Automated Theorem Proving in Higher-Order Logic

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

2020

Ahmed Bhayat

School of Computer Science

Contents

List of Figures **5**

List of Tables **6**

Declaration **7**

Copyright **8**

Abstract **9**

Acknowledgements **11**

1 Introduction **12**

1.1 Automated Theorem Proving 13

1.2 Higher-Order Logic 16

1.3 Vampire 17

1.3.1 Features of Vampire 18

1.4 Thesis Contributions 19

1.5 Thesis Structure 20

1.6 Published Papers 21

2 Technical Preliminaries **22**

2.1 The Syntax of First-Order Logic 22

2.1.1 Types 22

2.1.2 Terms 23

2.1.3 Formulas 24

2.2 The Semantics of First-Order Logic 25

2.3 The Syntax of Applicative First-Order Logic (Clausal) 26

2.3.1 Types 26

2.3.2 Terms 27

2.4 The Semantics of Applicative First-Order Logic . . . 28
2.5 The Syntax of Higher-Order Logic (Clausal) . . . 29
2.5.1 Terms . . . 29
2.6 The Semantics of Higher-Order Logic . . . 30
2.7 Correspondence . . . 30
2.8 Term Orders . . . 34
2.8.1 First-Order Knuth-Bendix Order (KBO) . . . 35
2.8.2 Higher-Order Knuth-Bendix Order . . . 36
2.9 Proof Calculi . . . 37
2.10 First-Order Superposition . . . 37
2.10.1 Redundancy Criteria . . . 38
2.10.2 Refutational Completeness . . . 39
2.10.3 Simplification Inferences . . . 43
2.11 Applicative First-Order Superposition . . . 44
2.11.1 Refutational Completeness . . . 45
2.12 Combinatory Logic . . . 47

3 A Modified Knuth-Bendix Order **50**
3.1 The Challenge . . . 51
3.2 The Combinatory Compatible KBO . . . 53
3.3 Properties . . . 56

4 Combinatory Superposition **66**
4.1 The Calculi . . . 67
4.1.1 Term Order . . . 67
4.1.2 Inference Rules . . . 68
4.1.3 Extensionality . . . 70
4.2 Examples . . . 72
4.3 Redundancy Criterion . . . 73
4.4 Refutational Completeness . . . 76
4.4.1 Candidate Interpretation . . . 77
4.4.2 Liftable Inferences . . . 78
4.4.3 Non-liftable Inferences . . . 82
4.4.4 Higher-Order Model Construction . . . 88
4.4.5 Dynamic Refutational Completeness . . . 93
4.5 Removing Combinator Axioms . . . 96

5 Extensions to the Calculi **98**
5.1 Dealing with Booleans . . . 98
5.1.1 The Syntax of Applicative First-Order Logic (Non-clausal) . . . 98

5.1.2 The Semantics of Applicative First-Order Logic (Non-clausal) . . 99
5.1.3 Axiomatisation . . . 100
5.1.4 Inference Rules . . . 101
5.2 Reasoning with Combinators . . . 108
5.3 Examples . . . 109
5.3.1 SET557^1.p . . . 109
5.3.2 SYN997^1.p . . . 110
5.3.3 SYO252^5.p . . . 110

6 Vampire Implementation 112
6.1 Preprocessing . . . 112
6.2 Terms, Literals and Clauses in Vampire . . . 113
6.3 Sorts and Types . . . 115
6.4 Indexing Data Structures . . . 115
6.4.1 Substitution Trees . . . 116
6.5 Given Clause algorithm . . . 119
6.6 Updating Vampire to Support Polymorphism . . . 121
6.7 Supporting Higher-Order Logic . . . 123
6.7.1 Applicative Encoding . . . 123
6.7.2 Flat Representation . . . 124
6.7.3 Vampire's Implementation . . . 124
6.8 Implementing the Combinatory Superposition-a and -b Calculi . . . 125
6.8.1 Implementing the Modified KBO . . . 125
6.8.2 Implementing the Inference Rules . . . 126
6.9 Higher-Order Schedule . . . 131

7 Evaluation 134
7.1 Polymorphic First-Order . . . 135
7.1.1 Applicative First-Order . . . 136
7.1.2 Clausal Higher-Order* . . . 138
7.1.3 Full Higher-Order* . . . 139
7.1.4 Evaluating Options . . . 140

8 Conclusion, Philosophical Implications & Further Work 142
8.1 Related Work . . . 143
8.2 Future Directions . . . 144

A Extended Proofs 156

List of Figures

1.1 Given clause algorithm 18

3.1 Self superposition with **S**-axiom. The subterms highlighted in red are the superposed subterms. The variables of the left and right premises have been renamed apart to avoid confusion. 51

5.1 SET557^1.p 109
5.2 SYN997^1.p 110
5.3 SYN997^1.p 110

6.1 Representation of terms in Vampire 114
6.2 How the term $\mathsf{f}(x_1, \mathsf{a}, x_0, \mathsf{a}, x_0)$ is stored in memory by Vampire 116
6.3 A substitution tree 117
6.4 Sample rank-1 polymorphic problem 122
6.5 Hashed substitution tree 130

7.1 Time taken by Vampire to prove SET problems 137

List of Tables

7.1 Number of non-arithmetic first-order (TF1, TF0, FOF, CNF) problems proved 135
7.2 Number of proofs found by Vampire on FOF and TH0 representations of problems 137
7.3 Number of applicative first-order problems proved. 138
7.4 Average time (in seconds) taken to solve applicative first-order problems. 138
7.5 Number of clausal higher-order problems proved. 139
7.6 Number of higher-order monomorphic and polymorphic problems proved. 140
7.7 Impact of various higher-order parameters 141

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Abstract

Proof assistants are rapidly gaining in importance and traction. Due to the complexity of modern mathematics, a number of mathematicians have advocated the use of proof assistants, most famously field medalist Vladimir Voevodsky. There are currently projects underway to encourage collaboration between mathematicians and proof assistant designers such as the Lean Forward project. However, one of the largest stumbling blocks to the uptake of proof assistants is their lack of automation. A single line of a pen and paper proof can translate to many lines of a formal proof. A major weakness in existing automation is the absence of strong higher-order automated theorem provers. Automated higher-order reasoning has long lagged behind its first-order counterpart. This is damaging since many proof assistants use higher-order logic as their core language. In this thesis, I explore methods of extending first-order theorem provers to support higher-order reasoning.

My work involves reasoning about the combinatory calculus, a version of higher-order logic that does not utilise binders. I present a novel term ordering for combinatory terms that orients all combinator axioms left to right. I then use this ordering to parameterise a modified version of the superposition calculus. I prove the calculus to be sound and complete for the clausal (Boolean free) fragment of combinatory logic. Due to the absence of binders in combinatory logic, the calculus I have developed can be implemented in a first-order prover relatively easily.

I have implemented the calculus in the leading first-order theorem prover Vampire. I discuss the implementation and describe the modifications required to Vampire's data structures and algorithms. Vampire's performance as compared with other leading higher-order provers displays promise.

عَلَّمَ الْإِنسَانَ مَا لَمْ يَعْلَمْ

He taught man what he knew not

المنطق للجنان نسبته كالنحو للسان

Logic is to the mind what grammar is to the tongue

Acknowledgements

The research underpinning this thesis and the thesis itself would have been impossible without the assistance and facilitation of many. I offer my thanks and gratitude to my supervisor Giles Reger for always being available to answer queries and support me. A PhD is rarely straightforward, but even at the most challenging of times, he was always ready with suggestions, comments and counsel. Without his expertise, coming to grips with the labyrinth that is Vampire's code base would almost certainly have been impossible.

A special thanks to Jasmin Blanchette who funded me to join the Matryoshka team for two weeks. Those weeks proved crucial to my research. Dr. Blanchette has also assisted me in so many other ways, that they cannot be recorded here. This thesis owes a debt to his aid and guidance without which it would look very different.

A thanks to the whole Vampire team past and present including Michael Rawson, Martin Suda, Martin Riener and Evgeny Kotelnikov. I learnt something from all of them. Vampire may be the dark side of theorem proving, but its members have a light side. A special thanks to Andrei Voronkov for accepting me as a PhD student and thereby starting this journey. I regret that we were not able to work together more.

