specification change some of the links' status are changed from proven to unproven. Of course, the goal is to prove all postulated theorem links[3.3] again which can be done either in the original graph or in the decomposed graph. But graph decomposition reduces complex proof obligations to smaller local proof obligations. A local change of some theory typically changes the status of the whole graph to unproven. Usually this can be repaired locally, i.e. without reproving all theorem links again. Decomposition to local links exactly serves the purpose to identify redundant proof obligations, since all the status of local theorem links are maintained. An unproven global theorem link typically decomposes to many local theorem links most of whom, however, are already proven (either as there is a parallel definitional link, or a proof was found before in the evolution of the graph). Thus development graph decomposition uncovers how to avoid redundant proofs (verification in-the-large). If there are still open proof obligations after the complete decomposition, then these obligations are passed to the external theorem prover (verification in-the-small).

Basic operations on the development graph. In order to build complex theories systematically and to relate (or interpret in Imps words) them to each other Maya provides the following basic operations:

- Nodes: insert/delete nodes
- Links: insert/delete global/local definition/theorem links
- Local signature: insert/delete symbols
- Local axioms: insert/delete/change local axioms

After modifying the development graph Maya runs a difference analysis between the old and the new graph.

The difference analysis aims at the preservation of as many validated conjectures during the transformation of the old proof to the new development graph. This means the difference of axioms and signatures (between old and new development graph) which were available during proof time must be determined.

The basic operations which change the development graph require a notion of *node-equivalence*. As there does not exist an optimal solution heuristics are applied based on the number of shared local signature symbols plus similarity of the incoming definition links. Thereby nodes and links of the new and old development graph are associated.

3.3. If this is not possible, modify the specification and then try again proving.

3.4 HETS

HETS, the Heterogenous Tool Set [39, 40], is the first system designed to support theory management with theories in different logics. IMPS is confined to one logic[3.4] (LUTINS) whereas in MAYA the user has the freedom to choose from many logics, but once chosen the whole specification must commit to this logic. HETS allows specifications in different logics simultaneously. The motivation for HETS relies on the observation that several logics are in use in the area of formal specification and logics:

- logics for specification of data types,
- process calculi and logics for the description of concurrent and reactive behavior,
- description logics for knowledge bases in artificial intelligence and for the Semantic Web,
- logics for reasoning about space and time,
- logics for specifying security requirements and policies, etc.

So far there is no logic equally suitable for all these applications. Even if such a logic will be found in future, such a monolithic formalism would be hardly manageable, if its consistency could be guaranteed at all. Often the biggest part of a system specification can be formalized in a simple dedicated formalism, whereas only little parts need extensions in expressiveness. Similarly, we observe in mathematics that most parts of mathematics can be formalized in first order logics, only the smaller part needs higher order constructs. But in some cases the more expressive logics allows for more elegant formalization. In general, being the most appropriate formalism is a matter of the object to be formalized, i.e. complex problems have different aspects that are best specified in different logics. HETS aims to satisfy this need by supporting heterogeneous multi-logic specifications and logic translations based on a rigorous formal semantics. Thus specifications developed in different logics with a different approach can be related, i.e. there is a formal interoperability between languages and tools. HETS gains flexibility by providing formal interoperability, i.e. integration of different formalisms on a clear semantic basis. Hence, HETS is a flexible, multi-lateral and formal integration tool. Unlike other tools, it treats logic translations as first-class citizens.

The formal background of HETS is the notion of institutions and entailment systems. It shares this part of formal background with the development graphs as implemented in MAYA, but extends to **institution comorphisms** as the formal basis of logic translation (cf. [50]). Basically an institution comorphism maps signatures to signatures, sentences to sentences, and models to models[3.5] such that satisfaction is preserved.

The architecture of the HETS system is shown in figure 3.1. For each logic there is a specific parser, that translates the logic from its specific syntax to an internal representation of an abstract syntax. A subsequent static analyzer transforms an abstract syntax tree to a development graph. Starting with a node corresponding to the empty theory, it successively extends (using the static analysis of basic specifications) and/or translates (along the logic translations) the theory, while simultaneously adding nodes and links to the development graph.

3.4. Farmer extended the little theory approach, however, with multi logic support in [18].

3.5. Actually models are mapped against the direction of the comorphism.

Syntax and semantics of heterogeneous specifications as well as their implementation in Hets is parametrized over an arbitrary logic graph. Currently supported logics and logic translations are shown in the middle of the figure.

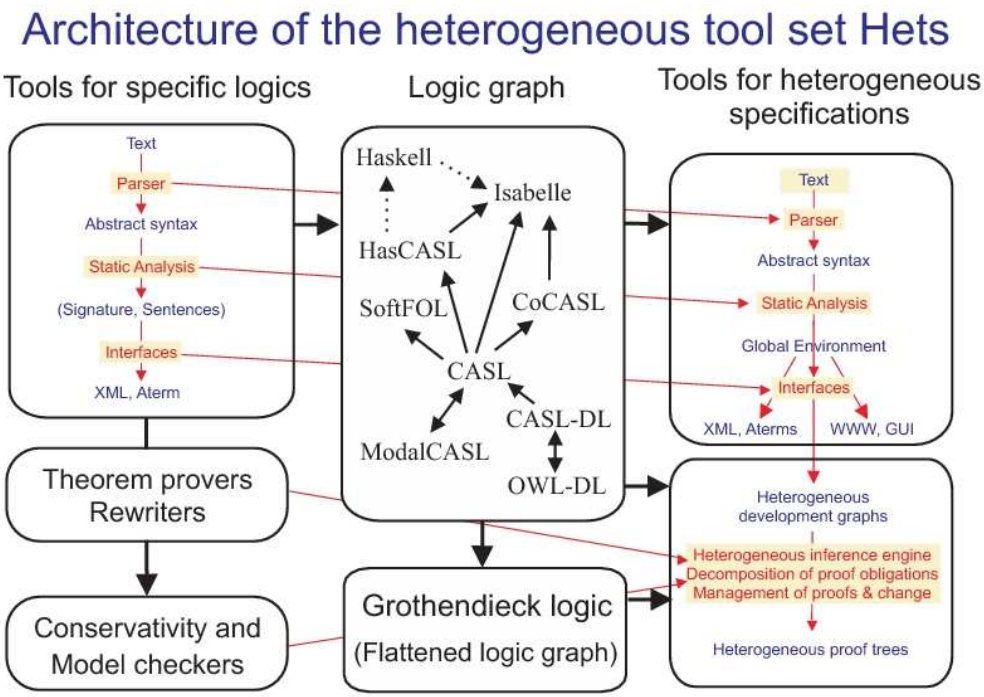

Figure 3.1. Architecture of the Hets system.

Indeed, the Hets modules implementing the logic graph can be compiled independently of the Hets modules implementing heterogeneous specifications, and this separation of concerns is essential to keep the tool manageable from a software engineering point of view. For proof management, Maya's calculus of development graphs has been extended with hiding and adapted to heterogeneous specifications.

3.5 Isabelle

The Isabelle system [43] provides a generic infrastructure for building deductive systems (programmed in Standard ML), with a special focus on interactive theorem proving in higher-order logics. In earlier versions of the system users had to conduct interactive proofs by means of Isabelle's built-in ML functions or by defining proofs tactics on top of them. This rather low level interaction with Isabelle has been improved by the interpreted language environment Isar [53], which has been specifically tailored for the needs of theory and proof development. Compared to raw ML, the Isabelle/Isar top-level provides a more robust and comfortable development platform, with proper support for theory development graphs, single-step transactions with unlimited undo, etc.

Apart from the technical advances over bare-bones ML programming, the main purpose of the Isar language is to provide a conceptually different view on machine-checked proofs [15, 17]. "Isar" stands for "Intelligible semi-automated reasoning". Drawing from both the traditions of informal mathematical proof texts and high-level programming languages, Isar offers a versatile environment for structured formal proof documents. Thus properly written Isar proofs become accessible to a broader audience than unstructured tactic scripts (which typically only provide operational information for the machine). Writing human-readable proof texts certainly requires some additional efforts by the writer to achieve a good presentation, both of formal and informal parts of the text. On the other hand, human-readable formal texts gain some value in their own right, independently of the mechanic proof-checking process.

The Isabelle/Isar framework is generic in that any object-logic is supported that conforms to the natural deduction view of the Isabelle/Pure framework. Similar to Maya various logics are supported. Major Isabelle logics like HOL [7], HOLCF [5], FOL [11], and ZF [12] have already been set up for end-users. Concerning logic translations, the user is free to choose one of the logics at the beginning of theory development, but then has to commit to the initial choice. Hets is more flexible in this respect, as it can handle theories in various logics simultaneously.

Isabelle's focus is much more on theorem proving than theory management like Maya and Hets. Nevertheless it incorporates a relatively limited theory management concept inspired by the development graph framework. It has its origin in Florian Kammüller's PhD thesis [34] where he established the concept of **locales**, a means in Isabelle to enable local definitions and assumptions of a limited or temporary scope in a proof. Markus Wenzel integrated locales into Isar [53] which appeared to be a very natural extension since both are based on the notion of context. Moreover, Wenzel extended locales by locale expressions, which allow to combine locales (in the sense of theory imports) more freely. Previously only linear inheritance was possible. The current notion of locale resembles the notion of theory in the development graph. Locale interpretations in Isabelle [5] correspond to theory interpretation as described above. A distinction between local and global connection between locales corresponding to Maya's local/global links does not exist for locales. Since Isabelle's focus is less on theory management this distinction as well as a decomposition calculus for the development graph has not been integrated. On the other hand the locale concept is more general than the theory concept in Hets or Maya: locales can be also very local contexts inside proofs – Maya and Hets do not have access to the inside of a proof. In fact this feature to allow for very local contexts reflects Isabelle's emphasis of verification in-the-small and is considered as control information that works well with particular proof procedures – or in Clemens Ballrin's words[3.6]: reasoning in the large is used to provide suitable contexts for reasoning in the small.

3.6 Conclusion

Maya is particularly strong for management of theory changes: its development graph calculus decomposes all global proof obligations caused by a theory change to local proof obligations. As typically most of the local proof obligations turn out to be already discharged beforehand the decomposed graph relieves the user from redundant proof work. Characterizing links between theories as local or global is a means for very selective

3.6. See in the conclusion of [5].

theory structuring – not known by the other systems. Maya is designed to work with many different logics, but specifications must be homogeneous, i.e. once a logic is taken it is fixed for all theories. Hets in contrast is heterogeneous, i.e. different theories may be formalized in different logics, thus it extends Maya in this respect. Theories in Hets can not just translated to different signatures, but also to different logics. Both systems have a rather limited notion of signature morphism which is essentially restricted to sort and symbol renaming[3.7].

Imps notion of translation is more expressive as it is based on a signature morphism that additionally allow to map symbols to terms and sorts to predicative relativations. Concerning the choice of logics, however, it is restricted to the one logic Lutins – which empirically proves to be expressive enough for formalization in all fields of mathematics, though. Management of theory change is not supported in Imps, however automated proving of postulated theory interpretation. Isabelle neither has a clear defined semantics for management of theory change, like the decomposition calculus. Like Maya it understands many logics, but not heterogeneously as Hets. Signature morphisms in Isabelle are basically mere renamings of sorts and constants as in Hets and Maya. Specific to Isabelle is the ability to enable context not just on the theory level, but also within proofs.

3.7 Critique

Theory interpretation is the key concept of this thesis and a very important concept in all the above mentioned systems. Its main purpose is **knowledge gain by theorem reuse**. But none of these systems assist the user to find the "right" signature morphisms for potentially useful theory interpretations between the "right" theories. In current systems it is left to the user to choose a source theory S for a given target theory T and to find (if possible at all) an appropriate signature morphism σ and finally prove that the σ-translated axioms of S (and thus the whole theory S) hold in T. However, finding all signature morphisms σ that map all axioms from S into valid statements in T is a many to many formulae matching task that can be automated to some extent. Thus some theory interpretations can be automatically found without generating any proof obligations. **Automated search for theory interpretations** is the principal theme of this thesis.

Such a service turns out to be very useful whenever a new theory is supposed to be integrated into a large knowledge base. Other theories can only benefit from the new theory if it is logically connected to them – via definition or theorem links. The new theory could be a source theory S for several target theories in the base. All the theorems of the source theory would spread out to those target theories. And those theories whose axioms subsume the axioms of the new theory S could use it to import those axioms and thus integrating S into a inheritance hierarchy. At the same time the new theory could also be a target theory T for several source theories in the base. In this case all the theorems from these source theories would propagate to T. And the source theories whose axioms are subsumed in the target theory T could be used for axiom inheritance. This service becomes even more desirable when we think of the integration of two independently developed knowledge bases. Here every theory of the one base is new to the other base. It is very likely that there are many theory interpretations between two such bases, since most libraries start with more or less the same concepts e.g. from basic algebra. We will introduce automated search for theory interpretation in the next chapter.

3.7. In Hets this depends on the logic though.

Often it may happen that there is no theory interpretation between a certain set of theories which nevertheless have very much in common. In mathematical history the family of geometries is a popular witness – just to give some examples: Euclidean, affine, projective, hyperbolic geometry, etc. These families of theories gave reason for the emergence of new theories representing exactly these commonalities. For instance *absolute geometry* (also known as *neutral geometry*) was introduced by Janos Bolay in 1832 by just omitting the parallel postulate in the Euclidean geometry. Obviously this new theory can be interpreted in the Euclidean geometry, but also in various non-Euclidean geometries such as hyperbolic geometry. We will call this process **theory factorization** inspired by the established term *code refactoring* in software engineering where common code is extracted from various places to a single place and than reused. In the regime of theories factorization is basically the creation of a new theory out of two existing theories such that the new one can be interpreted in the two old ones. If we neglect translations and consider theories as sets of sentences the new generated theory could simply be thought of the intersection of two sets of sentences. The existence of translations, however, requires a refined notion of intersection. We will propose a notion of **theory intersection**, based on theory interpretation, in chapter 5 and present a theory factorization algorithm that, in a sense, yields the best approximation to a theory intersection.

By theory factorization, similar to code factorization, we extract parts of a theory and define this extract as a new theory on its own. Thus we can also take the perspective that a subset of sentences in the original theory constitutes already a theory on its own, which we call a **fragment of an explored theory** – or for short: **partial theory**. Theory interpretations are relations between (complete) theories, likewise we will introduce the term **partial theory interpretation**. The motivation for this concept is the observation that in practice we have already this view of partial theories in the following sense: many lemmata or theorems of a theory actually do not rely on all axioms of its home theory, but can be derived already by a subset of them. Hence such theorems can also be considered as part of the partial theory containing only those axioms necessary to prove the theorem. Moreover, we mostly do not derive theorems directly from axioms, but rather from intermediate lemmata or other theorems, which are finally grounded on the axioms of the theory. We could also view those lemmata and theorems as axioms of a partial theory from which we can derive our theorem. In general this partial theory view can be considered as preliminary phase for the birth of new theories: transforming theorems of an old theory into axioms of a new theory is probably the most usual process for the emergence of abstract theories. For us the intention of partial theory interpretation is to obtain a much denser network of interpretation as possible between regular theories. A denser interpretation network in turn yields a greater amount of theorem reuse. The basic application will be called **theory completion** (chapter 9), i.e. given the set of axioms of a given theory, compute all reusable theorems by using all possible partial theory interpretations in the knowledge base. In terms of model generation, this corresponds to a forward chaining procedure building the transitive closure of the derivability relation of the explicit theorem derivations in the knowledge base. In order to get the most exhaustive approximation of theory completion[3.8] the knowledge base had to store for each theorem the minimal set of its assumptions to prove it. This maxim corresponds very much to Hilbert's one maxim for axiomatization of a theory: "to choose ... a simple and complete set of independent axioms ... to bring out as clearly as possible the significance of the different groups of axioms and the scope of the conclusions to be derived from the individual axioms" (s. above).

3.8. Of course real theory completion, i.e. computation of all derivable theorems from given axioms, is neither possible (undecidability of expressive logics) nor desirable (there are always infinitely many in particular infinitely many trivial theorems).

Above we said automated search for theory interpretation is our principle theme, and we supplied some arguments what this is good for. But we have not said a word yet, how this search could be automated at all. Of course we ca not expect an algorithm that actually returns all possible theory interpretations for a given theory and a given theory library – the derivability or consequence relation is not computable. What we can expect, though, is an approximation. The search algorithm presented in this thesis is based on formula matching modulo an **equational theory**. This equational theory identifies for instance the following three semantically equivalent, but syntactically unmatchable formulae – all describing that f is a continuous function:

1. $\forall\varepsilon.\varepsilon>0\Rightarrow\exists\delta.\forall x.\forall y.0<|x-y|\wedge|x-y|<\delta\Rightarrow|f(x)-f(y)|<\varepsilon$
2. $\forall\varepsilon.\exists\delta.\forall x,y.\varepsilon>0\Rightarrow(0<|x-y|\wedge|x-y|<\delta\Rightarrow|f(x)-f(y)|<\varepsilon)$
3. $\forall\varepsilon.\exists\delta.\forall x,y.\varepsilon>0\wedge|x-y|<\delta\wedge0<|x-y|\Rightarrow|f(x)-f(y)|<\varepsilon$

This example should illustrate the freedom an author of a theory has to formalize a certain concept in general. Which variant the author finally takes is arbitrary. **Normalization** is the technique used in this work to identify formulae modulo this arbitrariness. Hereby normalization goes beyond usual conjunctive/disjunctive normalization as extensively used in automated theorem proving. In particular a formula **AC-standardization** is presented to identify formulae modulo associativity and commutativity. Also the technique of **formula abstraction**, i.e. the separation of formula's structures from its parameter, is used to obtain an very efficient filter to check whether two formulae are matching candidates, namely only if there structure is syntactically identical. All this is presented in chapter 8. Since these techniques are used for theory interpretation search in general, theory intersection and theory completion (both implicitly based on theory interpretation) benefit as well.

It should be mentioned that with the help of automated theorem provers certainly more theory interpretations could be found. On the other hand we would not have much control on the termination of such a search engine. Moreover, if we search for possible theory interpretation in large libraries, like Mizar with over 4.5 million formulae, the search response time with automated theorem provers would be unacceptable slow. The search algorithm presented in this work aims a practical compromise between semantic potential and responsiveness. Experimental results are discussed in chapter 11 and the system implementation described in chapter 10. Search efficiency is basically accomplished by that formula abstraction and the use of relational databases to store the theories. None of the above mentioned systems provide comparable search support.

3.8 Remark

One may object that theory interpretations found in this manner are relatively trivial from a mathematicians perspective. This is not surprising since normalization is essentially based on pure logical equivalence transformation – sophisticated proofs as mathematicians appreciate are not involved. However, this perspective neglects an important aspect of our original goal, namely to improve the accessibility of knowledge in large formalized libraries. Whether a theory inclusion is trivial or not from a mathematicians point of view is secondary if our goal is to expand our knowledge base. Moreover what is folklore to one mathematician in one research area is sometimes completely unknown to another mathematician from a different area and certainly to a mathematically interested layman too.

The strength of automated detection of theory inclusion via normalization is the ability of scanning masses of formulae. Mathematicians are unsurpassable in their dedicated field, but machines are good in precision and mass processing – they can discover useful things which are simply overlooked by humans.

Chapter 4
Theory Interpretation

Knowledge gain by **theorem reuse** is the principal purpose of **theory interpretation** and our goal is to find theory interpretations automatically. We want to start with the theoretical background: **signature morphism** and **entailment** are the fundamental proof theoretic concepts to define what a theory interpretation is. We want to start with Meseguer's definition of entailment systems [38] in a slightly modified presentation:

Definition 4.1. (Entailment System) An **entailment system** is a triple $\mathcal{E} = \langle \mathit{Sign}, \mathit{sen}, \vdash \rangle$ with *Sign* a category whose objects are called **signatures** and its morphisms **signature morphisms**, *sen* a functor from the category *Sign* to the category *Set*, and $\vdash$ a function associating to each Σ in *Sign* a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(\mathit{sen}(\Sigma)) \times \mathcal{P}(\mathit{sen}(\Sigma))$ called **Σ-entailment** such that the following properties are satisfied:

1. *reflexivity*: $\Gamma \vdash_\Sigma \Gamma$;
2. *monotonicity*: if $\Gamma \vdash_\Sigma \Delta$, $\Delta \subseteq \Delta'$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_\Sigma \Delta'$;
3. *union*: if $\Gamma \vdash_\Sigma \Delta_1$ and $\Gamma \vdash_\Sigma \Delta_2$ then $\Gamma \vdash_\Sigma \Delta_1 \cup \Delta_2$;
4. *transitivity*: if $\Gamma \vdash_\Sigma \Gamma'$ and $\Gamma' \vdash_\Sigma \Delta$ then $\Gamma \vdash_\Sigma \Delta$;
5. $\vdash$-*translation*: if $\Gamma \vdash_\Sigma \Delta$ and $\sigma: \Sigma \to \Sigma'$ then $\mathit{sen}(\sigma)(\Gamma) \vdash_{\Sigma'} \mathit{sen}(\sigma)(\Delta)$.

We call $\mathit{sen}(\Sigma)$ **sentences** and $\mathit{sen}(\sigma)$ a **sentence translation**.

In fact this is a cosmetic modification of the original definition where the codomain of the entailment relation is $\mathit{sen}(\Sigma)$ instead of $\mathcal{P}(\mathit{sen}(\Sigma))$. However, the correspondence is straightforward: $\Gamma \vdash_\Sigma \Delta \Leftrightarrow \forall \varphi \in \Delta . \Gamma \vdash'_\Sigma \varphi$ where $\vdash_\Sigma$ denotes our and $\vdash'_\Sigma$ Meseguer's entailment relation. Later we may use both versions whenever convenient.[4.1] At this stage we prefer $\mathcal{P}(\mathit{sen}(\Sigma))$ as codomain since thereby the *reflexivity*, *monotonicity*, and *transitivity* axioms have a rather natural shape and concatenation of our relation is not a problem.

In this definition the *Sign* category can be any arbitrary category and there is no internal structure beyond. This should be noted since one may associate, just by the name of *Sign*, some tacitly assumed internal structure. For our work, however, we require some internal structure of signatures that is satisfied by many formal languages. We want to motivate this in the following.

4.1. This kind of operator overloading is common practise – at least for functions: e.g. let $x \in M$; one often finds $f(x)$ as well as $f(M)$ though the latter has not been defined explicitly beforehand. We extend this overloading practise to relations: let $\sim$ be an arbitrary relation, let x and y be individuals. We may write $x \sim y$ as well as $\{x\} \sim y$, $x \sim \{y\}$, or $\{x\} \sim \{y\}$. The correspondence principle is always in analogy to that above for $\vdash_\Sigma$ and $\vdash'_\Sigma$.

In many cases a signature of a formal language is a structured hierarchy of tuples, lists, and sets. For instance, in the language definition for first-order logic signatures are typically divided into function and relation symbols and these symbols in turn are separated according to their arity. Such signatures conform to the scheme $\langle[\mathcal{F}_0, \mathcal{F}_1, ...], [\mathcal{R}_0, \mathcal{R}_1, ...]\rangle$ (where $\mathcal{F}_k$ is the set of k-ary function symbols and $\mathcal{R}_k$ is the set of k-ary relation symbols). For such kind of signatures there is a natural definition of intersection and union. We are not going to formalize this in the most general way, but we are content with the basic idea provided by this example: consider two signatures $\Sigma_i = \langle\mathcal{F}_i, \mathcal{R}_i\rangle$ $(i = 1, 2)$. We define the intersection as $\Sigma_1 \cap \Sigma_2 := \langle\mathcal{F}_1 \cap \mathcal{F}_2, \mathcal{R}_1 \cap \mathcal{R}_2\rangle$ and analogously we would define the union. The basic idea is to apply these operations point-wise on tuples or families of sets. To summarize at this point: many formal languages are equipped with a notion of signatures that allows for a natural definition of intersection and union. Since these set operations are essential for our automated theory interpretation search (as becomes clear later on) our signatures are constrained to have a notion of intersection and union. In fact we can consider our intended signatures even as flat sets of symbols together with a classification function that maps all its symbols into a set of tokens like *unary-function, binary-function, ..., unary-relation, binary-relation,* etc. Via this flattening we would encode the hierarchy information into the classification function. Thus we even do not lose generality when we simply assume that signatures are flat symbol sets. It is then up to concrete formal languages to define their specific classification functions. It should be noted that we do not claim to cover any possible notion of signature in this manner, but all those that are structured hierarchies of tuples, lists, and sets. These play an important role in many logics.

With this view of signatures as flat sets together with a symbol classification function, signature morphisms are simply functions between symbol sets that preserve the classification (meaning that n-ary function symbols are mapped to n-ary function symbols and the like for other kind of symbols). As usual for any function we adopt for signature morphisms the notion of "equality on common support" as follows:

$$f = g \text{ on } M \;:\Leftrightarrow\; \forall x \in M.\, f(x) = g(x) \qquad \text{for any } M \subseteq \mathrm{dom}(f) \cap \mathrm{dom}(g).$$

With signatures as flat sets and signature morphisms as functions between symbols (preserving symbol types) we have characterized our restricted notion of the *Sign* category[4.2].

Apart from that, however, we want also to constrain the *sen*-functor that maps the *Sign* category to the category *Set* of sentences. Again it should be mentioned that in the definition of entailment systems there is no constraint on the *sen*-functor. The purpose having a completely unconstrained *Sign* category as well as an unconstrained *sen*-functor is certainly to characterize a very general notion of entailment that covers all standard logics but also the most uncommon ones, too. The trade off of this generality, however, is that we can define very obscure systems satisfying all of the entailment system axioms. For instance, consider *sen* to be a functor that maps any signature to always the same singleton set and all signature morphisms to the only existing identity function in this category. All axioms would be satisfied. But obviously this system, containing only one sentence, is completely uninteresting.

4.2. It should be mentioned, though, that this simple notion of signature rules out many-sorted signatures, as well as order-sorted signatures, etc. where some symbols may refer to other symbols. However, this thesis concentrates on unsorted first-order logic, where our notion of signature is sufficient. In fact there is so much content formalised in first-order logics that it was an early research decision to concentrate on that logic.

Our constraint that we want to impose on the *sen*-functor is motivated by the idea that sentences are not linked to signatures by an arbitrary function, but that they are constructed out of the symbols of a signature. This is probably the case in most formal languages (unless there exists a strange notion of signature that does not even have symbols). Thus a reasonable purpose of the *sen*-functor (concerning the object mapping part of a functor[4.3]) is to map a signature to the set of all sentences constructable out of the symbols from that signature. In concrete formal languages this is described by construction rules (as we will do in chapter 6, too) and the *sen*-functor abstracts from the concrete rules, but just returns – so to speak – all the well-formed sentences over a signature. As we have argued above a completely unconstrained *sen*-functor can lead to absolutely useless notions of "well formed sentences". We want to motivate in the following how we constrained the *sen*-functor in a way that fits to our constrained notion of the *Sign* category: usually sentences of formal languages are constructed inductively over symbols from a signature and we can determine by that construction the symbols occurring in a sentence (in fact we will define the signature of a formula later on in this chapter). Apart from the object mapping of the *sen*-functor we also want to constrain the morphism mapping: in our context a signature morphism is simply a renaming of symbols and we associate with each renaming a sentence mapping that is determined by replacing simultaneously all the symbols of the input sentence by the according target symbol determined by our symbol renaming. This is the natural notion of sentence translation for formal languages which inductively construct their sentences from the symbols of their signature – in fact the translation is typically defined over the same inductive construction principles (as we will do in chapter 6, too). For such a notion of sentence translation holds that if two renamings are equal on their common support of symbols then the associated sentence translations are also equal on the sentences associated with that common support.

We now want to summarize our motivated constraints in a refinement of the entailment system axiomatization from above. First we introduce our notion of signature:

Definition 4.2. (Signature) Let T be a globally fixed set of types. A **simple signature** is a pair $\Sigma = \langle S, \pi\colon S \to T \rangle$ where S is a set of symbols and $\pi\colon S \to T$ the classification function assigning each symbol a type. Two signatures $\Sigma_i = \langle S_i, T_i, \pi_i\colon S_i \to T_i \rangle$ $(i = 1, 2)$ are called **compatible** if for all $s \in S_1 \cap S_2$ holds that $\pi_1(s) = \pi_2(s)$. A **simple signature morphism** $\sigma\colon \Sigma_1 \to \Sigma_2$ is a function $\sigma\colon S_1 \to S_2$ such that for all $s \in S_1$ holds $\pi_1(s) = \pi_2(\sigma(s))$. The **category of simple signatures** is the category whose objects are simple signatures and whose morphisms are simple signature morphisms.

Recall that in the definition 4.1 of entailment systems, the category *Sign* is an arbitrary category. The first difference of our definition of derivability system below is that we require a specific category for signatures. The second difference is that we add a constraint on the *sen* functor. We call our specific entailment system "derivability system" to foster an association with proof theory[4.4]:

Definition 4.3. (Derivability System) A **derivability system** is an entailment system $\mathcal{E} = \langle \mathit{Sign}, \mathit{sen}, \vdash \rangle$ where *Sign* is the category of simple signatures and where the *sen*-functor has to satisfy the axiom of ***∩-invariance***:

$$\text{if } \sigma = \tau \text{ on } \mathrm{dom}(\sigma) \cap \mathrm{dom}(\tau), \text{ then } \mathit{sen}(\sigma) = \mathit{sen}(\tau) \text{ on } \mathit{sen}(\mathrm{dom}(\sigma) \cap \mathrm{dom}(\tau))$$

4.3. Recall: a functor is a mapping between categories which means it maps objects to objects and morphisms to morphisms.

4.4. From the authors experience "entailment" is understood proof theoretically or model theoretically very much depending on the community of discourse.

We call $\vdash$ the **derivability relation**.

In the following we will assume an arbitrary but fixed derivability system whenever we make use of the notions defined in the above definition. Moreover, we will omit in the remainder the adjective "simple" for our signatures and their morphisms, as we only deal with simple signatures in this work[4.5].

We continue now with other useful definitions that we will use later on in this chapter:

Definition 4.4. (Signature of a Sentence) Let Σ be a signature and $\varphi \in sen(\Sigma)$, we define the **signature** of φ, denoted by Σ_φ, as the intersection of all Σ-compatible signatures Σ' for which $\varphi \in \mathrm{sen}(\Sigma')$. The signature of a set of sentences, all being members of the same $sen(\Sigma)$, is defined as the union of the signatures of each sentence in the set.

The signature of a formula should be considered simply as the set of symbols (without logical symbols and variables) occurring in the formula, since we consider signatures as symbol sets and formulae as inductively defined over symbols.

Next, we want to define the general notion of **inclusion** followed by a simple lemma about when the subset relation on the signature level propagates to the sentence level. This lemma already relies on the $\cap$*-invariance* axiom.

Definition 4.5. (Inclusion) An **inclusion** $f\colon A \hookrightarrow B$ is a function $f\colon A \to B$ with $f(a) = a$ for all $a \in A$. Accordingly, we define **signature inclusion** and **sentence inclusion**.

Lemma 4.6. For all signatures Σ, Σ', and all signature morphisms σ holds

1. If $\sigma\colon \Sigma \hookrightarrow \Sigma'$ is an inclusion, then $sen(\sigma)\colon sen(\Sigma) \hookrightarrow sen(\Sigma')$ is an inclusion.
2. $\Sigma \subseteq \Sigma' \quad$ implies $\quad sen(\Sigma) \subseteq sen(\Sigma')$

Proof.

1. Assume $\sigma\colon \Sigma \hookrightarrow \Sigma'$ is an inclusion, then $\Sigma \subseteq \Sigma'$ and hence $\sigma = \mathrm{id}_{\Sigma'}$ on $\Sigma \cap \mathrm{dom}(\mathrm{id}_{\Sigma'}) = \Sigma$. By the $\cap$-invariance of *sen* we know $sen(\sigma) = sen(\mathrm{id}_{\Sigma'}) = \mathrm{id}_{sen(\Sigma')}$ on $sen(\Sigma)$, which means that $sen(\sigma)\colon sen(\Sigma) \hookrightarrow sen(\Sigma')$ is an inclusion.
2. Assume $\Sigma \subseteq \Sigma'$ then there is an inclusion $\sigma\colon \Sigma \hookrightarrow \Sigma'$. By 1) we know that $sen(\sigma)$ must be an inclusion too. Hence, $sen(\sigma)(sen(\Sigma)) = sen(\Sigma) \subseteq sen(\Sigma')$. □

Now let us turn to those concepts that are at the center of this work:

Definition 4.7. (Theory) A **theory** is a pair $T = \langle \Sigma, \Gamma \rangle$ with Σ a signature and $\Gamma \subseteq sen(\Sigma)$. The members of Γ are called the **axioms** of T – also referred to as $\mathrm{Ax}(T)$ – and the members of the set $\mathrm{Sen}(T) := \{\varphi \in sen(\Sigma) | \Gamma \vdash_\Sigma \varphi\}$ are called **(valid) sentences** of T and $\mathrm{Th}(T) := \mathrm{Sen}(T) - \Gamma$ the **derived theorems**.

A **theory interpretation** $\sigma\colon (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a signature morphism $\sigma\colon \Sigma \to \Sigma'$, such that $\Gamma' \vdash_{\Sigma'} sen(\sigma)(\Gamma)$. We then say the **theory** $S := \langle \Sigma, T \rangle$ is **included in** $T = \langle \Sigma', \Gamma' \rangle$.

Due to the $\cap$-invariance of *sen* a subset relation on the signature level induces some subset relations on the sentence level involving the derivability relation.

4.5. In chapter 12 we mention a promising way how our simple notion of signature could be generalized – that needs some future research, though.

Lemma 4.8. Suppose signatures $\Sigma \subseteq \Sigma'$, sentences $\Gamma, \Delta \subseteq sen(\Sigma)$, and $\Gamma' \subseteq sen(\Sigma')$. We have

1. $\Gamma \vdash_\Sigma \Delta \Rightarrow \Gamma \vdash_{\Sigma'} \Delta$,
2. $\mathrm{Sen}(\langle \Sigma, \Gamma \rangle) \subseteq \mathrm{Sen}(\langle \Sigma', \Gamma \rangle)$,
3. $\sigma\colon (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a theory morphism if $\Gamma \subseteq \Gamma'$.

Proof. Assume $\Sigma \subseteq \Sigma'$, $\Gamma \vdash_\Sigma \Delta$, and $\sigma\colon \Sigma \hookrightarrow \Sigma'$.

1. By the translation property we have $sen(\sigma)(\Gamma) \vdash_{\Sigma'} sen(\sigma)(\Delta)$ and hence $\Gamma \vdash_{\Sigma'} \Delta$ since $sen(\sigma)$ is a sentence inclusion induced by the signature inclusion σ.
2. is an immediate consequence of 1.
3. We have to show $\Gamma' \vdash_{\Sigma'} \Gamma$. By reflexivity we have $\Gamma \vdash_\Sigma \Gamma$, (1) implies $\Gamma \vdash_{\Sigma'} \Gamma$. Finally the monotonicity property gives $\Gamma' \vdash_{\Sigma'} \Gamma$. □

Note that axioms are also theorems in this nomenclature. And $\mathrm{Sen}(\,\cdot\,)$ is actually a closure operator: $\mathrm{Sen}(\langle \Sigma, \Gamma \rangle)$ is the closure of Γ under the derivability relation $\vdash_\Sigma$. In informal statements we will sometimes identify a theory T with its closure $\mathrm{Sen}(T)$. This motivates the saying "theory T includes a theory S", since as closures the theories are actually in a subset relation – i.e. a (set) inclusion. The following proposition states this more precisely.

Proposition 4.9. Let $\sigma\colon \Sigma \to \Sigma'$ be a signature morphism, and $T = \langle \Sigma, \Gamma \rangle$, $S = \langle \Sigma', \Gamma' \rangle$ be theories. Then $\sigma\colon S \to T$ is a theory interpretation if and only if $\mathrm{Sen}(S) \supseteq sen(\sigma)(\Gamma)$.

Proof. Follows directly from the definitions. □

4.1 The Explored Part of a Theory

As we have already mentioned in section 2.3, we usually do not know every theorem of a theory, and certainly we usually cannot write them all down: although the axiomatization of a derivability system does not enforce theories to have infinitely many theorems, all non-trivial theories of reasonable logics have an infinite number of theorems. Otherwise a logic must not allow to derive from a formula φ a another formula containing φ as a proper subformula. Something like $\varphi \vdash \varphi \wedge \varphi$ must not be admissible in such a logic. However, in every useful logic, having a notion of "and", $\varphi \vdash \varphi \wedge \varphi$ holds. Necessarily, we cannot record explicitly all theorems $\varphi \wedge \varphi$, $\varphi \wedge \varphi \wedge \varphi$, ... following from φ; apart from that it would not make any sense to record them at all.

Although it is a triviality, we have to explicitly differentiate between a theory and that part we explicitly know in order to clarify how **theory interpretation** can lead to **knowledge gain** via **theorem reuse**. The actual knowledge we have about a theory are the proven theorems and we record them practically as formulae in our formal knowledge base.

Definition 4.10. (Explored Theory) Let $T = \langle \Sigma, \Gamma \rangle$ be a theory and let Δ be a set of formulae disjoint from Γ such that $\Gamma \vdash_\Sigma \Delta$ holds. We introduce the following terminology:

- The tuple $\langle \Sigma, \Gamma, \Delta \rangle$ is called the **proven fragment of a theory** $T = \langle \Sigma, \Gamma \rangle$ or for convenience just the **explored theory**.

- Δ are called the **explicitly known theorems** or **proven theorems** from T,
- an explored theory is called **sequent** if it contains only one proven theorem, and
- $\Gamma \cup \Delta$ are called the **known sentences** from T.

If we talk about knowledge gain in the context of theorems then we mean that we know more proven theorems than we have known before. In other words the proven fragment of a theory is extended, more precisely its proven theorems (since signature and axioms always remain the same – otherwise we would change the theory).

Definition 4.11. (Knowledge Gain) Let $T = \langle \Sigma, \Gamma, \Delta \rangle$ be a explored theory; let P be a proof that a set of formulae Δ' is derivable in T. We call $\Delta' - \Delta$ the **knowledge gain** due to P. Note: the knowledge gain can thus be also empty.

It might be debatable to call an "empty knowledge gain" a knowledge gain at all, but this definition turns out to be practicable in our context. In case of a non-empty knowledge gain we will speak of a **proper knowledge gain** – similar to the convention it set theory to distinguish between subsets and proper subsets.

Our primary goal is knowledge gain via theory interpretation. A theory interpretation by definition presupposes a proof, namely that the axioms of the source theory can be derived from the valid sentences of the target theory. In this sense theory interpretations can lead to (a possibly empty) knowledge gain.

Definition 4.12. (Theorem Reuse) Let $S = \langle \Sigma, \Gamma, \Delta \rangle$ and $T = \langle \Sigma', \Gamma', \Delta' \rangle$ be two explored theories, and $\sigma\colon S \to T$ a theory interpretation with $sen(\sigma)(\Gamma) \subseteq \Gamma' \cup \Delta'$. We call $sen(\sigma)(\Delta)$ the **reused theorems** from S. If a reused theorem is from a proper knowledge gain then we call it a **new theorem** for T from S via σ.

4.2 Automated Search for Theory Interpretations

Our main goal in this section is to find an algorithm that finds some theory interpretations between two theories if there exist any. We will describe the algorithm purely functionally (and in fact it is also implemented functionally – cf. chapter 10). No results depend on the finiteness of the number of theorems in the target theory, except for the termination of the algorithm, of course. Termination of the algorithm, though, is ensured for finite sets of theorems in the actual target theory as the computation steps are limited by the number of these theorems.

4.2.1 Algorithm Outline

Let us now start with a sketch how we get to the algorithm for automated theory interpretation search, whereby we introduce already all the important notations that will be properly defined in section 4.2.3:

The input of the envisioned algorithm is a source theory S and a target theory T and the output is a (possibly empty) set of theory interpretations. A theory interpretation σ from a source theory S to a target theory T exists if and only if all translated axioms of the source theory S are theorems in the target theory T. Assume we have a signature morphism σ with $\sigma(\varphi) \in \mathrm{Sen}(T)$ for all $\varphi \in \mathrm{Ax}(S)$, then we know that σ is a theory interpretation. Blindly guessing such a signature morphism and then testing

these memberships would not be promising of course. So we do it the other way round: we look at each axiom $\varphi \in \mathrm{Ax}(S)$ and try to match it against each theorem $\psi \in \mathrm{Sen}(T)$. Thus we build for each axiom φ a set $\overrightarrow{\varphi\mathrm{Sen}(T)}$ (cf. definition 4.4) of matching signature morphisms μ – i.e. $\mu(\varphi) \in \mathrm{Sen}(T)$. This gives us a family of sets of signature morphisms: $\mathcal{S} := \{\overrightarrow{\varphi\mathrm{Sen}(T)} | \varphi \in \mathrm{Ax}(S)\}$. Now let us take from each set of this family one member, and put them into a collection M, i.e. for all $\varphi \in \mathrm{Ax}(S)$ there is a $\mu \in M$ with $\mu(\varphi) \in \mathrm{Sen}(T)$. If all of them were pairwise compatible (cf. definition 4.14), i.e. any two $\mu, \mu' \in M$ were equal on their common domain (denoted by $\mu \| \mu'$) we could merge[4.6] (cf. definition 4.4) them to a single composed signature morphism $\sigma = \bigcup_{\mu \in M} \mu$. This composed σ would translate all our axioms in S to theorems from T. If our choice M taken from the family $\mathcal{S}$ does not combine to a single composed signature morphism, then we can take another choice from $\mathcal{S}$ and try again. Testing all choices by brute force is of course practically unfeasible, because of the combinatorial explosion. One could think of many heuristics how to narrow down the search space. We want to present a solution that is simple and has been proven efficient in practice (as will be discussed in chapter 11). Moreover, it finds not just one but all composable signature morphisms from $\mathcal{S}$. This approach is in a sense orthogonal to the one just described: we take a set X from the family $\mathcal{S}$ and a set X' from $\mathcal{S}' := \mathcal{S} - X$ and build a new set of signature morphisms (cf. definition 4.21) $Y := X \otimes X'$ which contains all composable signature morphisms from the catersian product $X \times X'$. Then we proceed recursively by taking another set X'' from the remainder $\mathcal{S}'' := \mathcal{S}' - X'$ and building signature morphisms $Y := Y \otimes X''$ until all members from $\mathcal{S}$ have been consumed. The final Y contains all composable signature morphisms from $\mathcal{S}$, i.e. all members of Y translate the axioms from the source theory S into theorems of the target theory T.