Likewise, I owe thanks to the entire Matryoshka team. In particular, I benefited from conversations and collaboration with Alexander Bentkamp and Petar Vukmirović. Moreover, I am grateful to the wider theorem proving community and in particular to Geoff Suttcliffe who is so crucial a part of it.

The research community at the university of Manchester helped make the PhD an enjoyable experience. Thanks to my fellow PhD students and in particular Ahmed Alghamdi who I happily shared an office with for three years.

A number of teachers and mentors were inspirational and crucial in getting me to the starting line. I do not have the space to name them, but thank them all.

I proffer the sincerest thanks to the Elson family and Mary Elson in particular. It is via their kind help and assistance that this research has been possible.

I thank my family. It is probably from my father that I inherit my love of mathematics and logic. Along with him, thanks to my brother, my sisters and my wife.

Last, but certainly not least, I thank my mother who was my first teacher. The grammar of this thesis owes much to her diligence. I dedicate the thesis to her.

Chapter 1

Introduction

> For him who seeks the truth there is nothing of higher value than truth itself.
>
> *Al-Kindi*

Automated theorem proving is an important sub-field of computer science. For obvious reasons, it has strong links with logic and has benefited from, and driven development in the field. Automated theorem proving has practical uses in software verification and the formalisation of mathematics. Automated theorem provers (ATPs) are used as backends for interactive theorem provers (ITPs). Substantial efficiency gains have been achieved in writing formal proofs through the use of ATPs. For example, automated provers are able to prove some 60%+ of the 1268 goals in the Isabelle/Sledgehammer Judgement day benchmark suite [111]. Similarly, HOL(y)Hammer is able to prove some 47% of the goals in the much larger Flyspeck corpus [76]. Most hammers work by translating the logic of the ITP, commonly some form of set theory or higher-order logic (HOL), into the various logics supported by the automated provers, commonly some form of first-order logic [86]. The translations from higher-order logic are a limiting feature in the usefulness of hammers. The resulting first-order problems tend to be very different from the first-order problems that many ATPs have been tuned to. Furthermore, the translations rarely, if ever, are complete and are at times unsound. It is therefore of great practical use for the users of hammers to have access to better higher-order ATPs.

However, it is undoubtedly the case that automation in higher-order logic has lagged behind its first-order counterpart, and far behind automation in propositional logic. SAT solvers and SMT solvers have a wide range of practical and industrial usages [67]. On the other hand, first-order and particularly higher-order ATPs have, for the most part, yet to see practical usage, though some industrial uses are springing up.

The focus of the research discussed in this thesis has been on improving high-order automation. The primary vehicle for the research has been the Vampire theorem prover [83]. In the course of my research, I have updated both the theoretical underpinning

and the underlying algorithms and data-structures of Vampire to deal with higher-order logic. This is significant because Vampire is one of the most successful first-order theorem provers [113]. Vampire is based on the superposition calculus for first-order logic (FOL) with equality. Due to theoretical difficulties, superposition had not, till recently, been extended to higher-order logic. It has now been extended to HOL in two ways one of which is discussed extensively in this thesis.

1.1 | Automated Theorem Proving

Automated theorem proving has a long history. I present an overview of it here based primarily on the work of Bibel [34], Davis [49], MacKenzie [84] and Benzmüller and Miller [26]. Interested readers are encouraged to peruse these resources for more detail.

The theoretical background to automated deduction can be traced back to Leibniz [48] via Turing [114] and Church [45]. Practical attempts at automated deduction began in the 1950s. The early attempts fell into two broad categories. Some attempted to replicate human reasoning within a computer through the use of heuristics with little consideration for completeness, an example being the Logic Theory Machine developed by Newell and Simon in 1954 [34]. Others abandoned using human-like inferences in favour of inferences more easily suited to computers. One of the early pioneers in this area was Dag Prawitz, who designed and implemented a tableaux calculus [89] for proving theorems in the predicate calculus. This early work already contained many of the ideas that would go on to dominate the field of automated deduction. These included the use of Herbrand's famous theorem to search for a proof on the ground level. Prawitz's work included a forerunner of unification, close enough to its modern formulation for Bibel to credit him with the introduction of unification into automated reasoning [34, p. 7]. Unification is more commonly attributed to Robinson (e.g. [17]) and the seeds of the unification procedure can be traced to Herbrand [66].

A crucial step in the automation of first order reasoning taken around this time, was the introduction of Skolemization to remove existential quantifiers. However, the field only truly became active after Robinson introduced the resolution rule for first-order logic [98]. The resolution rule combined ideas introduced by Prawitz and others into a single inference rule that was easily implementable. Resolution, along with an inference known as factoring, form a calculus complete for first-order logic. The resolution rule can be given as follows:

$$\frac{C' \vee A \qquad D' \vee \neg B}{(C' \vee D')\sigma}\ \textsc{Resolution}$$

where σ is the *unifier* of A and B. The resolution calculus was refined by researchers at the Argonne National Laboratory. Important improvements made to the calculus included the introduction of the unit preference and set-of-support strategies [129]. One of the main

weaknesses with resolution was its inability to deal with equality efficiently. One method of dealing with equality is to treat it as an uninterpreted predicate symbol and axiomatise it. The following set of three axioms and two axiom schemas are necessary to completely axiomatise equality ($\approx$):

$$x \approx x$$

$$x \not\approx y \vee y \approx x$$

$$x \not\approx y \vee y \not\approx z \vee x \approx z$$

$$x_1 \not\approx y_1 \vee \ldots \vee x_n \not\approx y_n \vee \mathsf{f}(x_1, \ldots, x_n) \approx \mathsf{f}(y_1, \ldots, y_n)$$

$$x_1 \not\approx y_1 \vee \ldots \vee x_n \not\approx y_n \vee \neg\mathsf{p}(x_1, \ldots, x_n) \vee \mathsf{p}(y_1, \ldots, y_n)$$

However, resolution and factoring tend to be prolific in the presence of these axioms, hindering proof search. This led researchers to look for methods of proving in first-order logic with equality, i.e., of treating equality as a logical symbol. Once again, Robinson and Wos lead the way with the introduction of the paramodulation inference [97]. Paramodulation is the replacement of equals with equals and can be given as:

$$\frac{C \vee t \approx t' \qquad D[u]}{(C' \vee D[t'])\sigma} \textsc{Paramodulation}$$

where σ is the unifier of t and u. Paramodulation along with resolution and factoring is complete for first-order logic with equality. However, in the form it was introduced in, and that is given above, it is explosive and produces a large number of unnecessary clauses. Research continued at Argonne and elsewhere into refining paramodulation. It was shown that paramodulation into variables was unnecessary for completeness. Theoretically, practically and in the context of this thesis, the most important refinement to be introduced was that of *orderings*. An ordering is an irreflexive, transitive relation. Paramodulation was parameterised with an ordering over terms such that only larger terms are replaced by smaller ones. Various refinements of this concept led to superposition, paramodulation involving only the maximal terms in maximal literals of a clause. Superposition was proved complete by Bachmair and Ganzinger using a powerful proof technique called *model generation* [8]. Provers developed in the machine-oriented tradition discussed here have had some notable successes. Most famously, William McCune used the paramodulation based EQP prover to prove the Robbins conjecture [85]. More recently SAT solvers, rather than theorem provers for first-order logic, have been involved in the resolution of a number of open problems [41].

In parallel to the work carried out on automating first-order theorem proving, research was undertaken into automating higher-order theorem proving. According to some, higher-order logic was formalised prior to first-order in the work of Frege [58]. However, due to its more complex meta-theory, attempts to automate reasoning in it began later.

When work did begin on the automation of higher-order logic, most of it was based on Church's simple type theory [45] with support added for reasoning about extensionality and choice.

One of the challenges linked to automating higher-order reasoning is that many of the staple features of automated first-order reasoning no longer hold. For example, in the presence of predicate variables, it is no longer sufficient to transform a formula to conjunctive normal form (CNF) once, prior to proof search. Consider, for example, the CNF formula:

$$x\,\mathsf{a} \approx \top \vee t_1 \approx t_2$$

the variable x could well be instantiated by a term containing logical connectives during proof search resulting in a formula that is not in CNF. Challenges such as these meant that it was the mid 1960s before work began on developing the first automated prover for higher-order logic, the TPS system developed by Peter Andrews and his students [3]. TPS did not build directly on first-order proof methods, instead utilising a connection or mating calculus.

Andrews was a pioneer in extending resolution to simple type theory [4]. His work did not make use of higher-order unification instead relying on an unguided substitution rule. Huet introduced the concept of higher-order pre-unification [72] and developed a resolution calculus based on it [71]. Around a similar time, Jensen and Pietrzykowski developed a higher-order resolution calculus based on full higher-order unification [73].

Two of the early higher-order provers to implement these ideas and thus directly extend first-order calculi were the LEO and LEO-II provers based on an extension of resolution [25, 27]. A notable feature about LEO-II was its cooperation with the first-order prover E. LEO-II was refactored into Leo-III, based on an incomplete paramodulation calculus [108]. Another important higher-order prover to be developed was the Satallax prover [43] based on a tableaux calculus. Satallax has gone on to be a multiple winner of the higher-order category at the CASC system competition [113].

However, the most successful first-order proof calculus, superposition, had until recently resisted extension to higher-order logic. This has now been achieved by Bentkamp et al. as part of the Matryoshka project [21]. The implementation of their calculus in the Zipperposition prover scored a dominant victory in the higher-order division of the 2020 CASC system competition. My work is closely related to theirs, so further details will appear in the course of this thesis. In recent years, there has been somewhat of a resurgence in interest in higher-order logic driven, potentially, by its close links to the languages of ITPs. Work has been carried out on extending SMT solvers to higer-order logic [13]. There has also been research on extending high-performance first-order provers to higher-order logic using both complete and incomplete methods [103].

It would be incorrect to end this section without some discussion of the automation available via ITPs. ITPs support automation in two main ways. Firstly, as discussed above, through the use of hammers. Secondly, most ITPs support proof tactics. For example, the `simp` tactic in the Isabelle theorem prover attempts to prove a goal by rewriting with proven equalities. By chaining such tactics together, powerful automation can be achieved. The Isabelle ITP running in automatic mode has won the higher-order category in the CASC system competition on multiple occasions. Recently, this line of automation has been improved through the use of machine learning [62]. The use of machine learning to schedule ITP tactics is firmly in the human-oriented school of automation mentioned above.

1.2 | Higher-Order Logic

Historically, most research in higher-order logic has revolved around Church's elementary type theory. Church introduced the λ-calculus as a model of computation, but also explored adding logical connectives to turn it into a logic. It was quickly realised that adding logical connectives to untyped lambda-calculus leads to paradoxes [26]. This led to the introduction of simple types. Church originally formulated his logic as a series of axioms and inference rules. His formulation came to be known as elementary type theory (ETT). Later researchers added further axioms for equality, extensionality, choice etc. leading to a variety of formulations of ETT. HOL came to be almost synonymous with ETT in its various formulations. However, HOL is also often defined more broadly as any formalism that allows quantification over predicates and variables [105]. For example, the induction principle for natural numbers is a statement of higher-order logic since it quantifies over predicates:

$$\forall p : \mathbb{N} \to o.((p(0) \wedge \forall n \in \mathbb{N}.(p(n) \implies p(n+1))$$

The question naturally arises, does HOL, in its broad sense, provide greater descriptive power than first-order logic? The answer to this question lies in semantics, and in particular in the interpretation provided to the higher-order quantifiers. In *standard semantics* once the range of the first-order quantifiers is fixed, the range of any higher-order quantifiers likewise is fixed. For example, let L_1 be an untyped first-order language (no quantification of functions or predicates). Then, $\mathcal{I} = (\mathcal{U}, \mathcal{J})$ is an interpretation of L_1 if $\mathcal{U}$ is a set called the *domain* and $\mathcal{J}$ is a function that maps the non-logical symbols of L_1 onto $\mathcal{U}$. A variable assignment ξ is a function from the collection of variables to $\mathcal{U}$. The interpretation $\mathcal{I}$ can easily be considered the interpretation of a second order language L_2 by extending ξ to be a function from the collection of predicate variables to $\mathcal{P}(\mathcal{U})$ and from the collection of function variables to $\mathcal{U}^{\mathcal{U}}$. As can be seen, once the range of the first-order variables has been fixed as $\mathcal{U}$, the range of the higher-order variables is fixed

as well.

Gödel's celebrated first incompleteness result [63] draws a distinction between second and higher-order logics with standard semantics on the one hand, and first-order logic on the other, since it shows that truth is not recursively axiomatizable in the first whilst it is in the second.

There is another semantics that can be provided for the higher-order quantifiers, namely, Henkin semantics. In Henkin semantics, the second-order functional variables are not mapped to members of $\mathcal{U}^{\mathcal{U}}$, but to members of some subset of $\mathcal{U}^{\mathcal{U}}$. Likewise, the predicate variables are not mapped to members of $\mathcal{P}(\mathcal{U})$, but to members of some subset of $\mathcal{P}(\mathcal{U})$. Gödel's theorem does not apply to second and higher-order logic when Henkin semantics are used.

The status of higher-order logic as a logic is a question that has been much debated. Some logicians and philosophers of logic have been opposed to considering higher-order logic with standard semantics as a logic. Chief amongst the critics has been W. O. Quine. Quine opined that the ontological commitments necessary for accepting standard semantics, such as the commitment to the existence of power sets, made higher-order logic nothing but "set theory in disguise" [90]. Others such as Stewart Shapiro have been far more accepting of higher-order logic [105, 106].

What about higher-order logic with Henkin semantics? It is probably fair to say that most philosophers of mathematics would be reluctant to consider HOL with Henkin semantics to be separate from many-sorted first-order logic. Indeed, Herbert Enderton said about HOL with Henkin semantics:

> The main feature of the general semantics is a result of the "nothing but" type: Second-order logic with the general semantics is nothing but first-order logic (many-sorted) together with the comprehension axioms [56].

On the other hand, proof-theoreticians are likely to consider higher-order logic a separate logic to first-order since the proof calculi for higher-order logic tend to look very different to their first-order counterparts. In the conclusion to this thesis (Chapter 8), I make a few further remarks on this topic and the relationship of my work to the debate.

1.3 | Vampire

The Vampire theorem prover is an automated theorem prover for first-order logic with equality. Vampire implements a resolution-superposition calculus which is an instance of the given clause architecture (Figure 1.1). A more detailed discussion of superposition can be found in Chapter 2. In this section I provide a high-level description of the given clause architecture and then discuss some of the main features of Vampire. For more

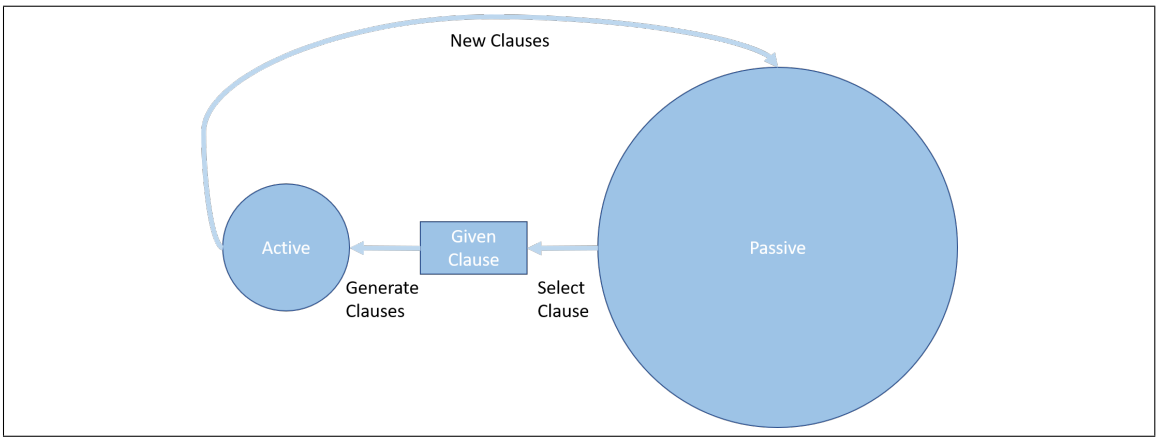

Figure 1.1: Given clause algorithm

details on the given clause algorithm and concrete implementations of it, please view articles by Schulz [101], and Kovács and Voronkov [83].

The given clause architecture is an example of saturation-based theorem proving. A prover based on the architecture takes the negated conjecture along with any axioms and attempts to derive a contradiction by systematically drawing all conclusions from this set. Before starting proof search most theorem provers convert the input to clausal normal form or CNF. CNF is a conjunction of disjunctions with each disjunction, known as clause, containing only universally quantified variables. It can be shown that every set of first-order formulas can be converted into an equi-satisfiable set of clauses. A large body of research exists on making this process efficient [6, 93, 126].