Though the basic idea is rather simple it needs some care to accommodate the algorithm rigorously to the formal definition of theory interpretations in derivability systems. So we have to consider, for instance, pathological cases where a signature morphism which translates all axioms of S into theorems of T which still is not a theory interpretation from S to T, because not every symbol from S's signature can be mapped to a symbol of T's signature. Such pathological cases can be easily constructed as we will see later on.

4.2.2 An Illustrative Example

An instructive example will help us to get a more concrete understanding of how the algorithm works, what the new operations on signature morphisms actually do, and what pathological cases can arise. We try to set up the examples as simple as possible and yet rich enough to illustrate the listed issues. The underlying formal language of our example is a very restricted version of predicate logic: we neither have quantifiers nor variables, but only function symbols of arity zero, one, or two. Hence, a signature of a theory is a triple $\langle \Sigma_1, \Sigma_2, \Sigma_3 \rangle$ of three disjoint sets of function symbols for each of these arities. Furthermore, we consider only signature morphisms that are simple symbol renamings (denoted by " $\rightsquigarrow$ "), and the sentence translations are inductively defined on the construction of formulae as already discussed above.[4.7]

4.6. Note, merging two signature morphisms, i.e. functions, means set theoretically nothing else than building the union of two sets of pairs. This union is a again function, i.e. a right unique binary relation, only if the functions are compatible: if they have a left value in common then their associated right value must be equal.

4.7. Note that also the kind of signature morphism and sentence translation is only an example. The algorithm is not restricted to that kind of morphisms!

Example 4.13. Let S be a source theory and T be target theory determined as in the table below.

S	T
$\Sigma_S = \langle\{a, b\}, \{P, Q\}, \{R\}\rangle$	$\Sigma_T = \langle\{c, d\}, \{R, P\}, \{Q\}\rangle$
$\mathrm{Ax}(S) = \{P(a), Q(a), R(a, b)\}$	$\mathrm{Sen}(T) = \{R(c), P(d), Q(d, d)\}$

Table 4.1. A source theory S and a target theory T.

We take the first axiom from S, i.e. $P(a)$, and look for all matching theorems in T and collect the corresponding signature morphisms in the set $\overrightarrow{P(a)\mathrm{Sen}(T)}$ and proceed analogously with the remaining axioms from S. Thus we get a family $\mathcal{S}$ of sets of partial signature morphisms:

- $\overrightarrow{P(a)\mathrm{Sen}(T)}$=$\{\{P \rightsquigarrow R,\, a \rightsquigarrow c\},\, \{P \rightsquigarrow P,\, a \rightsquigarrow d\}\}$
- $\overrightarrow{Q(a)\mathrm{Sen}(T)}$=$\{\{Q \rightsquigarrow R,\, a \rightsquigarrow c\},\, \{Q \rightsquigarrow P,\, a \rightsquigarrow d\}\}$
- $\overrightarrow{R(a,b)\mathrm{Sen}(T)}$=$\{\{R \rightsquigarrow Q,\, a \rightsquigarrow d,\, b \rightsquigarrow d\}\}$

Now we try to merge these signature morphisms. For instance we can merge $\{P \rightsquigarrow R,\, a \rightsquigarrow c\}$ with $\{Q \rightsquigarrow R,\, a \rightsquigarrow c\}$, because these signature morphisms are compatible, i.e. they have equal values on their common domain, namely $a \rightsquigarrow c$. All possible merges from the first two sets of signature morphisms give us:

$$\overrightarrow{P(a)\mathrm{Sen}(T)} \otimes \overrightarrow{Q(a)\mathrm{Sen}(T)} = \{\{P \rightsquigarrow R,\, Q \rightsquigarrow R,\, a \rightsquigarrow c\},\, \{P \rightsquigarrow P,\, Q \rightsquigarrow P,\, a \rightsquigarrow d\}\}$$

Finally we merge all signature morphisms from that set with the signature morphisms from the last set $\overrightarrow{R(a,b)\mathrm{Sen}(T)}$ of the family $\mathcal{S}$:

$$\left(\overrightarrow{P(a)\mathrm{Sen}(T)} \otimes \overrightarrow{Q(a)\mathrm{Sen}(T)}\right) \otimes \overrightarrow{R(a,b)\mathrm{Sen}(T)} = \{\{R \rightsquigarrow Q,\, P \rightsquigarrow P,\, Q \rightsquigarrow P,\, a \rightsquigarrow d,\, b \rightsquigarrow d\}\}$$

Hence we ended up with exactly one signature morphism that translates all axioms from S into theorems from T. Since it is also a mapping from Σ_S into Σ_T we have found with this signature morphism a theory interpretation – in fact the only existing one with respect to the actual theory T.

Now let us construct out of this result two pathological cases:

1. Assume we added to Σ_S another constant, say e. In order to become a theory interpretation, this surplus constant, which does not occur in any axiom, has to be mapped somewhere into Σ_T. Our procedure does not determine where to map it, but leaves a degree of freedom. Practically this is not a problem, because we can simply map e to a or b.

2. Assume our formal language also allowed symbols with arity three and we added such a set of arity three, say $\{L\}$, to Σ_S, but left an empty set for symbols of arity three to Σ_T. Although we would have found a sentence translation that translates axioms from S into theorems from T this signature morphism is not a theory interpretation from S to T, because the surplus symbol L would not be able to map any symbol into the target signature. However, we can extend T to T' with $\Sigma_{T'} := \Sigma_T \cup \{L\}$ such that we have at least a theory interpretation from S to T'.

4.2.3 Formal Development of the Algorithm

After we have seen the outline of the search algorithm and we have seen a concrete example, we want to proceed to the formal details. We start with a definition that makes our intuition of compatibility and merging precise, from which we can immediately derive useful properties.

Definition 4.14. (Combination of Compatible Morphisms) Two signature morphism $\sigma_i\colon \Sigma_i \to \Sigma_i'$ $(i = 1, 2)$ are called **compatible**, denoted by $\boldsymbol{\sigma_1 \| \sigma_2}$, iff $\sigma_1(s) = \sigma_2(s)$ for all $s \in \Sigma_1 \cap \Sigma_2$. Two sentence translations, theory translations, and theory morphisms respectively are called compatible iff their underlying signature morphisms are compatible.

For two compatible signature morphism $\sigma_i\colon \Sigma_i \to \Sigma_i'$ $(i = 1, 2)$ we define

- the union $\boldsymbol{\sigma_1 \cup \sigma_2}\colon \Sigma_1 \cup \Sigma_2 \to \Sigma_1' \cup \Sigma_2'$ by $(\sigma_1 \cup \sigma_2)(s) := \begin{cases} \sigma_1(s) \text{ if } s \in \mathrm{dom}(\sigma_1) \\ \sigma_2(s) \text{ otherwise} \end{cases}$ and
- the intersection $\boldsymbol{\sigma_1 \cap \sigma_2}\colon \Sigma_1 \cap \Sigma_2 \to \Sigma_1' \cup \Sigma_2'$ by $(\sigma_1 \cap \sigma_2)(s) := \sigma_1(s) = \sigma_2(s)$.

The union of signature morphisms preserves the compatibility in the following sense

Proposition 4.15. Let $\sigma_i\colon \Sigma_i \to \Sigma_i'$ $(i = 1, 2, 3)$ be signature morphisms.

1. if $\sigma_i \| \sigma_j$ for $i, j = 1, 2, 3$ then $\sigma_1 \cup \sigma_2 \| \sigma_3$.
2. if $\sigma_1 \| \sigma_2$ and $\sigma_1 \cup \sigma_2 \| \sigma_3$ then $\sigma_1 \| \sigma_3$ and $\sigma_2 \| \sigma_3$.

Proof. Straightforward from the definitions of union and compatibility. □

Hence a set of pairwise compatible signature morphisms is closed under union and intersection. This prepares the ground to a lattice structure whose properties we will use later on.

Lemma 4.16. Let $\mathcal{S}$ be a set of pairwise compatible signature morphisms. Every union and every intersection of compatible signature morphisms is again compatible with every other member of $\mathcal{S}$, i.e. $\mathcal{S}$ is closed with respect to $\cap$ and $\cup$. Moreover the structure $\langle \mathcal{S}, \cap, \cup \rangle$ is a lattice, i.e. for every $\sigma_1, \sigma_2, \sigma_3 \in \mathcal{S}$ holds

1. $\sigma_1 \cup (\sigma_2 \cup \sigma_3) = (\sigma_1 \cup \sigma_2) \cup \sigma_3$
2. $\sigma_1 \cap (\sigma_2 \cap \sigma_3) = (\sigma_1 \cap \sigma_2) \cap \sigma_3$
3. $\sigma_1 \cup \sigma_2 = \sigma_2 \cup \sigma_1$
4. $\sigma_1 \cap \sigma_2 = \sigma_2 \cap \sigma_1$
5. $\sigma_1 \cup (\sigma_1 \cap \sigma_2) = \sigma_1$
6. $\sigma_1 \cap (\sigma_1 \cup \sigma_2) = \sigma_1$

Proof. Assume compatible signature morphisms $\sigma_i\colon \Sigma_i \to \Sigma_i'$ for $i = 1, 2, 3$. The listed properties follow almost directly from the definition

1. $\sigma_1 \cup (\sigma_2 \cup \sigma_3)$ and $(\sigma_1 \cup \sigma_2) \cup \sigma_3$ is defined on $\Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ and $(\sigma_1 \cup (\sigma_2 \cup \sigma_3))(s) = ((\sigma_1 \cup \sigma_2) \cup \sigma_3)(s) = \sigma_i(s)$ if $s = \Sigma_i$ for $i = 1, 2, 3$.
2. $\sigma_1 \cap (\sigma_2 \cap \sigma_3)$ and $(\sigma_1 \cap \sigma_2) \cap \sigma_3$ is defined on $\Sigma_1 \cap \Sigma_2 \cap \Sigma_3$ and $(\sigma_1 \cap (\sigma_2 \cap \sigma_3))(s) = ((\sigma_1 \cap \sigma_2) \cap \sigma_3) = \sigma_1(s) = \sigma_2(s) = \sigma_3(s)$.

The commutativity properties (3) and (4) follow immediately. (5) $\sigma_1 \cup (\sigma_1 \cap \sigma_2)$ is defined on $\Sigma_1 \cup (\Sigma_1 \cap \Sigma_2) = \Sigma_1$ and hence $\sigma_1 \cup (\sigma_1 \cap \sigma_2)(s) = \sigma_1(s)$. A dual argument proves $\sigma_1 \cap (\sigma_1 \cup \sigma_2) = \sigma_1$. □

Due to the $\cap$-invariance axiom we can lift this lattice structure to the sentence level finally yielding a lattice of compatible theory interpretations.

Lemma 4.17. If σ_1: $(\Sigma_1, \Gamma_1) \to (\Sigma, \Gamma)$ and σ_2: $(\Sigma_2, \Gamma_2) \to (\Sigma, \Gamma)$ ar comepatible theory interpretations, then the signature morphism $(\sigma_1 \cup \sigma_2)$ induces a theory interpretation $(\Sigma_1 \cup \Sigma_2, \Gamma_1 \cup \Gamma_2) \to (\Sigma, \Gamma)$ and $(\sigma_1 \cap \sigma_2)$ induces a theory interpretation $(\Sigma_1 \cap \Sigma_2, \Gamma_1 \cap \Gamma_2) \to (\Sigma, \Gamma)$.

Proof. Assume σ_1: $(\Sigma_1, \Gamma_1) \to (\Sigma, \Gamma)$ and σ_2: $(\Sigma_2, \Gamma_2) \to (\Sigma, \Gamma)$ are compatible theory interpretations. By definition the signature morphisms σ_i: $\Sigma_i \to \Sigma_i'$ $(i = 1, 2)$ are compatible and thus $\sigma_1 = \sigma_1 \cup \sigma_2$ on Σ_1. From the $\cap$-invariance of *sen* follows $sen(\sigma_1) = sen(\sigma_1 \cup \sigma_2)$ on $sen(\Sigma_1)$ and in particular $sen(\sigma_1)(\Gamma_1) = sen(\sigma_1 \cup \sigma_2)(\Gamma_1)$ for the set of sentences $\Gamma_1 \subseteq sen(\Sigma)$. Since σ_1: $(\Sigma_1, \Gamma_1) \to (\Sigma, \Gamma)$ is assumed to be a theory interpretation, i.e. $\Gamma \vdash_\Sigma \sigma_1(\Gamma_1)$, we know now that $\Gamma \vdash_\Sigma sen(\sigma_1 \cup \sigma_2)(\Gamma_1)$. Analogously we derive $\Gamma \vdash_\Sigma sen(\sigma_1 \cup \sigma_2)(\Gamma_2)$ and hence conclude with $\Gamma \vdash_\Sigma sen(\sigma_1 \cup \sigma_2)(\Gamma_1 \cup \Gamma_2)$.

The intersection case follows from similar arguments. □

Definition 4.18. (Combination of Theory Interpretations) We define for two compatible theory interpretations σ_1: $(\Sigma_1, \Gamma_1) \to (\Sigma, \Gamma)$ and σ_2: $(\Sigma_2, \Gamma_2) \to (\Sigma, \Gamma)$ the **union of theory interpretations** as resulting theory interpretation $(\Sigma_1 \cup \Sigma_2, \Gamma_1 \cup \Gamma_2) \to (\Sigma, \Gamma)$ denoted by $\sigma_1 \cup \sigma_2$. Similarly we define the **intersection of theory interpretations**.

Lemma 4.19. Given a set of compatible theory interpretations $\mathcal{S}$ into some fixed target theory, the structure $\langle \mathcal{S}, \cap, \cup \rangle$ forms a lattice.

Proof. From lemma 4.17 we know that a set of compatible theory interpretations is closed with respect to $\cap$ and $\cup$. As compatible theory interpretations are in particular compatible signature morphisms they, share the lattice properties. □

Given two formulae φ and ψ, we can ask whether there is translation between them. Assume we have found σ: $\Sigma \to \Sigma'$ with $\sigma(\varphi) = \psi$. Then by the $\cap$-invariance of *sen* we know that any arbitrary extension of σ would also map φ to ψ. What we want, however, is the smallest signature morphism, with respect to its domain, that translates φ to ψ. Hence, we define for every formula the smallest signature that still contains the formula. In our example, with its particular *sen* functor, the minimal signature for a formula is simply computed by collecting all its symbols. Minimal translations for a formula are those whose domain is the minimal signature of this formula.

Definition 4.20. (Minimal Translation)

Let Σ and Σ' be signatures; let $\varphi \in sen(\Sigma)$ and $\psi \in sen(\Sigma')$ be formulae. We call μ: $\varphi \to \psi$ a **minimal formula translation** iff μ: $\Sigma_\varphi \to \Sigma'$ is a signature morphism with $sen(\mu)(\varphi) = \psi$.

Let G be a set of formulae. The set of **minimal translations from φ to G** is defined as

$$\begin{aligned} \overrightarrow{\varphi G} &:= \{\mu \,\text{minimal formula translation} | \varphi \in F, \mu(\varphi) \in G\} \\ \overrightarrow{F G} &:= \bigcup_{\varphi \in F} \overrightarrow{\varphi G} \end{aligned}$$

Note, any signature morphism σ: $\Sigma \to \Sigma'$ becomes a minimal formula translation σ: $\varphi \to sen(\sigma)(\varphi)$ just by restricting σ to $\sigma|_{\Sigma_\varphi}$. As we need such restrictions of signature morphisms we sometimes abbreviate $\sigma|_{\Sigma_\varphi}$ with σ_φ.

Now we define the operation that returns all possible signature morphism merges from two sets of signature morphisms and derive important properties afterwards.

Definition 4.21. (Product of Compatible Morphisms) Let X, X_1, ..., X_n, and Y be sets of signature morphisms. We define the **product of sets of signature morphisms** as:

$$\begin{aligned} X \otimes Y &:= \{\sigma \cup \tau \,|\, \sigma \in X, \tau \in Y, \sigma \| \tau\} \\ \bigotimes_{i=1}^{n} X &:= X_1 \otimes ... \otimes X_n. \end{aligned}$$

Proposition 4.22. The product of sets of signature morphisms is 1) commutative and 2) associative.

Proof.

1. Commutativity directly follows from the commutativity of the union operator and the symmetry of the compatibility relation.
2. Assume $\pi \in X \otimes (Y \otimes Z)$. Then there must be signature morphisms $\sigma \in X$, $\rho \in Y$, and $\tau \in Z$ such that $\pi = \sigma \cup (\rho \cup \tau)$ and
   1. $\sigma \cup (\rho \cup \tau) \in X \otimes (Y \otimes Z)$ and $\sigma \| (\rho \cup \tau)$,
   2. $(\rho \cup \tau) \in (Y \otimes Z)$ and $\rho \| \tau$.

   From $\sigma \| (\rho \cup \tau)$ we can infer $\sigma \cup \rho \| \tau$ and $\sigma \| \rho$ by proposition 4.15. From $\sigma \in X$, $\rho \in Y$, and $\sigma \| \rho$ we can infer $\sigma \cup \rho \in X \otimes Y$. From $\tau \in Z$, $\sigma \cup \rho \in X \otimes Y$, and $\sigma \cup \rho \| \tau$ we know $(\sigma \cup \rho) \cup \tau \in (X \otimes Y) \otimes Z$. Finally, with the associativity of the union operator, we have $\sigma \cup (\rho \cup \tau) \in (X \otimes Y) \otimes Z$. □

With this proposition we know for our algorithm that it does not matter in which order we merge the sets of signature morphisms. The next two lemmata prepare the main theorem for the algorithm.

Lemma 4.23. Let Σ be a non-empty signature and $\sigma: \Sigma \to \Sigma'$ be a signature morphism; let $F \subseteq sen(\Sigma)$ and $G \subseteq sen(\Sigma')$ be non-empty sets of formulae and F be finite. If $G \supseteq sen(\sigma)(F)$ then:

$$\bigcup_{\varphi \in F} \sigma_\varphi \in \bigotimes_{\varphi \in F} \overrightarrow{\varphi G}$$

Proof. First, we note that $\bigcup_{\varphi \in F} (\sigma_\varphi)$ is defined since for any σ with $\mathrm{dom}(\sigma) = \Sigma$ and $\varphi, \psi \in sen(\Sigma)$ we know by definition that $\sigma_\varphi \| \sigma_\psi$.

Let $F = \{\varphi_1, ..., \varphi_n\}$, $\sigma_k := \sigma_{\varphi_1} \cup ... \cup \sigma_{\varphi_k}$ and $X_k := \overrightarrow{\varphi_1 G} \otimes ... \otimes \overrightarrow{\varphi_k G}$. Since the union of signature morphisms and the product of sets of signature morphisms are commutative and associative, the element relation of the lemma holds iff $\sigma_n \in X_n$ holds, since the order of building the union and the product, respectively, does not matter. Hence we prove $\sigma_n \in X_n$ by induction: we know that $\sigma_\varphi \in \overrightarrow{\varphi G}$ for any $\varphi \in F$ since $\varphi \in F \Rightarrow sen(\sigma)(\varphi) \in G$ by $G \supseteq sen(\sigma)(F)$ and $sen(\sigma)(\varphi) \in G \Leftrightarrow \sigma_\varphi \in \overrightarrow{\varphi G}$ by $\cap$-invariance of sen. Thus we have shown in particular the basis step $\sigma_\varphi \in \overrightarrow{\varphi_1 G}$. Now assume $\sigma_{k-1} \in X_{k-1}$ holds for $k > 1$. Analogously to what is said at the beginning of this proof we know that $\sigma_{k-1} \| \sigma_{\varphi_k}$ and $\sigma_{k-1} \cup \sigma_{\varphi_k}$ is defined. Again we know $\sigma_{\varphi_k} \in \overrightarrow{\varphi_k G}$. Together with $\sigma_{k-1} \in X_{k-1}$ and $\sigma_{k-1} \| (\sigma_{\varphi_k})$ we can finally infer by definition of the product $\sigma_{k-1} \cup \sigma_{\varphi_k} \in X_{k-1} \otimes \overrightarrow{\varphi_k G}$ – i.e. $\sigma_k \in X_k$. □

As mentioned as pathological case in the example we may need to extend our union of minimal signature morphisms in order to map all symbols from the source signature to the target signature, but not just those needed for the construction of the axioms in the source theory. Therefore we define the extension of signature morphisms and relate them to our unions of minimal signature morphisms.

Definition 4.24. Let $\sigma_i : \Sigma_i \to \Sigma_i'$ be a signature morphisms with $\sigma_1 \| \sigma_2$ and $\Sigma_1' \supset \Sigma_2'$ for $i = 1, 2$. We call σ_1 an **extension of** σ_2, and say σ_2 can be **extended** to σ_1.

Lemma 4.25. Let Σ be a non-empty signature; let $F \subseteq sen(\Sigma)$ and $G \subseteq sen(\Sigma')$ be non empty sets of formulae and F be finite. For any signature morphism $\sigma : \Sigma \to \Sigma'$ holds

$$G \supseteq sen(\sigma)(F) \text{ iff there is a } \tau \in \bigotimes_{\varphi \in F} \overrightarrow{\varphi G} \text{ that can be extended to } \sigma.$$

Proof.
$\Rightarrow$: Let $\tau := \bigcup_{\varphi \in F} (\sigma|_{\Sigma_\varphi})$. From lemma 4.23 it follows that $\tau \in \bigotimes_{\varphi \in F} \overrightarrow{\varphi G}$ which can be extended to σ.
$\Leftarrow$: Assume $\tau \in \bigotimes_{\varphi \in F} \overrightarrow{\varphi G}$, then by definition follows $sen(\tau)(\varphi) \in G$ for any $\varphi \in F$. Assume now τ can be extended to σ, then $sen(\sigma)(\varphi) \in G$ by the $\cap$-invariance of sen. □

Theorem 4.26. Let $S = \langle \Sigma, \Gamma \rangle$, $T = \langle \Sigma', \Gamma' \rangle$ be theories. Every signature morphism σ: $\Sigma \to \Sigma'$ is a theory interpretation σ: $S \to T$ if and only if it is an extension of some member of $\bigotimes_{\varphi \in \Gamma} \overrightarrow{\varphi \mathrm{Sen}(T)}$.

Proof. It follows directly from the definition that σ: $S \to T$ is a theory interpretation if and only if $\mathrm{Sen}(T) \supseteq sen(\sigma)(\Gamma)$. Lemma 4.25 states $\mathrm{Sen}(T) \supseteq sen(\sigma)(\Gamma)$ if and only if there is a $\tau \in \bigotimes_{\varphi \in \Gamma} \overrightarrow{\varphi \mathrm{Sen}(T)}$ that can be extended to σ. □

Our algorithm that should find all the theory interpretations from the source theory to the target theory is given by $\bigotimes_{\varphi \in \mathrm{Ax}(S)} \overrightarrow{\varphi \mathrm{Sen}(T)}$. In fact, as mentioned above, these signature morphisms are restricted versions of theory interpretations: they still leave a degree of freedom to map the symbols outside the minimal signature of $\mathrm{Ax}(S)$ to arbitrary symbols in the target theory.

A concrete implementation in the functional programming language Haskell of theory interpretation for explored theories will be presented in section 10.1.6.

Chapter 5
Theory Intersection

Let us recall from chapter 3 about theory graphs the two basic organization principles for mathematical theories: theory import and theory inclusion. With theory imports we collect axioms from various source theories – possibly translated by some signature morphism – in a target theory. Semantically this means that all these collected axioms (together with the axioms explicitly stated in the target theory) are valid by definition inside the target theory. In contrast to theory import, a theory inclusion means the assertion that all these collected and translated axioms from the source theory are not necessarily axioms in the target theory, but are derivable from them.

Knowledge gain is the fundamental motivation for the application of both of these two organization principles: whenever we import source theories into a target theory or we have proven the inclusion of source theories, we know that their source theorems hold in the target theory, i.e. we can reuse the theorems from the source theories. Thus a graph of theories, i.e. theories linked with imports and inclusions, is a network of knowledge reuse. The more links there are between theories, the higher the potential of knowledge reuse. Automated theory inclusion search as presented in the previous section is only one possible support to increase number of links between theories. In this section we present a method which we called theory factorization in section 3.7.

Let us now imagine how new theories emerge in formal libraries. We could create a new theory T from scratch. For knowledge reuse we try to integrate this new theory into the already existing theory graph, i.e. we search for other theories that might serve as source theories for theory imports or theory inclusions. If T has many axioms, it is very likely that there are some source theories with fewer axioms which could be imported; whereas it is very unlikely to find many source theories – if any at all – including T. If T has only few axioms, it is rather the other way round. Informally spoken with the increase of axioms in T, we increase the chance of theory imports and decrease the chance of being included by some source theories. On the other hand, although the increase of number of axioms in T decreases the chance of being included it simultaneously increases the chance that at least a part of T's axioms are included in some source theory S. In other words, the chance of knowledge overlap between T and some source theories increases.

More generally, we intuitively understand knowledge overlap between two theories as a set of formulae these theories have in common. It is the purpose of this section to develop a precise meaning of such a knowledge overlap – or **theory intersection**, as we will call it.

We will start with an example to point out the difficulty of the notion of theory intersection even though we have a direct intuitive understanding. Our example is contrived in the sense that it does not describe a real historical discovery of knowledge overlap. But the claim is that knowledge overlaps of theories have been discovered in history by that principle. Our example is about variants of the ring theory. We list the axioms of ring theory and we give each axiom a name with parameters – i.e. the constant/function symbols occurring in the respective axiom. We will use these names to refer to formulae instead of writing the formulae themselves. Different parametrization denote thus different formulae with the obvious meaning.

Example 5.1. The theory of rings $\mathcal{R}$ is constituted by the axioms:

Name:	Formula:
com(+)	$\forall x, y.\ x + y = y + x$
ass(+)	$\forall x, y, z.\ x + (y + z) = (x + y) + z$
ass($*$)	$\forall x, y, z.\ x * (y * z) = (x * y) * z$
dist($*$, +)	$\forall x, y, z.\ x * (y + z) = x * y + x * z$
inv(+, 0)	$\forall x. \exists y.\ x + y = 0$
neut(+, 0)	$\forall x.\ x + 0 = x$

Based on the ring theory $\mathcal{R}$ there are two well known extensions:

1. Abelian ring theory $\mathcal{R}_a$ which we get by adding com($*$) as axiom to $\mathcal{R}$,
2. Ring theory with unit $\mathcal{R}_1$ which we get by adding neut($*$, 1) as axiom to $\mathcal{R}$.

So, given $\mathcal{R}$, we can construct $\mathcal{R}_a$ and $\mathcal{R}_1$ by theory import of $\mathcal{R}$. But let us assume the opposite initial situation, i.e. we have $\mathcal{R}_a$ in our theory graph but not $\mathcal{R}$ and we want to integrate $\mathcal{R}_1$ as a "new" theory into this theory graph. There is no chance to have a theory import or inclusion relation with $\mathcal{R}_1$. However, if we take the intersection of the sets of axioms of $\mathcal{R}_a$ and $\mathcal{R}_1$ we get $\mathcal{R}$ and we have the theory import relations as mentioned above. So in this particular case we achieve knowledge reuse with a straightforward notion of theory intersection, namely as intersection of sets of axioms. However, this notion is too naive as it succeeds in a sense only accidentally, as the following example demonstrates: assume we have $\mathcal{R}_a$ as before, i.e. its axioms are:

$$\mathrm{Ax}(\mathcal{R}_a)=\{\mathrm{com}(+), \mathrm{com}(*), \mathrm{ass}(+), \mathrm{ass}(*), \mathrm{dist}(*,+), \mathrm{inv}(+,0), \mathrm{neut}(+,0)\}$$

But instead of $\mathcal{R}_1$ as given by

$$\mathrm{Ax}(\mathcal{R}_1)=\{\mathrm{com}(+), \mathrm{ass}(+), \mathrm{ass}(*), \mathrm{dist}(*,+), \mathrm{inv}(+,0), \mathrm{neut}(+,0), \mathrm{neut}(*,1)\}$$

we now consider a variant $\mathcal{R}_1'$ whose axioms are

$$\mathrm{Ax}(\mathcal{R}_1')=\{\mathrm{com}(\oplus), \mathrm{ass}(\oplus), \mathrm{ass}(\otimes), \mathrm{dist}(\otimes,\oplus), \mathrm{inv}(\oplus, n), \mathrm{neut}(\oplus, n), \mathrm{neut}(\otimes, e)\}.$$

Obviously, we no longer obtain $\mathcal{R}$ with the naive intersection, i.e. $\mathrm{Ax}(\mathcal{R}_a) \cap \mathrm{Ax}(\mathcal{R}_1') = \varnothing$. This is unsatisfactory, since $\mathcal{R}_1$ and $\mathcal{R}_1'$ are isomorphic. Of course we can easily overcome this problem in this particular setting by applying the "right" signature morphism $\sigma = [\oplus \rightsquigarrow +, \otimes \rightsquigarrow *, n \rightsquigarrow 0, e \rightsquigarrow 1]$. We then have $\mathrm{Ax}(\mathcal{R}_1)=\mathrm{Ax}(\sigma\mathcal{R}_1')$ and hence $\mathrm{Ax}(\mathcal{R}) = \mathrm{Ax}(\mathcal{R}_a) \cap \mathrm{Ax}(\mathcal{R}_1) = \mathrm{Ax}(\mathcal{R}_a) \cap \mathrm{Ax}(\sigma\mathcal{R}_1')$. But how do we know what the "right" signature morphism is? In our example, σ seems to be the "right" signature morphism just by looking at the axioms of $\mathcal{R}_1$ and $\mathcal{R}_1'$. However, consider another variant $\mathcal{R}_1''$ with

$$\mathrm{Ax}(\mathcal{R}_1'')=\{\mathrm{com}(\oplus), \mathrm{ass}(\oplus), \mathrm{ass}(\otimes), \mathrm{dist}(\oplus,\otimes), \mathrm{inv}(\oplus, n), \mathrm{neut}(\oplus, n), \mathrm{neut}(\otimes, e)\}.$$

$\mathcal{R}_1$ and $\mathcal{R}_1''$ differ only in the axiom dist($\oplus$, $\otimes$) where the parameters are swapped. Applying σ from above on $\mathcal{R}_a''$ and building the intersection $\mathrm{Ax}(\mathcal{R}_a) \cap \mathrm{Ax}(\sigma\mathcal{R}_1'')$ yields

$$\{\mathrm{com}(+),\ \mathrm{ass}(+),\ \mathrm{ass}(*),\ \mathrm{inv}(+,0),\ \mathrm{neut}(+,0)\}.$$

So σ does not seem to be the "right" signature morphism for this intersection, as dist($*$, +) is missing here. Maybe we get the "right" one by swapping $\oplus$ and $\otimes$ in σ as we did in dist($\oplus$, $\otimes$), so let us build $\mathrm{Ax}(\mathcal{R}_a) \cap \mathrm{Ax}(\sigma'\mathcal{R}_1'')$ with $\sigma' = [\otimes \rightsquigarrow +, \oplus \rightsquigarrow *, n \rightsquigarrow 0, e \rightsquigarrow 1]$

$$\{\mathrm{com}(+), \mathrm{ass}(+), \mathrm{ass}(*), \mathrm{dist}(*,+)\}.$$

We got a different and even smaller subset of $\mathrm{Ax}(\mathcal{R})$. Naturally we want to find a signature morphism that leads to the biggest possible intersection. Finding the "right" signature morphism in this sense is not that obvious here. In fact $\tau = [\otimes \rightsquigarrow +, \oplus \rightsquigarrow *, n \rightsquigarrow 1, e \rightsquigarrow 0]$ is such a signature morphism:

$$\mathrm{Ax}(\mathcal{R}_a) \cap \mathrm{Ax}(\tau\mathcal{R}_1'') = \{\mathrm{com}(*),\ \mathrm{ass}(+),\ \mathrm{ass}(*), \mathrm{dist}(*,+),\ \mathrm{inv}(+,0),\ \mathrm{neut}(+,0)\}$$

This example already suggests that there are more or less appropriate signature morphisms to build an intersection the way we did. τ is clearly better than σ and σ', since τ yields an intersection between $\mathcal{R}_a$ and $\mathcal{R}_1''$ that has more elements than both of the other intersections resulting from σ and σ'.

In general, given two sets of formulae F and G, we are certainly most interested in a signature morphism that makes $\sigma F \cap G$ as large as possible. However, the signature morphism σ thus is not determined uniquely in all cases as the example below demonstrates. For this and the following examples in this section let a, b, c, ... be constants, and let $\varphi(...)$ and $\psi(...)$ be formula names with any number of parameters where $\varphi(...) \neq \psi(...)$, independently of their concrete parameters.

Example 5.2. Suppose we have two formula sets $F = \{\varphi(a, a, b), \psi(a, b, b)\}$ and $G = \{\varphi(c, c, d), \psi(d, c, c)\}$. Then there exist two maximal signature morphisms:

- $\sigma_1 := [a \rightsquigarrow c, b \rightsquigarrow d] \quad \Rightarrow \quad \sigma_1 F \cap G = \{\varphi(c, c, d)\}$
- $\sigma_2 := [a \rightsquigarrow d, b \rightsquigarrow c] \quad \Rightarrow \quad \sigma_2 F \cap G = \{\psi(d, c, c)\}$

Moreover we were investigating only signature morphisms applied on one of the two formula sets. Which of them to take was an unmotivated choice. In example 5.2 the signature morphisms σ_1 and σ_2 are bijective. Thus we observe that $F \cap \sigma_1^{-1}G$ and $F \cap \sigma_2^{-1}G$ are the isomorphic counterparts of $\sigma_1 F \cap G$ and $\sigma_2 F \cap G$ respectively, since we have

$$\sigma_i F \cap G = \sigma_i \circ \sigma_i^{-1}(\sigma_i F \cap G) = \sigma_i(F \cap \sigma_i^{-1}G) \qquad \text{for } i = 1, 2.$$

In case of non-bijective signature morphisms, however, there is no isomorphic counterpart $F \cap \sigma' G$ to $\sigma F \cap G$, as example 5.3 demonstrates:

Example 5.3. Suppose we have two formula sets $F = \{\varphi(a, a), \psi(b, c)\}$ and $G = \{\varphi(a, b), \psi(c, c)\}$ then we have two directed intersections:

- $\sigma_1 := [a \rightsquigarrow a, b \rightsquigarrow c, c \rightsquigarrow c] \quad \Rightarrow \quad \sigma_1 F \cap G = \{\psi(c, c)\}$
- $\sigma_2 := [a \rightsquigarrow a, b \rightsquigarrow a, c \rightsquigarrow c] \quad \Rightarrow \quad F \cap \sigma_2 G = \{\varphi(a, a)\}$

We are not interested in those intersections, though, for the following reason: our intention with the notion "theory *intersection*" is something related to the notion of intersection in set theory. Let us build a correspondence for theory intersection of two theories T_1 and T_2. We have seen above that the mere intersection of all their sentences is an unsatisfactory definition for theory intersection. We gave arguments that some signature must be involved which maximizes the theory intersection in some sense. In example 5.3 every injective signature morphism would yield an empty intersection of formulae, whereas the two given non-injective signature morphisms both return non-empty intersection of formulae. Hence, non-injective signature morphisms could be considered as appropriate for constructing a theory intersection. But moreover our theory intersection should also correspond to this basic property from set theory: for all sets M and N holds $M \cap N \subseteq M$ and $M \cap N \subseteq N$.

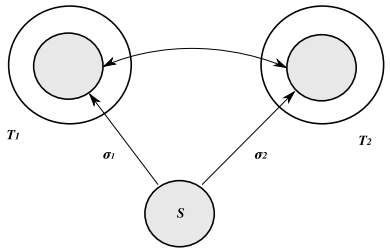

Figure 5.1. Theory intersection

The natural correspondence is to require that a theory intersection S of two theories T_1 and T_2 should be included in both of them, i.e. there should be two theory interpretations $\sigma_i : S \to T_i$ for $i = 1, 2$. Our example indicates a violation of this correspondence since we cannot translate $\sigma_1 F \cap G$ into F and we cannot translate $F \cap \sigma_2 G$ into G, because of the non-injectivity of σ_1 and σ_2. Hence, if our construction of a theory intersection has the form $\sigma T_1 \cap T_2$, then σ should be injective, which guarantees that $\sigma T_1 \cap T_2$ is included in both T_1 and T_2.

Upon these motivations we come to these definitions:

Definition 5.4. (Intersection of Formula Sets) Let F and G be sets of formulae and let σ be a bijective signature morphism defined on Σ_F. We call the set $sen(\sigma)(F) \cap G$ a **maximum intersection** of F and G, and σ an **optimally intersecting** signature morphism w.r.t. F and G iff there is no other bijective signature morphism τ defined on Σ_F such that $sen(\tau)(F) \cap G$ is a proper superset of $sen(\sigma)(F) \cap G$.

By maximum intersection we get almost directly to our intended notion of theory intersection: we simply consider F and G as the theorems from theory T_1 and T_2 respectively, then our maximum intersection $sen(\sigma)(F) \cap G$ is a set of all theorems shared by T_1 and T_2. Apart from that, however, $sen(\sigma)(F) \cap G$ should be also closed under derivability, otherwise we could not consider this set of formulae as the complete set of theorems of an appropriate theory. That is the goal of the following lemma.

Lemma 5.5. If the formula sets F and G are closed under derivability then $F \cap G$ is closed under derivability, too.

Proof. A set F is closed under derivability iff $F \vdash \varphi$ implies $\varphi \in F$. Assume $F \vdash \varphi \Rightarrow \varphi \in F$, $G \vdash \varphi \Rightarrow \varphi \in G$, and $F \cap G \vdash \varphi$. We infer from $F \cap G \vdash \varphi$ and the monotonicity axiom $F \vdash \varphi$ and $G \vdash \varphi$. Hence $\varphi \in F \cap G$. □

So for theories T_1 and T_2 the intersection $\mathrm{Sen}(T_1) \cap \mathrm{Sen}(T_2)$ is closed under derivability and can thus be considered as the set of all theorems of some theory. Since we have defined a theory as a pair of signature and axioms this intersection does not give us a theory in that strict sense. But this is not a problem as we always know for any formula set closed under derivability at least one set of axioms that generates these theorems: the set itself. We will always consider a set closed under derivability as a theory with the set itself as its fallback axioms.

Definition 5.6. (Theory Intersection) Let T_1 and T_2 be theories and σ be a bijective signature morphism defined on the signature of T_1. We call a theory S a **theory intersection** of T_1 and T_2 if and only if $S = sen(\sigma)(\mathrm{Sen}(T_1)) \cap \mathrm{Sen}(T_2)$ is a maximum intersection.

5.1 Algorithm for Theory Intersection Search

After a general definition of theory intersection is given, it would be nice to know how to find them in real formal libraries. First of all we have to recall that we have explored theories in formal libraries (cf. section 4.2). Similarly to the algorithm for theory interpretations our search of theory intersection is based on the actual given knowledge in the formal library; i.e. we do not consider all theorems (potentially) derivable from the given axioms of a theory, but only those actually derived. Hence, our intersections will be mostly smaller than possible as there are always more theorems to derive in every theory, which may enlarge our actual intersection again. Consider the extreme example where two disjoint sets of formulae axiomatize the same theory (e.g. topology can be axiomatized either by a system of neighborhoods or open sets). In a formal library at a certain stage there might be two actual theories solely consisting of these axioms. Our intersection algorithm would find the empty intersection, although the real intersection is the whole theory itself. Thus the claim of our algorithm is restricted to find a maximum intersection of formulae explicitly given in actual theories.

5.1.1 Algorithm Outline

Important techniques for the theory intersection search are taken from theory interpretation search, since the tasks are partly the same:

1. build all minimal translations,
2. look for compatibility of minimal translations,
3. merge them,
4. find the (merged) signature that yields the maximum intersection.

Unlike for theory interpretation search we are now considering only bijective signature morphisms, which therefore needs a slightly adapted notion of compatibility. The main difference, however, is the last item: For the theory interpretation search we know beforehand that every axiom of the source theory must have its matching theorem in the target theory, otherwise there cannot be a theory interpretation. So whenever we find a single source axiom that cannot be matched to some target theorem, we can stop our algorithm. When searching for a maximal intersection we do not have that simple yes/no criterion, but a much larger search space.

The basic idea to handle this search as efficiently as possible is to map it on a known problem that is already well studied. Such a mapping is presented here, notably to the maximum clique problem – known from graph theory: given a graph, where the vertices represent persons and the edges friendships, the task is to find a maximum clique, i.e. a maximum community of persons where all are friends of each other. In our algorithm the vertices represent minimal translations associated with their formulae and the edges compatibility. The union of a maximum collection of a minimal translations gives us a maximum intersection.

5.1.2 Maximum Intersection as Maximum Clique Problem

As mentioned above we have to refine the notion of compatibility to bijective signature morphisms.