The given clause algorithm maintains two sets of clauses, the *passive* and *active* clauses. Initially, all clauses are placed in the passive set. Based on heuristics, a clause is selected from the passive set. The selected clause is known as the *given clause*. All inferences between the given clause and clauses in active are carried out, and the clause is then inserted into active. Any new clauses generated by the inferences are added to the passive set and the process restarts.

Because first-order logic is semi-decidable, if the original conjecture is a theorem, eventually the prover will derive the empty clause. In the case where the original conjecture is not a theorem, the prover may *saturate* or run forever. Saturate means that the passive set is empty and the empty clause has not been derived. In practice even when the conjecture is a theorem, the prover often times out and is unable to find a proof due to the large search space.

1.3.1 | Features of Vampire

Vampire is written in C++ and consists of over 200,000 lines of code. Vampire has been highly successful, winning numerous prizes at the CASC system competition. Some of its main features are:

- Implementation of powerful *simplification* techniques. The given clause algorithm as provided above is inefficient. The set of passive clauses tend to increase in size rapidly. What makes it practically useful are simplification inferences that can be used to remove clauses from the active and passive sets. This removal of clauses does not impact completeness so long as the removed clauses are *redundant* [88]. Vampire implements a large number of simplifications such as SUBSUMPTION, DEMODULATION, INNERREWRITING and TAUTOLOGYDELETION.
- Efficient data-structures and algorithms. Even in the presence of simplification techniques, the clause sets can grow dramatically in size during proof search at times reaching millions of clauses in size. Therefore, efficient data structures that allow the quick retrieval of clauses with certain properties are vital. Vampire uses *substitution trees* to index clauses [64]. Unification is the process of making two terms syntactically identical by substituting terms for variables. Its use is widespread throughout automated reasoning and is heavily used in the superposition calculus. Unification of first-order terms can be done in linear time, but in practice, Robinson's exponential time algorithm tends to be more efficient than linear time alternatives [70]. Vampire implements a modification of Robinson's algorithm that has a polynomial time worst case complexity [70].
- Implementation of the AVATAR architecture for clause splitting [118]. The concept of clause splitting arises because large clauses are often bad for proof search. Consider S to be a set of clauses and $C = C_1 \vee C_2$ to be a clause such that C_1 and C_2 are variable disjoint. Then $S \cup C$ is unsatisfiable iff $S \cup C_1$ and $S \cup C_2$ are unsatisfiable. Many schemes have been expounded for handling splitting and keeping track of the clauses relevant to the current branch [96, 127]. The AVATAR architecture differs from other schemes by using a SAT solver to keep track of the clauses active on a particular branch.
- Support for Booleans. Vampire has been extended to support Booleans as a first-class sort in three ways. Firstly, via axiomatisation, adding the clauses $\bot \not\approx \top$ and $x \approx \bot \vee x \approx \top$. The performance of the axiomatisation is extremely poor, so support for Booleans is now provided through a second and third means, namely, a special inference rule FOOLPARAMODULATION, or preprocessing techniques [81, 82]. Vampire's support for Booleans is very relevant to the current work and is discussed further in Chapters 5 and 6.

Most of the above features of Vampire had to be modified or adapted in some way to support higher-order logic. The modifications undertaken are detailed in Chapter 6.

1.4 | Thesis Contributions

This thesis provides three main contributions to the field of automated reasoning:

- The Knuth-Bendix order is commonly used in first-order ATPs to restrict inferences and control the size of the search space. I present a modification of the order that is *weak-compatible*. That is, for all terms s and t such that s weak reduces to t, s is greater by the order than t. The order is able to compare terms that are incomparable by other methods. It should be possible to extend the order to a β-compatible ordering and thereby open new avenues of research.
- Building on the novel order described above, I present two related modifications of the superposition calculus that are complete for a combinatory representation of HOL. My work is the second extension of superposition to HOL. Saturation based techniques and superposition in particular have been influential in the field of automated reasoning. Besides being the leading proof calculus for full first-order logic, superposition has been adapted as a decision procedure for various fragments of FOL [11, 59] and to reason about a variety of theories [5, 36, 57, 121]. Thus, its extension to HOL is notable. As mentioned previously, the extension to HOL devised by Bentkamp et al. has already lead to impressive results.
- Finally, I have updated Vampire to support both polymorphism and HOL. Vampire is now one of the leading higher-order provers and the leading prover in polymorphic first-order reasoning. In this thesis, I provide a detailed description of the implementation in the hope of encouraging and assisting the developers of other first-order provers to extend their solvers to more expressive logics.

1.5 | Thesis Structure

In Chapter 2, the mathematical tools used in this thesis are introduced and the syntax and semantics of higher-order and first-order logic are discussed. A translation from HOL to FOL is presented and a proof of its soundness and completeness is provided. In Chapter 3, a modified version of the Knuth-Bendix ordering is presented. The ordering has the interesting property of orienting all ground instances of combinator equations left to right. In Chapter 4, I show how this ordering can be used to parameterise two tweaked versions of the superposition calculus to attain calculi that are complete for the translation of higher-order problems presented in Chapter 2. The calculi are also *graceful* in that they extends first-order superposition whilst behaving exactly like first-order superposition on first-order problems. In Chapter 5, various extensions to the calculi, to deal with reasoning about Booleans, choice and defined equality, are described. The implementation of one of the base calculi along with all its extensions is detailed in Chapter 6. I analyse the performance of the Vampire implementation in detail and compare it with the performance of other leading higher-order solvers in Chapter 7. Finally, I end the thesis with some remarks on the practical and philosophical implications of my work and the research avenues that it opens for exploration in Chapter 8.

1.6 | Published Papers

Much of the work presented in this thesis is based on work published during the course of my doctoral studies.

- Chapter 3 is based in part on the following paper: Ahmed Bhayat and Giles Reger. "A Knuth-Bendix-like ordering for orienting combinator equations". In: *IJCAR*. vol. 12166. LNCS. Springer, 2020, pp. 259–277
- Chapter 4 contains research originally published in: Ahmed Bhayat and Giles Reger. "A combinator-based superposition calculus for higher-order logic". In: *IJCAR*. vol. 12166. LNCS. Springer, 2020, pp. 278–296
- Chapter 6 relies in part on work published in the paper: Ahmed Bhayat and Giles Reger. "A polymorphic Vampire". In: *IJCAR*. vol. 12167. LNCS. Springer, 2020, pp. 361–368

During the course of my doctoral studies, I also co-authored other papers that are not as directly relevant to this thesis.

- Ahmed Bhayat and Giles Reger. "Set of support for higher-order reasoning." In: *PAAR*. 2018

  This paper explores using the set of support strategy to reason about combinatory logic. We experimented with a variety of strategies such as placing the combinator axioms in set of support and placing all other axioms in set of support.
- Ahmed Bhayat and Giles Reger. "Restricted combinatory unification". In: *CADE*. vol. 11716. LNCS. Springer, 2019, pp. 74–93

  Unification of applicative first-order terms modulo the combinator axioms is the combinatory equivalent of higher-order unification for λ-terms. No efficient combinatory unification algorithms are known. In this paper, we describe and explore an efficient, but incomplete combinatory unification algorithm.
- Michael Rawson, Ahmed Bhayat, and Giles Reger. "Reinforced external guidance for theorem provers". In: *PAAR*. 2020

  We investigated the application of reinforcement learning techniques to theorem proving. My contribution to the research centered on the use of reinforcement learning to solve problems about Church numerals. Such problems are normally easy to state but difficult to solve because they require the synthesis of a complex unifier.

Chapter 2

Technical Preliminaries

> There is a difference between a thing and talking about a thing.
>
> *Kurt Gödel*

A logic has been defined to be a syntax along with a semantics and a deduction system [106]. In this chapter I discuss three logics.

- I present the syntax and semantics of monomorphic **first-order logic**. Monomorphic first-order logic is utilised in Chapter 4 of this thesis. Furthermore, many of the concepts I discuss were originally introduced in the context of (untyped) FOL. First-order superposition, a sound and complete proof calculus for first-order logic, is also presented.
- The syntax and semantics of polymorphic **applicative first-order logic** is provided. A recent extension of superposition to applicative first-order logic is discussed
- The syntax and semantic of polymorphic **higher-order** logic are detailed.

I describe a translation between higher-order and applicative first-order logic, and prove the translation to be sound and complete. Along the way, mathematical concepts that are necessary for understanding the remainder of this thesis are introduced. A certain level of familiarity with the field of automated reasoning is assumed. Readers can refer to Baader and Nipkow's *Term Rewriting and All That* [7] for terminology and concepts used in the field and to [10, 83, 101, 128] for more focused information.

2.1 | The Syntax of First-Order Logic

2.1.1 | Types

A type signature Σ_{ty} is a set of symbols known as *sorts* or *atomic types* $\mathcal{T}ya_{\Sigma_{\text{ty}}}$. The set of *types* $\mathcal{T}y_{\Sigma_{\text{ty}}}$ is defined as follows:

- Every atomic type is a member of $\mathcal{T}y_{\Sigma_{\text{ty}}}$;

- If $\tau_1 \ldots \tau_n, \tau$ are atomic types, then $(\tau_1 \times \ldots \times \tau_n) \rightarrow \tau$ is a type known as a *function type*.

Remark 1. Throughout the remainder of this thesis, an overbar is used to denote tuples and lists. For example $\overline{\tau}_n$ denotes a tuple or list of n types. Where unimportant, the subscript n is dropped to denote a tuple or list of unknown length. The reader's discernment is relied on to distinguish between a tuple and a list.

Remark 2. It is assumed that the set Σ_{ty} contains two sorts i and o, where i is the sort of individuals and o is the sort of propositions. The sort o can only occur on the right hand side of $\rightarrow$ in a function type.

Remark 3. Throughout the remainder of this thesis, the convention adhered to is to use $\tau, \tau_1, \tau_2 \ldots$ to denote types.

2.1.2 | Terms

For a given type signature Σ_{ty}, a term signature Σ is a set of symbols and associated members of $\mathcal{T}y_{\Sigma_{\mathsf{ty}}}$. The notation $\mathsf{f} : \tau$ denotes that the symbol f is associated with the type τ. It is assumed that Σ contains symbols $\bot : o$ and $\top : o$ representing falsity and truth. A type signature and related term signature $(\Sigma_{\mathsf{ty}}, \Sigma)$ form a *signature*. Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be some arbitrary but fixed signature. Let $\mathcal{V}$ be a countably infinite set of typed variables whose types are members of $\mathcal{T}ya_{\Sigma_{\mathsf{ty}}}$. The set of terms $\mathcal{T}_\Sigma(\mathcal{V})$ over $(\Sigma_{\mathsf{ty}}, \Sigma)$ and $\mathcal{V}$ is defined inductively:

- If $x : \tau \in \mathcal{V}$, then x is a term of type τ;
- Let $\mathsf{f} : (\tau_1 \times \ldots \times \tau_m) \rightarrow \tau$ be a member of Σ. Let $t_1, \ldots, t_m$ be terms of type $\tau_1, \ldots, \tau_m$ respectively. Then, $\mathsf{f}(t_1, \ldots, t_m)$ is a term of type τ.

The set of all variables occurring in a term t is denoted $vars(t)$. For example, if $t = \mathsf{f}(x, x)$, then $vars(t)$ is the set $\{x\}$. The multiset of variables occurring in a term t is denoted $vars_{\#}(t)$. The subset relation $\subseteq$ is also used for multiset inclusion with the context making clear which is intended.

Definition 1 (Subterms of a Term)**.** Let t be a term. The subterms of t are defined inductively as follows:

- t is a subterm of itself;
- If $t = \mathsf{f}(t_1, \ldots, t_n)$, then the subterms of each t_i are subterms of t.

Definition 2 (Positions)**.** A *position* is either the empty position ϵ, or $n.p$ where n is a natural number and p a position. A position p is *strictly above* a position p' (denoted $p < p'$) if $\exists p''.p'' \neq \epsilon \ \wedge \ p' = p.p''$. Positions p and p' are *incomparable* (denoted $p \parallel p'$) if $p \not< p'$ and $p' \not< p$, and $p \neq p'$.

Definition 3 (Subterm at Position)**.** A term s is a subterm of a term t at position p (written $s = t|_p$) if either one of the following holds:

- $p = \epsilon$ and $s = t$;
- $t = \mathsf{f}\langle\overline{\tau}_n\rangle(t_1, \ldots, t_n)$, $p = i.p'$ and s is a subterm of t_i at position p'.

Definition 4 (Valid Positions)**.** If a term t has a subterm s at a position p, then p is said to be a valid position in t. The set of all valid positions in t is denoted $pos(t)$.

Definition 5 (Context)**.** A term with a hole at some position p is known as a *context* and denoted $u[\,]_p$. If the position of the hole is not important, it is dropped from the notation. A context $u[\,]$ with its hole filled by a term t is denoted $u[t]$.

Definition 6 (Ground Term)**.** A ground term is a term that contains no variables.

Definition 7 (Substitution)**.** A substitution is a mapping of variables of some type τ to terms of the same type. For example, $\sigma = \{x \to \mathsf{f}(\mathsf{a})\}$ is a substitution that maps the variable x to the term $\mathsf{f}(\mathsf{a})$. The application of a substitution σ to a term t is written $t\sigma$. A substitution is ground if it maps variables to ground terms. I assume that every ground substitution is grounding. If t is a term and σ a ground substitution, it is assumed that $t\sigma$ is ground. In other words, it is assumed that $vars(t) \subseteq domain(\sigma)$.

Definition 8 (At a Variable, Below a Variable)**.** Let t be a term and θ a grounding substitution. Let $t\theta = s[u]_p$. Then u occurs at a variable in t if $t|_p$ is a variable. Similarly, u occurs below a variable in t if there exists a proper prefix p' of p such that $t|_{p'}$ is a variable.

2.1.3 | Formulas

First-order logic enforces a separation between terms and formulas. The set of *atomic* formulas is defined as:

- If t is a term of type o, then t is an atomic formula.
- If t and t' are terms of type τ, then $t \approx t'$ is an atomic formula known as a *literal*.

The set of formulas over a signature $(\Sigma_{\mathsf{ty}}, \Sigma)$ and set of typed variables $\mathcal{V}$ can be defined inductively as:

- $\bot$ is a formula;
- $\top$ is a formula;
- All atomic formulas are formulas;
- If f is a formula then $\neg f$ is a formula. If f is of the form $t \approx s$, then $\neg f$ is denoted $t \not\approx s$;
- If f and g are formulas, then $f \vee g$ is a formula;
- If f is a formula and $x : \tau \in \mathcal{V}$ for some τ, then $\forall x.f$ is a formula.

The other common logical connectives can be defined in terms of these, and therefore do not need to be considered logical symbols.

Definition 9 (Free Variable, Closed Formula)**.** A free variable in a formula is a variable that is not bound by a quantifier. A formula that contains no free variables is *closed*.

Definition 10 (Clause)**.** A *clause* is a disjunction of literals or negative literals. For example, $t_1 \approx \top \vee t_2 \approx t_3$ is a clause. All variables in a clause are implicitly assumed to be universally quantified. The clause containing no literals is known as the *empty clause* and denoted $\bot$.

It is possible to convert any formula into an equi-satisfiable (but not necessarily model-preserving) set of clauses. A set of clauses is said to be in *conjunctive normal form* (CNF). There is a large body of research on converting formulas to CNF efficiently [6, 82, 126].

2.2 | The Semantics of First-Order Logic

The purpose of a semantics is to provide meaning to the terms of a language. First-order logic has a well-studied set-theoretic semantics.

A type interpretation $\mathcal{I}_{\mathsf{ty}}$ is a pair $(\mathcal{U}, \mathcal{J}_{\mathsf{ty}})$ where $\mathcal{U}$ is a family of non-empty sets called *universes* and $\mathcal{J}_{\mathsf{ty}}$ is a *type interpretation* function that maps a sort κ to a member of $\mathcal{U}$. It is assumed that $\mathcal{U}$ contains a set $\mathcal{U}_o = \{0, 1\}$ and that $\mathcal{J}_{\mathsf{ty}}(o) = \mathcal{U}_o$. The denotation of an atomic type τ in $\mathcal{I}_{\mathsf{ty}}$ is represented as $[\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}}}$ and defined as $[\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}}} = \mathcal{J}_{\mathsf{ty}}(\tau)$.