Definition 5.7. (Bijective Compatibility) Let σ and τ be two bijective signature morphisms. We call them **bijectively compatible**, denoted by $\sigma \dot{\|} \tau$, if and only if $\sigma \| \tau$ and the union $\sigma \cup \tau$ is a bijective signature morphism.

We adapt the notion "(maximal/maximum) clique" from graph theory to our needs:

Definition 5.8. (Clique of Minimal Signature Morphisms) Let F and G be sets of formulae, let B the set of all bijective signature morphisms. The structure $\mathbb{B} = \langle V, E \rangle$ is called **signature bijection graph** between F and G iff

$$\begin{aligned} V &:= \{(\mu, \varphi) | \varphi \in F \text{ and } \mu \in \varphi\overrightarrow{G}\} \\ E &:= \{((\mu, \varphi), (\mu', \varphi')) | \, (\mu, \varphi), (\mu', \varphi') \in V \text{ and } \mu \| \mu'\} \end{aligned}$$

A subset $C \subseteq V$ is called **clique of minimal signature morphisms for** F and G iff $\mu \| \mu'$ for all $(\mu, \varphi), (\mu', \varphi') \in C$ – i.e. $C \times C \subseteq E$. It is said to be **maximal** iff there is no proper superset C' of C that is also a clique for F and G. A maximal clique C is called a **maximum clique** iff no other maximal clique contains more elements than C. Note: not every maximal clique is a maximum clique.

Since all signature morphisms from C are pairwise compatible we can build the union $\sigma_C := \bigcup_{(\mu,\varphi)\in C} \mu$ which we call C's **induced signature morphism**.

Actually, in graph theory we would speak of a clique of a **graph** $\mathbb{B} = \langle V, E \rangle$ (iff $C \times C \subseteq E$) whereas we speak of a clique for two **formulae sets**. Since we define our graph $\mathbb{B} = \langle V, E \rangle$ over these two formulae sets we have thus only accommodated the general graph-theoretic notion **clique** to our particular sort of graphs – the signature bijection graphs. In the following we will use "clique" to refer to our clique of minimal signature morphisms.

The purpose of cliques is to combine their partial signature morphisms to a single coherent total signature morphism that translates all the involved formulae in the clique back and forth.

Definition 5.9. Let F and G be sets of formulae and let σ be a bijective signature morphism defined on Σ_F. We define the σ **induced clique** from F to G as:

$$C_\sigma := \{(\sigma_\varphi, \varphi) | \varphi \in F \text{ and } \sigma_\varphi \in \varphi\overrightarrow{G}\}$$

Proposition 5.10. Let F and G be sets of formulae and let σ be a bijective signature morphism defined on Σ_F.

1. $sen(\sigma)(F) \cap G = \{sen(\sigma_\varphi)(\varphi) | (\sigma_\varphi, \varphi) \in C_\sigma\}$
2. $|sen(\sigma)(F) \cap G| = |C_\sigma|$

Proof.

1. by expansion of the definitions.
2. In fact it can be shown again by expansion of the definitions that we have for every $\psi \in sen(\sigma)(F) \cap G$ exactly one $(\sigma_\varphi, \varphi) \in C_\sigma$ such that $sen(\sigma_\varphi)(\varphi) = \psi$ as well as the other way round. □

Theorem 5.11. If C is a maximum clique for F and G, and σ_C its induced signature morphism, then there is an expansion σ of σ_C defined on Σ_F, such that $sen(\sigma)(F) \cap G$ is a maximum intersection.

Proof. Suppose C is such a maximum clique and σ such an expansion of σ_C. We prove by contradiction that $sen(\sigma)(F) \cap G$ is a maximum intersection: assume there is a signature morphism τ defined on Σ_F such that $sen(\tau)(F) \cap G$ is a proper superset of $sen(\sigma)(F) \cap G$. By proposition 5.10 this implies for the induced cliques C_σ and C_τ that $|C_\sigma| < |C_\tau|$. Moreover since σ is an expansion of σ_C induced by the maximum clique C, it must be $C = C_\sigma$. Hence, we infer $|C| < |C_\tau|$ which contradicts the assumption that C is a maximum clique. □

5.1.3 Illustrative Example

Let us return to our introductory example about how to find the ring theory $\mathcal{R}$ as intersection of the other ring theories $\mathcal{R}_1$ and $\mathcal{R}_a$. Our interest in this example is to find the maximum intersection of the axioms of the latter theories. Suppose these two theories are given as $\mathcal{R}_1 = \langle \Sigma_1, \Gamma_1 \rangle$ and $\mathcal{R}_a = \langle \Sigma_a, \Gamma_a \rangle$ with

$$\begin{aligned} \Sigma_1 &= \langle \{0,1\}, \{+, *\} \rangle \\ \Gamma_1 &= \{\mathrm{com}(+), \mathrm{ass}(+), \mathrm{ass}(*), \mathrm{dist}(*,+), \mathrm{inv}(+,0), \mathrm{neut}(+,0), \mathrm{neut}(*,1)\} \end{aligned}$$

and

$$\begin{aligned} \Sigma_a &= \langle \{e\}, \{\oplus, \otimes\} \rangle \\ \Gamma_a &= \{\mathrm{com}(\oplus), \mathrm{com}(\otimes), \mathrm{ass}(\oplus), \mathrm{ass}(\otimes), \mathrm{dist}(\otimes, \oplus), \mathrm{inv}(\oplus, e), \mathrm{neut}(\oplus, e)\}. \end{aligned}$$

All possible minimal translations from axioms of Γ_1 to axioms of Γ_a are listed in the table below:

	Γ_1	Γ_a	μ
1	$\mathrm{com}(+)$	$\mathrm{com}(\otimes)$	$[+ \rightsquigarrow \otimes]$
2	$\mathrm{com}(+)$	$\mathrm{com}(\oplus)$	$[+ \rightsquigarrow \oplus]$
3	$\mathrm{ass}(+)$	$\mathrm{ass}(\oplus)$	$[+ \rightsquigarrow \oplus]$
4	$\mathrm{ass}(+)$	$\mathrm{ass}(\otimes)$	$[+ \rightsquigarrow \otimes]$
5	$\mathrm{ass}(*)$	$\mathrm{ass}(\oplus)$	$[* \rightsquigarrow \oplus]$
6	$\mathrm{ass}(*)$	$\mathrm{ass}(\otimes)$	$[* \rightsquigarrow \otimes]$
7	$\mathrm{dist}(*,+)$	$\mathrm{dist}(\otimes, \oplus)$	$[* \rightsquigarrow \otimes, + \rightsquigarrow \oplus]$
8	$\mathrm{inv}(+,0)$	$\mathrm{inv}(\oplus, e)$	$[+ \rightsquigarrow \oplus, 0 \rightsquigarrow e]$
9	$\mathrm{neut}(+,0)$	$\mathrm{neut}(\oplus, e)$	$[+ \rightsquigarrow \oplus, 0 \rightsquigarrow e]$
10	$\mathrm{neut}(*,1)$	$\mathrm{neut}(\oplus, e)$	$[* \rightsquigarrow \oplus, 1 \rightsquigarrow e]$

Table 5.1. Minimal formula translations from the axioms of $\mathcal{R}_1$ (ring with one) and $\mathcal{R}_a$ (Abelian ring). Each row is represented by a node in the bijective signature graph from $\mathcal{R}_1$ to $\mathcal{R}_a$.

These ten minimal formula translations together with their associated formulae are represented as nodes of the bijective signature graph from $\mathcal{R}_1$ to $\mathcal{R}_a$ depicted below. The edges in the graph represent the bijective compatibility of the minimal formula translations in the connected nodes. There are two maximal cliques: $C_1 = \{1, 4, 5, 10\}$ and $C_2 = \{2, 3, 6, 7, 8, 9\}$. Only the latter is a maximum clique. From the union of all the nodes in the maximum clique we finally get the expected intersection $\mathcal{R}$ together with its theory morphism to $\mathcal{R}_1$ and $\mathcal{R}_a$ respectively: the union of the Γ_1-axioms from the clique C_2 gives us the axioms of $\mathcal{R}$, i.e

$$\mathrm{Ax}(\mathcal{R}) = \{\mathrm{com}(+), \mathrm{ass}(+), \mathrm{ass}(*), \mathrm{dist}(*, +), \mathrm{inv}(+, 0), \mathrm{neut}(+, 0)\}.$$

Alternatively we can get the isomorphic intersection $\mathcal{R}'$ by the union of the Γ_a-axioms from the clique C_2:

$$\mathrm{Ax}(\mathcal{R}) = \{\mathrm{com}(\oplus), \mathrm{ass}(\oplus), \mathrm{ass}(\otimes), \mathrm{dist}(\otimes, \oplus), \mathrm{inv}(\oplus, e), \mathrm{neut}(\oplus, e)\}.$$

And the union of the minimal formula translations from clique C_2 gives us the signature morphism $\sigma = \{+ \rightsquigarrow \oplus, * \rightsquigarrow \otimes, 0 \rightsquigarrow e\}$ with $\sigma(\mathcal{R}_1) \cap \mathcal{R}_a$ and $\mathcal{R}_1 \cap \sigma^{-1}(\mathcal{R}_a)$ – which is in fact the theory morphism $\sigma: \mathcal{R} \to \mathcal{R}_a$ (or alternatively $\sigma^{-1}: \mathcal{R} \to \mathcal{R}_1$).

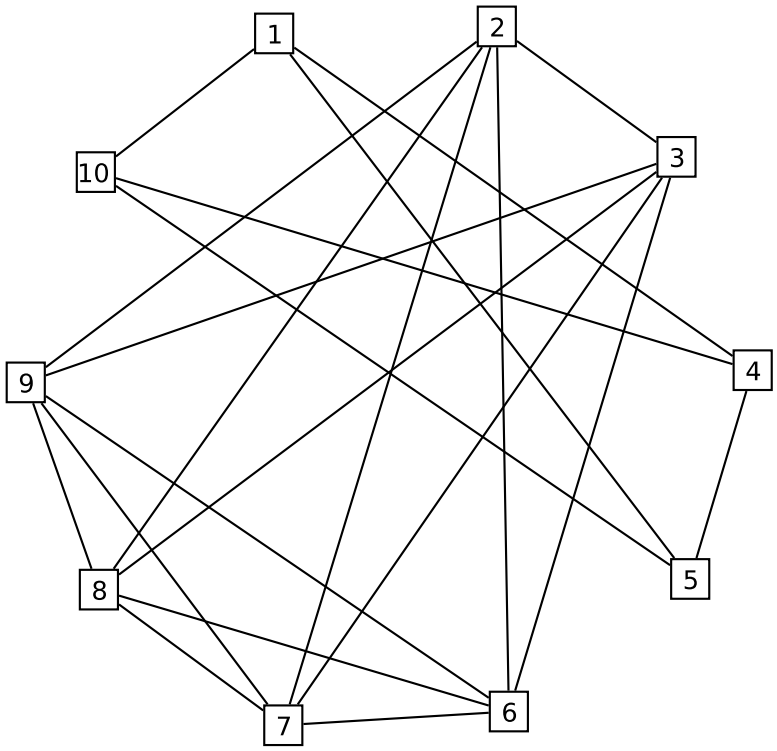

Figure 5.2. A bijective signature graph from $\mathcal{R}_1$ to $\mathcal{R}_a$. The node numbers represent the rows from table 5.1 – i.e. formulae with minimal translations. The edges represent the bijectively compatibility of the minimal formula translations in the connected nodes. The dotted edges build the maximal clique C_1 and the continued edges build the maximum clique C_2.

Corresponding to the bijective minimal formula translations we get the set of axioms which constitute the theory intersection. The figure below shows the ring axioms as intersection of the axioms of an Abelian ring with a ring with a unit.

	Γ_1	Γ	Γ_a	μ
1	com$(+)$		com$(\otimes)$	$[+ \rightsquigarrow \otimes]$
2	com$(+)$	com$(+)$	com$(\oplus)$	$[+ \rightsquigarrow \oplus]$
3	ass$(+)$	ass$(+)$	ass$(\oplus)$	$[+ \rightsquigarrow \oplus]$
4	ass$(+)$		ass$(\otimes)$	$[+ \rightsquigarrow \otimes]$
5	ass$(*)$		ass$(\oplus)$	$[* \rightsquigarrow \oplus]$
6	ass$(*)$	ass$(*)$	ass$(\otimes)$	$[* \rightsquigarrow \otimes]$
7	dist$(*,+)$	dist$(*,+)$	dist$(\otimes,\oplus)$	$[* \rightsquigarrow \otimes, + \rightsquigarrow \oplus]$
8	inv$(+,0)$	inv$(+,0)$	inv$(\oplus,e)$	$[+ \rightsquigarrow \oplus, 0 \rightsquigarrow e]$
9	neut$(+,0)$	neut$(+,0)$	neut$(\oplus,e)$	$[+ \rightsquigarrow \oplus, 0 \rightsquigarrow e]$
10	neut$(*,1)$		neut$(\oplus,e)$	$[* \rightsquigarrow \oplus, 1 \rightsquigarrow e]$

Table 5.2. The intersection of $\mathcal{R}_1$ (ring with one) and $\mathcal{R}_a$ (Abelian ring) yields the the ring theory $\mathcal{R}$ whose axioms Γ are determined from the axiom Γ_1 of $\mathcal{R}_1$ and Γ_a of $\mathcal{R}_a$ respectively.

A concrete implementation in the functional programming language Haskell of theory intersection for explored theories will be presented in section 10.1.7.

Chapter 6
Formal Language

Theory interpretation and intersection search as well as theory completion as presented in the previous sections rely on a matching function that returns a minimal formula translation. We have not specified this function further, as it depends on the concrete nature of signature morphisms and sentence translations and how they are tied to each other concretely. As we mentioned already in the introduction of derivability systems (cf. chapter 4) this is typically done by inductively extending signature morphisms to sentence translations based on the inductive construction of sentences from signatures. Consequently we have to define a formal language at first. Since the definition scheme of inductive term construction allows for arbitrarily many variants of formal languages, we have to make a choice. In the literature on logics one typically finds formal languages falling into the classical dichotomies *typed/untyped* and *first-order/higher-order*. We will restrict to an untyped formal language since this is the easier start to define formula translation up on. Concerning the order, our language will be higher order in principle – though all our examples will be from first-order. Of course, we should be aware of the well known fact that untyped higher order logic languages lead to inconsistency (e.g. the Russell Paradox). On the other hand, we know that there are several ways to overcome this dilemma by imposing additional constraints – via type systems – on a higher-order language. Since semantics is not in our focus, adherence to a particular type system would needlessly complicating our goal: the basic idea of defining formula translation up on inductive formula construction is certainly neither restricted to first-order logic nor to exactly one specifically typed higher-order logic. Informally, our general idea of formula translation is to replace the unbound non-logical symbols by other non-logical symbols. Obviously, this idea does not depend on whether our language is first-order or higher-order. Even though we will not provide our formal language with semantics we dare the following semantic claim: if we replace all unbound non-logical symbols of all formulae in a theory consistently, then this change would still preserve the entailment relation (cf. the translation axiom in definition 4.1). That means in particular that all translated theorems are again theorems derivable from the translated axioms – in other words: such translations would not destroy a theory as a logical unit[6.1].

After we have committed to a higher-order untyped[6.2] language, we still have to make further agreements. In order to formulate our informal idea of formula translation as general as possible, we will consider only general basic concepts that can be found in most logics. The formal language we are going to define is very much oriented on λ-terms since that has proven to be powerful and general for many formalisms. In every logic we distinguish between **logical symbols** and **non-logical symbols**. In many logics non-logical symbols are differentiated into constants, function and relation symbols, and variables. Furthermore all these symbol sets are usually partitioned into ari-

6.1. What Hilbert meant in the context of Euclidean geometry: *point, line, plane*, and others, could be substituted by *tables, chairs, glasses of beer* and other such objects.

6.2. Those readers who still feel uncomfortable with an untyped higher-order language may imagine it as appropriately typed in order to avoid inconsistency.

ties: for every $k \in \mathbb{N}$ there is a distinguished set of k-place function symbols, and similarly for relation symbols. We abandon these further differentiations, since it depends very much on the kind of logic. Concerning the arity of function and relation symbols, we tacitly assume that symbol mappings will respect symbol arity. Alternatively, we could replace any n-ary (for $n > 1$) function by an isomorphic higher order unary function – this transformation is known as **currying**: consider a binary function $f\colon A \times B \to C$. Then its curried variant is a function $f'\colon A \to \{g\colon B \to C\}$ (i.e. a mapping from A to the set of all functions from B to C) defined by $f(a,b) = (f'a)(b)$ for all $a \in A$ and $b \in B$. A similar transformation can be applied to relations: for every binary relation R we can define an isomorphic higher-order[6.3] unary relation R' defined by $R(a,\ b) \Leftrightarrow (R'(a))(b)$.

Depending on the formalism one can also find various binders like the logical quantifiers $\forall$, and $\exists$ (classical first order logic), and non-logical binders like Σ (sum), Π (product), $\int$ (integral), ∂ (derivative), etc. We will have $\boldsymbol{\lambda}$ as the only **binder**, since in most logics[6.4] (inparticular classical logics) it is possible to represent all the other binder constructs by the λ-binder together with a constant. To give an example: in introductiory textbooks like [2] the expression $\forall x.P(x)$ is represented by $\Pi(\lambda x.P(x))$ [6.5]. Similarly, we can represent any other binder in most logics by a constant and λ. Yet, for readability, we will not adhere strictly to pure λ-notation of quantification, e.g. we will prefer to write $\forall x.P(x)$ instead of $\Pi(\lambda x.P(x))$. A λ-binder always binds a **variable**, which is a non-logical symbol. All the unbound non-logical symbols in a formula are its **parameters**. For simplicity, we assume that variables and parameters are from disjoint sets whose union is the set of all non-logical symbols. This implies that variables will occur in our language only in a bound position – i.e. our formulae are assumed to be in **closed form**.

Definition 6.1. An **(untyped) logical language** is a triple $\mathcal{L} = \langle L, P, V \rangle$, where L is called the **set of logical symbols**, P the **set of parameters**, and V the **set of variables**. We call the set of non-logical symbols (i.e. $P \cup V$) just **symbols**. L is assumed to be finite whereas $V$and P are assumed to be infinite.

The set of $\boldsymbol{\mathcal{L}}$**-terms** is defined as the smallest set[6.6] $\mathcal{L}$ meeting:

- $L, P, V \subset \mathcal{L}$ the **atoms** or **atomic terms**.
- If f is an atom and $t_1, ..., t_n \in \mathcal{L}$, then $f(t_1, ..., t_n) \in \mathcal{L}$ is an **application**.
- If $v \in V$ and $t \in \mathcal{L}$, then $\lambda v.t \in \mathcal{L}$ is a **binding term**.

We call any set $\Sigma \subset P$ a **signature**. **Terms (constructable) from a signature** Σ, denoted by $\boldsymbol{\mathcal{L}(\Sigma)}$, are defined as all those $\mathcal{L}$-terms that do not contain any parameter from $P - \Sigma$. The set of variables occurring in a term φ is denoted by $\boldsymbol{\mathcal{V}ar(\varphi)}$ and $\boldsymbol{\Sigma_\varphi}$ for the parameters, and $\boldsymbol{\mathcal{S}ymb(\varphi)}$ of both parameters and variables occurring in it.

In fact this formal definition does not enforce formulae to be in closed form. If a formula of a theory is not in closed form then at least in classical logic it is implicitly universally quantified. We want to consider here only languages where any open formula is implicetly quantified by some appropriate quantifier, so that we can assume without loss of generality that we only have formulae in closed form. Moreover, we assume unique binding, i.e. that no variable in a formula is bound by more than one binder.

6.3. In fact the notion of "higher order relations" is rather uncommon in literature. However, since functions can be considered as special (i.e. right unique) relations – or the other way round relations as functions with truth values it is quite natural to extend the notion "higher order" from functions to relations.

6.4. A typical exception is monadic dynamic logic.

6.5. We neglected again the type assignments to the symbols.

6.6. Note, we use the letter $\mathcal{L}$ for both the triple $\mathcal{L} = \langle L, P, V \rangle$ and the set of $\mathcal{L}$-terms.

The syntactic form of applications is called **prefix notation**[6.7] when the operator is always in front position followed by its arguments. In mathematics we usually find very often **infix notations** (e.g. "$a+b$"), **postfix notation** (e.g. "$n!$"), and **mixfix notation** (e.g. "$\langle a, b\rangle$"). All these notations can be isomorphically represented in prefix notation. For formal proofs it is easier to allow only one notation, but in informal text we will use in examples all the other notations as well in favor of readability.

With our definition, we commit to a very basic notion of signature whose elements are just plain symbols (parameters) without any internal structure. In typed languages symbols have a substructure, since they are annotated with types. And types are usually compound expressions on their own, which make the language more involved. On the other hand, as type expressions are also inductively defined, we had to deal with a kind of nested inductive formula construction. Therefore, it would be an unnecessary complication to consider a typed language from the start to introduce the principal idea of mapping signatures to sentences. By convention we will reserve the letters a, b, c, ..., r, possibly with indices, as well as their capital variants for parameters, and u, v, w, x, y, z, accordingly for variables.

Sentences in our language are called "$\mathcal{L}$-terms", but in the following we will treat "$\mathcal{L}$-term", "**term**", "**sentence**", and "**formula**" as synonyms. Here the minimal signature of a formula φ (cf. definition 4.20) is the set of all parameters occurring in φ. Moreover, we have defined a concrete mapping from any signature $\Sigma \subset P$ to sentences. Since a theory is a pair of signature and sentences built from this signature, we have also specified the notion of theory with respect to our language $\mathcal{L}$. Concerning the *sen* functor from definition 4.7 we have thus instantiated the object mapping part of the functor.

Now we turn to the functor's mapping of morphisms, i.e. how signature morphisms are mapped to sentence translations. Our signatures are sets of parameters, hence signature morphisms are just **parameter mappings**. But we will also need variable mappings for our matching technique (cf. chapter 7). In contrast to parameter mappings, that are possibly non-injective, **variable mappings** must be one-to-one mappings, because non-injective variable replacements usually destroy a theories by so called variable capturing. For instance an axiom of assymmetry $\forall x, y.\ x < y \Rightarrow \neg\, y < x$ subjected to the non-injective variable replacement $[x \rightsquigarrow z, y \rightsquigarrow z]$ yields the antinomy $\forall x, x.\ x < x \Rightarrow \neg\, x < x$. Finally we will introduce for any composition of parameter and variable mappings the general concept of **symbol mapping** (recall that we called non-logical symbols for brevity just symbols, so symbol mappings never affect logical symbols!).

Definition 6.2. (Symbol mappings) Let $\mathcal{L} = \langle L, P, V\rangle$ be a formal language and let $\mathcal{S} = P \times V$. A bijective mapping from V to V is called **variable bijection**, a mapping from P to P is called **parameter mapping** (sometimes **signature morphism**). Any mapping from $P \times V$ into $P \times V$ that is a composition of parameter mappings and variable bijections is called **non-logical symbol morphism** – or for short **symbol morphism**. We define the domain of σ as $\mathbf{dom(\sigma) := \{x \in \mathcal{S} | x \neq \sigma(x)\}}$ and the codomain as $\mathbf{cod(\sigma) := \{\sigma(x) | x \in dom(\sigma)\}}$[6.8]. Moreover, we call σ a **(minimal) symbol morphism for** φ if the domain of σ equals the set of all parameters of φ (i.e. $\mathrm{dom}(\sigma) \subseteq \mathcal{S}ymb(\varphi)$).

Based on symbol mappings we define inductively the translation of terms (or formulae, or sentences):

6.7. Also known as Polish notation.

6.8. Note that σ is defined on the whole set $\mathcal{S}$ and that $\mathrm{dom}(\sigma)$ is only the subset of $\mathcal{S}$ where σ is not the identity function – $\mathrm{dom}(\sigma)$ as used here is also called literature the support of the mapping σ, often denoted by $\mathrm{supp}(\sigma)$.

Definition 6.3. (Formula Translation) Let $\mathcal{L} = \langle L, P, V \rangle$ be a formal language and let $\mathcal{S} = P \times V$. We extend a symbol morphism $\sigma: \mathcal{S} \to \mathcal{S}$ inductively to a term mapping – called **formula translation** (or sometimes simply **translation** or **renaming**) and denoted by σ^*:

$$\begin{aligned} \sigma^*(s) &:= \sigma(s) \text{ if } s \in \mathcal{S} \\ \sigma^*(s(\varphi_1, ..., \varphi_n)) &:= \sigma^*(s)(\sigma^*(\varphi_1), ..., \sigma^*(\varphi_n)) \\ \sigma^*(\lambda v.\varphi) &:= \lambda(\sigma(v)).\sigma^*(\varphi) \end{aligned}$$

We say φ **matches on** ψ iff there is a σ with $\sigma^*(\varphi) = \psi$.

Now we have defined a complete instance of a *sen* functor for our language: symbol morphisms are signature morphisms and the corresponding term mappings are sentence translations. This instance of a *sen* functor clearly satisfies the $\cap$*-invariance* condition – basically due to the following proposition:

Proposition 6.4. If two symbol mappings σ and τ are equal on their domain then the corresponding term mappings σ^* and τ^* are equal too.

Proof. Follows immediately from the inductive definition of our formal language and symbol mappings. □

Let us now recall the purpose of these definitions: our ultimate goal is automated theory interpretation and intersection search, which essentially relies on the $\cap$*-invariance* of the involved *sen* functor. Provided that *sen* is $\cap$*-invariant*, an elementary part of the search algorithms is the matching problem: given two formulae φ and ψ which signature morphism σ (if any) solves the equality $sen(\sigma)(\varphi) = \psi$.

Symbol morphisms are $\cap$*-invariant* and thus in the context of the formal language $\mathcal{L}$, our **symbol renaming problem** is to find for two given formulae φ and ψ a symbol morphism σ such that the equality $\sigma^*(\varphi) = \psi$ holds. In fact a very similiar but even more complex task, namely matching variables against terms, is implemented very efficiently in many theorem provers. Nevertheless, we dedicate the following sections to this matching problem for two reasons:

1. Since our symbol matching problem is even simpler than a theorem prover's matching problem, it allows for further optimization techniques. We will investigate the technique of **formula abstraction** that particularly pays off for matching a formula against millions of formulae in a database.
2. Matching formulae modulo some equality tends to be rather expensive for the general matching problem. In case of mere symbol matching, the technique of formula **normalization** and **standardization** can improve the search significantly.

Chapter 7
Matching

This chapter is about two kinds of matching problems: 1) the simple renaming problem and 2) the equational renaming problem (or renaming modulo equivalence problem). The former is a specialisation of the latter as it can be specified as a renaming "modulo identity" problem. Furthermore, a renaming problem can be considered as special case of a matching problem which in turn is a special case of a unification problem – both in its "simple" and "modulo equivalence" variants. We want to sketch this relationship of problem specialisation. In all cases we have to formulae, φ and ψ, and we want to make them equal (modulo some equivalence relation) via renaming, matching, or unification. The most general case is unification modulo some equivalence relation $\sim$, where we are looking for two substitutions σ and τ such that $\sigma(\varphi) \sim \tau(\psi)$. In the "simple" unification case the identity $\sigma(\varphi) = \tau(\psi)$ has to be solved. Matching is the specialisation of unification in that τ is the identity – i.e. we are looking for only one substitution that solves the equation $\sigma(\varphi) \sim \psi$ (or $\sigma(\varphi) = \psi$ respectively). In both, the matching and unification problem, we are looking for substitutions. Our renaming problem is a specialisation of matching as it has the same form: $\sigma(\varphi) = \psi$ for the **simple renaming problem** and $\sigma(\varphi) \sim \psi$ for the **equational renaming problem**, but we are looking only for symbol morphisms and not for substitutions. A symbol morphism maps symbols to symbols, but a substitution maps symbols to terms, so the former is a special case of the latter. To apply substitutions not just on symbols, but on terms, the corresponding term mapping is inductively defined[7.1] like in definition 6.3. Actually, it is not quite correct to consider a substitution as a generalisation of a renaming, since substitutions map only variables to terms, but leave the parameters untouched. In this respect our notion of symbol morphism is more general than that of substitutions as we want be able to change both, variables and parameters. So strictly speaking our renaming problem is a specialisation of the matching problem, only if we abstract from the domain of mappings.

Unification and matching is a thoroughly investigated research area (e.g. [35] for an extensive survey) and very efficient algorithms have been implemented (e.g. [27]) and applied in the field of automated theorem proving. The special case of efficiently solving mere symbol matching problems has not been investigated on its own – probably because there is no usage for such restricted problems in automated theorem proving (ATP). For theory interpretation search (a research area not yet explored by the ATP community), this restricted problem is relevant though. As there exist already efficient matching algorithms and as the symbol matching problem is a subproblem of the matching problem, the question arises why to develop a dedicated search technique for the symbol matching problem instead of just taking the efficient matching algorithms. Yet, the easier a problem the easier it is to find a solution. Hence, for the symbol matching problem, there should be an easier and faster algorithm than for the matching

7.1. In fact we should have written $\sigma^*(\varphi)$ and $\tau^*(\psi)$ instead of merely $\sigma(\varphi)$ and $\tau(\psi)$ to be correct. However, it is common to identify the underlying mapping σ with the term mapping σ^* after defining the latter inductively in terms of the former. We follow this practise only in informal text, but not in the formal parts like lemmas and proofs, where we prefer explicit rigour.

problem. Our approach is very simple and yet efficient, but not transferable to the matching problem. The basic idea is to divide the problem into two parts: 1) a syntactical identity test and 2) a renaming problem. For that we separate the parameter of a formula from its structure. We call this process of separation **formula abstraction** (also **standardisation**) and the structure of a formula **skeleton**. Thus the two parts of the problem are: 1) test of structural identity and 2) identification of a consistent parameter mapping – if there exist any. Efficiency for solving the whole problem is mainly achieved by the fact that firstly structural identity can be checked in almost constant time and, secondly, a failure of this test supersedes the more expensive second part of the problem, since two formulae never match via symbol morphism whenever they have a different skeleton. Experiments on the largest library of formalised mathematics (more details in chapter 11) have demonstrated that this facilitates a significant reduction of the search space: the over 4 million formulae are instances of only ca. 40000 skeletons, i.e. on average 100 formulae share the same skeleton. By filtering via skeleton identity, we reduce the amount of possible matching candidates from possibly 4 million to 100 on average.

Both renaming problems, the simple and the equational one, are tackled with a common solution strategy that is presented in section 7.4.1 below. But we will approach that theoretical section with two preceding sections to develop the right intuition on the central notions of "skeleton" and "parametrisation" by examples, first for the simple and then for the equational case. The theoretical section provides then rigour definitions of these notions and proves how they solve the renaming problems in principle. An algorithm that solves the simple renaming by means of formula abstraction is described and illustrated by example in section 7.5.2. For the equational renaming problem an algorithm for equivalence modulo associativity and commutativity is developed in section 7.5.3.

7.1 Introduction to the Simple Renaming Problem

Consider the formula $\varphi := \forall x, y.\, f(x) > g(x - y)$ which has the parameters "f", "$>$", "g", and "$-$" (in the following we will say the formula has the **parameter list** $[f, >, g, -]$). **Formula abstraction** can be viewed as a two step process (with arbitrary order): 1) standardising variables and 2) replacing all parameter occurrences by fresh parameters. About the first step we make a general assumption that holds in every logic[7.2] relevant for mathematicians: bijective variable renaming (of bound variables) – commonly known as α-equivalence transformation – does not change the semantics of our terms. For instance $\forall x,\, y.\, f(x) > g(x - y)$ and $\forall a,\, b.\, f(a) > g(a - b)$ are equivalent (though not identical) statements. Obviously we can generate for every formula infinite many α-equivalent variants. α-standardisation is a procedure that determines for every class of α-equivalent formulae a unique representative. One simple way to α-standardise formulae is to rename the variables such that their occurrence in binding positions is in the order $v_1, v_2, \ldots$ – in our example: $\forall v_1, v_2.\, f(v_1) > g(v_1 - v_2)$. The choice of α-standardisation is irrelevant, but to committing to one is decisive. After we have α-standardised our formula we now turn to the next step of formula abstraction: replacing all parameter occurrences by fresh parameters. This gives us finally our skeleton: $\forall v_1, v_2.\, p_1(v_1)p_2p_3(v_1p_3v_2)$. Alternatively, we could consider the two steps of formula abstraction as a λ-abstraction of a formula to the following λ-expression[7.3] that represents the **skeleton**:

$$\lambda v_1, v_2, c_1, c_2, c_3, c_4.\ \forall v_1, v_2.\, c_1(v_1)c_2c_3(v_1\, c_4 v_2).$$

7.2. More precisely, we consider logics whose semantics may change by α-renaming as irrelevant for mathematics

if we apply the skeleton on the symbol list $[x, y, f, >, g, -]$ we get the original formula:

$$\lambda v_1, v_2, c_1, c_2, c_3, c_4.\ \forall v_1, v_2. c_1(v_1)c_2c_3(v_1\, c_4 v_2)\,[x, y, f, >, g, -] \ = \ \forall x, y. f(x) > g(x - y).$$

The application of the λ-term on a parameter (in λ-calculus terminology called reduction) is defined by the application of a corresponding substitution: $(\lambda v.\varphi)\psi = [v \rightsquigarrow \psi]\varphi$. In our case the substitution is a symbol renaming that will later be called the **parametrisation** of our formula: $\delta := [v_1 \rightsquigarrow x,\ v_2 \rightsquigarrow y,\ p_1 \rightsquigarrow f,\ p_2 \rightsquigarrow >,\ p_3 \rightsquigarrow g,\ p_4 \rightsquigarrow -]$. So the application of the above λ-term on our symbol list is the same as applying our parametrisation on the body of that λ-term:

$$\delta\,(\forall v_1, v_2. c_1(v_1)c_2c_3(v_1\, c_4 v_2)) \ = \forall x, y. f(x) > g(x - y).$$

After we have a got a good intuition how formula abstraction works, we want to see, by extending this example, how this method is used to solve a symbol matching problem. Consider therefore a second formula $\psi := \forall M,\ N. \#(M) > \#(M/N)$. The formula abstraction of ψ yields exactly the same skeleton as φ: $\forall v_1, v_2. p_1(v_1)p_2p_3(v_1p_3v_2)$. Thus the precondition for matching is fulfilled. Their parameters are different of course: $[f, >, g, -]$ for φ and $[\#, >, \#, /]$ for ψ. Note that "#" occurs twice in the latter parameter list. There are the two symbol matching problems: 1) $\sigma^*(\varphi) = \psi$ or 2) $\varphi = \tau^*(\psi)$. With our approach this reduces now to finding consistent parameter mappings. For the first problem we look for a parameter mapping that translates $[f, >, g, -]$ to $[\#, >, \#, /]$, and for the second problem in the opposite direction. We immediately see that the first problem has the solution $\sigma = [f \rightsquigarrow \#,\ > \rightsquigarrow >,\ g \rightsquigarrow \#,\ - \rightsquigarrow /]$, whereas the second problem does not have a solution – as σ is not an invertible mapping.

7.2 Introduction to the Equational Renaming Problem

We have already characterised explored theories in contrast to the pure logical definition of theories: an explored theory necessarily contains only a finite number of theorems derivable from its axioms. The principal goal of a formal mathematician is to extend the explored theories by new theorems. However, this is not just a matter of speed in terms of the number of theorems over a period of time, but much more a matter of quality. Otherwise, theorem production would only be bound to computation speed of fast computers, since applying inference rules can be encoded as a computer program. For instance, suppose a explored theory containing the axiom $\forall x. x + 0 = x$ and a calculus having the $\vee$-introduction as inference rule: $\frac{\varphi}{\varphi \vee \psi}$ for arbitrary ψ. A computer program implementing this could easily derive within a second thousands of new "theorems" from that axiom like $(\forall x. x + 0 = x) \vee 0 = 0$, $(\forall x. x + 0 = x) \vee 0 \neq 0$, $(\forall x. x + 0 = x) \vee 0 = 0 \vee 0 \neq 0$, etc. Obviously these new "theorems" are completely uninteresting to any mathematician – they lack new information. In fact from just looking at the two formulae $\forall x. x + 0 = x$ and $(\forall x. x + 0 = x) \vee 0 = 0$ everybody with a minimal competence in formal logic can immediately see which of them is the uninteresting formula, even independently of the context. Messing up an explored theory with trivial theorems like above is something that should be always avoided. We call this the **no triviality maxim**[7.4].

7.3. Note that this λ-expression has to be considered on the meta level and not as part of our formal language.

7.4. This is in fact a general maxim corresponding to the principle from information theory: the more noise in a system the less information it contains.

So it might be an idea to implement a clever computer program that discards all the trivial theorems. Yet, to define a formal criterion for "trivial" is a matter of agreement of the explored theories' users and their mathematical competence. Moreover, in general such a criterion cannot be independent of those formulae already contained in the explored theory. Taking again our example from above: if $\forall x.x+0=x$ is in our explored theory then $\forall x.x = x + 0$ would be considered a trivial theorem. If, however, we had $\forall x.0+x=x$ instead of $\forall x.x+0=x$ as axiom, then $\forall x.x=x+0$ might be interesting – for instance in group theory to make explicit that the left neutral implies the right neutral.

Actually, it will not be our concern to provide a definition of triviality. The only aspect we want to focus on here is the fact that the set of theorems in a explored theory is very selective. But this raises a problem for automated theory interpretation search where the simple renaming approach fails: consider for instance a target theory that contains a theorem $\forall a.a=a\circ e$ and the source theory with $\forall x.x+0=x$ as axiom. Due to the "no triviality maxim" we would find neither $\forall x.x = x + 0$ in the source theory nor $\forall a.a\circ e=a$ in the target theory, and hence our theory interpretation search would fail. This is unsatisfactory, because a theory interpretation search method should be able to close this inference gap.

Yet, every inference gap our search method could close can be only an approximation of the infinite set of formula equivalences, because it is well known that sufficiently expressive formal languages are undecidable – i.e. no algorithm can be able to decide the provability of every formula. In particular there is no algorithm that could decide every equivalence of two formulae. **Matching modulo equality** aims to support our search method with an effective approximation of general formula equivalence. It should be effective in the following respects: 1) It should not slow down the search process significantly and 2) it should cover relevant formula equivalence classes.

In this thesis we distinguish to approaches to tackle the equational renaming problem: 1) standardisation, i.e. formula abstraction into skeleton and parametrisation, and 2) normalisation via term rewriting based on logical equivalence transformations as rewrite rules. Chapter 8 is dedicated to the latter approach, whereas the former is subject of this chapter. More precisely, the theory of standardisation modulo equivalence for arbitrary equivalence relations is presented in section 7.4.1. An introductory example for equivalence modulo associativity and commutativity (AC) is given in the next section and the last section of this chapter provides an algorithm for AC-standardisation.

7.3 Introduction to the AC-Renaming Problem

Many theories in mathematics have associative and commutative operators as e.g. union and intersection in set theory. We call such an operator an **AC-operator** and the operator together with its argument an **AC-term**. For AC-operators the order of their argument can be permuted without changing the semantics of the whole expression – we say they are all **AC-equal**. Any permutation of arguments of an AC-operator is called **AC-transformation**.

As mentioned above this diversity of semantically equivalent variants is a problem for automated theory interpretation search, because in explored theories only a single representative of the equivalence class of formulae is explicitly displayed. For instance, consider the formula $\psi_1 := \forall A, B, C.\, A\cap(B\cup C)=(A\cap B)\cup(A\cap C)$ from set theory, which is equivalent to $\psi_2 := \forall A,\, B,\, C.\, (B\cap A)\cup(C\cap A)=(B\cup C)\cap A$. In fact there are 64 possible AC-equal variants of that formula, but usually at most one of them will be

present in an explored theory. Note that the notion of symmetry in context of relations (in particular "=") is a synonym for commutativity. For our purpose it is more convenient to call a relation commutative whenever it is usually called symmetric, since we have omitted a dedicated symbol category for relations in our language anyway. Now consider we have in a source theory the formula $\varphi := \forall A, B, C.\, A \otimes (B \oplus C) = (A \otimes B) \oplus (A \otimes C)$, then our current matching method would succeed to match φ to ψ_1 with the parameter mapping $\sigma := [\, \oplus \rightsquigarrow \cap\,,\ \otimes \rightsquigarrow \cup\,]$. However, it would fail to match φ to ψ_2 with that method. We are only able to match φ with ψ_2 if we allow AC-transformation after (or before) the parameter mapping. Such a type of matching is called **symbol matching modulo AC-equality** or just **AC-matching**. Since most logics have associative and commutative logical connectives, AC-matching is also relevant independently of specific theories. Our goal is formula matching modulo AC-equality given the set of all AC-operators from the logical symbols (logic dependence) and parameters (theory dependence). Classical logic-dependent AC-operators are disjunction and conjunction, but also the exclusive disjunction, which will play a prominent role in chapter 8.

Our previous symbol matching problem was to find for given formulae φ and ψ a parameter mapping σ such that $\sigma^*(\varphi) = \psi$ if it exists. Now our **AC-matching problem** is to solve the equality $\sigma^*(\varphi) \sim \psi$ where $\sim$ denotes AC-equality modulo α-renaming. A naive approach to tackle this problem would be generating from an existing target formula ψ the class $[\psi]$ of all formulae AC-equal to that formula and then trying to match φ against each member of this target AC-equivalence class $[\psi]$. Since the amount of formulae in $[\psi]$ grows exponentially with the number of AC-operators occurring in ψ, this approach soon becomes intractable.

Our approach to solve this problem is an adaption of that from the simple renaming problem where a formula φ can be isomorphically represented by its skeleton $\varphi\!\downarrow$ together with its parametrisation δ_φ. Now we want to represent a whole equivalence class $[\varphi]$ by a single skeleton, called the **AC-skeleton**, and its parametrisation. Again we define the skeleton as a least element of an equivalence class of linearisations. However, this time we have to consider all linearisations not only for one formula φ, but for all formulae from the AC-equivalence class $[\varphi]$. The purpose of the AC-skeleton is the same as in the simple matching problem: to serve as efficient filter criterion for matching candidates: two formulae can match modulo AC only if they have an identical AC-skeleton. In the case of AC-matching we reduce a much larger search space than in the case of simple matching.

The second task of the matching problem, though, is different. For the simple symbol matching problem we had to check whether the two parametrisation of the two involved formulae can be composed to a parameter mapping. For the AC-matching problem it is not sufficient to consider only two parametrisations to find all possible parameter mappings modulo AC. This immediately becomes clear by the following simple example: consider the two propositional formulae $\varphi := a \wedge b$ and $\psi := c \wedge d$ where $a, b, c,$ and d are parameters. The corresponding AC-equivalence classes are $[\varphi] = \{a \wedge b, b \wedge a\}$ and $[\psi] = \{c \wedge d, d \wedge c\}$. Their common representative skeleton looks like $[\varphi]\!\downarrow = p_1 \wedge p_2$. Obviously we could match φ to ψ modulo AC with both parameter mappings $\sigma_1 := [a \rightsquigarrow c, b \rightsquigarrow d]$ and $\sigma_2 := [a \rightsquigarrow d, b \rightsquigarrow c]$. These can be reconstructed then from the set of all parametrisation for which $\delta^*_\varphi([\varphi]\!\downarrow) \sim \varphi$ holds – i.e. we have two parametrisations for $[\varphi]$, namely $\delta_{1,\varphi} := [p_1 \rightsquigarrow a, p_2 \rightsquigarrow b]$ and $\delta_{2,\varphi} := [p_1 \rightsquigarrow b, p_2 \rightsquigarrow a]$, and for $[\psi]$ respectively $\delta_{1,\psi} := [p_1 \rightsquigarrow c, p_2 \rightsquigarrow d]$ and $\delta_{2,\psi} := [p_1 \rightsquigarrow d, p_2 \rightsquigarrow c]$. Thus we have $\sigma_1 = \delta_{1,\psi} \circ \delta^{-1}_{1,\varphi} = \delta_{2,\psi} \circ \delta^{-1}_{2,\varphi}$ and $\sigma_2 = \delta_{1,\psi} \circ \delta^{-1}_{2,\varphi} = \delta_{1,\psi} \circ \delta^{-1}_{2,\varphi}$. This example suggests that we have to represent an AC-equivalence class by one AC-skeleton but many **AC-parametrisations** in order to find all possible parameter mappings modulo AC.

After these introductory examples for the simple and the equational renaming problems we want to provide the formal definitions of skeleton and parametrisation, and see how they can be used to solve those renaming problems in theory. Concrete algorithms to solve them in practise are described in the subsequent sections, whereby the AC-standardisation is the only instance we consider here for the equational renaming problem.

7.4 Formula Abstraction

In the introductory sections we have already seen that a renaming will be a composition of two parametrisation where one is even inverted. Since a parametrisation is a mapping and a mapping is not always invertible, this is a problem. We solve this problem by taking a set theoretic view on mappings, namely as **right unique** binary relations. Thus, $(x, y) \in \sigma$ is interpreted as $\sigma(x) = y$ for every $x \in \mathrm{dom}(\sigma)$. Right uniqueness means that $(x, y) \in \sigma$ and $(x, z) \in \sigma$ implies $y = z$, which guarantees that for every argument $x \in \mathrm{dom}(\sigma)$ there is exactly one value associated. Furthermore, we need to extend our notion of **composition** and **inversion** from function to relations.

Definition 7.1. (Composition and Inversion for Relations) Let R and Q be binary relations on a common set A (i.e. $R, Q \subset A \times A$).

- $R^{-1} := \{(y, x) | (x, y) \in R\}$ is called the **inversion** of R, and
- $R \circ Q := \{(x, y) | \exists z.(x, z) \in Q \wedge (z, y) \in R\}$ the **composition** of R and Q.

Relations, in contrast to functions, are always invertible: the inversion of a binary relation is again a binary relation, but the inversion does not preserve the right uniqueness. Hence, the inversion of a function as binary relation is in general no longer a function. The one-to-one functions are the only ones remaining right unique after inversion. However, we want to invert any function and compose it with others. This is now possible with the interpretation of functions as binary relations. After all, we are only looking for functions – parameter symbol matchings – but to find them we turn functions into non-functions: given two parameter mappings σ and τ, where maybe neither is invertible as function, we look at the composition $\sigma \circ \tau^{-1}$. Although τ^{-1} is in general not a function, the composition might be one, as the following example demonstrates:

Example 7.2. Suppose two parameter mappings $\sigma := \{(c_1, b_1), (c_2, b_1), (c_3, b_1), (c_4, b_2)\}$ and $\tau := \{(c_1, a_1), (c_2, a_1), (c_3, a_2), (c_4, a_3)\}$, represented as binary relations. The inverse τ^{-1} is no longer right unique, whereas the composition $\sigma \circ \tau^{-1}$ is right unique:

$$\sigma \circ \tau^{-1} \;=\; \{(a_1, b_1), (a_2, b_1), (a_3, b_2)\}$$

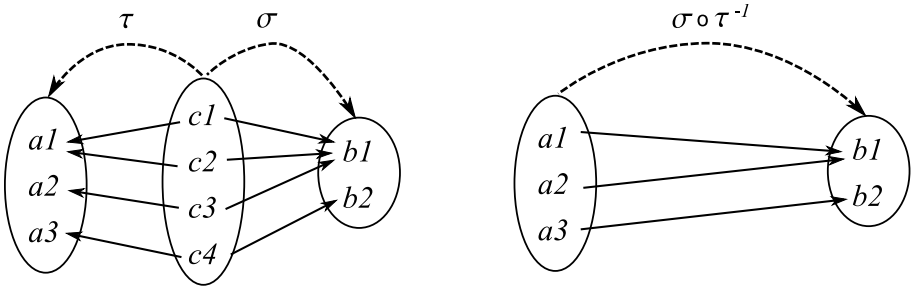

Figure 7.1.

If we consider σ and τ as parametrisations, then we are exactly interested in such compositions $\sigma \circ \tau^{-1}$. The following lemma makes this purpose explicit. Beforehand, we list some general basic properties of relations.

Lemma 7.3. Let f, g, and h functions, where the image of h equals the domain of g. If $f = g \circ h$ then $f \circ h^{-1} = g$.

Proof. We consider f, g, and h as right unique binary relations. Assume $f = g \circ h$. We show $f \circ h^{-1} \subseteq g$ and $f \circ h^{-1} \supseteq g$.
$\subseteq$: Assume $(x, y) \in f \circ h^{-1}$. By definition of composition and inversion, there is a z with $(z, x) \in h$ and $(z, y) \in f$. From $(z, y) \in f$ and $f = g \circ h$ follows that there must be a x' with $(z, x') \in h$ and $(x', y) \in g$. Since h is right unique it follows $x = x'$ and hence $(x, y) \in g$
$\supseteq$: Assume $(x, y) \in g$. Since the image of h equals the domain of g, there must be a z with $(z, x) \in h$ and furthermore $(z, y) \in f$ due to $f = g \circ h$. Hence there is a z with $(x, z) \in h^{-1}$ and $(z, y) \in f$ which means $(x, y) \in f \circ h^{-1}$. □

With this background on relations at hand, we are ready to some assertions on skeletons and their parametrisation. But at first we have to introduce these notions formally.

7.4.1 Skeleton and Parametrisation of a Formula

In our informal idea of a skeleton of a term all its parameters were replaced by fresh parameters for each parameter occurrence in the original term. A formula where no parameter occurs twice is called **linear**. We characterise the property of linearity via the number of symbol occurrences. The latter is inductively defined as follows:

Definition 7.4. (Number of symbol occurrences) Let $\mathcal{L} = \langle L, P, V \rangle$ be a formal language. Let $T \in \{L, V, P\}$. The number of symbol occurrences of type T in formula φ is recursively defined as:

$$\begin{aligned} \#\mathrm{occ}(T, s) &:= \text{if } s \in T \text{ then } 1 \text{ else } 0 \\ \#\mathrm{occ}(T, s(\varphi_1, ..., \varphi_n) &:= \begin{cases} 1 + \sum_{k=1}^{n} \#\mathrm{occ}(T, \varphi_k) \text{ if } s \in T \\ \sum_{k=1}^{n} \#\mathrm{occ}(T, \varphi_k) \text{ otherwise} \end{cases} \end{aligned}$$

To check the linearity of a formula φ, we can extract all its parameters and check if its cardinality ($|\Sigma_\varphi|$) equals the number of parameters ($\#\mathrm{occ}(P, \varphi)$) in this formula.

Definition 7.5. (Linearisation) A formula ψ is called **linear** (w.r.t. its parameter) iff each of its parameter occur only once in it (i.e. $\#\mathrm{occ}(P, \psi) = |\Sigma_\varphi|$). If ψ is linear and σ is a symbol morphism for ψ with $\sigma^*(\psi) = \varphi$ then σ is called a **lineariser** and ψ the **linearisation** for φ. In more general settings where $\sigma^*(\psi) \sim \varphi$ holds for a given equivalence relation $\sim$, we call σ a **lineariser** and ψ the **linearisation** for φ **modulo** $\sim$.

We are only interested in equivalence relations between formulae that keep to be equal after formula translations, such that we are compliant with the translation axiom from the definition of entailment system (cf. definition 4.1). Therefore we will consider in the remainder only equivalence relations that are invariant under formula translation (i.e. $\varphi \sim \psi \Rightarrow \sigma^*(\varphi) \sim \sigma^*(y)$), and whenever we talk about equivalence relations we tacitly assume translation invariance.

Our general method for standardisation of formulae (i.e. determining its skeleton) is the agreement that always the least element of an equivalence class stands as its representative element. By the well-ordering theorem[7.5] every non-empty set can be well-ordered: there is an order relation such that every non-empty subset has a least element. In our case the set to be well-ordered is the set of all formulae of our formal language. For this agreement the concrete definition of the order relation is irrelevant until we study concrete algorithms for standardisation (cf. sections below). In this section, though, we assume an arbitrary but fixed ordering of our formulae.

We define the skeleton of a term as its least linearisation and its parametrisation as the corresponding linearising symbol morphism:

Definition 7.6. (Skeleton) Let $\mathcal{E}$ be an equivalence relation on formulae, also denoted by $\sim_{\mathcal{E}}$ as infix operator. The **$\mathcal{E}$-skeleton** of a formula φ, denoted by $\varphi{\downarrow^{\mathcal{E}}}$, is defined as the minimal element of all linearisations for φ modulo $\mathcal{E}$ (i.e. $\min\{\psi \mid \delta^*(\psi) \sim_{\mathcal{E}} \varphi, \psi \text{ linear}\}$). A **$\mathcal{E}$-parametrisation** is a lineariser δ for φ modulo $\mathcal{E}$ such that $\delta^*(\varphi{\downarrow^{\mathcal{E}}}) \sim_{\mathcal{E}} \varphi$.

Our skeleton filter criterion is thus the following:

Theorem 7.7. Let $\sim$ be a length neutral equivalence relation, let φ and ψ formulae and σ a symbol morphism for φ:

$$\sigma^*(\varphi) \sim \psi \quad \Rightarrow \quad \varphi{\downarrow^{\sim}} = \psi{\downarrow^{\sim}}.$$

Proof. Assume $\sigma^*(\varphi) \sim \psi$. By the definition of skeleton we have $(\sigma^*(\varphi)){\downarrow^{\sim}} \sim \psi{\downarrow^{\sim}}$ and and hence $\varphi{\downarrow^{\sim}} = \psi{\downarrow^{\sim}}$ since $\forall\varphi.\, (\sigma^*\varphi){\downarrow^{\sim}} = \varphi{\downarrow^{\sim}}$. □

Having identical skeletons in order to translate one formula to another modulo equality is a necessary condition. If that is given then we need to check whether there are parametrisations for these formulae that can be composed to a single symbol morphism. If that is the case then this composition forms the searched translation:

Theorem 7.8. Let $\sim$ be a length neutral equivalence relation on formulae. Let φ_1 and φ_2 be formulae and δ_1, δ_2 resp. $\sim$-parametrisations (i.e. $\varphi_i \sim \delta_i^*(\varphi_i{\downarrow^{\sim}})$ for $i = 1, 2$). Let σ be a symbol morphism for φ_1 then

$$\sigma = \delta_2 \circ \delta_1^{-1} \quad \wedge \quad \varphi_1{\downarrow^{\sim}} = \varphi_2{\downarrow^{\sim}} \quad \Rightarrow \quad \sigma^*\varphi_1 \sim \varphi_2.$$

Proof. From $\sigma = \delta_2 \circ \delta_1^{-1}$ and lemma 7.3 we can derive $\delta_2 = \delta_2 \circ \delta_1^{-1} \circ \delta_1$ and we derive $\delta_2\varphi_1{\downarrow^{\sim}} = \delta_2\varphi_2{\downarrow^{\sim}}$ from $\varphi_1{\downarrow^{\sim}} = \varphi_2{\downarrow^{\sim}}$. Putting these identities together we have $(\delta_2 \circ \delta_1^{-1} \circ \delta_1)^*(\varphi_1{\downarrow^{\sim}}) = \delta_2^*(\varphi_2{\downarrow^{\sim}})$. By definition $\varphi_i \sim \delta_i(\varphi_i{\downarrow^{\sim}})$ for $i = 1, 2$ this transforms to $(\delta_2 \circ \delta_1^{-1})^*(\varphi_1) \sim \varphi_2$ and finally $\sigma^*\varphi_1 \sim \varphi_2$. □

In the introduction to the AC-renaming problem (cf. section 7.3) we saw that formula abstraction in general leads not just to one, but many parametrisations for a given formula. This is different in the case of the simple renaming problem:

Lemma 7.9. For every formula φ there is exactly one lineariser σ for φ such that $\sigma^*(\varphi{\downarrow}) = \varphi$.

Proof. Suppose σ_1 and σ_2 are both lineariser for φ. That means $\sigma_1^*(\varphi) = \sigma_2^*(\varphi)$ and $\mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2) = \mathcal{S}ymb(\varphi)$. From the structural definition of formula translation we can derive $\sigma_1 = \sigma_2$. □

Hence, we can speak in the simple case of *the* parametrisation for a formula:

7.5. The well-ordering theorem is implied by the axiom of choice and vice versa.

Definition 7.10. (Parametrisation) Let φ be a formula. The **parametrisation** for φ, denoted by $\boldsymbol{\delta_\varphi}$, is the unique lineariser for which $\sigma^*(\varphi\downarrow) = \varphi$ holds.

For the simple renaming problem it is also possible to prove the opposite direction of theorem 7.8: if there is a symbol morphism that translates one formula to the other then this symbol morphism is a composition of the parametrisations of these two formulae:

Theorem 7.11. Let φ_1 and φ_2 be formulae and δ_1, δ_2 their respective parametrisations. If σ is a symbol morphism for φ_1 then

$$\sigma = \delta_2 \circ \delta_1^{-1} \quad \wedge \quad \varphi_1\downarrow = \varphi_2\downarrow \quad \Leftrightarrow \quad \sigma^*\varphi_1 = \varphi_2$$

Proof. The " $\Rightarrow$ " direction is already shown by theorem 7.8. For " $\Leftarrow$ " we start with $\sigma^*\varphi_1 = \varphi_2$ and firstly derive $\varphi_1\downarrow = \varphi_2\downarrow$, since $\forall\varphi.(\sigma^*\varphi)\downarrow^\sim = \varphi\downarrow^\sim$, and secondly from that and $\varphi_i = \delta_i^*(\varphi_i\downarrow)$ $(i = 1, 2)$ we derive $(\sigma \circ \delta_1)^*(\varphi_1\downarrow) = \delta_2^*(\varphi_1\downarrow)$. From the inductive definition of term renamings it follows $\sigma \circ \delta_1 = \delta_2$ since $\mathrm{dom}(\delta_1), \mathrm{dom}(\delta_2) \subseteq \mathcal{S}ymb(\varphi)$. Finally we derive $\sigma = \delta_2 \circ \delta_1^{-1}$ by lemma 7.3. □

So far we have characterised how formula abstraction can be used to find formula translations (optionally) modulo equivalence in theory. Now we want to investigate how they can be practically computed.

7.5 Standardisation Algorithms

For concrete algorithms we first have to specify a concrete term ordering. Then we will describe an algorithm for the simple standardisation followed by the AC-standardisation as important representative for equational standardisation.

7.5.1 Term Ordering

Our term ordering will be a lexicographical ordering. For that we start with a one-to-one transformation between formulae and its string representation.

Definition 7.12. ($\mathcal{L}$-strings) Let s be a symbol, v be a variable, and φ_i terms. We define recursively the **prefix string encoding**, denoted by enc, as:

$$\begin{aligned} \mathrm{enc}(s) &:= s^0 \quad \text{if } s \in \mathcal{S} \\ \mathrm{enc}(s(\varphi_1, ..., \varphi_n)) &:= s^n\,\mathrm{enc}(\varphi_1)\cdots\mathrm{enc}(\varphi_n) \\ \mathrm{enc}(\lambda v.\varphi) &:= \lambda^2 v^0\,\mathrm{enc}(\varphi) \end{aligned}$$

The inverse transformation of prefix encoding is called **prefix string decoding**. We call the set $\mathrm{enc}(\mathcal{L})$ the set of **$\mathcal{L}$-strings**.

For instance the $\mathcal{L}$-term $\forall(\lambda x. = (\circ(x, e), x))$ – which corresponds to the less formal infix notation $\forall x.x \circ e = x$ – corresponds to the $\mathcal{L}$-string $\forall^1\lambda^2 v^0 = {}^2 \circ {}^2 x^0 e^0 x^0$. Since a formula and its string representation is isomorphic, we will identify these objects and speak in the following about a term ordering for a formal language although it is strictly speaking an ordering for its string representation.

Definition 7.13. (Lexicographical Term Ordering) Let $\mathcal{L} = \langle L, P, V \rangle$ be a formal language where L, P, and V are totally ordered symbol sets. We define the total order of all symbols $L \cup P \cup V$, called the **alphabet** of $\mathcal{L}$, as follows:

- $l^i \preccurlyeq p^j \preccurlyeq v^k$ for all $l \in L, p \in P, v \in V$, and $i, j, k \in \mathbb{N}$

- $s^k \preccurlyeq s^l$ iff $k \leqslant l$ for all $s \in P \cup V$ and $k, l \in \mathbb{N}$

For all strings over this alphabet we define the **lexicographical term ordering** $\preccurlyeq$ over $\mathcal{L}$ as the smallest reflexive, antisymmetric, and transitive relation (i.e. ordering) on $\mathcal{L}$-terms that obeys the following condition:

- Let $\varphi, \psi \in \mathcal{L}$ with $\mathrm{enc}(\varphi) = a_1 a_2 \cdots a_m$ and $\mathrm{enc}(\psi) = b_1 b_2 \cdots b_n$:
  $\varphi \preccurlyeq \psi$ iff $a_k \preccurlyeq b_k$ for all $k = 1, 2, ..., \min(m, n)$

The relation $\prec$ and the dual relations $\succ$ and $\succcurlyeq$ are defined based on $\preccurlyeq$ as usual for order relations.

For the examples we will see below, we want to be even more concrete regarding the ordering of parameters and variables: we commit to $P = \{p_1, p_2, p_3, ...\}$ and $V = \{v_1, v_2, v_3, ...\}$, and impose the obvious total order on these sets. By convention, P contains also letters like a, b, or r, but all of them are considered greater than any of the p_i's. Besides that we do not care about how they are ordered (and analogously for V).

We said that a skeleton is a minimal element w.r.t. to a given term ordering. For our concrete lexicographical ordering the term $p_1^2 p_2^1 v_1^0 v_2^0$ is an example for a skeleton, because there is no lineariser that could make this term smaller, in contrast to the terms $p_1^2 p_2^1 v_1^0 v_{\mathbf{3}}^0$ or $p_1^2 p_{\mathbf{1}}^1 v_1^0 v_2^0$.

The following two subsections present the algorithms for standardisation; at first for the simple and then for the equational renaming problem. Their main difference is that the former compute a unique parametrisation (for reasons explained above) where as the second compute (potentially) many parametrisations for a given formula. As for the skeleton, both algorithms compute only one as the skeleton is unique in both cases. The basic idea for the skeleton computation is to process the formula (in its string representation) from left to right where one step goes one symbol further to the right. In each step we try to make the current symbol as small as possible by a symbol morphism whose codomain does not contain any symbol that has already been used in the prefix string that has been processed before. The definitions 7.14 (for the simple case) and 7.18 (for the AC case) define these steps formally. The latter case is more complex in that each step may continue with several variants of the rest of the formula (more precisely: the tail of the formula string) due to the fact that permutations of arguments in AC-terms have to be considered.

7.5.2 Algorithm for Simple Standardisation

We start with a standardisation algorithm for the simple renaming problem.

Definition 7.14. (Standardisation Step) Let σ be a symbol morphism and v_n the maximum variable in $\mathrm{cod}(\sigma)$. We call v_{n+1} the **successor variable** for the symbol morphism σ. Similarly we define **successor parameter**. A pair $\langle \sigma, \varphi \rangle$ consisting of a symbol morphism and a term is called **standardisation state**. Let $\alpha s \omega \in \mathcal{L}$ where α is the standardised prefix and s is a symbol. The **successor state** of $\langle \sigma, \alpha s \omega \rangle$ is defined as:

- $\langle \sigma \cup [p \rightsquigarrow s], \alpha p \omega \rangle$ if s is a parameter and p the successor parameter for σ
- $\langle \sigma \cup [v \rightsquigarrow s], \alpha v \omega \rangle$ if s is a variable, $s \notin \mathrm{cod}(\sigma)$, and v the successor variable for σ
- $\langle \sigma, \alpha s \omega \rangle$ if s is a variable and $s \in \mathrm{cod}(\sigma)$

Note that we have not considered in the definition of the successor state the case where s is a logical symbol, because if s were a logical symbol then α could not be a standardised prefix, since αs is standardised for logical symbols whenever α is standardised.

Formula abstraction starts with an identity symbol morphism and a string without standardised prefix and applies the standardisation steps until the formula is finally a skeleton: let φ_0 be the input formula containing n symbols; let $\sigma_0 = []$ be the identity symbol matching. Let $\langle \sigma_{k+1}, \varphi_{k+1} \rangle$ be the successor state of $\langle \sigma_k, \varphi_k \rangle$ starting with $\langle \sigma_0, \varphi_0 \rangle$. Then the skeleton of φ_0 is φ_n and its parametrisation is σ_n from the n-th standardisation state $\langle \sigma_n, \varphi_n \rangle$. Obviously, for a formula of n symbol occurrences this standardisation process reaches the end of the string after n steps. Let us illustrate this with a little example:

Example 7.15. Consider the formula $\varphi_0 := \forall x,\ y.R(f(x),\ f(y))$. Its standardisation process:

	formula string:	symbol matching:
$\varphi_0 =$	$\forall^3 x^0 y^0 R^2 f^1 x^0 f^1 y^0$	$\sigma_0 = []$
$\varphi_1 =$	$\forall^3 v_1^0 y^0 R^2 f^1 x^0 f^1 y^0$	$\sigma_1 = [v_1 \rightsquigarrow x\,]$
$\varphi_2 =$	$\forall^3 v_1^0 v_2^0 R^2 f^1 x^0 f^1 y^0$	$\sigma_2 = [v_1 \rightsquigarrow x, v_2 \rightsquigarrow y]$
$\varphi_3 =$	$\forall^3 v_1^0 v_2^0 p_1^2 f^1 x^0 f^1 y^0$	$\sigma_3 = [v_1 \rightsquigarrow x, v_2 \rightsquigarrow y\,, p_1 \rightsquigarrow R]$
$\varphi_4 =$	$\forall^3 v_1^0 v_2^0 p_1^2 p_2^1 x^0 f^1 y^0$	$\sigma_4 = [v_1 \rightsquigarrow x, v_2 \rightsquigarrow y\,, p_1 \rightsquigarrow R, p_2 \rightsquigarrow f]$
$\varphi_5 =$	$\forall^3 v_1^0 v_2^0 p_1^2 p_2^1 v_1^0 f^1 y^0$	$\sigma_5 = [v_1 \rightsquigarrow x, v_2 \rightsquigarrow y\,, p_1 \rightsquigarrow R, p_2 \rightsquigarrow f]$
$\varphi_6 =$	$\forall^3 v_1^0 v_2^0 p_1^2 p_2^1 v_1^0 p_3^1 y^0$	$\sigma_6 = [v_1 \rightsquigarrow x, v_2 \rightsquigarrow y\,, p_1 \rightsquigarrow R, p_2 \rightsquigarrow f\,, p_3 \rightsquigarrow f]$
$\varphi_7 =$	$\forall^3 v_1^0 v_2^0 p_1^2 p_2^1 v_1^0 p_3^1 v_2^0$	$\sigma_7 = [v_1 \rightsquigarrow x, v_2 \rightsquigarrow y\,, p_1 \rightsquigarrow R, p_2 \rightsquigarrow f\,, p_3 \rightsquigarrow f]$

The skeleton of φ_0 is φ_7 and σ_7 its parametrisation.

This algorithm is straightforward, and we have formalised it with the definitions above mainly to ease the understanding of the more involved process of AC-standardisation that is similar in the concept of successor state.

We want to conclude this subsection with an example of two formulae where one can be mapped on the other, but not the other way around, because the parametrisations fit together only in one direction.

Example 7.16. Consider a term $\varphi := \forall x,\ y.\ >(f(x),\ g(\ -\ (x,\ y)))$ and a second term $\psi := \forall M, N.\ >(\#(M), \#(/(M, N)))$. Then we have:

$$\begin{aligned} \varphi\downarrow = \psi\downarrow &= \forall v_1, v_2.a_1(a_2(v_1), a_3(a_4(v_1, v_2))) \\ \delta_\varphi &= [a_1 \rightsquigarrow >\,, a_2 \rightsquigarrow f, a_3 \rightsquigarrow g, a_4 \rightsquigarrow -\,] \\ \delta_\psi &= [a_1 \rightsquigarrow >\,, a_2 \rightsquigarrow \#, a_3 \rightsquigarrow \#, a_4 \rightsquigarrow /] \\ \sigma := \delta_\psi \circ {\delta_\varphi}^{-1} &= [> \rightsquigarrow >\,, f \rightsquigarrow \#, g \rightsquigarrow \#, - \rightsquigarrow /] \\ \sigma^*(\varphi) &=_\alpha \psi \end{aligned}$$

However, $\delta_\varphi \circ \delta_\psi^{-1}$ is obviously not a function, hence we could not match ψ on φ.

7.5.3 Algorithm for AC-Standardisation

Equality modulo associativity and commutativity, denoted by "$\sim$", is defined by the two laws:

$$\begin{aligned} (\varphi + \psi) &\sim (\psi + \varphi) \\ \varphi + (\psi + \phi) &\sim (\varphi + \psi) + \phi \end{aligned}$$

Any operator "+" satifying these two laws is called AC-operator. Given such an AC-operator, it is common to omit brackets: we can write $a_1 + a_2 + ... + a_n$ instead of any kind of nested brackets. With this convention in mind the following equality holds for any $1 \leqslant i, j \leqslant n$:

$$a_1 + ... + a_i + ... + a_j + ... + a_n \sim a_1 + ... + a_j + ... + a_i + ... + a_n$$

That means that we can consider an binary AC-operator as an n-ary opertor and that we can change the argument order of such AC-operators arbitrarily.

AC-standardisation is not that straightforward, as it has to regard implicitly all possible permutations of arguments of AC-operators. In our approach we do not consider all possible AC-permutations in each standardisation step: since we standardise in each step only the front symbol, it is irrelevant which AC-permutation we have in the tail. Let us investigate this claim by an example in the tree representation. Consider the formula $p_1(f(a,\, b),\, g(c,\, d))$ (whose first symbol is already standardised) and assume that p_1, f, and g are AC-operators. The only symbols to be considered for the following standardisation step are obviously f and g. To standardise g we had to get it to the front position at first. This can be achieved by various AC-permutations. The simplest of them returns $p_1(g(c,\, d),\, f(a,\, b))$ and $p_1(g(d,\, c),\, f(a,\, b))$, $p_1(g(d,\, c),\, f(b,\, a))$, and $p_1(g(c,\, d),\, f(b,\, a))$ are the remaining possible results of AC-permutation. For standardising the front symbol g, it does not matter which of these variants we take. Translating the above four variants of formula trees into string representations yields $p_1^2g^2c^0d^0f^2a^0b^0$, $p_1^2g^2d^0c^0f^2a^0b^0$, $p_1^2g^2d^0c^0f^2b^0a^0$, and $p_1^2g^2c^0a^0f^2b^0a^0$ which makes clear that we do not care about the tail behind $p_1^2g^2$. Hence, we take the simplest AC-permutation, namely a cyclic permutation of the AC-operator's arguments. The conventional cyclic permutation can be defined as:

$$\pi_k^n(m) = \begin{cases} m-k & \text{if } m > k \\ n-k+m & \text{otherwise} \end{cases} \qquad \text{for } m, k, n \in \mathbb{N} \text{ and } m, k \leqslant n.$$

Here is a point-wise application on a list:

$$\pi_5^7(1,2,3,4,5,6,7) = (3,4,5,6,7,1,2)$$

We extend this concept specifically to formula trees with AC-terms:

Definition 7.17. (AC-permutation Set) Let $\alpha s\, c_1 \cdots c_n \omega \in \mathcal{P}$, where α is the standardised prefix and s is a symbol. The **AC-permutation set** is defined as

$$\Pi(\alpha s\, c_1 \cdots c_n \omega) := \begin{cases} \{\alpha s\, c_{\pi_k^n(1)} \cdots c_{\pi_k^n(n)} \omega \,|\, k = 1, ..., n\} & \text{if } s \text{ an AC-operator} \\ \{\alpha s\, c_1 \cdots c_n \omega\} & \text{otherwise} \end{cases}$$

Informally spoken, the set $\Pi(\varphi)$ is a collection of all admissible cyclic permutations of the left-most non-standardised subtree. If the operator of that subtree is not AC then it contains only one element.

For the overall AC-standardisation process, we have to consider all strings from the AC-permutation set when we apply the next standardisation step – we therefore call it **distribution step**. Thereby we span a tree of standardisation states as nodes – which we call the **standardisation tree**. Every level of this tree represents the result of a standardisation step. Finally, the states at the leaves of this tree only contain states with completely standardised strings, as in each step another one of the finite number of symbols in the string is standardised. Moreover, this process generates all standardised strings that are possible regarding AC-permutations.

However, they will not all be equal with respect to our term ordering; only some of them will be minimal and thus AC-standardised. Let us illustrate this by an example on the simple formula $\varphi := R(f(a), b)$ where R is an AC-operator. The standardisation tree of φ (as string) looks like this:

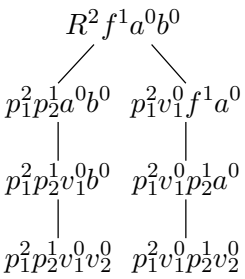

Figure 7.2. Standardisation tree of the formula $R(f(a), b)$ with R as AC-operator. Only the left leaf is an AC-skeleton, as it is the smallest of all leaves.

In this example the left leaf of the standardisation tree is a smaller skeleton than that of the right leaf. Concerning efficiency, it was not necessary to standardise both branches down to the leaves: already at the second level, we can see that the left branch will end up in a smaller skeleton than the right branch, because its standardised prefix is already smaller at the second level (because $p_1^2 p_2^1 \prec p_1^2 v_1^0$). So the general idea to make the standardisation process more efficient is to **prune** the tree after every distribution step as much as possible, i.e. to cut off all branches whose standardised prefix is smaller than that of some other branch. Our overall algorithm thus is an alternation of distribution and pruning of the standardisation tree along its growth:

Definition 7.18. (AC-Standardisation Algorithm) A **standardisation state** is a pair $\langle \sigma, \varphi \rangle$ of a symbol morphism and a formula. Let Ω be a set of standardisation states. The **distribution step** is a function d defined as:

$$d(\Omega) := \{\text{successor state of } \langle \sigma, \varphi \rangle \mid \varphi \in \Pi(\psi),\ \langle \sigma, \psi \rangle \in \Omega\}$$

The **pruning step** is a function p defined as:

$$p(\Omega) := \{\langle \sigma, \varphi \rangle \mid \forall \langle \sigma', \varphi' \rangle \in \Omega.\ \mathrm{pre}(\varphi) \preccurlyeq \mathrm{pre}(\varphi')\}$$

Let σ_0 be the identity symbol morphism and φ_0 the input formula with n symbol occurrences. $\Omega_0 = \{\langle \sigma_0, \varphi_0 \rangle\}$ is the **initial state set** and $\Omega_{k+1} = p(d(\Omega_k))$ the **AC-standardisation step**. The **final state set** is Ω_n.

Every distribution step only generates finitely many successor states (limited by the finite number of possible AC-permutations), and only finitely many AC-standardisation steps are applied (limited by the number of symbol occurrences in the input formula). Hence the final state set is computable. The alternation of distribution and pruning guarantees that, after the kth step, we have all possible smallest standardised prefixes in the formulae of Ω_k. In the final state Ω_n the formulae coincide with the standardised prefix, hence, they are all identical and minimal – i.e. they are the AC-skeleton.

The depth of the standardisation tree is determined by the number n of symbol occurrences in φ_0. Its breadth is limited by the number of AC-operators contained in φ_0. In the worst case the breadth is n times the number of AC-operators times the factorial of the number of its arguments. Formulae leading to the worst case, however, are very unlikely, as they must have a *homogeneous structure*: two subformulae being an argument of a common AC-operator in a formula must have the same skeleton.

An instance of such a worst-case formula is, for instance, a homogeneous formula like $P(R(a,b,c),Q(e,f,g),S(h,i,j))$ where $P, R, Q,$ and S are AC-operators. The depth of its standardisation tree would be 13 and its breadth 36. In explored theories formulae usually do not have just one but many occurrences of a parameter, and their internal structure is hardly as homogeneous as in worst-case formulae. As soon as two arguments of an AC-term are terms of different arity pruning cuts off one of the AC-permutations (cf. the example $R(f(a),b)$ above). Moreover, multiple occurrences of variables – a natural fact for formulae of explored theories – also gives rise for pruning. For instance, the standardisation of the AC-term $R(x,\ y,\ x)$ without pruning would lead to the three standard forms $p_1(v_1,\ v_1,\ v_2)$, $p_1(v_1,\ v_2,\ v_1)$, and $p_1(v_2,\ v_1,\ v_1)$, but only the first variant would survive pruning.

We want to illustrate the AC-standardisation algorithm on an example input formula $\forall x,\ y.R(f(x),\ g(x)) \vee Q(h(x),\ h(y))$. Since this formula contains 12 symbol occurrences, 12 AC-standardisation steps are needed.

step	state	symbol morphism	formula
0	S_o	$[]_V\ []_P$	$\forall^3 x^0 y^0 \vee {}^2 R^2 f^1 x^0 g^1 x^0 Q^2 h^1 x^0 h^1 y^0$
2	S_2	$[x,y]_V\ []_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 R^2 f^1 x^0 g^1 x^0 Q^2 h^1 x^0 h^1 y^0$
3	$S_{3.1}$	$[x,y]_V\ [R]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 f^1 x^0 g^1 x^0 Q^2 h^1 x^0 h^1 y^0$
	$S_{3.2}$	$[x,y]_V\ [Q]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 h^1 x^0 h^1 y^0 R^2 f^1 x^0 g^1 x^0$
5	$S_{5.1}$	$[x,y]_V\ [R,f]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 g^1 x^0 Q^2 h^1 x^0 h^1 y^0$
	$S_{5.2}$	$[x,y]_V\ [R,g]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 f^1 x^0 Q^2 h^1 x^0 h^1 y^0$
	$S_{5.3}$	$[x,y]_V\ [Q,h]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 h^1 y^0 R^2 f^1 x^0 g^1 x^0$
	$S_{5.4}$	$[x,y]_V\ [Q,h]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_2^0 h^1 x^0 R^2 f^1 x^0 g^1 x^0$
7	$S_{7.1}$	$[x,y]_V\ [R,f,g]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 p_3^1 v_1^0 Q^2 h^1 x^0 h^1 y^0$
	$S_{7.2}$	$[x,y]_V\ [R,g,f]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 p_3^1 v_1^0 Q^2 h^1 x^0 h^1 y^0$
	$S_{7.3}$	$[x,y]_V\ [Q,h,h]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 p_3^1 v_2^0 R^2 f^1 x^0 g^1 x^0$
12	$S_{12.1}$	$[x,y]_V\ [R,f,g,h,h]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 p_3^1 v_1^0 p_4^2 p_5^1 v_1^0 h^1 v_2^0$
	$S_{12.2}$	$[x,y]_V\ [R,g,f,h,h]_P$	$\forall^3 v_1^0 v_2^0 \vee {}^2 p_1^2 p_2^1 v_1^0 p_3^1 v_1^0 p_4^2 p_5^1 v_1^0 h^1 v_2^0$

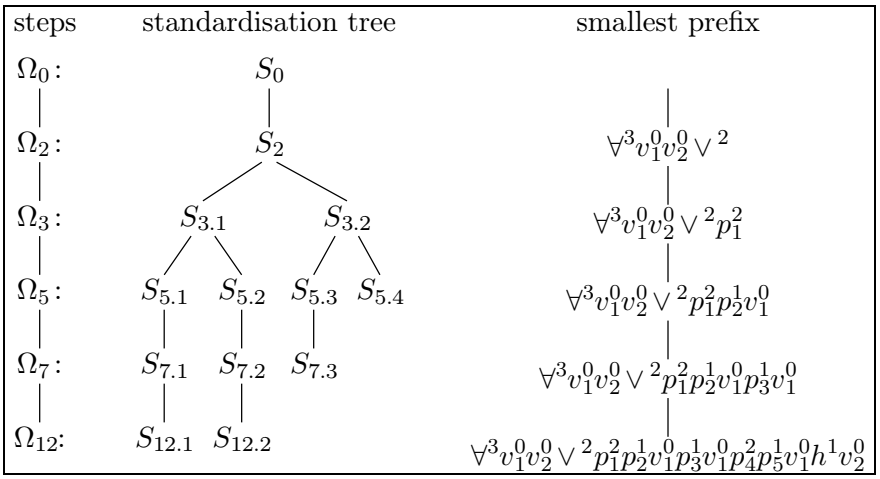

Table 7.1. AC-standardisation procedure: an initial formula $\forall x,\ y.R(f(x),\ g(x)) \wedge Q(h(x),\ h(y))$ is AC-standardised in 12 steps. The nodes in the tree refer to the states in the upper table. The standardisation tree is condensed: only those steps with pruning or branching are displayed.

Finally we present an example of two formulae each having two parametrizations that can be composed to two one-to-one translations between these formulae.