A *valuation* is a function that maps variables of type τ to members of $[\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}}}$. An interpretation function $\mathcal{J}$ maps a function symbol $\mathsf{f} : (\tau_1 \times \ldots \times \tau_m) \to \tau$ to a function of type $([\![\tau_1]\!]_{\mathcal{I}_{\mathsf{ty}}} \times \ldots \times [\![\tau_m]\!]_{\mathcal{I}_{\mathsf{ty}}}) \to [\![\tau]\!]_{\mathcal{I}_{\mathsf{ty}}}$. An interpretation is a tuple consisting of a type interpretation and an interpretation function. For an interpretation $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J})$ and a valuation ξ, the denotation of a term is as follows:

- $[\![x]\!]^{\xi}_{\mathcal{I}} = \xi(x)$;
- $[\![\mathsf{f}(\overline{s}_m)]\!]^{\xi}_{\mathcal{I}} = \mathcal{J}(\mathsf{f})([\![s_1]\!]^{\xi}_{\mathcal{I}}, \ldots, [\![s_m]\!]^{\xi}_{\mathcal{I}})$.

The denotation of a formula for an interpretation $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J})$ and a valuation ξ is:

- $[\![\bot]\!]^{\xi}_{\mathcal{I}} = 0$;
- $[\![\top]\!]^{\xi}_{\mathcal{I}} = 1$;
- $[\![t \approx s]\!]^{\xi}_{\mathcal{I}} = [\![t]\!]^{\xi}_{\mathcal{I}} = [\![s]\!]^{\xi}_{\mathcal{I}}$;
- $[\![\mathsf{p}(\overline{s}_m)]\!]^{\xi}_{\mathcal{I}} = \mathcal{J}(\mathsf{p})([\![s_1]\!]^{\xi}_{\mathcal{I}}, \ldots, [\![s_m]\!]^{\xi}_{\mathcal{I}})$;
- $[\![\neg f]\!]^{\xi}_{\mathcal{I}} = 1 - [\![f]\!]^{\xi}_{\mathcal{I}}$
- $[\![f \vee g]\!]^{\xi}_{\mathcal{I}} = max([\![f]\!]^{\xi}_{\mathcal{I}}, [\![g]\!]^{\xi}_{\mathcal{I}})$;
- $[\![\forall x : \tau.\, f]\!]^{\xi}_{\mathcal{I}} = min([\![f]\!]^{\xi[x \to \mathsf{a}]}_{\mathcal{I}})$ for all $\mathsf{a} \in [\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$.

A formula f holds in an interpretation $\mathcal{I}$ for a valuation function ξ if $[\![f]\!]^{\xi}_{\mathcal{I}} = 1$. Moreover, if f is closed, then the valuation function is unimportant. For a closed f that holds in an interpretation $\mathcal{I}$, I write $\mathcal{I} \models f$. A clause C holds in an interpretation $\mathcal{I}$, for valuation function ξ, if $[\![l]\!]^{\xi}_{\mathcal{I}} = 1$ for some literal l in C. An interpretation $\mathcal{I}$ *models* a clause C, written $\mathcal{I} \models C$, if C holds in $\mathcal{I}$ for every valuation function. An interpretation $\mathcal{I}$ models a set of clauses N if it models every clause in N. A set of clauses N entails a set of clauses N' ($N \models N'$) if every model $\mathcal{I}$ of N is also a model of N'. A set of clauses is *unsatisfiable* if it has no model.

Remark 4. Strictly speaking, entailment is between *sets* of clauses. If the set N' being entailed contains a single clause C, I abuse notation and write $N \models C$ rather than $N \models \{C\}$.

Remark 5. Where useful for clarity of presentation, I abuse notation and apply an operation that should be applied to a single element to a tuple of elements. The intended meaning is the mapping of the operation to the tuple. For example, the notation $[\![\overline{\tau}_n]\!]_{\mathcal{I}_{\mathsf{ty}}}$ represents the tuple $([\![\tau_1]\!]_{\mathcal{I}_{\mathsf{ty}}}, \ldots, [\![\tau_n]\!]_{\mathcal{I}_{\mathsf{ty}}})$.

2.3 | The Syntax of Applicative First-Order Logic (Clausal)

2.3.1 | Types

A *type constructor* is a symbol related to a natural number called its *arity*. Let Σ_{ty} be a set (called a type signature) of type constructors containing the binary type constructor $\rightarrow$ and at least one nullary type constructor. Let $\mathcal{V}_{\mathsf{ty}}$ be an infinite set of variables. The set of *atomic types* $\mathcal{T}\!\mathit{ya}_{\Sigma_{\mathsf{ty}}}(\mathcal{V}_{\mathsf{ty}})$ and types $\mathcal{T}\!\mathit{y}_{\Sigma_{\mathsf{ty}}}(\mathcal{V}_{\mathsf{ty}})$ over a *type signature* Σ_{ty} and set of variables $\mathcal{V}_{\mathsf{ty}}$ are defined as follows:

- If $\alpha \in \mathcal{V}_{\mathsf{ty}}$ then $\alpha \in \mathcal{T}\!\mathit{ya}_{\Sigma_{\mathsf{ty}}}(\mathcal{V}_{\mathsf{ty}})$;
- If κ is a member of Σ_{ty} that has arity n and $\tau_1 \ldots \tau_n$ are types, then $\kappa(\tau_1, \ldots, \tau_n) \in \mathcal{T}\!\mathit{ya}_{\Sigma_{\mathsf{ty}}}(\mathcal{V}_{\mathsf{ty}})$. A type constructor κ with arity 0 is written κ rather than $\kappa()$. The type constructor $\rightarrow$ is written infix;
- All atomic types are types;
- If $\alpha \in \mathcal{V}_{\mathsf{ty}}$ and $\tau \in \mathcal{T}\!\mathit{y}_{\Sigma_{\mathsf{ty}}}(\mathcal{V}_{\mathsf{ty}})$, then $\Pi\alpha.\tau \in \mathcal{T}\!\mathit{y}_{\Sigma_{\mathsf{ty}}}(\mathcal{V}_{\mathsf{ty}})$.

Remark 6. Rather than write $\Pi\alpha_1.\Pi\alpha_2. \ldots \Pi\alpha_n$, I write $\Pi\overline{\alpha}_n$.

Remark 7. Throughout the remainder of this thesis, the convention adhered to is to use α, β, γ to denote type variables.

Remark 8. Where the set of type variables involved is not of importance, it is dropped from terminology. Therefore, $\mathcal{T}\!\mathit{y}_{\Sigma_{\mathsf{ty}}}$ refers to a set of types over the set of constructors Σ_{ty} and some arbitrary set of type variables.

Definition 11 (Closed type)**.** If every variable in a type is bound by a type quantifier, the type is closed.

2.3.2 | Terms

For a given type signature Σ_{ty}, a term signature Σ is a set of symbols associated to closed members of $\mathcal{T}y_{\Sigma_{\mathsf{ty}}}$. A type signature and related term signature form a *signature*. Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be a signature and $\mathcal{V}$ be a countably infinite set of variables whose types come from $\mathcal{T}ya_{\Sigma_{\mathsf{ty}}}$. The set of terms $\mathcal{T}_\Sigma(\mathcal{V})$ over $(\Sigma_{\mathsf{ty}}, \Sigma)$ and $\mathcal{V}$ is defined inductively as:

- If $x : \tau \in \mathcal{V}$ then x is a term of type τ;
- Let $\mathsf{f} : \Pi\overline{\alpha}_n.\tau$ be a member of Σ; Let $\overline{\tau}_n$ be a tuple of types. Let σ be the substitution $\{\alpha_1 \to \tau_1, \ldots, \alpha_n \to \tau_n\}$. Then, $\mathsf{f}\langle\overline{\tau}_n\rangle$ is a term of type $\tau\sigma$;
- If t_1 is a term of type $\tau_1 \to \tau_2$ and t_2 is a term of type τ_1 then $t_1\, t_2$ is a term of type τ_2.

A term of the form $s\, t$ is known as an *application*. All other terms are known as *heads*. Heads of the form $\mathsf{f}\langle\overline{\tau}\rangle$ are known as *first-order*. Application is assumed to be left associative. Every term can be uniquely decomposed into the form $h\, s_1\, \ldots, s_n$, where h is a head. The terms $\overline{s}_n$ are referred to as the arguments of h. By an abuse of notation a head and n arguments is written as $h\, \overline{s}_n$. By $head(t)$, I refer to the unique head of the term t. A literal is an equality between terms of the same type written $s \approx t$. A negative literal is denoted $s \not\approx t$. A clause is a finite multiset of literals.