Example 7.19. Consider the following two lemmata – the first from set theory and the second from number theory:

$$\begin{aligned}\varphi &:= \forall A, B.\ A \cup B = A \Rightarrow A \cap B = B\\ \psi &:= \forall m, n.\ m = \mathrm{lcm}(m,n) \Rightarrow n = \gcd(n,m)\end{aligned}$$

where "lcm" is the least common multiple and "gcd" the greatest common divisor. Both are AC-operators, as well as equality, the union and intersection from set theory, and the disjunction. Obviously these formulae have different (simple) skeletons (presented in prefix notation):

$$\begin{aligned}\varphi{\downarrow} &= \forall v_1, v_2. \Rightarrow (=(p_1(v_1,v_2), v_1), =(p_2(v_1,v_2), v_2))\\ \psi{\downarrow} &= \forall v_1, v_2. \Rightarrow (=(v_1, p_1(v_1,v_2)), =(v_2, p_2(v_2,v_1))\end{aligned}$$

However, they have the same AC-skeleton $\varphi{\downarrow}^{\mathrm{AC}} = \psi{\downarrow}^{\mathrm{AC}} = \varphi{\downarrow}$, assuming a lexicographical term ordering as in the last example. To each of these AC-skeletons, there are two AC-parametrizations associated:

$$\begin{aligned}\delta_{1,\varphi} &= [p_1 \rightsquigarrow \cup\,, p_2 \rightsquigarrow \cap\,]\\ \delta_{2,\varphi} &= [p_1 \rightsquigarrow \cap\,, p_2 \rightsquigarrow \cup\,]\\ \delta_{1,\psi} &= [p_1 \rightsquigarrow \mathrm{lcm}, p_2 \rightsquigarrow \gcd]\\ \delta_{2,\psi} &= [p_1 \rightsquigarrow \gcd, p_2 \rightsquigarrow \mathrm{lcm}]\end{aligned}$$

From these we can construct all possible AC-symbol morphisms σ_i for the AC-matching problem $\sigma_i^*(\varphi) \sim_\alpha \psi$:

$$\begin{aligned}\sigma_1 := \delta_{1,\psi} \circ \delta_{1,\varphi}^{-1} = \delta_{2,\psi} \circ \delta_{2,\varphi}^{-1} &= [\cup \rightsquigarrow \mathrm{lcm}, \cap \rightsquigarrow \gcd]\\ \sigma_2 := \delta_{1,\psi} \circ \delta_{2,\varphi}^{-1} = \delta_{2,\psi} \circ \delta_{1,\varphi}^{-1} &= [\cap \rightsquigarrow \mathrm{lcm}, \cup \rightsquigarrow \gcd].\end{aligned}$$

Chapter 8
Normalization

In this section, we want to further extend the equivalence class modulo which we rename formulae. As opposed to the AC-matching problem, which is not tied to a specific logic, we will now exploit logic specific equivalences, i.e. formulae equivalences that hold due to logical laws involving certain logical connectives and quantifiers. For instance

$$\begin{aligned} a \wedge b \Rightarrow c &\;\Leftrightarrow\; a \Rightarrow (b \Rightarrow c) \\ \varphi \Rightarrow \exists x.\psi &\;\Leftrightarrow\; \exists x.\varphi \Rightarrow \psi \end{aligned}$$

Such equivalences[8.1] hold in all theories, and many of them are considered as trivial from a mathematical point of view, as they can be proven mechanically. As a consequence, these trivial equivalences lead to trivial possible variants of mathematical statements and the mentioned problem of the no triviality maxim (cf. section 7.2). If three formalists are asked to formalize a uniformly continuous function f, then they may come up with these three variants:

1. $\forall \varepsilon.\varepsilon > 0 \Rightarrow \exists \delta.\forall x.\forall y.0 < |x-y| \wedge |x-y| < \delta \Rightarrow |f(x)-f(y)| < \varepsilon$
2. $\forall \varepsilon.\exists \delta.\forall x, y.\varepsilon > 0 \Rightarrow (0 < |x-y| \wedge |x-y| < \delta \Rightarrow |f(x)-f(y)| < \varepsilon)$
3. $\forall \varepsilon.\exists \delta.\forall x, y.\varepsilon > 0 \wedge |x-y| < \delta \wedge 0 < |x-y| \Rightarrow |f(x)-f(y)| < \varepsilon$

And there are many other variants easily derivable by applying basic logical laws, e.g.:

4. $\neg\exists \varepsilon.\, \varepsilon > 0 \wedge \forall \delta.\exists x, y.\neg(\,|f(x)-f(y)| < \varepsilon) \wedge 0 < |x-y| \wedge |x-y| < \delta$

This latter variant shows how much formulae can deviate from each other by applying many simple logical transformation rules. Probably, even the trained formalist needs a second to recognize the equivalence of this formula with the three upper formulae. Nevertheless, it would be considered as a trivial variant and thus – following the no triviality maxim – would not be considered worth mentioning in an explored theory. In fact, in an explored theory it is much more likely to find one of the first three variants than the fourth variant. Mathematicians and formalists certainly have some preference among all possible trivial variants. In the given example, the last variant is probably not a preferred formalization, as it starts with a negation, whereas mathematicians prefer to formulate assertions positively. But still, the preferences leave space enough for some redundant variants like the first three above.

The goal of this chapter is to develop methods for narrowing down the space of trivial variants via normalization.

8.1 Simple Logical Language

The trivially equivalent variants of formulae we are considering depend on certain logical laws, hence we need to define the language and concrete logical symbols occurring in it. For that we take the general definition of an untyped logical language as a basis as, introduced in chapter 6.

8.1. In the second equivalence we assume that φ does not contain x.

Definition 8.1. (Simple Logical Language) We define a **simple logical language** as a logical language $\mathcal{L} = \langle L, P, V \rangle$, where the set of logical symbols L contains in particular the symbols: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$, and the quantifiers $\forall, \exists$. A quantifier must have a binding term as sole argument. For convenience we write $\forall x.\varphi$ instead of $\forall(\lambda x.\varphi)$ and analogously for $\exists$.

We make three additional assumptions about the formulae in our language:

1. **closed form**: they are in closed form, i.e. all variables are bound by some quantifier,
2. **unique binding**: variables must be different in different binding occurrences, and
3. **non-empty scope**: every variable that occurs in a binding must also occur in the scope of this binding.

These assumptions do not restrict the expressivity of the language for the well-known reasons

1. closed form: a non-closed formula has the semantics of being implicitly universally quantified, so it can be closed by making this quantification explicit.
2. unique binding: renaming bound variables by introducing fresh variables never changes the semantics of a formula, hence we can rename each bound variable by a fresh variable, thus enforcing different variables for different binding occurrences.
3. non-empty scope: removing a quantification over a variable v, where v does not occur in the quantification scope, does not change the semantics of that formula[8.2].

8.2 Preliminaries from Term Rewriting Theory

Our main technique – *normalization* – has its origin in *term rewriting*. This subsection provides some useful elementary results from term rewriting taken from [22].

Definition 8.2. (Subterm at Position) Let φ be a formula, let v be a variable, and let s be a symbol. The **subterm** of φ at **position** p, denoted by $\varphi|_p$, is defined by induction on the length of p (where ε denotes the empty string or **root position**):

$$\begin{aligned}
\varphi|_\epsilon &:= \varphi \\
s(\varphi_1, ..., \varphi_n)|_0 &:= s \\
s(\varphi_1, ..., \varphi_n)|_{iq} &:= \varphi_i|_q \quad \text{if } 0 < i \leqslant n \\
\lambda v.\varphi|_0 &:= \lambda \\
\lambda v.\varphi|_1 &:= v \\
\lambda v.\varphi|_{2q} &:= \varphi|_q
\end{aligned}$$

A position p is called **lambda, variable, parameter position** if $\varphi|_p$ is λ, a variable, or a parameter, respectively, and we call it **symbol position** if it is one of those positions.

8.2. Thus we dispense with logics that admit empty carrier sets in models.

We need these notions for very local manipulations inside terms. In particular we want to be able to describe local symbol mappings, called **replacements**, where we want to rename a symbol only at a certain position without changing the same symbol at other positions.

Definition 8.3. (Replacement at Position) Let φ and φ_i be formulae; let v and w be variables; let s and s' be symbols. We denote by $\varphi[\psi]_p$ the term that is obtained from φ by **replacing the subterm at position** p **by** ψ, i.e.

$$\begin{aligned}
\varphi[\psi]_\epsilon &:= \varphi \\
s(\varphi_1, ..., \varphi_n)[s']_0 &:= s'(\varphi_1, ..., \varphi_n) \\
s(\varphi_1, ..., \varphi_n)[\psi]_{iq} &:= s(\varphi_1, ..., \varphi_i[\psi]_q, ..., \varphi_n) \\
\lambda v.\varphi[w]_1 &:= \lambda w.\varphi \\
\lambda v.\varphi[\psi]_2 &:= \lambda v.\psi
\end{aligned}$$

Lemma 8.4. Let φ and ψ be formulae and let σ be a symbol mapping defined on all (non-logical) symbols occurring in φ and ψ. Then $\sigma^*(\varphi[\psi]_p) = (\sigma^*(\varphi))[\sigma^*(\psi)]_p$ for any symbol position p.

Proof. We prove inductively on the structure of formula. Base case: assume our formula φ is a non-logical symbol s, then its only position is 0. Hence we have $\sigma^*(s[\psi]_0) = \sigma^*(\psi)$. Inductive step: assume our φ is of the form $s(\varphi_1, ..., \varphi_i, ..., \varphi_n)$ and our induction hypothesis is $\sigma^*(\varphi_i[\psi]_p) = (\sigma^*(\varphi_i))[\sigma^*(\psi)]_p$:

$$\begin{aligned}
\sigma^*(\varphi[\psi]_{ip}) &= \sigma^*(s(\varphi_1, ..., \varphi_i, ..., \varphi_n)[\psi]_{ip}) \\
&= \sigma^*(s(\varphi_1, ..., \varphi_i[\psi]_p, ..., \varphi_n)) \\
&= (\sigma(s))(\sigma^*(\varphi_1), ..., \sigma^*(\varphi_i[\psi]_p), ..., \sigma^*(\varphi_n)) \\
&= (\sigma(s))(\sigma^*(\varphi_1), ..., (\sigma^*(\varphi_i))[\sigma^*(\psi)]_p, ..., \sigma^*(\varphi_n)) \\
&= (\sigma^*(\varphi))[\sigma^*(\psi)]_{ip}
\end{aligned}$$

In the last case, where φ is an λ-expression, the argument is analogous. □

Definition 8.5. (Rewrite System) An **abstract rewrite system** is a pair $\mathcal{R} = \langle A, \rightarrow \rangle$, where the **rewrite relation** $\rightarrow$ is a binary relation on the set A, i.e. $\rightarrow \subseteq A \times A$. Instead of $(a, b) \in \rightarrow$, we write $a \rightarrow b$ or $b \leftarrow a$ and $\leftrightarrow := \rightarrow \cup \leftarrow$ is the symmetric closure. The reflexive transitive closure of $\rightarrow$ ($\leftrightarrow$) is denoted by $\overset{*}{\rightarrow}$ ($\overset{*}{\leftrightarrow}$).

Alternatively we can view the reflexive transitive closure $a \overset{*}{\rightarrow} b$ as a chain of indefinite length: $a \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow b$. We read this as "$a$ rewrites to b with the intermediate values $a_1, a_2, ...$". The relation $\overset{*}{\leftrightarrow}$ is also the least equivalence relation containing $\rightarrow$.

Let us add some further terminology:

Definition 8.6. (Properties of a Rewrite System) Let $\mathcal{R} = \langle A, \rightarrow \rangle$ be an abstract rewrite system, and $x, y \in A$.

1. x is **reducible** iff there is a y such that $x \rightarrow y$.
2. x is in **normal form** iff it is not reducible.
3. y is **a normal form of** x iff $x \overset{*}{\rightarrow} y$ and y is in normal form. If x has a uniquely determined normal form, the latter is denoted by $x\!\downarrow$.
4. x and y are **joinable** iff there is a z such that $x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$, in which case we write $x \downarrow y$.

$\mathcal{R}$ is called

5. **Church-Rosser** iff $x \overset{*}{\leftrightarrow} y \Rightarrow x \downarrow y$,
6. **confluent** iff $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2 \Rightarrow y_1 \downarrow y_2$,
7. **locally confluent** iff $y_1 \leftarrow x \rightarrow y_2 \Rightarrow y_1 \downarrow y_2$,
8. **terminating** iff there is no infinite descending chain $a_0 \rightarrow a_1 \rightarrow \ldots$,
9. **normalizing** iff every element has a normal form,
10. **convergent** iff it is both confluent and terminating.

Our general goal is to find rewrite systems that implicitly define interesting equivalence classes of formulae and simultaneously describe an effective procedure to answer the question whether two formulae are in the same equivalence class, i.e. we want to be able to answer the question $\varphi \overset{*}{\leftrightarrow} \psi$, given a finite amount of rewrite relations $\varphi_i \rightarrow \psi_i$. The general theory of rewrite systems already provides useful results for abstract rewrite systems which we will use for our term rewrite systems. First of all, we notice directly from the definition that every terminating rewrite system is also normalizing[8.3]. The central theorem (cf. [22], page 12) for us is the following:

Theorem 8.7. If $\rightarrow$ is convergent then $x \overset{*}{\leftrightarrow} y \Leftrightarrow x\downarrow = y\downarrow$.

The effective method to answer the question $x \overset{*}{\leftrightarrow} y$ is hence to answer the easier question $x\downarrow = y\downarrow$ provided $\rightarrow$ is convergent. The latter question, $x\downarrow = y\downarrow$, is easier to answer as a normal form is essentially nothing else than the finite application of rewrite relations. We only have to prove once that our rewrite system is convergent, i.e. confluent and terminating. However, this is the actual difficulty. Concerning confluence, there exists another theorem, known as Newman's Lemma, that helps us:

Theorem 8.8. If $\rightarrow$ is terminating and locally confluent, then it is confluent.

The termination problem of an abstract rewrite system $\langle A, \rightarrow \rangle$, on the other hand, is generally tackled by **well-founded induction**, which is formally expressed by the following inference rule:

$$\frac{\forall x \in A.\, (\forall y \in A.\, x \overset{*}{\rightarrow} y \Rightarrow P(y)) \Rightarrow P(x)}{\forall x \in A.\, P(x)} \quad \text{(WFI)}$$

where P is some property of elements of A. The well-founded induction (in computer science literature known as **Noetherian induction**) is an alternative characterization for terminating rewrite systems:

Theorem 8.9. WFI holds iff $\rightarrow$ terminates.

The most basic method for proving termination of some $\langle A, \rightarrow \rangle$ is to embed it into another abstract rewrite system $\langle B, > \rangle$ that is known to terminate. This requires a monotone mapping $\mu\colon A \rightarrow B$ (where monotone means $x \rightarrow y \Rightarrow \mu(x) > \mu(y)$). Now $\rightarrow$ terminates because an infinite chain $x_0 \rightarrow x_1 \rightarrow \cdots$ would induce an infinite chain $\mu(x_0) > \mu(y) > \cdots$. The mapping μ is often called the **measure function**.

8.3. Note, the opposite does not hold: consider a rewrite system for which $n \rightarrow 1$ and $n+2 \rightarrow n+3$ for $n \in \mathbb{N}$ holds.

So far, we have compiled useful results, which hold for any kind of rewrite systems. Our concern are term rewrite systems, which we want to define now. A basic characteristic of term rewriting is that we only need a finite number of term pairs, called **rewrite rules**, to induce an infinite relation, i.e. our term rewrite system, which in turn induces an infinite equivalence class of terms (via the above mentioned closure). We start with the rewrite rule definition.

Definition 8.10. (Rewrite Rules) A **rewrite rule** is a pair (φ, ψ) of formulae from $\mathcal{L}$, written as $\varphi \rightsquigarrow \psi$, whereby φ must not be a variable and all variables contained in ψ must be present in φ too. φ is called the **left hand side** (**lhs**) and ψ the **right hand side** (**rhs**) of the rewrite rule. A **constrained rewrite rule** is a rewrite rule $\varphi \rightsquigarrow \psi$ with an associated constraint on the formulae φ and ψ.

A term rewrite system is inductively defined on rewrite rules via substitutions, which are defined as follows:

Definition 8.11. (Substitution) Let $\mathcal{L} = \langle L, P, V \rangle$ be a simple logical language and let $\mathcal{L}$ also denote the set of $\mathcal{L}$–terms. A **substitution** is a mapping $\sigma\colon V \to \mathcal{L}$ such that $\sigma(v) \neq v$ for only finitely many $v \in V$. We extend substitution inductively to term mappings, denoted by σ^*:

$$\begin{aligned} \sigma^*(s) &:= \sigma(s) \quad \text{if } v \in \mathcal{S} \\ \sigma^*(s(\varphi_1, ..., \varphi_n)) &:= \sigma^*(s)(\sigma^*(\varphi_1), ..., \sigma^*(\varphi_n)) \\ \sigma^*(\lambda v.\varphi) &:= \lambda(v).\sigma^*(\varphi) \end{aligned}$$

Note, the last line of this inductive definition is slightly different from most definitions found in λ-calculus (e.g. in [31]). There, case distinctions are made depending on the value of $\sigma^*(v)$. We avoid these case distinctions since 1) our closed form and unique binding constraints imposed on our language $\mathcal{L}$ and 2) we impose extra constraints on rewrite rules. We will explicate this below, where we need rewrite rules involving quantified expressions. Now we define the term rewrite system, which is an instance of the abstract rewrite system.

Definition 8.12. (Term Rewrite System) Let R be a set of (possibly constrained) rewrite rules. A **term rewrite system** is a relation $\rightarrow_R \subseteq \mathcal{L} \times \mathcal{L}$ defined as

$$\varphi \rightarrow_R \psi \;\text{ iff }\; \exists (l, r) \in R, p, \sigma.\ \varphi|_p = \sigma^*(l) \text{ and } \psi = \varphi[\sigma^*(r)]_p$$

where p is a valid position in φ and σ is an substitution.

A little example should demonstrate this technical definition:

Example 8.13. Consider the formula $\varphi := f(i(e), f(e, e))$ and the set of rewrite rules

$$R = [f(x, f(y, z)) \rightsquigarrow f(f(x, y), z),\ \ f(e, x) \rightsquigarrow x,\ \ f(i(x), x) \rightsquigarrow e]$$

Applying $\sigma_1 := [x \rightsquigarrow i(e), y \rightsquigarrow e, z \rightsquigarrow e]$ on the lhs of the first rewrite rule gives φ which, hence rewriting (at root position of) φ yields the rhs $\varphi_1 := f(f(i(e), e), e)$. Next we can apply the third rewrite rule at position $p = 1$ of φ_1 with $\sigma_2 := [x \rightsquigarrow e]$ which gives $\varphi_2 := f(e, e)$. Finally, we apply the second rewrite (at the root position of) φ_2 with the same σ_2 yielding e. Altogether, we get this rewrite chain ending with a normal form:

$$f(i(e), f(e, e)) \rightarrow f(f(i(e), e), e) \rightarrow f(e, e) \rightarrow e$$

Let us return to our prior goal, i.e. showing the convergence of a term rewriting system. For termination, we will use intuitive measure functions, which will make termination evident, as we will see below in the concrete situation. For local confluence, we want to make use of a general result from term rewriting, which is based on **critical pairs of rewrite rules**, i.e. when two rewrite rules are simultaneously applicable inside a certain term where the lhs of the first rewrite rule is a subterm of the lhs of the second rewrite rule. We also say in this case that such rewrite rules **interfere**. Consider, for instance, the rewrite rules from the above example and an initial term $\varphi := f(e, f(i(e), e))$. Here, we could apply again the first rewrite rule at root position of φ, but also the third rewrite rule on $\varphi|_1$. Critical pairs of rewrite rules are problematic for showing local confluence. The following theorem provides a sufficient condition for local confluence. Beforehand, we need a proper definition of critical pair.

Definition 8.14. (Critical Pair) Two $l_i \rightsquigarrow r_i$ rewrite rules $(i = 1, 2)$ are called **critical** iff there is a substitution σ and a non-variable position p in l_1 such that $\sigma^*(l_1|_p) = l_2$. The pair $\langle \sigma^*(r_1),\ \sigma^*(l_1)[\sigma^*(r_2)] \rangle$ is called the **critical pair** and $\sigma^*(l_1)$ the **critical overlap**.

Theorem 8.15. (Critical Pair) A term rewriting system is locally confluent iff all its critical pairs are joinable (cf. [22], p.140).

Let us summarize as a result of all the preceding theorems the sufficient conditions to show convergence of a term rewriting:

Corollary 8.16. A term rewriting system is convergent iff it is terminating and all its critical pairs are joinable.

8.3 Overview of Normalization Steps

With the provided background in term rewriting we are now prepared to look at our concrete equivalence classes expressed by rewrite rules. For readability we will not stick to prefix notation as strictly as required by the definition of our simple logical language, but prefer infix notation for all (binary) logical connectives. Moreover, as already mentioned, we prefer the usual quantification notation $Q\, x.\varphi$ instead of $Q\,(\lambda x.\varphi)$ for $Q = \forall$, $\exists$. Inside our rewrite rules φ and ψ serve as (term) variables which belong to the domain of the substitution that instantiates the rewrite rules.

The normalization steps presented in the following are taken from different research areas such as automated theorem proving (ATP) and term rewriting. The new contribution is rather the combination of these steps. From ATP (cf. [21]) we take **negation normal form** (NNF) and **prenex normal form** (PNF). Rewriting towards NNF is essentially a equivalence transformation that pushes negation towards the leaves of a formula tree, whereas prenex normalization pushes the quantifiers to the top of a formula tree. With the classical rewrite rules for NNF and PNF we cannot get a unique normal form, though. The problem is the PNF: in general a formula can have more than one PNF. We will overcome this problem via an intermediate **minimal scope form**, where the quantifiers are pushed down the formula tree as far as possible. From that we build a unique PNF. Once in PNF the typical ATP approach would be to normalize the quantifier free body of the PNF to the conjunctive or disjunctive normal form (CNF/DNF). The problem with these normal forms is again the lack of uniqueness. This problem can be solved in a **Boolean Ring** representation where a unique normalization (modulo AC) can constructed which is a result from term rewriting.

8.4 Prenex Normal Form

Although we mentioned two sorts of normal forms, namely NNF and PNF, we will define a single rewrite system that determines a unique normal form (modulo something) that is both in NNF and PNF. Nevertheless we start with the classical prenex rewrite system on its own in order to show its lack of uniqueness. In fact we will need a variants of this classical prenex normalization too, but only as a post-processing step of our minimal scope form. For now we assume that $\neg$, $\wedge$, and $\vee$ are our only connectives, i.e. $\Rightarrow$ and $\Leftrightarrow$ are eliminated. Later we will integrate the according rewrite rules. Beforehand we define the classical prenex rewrite system:

Definition 8.17. (Prenex Normalization) Let φ and ψ be formulae and $Q = \forall, \exists$ and $\circ = \wedge, \vee$. The **prenex rewrite system** is defined by the following rewrite rules:

$$\begin{array}{rr} \varphi \circ (Q x.\psi) \rightsquigarrow Q x.\varphi \circ \psi & (Q\, x.\varphi) \circ \psi \rightsquigarrow Q x.\varphi \circ \psi \\ \neg\forall x.\varphi \rightsquigarrow \exists x.\neg\varphi & \neg\exists x.\varphi \rightsquigarrow \forall x.\neg\varphi \end{array}$$

A normal form of this rewrite system is called **prenex form**. It has the form $Q_1x_1.\cdots Q_nx_n.\varphi$ (with $Q_i = \forall, \exists$), where φ is free of quantifiers. $Q_1x_1.\cdots Q_nx_n.$ in this expression is called its **prenex** and φ its **body**.

For these rewrite rules the unique binding assumption is important. Without this assumption there would be formulae where these rewrite rules would change their semantics as this example shows: $(\exists x.P(x)) \wedge \forall x.Q(x) \rightarrow \exists x.\forall x.P(x) \wedge Q(x)$. The purpose of unique binding is exactly to avoid this effect called **variable capturing**.

Investigating the properties of this rewrite system, we observe immediately that it is terminating: every rewrite step moves the quantifier up in the formula tree, which can be done only finitely often. But the system is not confluent as the following example shows:

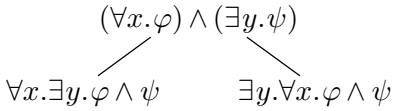

Figure 8.1. The prenex rewrite system is not confluent

In the left branch in figure.8.1 we apply at first the rewrite rule matching the universal quantification and afterward the rewrite rule matching the existential quantification, and in the right branch we do it the other way round. Neither result can be further subjected to any rewrite rules, i.e. the prenex rewrite system does not yield unique normal forms. Only the body is always unique.

However, we can get a unique normal form if we take the inverse of the prenex rewrite system, the **minimal scope** rewrite system (where $Q = \forall, \exists$ and $\circ = \wedge, \vee$):

Definition 8.18. (Minimal Scope) Let $Q = \forall, \exists$ and $\circ = \wedge, \vee$. The **minimal scope rewrite system** is defined by the following rewrite rules:

1. $Q x.\varphi \circ \psi \rightsquigarrow \varphi \circ (Q x.\psi)$ if $x \notin \mathcal{V}ar(\varphi)$
2. $Q x.\varphi \circ \psi \rightsquigarrow (Q\, x.\varphi) \circ \psi$ if $x \notin \mathcal{V}ar(\psi)$

The constraint $x \notin \mathcal{V}ar(\varphi)$ prevents a **variable escape**, i.e. bounded variables becoming unbound by the downsizing of the quantification scope. Variable escape is in this sense the inverse of variable capturing.

Lemma 8.19. The minimal scope rewrite system is terminating.

Proof. Every rewrite step pushes a quantifier towards the leaves of the formula tree. As the depth of this tree is finite and the quantifiers are finite this process must be finite. □

Lemma 8.20. The minimal scope rewrite system is free of critical pairs.

Proof. All of the left hand sides of our rewrite rules have the quantifiers $Q = \forall, \exists$ only in front position. Hence, there is no $\sigma^*(l_i|_p) = l_j$ at a non-variable position p. □

With these two lemmata and the corollary 8.16 we know that the minimal scope rewrite system is convergent. This is not yet our intended prenex form: we want to integrate now the rewrite rules that lead to the negation normal form.

Definition 8.21. (Negation Normal Form) The **negation normal form rewrite system** is defined by the following rewrite rules:

1. $\neg\neg\varphi \rightsquigarrow \varphi$
2. $\neg(\varphi \wedge \psi) \rightsquigarrow \neg\varphi \vee \neg\psi$
3. $\neg(\varphi \vee \psi) \rightsquigarrow \neg\varphi \wedge \neg\psi$
4. $\neg\forall x.\varphi \rightsquigarrow \exists x.\neg\varphi$
5. $\neg\exists x.\varphi \rightsquigarrow \forall x.\neg\varphi$

Lemma 8.22. The negation normal form rewrite system is terminating.

Proof. $\neg(\varphi \wedge \psi) \rightsquigarrow \neg\varphi \vee \neg\psi$ and $\neg(\varphi \vee \psi) \rightsquigarrow \neg\varphi \wedge \neg\psi$ push negation down towards the leaves of the term tree. Finally, there is neither a negation above " $\wedge$ " nor above " $\vee$ ", hence these rewrite rules are no longer applicable at the same position. $\neg\neg\varphi \rightsquigarrow \varphi$ reduces the number of negations until there is no "$\neg\neg$" left. This cancellation of double negations terminates as the other rewrite rules can produce only finitely many double negations. □

Lemma 8.23. The negation normal form rewrite system is free of critical pairs.

Proof. The critical pairs of rewrite rules from the list above are (1,2), (1,3), (1,4), and (1,5). We show that they are all joinable:

(1,2): the critical overlap $\neg\neg(\varphi \wedge \psi)$ rewrites with rule 1 to $\varphi \wedge \psi$ and with 2 to $\neg(\neg\varphi \vee \neg\psi)$. The latter rewrites with rule 3 to $\neg\neg\varphi \wedge \neg\neg\psi$ and with repeated application of rule 1 also to $\varphi \wedge \psi$. Hence the critical pair is joinable. For the critical rewrite rules (1,3) the argument is analogous.

(1,4): the critical overlap $\neg\neg\forall x.\varphi$ rewrites with rule 1 to $\forall x.\varphi$ and with rule 4 to $\neg\exists x.\neg\varphi$. With rule 5 we rewrite the latter to $\forall x.\neg\neg\varphi$ and finally with rule 1 to $\forall x.\varphi$. Hence the critical pair is joinable. For the critical rewrite rules (1,5) the argument is analogous.

□

Again with these two lemmata and the corollary 8.16, we know that the negation normal form rewrite system is convergent. The last step is to merge these both rewrite systems to a single rewrite system and to prove its convergence.

Definition 8.24. (Final Minimal Scope) We define the **final minimal scope rewrite system** as the union of a rewrite system for $\Rightarrow$ and $\Leftrightarrow$ *elimination* (1,2), the negation normal form rewrite system (3–7), the minimal scope rewrite system (8,9):

1. $\varphi \Leftrightarrow \psi \rightsquigarrow \varphi \Rightarrow \psi \wedge \psi \Rightarrow \varphi$
2. $\varphi \Rightarrow \psi \rightsquigarrow \neg\varphi \vee \psi$
3. $\neg\neg\varphi \rightsquigarrow \varphi$
4. $\neg(\varphi \wedge \psi) \rightsquigarrow \neg\varphi \vee \neg\psi$
5. $\neg(\varphi \vee \psi) \rightsquigarrow \neg\varphi \wedge \neg\psi$
6. $\neg\forall x.\varphi \rightsquigarrow \exists x.\neg\varphi$
7. $\neg\exists x.\varphi \rightsquigarrow \forall x.\neg\varphi$
8. $Qx.\varphi \circ \psi \rightsquigarrow \varphi \circ (Qx.\psi)$ if $x \notin \mathcal{V}ar(\sigma^*(\varphi))$
9. $Qx.\varphi \circ \psi \rightsquigarrow (Qx.\varphi) \circ \psi$ if $x \notin \mathcal{V}ar(\sigma^*(\psi))$

where $Q = \forall, \exists$ and $\circ = \wedge, \vee$.

A normal form of this rewrite system is called **minimal scope form**, denoted by $\breve{\varphi}$ (for a given input term φ).

Lemma 8.25. The final minimal scope rewrite system is terminating.

Proof. The rules 1 and 2 can obviously applied only finitely often. The rules 3–7 push the negation to the leaves of the formula (i.e. the leaves are literals) and no other rule pushes it upwards. Rule 8 and 9 push the quantifiers towards the leaves, but rule 6 and 7 push them upwards. Hence, the rules 6 and 7 are the only problematic. However, since no rule pushes negation upwards the quantifiers are pushed down as far as possible once the negation has reached the leaves. □

Lemma 8.26. The final minimal scope rewrite system is free of critical pairs.

Proof. We have already investigated the critical pairs inside the negation normal form rewrite system and we have shown that the minimal scope rewrite system is free of critical pairs. It remains to investigate the possible critical pairs between these two rewrite systems. ($\Rightarrow$ and $\Leftrightarrow$ *elimination* rules obviously do not interfere with the other rule systems). The new critical pairs we have to consider are (6,8), (6,9), (7,8), and (7,9). Since they are structurally all analogous, it is sufficient to show joinability for one of these critical pairs. Let us take (6,8) with $Q = \forall$ and $\circ = \wedge$. The critical overlap $\neg(\forall x.\varphi \wedge \psi)$ rewrites with rule 6 to $\exists x.\neg(\varphi \wedge \psi)$ and with rule 8 to $\neg(\varphi \wedge \forall x.\varphi)$. We can join this critical pair: in the first case we apply rule 4 giving $\exists x.\neg\varphi \vee \neg\psi$ followed by rule 8 (with $Q = \exists$ and $\circ = \vee$) which gives $\neg\varphi \vee \exists x.\neg\psi$. In the second case we continue with rule 4 that gives us $\neg\varphi \vee \neg\forall x.\psi$, and finally we apply rule 6 thus joining to $\neg\varphi \vee \exists x.\neg\psi$. □

From these two lemmata and the corollary 8.16 we conclude :

Theorem 8.27. The final minimal scope rewrite system is convergent.

8.4.1 From Minimal Scope Back to Prenex Normal Form

As mentioned in the overview above our minimal scope normal form is only an intermediate normal form which we transform back to a unique prenex form. The reason for this apparently paradox step is that we want to apply another normalization step in the Boolean ring representation. Moreover, the rewrite system for Boolean ring normalization operates on quantifier free expressions.

We recall that prenex normalization and minimal scope are in a sense inverse operations: the former pushes the quantifiers towards the root and the latter to the leaves of a formula tree. Moreover, only the latter rewrite system produces a unique normal form. Our dilemma is that we need a prenex normal form (more precisely its quantifier free body) for our subsequent Boolean ring normalization, but we want to keep the uniqueness obtained by the minimal scope normalization. One possible solution would be to enforce a deterministic application order of the prenex rewrite system. Thus a unique minimal scope normal form would be uniquely mapped to a prenex normal form. A possible enforcement would be given by the constraint to apply rules always at the left-most possible position of the formula. A minimal scope form like $(\exists x.P(x)) \wedge (\forall y.Q(y)$ is mapped by this deterministic prenex rewrite system to $\exists x.\forall y.\, P(x) \wedge Q(y)$ whereas the indeterministic prenex rewrite system would generate also $\forall y.\forall x.\, P(x) \wedge Q(y)$.

Though this enforcement of a unique prenex normal form seems to be a simple solution it has some drawbacks. In fact, if a prenex normal form would be our final goal then this simple solution would be sufficient. But for our subsequent Boolean ring normalization, it turns out that this simple solution looses uniqueness. This will become clear later on. Therefore we follow an alternative approach which is a bit more involved, but which will pay off for Boolean ring normalization.

Our alternative approach is a combination of term rewriting and standardization: we generate from a given minimal scope form all possible prenex normal forms via the (indeterministic) rewrite system and define the minimal element (w.r.t. a term ordering) as the unique prenex normal form.

Definition 8.28. (Unique Prenex Form) Let φ be a term and $\breve{\varphi}$ its minimal scope form. Let $\prec$ be an arbitrary but fixed term ordering. The **unique prenex form** of φ, denoted by $\varphi{\downarrow^{p}}$, is defined as the least element from the set of all prenex forms of $\breve{\varphi}$ with respect to $\prec$.

Since the body of all prenex forms of a given formula is always identical, we can alternatively consider the unique prenex form as that prenex form with the least prenex (w.r.t. a term ordering).

In general the set of all prenex forms of a minimal scope form does not contain all permutations of quantifications as prenexes. For instance the minimal scope form $\forall x.(\exists y.R(x,\, y)) \vee (\exists z.Q(x,\, z))$ has $\forall x.\exists y.\exists z.R(x,\, y) \vee Q(x,\, z)$ and $\forall x.\exists z.\exists y.R(x,\, y) \vee Q(x,\, z)$, but it has no prenex form with e.g. $\exists y.\forall x.\exists z$ as prenex. The minimal scope form thus implicitly encodes all admissible orderings of quantifications in derivable prenex forms. Such ordering can be naturally encoded in a tree. Instead of a formal definition of such a tree encoding we illustrate the idea by an example:

Example 8.29. The formula $\forall x.\exists y'.\forall y.\exists z'.\exists z.R(x, y, z) \wedge R(x, y', z')$ is in prenex form. Its minimal scope form is $\forall x.\,(\forall y.\exists z.R(x,\, y,\, z)) \wedge (\exists y'.\exists z'.R(x,\, y',\, z'))$. Another prenex form that has the same minimal scope form is for instance $\forall x.\forall y.\exists z.\exists y'.\exists z'.R(x,\, y,\, z) \wedge R(x,\, y',\, z')$. All prenex variants are determined by the scope order from that minimal scope form as presented in figure 8.2 below.

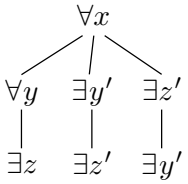

Figure 8.2. Scope order of $\forall x.(\forall y.\exists z.R(x,y,z)) \wedge (\exists y'.\exists z'.R(x,y',z'))$.

8.5 Boolean Ring Normalization

In this section we will focus on the body of the unique prenex form, i.e. we will deal only with quantifier free formulae. The only logical symbols appearing in these formulae are conjunction, disjunction, and negation – all other symbols are already removed in the previous normalization step.

Considering all formulae that can be built from atomic formulae and $\wedge$, $\vee$, and $\neg$, we have of course many equivalence classes. The purpose of the normalization discussed here, is to compute the normal forms with respect to these classes induced by the following well known equations from Boolean algebra:

$$\begin{aligned}
\neg\top &\approx \bot \quad \neg\bot \approx \top \quad \neg\neg\varphi \approx \varphi \\
\varphi \wedge \top \approx \varphi \quad \varphi \wedge \bot \approx \bot \quad \varphi \wedge \varphi \approx \varphi &\qquad \varphi \vee \top \approx \top \quad \varphi \vee \bot \approx \varphi \quad \varphi \vee \varphi \approx \varphi \\
\neg(\varphi \wedge \psi) &\approx \neg\varphi \vee \neg\psi \qquad \neg(\varphi \vee \psi) \approx \neg\varphi \wedge \neg\psi \\
\varphi \wedge (\psi \vee \phi) &\approx (\varphi \wedge \psi) \vee (\varphi \wedge \phi) \qquad \varphi \vee (\psi \wedge \phi) \approx (\varphi \vee \psi) \wedge (\varphi \vee \phi) \\
\varphi \wedge \psi &\approx \psi \wedge \varphi \qquad \varphi \vee \psi \approx \psi \vee \varphi \\
\varphi \wedge (\psi \wedge \phi) &\approx (\psi \wedge \phi) \wedge \varphi \qquad \varphi \vee (\psi \vee \phi) \approx (\psi \vee \phi) \vee \varphi
\end{aligned}$$

Figure 8.3. Identities in the Boolean algebra

These equations commonly suggest a rewrite system leading to the well known conjunctive normal form (CNF) or disjunctive normal form (DNF) – which normal form is computed, essentially depends on how we direct the distributive equations. But independently of our choice how we direct these equations to a rewrite system neither normal form (CNF nor DNF) is unique modulo AC. To make this claim evident, assume we decide for a rewrite system leading to CNF (the case for DNF is dual). The two formulae $(a \vee b) \wedge a \wedge b$ and $a \wedge b$ are obviously in conjunctive normal form and they are equal with respect to the above equations but they are not equal modulo AC[8.4].

A solution to this problem is an isomorphism from Boolean algebra to Boolean ring theory by this translation:

$$\begin{aligned}
\varphi \wedge \psi &\rightsquigarrow \varphi * \psi \qquad \varphi * \psi \rightsquigarrow \varphi \wedge \psi \\
\varphi \vee \psi &\rightsquigarrow \varphi + \psi + \varphi * \psi \qquad \varphi + \psi \rightsquigarrow (\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi) \\
\neg\varphi &\rightsquigarrow 1 + \varphi
\end{aligned}$$

Figure 8.4. Translation from Boolean algebra to Boolean ring (left column) and backward (right column).

8.4. Even the Quine McClusky algorithm as well as the Karnaugh Veitch method, both trying to minimize DNFs, are not deterministic which make the result formula not unique modulo AC.

Note that we use only the translation to the Boolean ring representation, but not backwards. We list the latter only to show that Boolean algebra and Boolean ring are in fact isomorphic structures. The meaning of $+$ in the Boolean ring is thus "either-or" – which is associative, hence we omit the brackets. By convention $*$ binds stronger than $+$. We replace $\wedge$, $\top$, and $\bot$ correspondingly by $*$, 1, and 0 as these operators and constants in the Boolean ring have related properties in ordinary arithmetic:

$$\begin{gathered}\varphi * 1 \approx \varphi \quad \varphi * 0 \approx 0 \quad \varphi * \varphi \approx \varphi \quad \varphi + 0 \approx \varphi \quad \varphi + \varphi \approx 0 \\ \varphi * \psi \approx \psi * \varphi \quad \varphi + \psi \approx \psi + \varphi \\ \varphi * (\psi * \phi) \approx (\psi * \phi) * \varphi \quad \varphi + (\psi + \phi) \approx (\psi + \phi) + \varphi \\ \varphi * (\psi + \phi) \approx (\varphi * \psi) + (\varphi * \phi)\end{gathered}$$

Figure 8.5. Identities in the Boolean ring

The isomorphic relationship between Boolean algebras and Boolean rings was found in 1936 by Stone (according to [32]). It becomes particularly interesting for us, as it exhibits an important feature that is missing in the rewrite systems for CNF and DNF:

Theorem 8.30. ***(Stone 1936)*** There exists a unique normal form modulo AC for each Boolean function[8.5] with n variables.

Even more important, there also exist a convergent rewrite system for this normal form modulo AC which was first given by Hsiang & Dershowitz in 1983:

$$\begin{gathered}\varphi * 1 \rightsquigarrow \varphi \quad \varphi * 0 \rightsquigarrow 0 \quad \varphi * \varphi \rightsquigarrow \varphi \quad \varphi + 0 \rightsquigarrow \varphi \quad \varphi + \varphi \rightsquigarrow 0 \\ \varphi * (\psi + \phi) \rightsquigarrow (\varphi * \psi) + (\varphi * \phi)\end{gathered}$$

Figure 8.6. Rewrite system for the Boolean ring

In fact this rewrite system can be obtained automatically by using the Knuth-Bendix procedure enhanced with associative-commutative unification. The resulting normal form is commonly called **polynomial normal form** or **Boolean ring normal form** and we denote it by $\varphi{\downarrow^{B}}$ for a given input term φ. Let us convince now how the two formulae $a \wedge b$ and $(a \vee b) \wedge a \wedge b$, which are not joinable in a CNF rewrite system, get a unique polynomial normal form:

	$(a \vee b) \wedge a \wedge b$
Translated to Boolean ring:	$(a + b + a * b) * a * b$
$\varphi * (\psi + \phi) \rightarrow (\varphi * \psi) + (\varphi * \phi)$ modulo AC:	$a * a * b + b * a * b + a * b * a * b$
$\varphi * \varphi \rightarrow \varphi$:	$a * b + a * b + a * b$
$\varphi + \varphi \rightarrow 0$:	$0 + a * b$
$\varphi + 0 \rightarrow \varphi$: the polynomial normal form!	$a * b$
Backward translation to Boolean algebra:	$a \wedge b$

Table 8.1. Unique polynomial normal form of the conjunctive normal forms $a \wedge b$ and $(a \vee b) \wedge a \wedge b$ are unjoinable by a CNF rewrite system.

8.5. Boolean function in our terminology is a term $\varphi \in \mathcal{L}$ whose only logical symbols are 0, 1, $*$, and $+$.

With the unique normal form in the Boolean ring, we have implicitly a normal form for the Boolean algebra due to the isomorphism between the two theories, i.e. we know that two formulae are equal with respect to the Boolean algebra identities if and only if they have a common polynomial normal form.

8.6 Combining the Rewrite Systems

So far we have investigated two rewrite systems (we consider the translation to Boolean ring as part of the Boolean ring rewrite system). Both of them are convergent (the second modulo AC). However, the combination of two convergent rewrite systems does not yield a new convergent rewrite system[8.6] in general. It is easy to construct a witness for this claim: consider for a set $A = \{a, b, c\}$ the two abstract rewrite systems $\rightarrow_1 := \{a \rightsquigarrow b\}$ and $\rightarrow_2 := \{a \rightsquigarrow c\}$. Obviously both rewrite systems are convergent, but the union $\rightarrow_{1,2} := \rightarrow_1 \cup \rightarrow_2 = \{a \rightsquigarrow b, a \rightsquigarrow c\}$ is not. The table below shows the induced equivalence classes and the normal forms in bold face. Note the union $\rightarrow_{1,2}$ induces a single equivalence class, but with two normal forms.

	$\rightarrow_1$	$\rightarrow_2$	$\rightarrow_1 \cup \rightarrow_2$
equivalence classes	$\{a, \boldsymbol{b}\}, \{\boldsymbol{c}\}$	$\{\boldsymbol{b}\}, \{a, \boldsymbol{c}\}$	$\{\boldsymbol{b}, \boldsymbol{c}\}$

Table 8.2. Induced equivalence classes and normal forms (bold face) for the rewrite systems $\rightarrow_1 := \{a \rightsquigarrow b\}$ and $\rightarrow_2 := \{a \rightsquigarrow c\}$ with a base set $A = \{a, b, c\}$.

We already combined two convergent rewrite systems, namely those for negation normal form and minimal scope to the final minimal scope form. Hence, we want to see how far we get with combining our two convergent rewrite systems for unique prenex and the Boolean rings. For that we have to look again for critical pairs in the set of all rewrite rules from both rewrite systems which we list here:

	unique prenex rules	Boolean ring rules
1.	$\neg(\varphi \wedge \psi) \rightsquigarrow \neg\varphi \vee \neg\psi$	$\varphi \wedge \psi \rightsquigarrow \varphi * \psi$
2.	$\neg(\varphi \vee \psi) \rightsquigarrow \neg\varphi \wedge \neg\psi$	$\varphi \vee \psi \rightsquigarrow \varphi + \psi + \varphi * \psi$
3.	$\neg\forall x.\varphi \rightsquigarrow \exists x.\neg\varphi$	$\neg\varphi \rightsquigarrow 1 + \varphi$
4.	$\neg\exists x.\varphi \rightsquigarrow \forall x.\neg\varphi$	$\neg\varphi \rightsquigarrow 1 + \varphi$

Table 8.3. Critical pairs: each row displays a pair of critical rewrite rules.

Let us check these critical pairs for joinability – the list numbers below correspond to the critical rules in table 8.3:

1. The critical overlap is $\neg(\varphi \wedge \psi)$ leading to the critical pair $\neg\varphi \vee \neg\psi$ and $\neg(\varphi * \psi)$ which is joinable to $1 + \varphi * \psi$ after translation to the Boolean ring representation followed by several applications of the Boolean ring rewrite rules.
2. The critical overlap is $\neg(\varphi \vee \psi)$ leading to the critical pair $\neg\varphi \wedge \neg\psi$ and $\neg(\varphi + \psi + \varphi * \psi)$. They join to $1 + \varphi + \psi + \varphi * \psi$ after translation to the Boolean ring representation followed by several applications of the Boolean ring rewrite rules.
3. The critical overlap is $\neg\forall x.\varphi$ leading to the critical pair $\exists x.\neg\varphi$ and $1 + \forall x.\varphi$ which is not joinable with the given rewrite rules.

8.6. [13] gives a survey under what conditions the combination of rewrite systems preserve confluence, termination, and convergence.

4. Analogous to 3.

The 3. and 4. critical pairs are problematic. In principle if a rewrite system is not convergent, one can try to extend it with additional rewrite rules such that all critical pairs can be joined again. In our case we can make the 3. critical pair joinable by adding the rewrite rule $\exists x.1 + \varphi \rightsquigarrow 1 + \forall x.\varphi$, but this raises new problems to solve: 1) new critical pairs like $\exists x.1 + \varphi \rightsquigarrow 1 + \forall x.\varphi$ and $\neg\exists x.\varphi \rightsquigarrow \forall x.\neg\varphi$, and 2) termination must be also reinvestigated. Actually, completion procedures have a research tradition on its own with the Knuth-Bendix completion procedure as its most famous representative. Some elaborated completion systems are already implemented. In general completion is an undecidable task, though. A definitive answer on our particular completion problem is not in the scope of this thesis and would require further research for the following reasons:

1. The critical pair lemma for rewrite systems modulo an equational theory requires actually a bit more than the standard critical pair lemma and thus the completion techniques become more complicated (cf. [33]).
2. It is unclear how our constrained rewrite rules[8.7] for the minimal scope fits to the completion methods where such constraints do not exist.

Even if we had successfully completed the two rewrite systems, there still remains an open issue. Consider the formula $(\forall x.P(x)) \vee a$, in our above introduced stratified rewrite approach: we first normalize this formula to a unique prenex form $\forall x.(P(x) \vee a)$ and translate it to Boolean ring representation $\forall x.P(x) + a + P(x) * a$. Since no further Boolean ring rewrite rules are applicable this is our normal form. Now let us skip the prenex step and translate $(\forall x.P(x)) \vee a$ directly to the Boolean ring $(\forall x.P(x)) \vee a$ which yields $(\forall x.P(x)) + a + (\forall x.P(x)) * a$. These both Boolean ring formulae are obviously equivalent, since we applied only equivalence transformations, but they are not identical. It is unclear which rewrite rules could bring them together again without compromising termination.

In this thesis, we must content with a concatenation of two convergent rewrite systems whose union is no longer convergent. Nevertheless, this stratified normalization is an improvement for our renaming problem, as we are able to identify many more formulae via normalization than without: even in a non-convergent but normalizing rewrite system the normal forms represent a whole set of formulae. A matching of normal forms thus represents a matching of sets of formulae. The more formulae a normal form represents the more effective the "semantic" matching via normalization. The list below summarizes the benefit and limit of our normalization more formally. For that we introduce the following notations: $=_p$ and $=_B$ for the equivalence relation induced by the final prenex and Boolean ring rewrite system respectively, $=_{p,B}$ for the equivalence relation induced by the transitive closure of the union $=_p \cup =_B$, and finally $\sim$ for AC-equality. If the combination of our rewrite system were convergent then we would get AC-equal normal forms for φ and ψ iff $\varphi =_{p,B} \psi$. Our stratified normalization process does not achieve this desirable goal, but at least an approximation to that. This approximation is expressed by the following properties:

1. If $\varphi =_p \psi$ or $\varphi =_B \psi$ then $(\varphi\downarrow^p)\downarrow^B \sim (\psi\downarrow^p)\downarrow^B$
2. If $(\varphi\downarrow^p)\downarrow^B \sim (\psi\downarrow^p)\downarrow^B$ then $\varphi =_{p,B} \psi$
3. for some φ and ψ holds $\varphi =_{p,B} \psi$, but not $(\varphi\downarrow^p)\downarrow^B \sim (\psi\downarrow^p)\downarrow^B$

8.7. Our constrained rewrite rules should not be mixed up with conditional rewriting. The latter is a research field on its own and there exist completion techniques. In conditional rewriting an equation can depend on other equations, i.e. $\varphi_1 \approx \psi_1 \wedge \ldots \wedge \varphi_n \approx \psi_n \Rightarrow \varphi \approx \psi$ (cf. [22] p. 269).

4. for some φ and ψ holds $(\varphi{\downarrow^p}){\downarrow^B} \sim (\psi{\downarrow^p}){\downarrow^B}$, but neither $\varphi =_p \psi$ nor $\varphi =_B \psi$

The first two item hold by the definition of the rewrite systems. The third item is about the limits of our combined rewrite system (we have seen an example above). The last item states essentially the gain of the combination of our rewrite systems – the whole is more than the sum of its parts: for instance, for $\varphi := \neg\forall x.P(x)$ and $\psi := \exists x.1 + P(x)$ we have $(\varphi{\downarrow^p}){\downarrow^B} = (\psi{\downarrow^p}){\downarrow^B} = \psi$, but neither $\varphi =_p \psi$ nor $\varphi =_B \psi$.

In other words: our stratified normalization identifies more formulae than $[\varphi]_p \cup [\psi]_B$, but less than $[\varphi]_{p,B}$ where $[\varphi]_x$ denotes the equivalence induced by the equivalence relation $=_x$.

8.7 Finalizing Normalization with AC-Standardization

We said that Boolean ring normalization is "only" a normalization modulo AC. Therefor AC-standardization suggests itself as a final step after the normalization. Actually, AC-standardizing the polynomial normal forms from Boolean ring normalization is probably the highest benefit of AC-standardization at all.

At this point we will also understand the reason for our less simple definition of unique prenex form. Recall that an AC-skeleton of a formula φ is the minimal formula equal to the (simple) skeleton $\varphi{\downarrow}$ modulo AC and one-to-one symbol morphism. The simple definition of unique prenex form is not unique modulo AC. For instance, suppose we would make the prenex rewrite system deterministic by the constraint to apply rules always at the left-most possible position of the formula: consider the formula $(\exists x.P(x)) \wedge (\forall y.Q(y))$ which is AC-equal to $(\forall y.Q(y)) \wedge (\exists x.P(x))$, but both formulae would have different (simple) unique prenex forms, notably $\exists x.\forall y.P(x) \wedge Q(y)$ and $\forall y.\exists x.P(x) \wedge Q(y)$. Our actual definition of unique prenex form in contrast is a normal form modulo AC in the sense that the prenex is uniquely defined independently of any AC-transformation in the body. In our example both formulae had the unique prenex $\forall y.\exists x.$ if we assume a term ordering with $\forall \prec \exists$.

Lemma 8.31. If two formulae are AC-equal then they have the same unique prenex form.

Proof. Assume that φ and ψ are AC-equal. The application of minimal scope rewrite rules preserves AC-equality, hence we get AC-equal minimal scope forms $\breve{\varphi}$ and $\breve{\psi}$. On the other hand, we can follow from the prenex rewrite rules that two AC-equal formulae have the same set of possible prenexes and thus the identical minimal prenex, i.e. the unique prenex form. □

The whole normalization process of a term φ, including AC-standardization, can be summarized as: $((\varphi{\downarrow^p}){\downarrow^B}){\downarrow^{AC}}$. In section 10.1.4 we sketch the implementation of the normalization process and what function it plays within theory interpretation and intersection search.

We want to conclude with a demonstration of the normalization process on the formula presented at the beginning of this chapter that served as motivation for normalization:

Example 8.32. Consider the formula from the beginning of this chapter:

$$\forall\varepsilon.\varepsilon > 0 \Rightarrow \exists\delta.\forall x.\forall y.0 < |x-y| \wedge |x-y| < \delta \Rightarrow |f(x) - f(y)| < \varepsilon$$

For readability we want to replace the non-logical terms by relations as follows:

$$P(\varepsilon) := \varepsilon > 0 \qquad Q(x,y) := 0 < |x-y|$$
$$R(x,y,\delta) := |x-y| \qquad S(x,y,\varepsilon) := |f(x)-f(y)| < \varepsilon$$

We subject this formula to normalization and standardization:

$$\begin{aligned}
\varphi &:= \forall\varepsilon.P(\varepsilon) \Rightarrow \exists\delta.\forall x.\forall y.Q(x,y) \wedge R(x,y,\delta) \Rightarrow S(x,y,\varepsilon) \\
\varphi\downarrow^{P} &= \forall\varepsilon.\exists\delta.\forall x.\forall y.\neg P(\varepsilon) \vee \neg Q(x,y) \vee \neg R(x,y,\delta) \vee S(x,y,\varepsilon) \\
(\varphi\downarrow^{P})\downarrow^{B} &= \forall\varepsilon.\exists\delta.\forall x.\forall y.1 + P(\varepsilon) * Q(x,y) * R(x,y,\delta) \\
&\qquad + P(\varepsilon) * Q(x,y) * R(x,y,\delta) * S(x,y,\varepsilon) \\
((\varphi\downarrow^{P})\downarrow^{B})\downarrow^{\mathrm{AC}} &= \forall v_1.\exists v_2.\forall v_3.\forall v_4.1 + p_1(v_1) * p_2(v_3,v_4) * p_3(v_3,v_4,v_2) \\
&\qquad + p_4(v_1) * p_5(v_3,v_4) * p_6(v_3,v_4,v_1) * p_7(v_3,v_4,v_2)
\end{aligned}$$

To the AC-skeleton belongs the only AC-parametrization list $\delta_{\varphi}^{\mathrm{AC}} = [P,Q,R,P,Q,S,R]$. Note that all four formulae listed at the beginning of this chapter have the same normalization and standardization.

Chapter 9
Theory Completion

The techniques of automated theory interpretation search can be used to gain even more knowledge by means of two additional techniques:

- **theory completion**: Since knowledge gain increases the chance to map source axioms into a target theory that could not mapped into before, repeated theory interpretation can lead to more knowledge gain than single theory interpretation.
- **fragments**: in many cases theorems of a theory are not proven with all axioms as premises, but only with some of them or even with other (beforehand proven) theorems as premises. These premises can be viewed as axioms of a theory on its own thus forming a subtheory of its host theory.

We want to elaborate this further beginning with theory completion. We defined a explored theory as a triple $\langle \Sigma,\ \Gamma,\ \Delta \rangle$ of signature, axioms, and proven theorems in chapter 4. To be precise: theory completion only deals with explored theories. For convenience, however, we will sometimes omit this adjective.

Now, we introduce some further terminology appropriate to describe knowledge gain.

Definition 9.1. (Augmentation of Explored Theories) Let $S = \langle \Sigma, \Gamma, \Delta \rangle$ and $T = \langle \Sigma',\ \Gamma',\ \Delta' \rangle$ be two explored theories, and $\sigma\colon\ S \rightarrow T$ a theory interpretation with $sen(\sigma)(\Gamma) \subseteq \Gamma' \cup \Delta'$. We call σ an **explored inclusion** (from S to T) and we define the **augmentation** of T by S via σ as:

$$S \oplus_\sigma T \ :=\ \langle \Sigma', \Gamma', \Delta \cup sen(\sigma)(\Delta') \rangle$$

A **knowledgebase** is a collection of explored theories. Let $\mathcal{T}$ be a knowledgebase containing S and T and let $\sigma\colon S \rightarrow T$ be an explored inclusion. We define the **knowledgebase extension** of $\mathcal{T}$ by $\sigma\colon S \rightarrow T$ as:

$$(\sigma\colon S \rightarrow T) \oplus \mathcal{T} \ :=\ \{S \oplus_\sigma T\} \cup (\mathcal{T} - T)$$

Every augmentation of an explored theory potentially leads to a proper knowledge gain – in the case above it is $sen(\sigma)(\Delta') - \Delta$. As soon as we put a new explored theory into the knowledgebase this new explored theory may give rise to many new knowledgebase extensions. It is not just that the new explored theory may augment all the old explored theories in the knowledgebase, but the old may also augment the new explored theory. And also once the old explored theories are augmented it might be possible that the old theories augment each other. We call the exhaustive repetition of such knowledgebase extensions **knowledgebase completion**.

We want to illustrate this process in figure 9.1 using the metaphor of fried egg in a pan: for a given explored theory $T = \langle \Sigma, \Gamma, \Delta \rangle$ the pan represents the set of all sentences derivable from Γ. The whole fried egg inside the pan represents all proven sentences $\Gamma \cup \Delta$ of the explored theory, whereby the yellow of egg represents the axioms Γ and the white of egg the proven theorems Δ.

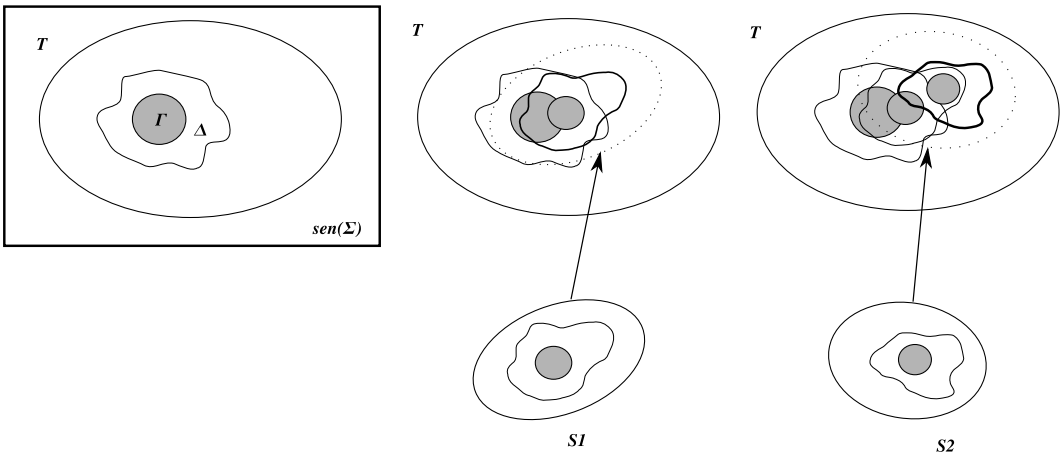

Figure 9.1. Augmentation of an explored theory in a *fried egg* metaphor: in the initial state (a) we have a pan representing a theory $T = \langle \Sigma, \Gamma \rangle$ with a fried egg in it representing the known or proven sentences $\Gamma \cup \Delta$ of it. The yellow of egg Γ represents the axioms and and the white of egg Δ represents the proven theorems. The area of the pan not covered by the fried egg represents the unproven theorems of T. The pan itself lies on a table which represents all the sentences of the signature – only those in the pan are valid in T. In step (b) we have another pan S_1. A theory interpretation from S_1 to T means that we are able to move the pan S_1 over T such that S_1 fits into T and the yellow of egg must be placed inside the fried egg of T. The knowledge gain due to this theory interpretation is that area of T which is covered by the fried egg of S_1, but has not been covered by the fried egg of T before. In step (c) we take another pan S_2 gain even more knowledge in T. Note: we were not able to place the fried egg of S_2 if we had not augmented the fried egg of T by that of S_1 beforehand! We can imagine how this procedure continues with more pans and fried eggs S_k and how we augment more and more the area of pan T with overlapping fried eggs from other pans. Repeating this procedure until we cannot cover any more area of the pan represents theory completion.

After we have seen how knowledgebase completion can increase knowledge gain. We want to see how fragments of theories can contribute even more.

Definition 9.2. (Fragment of Theory) Let $T = \langle \Sigma, \Gamma, \Delta \rangle$ be a explored theory. Any explored theory $S = \langle \Sigma', \Gamma', \Delta' \rangle$ with $\Sigma' \subseteq \Sigma$ and $\Gamma' \subseteq \Gamma \cup \Delta$ is called **fragment** of T.

Consider two explored theories S and T where we are not able to map the axioms of S into the known sentences of T. Then there is no way to gain knowledge from S in T with the information we have at hand. Now, consider we know internal details about the logical dependence between the sentences in S. In fact we usually have such theory internal knowledge at the moment when we finish a proof: to prove a theorem we have used a small set of premises which is in many cases not equivalent to the axioms of the theory: any set of known sentences from the theory can be used as premises to prove a new theorem. The premises and the conclusion (i.e. the theorem) form a fragment on its own. Since the axioms from this fragment are different from the axioms of the explored theory there is a different chance to map those axioms into the target fragment. Thus the fragment can lead to knowledge gain where the explored theory cannot. Again we want to illustrate this with the fried egg metaphor in figure 9.2.

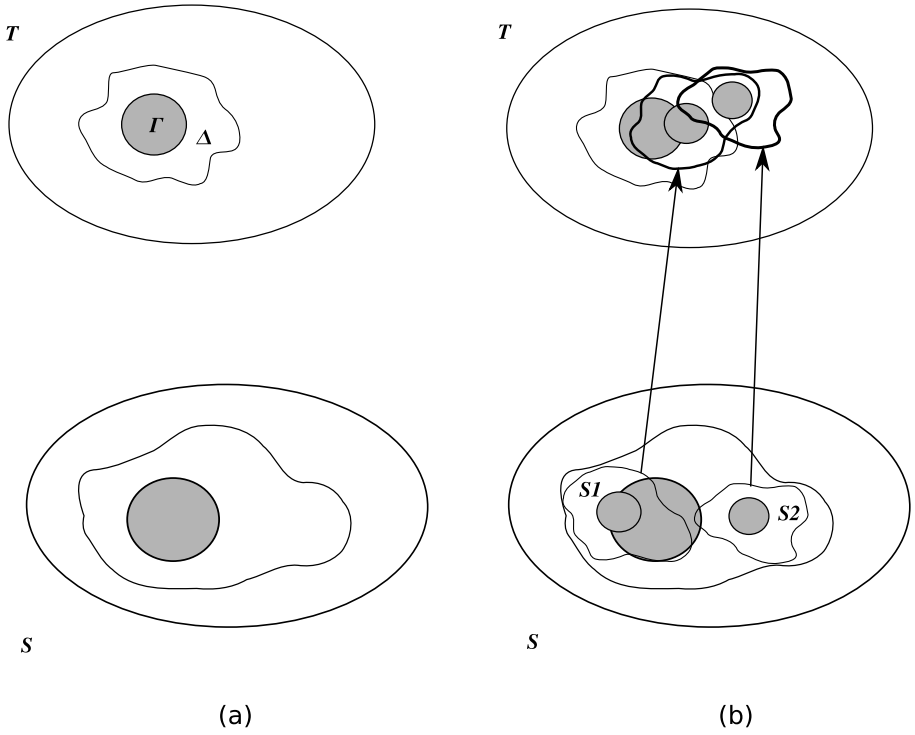

Figure 9.2. *Fried egg* metaphor for augmentation of explored theory with fragments: consider two pans, a source pan S and a target pan T. In many cases we are not able to place the yellow of egg inside the target fried egg of T. However, after an analysis of the large fried egg we may find many small fried eggs (e.g. S_1 and S_2) both covered by the large one. For each of those we may find a way to place them inside the target fried egg of T. Thus we have gained more knowledge about T from fragments S_1 and S_2 which we could not get from S.

Moreover, we get the best results if we choose our fragments so that all their premises are really used to prove their theorems, because every not used premise could cause an avoidable theory interpretation failure. The smallest fragments with respect to the number of premises are those with only one theorem, because in a fragment with two theorems there might be one or more premises that are not used to prove both theorems. We call a fragment with only one theorem a **sequent**. In order to get the highest knowledge gain from the knowledge with our theory interpretation search method we should consider all the sequents inside the theories when we do knowledge completion.

In fact knowledge completion based on sequents is a task that reduces to well known method in artificial intelligence, called **forward chaining**, which is basically a recursive application of the modus ponens inference rule. A task that can be perfectly automatized with logical programming languages like Prolog or even Datalog.

Let us see how we can apply forward chaining on sequents for knowledge gain: at first we assume that each formula φ of each theory is already abstracted, i.e. dissected into a pair $\langle \varphi\downarrow, [p_1, ..., p_n]\rangle$ of skeleton and a list of its parameters. Suppose we have for each existing skeleton $\varphi\downarrow$ a unique number k. Then we can isomorphically represent the abstracted formula as a pair $\langle k, [p_1, ..., p_n]\rangle$ or as a term $s_k(p_1, ..., p_n)$ and even shorter as $s_k(\bar{p})$ – where s_k denotes the k-th skeleton. A sequent in this representation has the form $\left\langle \Sigma, \{s_{k_1}(\bar{p}_{k_1}), ..., s_{k_n}(\bar{p}_{k_n})\}, s_j(\bar{p}_j)\right\rangle$ which means that we can derive the conclusion (theorem) $s_j(\bar{p}_j)$ from the premises (axioms) $s_{k_1}(\bar{p}_{k_1}), ..., s_{k_n}(\bar{p}_{k_n})$:

$$s_{k_1}(\bar{p}_{k_1}), ..., s_{k_n}(\bar{p}_{k_n}) \ \vdash_\Sigma \ s_j(\bar{p}_j)$$

We call this an **abstracted sequent**. According to what we have discussed in chapter 7, a translation $\sigma(\varphi)$ of a formula corresponds in our representation to translation of the parameters, i.e. $s_k(\sigma(\bar{p}))$. Due to $\vdash$ translation axiom of a derivability system we thus know if the above sequent holds then for any renaming $\sigma\colon \Sigma \to \Sigma'$ holds:

$$s_{k_1}(\sigma(\bar{p}_{k_1})), ..., s_{k_n}(\sigma(\bar{p}_{k_n})) \ \vdash_{\Sigma'} \ s_j(\sigma(\bar{p_j}))$$

The $\vdash$ translation axiom thus says that the parameters of an abstracted sequence can be arbitrarily (but simultaneously) renamed. By replacing the renameable parameters p by variables X our sequent is represented as a **Horn clause**:

$$s_{k_1}(\bar{X}_{k_1}), ..., s_{k_n}(\bar{X}_{k_n}) \ \Rightarrow \ s_j(\bar{X}_j).$$

The possibility of arbitrary renaming of parameters is thus expressed by the variables. We have defined a knowledgebase as collection of explored theories, but now we want to view it as a collection of sequents, namely all those contained in their explored theories. The maximal knowledge extension in our knowledgebase is the maximal augmentation of all its explored theories. Hence let us consider a single explored theory: its axioms in our representation are a set of abstract formulae $\{s_{k_1}(\bar{p}_{k_1}), ..., s_{k_n}(\bar{p}_{k_n})\}$. Since a theory – as well as any fragment of it – comes with a concrete signature, the parameters of each axiom are fixed. An axiom as abstracted formula $s_{k_i}(\bar{p}_{k_i})$ is thus a ground term (or fact in Prolog terminology) in our representation. From these ground terms we can derive a new fact if there is a fitting Horn clause for them – meaning: there is an instantiation of a Horn clause that makes its premises identical to these ground terms. Derived facts can be added to the already known facts which increases the facts base and possibly new Horn clauses in the knowledgebase can be applied to generate new facts, and so forth. The exhaustive repetition of this procedure is known as forward chaining in logical programming or as a method of model generation in proof theory.

Chapter 10
System Description

An important part of this doctoral research has been the implementation of most of the algorithms described in the previous part of this thesis in order to evaluate their practical usability for large mathematical knowledge bases.

The major outcome of this effort is a **prototype system** that has been used for experiments, as will be described in chapter 11. Though the heading of this section is "system description", it would not be very enlightening for the reader to describe the prototype system as it is right, since it is in the nature of a prototype that it is unfinished in various aspects. At the end of this chapter, we will be explicit in what has actually been implemented in the prototype system. Based on this implementation, the experiments, described in the next chapter, are carried out. Now we prefer, however, to begin with a **didactic system** that presents the basic implementation ideas of the prototype system which would be obscured by technicalities in a direct description of the prototype system.

The outline of this chapter is as follows: we recall the primary functionality of our didactic system (section 10.1); we give a very short and elementary introduction to Haskell (section 10.1.1) which is the underlying programming language used for both the didactic system as well as the actual prototype system; we briefly sketch the deliberate limitations of the didactic system; and finally, we step through the code of all important functions of the didactic prototype (section 10.1.3–10.1.7). It follows a section about the envisioned system (section 10.2) which summarizes on an abstract level the functionality to be added to the didactic system in order to become a real world application. Section 10.3 finally makes explicit to what extend the envisioned system is actually implemented.

10.1 The Didactic System

We want to recall briefly the core services aimed by the prototype system:

- **Theory interpretation search**: Given a set of theories in a library and a query theory S, find all pairs $(T,\ \sigma)$ of theories and signature morphisms such that $T \vdash \sigma(S)$.

- **Theory intersection search**: Given two theories T_1 and T_2 find a maximal theory S with appropriate signature morphisms σ_1 and σ_2 such that $T_1 \vdash \sigma_1(S)$ and $T_2 \vdash \sigma_2(S)$.

Both of these services involve AC-**standardization** and **normalization** and both are **query** services. In case of theory interpretation search a separate process, called **indexing**, is performed: the operation of parsing theory files from formal libraries, standardizing and normalizing their formulae and feeding them into a database. Before any theory interpretation query can be requested, the indexing process must be completed. Thus at query time only the query theory needs to be standardized and normalized, whereas the bulk of computation time for standardizing and normalizing does not contribute to query time.

The didactic system provided below is deliberately incomplete: we are only complete in those functions satisfying the following conditions:

- The function covers parts of the theory presented in previous sections.
- There is a concise implementation for it that even
- clarifies its pure theoretical analysis from the respective sections above.

In particular we abstract from all pure technical functions, like reading and parsing a file, writing to a database, etc. And we also refrain from giving explicit implementations of normalization, since normalization is based on term rewriting whose implementation is straight forward and would not give us any insights in addition to the rewrite rules themselves.

10.1.1 Haskell in a Nutshell

The programming language for our didactic system (as well as the prototype system) is Haskell – a functional programming language[10.1] with a powerful type system. In contrast to imperative languages, Haskell allows for a coding that comes very close to mathematics. Thus it allows, more than most imperative languages, for a concise implementation of what is presented in the theoretical part of this thesis. We will make use of only a few powerful Haskell features in our didactic system – in particular pattern matching and higher order functions. Since the code of the didactic system presented below should be understandable even by a reader without any Haskell background, we start with a short introduction to Haskell, covering only those basic language concepts and built in functions actually needed in the system.

Every function that we will present here has a type declaration part followed by the function definition. A type declaration has the form *functionname* :: *type*. If `A` and `B` are types then `A → B` is a type. A function declaration `f :: A → B` is read as "function `f` is a function from `A` to `B`". This means that `f` can be applied to an argument `x` only if it is of type `A`. The function application is denoted by `f x` (which corresponds to the mathematical notation $f(x)$) and its type is `B`. Two functions `f :: A → B` and `g :: (B → C)` can be composed (in mathematical notation $g \circ f$) in Haskell with the composition combinator ".", i.e. we would write "`g . f`".

A function of type `g :: A → (B → C)` is a higher order function, because the function application `g x` (where `x` is of type `A`) returns a function itself, namely of the type `B → C`. Hence `g x` can be applied on an argument `y` of type `B` such that `(g x) y` is finally of type `C`. To avoid brackets, Haskell reads function applications left associative and types right associative, i.e. `(g x) y` is the same as `g x y` and `A → (B → C)` is the same as `A → B → C`. Apart from function types `A → B`, Haskell the tuple type `(A,B)` built in. Combining both, we can have a function declaration like `g': (A,B) → C` which would correspond to the mathematical notation $g'\colon A \times B \to C$, i.e. a binary function. Note that `g` is a unary function in contrast to `g'`: it takes a single argument

10.1. For an introduction to Haskell see e.g. [6].

and returns a function which in turn takes another argument. However, these functions are isomorphic: they can be transformed[10.2] to each other without loss of information. For that reason we may refer sometimes to a second argument of `g` though it does not exist, strictly speaking – similar in case of n arguments. The last built in complex type to mention here, is the list type: `[A]` denotes the type of a list whose elements are of type `A`; similarly `[(A,B,C)]` is a list type whose elements are triples of type `(A,B,C)` and so on. Lists are central data structures in Haskell and in functional programming languages in general. We will use lists to represent sets of objects in our didactic system to keep it simple and sometimes we call those lists sets.

Types in Haskell can have different names which is particularly useful as an abbreviation for complex types; such type definitions are declared with the `type` keyword. Thus we could introduce for instance the type name `Theory` based on beforehand introduced types `Axiom` and `Signature` as follows:

```
type Theory = (Signature,[Axiom])
```

This example corresponds to the mathematical phrase "a theory is a pair, consisting of a signature and a set of axioms". Besides giving types new names or abbreviating types it is also possible to define completely new **data types** – they are introduced via the keyword `data`. We explain all we need to know about this on an example data type which we will use later in the didactic system:

```
data Maybe x = Just x | Nothing
```

This line declares the new parametrized data type `Maybe` with a type parameter `x`. It has two **constructors**: `Just` and `Nothing` whereby `Just` has the type parameter `x` as argument. The type parameter allows us to have different data type instances like e.g. `Maybe Int` or `Maybe String`, etc. An object of type `Maybe Int` can be constructed by applying the `Just` constructor to an object of type `Int` – e.g. `Just 4711` and similarly `Just "text"` has type `Maybe String`.

The main purpose of types in programming languages is to make programs safer against runtime errors. More important for our focus is the fact that type annotations increase the readability of the code such that large parts of the code should be self-explaining. Moreover, types also enable **pattern matching** that makes program code even more concise. We want to explain this by simple examples like they occur in the didactic system. Consider the following snippet:

```
justFirst :: Maybe (Int,Int) → Int
justFirst (Just (x,y)) = x
justFirst Nothing = 0
```

This function has two definitions depending on two patterns. Both patterns `(Just (x,y))` and `Nothing` have the same type `Maybe (Int,Int)`. When `justFirst` is called, either the first or the second definition is applied, depending on the pattern of the argument. A function application `justFirst (Just (1+1,0))` would return `2` whereas `justFirst (1,0)` would yield a type error at compile time. Often we do not care about some parts of a pattern, as for instance `y` in `justFirst (Just (x,y)) = x`. For that an underscore can be used as placeholder, i.e. `justFirst (Just (x,_)) = x` is an equivalent definition.

10.2. This transformation is known under the notion "Currying" named after the logician Haskell Curry after whom the programming language is named, too. In fact, Moses Schönfinkel investigated the "Currying" isomorphism before and independently of Curry, so that it should be called "Schönfinkeling".

Since lists are used almost everywhere in Haskell, they have convenient constructors: `[]` denotes the empty list and `(x:lst)` constructs a list by prepending a new element `x` to the list `lst`. Another way to construct lists in Haskell is by comprehension that has the following syntactic scheme:

```
[f x | x ←xs, P x]
```

For the understanding of our didactic system, it is sufficient to think of list comprehension as set comprehension in mathematics: $\{f(x)|x \in \mathrm{X},\ \mathrm{P(x)}\}$.

To get a better idea how pattern matching works with list constructors we look at the `takeJust` function which we will actually use in the didactic system:

```
takeJust :: [Maybe a] → [a]
takeJust ((Just x):xs) = x:(takeJust xs)
takeJust (Nothing:xs) = takeJust xs
takeJust [] = []
```

This function applied on a list `[Just 5,Nothing,Just 3, Nothing]` returns `[5,3]`. Moreover we will make use of these basic list functions:

- `map :: (a→b) → [a] → [b]`
- `concat :: [[a]] → [b]`
- `concatMap :: (a → [b]) → [a] → [b]`
- `foldl1 :: (a → a → a) → [a] → a`

They are all higher order, because they take as first argument a function. `map` applies a function point-wise on the list: `map f [x1,...,xn]` returns `[(f x1),...,(f xn)]`. The `concat` function flattens a list: `concat [[1,2,3], [4,5]]` returns `[1,2,3,4,5]`; and `concatMap` combines `map` and `concat`, i.e. `concatMap f lst = concat (map f lst)`. The `foldl1` function takes a binary function, for instance `+`, and intersperses it into the list: `foldl1 (+) [1,2,3,4]` equals `1+2+3+4` and hence returns `10`.

Functional programming is often characterized as programming without side effects. Since any input and output of a program is a side effect, there cannot be a useful program without side effects – after all you want to see at least some output on the screen (or on other devices). Haskell solves this problem with **monads** – derived from category theory. It is far beyond the scope of this section to provide the theory behind monads and how they handle side effects[10.3]. Here we consider them just as means to enable imperative aspects in Haskell: every function that communicates with the outside (reads from or writes to files, screen, database, etc.) uses the IO-monad whose return type is `IO R` where `R` is the type of the actual returned value. In other words, a function with return type `IO R` is like a function with return type `R`, but with side effect. Such functions are often called **actions** and we want to reserve the term "function" for functions in the strict sense i.e. those without side effect. Simple examples for actions are `getChar :: IO Char` that takes no argument, but reads a character from standard input and returns it, and `putChar :: Char →IO ()` which takes a character as argument, writes it on the standard output and returns nothing.

10.3. In [44] Simon Peyton Jones, one of the leading language designer of Haskell, gives a valuable introduction on "monadic input/output, concurrency, exceptions and foreign-language calls in Haskell". There he argues how the monadic approach let become Haskell "the world's finest imperative language" though it is rather famous for being one of the purest functional programming languages.

Return values of actions cannot be assigned to constants via the equality symbol: `myInput = getChar`, for instance, would not assign an input character to `myInput`. Haskell provides for actions syntactic constructs to mimic imperative programming: the `do` keyword and the $\leftarrow$ token for assignments, and a `return`. The following example should make the usage obvious:

```
sampleIOaction :: String → IO Char
sampleIOaction str = do c ← getChar
                        putStr str
                        return c
```

This action takes one string (`str`) as argument, waits for another character (`c`) from standard input, writes `str` to the standard output, and finally returns `c`. In fact this `do` construct is just syntactic sugar that makes use of two basic monadic operations, but we refrain from going in details here.

10.1.2 Limits of the Didactic System

Search speed is one of our main concerns. As we have mentioned in chapter 7, a skeleton filter is the central idea for an efficient search algorithm. The didactic system will show more concretely where this filter method works within the whole search algorithm. The experienced Haskell programmer will discover many obvious ways how to gain more efficiency out of our code in particular by taking the obvious optimized data structures for sets and mappings. We refrain from such kind of optimization in our didactic system in order to convey the main idea with a minimum of Haskell background knowledge such that readers without Haskell experience should be able to understand the code without much effort whereas the Haskell experienced reader immediately sees how to improve speed with the right representation. More specifically we will use only lists and tuples to represent sets and mappings. For instance a list like `[('a',[1,2,3]),('b',[4,5])]` would be used to represent a mapping that maps a to the set $\{1, 2, 3\}$ and b to the set $\{4, 5\}$. Of course such representation has its typical problems – for instance: what mapping is represented by `[('a',[1,2]),('a',[2,3])]`? We ignore all such technical difficulties as they can be easily solved with the right choice of built-in datatypes. The reader with Haskell experience knows how to do that and the reader without this experience should not bother here. It should be mentioned that in the prototype system the efficient representations for mappings and sets are taken. Finally we make use of several functions whose implementation is straight forward. For those functions the type declarations are provided, but the definition body is explicitly left `undefined`. Thus the complete code is compilable, but only executable when all occurrences of `undefined` are implemented.

10.1.3 Overview

At first, we want to get an overview of the main functions whose implementation will be given explicitly. Figure 10.1 shows the three user interface (UI) functions `index`, `search`, and `theoryIntersection`. They represent the three possibilities to interact with the system. Apart from them we have several internal functions linked to the UI functions. Hereby an arrow from e.g. `match` to `matchMany` means that the function `matchMany` uses (among others) the function `match`. For clarity only the most important functions are depicted in this network of functions.

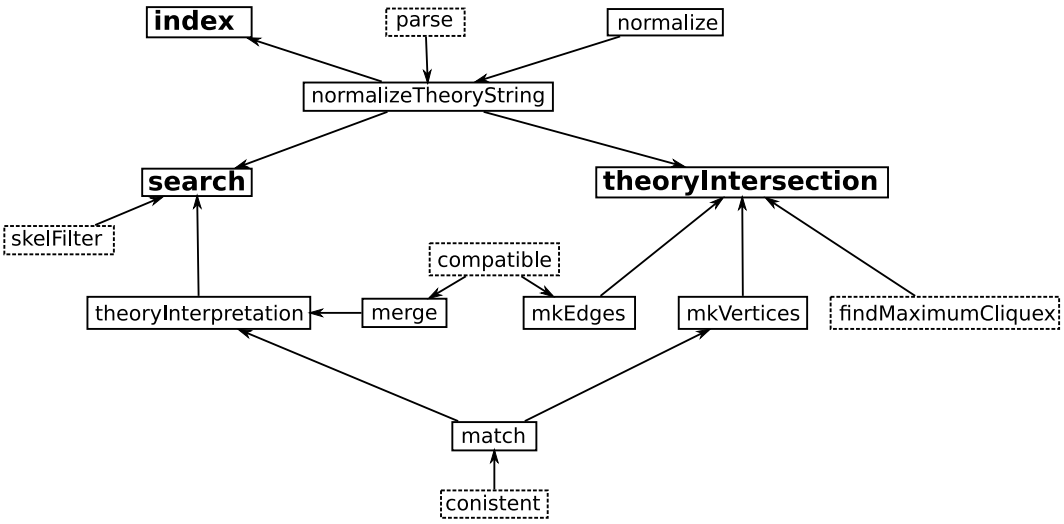

Figure 10.1. The three user interface functions `index`, `search`, and `theoryIntersection` and its subfunctions. Functions in dashed boxes are left undefined.

10.1.4 Normalization

Since `normalizeTheoryString` is the only function which is involved in all UI-functions, we want to start with it here: `normalizeTheoryString` parses a theory, given as string, and normalizes as well as standardizes all its formulae:

```
type Statement = (IsAxiom,Formula)
type NStatement = (IsAxiom,[Param],Skeleton)
type TheoryProfile = (TheoryName,[NStatement])

normalizeTheoryString :: TheoryString → TheoryProfile
normalizeTheoryString theoryString = (theoryName,map normalize statements)
    where (theoryName,statements) = parse theoryString

normalize :: Statement → NStatement
normalize = acStandardize . booleanNormalize .
            prenexForm . negationNormalForm . preprocess

parse :: TheoryString → (TheoryName,[Statement])
parse = undefined
```

Listing: 1

We consider the `parse` function as black box that reads a string representing the whole theory and returns the theory name with a list of statements. A statement is a formula (`Formula`) together with a truth value (`isAxiom`) indicating whether the corresponding formula is an axiom or a theorem. The `normalize` function normalizes and standardizes statements, thus it returns the statement whereby the formula is replaced by its skeleton and parameter. The `normalize` function is a sequence of normalization steps in the order: `preprocess`, `negationNormalForm`, `prenexForm`, and `booleanNormalize`; and a final standardization by `acStandardize` as discussed in section 8.7. Each normalization step implements a set of rewrite rules as presented in different places of chapter 8.

Rewrite rules can be easily implemented in Haskell by means of pattern matching. We are content to give one example: consider the rewrite rule $\varphi \Rightarrow \psi \rightsquigarrow \neg\varphi \vee \psi$. With the data type

```
data Formula = Implies Formula Formula
             | And Formula Formula
             | Neg Fomrula.
```

for logical expressions we can implement this rewrite rule simply as:

```
impToAnd :: Fomrula → Formula
impToAnd (Implies f g) = And (Neg f) g
```

Similarly we can implement in a straight forward way the other rewrite rules. Concerning `acStandardize` we refrain from a concrete implementation here as this would be too technical and hence would not explain more than what is already presented in section 7.5.3.