Remark 9. The logic presented above is *clausal* because it does not support Boolean subterms and logical connectives. It is possible to add support for Booleans by using axiomatised proxy symbols.

Definition 12 (Set of Ground Instances)**.** For a signature $(\Sigma_{\mathsf{ty}}, \Sigma)$ and clause C over the signature, $\mathcal{G}_\Sigma(C)$ represents the set of all Σ-ground instances of C. For a clause set N, the set of all Σ-ground instances of clauses in N is denoted $\mathcal{G}_\Sigma(N)$

Due to the different term structure, the definitions of subterms, subterms at position and contexts have to be updated.

Definition 13 (Subterms of a Term)**.** Let t be a term. The subterms of t are defined inductively as follows:

- t is a subterm of itself;
- If $t = t_1\, t_2$, then the subterms of t_1 and t_2 are subterms of t.

Example 1. Consider the first-order term $t = \mathsf{f}(\mathsf{a}, \mathsf{b})$ and the applicative first-order term $t' = \mathsf{f}\,\mathsf{a}\,\mathsf{b}$. The term t has three subterms; $\mathsf{f}(\mathsf{a}, \mathsf{b})$, a and b. On the other hand, t' has five subterms; $\mathsf{f}\,\mathsf{a}\,\mathsf{b}$, $\mathsf{f}\,\mathsf{a}$, f, a and b.

Definition 14 (Size of a Term)**.** By $|t|$, the number of symbols occurring in t is denoted.

Definition 15 (Subterm at Position)**.** A term s is a subterm of a term t at position p (written $s = t|_p$) if either of the following holds:

- $p = \epsilon$ and $s = t$;
- $t = t_1\, t_2$, $p = i.p'$ for some $i \in \{1, 2\}$ and s is a subterm of t_i at position p'.

Definition 16 (Prefix Subterms)**.** Subterms that occur at positions of the form $p.1$ are prefix subterms. In Example 1, the prefix subterms of t' are $\mathsf{f}\ \mathsf{a}$ occurring at position $\epsilon.1$ and f occurring at position $\epsilon.1.1$.

Definition 17 (Green Subterms)**.** The green subterms of a term t are all its non-prefix subterms. In Example 1, the green subterms of t' are $\mathsf{f}\ \mathsf{a}\ \mathsf{b}$, a and b. If t is a green subterm of $s[t]$, this is denoted $s\langle t\rangle$. Analogously, $s\langle\ \rangle$ refers to a context with a hole at a green subterm position.

Definition 18 (Substitution)**.** The definition of substitution is extended to include type variables. A substitution can map type variables to arbitrary types. It is ground, if it maps all type *and* term variables in its domain to ground types and terms.

2.4 | The Semantics of Applicative First-Order Logic

I follow Bentkamp et al. [19] closely in specifying the semantics. A type interpretation $\mathcal{I}_{\mathsf{ty}}$ is a pair $(\mathcal{U}, \mathcal{J}_{\mathsf{ty}})$ where $\mathcal{U}$ is a family of non-empty sets called *universes*, and $\mathcal{J}_{\mathsf{ty}}$ is a *type interpretation* function that maps a type constructor κ of arity n to a member of $(\mathcal{U}_1 \times \ldots \times \mathcal{U}_n) \to \mathcal{U}$. A *type valuation* ξ is a function that maps type variables to members of $\mathcal{U}$. The denotation of an atomic type τ in $\mathcal{I}_{\mathsf{ty}}$ is represented as $[\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$ and defined as follows:

- If τ is a type variable α, then $[\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}} = \xi(\alpha)$;
- If $\tau = \kappa(\overline{\tau}_n)$, then $[\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}} = \mathcal{J}_{\mathsf{ty}}(\kappa)([\![\tau_1]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}, \ldots, [\![\tau_n]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}})$.

A type valuation ξ is extended to a *valuation* by setting $\xi(x : \tau)$ to be a member of $[\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$. An interpretation function $\mathcal{J}$ maps a function symbol $\mathsf{f} : \Pi\overline{\alpha}_n.\, \tau$ and a tuple of universes $\overline{\mathcal{U}}_n$ to a member of $[\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$ where ξ is the valuation that maps α_1 to $\mathcal{U}_1$ and α_2 to $\mathcal{U}_2$ and so on. An extension function $\mathcal{E}$ is a family of functions $\mathcal{E}_{\mathcal{U}_1,\mathcal{U}_2} : \mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{U}_1, \mathcal{U}_2) \to (\mathcal{U}_1 \to \mathcal{U}_2)$ such that there exists a function $\mathcal{E}_{\mathcal{U}_1,\mathcal{U}_2}$ for every pair of universes $\mathcal{U}_1, \mathcal{U}_2 \in \mathcal{U}$. An interpretation is a tuple consisting of a type interpretation, an interpretation function and an extension function. For a given interpretation $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{E})$, $\mathcal{I}$ is *extensional* if $\mathcal{E}_{\mathcal{U}_1,\mathcal{U}_2}$ is injective for all $\mathcal{U}_1, \mathcal{U}_2$ and is *standard* if $\mathcal{E}_{\mathcal{U}_1,\mathcal{U}_2}$ is bijective for all $\mathcal{U}_1, \mathcal{U}_2$.

For an interpretation $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{E})$ and a valuation ξ, the denotation of a term is as follows:

- $[\![x]\!]^{\xi}_{\mathcal{I}} = \xi(x)$;
- $[\![\mathsf{f}\langle\overline{\tau}_n\rangle]\!]^{\xi}_{\mathcal{I}} = \mathcal{J}(\mathsf{f}, [\![\tau_1]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}, \ldots, [\![\tau_n]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}})$;
- $[\![s\,t]\!]^{\xi}_{\mathcal{I}} = \mathcal{E}_{\mathcal{U}_1,\mathcal{U}_2}([\![s]\!]^{\xi}_{\mathcal{I}})([\![t]\!]^{\xi}_{\mathcal{I}})$ where s is of type $\tau \to v$, t is of type v, $\mathcal{U}_1 = [\![\tau]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$ and $\mathcal{U}_2 = [\![v]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}$.

An equation $s \approx t$ is true in an interpretation $\mathcal{I}$ with valuation function ξ, if $[\![s]\!]^{\xi}_{\mathcal{I}}$ and $[\![t]\!]^{\xi}_{\mathcal{I}}$ are the same object, and is false otherwise. A disequation $s \not\approx t$ is true if $s \approx t$ is false. A clause is true if one of its literals is true, and a clause set is true if every clause in the set is true. An interpretation $\mathcal{I}$ is a model of a clause C, written $\mathcal{I} \models C$, if C is true in $\mathcal{I}$ for all valuation functions. An interpretation is a model of a clause set if it is a model of each of its clauses. The definition of an unsatisfiable clause set and of entailment are as in the first-order case.

Since the above specifies a weak Henkin semantics without choice, naive Skolemization is unsound as pointed out by Bentkamp et al. [21]. For any calculus that wishes to be sound with respect to the semantics above, one solution would be to implement Skolemization with mandatory arguments as explained in [87]. However, the introduction of mandatory arguments considerably complicates both the calculus and any implementation. Therefore, in Chapter 4 where I develop a calculus that is sound with respect to the above semantics, I resort to the same 'trick' as Bentkamp et al., namely, claiming completeness for my calculus with respect to models as described above. This holds since I assume problems to be clausified. Since actual problems are not normally clausified, soundness is claimed for the implementation with respect to models that satisfy the axiom of choice, and completeness can be claimed if the axiom of choice is added to the clause set.

2.5 | The Syntax of Higher-Order Logic (Clausal)

The set of types is defined as per applicative first-order logic.