10.1.5 Indexing

The `index` function takes a database connection and a list of file paths where the theories to be indexed are stored. Each file is read into a string representing a theory and then passed to the function `theoryToRecords` that transforms it to a data base record. Finally these records are fed into the connected database. The `theoryToRecords` function applies `normalizeTheoryString` on theories as list of strings and transforms the out-coming theory profiles into database records with the helper function `toRecord`.

```
type DBRecord = (TheoryName,IsAxiom,[Param],Skeleton)

index :: DBConnection → [FilePath] → IO ()
index dbConnection filePaths =
    do theoryStrings ← mapM readFile filePaths
       dbInsert dbConnection (theoryToRecords theoryStrings)

theoryToRecords :: [TheoryString] → [DBRecord]
theoryToRecords theoryStrings = toRecords theoryProfiles
    where theoryProfiles = map normalizeTheoryString theoryStrings

toRecords :: [TheoryProfile] → [DBRecord]
toRecords lst = concatMap g lst
    where f a (b,c,d) = (a,b,c,d)
          g (a,lst) = map (f a) lst

dbInsert :: DBConnection → [(TheoryName,IsAxiom,[Param],Skeleton)] → IO ()
dbInsert = undefined
```

Listing: 2

10.1.6 Search

The `search` function takes as input a database connection to the database where all the indexed theories are stored, and a file path to the source or query theory. It returns all

the target theories from the database to which there is a theory interpretation from the source theory and moreover all the corresponding theory interpretations (`[Renaming]`) themselves. For that it reads in the theory file and normalizes/standardizes it, thus getting the `sourceProfile` from which we need only its axioms (`sourceAxioms`). Moreover, we need the skeletons of these axioms (`sourceAxiomSkels`), since the main idea to make theory interpretation search fast was the skeleton filter (cf. chapter 7): the function `skelFilter` takes a database connection and all the axiom skeletons from the source theory and looks up from the connected database all those records whose skeleton is identical to one of these axiom skeletons. This is the first stage of search space reduction. Afterward, all the records are sorted by theory name and bundled to theory profiles. In the second stage of search space reduction only those theory profiles pass whose skeletons comprise all the source skeletons. These remaining theory profiles (`targetProfiles`) are the only possible candidates from the database to become target theories for our query theory – they have the "right" skeletons. In fact these two stages exist only conceptually and the `skelFilter` can achieve this result with a single, though complicated, SQL query. We refrain from looking to this at how this can be done in SQL.

```
type Renaming = [(Param,Param)]

search :: DBConnection → FilePath → IO ([(TheoryName,[Renaming])])
search dbConnection filePath =
    do theoryString ← readFile filePath
       let (_,sourceStatements) = normalizeTheoryString theoryString
           sourceAxioms = filter isAxiom sourceStatements
           isAxiom (b,_,_) = b
           sourceAxiomSkels = map getSkel sourceAxioms
           getSkel (_,_,skel) = skel
           in do targetProfiles ← skelFilter dbConnection sourceAxiomSkels
                 return (map (theoryInterpretation sourceAxioms)
                             targetProfiles)

skelFilter :: DBConnection → [Skeleton] → IO ([TheoryProfile])
skelFilter = undefined
```

Listing: 3

The purpose of the `theoryInterpretation` function is to find all the actual theory interpretations (`[Renaming]`) for a given list of source axioms and a given candidate target theory. Passing through the skeleton filter does not guarantee the existence of a matching between parameters of two formulae (from the source and the target respectively) with identical skeletons. Hence, the list of renamings might be empty, of course. Since `theoryInterpretation` can be only applied on statements (`sourceAxioms`) it returns a function that in turn takes a theory profile as input. Thus we can map `(theoryInterpretation sourceAxioms)` over the candidate target profiles (`targetProfiles`). As final result we get a list of pairs of type `(TheoryName, [Renaming])` with the name of the target theory and all its theory interpretations.

Now we look inside the `theoryInterpretation` function from bottom up (listing 4), starting with its elementary functions: the `consistent` property checks whether two lists of parameters are consistent meaning that a point-wise relation between these lists con-

stitutes a mapping (i.e. is `rightUnique`). The `match` function returns a point-wise mapping from the parameters of the first to those of the second normalized statement, but only if the skeletons of these statements are identical and its parameters consistent. Hence, the result type of `match` is `Maybe Renaming`. If `match s t` returns `Just r` we say the source statement `s` matches the target statement `t` with the renaming `r`. We extend the `match` function to a `matchMany` function whose purpose is to match a single source statement against many target statements which is reflected in `(map (match f) fs)`. Prepending the `takeJust` function to do that, removes all failed matches as we are only interested in the positive results. Note the swapped order of arguments: `matchMany targets source` matches one `source` against many `targets`. It implements what we called in definition 4.20 the (set of) minimal translations $\varphi\overrightarrow{G}$ from a source formula φ to the set of target formulae G. With `(matchMany targets)` we thus have a function that can be mapped in turn over the source statements and thus corresponds to $\overrightarrow{FG}$ in definition 4.20. The `merge` function implements (by means of the `compatible` predicate) the product of compatible morphisms ($\otimes$) as defined in definition 4.21. With the corresponds of F as `sourceAxioms` and G as `targets`, the function

```
foldl merge [] (map (matchMany targets) sourceAxioms)
```

in `theoryInterpretation` implements $\bigotimes_{\varphi\in F}\varphi\overrightarrow{G}$ from lemma 4.23 in section 4 which gives us the set of all signature morphisms (`renamings`) from F to G.

```
theoryInterpretation :: [NStatement] → TheoryProfile
                     → (TheoryName,[Renaming])
theoryInterpretation sourceAxioms (thName,targets) = (thName,renamings)
    where renamings = foldl1 merge (map (matchMany targets) sourceAxioms)

merge :: [Renaming] → [Renaming] → [Renaming]
merge rs qs = [union r q | r ← rs, q ← qs, compatible r q]

compatible :: Renaming → Renaming → Bool
compatible = undefined

matchMany :: [NStatement] → NStatement → [Renaming]
matchMany fs f = takeJust (map (match f) fs)

match :: NStatement → NStatement → Maybe Renaming
match (_,p1,s1) (_,p2,s2) =
    if s1 == s2 && consistent p1 p2
    then Just (zip p1 p2) -- and remove duplicates
    else Nothing

consistent :: [Param] → [Param] → Bool
consistent ps1 ps2 = rightUnique (zip ps1 ps2)
```

Listing: 4

10.1.7 Theory Intersection

The `theoryIntersection` function takes two strings, each representing a theory, and returns a triple `([NStatement],Morphism,Morphism)` where `[NStatement]` represents the list of statements that are contained in both input theories modulo the corresponding morphisms and modulo normalization and standardization. Like in `index` and in `search`, the first thing to do with the theories is to normalize and standardize them with `normalizeTheoryString`. From the returned two lists of normalized statements (`statements1`, `statements2`) the maximum intersection (cf. definition 5.4) modulo a bijective renaming is calculated. As discussed in chapter 5 this task can be mapped to the graph-theoretical problem of finding a maximum clique: the function `mkVertices` generates the `vertices` of the graph out of these two lists of statements. A `Vertex` consists of two normalized statements and a bijective renaming such that they can be mapped to each other. To create a single vertex from two statements the `mkVertex` function is used which is essentially the `match` function from listing 4. Because the matching can fail its return type is `Maybe Vertex` and `mkVertices` takes only the successful matchings (with `takeJust`). From the Cartesian product of these `vertices` the function `mkEdges` selects those `edges`, i.e. pairs of vertices, where the two renamings belonging to a pair of vertices are compatible (by means of the `compatible` predicate as used in listing 10.1.6). Finally a `findMaximumClique` function, which is part of an open source graph library[10.4], finds a maximum clique which means in our context a maximal set of vertices whose renamings are pairwise compatible. This clique becomes the input to the helper function `theoryIntersection'`: at first it extracts all the `renamings` from the vertices in the clique and merges them to a single `renaming`; which still is a bijective mapping since all the bijective `renamings` are compatible. In fact the clique is already an isomorphic representation of the final theory intersection. The purpose of the remaining steps (including `neutralRenamings`) in the `theoryIntersection'` function is only to transform the clique type, i.e. `[(NStatement, NStatement, Renaming)]` into a more natural type for theory intersection: `([NStatement], Renaming, Renaming)`. We refrain from going through these simple but technical steps, we only summarize in mathematical notation what happens inside `theoryIntersection'`: given a triple (F, G, σ) where F and G are finite sets of formulae with $|F| = |G|$ and σ is a (finite) bijective renaming between F and G (compare with theorem 5.11), then our `theoryIntersection'` function computes an isomorphic triple (H, σ_1, σ_2) with $F = \sigma_1(H)$, $G = \sigma_2(H)$ and $\sigma = \sigma_1 \circ \sigma_2$. Thus H are the formulae from the intersection theory and σ_1 (σ_2) is the morphism to the theory containing the formulae of F (G).

10.4. The standard graph library for Haskell, *fgl* developed by Martin Erwig, can be downloaded from `http://web.engr.oregonstate.edu/~erwig/fgl/haskell/` and is already part of the Haskell compiler ghc6.8. Actually this library provides a function that finds the maximum set of independent nodes instead of a maximum. But this is just the dual problem: we only have to replace the set of edges by its complement set w.r.t. the set of all pairs of vertices.

```
type TheoryIntersection = ([NStatement],Morphism,Morphism)
type Morphism = (Renaming,TheoryName)

theoryIntersection :: TheoryString → TheoryString → TheoryIntersection
theoryIntersection theory1 theory2 = (statements,morph1,morph2)
    where (theoryName1,statments1) = normalizeTheoryString theory1
          (theoryName2,statments2) = normalizeTheoryString theory2
          vertices = mkVertices statments1 statments2
          edges = mkEdges vertices vertices
          clique = findMaximumClique vertices edges
          (statements,renaming1,renaming2) = theoryIntersection' clique
          morph1 = (renaming1,theoryName1)
          morph2 = (renaming2,theoryName2)

type Vertex = (NStatement,NStatement,Renaming)
type Edge = (Vertex,Vertex)

mkVertices :: [NStatement] → [NStatement] → [Vertex]
mkVertices ss1 ss2 = takeJust [mkVertex s1 s2 | s1 <- ss1, s2 <- ss2]

mkVertex :: NStatement → NStatement → Maybe Vertex
mkVertex s1 s2 =
    case match s1 s2
    of (Just r) -> Just (s1,s2,r)
       _ -> Nothing

mkEdges :: [Vertex] → [Vertex] → [Edge]
mkEdges vs1 vs2 = [(v1,v2) | v1 <- vs1, v2 <- vs2, compatible' (v1,v2)]
    where compatible' ((_,_,r1),(_,_,r2)) = compatible r1 r2

findMaximumClique :: [Vertex] → [Edge] → [Vertex]
findMaximumClique = undefined

theoryIntersection' :: [Vertex] → ([NStatement], Renaming, Renaming)
theoryIntersection' vertices = (neutralNStatements,renaming1,renaming2)
    where getRen (_,_,r) = r
          renamings = map getRen vertices
          renaming = foldl1 union renamings
          (renaming1,invRen1,renaming2) = neutralRenamings renaming
          getNStatement1 (s1,_,_) = s1
          statements1 = map getNStatement1 vertices
          neutralNStatements = map (rename invRen1) statements1
          rename = undefined

neutralRenamings :: Renaming → (Renaming,Renaming,Renaming)
neutralRenamings renaming = (renaming1,invRenaming1,renaming2)
    where freshParams = getFreshParams (length renaming)
          getFreshParams = undefined
          (params1,params2) = unzip renaming
          renaming1 = zip freshParams params1
          invRenaming1 = zip params1 freshParams
          renaming2 = zip freshParams params2
```

Listing: 5

10.2 Features of the Envisioned System

A didactic system implements by its nature only the core functionality of a real system which is usually much more complex. One common reason is efficiency: in order to improve performance, optimized data structures must be designed. For example, as we mentioned already above, lists are not the most efficient means to represent objects like sets, maps or graphs. Many functions that can be coded very compactly in a declarative style with lists are not very efficient. Suppose we represent a set by a list, then we might want to remove all its duplicate elements. The built-in function for removing duplicates in lists, however, is very slow for long lists. A transformation of a list object into an object of type `Set` removes all duplicates significantly faster. On the other hand for objects of type `Set` there does not exist an analogous convenient syntactic construct as for list comprehension – i.e. there is no scheme `{f x | x ← M, P x}` to generate lists like there is one for lists `[f x | x ← M, P x]`. In general some compact expressions in list representation become less compact, but more computationally efficient in other optimized representations. Moreover, additional code is needed to transform between optimal representation depending on the context – code that is not needed when you stay in list representation.

10.2.1 Input and Output of a Real System

All of our three main functions `index`, `search`, and `theoryIntersection` consume theories. In our didactic system `index` reads each theory from a theory file; `search` reads the query theory from a file, but the target theories from a database; `theoryIntersection` reads in their theories as strings – which we can consider as output of parsed theory files. The choice of these input sources has a practical motivation: formal libraries are commonly available as large amounts of text files, hence the `index` function gets its input from files. After processing the input theories the `index` function writes the output to a database which is eventually the input source for the `search` function. Only the query theory is read from a file. In case of the `theoryIntersection` function two files are a practical input source.

A real system, however, should be more flexible regarding its input: Wherever we chose files as input source in our didactic system, a real system should be able to get theories not just from files, but also from databases, web-services, streams and the like. Moreover, in a real system the database, which the `search` function connects to, can be either local or a remote database, it could be persistent or just in memory, file based or a dedicated database application. It mainly depends on the size of the indexed formal library how it should be stored for search access. Only for small libraries it is reasonable to keep them in memory – not just because of memory space, but also to save indexing time: A small library might be indexed in a few seconds so that it is even acceptable to index such a library at search time. Large libraries, as investigated in chapter 11, need several hours to be indexed. For them a persistent storage of indexed theories is preferable in order to avoid a rerun of indexing whenever the program has stopped.

Apart from the input, the functions of a real system should have a more elaborated output than the didactic system. Regarding the `search` function, a real system may not only return pairs of theory name and renamings, but also information about what source axiom is equivalent to which target sentence (modulo renaming). Moreover, the knowledge gain of theory inclusion search should be made explicit: show all source theorems which are actual new theorems in the target theory. These certainly useful features were deliberately not included in the didactic system in order to get an uncluttered overview on the core of the search and intersection algorithms. Since the implementation of these features are straight forward we confine ourselves to look at the declaration of the search and intersection function as we may expect them from a real system. In our didactic system the return type of the `search` function is:

```
type SearchResult = [(TheoryName,[Renaming])]
```

In a real system we want to have rather:

```
type SearchResult = [(TheoryName,[(Renaming,SMap,GainedTheorems)])]
type SMap = [(Formula,Formula)]
type GainedTheorems = [Formula]
```

Here a `SearchResult` is a list of target theories referenced by `TheoryName` whereby each of the target theory can include the source theory via possibly many `Renaming`s. Suppose S is the source theory and T one of the found target theories[10.5] and σ one of the possibly many signature morphisms with $\sigma(S) \subseteq T$. Then `SMap` are pairs of formulae (φ, ψ) where φ is an axiom in S and ψ is an axiom or theorem in T which are equivalent modulo σ and the equational theory implemented by `normalize`. Thus we know which source theory axiom is mapped to which formula from the target theory. This mapping might be used to let a theorem prover verify that each pair contains indeed equivalent formulae (in order to control the correctness of `normalize`). Finally `GainedTheorems` is a list of theorems Φ from S which have not been found in T before – or more formally: $\sigma(\Phi) \cap T = \varnothing$.

Similarly we expect more information from the output of `theoryIntersection` than the didactic system provides us:

```
type TheoryIntersection = ([NStatement],Morphism,Morphism)
type NewTheory = [NStatement]
type Morphism = (Renaming,TheoryName)
```

Hereby `NewTheory` represents the common formulae (modulo `Renaming`) of the two target theories (referenced by `TheoryName`). In a real system we want to have as new theory in fact a theory consisting of a signature and a list of axioms:

```
type NewTheory = (Signature,[Formula])
```

Moreover from the `Morphism` we want to know not just the `Renaming`, but also which axiom of the `NewTheory` has which corresponding axiom in the respective target theory. This information is again kept in a `SMap`:

10.5. Note that we are always talking about explored theories in this context – i.e. the finite set of explicitly given formulae.

```
type Morphism = (Renaming,SMap,TheoryName)
```

10.2.2 Supporting Theories in Different Logics and Formats

One goal of this thesis was to experiment with the prototype system on different formal libraries. Since every library contains some fundamental theory like the basic algebraic theories (e.g. theories of monoids, groups, rings, etc.), their knowledge overlap on a conceptual level should be detectable with the `search` and `theoryIntersection` functions. Unfortunately these libraries are formalized in their own formats or even in different logics. Supporting a heterogeneous collection of formal libraries was thus an implicit goal. As presented already in section 3.4 the support of different logics is a central feature of the Hets system. Hence it was an early design decision to develop a system that can be integrated into Hets. Since the Hets system is a rather complex system whose development goes back to the late nineties and is still continuing, the integration of the prototype system has reached only to the level that was necessary for the experiments as described in chapter 11.

The Hets system supports several kinds of logics like propositional, descriptional, modal, first-order and higher-order logics; most of them are typed and some are untyped. One of Hets basic features is the translation of theories from one logic into another (unless the target logic is less expressive than the source logic; for instance theories in first-order logic cannot be translated to propositional logic). Logics differing only on a syntactic level can be considered as different **formats** of the same logic. In this sense some logic translations can be seen as **format translations**. Moreover, theories in a typed first-order logic with only a single type can be also considered as theories in an untyped first-order logic, since the annotated types can be omitted without loss of information (such theories are typically the result of a translation from untyped logic to typed logic). The most prominent logic in Hets is the typed first-order logic Casl [10, 9]. In this subsection, however, we will treat it as an untyped first-order language for theories with just one type. Thus we can say that Hets supports various formats for untyped first-order logics. Apart from Casl (with single type) there is for instance SoftFOL which is the implementation of first-order language which is defined in [29] and mainly used by the automated theorem prover Spass. We do not go into the details of these language implementations, but content an illustrative example:

Example 10.1. Two internal representations of the formula $\forall n.\ \text{even}(n) \vee \text{even}(n + n)$ in Hets – for Casl (without type annotation):

```
Quantification Universal ["n"]
  (Disjunction [Predication "even" [Var "n"],
                Predication "even"
                [Application "plus" [Var "n",Var "n"]]])
```

and for SoftFOL:

```
SPQuantTerm SPForall [SPSimpleVar "n"]
  (SPComplexTerm SPOr
    [SPComplexTerm "even" [SPSimpleVar "n"]],
    [SPComplexTerm "even"
```

```
    [SPComplexTerm "plus" [SPSimpleVar "n", SPSimpleVar "n"]]])
```

For the main goal – the integration of the prototype system into Hets – the functionality (`index`, search, and `theoryIntersection`) does conceptually not depend on the format of the involved theories. Hence, it is sufficient to implement these functions for a single format and translate all theories to that standard format beforehand. Concretely we have to use a logic dependent `parse` function (which is already provided by Hets) inside `normalizeTheoryString` and append to it a format `converter`[10.6] that converts the parser's output to the standard format as shown in the following listing.

```
normalizeTheoryString :: (parse,convert) → TheoryString → Theory f
normalizeTheoryString theoryString = nTheory
    where (Theory f theoryName statements) = convert (parse theoryString)
          nTheory = Theory f theoryName (map normalize statements)
```

Listing: 6

The logic dependency of the `parse` and `convert` function thus becomes an additional argument of the `normalizeTheoryString` function. Eventually this additional argument propagates to our three main functions `index`, `search`, and `theoryIntersection` as all of them use the `normalizeTheoryString` function.

The target format of the `convert` function is the untyped formal language, which we have defined in chapter 6 and can be directly implemented as datatype in Haskell as follows:

```
data Formula l p v = Log l [Formula v p l]
                   | Par p [Formula v p l]
                   | Var v [Formula v p l]
                   | Lambda v Formula v p l
```

Listing: 7

Apparently this format is very close to the SoftFOL format. But apart from the shorter term constructor names this format is completely restricted to the relevant constructs for our theory search and intersection. The SoftFOL format has several special constructs like disjunctive or conjunctive clause lists for instance. Since these additional constructs do not enrich the semantics of the language they are redundant for our purpose. Our target format is tailored to the essential constructs and thereby simple for all the processing steps inside the `normalize` function. Our example formula from above looks in this format as follows:

```
Log Forall
  [Lambda "n"
    (Log Or
     [Par "even" [Var "n" []],
      Par "even" [Par "plus" [Var "n" [],Var "n" []]]])]
```

10.6. In the "institution community" and in Hets the logic and format converter or translater are called comorphisms.

10.3 Functions Implemented in the Prototype System

As usual for the software outcome of a doctoral research project not all features of an envisioned system are implemented in the prototype system. In fact the envisioned system as described above was not fixed at the beginning of this research, but it evolved over time due to the experiences made with the evolving prototype system. At the end only those features were implemented which were considered substantial to test theory interpretation search on a large knowledge base of formalized mathematics and to test theory intersection on various theories. In the following we describe the restrictions of the prototype system with respect to the envisioned system.

10.3.1 Support of Different Formats in the Prototype System

Though the prototype system was designed to support many formats, actually only two converters were implemented: one for CASL and one for SoftFOL. This choice was motivated by the availability of parsers in the HETS system on one hand and by the availability of formal libraries which were intended for experiments (cf. chapter 11) on the other hand. As the far more extensive content was available in the SoftFOL format, the prototype system was optimized to handle the content available in this format. For the comparably small amount of available CASL theories an in-memory database was sufficient and thus the prototype system was equipped with an in-memory database only. Eventually, however, the large amount of SoftFOL theories (for details see next section) made an persistent database indispensable. At that time all the functions used inside the index and search functions for CASL theories were not implemented flexibly enough such that they could be easily ported to access an in-memory as well as a persistence database (a future implementation task). Moreover, the evolution of the HETS system itself required constant adjustment of the prototype system. In order to keep this adjustment work on a minimum the support of CASL was finally dropped in the prototype system. The preference of SoftFOL over CASL was twofold: 1) First of all it is the mass of SoftFOL content such that experiments with many SoftFOL theories is more challenging than with the comparable few CASL theories; 2) the adjustment for CASL turned out to be more involved than that for SoftFOL. The latter is due to the fact that each SoftFOL file contains only one theory whereas a CASL file generally contains a structured collection of theories. Such CASL file, parsed by the HETS system, is internally represented as graph (cf. development graph in section 3.3). Thus the parse process from a file to a theory is not that straightforward as listing 1 suggests, as it involves operations on the development graph. SoftFOL files, in contrast, are always flat – meaning there is no internal development graph necessary. Although there is no proper support for CASL in the final prototype system, the CASL library remains indirectly accessible for experiments with this system by a simple work around: HETS allows to convert a CASL file containing n theories to n SoftFOL files on the command line.

10.3.2 Input and Output in the Prototype System

Concerning the `index`, `search`, and `theoryIntersection` functions, the provided input sources of the prototype system are the same as sketched in the didactic system: `index` reads theory files and stores the processed theories in a persistent database; `search` reads the query theory from a file, but the target theories from the persistent database; `theoryIntersection` reads both theories from files.

Concerning the output of these functions, the prototype system provides all the information as described in the envisioned system and even more, notably statistical observables used for the analysis of the experiments.

10.3.3 Particular Restrictions in the Prototype System

Some theoretical insights concerning the normalization process emerged relatively late during the research phase and therefore have not been implemented yet. Instead of the Boolean ring normalization – which is unique modulo AC – only the normalization to conjunctive normal form is implemented. The latter is not unique modulo AC as the following simple example of two (semantically) equivalent propositional formulae demonstrate: 1) $a \wedge (b \vee \neg a)$ and 2) $(a \vee \neg b) \wedge a$. In the Boolean ring representation both would normalize to $a \wedge b$.

Instead of the unique prenex normal form only the straight forward (non unique) prenex normal form is implemented. Finally the AC-standardization implemented slightly different as described in section 7.5.3 – with the effect that only one of possibly many AC-parametrization per AC-skeleton is computed.

Chapter 11
Experiments

The experiments described in this chapter are conducted with the prototype system (as presented in the previous chapter) on a contemporary PC. They exhibit the performance of theory interpretation search and demonstrate strength and limits of theory intersection.

11.1 Experiments on Automated Theory Interpretation

There were two motivations to experiment with the prototype system on large formal libraries: firstly to test the scalability of the implemented search algorithm and secondly to measure the degree of knowledge reuse in terms of the number of detectable theory inclusions. The world's largest library of formalized mathematics is the Mizar Mathematical Library (MML) [48, 54] that we have therefore taken as testbed for our experiments on theory interpretation search.

11.1.1 The Mizar System and its Library

Mizar [47] is a system for proof verification of formalized mathematics. It was initiated by Andrej Trybulec in 1974 and is one of the oldest systems of its kind with an active community (system developers as well as content providers). So far more than 207 authors have formalized mathematics in 1011 Mizar articles altogether comprising 43149 theorems and 8185 definitions. All proofs of these articles were validated by the Mizar proof checker. The Mizar Mathematical Library of all Mizar articles is hosted on `mizar.org` and all the articles are also published in the *Journal of Formalized mathematics*. Some of the most popular theorems formalized and proof checked in MML are: *Fundamental Theorem of Arithmentic*, *Gödel Completeness Theorem*, *Jordan Curve Theorem*.

Every Mizar article declares its specific environment via references to objects like definitions, theorems, notations, redefinitions etc. in previous Mizar articles such that these imported objects can be used in the current article to construct its own objects – in particular theorems are imported to be used in proofs. Via such imports all articles from the MML are connected to a graph that resembles the development graph if we view articles as theory nodes, (some kind of) redefinitions as translations, and the import references as definitional links. It must be stressed, however, that this analogy is

limited as Mizar and the development graph follow those two substantially different approaches discussed in chapter 4: Mizar is based on a single foundational theory whereas the development graph follows the axiomatic program where each node is a theory on its on right. We call libraries like MML **foundational** and those following the development graph **axiomatic** libraries. MML is based on Tarski/Grothendieck set theory which is formalized in the *TARSKI* article. Every Mizar article must (directly or indirectly) import this article and thereby Mizar's foundational theory. In fact the *TARSKI* article is the only one that provides axioms. Seen as theories, all other articles are **definitional extensions** of that *TARSKI*-theory; their extension is due to the definition of new objects, but for each theorem using these new objects there is a (semantically but not syntactically) equivalent theorem derivable from the axioms of the *TARSKI* theory and using only objects declared there in.

All possible theory morphisms in a foundational library map a theory T only to conservative extensions of T. For theory interpretation search this restriction makes foundational libraries less interesting than axiomatic libraries.

11.1.2 MML as Axiomatic Library

Since mathematics can be formalized in both kind of libraries – foundational as well as axiomatic – it should be possible to transform a foundational library into an axiomatic library and vice versa. Here we are only interested in the former direction. In fact there is a simple principle how to construct an axiomatic library out of a foundational: any collection of theorems in foundational library can be used as set of axioms for a theory inside an axiomatic theory. If the foundational theory is consistent then every in such manner generated axiomatic theory is consistent[11.1].

This principle, however, does not determine which collections of theorems are worth to serve as axioms for a new axiomatic theory on its own. To make the right choice in real world mathematics is an intellectual challenge: history has shown that e.g. there is a use for a semi-ring theory whose only axiom is associativity, but there is no use for a theory whose only axiom is for instance distributivity. Some axiomatizations are successful whereas others have only a short live time – and most possible axiomatic theories have not been considered at all. The reasons for the choices of these axiomatizations are so many-fold such that we are content with considering a simple choice heuristic – motivated as follows: to prove a theorem a mathematician almost never uses the axioms of a foundational theory, instead he uses axioms, definitions, and theorems from (possibly many) axiomatic theories. In a foundational library all axioms from an axiomatic theory are theorems themselves, so that effectively almost every theorem is proved based on other theorems and definitions. In fact this is what can be observed in the Mizar library. This suggests the experimental heuristic to take the collection of theorems and definition used to prove another theorem as axiomatization for a new axiomatic theory.

Such a transformation from the foundational Mizar library into an axiomatic Mizar library is a side effect of Joseph Urban's work in [51], which is the topic of the following subsection.

11.1. Due to the transitivity of the derivability relation: if a contradiction would be derivable from the generated axiomatic theory than it would be derivable from the foundational theory too.

11.1.3 MML in an Untyped First-Order Logic

Actually Joseph Urban's intention in [51] was to make the MML accessible to external theorem provers such that the MML theorems can be proved by third parties. Foremost it is the Mizar language that makes it difficult to access the MML.

The main objective in the development of the Mizar language has always been an intuitive presentation as close as possible to mathematical vernacular. This is achieved by various linguistic features like composable attributes (e.g. `finite non empty set`); predicate properties (`symmetry`, `reflexivity`, `connectedness`,...); functor properties (`associativity`, `commutativity`, ...); user definable mixfix notations; etc. All these expressive linguistic features have precise formal semantics implemented in a first-order logic with second-order schemas and a very involved type system of attributed dependent types. Moreover, these rich linguistic constructs are further enriched with a lot syntactic sugar to enhance the readability. However, all these user oriented features make it extremely difficult to implement a parser for the Mizar language – which then could be integrated into Hets, for instance. In fact, outside the Mizar developer community there has no system been developed that could handle Mizar articles with a full understanding of its semantics.

As a matter of this observation, Joseph Urban implemented a translation of the whole MML into an untyped first-order language that many theorem provers can understand. This target language is specified in [29] and called *DFG-Syntax* as it was intended to be used within a DFG-project about deduction – we want to give it the more intuitive name **SoftFOL** which is used in the Hets system, too. SoftFOL was thought to be a format that can easily be parsed, yet expressible enough for different interest groups. Its grammar is even simpler than the more popular and simple format TPTP that targets almost the same community as SoftFOL. It was an easy task to be accomplished for this doctoral research to implement a SoftFOL parser that is integrated in the Hets system. Thus MML becomes accessible to Hets as well as to the prototype system.

A SoftFOL file starts with some meta description followed by the logical part. The latter is composed of symbol-, declaration-, formula-, and proof-lists. For our purpose it is sufficient to consider the formula lists of which there are two kinds: `axiom` and `conjecture` lists of formulae. All various kinds of definitions of functions, predicates, adjectives, types, structures, etc. in MML are translated into a list of `axiom`s in SoftFOL, whereas theorems translated into the list of `conjecture`s (in SoftFOL there is no formula kind "`theorem`"). Formulae in SoftFOL are always in prefix notation. Binders have as first argument the list of variables they bind. All basic logical symbols one expects are already part of the language. To give an example, a mathematical expression like $\forall x.x > 0 \Rightarrow x = |x|$ would be represented in SoftFOL as:

```
forall([x],implies(less_then(x,0),equal(x,abs(x))))
```

Formula construction in SoftFOL thus is almost the same as in our formal language (cf. definition 6.1).

11.1.4 Translating Mizar language to SoftFOL

An detailed description of the translation from Mizar language to SoftFOL is given in [29]; here we want to summarize some details that will explain why some search results do not meet the obvious expectations.

In general a typed first-order logic is not more expressive than an untyped first-order logic. To keep all the information encoded in its type annotations a typed formula usually translates into a larger formula (i.e. it has more subformulae than the original typed formula has) and often even into several formulae. Instead of a complete specification of the translation rules we want to illustrate some basic principles on three examples involving dependent types (called *modes* in Mizar), functions, and predicates. The language fragment presented in the Mizar snippets should be self-explaining with some additional explanation given on the fly.

Modes are parametric types with constraints on its parameters. A mode with n parameters is translated into a predicate with $n+1$ arguments. For instance in the article `subset_1.miz` we can find the definition of the mode "`Element of X`":

```
definition let X;
   mode Element of X means
        it in X if X is non empty
        otherwise it is empty;
```

In definitions the keyword "`it`" always refers to the expression to be defined. Interpreted as a predicate this dependent type reads as: $\forall X, Y.\mathrm{elem}(Y, X) \colon\Leftrightarrow \begin{cases} Y \in X & \text{if } X \neq \varnothing \\ Y \notin X & \text{else} \end{cases}$ which is represented in SoftFOL as:

```
forall([X,Y],equiv(element_of(Y,X),
                   or(and(in(Y,X),not(empty(X)))
                      and(empty(Y),empty(X)))))
```

Functions in Mizar have to define the types of their arguments and the type of their results. In the article `real_1.miz` the negation function is defined as follows:

```
definition let x be real number;
  func -x -> real number means x + it = 0;
```

In mathematical text book notation we may write this as: "For all $x \in \mathbb{R}$ the negation function $-x \in \mathbb{R}$ is implicitly defined by the equation $x + (-x) = 0$". So we have in fact two expressions to translate: firstly the type declaration $- : \mathbb{R} \to \mathbb{R}$ and secondly the equation $x + (-x) = 0$.

```
forall([x],implies(real(x),real(minus(x))))
forall([x],implies(real(x),equal(plus(x,minus(x)),0)))
```

In general every typed context – i.e. a pattern `let <var> be <type>` – in a definition results in at least one additional formula after the translation to SoftFOL. If no type is provided in the `let` context then the base type `set` is assumed. Since in Mizar everything is a set the corresponding predicate `set` holds for every object and therefore can be omitted. However, the actual SoftFOL translation of MML is full of redundant tautologies most likely coming from the translation of types containing `set`.

Our next example is from the article `tarski.miz` which contains the foundational *TARSKI* theory, where the subset relation is defined (denoted by "`c=`"):

```
definition let X,Y;
  pred X c= Y means x in X implies x in Y;
```

```
  reflexive;
```

Predicate (as well as function) definitions can have attributes attached (here "`reflexive`"). They can be understood as second-order predicates of predicates (and functions respectively) – in the above example "`reflexive`" is an attribute for the subset relation "`c=`". Each attribute generates an extra formula in the SoftFOL translation:

```
forall([X,Y],equiv(subset(X,Y),and(in(x,X),in(y,Y))))
forall([X],subset(X,X))
```

So far we have seen examples for definitions of modes, functions, and predicates. Apart from that, Mizar allows for definitions of structures (e.g. for groups, fields, topology, etc.) and adjectives (i.e. type constraints; in particular usable for type hierarchies), but we skip the details here.

We want to conclude our summary about the translation with the remark: the relativation of typed formulae to untyped formulae increases significantly the number as well as the size of formulae. The following example demonstrates this effect clearly – a theorem from `euclidlp.miz`:

```
theorem Th96:
  x1 in P & x2 in P & x3 in P implies plane(x1,x2,x3) c= P
```

and its corresponding SoftFOL representation:

```
forall([A],implies(m2_subset_1(A,k1_numbers,k5_numbers),forall([B],implies(
m2_finseq_2(B,k1_numbers,k1_euclid(A)),forall([C],implies(m2_finseq_2(C,k1_
numbers,k1_euclid(A)),forall([D],implies(m2_finseq_2(D,k1_numbers,k1_euclid
(A)),forall([E],implies(m2_finseq_2(E,k1_numbers,k1_euclid(A)),equiv(r2_hid
den(B,k4_euclidlp(A,C,D,E)),exists([F],and(m1_subset_1(F,k1_numbers),exists
([G],and(m1_subset_1(G,k1_numbers),exists([H],and(m1_subset_1(H,k1_numbers)
,and(equal(k3_real_1(k3_real_1(F,G),H),1),equal(B,k7_euclid(A,k7_euclid(A,k
9_euclid(A,F,C),k9_euclid(A,G,D)),k9_euclid(A,H,E))))))))))))))))))))))))
```

11.1.5 Remarks on SoftFOL-MML

We want to recall that the purpose of the translation from the Mizar to the SoftFOL was to make MML accessible to other theorem provers, but our interest is the side effect of this translation, namely the conversion of a foundational library into an axiomatic library. The generation of axiomatic theories was driven by heuristics to create for every theorem its own theory where all the definitions and lemmas became axioms that were needed to prove the given theorem. We want to call this the **single theorem per theory** principle. Above we characterized this principle as a simple experimental heuristic that reflects some aspects of mathematical practice. We should mention here that this was not at all Joseph Urban's motivation to follow this principle. His interest was to keep the theorem provers search space small for the proof of a theorem by automated theorem provers; i.e. to keep the set of axioms as small as possible. Independently of the actual intention the outcome of the translation is a set of axiomatic theories – which we want to call **SoftFOL-MML**. Being an axiomatic library does not automatically mean to be well structured. Quite the contrary, all the inheritance structures between Mizar articles via object references across articles are completely lost in the translation:

SoftFOL-MML is a collection of flat theories (in SoftFOL called *problems*) without any cross references. For the purpose of making MML accessible to theorem provers this was not at all a loss, but a necessity as theorem provers are usually designed to consume problems and for the same reason the SoftFOL does not provide means to construct something like a development graph. This fact is the reason for a large amount of redundancy: whenever a Mizar formula is used to prove a theorem it will be translated into an axiom. Due to the single theorem per theory principle there are as many duplicates of this formula as there are theorems whose proofs make use of this formula. It should be noted that a few theorems whose proofs involve second-order formulae are naturally not expressible in the first-order language SoftFOL and hence there are no corresponding theories for these theorems.

With some numbers we want to compare the effect of formula blow up caused on the one hand by the single theorem per theory principle and on the other hand by type decoding as described above: the 1031 Mizar articles are translated to 41759 SoftFOL theories[11.2] overall containing 4294929 formulae of which 4221453 are duplicates of 73476 distinct formulae. In terms of memory the difference is 75MB of Mizar articles versus 1.1GB of SoftFOL theories. On average every SoftFOL theory contains 104 axioms.

11.1.6 Indexing SoftFOL-MML

To enable fast queries on SoftFOL-MML (cf. next subsection) its content is processed by the prototype system as described in section 10.1 and stored at first in a flat relational database with a single table with the following columns:

column name	*datatype*
`theory`	char(255)
`skeleton`	text
`skeleton_md5`	char(32)
`parameter`	text
`role`	enum('axiom','theorem')

Table 11.1. Columns of the table containing all processed formulae from SoftFOL-MML

The column `skeleton_md5` is a check sum (calculated by the well-known md5 algorithm) of the skeleton. It is introduced as the primary key for this table. Since the query speed-up for theory interpretation search is based on an efficient skeleton filter, we need fast access to the skeletons. However, the calculated skeletons are in some cases very long (from 25 to 25553 characters and 107 on average) such that it was not possible to generate a database index on the skeletons themselves, but their check sum.

The long formulae (due to type decoding as described above) also caused other problems that forced an adaption of the initial normalization procedure:

- Normalizing, in particular to CNF, becomes too expensive for large formulae. Formulae with more than 100 subterms tend to cause a stack overflow and were therefore excluded from normalization.
- Formulae where an associative and commutative operator have more than 10 arguments are excluded from AC standardization.

11.2. Due to the single theory per theorem principle and the existence of 43149 theorems in MML we should expect 43149 SoftFOL theories. However, for some reason unknown to the author some of these 43149 theorems have not been considered in the translation to SoftFOL.

- The frequent existence of `true` as part of formulae suggested an additional normalization step to eliminate all these `true` occurrences. For instance a formula like `implies(true,and(true,r(x)))` reduces to `r(x)`. Moreover all axioms which reduce to a single `true` are excluded from the insertion into the database since they do not influence the semantics of a theory.
- The `true` elimination normalization step, however, induced the subsequent artifact of con- and disjunctions with only one argument, which were handled the obvious way; e.g. `and(true,r(x))` reduces to `and(r(x))` and finally to `r(x)`.

As a statistical result of the database insertion process we found that almost 10% of all formulae were tautologies. From the non trivial formulae about 3% were of the said large size that they were excluded from normalization. With the exclusion of large formulae, normalization ran in about two hours on a contemporary PC, which is acceptable for an indexing-time step.

11.1.7 Profiling Theory Interpretation Search on SoftFOL-MML

As we said above SoftFOL-MML is a completely unstructured library with a high degree of redundancy as opposed to the original structured MML. Hence there must be structure hidden in SoftFOL-MML which should be detectable through theory interpretation search. And hopefully such queries return some theory interpretations revealing more structure than what already is in the explicit dependency of the Mizar articles. Therefore a **crossproduct query** was performed, meaning that the prototype system had to find all theory interpretations from 41759 to 41759 SoftFOL theories. First of all this is a strong performance test whose result is presented in the following.

As expected the first successful run of a crossproduct query was only possible after several failures, where the system did not terminate for many query theories. Each failed trial triggered an improvement in the implementation of the prototype system. We want to summarize the most significant results learned during this implementation cycles until the crossproduct query finally succeeded. Hereby we want to characterize the intermediate performance of the prototype system with some numbers, but the final performance with a detailed statistics.

In the first run about twenty queries went through until the first query did not terminate. Search time per query was ranging from a few seconds to several minutes. The surprising result was that some queries yielded thousands of theory interpretations. The reason for such unexpected large search results became clear after a close look into database and the query theory: there are not just duplicates across theories, but also within theories. The cross theory duplicates are the logical outcome of the Mizar to SoftFOL translation of MML, but the inner theory duplicates were unexpected artifacts of this translation. The problem with the inner duplicates is the following: a single search result of the prototype system does not only return a target theory together with a renaming, but in addition to that a mapping of line numbers indicating which source theory formula is translated to which target theory formula. Consider now two copies of a formula φ in the source theory, two copies of ψ in the target theory, and a renaming σ with $\sigma(\varphi) = \psi$. For this single renaming we have four combinations to map occurrences of φ to occurrences of ψ. For each combination a search result is returned. This combinatorial blow up was responsible for the unexpected large number of search results for some query theories. The obvious repair for this artifact was to remove all duplicates inside a theory.

In the subsequent run the number of search results per theory was drastically reduced as expected: a query theory for which previously about a thousand mappings into a target theory were found, now returned not more than five. With this reduction of search results the search time per query was naturally also reduced and many queries which did not terminate within five minutes now terminates within a few seconds. Thus it was possible compute about 100 queries of the crossproduct search until the first query did not terminate.

By a careful analysis of several very slow queries a central bottleneck, namely inside the skeleton filter, was identified and after testing various alternative implementations a solution was found that enabled a search speed up about a factor of 100. We want to summarize the problem and its solution a bit more in detail. First we want to recall the task that the skeleton filter has to accomplish: let Γ be the set of skeletons from the source theory axioms and let $R \subset \mathcal{S} \times \mathcal{T}$ be the relation that indicates which skeleton of all existing skeletons $\mathcal{S}$ occurs in which theory of all theories $\mathcal{T}$ in the database. A skeleton filter then has to compute the set $\tau = \{T \in \mathcal{T} | \forall s \in \Gamma.(s, T) \in R\}$. The expensive part of such a query is the universal quantification. In the query language SQL for relational databases there is no universal quantification, but it is possible via the existential quantification in SQL and only via a twofold nested select query. Unfortunately the existential construct was not supported by the Haskell binding of SQL. A work around with flat select statements is also possible, for instance by simply omitting the universal constraint in the SQL request; then the result set is a superset of the intended result set and inside the Haskell program this set must be reduced according to the universal constraint. However, this and several other similar variants with flat SQL statements did not change the fact that the SQL based skeleton filter was a bottleneck of the search algorithm. Eventually a significant speed up was only possible with alternative representation of the relation $R \subset \mathcal{S} \times \mathcal{T}$ that is optimized to compute the universal constraint: we construct an isomorphic relation to R with $R' \subset \mathcal{S} \times \{0,1\}^n$ where $n = 41759$ (i.e. the number of theories) and $(s, b_1, ..., b_n) \in R'$ iff $b_k = \begin{cases} 1 & \text{if } (s, T_k) \in R \\ 0 & \text{else} \end{cases}$. Given our set of skeletons Γ from the source axioms we now compute the set of byte strings $B = \{\boldsymbol{b} | s \in \Gamma, (s, \boldsymbol{b}) \in R'\}$. Finally we calculate the point-wise product $\beta = \prod_B \boldsymbol{b}$. From the resulting bytestring β we can easily compute our aimed set of theories τ since we have $\beta_k = 1$ iff $T_k \in \tau$. The advantage of this bytestring representation is twofold: 1) it is very compact – every skeleton is associated with a bytestring of at most 5kB. 2) Multiplication on bytestrings is a very fast operation. Both together make the skeleton filter extremely fast such that its time consumption is negligible within the whole search algorithm.

In the third run most of the beforehand very slow queries – more than one minute – performed all within around one second. However, there were still some theories that caused unacceptable search times over one minute. Since the skeleton filter was no longer a bottleneck the last serious bottleneck was identified in the way the `merge` function is applied inside the `theoryInterpretation` function. We recall the `merge` function:

```
merge :: [Renaming] → [Renaming] → [Renaming]
merge rs qs = [union r q | r ← rs, q ← qs, compatible r q]
```

Inside `theoryInterpretation` this function is applied in the form of "`foldl1 merge rr`" where `rr` is of type `[[Renaming]]`. This operation is on average much faster if `rr` is sorted by the length of its elements, i.e. by "`foldl1 merge (map sortByLength rr)`".

With this final optimization step it was eventually possible to conduct the whole crossproduct search with only 128 non terminating queries. The reason for these remaining failures will be discussed below. Beforehand, we want to describe the final result with some statistics.

11.1.8 Statistics on the Crossproduct Queries

Altogether the crossproduct queries of 41631 source theories returned 321910 target theories and 2512651 theory interpretations. These queries were performed within 13 hours. On average a query takes 1.14 seconds per source theory, 0.14 seconds to find a target theory, or 0.019 seconds to determine a theory interpretation.

Queries taking more than five minutes were canceled which affected 129 of the 41759 theories. About 80% of the queries were performed below 1.4 seconds, 90% below 1.9 seconds, and 99% below 5 seconds. Figure 11.1 shows the distribution of queries over search times.

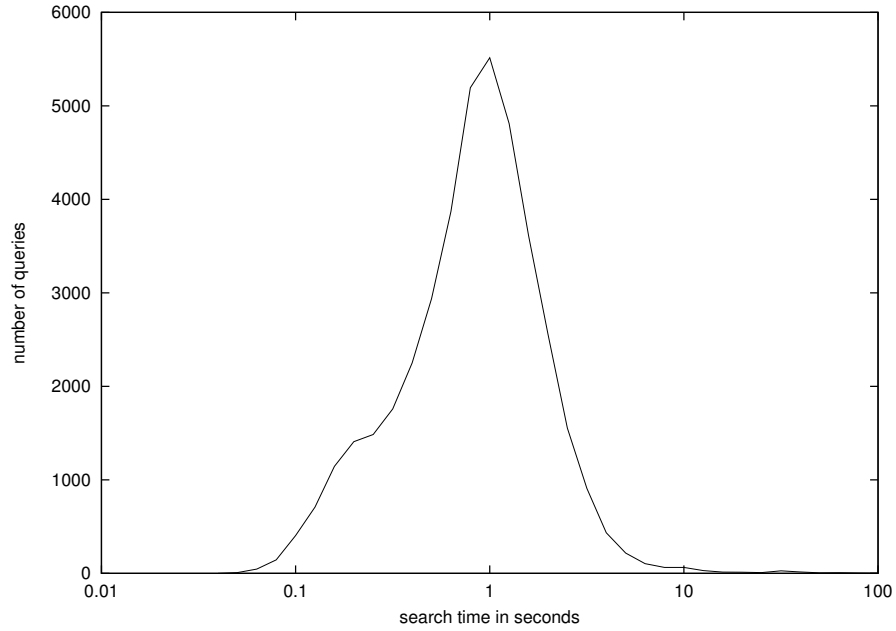

Figure 11.1. Distribution of queries over search times

Search time of a query naturally depends on the number of target theories found and their theory interpretations. These numbers vary considerably: the maximum number of target theories per single source theory is 9679 and the maximum number of theory interpretations is 276826. On average there are 60.7 theory interpretations per source theory and a source theory is included in 6.8 target theories (excluding the source theory itself as target theory). Figure 11.2 depicts the relationship between numbers of source theories and numbers of target theories – for instance there are 5550 including exactly two target theories, 1776 including exactly three target theories, etc. Figure 11.3 depicts the corresponding relationship for theory interpretations.

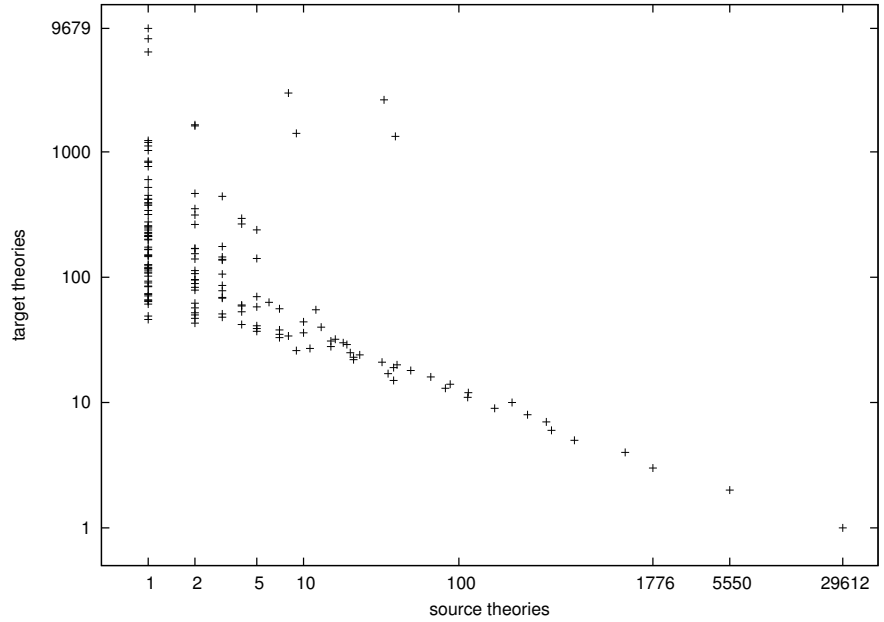

Figure 11.2. Relationship between numbers of source theories and of target theories

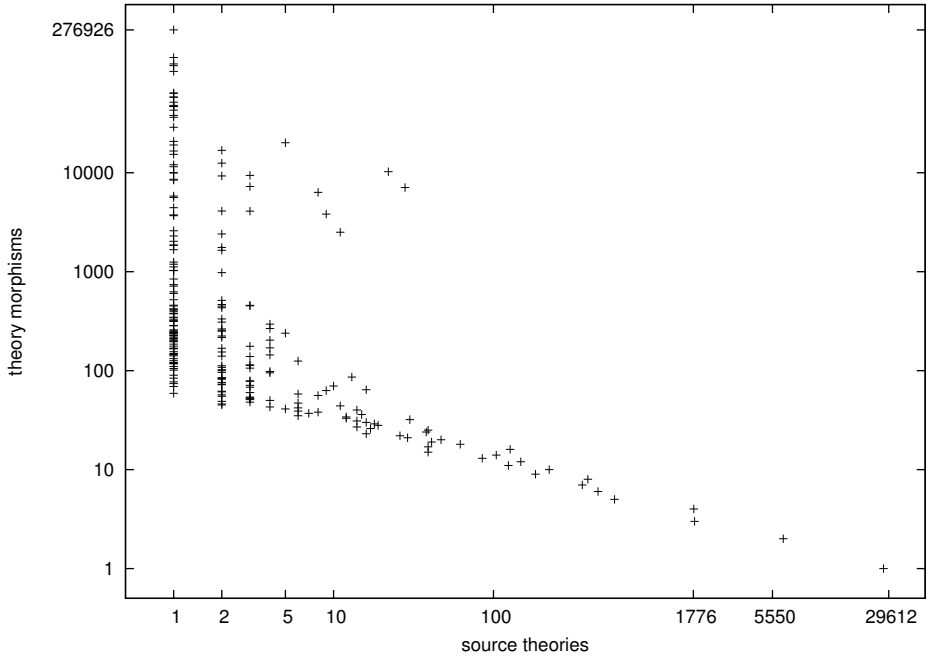

Figure 11.3. Relationship between numbers of source theories and of theory interpretations

We recall that knowledge gain is our main purpose of theory interpretation. According to our formal definition of knowledge gain (cf. definition 4.11) it might be empty. Certainly we are interested in the proper knowledge gain, i.e. the amount of reused source theorems which are not already found in the target. In our experiment with SoftFOL-MML this number amounts to 36346 new theorems. Hence, on average only 1.4% of all found theory interpretations yielded proper knowledge gain. However, if we divide the number of new theorems (36346) by the number of all theories (41631) we get 0.87 – this means, so to speak, almost one new theorem per theory.

11.2 Discussion

In summary this experiment demonstrated two things at the first place:

- **Scalability**: applying automated theory interpretation search applied on 10 thousands of theories containing all together millions of formulae is a feasible task for the prototype system.
- **High potential of theorem reuse**: 60.7 theory interpretations per source theory.

In this respect the prototype system is a successful proof of concept. We want to discuss now the limits and possible improvements, starting with scalability. Above we have already listed the limits regarding the indexing of formulae due to their size. Apart from that there are obviously limits occurring in the query process: some of the queries did not terminate in reasonable time or caused a stack overflow. Experience from the experiment shows a direct correspondence between query time for a given source theory and the number of found theory interpretations. This suggests that a theory interpretation search fails, because the algorithm needs for the huge amount of theory interpretations more memory than available. As soon as the computer starts to swap memory the processing becomes so slow that it makes no sense to keep it running. A mature system, by contrast, should not crash in such a situation. Instead it should present the user with a stream of the most recently computed theory interpretations and the option to abort this search process.

Concerning the afore mentioned "high potential of theorem reuse": although theorem reuse was the main goal of automated theory interpretation search, we should scrutinize the benefit of such theorem reuse. We recall that the highest amount of found theory interpretations per source theory was 276926! It is more than questionable that all of these theory interpretations lead to useful reused theorems. A closer look at these theories reveals the following main reason why they are so excessively included by other theories: 1) the theory is very basic (for instance the *TARSKI* theory is by construction of MMLcontained in every other theory), 2) the theory is very small, or 3) the theory has a high **skeleton multiplicity** – i.e. the amount of multiple occurrences of formulae with identical skeletons. Let us try to explain or comment these three kind of observations:

1) The excessive inclusion of basic theories is of course not surprising and the reusable theorems can be considered as mathematical folklore.

2) Small theories are more likely to be included than large theories simply because the fewer axioms the less constrained are the possible renamings for a theory interpretation. About the usefulness of their reusable theorems, there is not much to say.

3) Typically most of the reusable theorems from a source theory with a high skeleton multiplicity are probably not very interesting. For instance theories containing digits 0 to 9 also contain 10 formulae whose only purpose is to characterize those symbols as a certain type, which means that these 10 formulae are syntactical modulo renaming the digits. If we have two sets of such formulae then there are already 10^{10} combinations to translate the formulae of the one set to the formulae of the other. Other axioms of the source theory counting digits may enforce some restriction of the admissible renamings. Still it happens easily that 100 thousands of renamings lead to theory interpretations. It is very likely, however, that most of these interpretations are not useful from a mathematician's perspective.

In general probably most theory interpretations from a single source theory to excessively many target theories are trivial. It also may happen that all theory interpretations become trivial as soon as once a first one is understood. We could also consider to forget all theory interpretations which did not yield any proper knowledge gain. But among those interpretations there might be some which could give our target theory very interesting new theorems in future.

For this thesis the scalability issues turned out to be much more involving than expected so that there was unfortunately not enough time to analyze the results of the crossproduct query with respect to interesting theorem reuse from a mathematician's perspective. To do so there is also a rather technical problem to solve, namely the back translation of the reused theorems from SoftFOL to Mizar, since it is almost impossible to comprehend the meaning of Mizar formulae in the SoftFOL format (which should become obvious from the description of the translation process in section 11.1.4).

For future work an interesting heuristic to find surprising new theorems would be to focus on those theory interpretations which connect far away theories, which means the following: in foundational libraries all theories are nodes of an acyclic directed graph with one root theory – namely the founding theory. Two nodes in such a graph are far away when their first common ancestor graph is far away. Distance in this graph view on the library reflects very likely also a conceptual distance: the farther away to theory nodes are the less symbols they share and symbols are usually also associated to concepts.

11.2.1 Interlibrary Theory Interpretation Search

Almost all formalized libraries share some fundamental theories like the basic algebraic theories like those of monoids, groups, rings, etc. So it was another goal to experiment with theory interpretation search across libraries. As already mentioned in section 10.2.2 different libraries are formalized in different logics which makes a logic translation necessary before any theory interpretation search can be conducted. In principle HETS is the most appropriate system for this translation task as it provides parsers and translaters ("comorphisms" in HETS terminology) for several formats. Most of the languages supported by the HETS system come from the domain of algebraic specification. Unfortunately, those languages of the big libraries of formalized mathematics, like Coq, NuPRL, IMPS, TPS, PVS, etc[11.3]. are not supported by HETS at the stage of this thesis. This has at least two different reasons:

- These libraries are outside the focus of interest of the HETS community,
- implementing parsers and comorphisms for those languages is conceptually difficult and work intensive and therefore not accomplished by the limited programmer resources.

In fact implementing the (partial) integration of FoC[11.4] was an effort of this doctoral research with a limited success due to several technical as well as conceptual difficulties: at that time started the integration of the open markup format for mathematical documents OMDoc [36] as import and export format for the HETS system and also as export format for FoC. Thus the idea was to make FoC accessible to HETS via OMDoc. Both implementations of OMDoc supports have not reached the compliance to the standard and the stability to make FoC effectively accessible to HETS. To solve all the remaining

11.3. A comparative survey of theorem provers and there libraries can be found in [55].

11.4. The Foc project aims at building an environment to develop certified computer algebra libraries – see: `http://www-calfor.lip6.fr/foc/index-en.html`

technical issues would have consumed too much time. Besides these technical issues there is the problem that the underlying logic of FoC is higher-order whereas the prototype system supports only first-order – more precisely the SoftFOL format. So the idea was to extract from FoC the first-order parts. This turned out to be more intricate than expected as it required sophisticated refactoring of theories. After all the integration of FoC was canceled since the effort to solve these problems became unpredictable. This experience suggested that the integration of any of the other above mentioned libraries into Hets would not be accomplishable within a reasonable time frame.

Hence the only practicable choice to investigate theory interpretations across libraries was those between SoftFOL-MML and the Casl library. For that the Casl theories had to be translated at first into SoftFOL. With Hets this can be done very conveniently on the command line. This translation produces also an effect which is problematic for our automated interpretation search: Casl is a first-order language with types ("sorts" in Casl terminology), hence they must be relativized when translated to untyped SoftFOL, but Hets relativizes a theory only if it has more than one sort (since a one-sorted language is effectively the same as an unsorted language). This different behavior leads to the effect that we cannot find a theory interpretation from Ring theory to the Field theory though in the Casl library the Ring theory is imported into the Field[11.5] theory. To illustrate this we look at the axiom of left unit: the unrelativized version found in the SoftFOL version of Ring has the form $\forall x.\, e * x = x$ where as in the Field theory it has the form $\forall x.\mathrm{elem}(x) \Rightarrow e * x = x$ (where the predicate "elem" was a sort in Casl). Obviously these formulae cannot match, because of their different representation in the untyped first-order language.

With this observation we only want to emphasize how logic translation can break our search algorithm even when apparently simple theories are involved. It becomes much worse if source and target theory come from very different logics as Casl and Mizar – though both are first-order logics[11.6], their type system differ significantly which results in different representations when translated to untyped first-order logic.

Nonetheless we tried to find with our algorithm some theory interpretations between some Casl and Mizar theories from elementary algebra (e.g. from Casl to Mizar ring theory or vice versa) – yet without success. However, the main reason is probably less the different underlying type system than the kind of libraries: the Mizar library is foundational whereas the Casl library is axiomatic. But this is something to be investigated in future.

In summary it must be said that the interpretations algorithm was not successfully applicable between libraries of different logics. There is certainly more to be accomplished, in particular more parsers for large libraries as Coq, NuPRL, IMPS, TPS, PVS should be implemented which is first of all a matter of resources for coding.

11.3 Experiments on Automated Theory Intersection

In chapter 5 we reduced the intersection task to the maximum clique problem from graph theory. Since it is a well-known fact that this problem is NP-hard, we must be prepared that there are theories whose intersection cannot be determined in reasonable time. In fact experiments revealed very soon problematic cases. As we could expect from the above discussion all problematic cases are due to high skeleton multiplicity.

11.5. To be precise, in fact between Field and Ring theory in the Casl library there are some other intermediate libraries, but this is irrelevant for our considerations.

11.6. Only very few parts of the MMLare beyond first-order.

The experiments of theory intersection were restricted as proof of concept to the CASL library. Here the major source of problems for theory intersection is the theory `Nat` of natural numbers, because it introduces digits. As digits were a source of problem for theory interpretation search, we could expect similar problems here. Any trial to intersect two theories importing (possibly indirectly) `Nat` is deemed to fail with our algorithm for similar reasons as discussed above.

However, in cases of theories with few duplicate skeletons the intersection can be computed quite fast. We want to discuss four paradigmatic cases from the CASL library (whose theories where exported to SoftFOL beforehand):

1. The intersection of the theory IntegralDomain from the Algebra_I library with BooleanAlgebra from the RelationsAndOrders library. The IntegralDomain contains 14 formulae (always in SoftFOL format) and the BooleanAlgebra contains 21 formulae. The intersection can be computed on a contemporary PC within 0.1 seconds; and it contains 9 formulae.

2. The intersection of the theory EuclidianRing from the Algebra_II library with RichFreeAlgebra from the LinearAlgebra_II library. The RichFreeAlgebra is one of the largest CASL theories at all, it contains 536 formulae whereas the EuclidianRing contains only 24 formulae. The intersection for them takes 12.6 seconds and it has 9 formulae.

3. The intersection of theory RichFreeAlgebra (contains 536) with the even one magnitude larger theory Merge (containing 3399 formulae) which is part of the SUMO ontology. Computing the intersection, which has seven formulae, took only 14.5 seconds.

These examples indicate that in case of small theories with few duplicate skeletons the intersection can be computed very fast. As soon as one of the theories is big the intersection computation slows down, but is still acceptable. The last example demonstrates that even very large theories can be intersected in acceptable time. Theory size on its own is not the primary criterion for the time cost of intersection. Much more critical is the skeleton multiplicity. A typical example of a theory with a high skeleton multiplicity is the Vector theory from the LinearAlgebra_I library. Computing the intersection of the Vector theory with itself fails due to stack overflow – although Vector contains only 94 formulae.

11.4 Discussion

We conclude that the computational complexity of automated theory intersection has the practical consequence that some theories are problematic for automated theory intersection and that theories with a high skeleton multiplicity are likely problematic. One of the envisioned applications of theory intersection were ontologies formalized in description logic[11.7]. Some preliminary experiments have shown that most of the investigated ontologies are problematic theories. This can be explained by their usually high skeleton multiplicity. One reason might be seen in the syntactic restriction of description logic – for instance that it allows only binary relations – such that it even does not allow for such a variety of skeletons as first-order logic.

11.7. An introduction to descriptions logics as well as a survey on their applications and tools can be found in [4].

Although it is not possible to compute in all cases the optimal solution for an intersection – in the sense of maximum clique – it is always possible to relax the goal and compute smaller cliques. There are many ways to reduce the complexity in this manner. The following might be considered practically useful from a user perspective:

- Whenever the computer finds a clique that is bigger than a beforehand found clique to the same problem then present it to the user. Thus the user sees useful results immediately – also in case of problematic theories – as stream of increasing cliques until a maximum is reached.
- Let the user constrain the admissible renamings, e.g. usually digits should not be renamed. In particular in case of poorly axiomatized ontologies a renaming constraint is desirable.
- Both approaches could be combined: a newly popped up clique may give the user a idea what renamings should be excluded and which should be kept for the subsequent computed cliques.
- In combination with the renamings the user may also decide interactively which formulae should be part of the final intersection once some possible formulae candidates are computed.

All these ideas would require a non-trivial user interface. From the backend perspective, however, an improvement could be expected if the maximum clique algorithm implemented in Haskell (based on the functional graph library fgl[11.8]) would be replaced by an optimized maximum clique algorithm of which several are implemented in C.

11.8. fgl can be downloaded at http://web.engr.oregonstate.edu/~erwig/fgl/haskell/

Chapter 12
Conclusion

We summarize our main results, list open problems and possible extensions.

In Chapter 2 we introduced intuitively the central notions of logics relevant for this thesis – in particular: *derivability system, theory, theory interpretation*. We characterized the role of theory of interpretations in history of mathematics and later on in the area of mechanized reasoning.

In Chapter 3 we introduced the idea of *little theories* and the corresponding data structure called *development graph* that has theories as nodes and theory imports and theory interpretations as edges. We gave a representative overview of computer systems supporting the management of theories organized in a development graph. We identified as shortcoming of these systems that all the theory interpretations must be discovered by humans which triggered the core idea of this research: *automated theory interpretation* and based on that *theory intersection* and *theory completion*.

In Chapter 4 a rigorous definition of the concepts *derivability system* and *theory interpretations* was presented. Our definition of *derivability system* is a refinement of what is called *entailment system* in [38] in that we added the constraint of $\cap$*-invariance*. This was motivated by the fact that in many formal languages formulae are constructed inductively over the (logical and non-logical) symbols. For this principle it holds that the formulae constructable from the intersection of two symbol sets is the intersection of formulae constructed from these two symbol sets. The $\cap$*-invariance* is an important assumption for various definitions and theorems, in particular for the definition of a *minimal translation* which is a translation of a single formula which is defined on no other symbols then those occurring in this formula. Based on these minimal translations we developed an abstract algorithm for theory interpretation search: in the first step for each axiom of the source theory all minimal translations to formulae in the target theory are calculated, in the second step all compatible minimal translations are determined and finally merged to the resulting theory morphisms – if they exist (it does not exist if there are two source axioms such that all there minimal translations are pairwise incompatible).

In Chapter 5 we investigated two theories and their shared axioms that gave us an idea of what *theory intersection* intuitively means and finally gave a rigorous definition of theory intersection which is essentially the (possibly not unique) largest theory included in both target theories. Unfortunately, to determine an intersecting theory for two given target theories is an undecidable task. For explored theories, however, it is an approximated solution to restrict the intersection on the given formulae of the target

theories and to consider their theory intersection as a set of formulae which is isomorphic to a maximal set of compatible minimal translations between the involved theories. We identified the intersection task then as maximum clique problem from graph theory where nodes are minimal translations and edges their compatibility. We concluded the chapter with an illustration of the intersection algorithm on the introductory example theories.

In Chapter 6 we specified a simple, but general formal language, namely untyped λ-terms with variables and two kinds of constants: logical and non-logical – the latter are called "parameter". We introduced parameter and variable mappings, where we required that the latter are one-to-one. Any compositions of these two kinds of mappings form our symbol morphisms.

The motivation behind the chosen language is the conviction that it is a formal language with the fewest assumptions such that most formal logical languages could be mapped into. This is, of course, only a thesis since there is no general syntactical definition of what a formal logical language is. To elaborate this thesis further: all existing logical formal languages are probably either isomorphic to such λ-terms (i.e. equal modulo some syntax features like infix instead of prefix notation) or they are restrictions of such isomorphic languages due to additional constraints like type constraints.

Crucial at this point is the assumption that the syntax of all these languages (including the constrained languages) follow some inductive rules that construct their sentences out of atomic entities like types, variables, parameters, constants, quantifiers, binders, etc. and that due to this inductive principle the $\cap$-invariance property holds. Since this property, which we have added to the derivability-system definition, is so important for the whole work, our particular chosen formal language is regarded as a paradigmatic instance of a formal language for which the $\cap$-invariance property obviously holds. But it is open to further investigation what kind of (type)constraints could compromise this $\cap$-invariance property.

In Chapter 7 we introduced the *renaming problem* and investigated how theory interpretations can be found by solving the renaming problem. We distinguished the *simple* and the *equational renaming problem* (or *renaming modulo equality*). We mentioned how they are related to the more general matching and unification problems and argued that for the simpler renaming problem specialized techniques can be used that solve this problem significantly faster than with conventional matching techniques. The main idea is a very efficient filter technique: if two formulae match by parameter renaming then they must have the identical skeletons (which can be checked almost in constant time). So if we want to find a formula modulo renaming in a formalized library of millions of formulae then we can reduce the search space with the skeleton filter significantly. For that, all formulae in a library must be divided into their skeleton and their parameters – we defined this process under the name *formula abstraction*. The computation of the skeleton must also regard α-equivalence. To solve this problem we already introduced at this point a very general notion of *standard form* (or *skeleton*): Given a equivalence relation on formulae, the standard form of a formula is the least element of induced equivalence class with respect to a (arbitrary but fixed) term ordering.

We described two algorithms to solve renaming problems, one for the simple renaming problem and one for the equational renaming problem for the concrete instance of equivalence modulo associativity and commutativity (AC). For the former

we showed that once two formulae are checked to have identical skeletons, a signature morphism between them can be derived from their parameter list if and only if it exists. Solving the AC-renaming problem is very useful as many formulae contain AC-subterms. For such AC-terms the order of its arguments is arbitrary. Simple renaming techniques fail to match such terms unless they are by accident in the appropriate order. But already with few AC-subterms this becomes rather unlikely. In contrast to the simple simple renaming problem we have in the AC case in general more than one parametrization per AC-skeleton. This reflects the fact that we also have in general more than one parameter renaming to match two formulae modulo AC. With the AC-skeleton we have again a very efficient skeleton filter technique at hand, but now to filter modulo AC. And analogously with the AC-parametrization alone the signature morphism modulo AC can be computed if and only if two formulae of identical AC-skeleton are AC-equal.

One of the most challenging problems, however, was to find an algorithm to compute AC-standardization (= AC-skeleton together with their corresponding AC-parametrization) for which a solution was presented.

In Chapter 8 we focused on formula equality relying on pure logical equivalences (in contrast to AC-equality). This is relevant to semantically match formulae which differ contingently like the two equivalent propositions $a \wedge b \Rightarrow c$ and $a \Rightarrow (b \Rightarrow c)$. These logical equivalences naturally depend on concrete logics. We considered a simple logic having the most common logical connectives and quantifiers that can be (possibly implicitly) found in most logics.

The method to accomplish parameter matching modulo logical equivalences is mostly adopted from term rewriting for which we provided some preliminaries. All logical equivalences that we have considered were formulated as rewrite rules (RS). From term rewriting we know that two expressions are equal in a equational theory if there is a convergent RS of directed equalities in that theory. The unique result of an exhaustive application of convergent rewrite rules is called *normal form* which we were after. Automated theorem proving also has a notion of normal form. However, in that notion an expression is already called in normal form if it is the result of a terminating RS – the second condition for convergence, namely confluence, is not there concern. Thus the well known RSs to transform formulae to conjunctive (or disjunctive) normal form in fact do not determine a unique computation.

It was the major challenge in the research of this chapter to explore to what extent uniqueness can be achieved. Hereby uniqueness was relaxed to uniqueness modulo AC, because AC was already covered by the above mentioned AC-standardization. To achieve this goal we specified at first several equivalence classes according to particular logical equivalences. For each of these classes we were able to specify a corresponding convergent RSs. Based on that the ideal goal was to specify a convergent RS for the symmetric-transitive closure of these equivalence classes (which is so to say the big equivalence class induced by the many small equivalence classes). Though we have not achieved this goal, we were able to bundle our RSs into two big RSs both of them being convergent. The union of these two convergent RSs, however, does not form a convergent system. Yet, this does not mean that this overall RS is useless – it only means that our RS is not able to identify all members of the intended equivalence class. A typical example where our RS fails to show equivalence is the the equivalence of $\forall x.\exists y.P(x) \vee \neg P(y)$ with $\top$. It remains an open problem how to complete our RS (and standardization techniques) to cover such problematic equivalences. On the other hand all found problematic formulae seem to be rather contrived anyway.

In Chapter 9 we extended the idea of automated theory interpretation search in two ways: 1) theory completion, i.e. the repeated application of theory interpretation search which we identified as a classical forward chaining task from logical programming. 2) We motivated the notion of sequent that as a fragment of a theory. And argued that sequents potentially lead to more knowledge gain than the bigger theories can.

In Chapter 10 we describe how the algorithms investigated in the theoretical part can be actually implemented. The preferred programming language was functional language Haskell as it allows a comparable compact and concise coding which is very close to a mathematical jargon. After a short tutorial to Haskell in a nutshell we presented a didactic system implementation for most of the discussed algorithms. Hereby we ignored obvious efficiency techniques in favor of a short and readable presentation of the basic implementation ideas which could be used as starting point for own reimplementation. We briefly sketched the features the envisioned system should have apart from what is presented in the didactic prototype to make it practically more usable. This includes in particular interfaces to the user, to the HETS system, and to a database. We conclude the chapter with a summary of what has been actually implemented in the final prototype that has been used for the experiments.

Chapter 11 is dedicated to experiments with the implemented prototype system. The major testbed was an untyped first-order version of the Mizar library. It was chosen, because it is the largest library of formalized mathematics and because it was comparable easy to implement a parser for the given format. One major goal was successfully accomplished with the proof of concept for scalability of the prototype system: given a source theory the system finds within 1.14 seconds on average hundreds of theory interpretations to the 41759 theories in the library. The crossproduct search has shown a high potential of theorem reuse (altogether more than 2.5 million theory interpretations). However, this gives rise for the unaccomplished challenge: which of these interpretations are interesting from a mathematicians perspective? To answer this question, a back translation from SoftFOL to Mizar would be needed and a good heuristic to reduce the search space. We sketched the idea to focus on interpretations between distant theories as well as the idea of an improved interface to control large amounts of search results.

We discussed the general limits of automated theory intersection due to the inherent complexity of this operation and presented typical examples of failure. The general reason is as in theory interpretation search high skeleton multiplicity. We have shown that in case of low skeleton multiplicity (i.e. not more than five axioms with identical skeletons) intersections of theories with small size (about 20 formulae) can be calculated in a tenth of a second. Even intersection between large theories (thousands of formulae) can be computed within few seconds as longs as the skeleton multiplicity is low. We proposed some approaches how to cope with the complexity of intersection computation. Two basic ideas are: 1) not to wait for an optimal solution of intersection, but to present the user with a stream of currently best solution while the algorithm continuous search better solutions; and 2) to let the user narrow down the search space by setting constraints like which renamings should be admissible and which formulae should be considered for the construction of an intersection. In particular in the area of ontology matching such user interaction features would be very desirable.

12.1 Outlook

Automated knowledge reuse or discovery of common content across libraries remains an open task. Hets probably provides the best infrastructure to integrate most of the formal libraries. First of all it is a matter of programmer resources to implement all the parsers for the various logic formats and the translaters (institution comorphisms) between them. On the other hand it is not a straightforward task for all logic formats. In general there is not a unique way to translate from one format to another. Though correct logic translations preserve the semantics of theories in model theoretic sense, they can vary in the degree of information loss: no loss of information means that the process of translation followed by retranslation is an identity operations which is almost never the case. In some cases a satisfactory retranslation is probably very difficult, as for instance from SoftFOL to Mizar. If such kind of information problem would not be a problem, then it would be sufficient to have the theory intersection and interpretation search implemented only for one logic. The involved theories would be translated to that logic (provided it is expressive enough) and the results would be translated back to the source logic of the involved theories. It is open to further research to what extent this approach is reasonable. The alternative would be to implement these search algorithms for several logics. At least for Casl this is already planned.

However, this calls for research in theory, since Casl an its relatives are languages whose signatures are not simply sets of symbols in contrast to what we assumed in our definition of a entailment system (cf. definition 4.3). Hence a more abstract notion of signature intersection is needed. Following the categorical path (as the notions of institution and entailment systems) the most promising approach is probably via the concept of *inclusion systems* as presented in the book *Institution independent Model Theory* [14].

A different issue is the storage of large formal libraries. At the current stage this aspect is not well supported by the Hets system, but efforts in this direction are already initiated. A first important step, already accomplished to a great part, is the import and export of theory graphs from and to OMDoc – a semantic markup format for mathematical documents [36]. For this format a dedicated database is currently under development[12.1]. It is an open question, how the specialized index for theory interpretation search would cooperate with this database.

From the user perspective, the most important aspect that has to be improved is a graphical user interface (GUI) for our algorithms. A first prototype was already implemented as a master student's research project. The main lesson learned from that project was a list of desirable features for a GUI. Here we want to mention only the most important (cf. for more details section 11.4):

1. Incremental presentation of search results in case of large result sets,
2. interactive search space reduction by user defined constraints on admissible renamings, and
3. integration of a theory editor to write query theories conveniently

The second point is particularly interesting in the area of ontology engineering where the user typically has an idea beforehand which concept renamings are intended and which are not.

12.1. http://www.mathweb.org/wiki/OMBase

Bibliography

[1] A. Mostowski A. Tarski and R.M. Robinson. *Undecidable Theories*. North Holland, 1953.

[2] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, second edition, 2002.

[3] Serge Autexier, Dieter Hutter, Till Mossakowski, and Axel Schairer. The development graph manager MAYA (system description). In Hélene Kirchner, editor, *Proceedings of 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*. Springer Verlag, 2002.

[4] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook –- Theory, Implementation and Applications*. Cambridge University Press, 2003.

[5] Clemens Ballarin. Interpretation of locales in isabelle: Theories and proof contexts. In *MKM*, pages 31–43, 2006.

[6] Richard Bird. *Introduction to Functional Programing using Hasekll*. Prentice-Hall, second edition, 1998.

[7] R. M. Burstall and J. A. Goguen. Semantics of CLEAR, a Specification Language. In D. Bjorner, editor, *Tutorial: Software Reusability*. Springer-Verlag, 1980.

[8] Rod M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *IJCAI*, pages 1045–1058, 1977.

[9] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[10] CoFI (The Common Framework Initiative). CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004.

[11] H. B. Curry. *Foundations of Mathematical Logic*. McGraw-Hill, 1963.

[12] Nicolaas G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, pages 29–61, 1970.

[13] Nachum Dershowitz. When are two rewrite systems more than none? In *MFCS'97*, volume 1295 of *LNCS*, pages 37–43. Springer-Verlag, 1997.

[14] Rasvan Diaconescu. *Institution independent Model Theory*. Birkhäuser, 2008.

[15] Francisco Durán and José Meseguer. Structured theories and institutions. In M. Hofmann, G. Rosolini, and D. Pavlović, editors, *Proceedings of 8th Conference on Category Theory and Computer Science, Edinburgh, Scotland, September 1999*, volume 29 of *entcs*, pages 71–90. Elsevier, 1999.

[16] William Farmer, Josuah Guttman, and Xavier Thayer. Little theories. In D. Kapur, editor, *Automated Deduction –- CADE-11*, volume 607 of *LNCS*, pages 467–581, Saratoga Springs, NY, USA, 1992. Springer Verlag.

[17] William M. Farmer. Theory interpretation in simple type theory. In *HOA'93, an International Workshop on Higher-order Algebra, Logic and Term Rewriting*, volume 816 of *LNCS*, Amsterdam, The Netherlands, 1993. Springer Verlag.

[18] William M. Farmer. An infrastructure for intertheory reasoning. In *Automated Deduction/CADE-17*, pages 115–131. Springer-Verlag, 2000.

[19] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[20] William M. Farmer and Martin Mohrenschildt. An overview of a formal framework for managing mathematics. 38:165–191, 2003. special issue of Annals of Mathematics and Artificial Intelligence.

[21] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer, NY, second edition, 1996.

[22] Tobias Nipkow Franz Baader. *Term Rewriting and All That*. Cambridge University Press, first edition, 1999.

[23] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[24] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992. Predecessor in: LNCS 164, 221–256, 1984.

[25] J.A. Goguen. OBJ as a Theorem Prover with Applications to Hardware Verification. Technical Report SRI-CSL-88-4R2, SRI International, August 1988.

[26] Joseph A. Goguen and Rod M. Burstall. Introducing institutions. *Lecture Notes in Computer Science*, 164:221–256, 1984.

[27] Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer, 1996.

[28] J.V. Guttag, J.J. Horning, and J.M. Wing. The LARCH Family of Specification Languages. *IEEE Software*, 2:24–36, September 1985.

[29] Reiner Hähnle, Manfred Kerber, and Christoph Weidenbach. Common Syntax of DFG-Schwerpunktprogramm "Deduktion". Interner Bericht 10/96, Universität Karlsruhe, Fakultät für Informatik, 1996.

[30] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987*, pages 194–204. IEEE Computer Society Press, New York, 1987.

[31] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge University Press, 1986.

[32] Jieh Hsiang and Guan Shieng Huang. Some fundamental properties of boolean ring normal forms. `http://mack.ittc.ku.edu/143687.html`.

[33] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.

[34] K. Kammüller. Modular reasoning in Isabelle (PhD thesis), 1999.

[35] Claude Kirchner, editor. *Unification*. Academic Press, London, 1990.

[36] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.

[37] Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976. Edited by John Worrall and Elie Zahar.

[38] J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.

[39] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set, Habilitation Thesis, 2005.

[40] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Bernhard Beckert, editor, *VERIFY 2007*, volume 259 of *CEUR Workshop Proceedings*. 2007.

[41] Immanuel Normann. Enhanced theorem reuse by partial theory inclusions. pages 40–52.

[42] Immanuel Normann and Michael Kohlhase. Extended formula normalization for ϵ-retrieval and sharing of mathematical knowledge. pages 266–279.

[43] Lawrence C. Paulson. Isabelle reference manual. Technical report, Computer Laboratory, University of Cambridge, oct 2005.

[44] Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, Marktoberdorf Summer School, 2002.

[45] Frank Pfenning and Carsten Schürmann. System description: Twelf –- a meta-logical framework for deductive systems. In *Proceedings of the 16th Conference on Automated Deduction*, pages 202–206, 1999.

[46] R. Rogers. *Mathematical logic and formalized theories*. North-Holland, 1971.

[47] Piotr Rudnicki. An overview of the mizar project. In *Proceedings of the 1992 Workshop on Types and Proofs as Programs*, pages 311–332, 1992.

[48] Piotr Rudnicki, Christoph Schwarzweller, and Andrzej Trybulec. Commutative algebra in the Mizar system. *Journal of Symbolic Computation*, 32:143–169, 2001.

[49] Donald Sannella and Rod M. Burstall. Structured theories in LCF. In *CAAP*, pages 377–391, 1983.

[50] Razvan Diaconescu Till Mossakowski, Joseph Goguen and Andrzej Tarlecki. What is a logic? In Jean-Yves Beziau, editor, *Proceedings of First World Conference on Universal Logic*, pages 113–133, 2005.

[51] Josef Urban. Translating mizar for first-order theorem provers. pages 203–215.

[52] F. J. Thayer W. M. Farmer, J. D. Guttman. Imps theory library.

[53] Markus M. Wenzel. *Isabelle/Isar –- a versatile environment for human-readable formal proof documents*. PhD thesis, Institut fur Informatik, TU München, 2002.

[54] Freek Wiedijk. Mizar: An impression, 1999. `http://www.cs.kun.nl/~freek/notes`.

[55] Freek Wiedijk. *The Seventeen Provers of the World*. Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence. Springer-Verlag, New York, 2006.

[56] P. Windley. Abstract theories in HOL. In *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, volume A-20 of *IFIP Transactions*, pages 197–210, Leuven, Belgium, September 1992. North-Holland/Elsevier.

Index