2.5.1 | Terms

Following Bentkamp et al. terms are defined as $\beta\eta$-equivalence classes [21]. For a given type signature Σ_{ty}, let Σ be a term signature. A type signature and term signature form a *signature*. Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be a signature. Let $\mathcal{V}$ be a countably infinite set of variables whose types come from $\mathcal{T}\!\mathit{ya}_{\Sigma_{\mathsf{ty}}}$. The set of *raw λ-terms* $\mathcal{T}_{\Sigma}(\mathcal{V})$ over $(\Sigma_{\mathsf{ty}}, \Sigma)$ and $\mathcal{V}$ is defined as:

- If $x : \tau \in \mathcal{V}$ then x is a raw λ-term of type τ;
- Let $\mathsf{f} : \Pi\overline{\alpha}_n.\tau$ be a member of Σ. Let $\overline{\tau}_n$ be a tuple of types. Let σ be the substitution $\{\alpha_1 \to \tau_1, \ldots, \alpha_n \to \tau_n\}$. Then, $\mathsf{f}\langle\overline{\tau}_n\rangle$ is a raw λ-term of type $\tau\sigma$;
- If $x \in \mathcal{V}_\tau$ and $t : v$, then $\lambda x.\, t$ is a raw λ-term of type $\tau \to v$ known as a λ-*expression*;

- If t_1 is a raw λ-term of type $\tau_1 \to \tau_2$ and t_2 is a raw λ-term of type τ_1 then $t_1\, t_2$ is a raw λ-term of type τ_2.

The concepts *free variable*, *bound variable*, *variable capture*, *capture avoiding substitution* and *alpha-renaming* are defined as per the norm in the literature [69, Chapter 1]. The set of raw λ-terms considered modulo alpha-renaming form λ-terms.

A λ-term $t_1 = (\lambda x.\, s)\, t$ β-reduces to a λ-term t_2 (written $t_1 \to_\beta t_2$) if $t_2 = s\{x \to t\}$, where the substitution is assumed to be capture avoiding. A λ-term $t_1 = \lambda x.\, s\, x$ η-reduces to a λ-term t_2 (written $t_1 \to_\eta t_2$) if $t_2 = s$ and x does not occur free in s.

The set of λ-terms considered modulo β- and η-reduction form the set of *terms*. A literal and clause are defined as per clausal applicative first-order logic.

2.6 | The Semantics of Higher-Order Logic

A type interpretation is defined in a like manner to the applicative case with one addition. For every pair of universes $\mathcal{U}_1, \mathcal{U}_2$, $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{U}_1, \mathcal{U}_2)$ must be a subset of $\mathcal{U}_1^{\mathcal{U}_2}$.

The definition of a valuation and interpretation function is again as per the applicative case. A λ-*designation function* is a function that takes a valuation and a λ-expression of type τ and returns a member of $[\![\tau]\!]^\xi_{\mathcal{I}_{\mathsf{ty}}}$. A type interpretation, interpretation function and λ-designation function form an interpretation. Let $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J}, \mathcal{L})$ be an interpretation. The denotation of a term in $\mathcal{I}$ for valuation function ξ is:

- $[\![x]\!]^\xi_{\mathcal{I}} = \xi(x)$;
- $[\![\mathsf{f}\langle\overline{\tau}_n\rangle]\!]^\xi_{\mathcal{I}} = \mathcal{J}(\mathsf{f}, [\![\tau_1]\!]^\xi_{\mathcal{I}_{\mathsf{ty}}}, \ldots, [\![\tau_n]\!]^\xi_{\mathcal{I}_{\mathsf{ty}}})$;
- $[\![\lambda x.\, t]\!]^\xi_{\mathcal{I}} = \mathcal{L}(\xi, \lambda x.\, t)$;
- $[\![s\, t]\!]^\xi_{\mathcal{I}} = [\![s]\!]^\xi_{\mathcal{I}}([\![t]\!]^\xi_{\mathcal{I}})$.

A interpretation $\mathcal{I}$ is *proper* if $\mathcal{L}(\xi, \lambda x.\, t)\,(\mathsf{a}) = [\![t]\!]^{\xi[x \to \mathsf{a}]}_{\mathcal{I}}$ for all valuations ξ and λ-expressions. The truth of an equation, disequation, clause and clause set is as per the applicative case with proper interpretations being considered instead of interpretations.

2.7 | Correspondence

I define a translation $(\!|\ |\!)$ from higher-order terms over the signature $(\Sigma_{\mathsf{ty}}, \Sigma)$ to applicative first-order terms over the signature $(\Sigma_{\mathsf{ty}}, \Sigma \cup \{\mathbf{S}, \mathbf{K}, \mathbf{C}, \mathbf{B}, \mathbf{I}\})$. Without loss of generality, assume that Σ does not contain any symbols from amongst $\mathbf{S}, \mathbf{K}, \mathbf{C}, \mathbf{B}, \mathbf{I}$ known as *combinators*. The type of each of the combinator symbols are as given below:

$$\mathbf{S} : \Pi\alpha\beta\gamma.\, (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$
$$\mathbf{C} : \Pi\alpha\beta\gamma.\, (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$$

$$\mathsf{B} : \Pi\alpha\beta\gamma.\,(\alpha \to \gamma) \to (\beta \to \alpha) \to \beta \to \gamma$$
$$\mathsf{K} : \Pi\alpha\gamma.\,\alpha \to \gamma \to \alpha$$
$$\mathsf{I} : \Pi\alpha.\,\alpha \to \alpha$$

The translation $(\!|\ |\!)$ from higher-order terms to applicative first-order terms:

$$\begin{array}{lcl}
(\!|\, x \,|\!) & = & x \\
(\!|\, \mathsf{f}\langle\overline{\tau}\rangle \,|\!) & = & \mathsf{f}\langle\overline{\tau}\rangle \\
(\!|\, \lambda x : \tau.\, x \,|\!) & = & \mathsf{I}\langle\tau\rangle \\
(\!|\, \lambda x : \tau_2.\, t \,|\!) & = & \mathsf{K}\langle\tau_1, \tau_2\rangle\,(\!|\, t \,|\!), \quad x \text{ doesn't occur in } t \text{ and } t : \tau_1 \\
(\!|\, \lambda x : \tau_1.\, t\, t' \,|\!) & = & \mathsf{B}\langle\tau_1, \tau_2, \tau_3\rangle\,(\!|\, t \,|\!)\,(\!|\, \lambda x.\, t' \,|\!) \\
 & & x \text{ only occurs free in } t', t : \tau_2 \to \tau_3 \text{ and } t' : \tau_2 \\
(\!|\, \lambda x : \tau_1.\, t\, t' \,|\!) & = & \mathsf{C}\langle\tau_1, \tau_2, \tau_3\rangle\,(\!|\, \lambda x.\, t \,|\!)\,(\!|\, t' \,|\!) \\
 & & x \text{ only occurs free in } t, t : \tau_2 \to \tau_3 \text{ and } t' : \tau_2 \\
(\!|\, \lambda x : \tau_1.\, t\, t' \,|\!) & = & \mathsf{S}\langle\tau_1, \tau_2, \tau_3\rangle\,(\!|\, \lambda x.\, t \,|\!)\,(\!|\, \lambda x.\, t' \,|\!) \\
 & & x \text{ occurs free in } t \text{ and } t', t : \tau_2 \to \tau_3 \text{ and } t' : \tau_2 \\
(\!|\, \lambda x.\, \lambda y.\, t \,|\!) & = & (\!|\, \lambda x.\, (\!|\, \lambda y.\, t \,|\!) \,|\!) \\
(\!|\, s\, t \,|\!) & = & (\!|\, s \,|\!)\,(\!|\, t \,|\!)
\end{array}$$

The translation is extended to literals, clauses and clause sets. The translation of a literal is carried out component-wise; $(\!|\, s \approx t \,|\!) = (\!|\, s \,|\!) \approx (\!|\, t \,|\!)$. A clause is translated by translating each of its literals. For a clause set N, $(\!|\, N \,|\!)$ is the set containing the translation of each clause in N. Next, I refine the concept of an applicative interpretation to ensure that combinators possess the desired semantics.

Definition 19 (Combinatory Interpretation)**.** Consider the set CA consisting of the following five equations that define the combinators.

$$\begin{array}{lcllcl}
\mathsf{S}\langle\alpha_1, \alpha_2, \alpha_3\rangle\, x\, y\, z & \approx & x\, z\, (y\, z) & \mathsf{I}\langle\alpha\rangle\, x & \approx & x \\
\mathsf{C}\langle\alpha_1, \alpha_2, \alpha_3\rangle\, x\, y\, z & \approx & x\, z\, y & \mathsf{K}\langle\alpha_1, \alpha_2\rangle\, x\, y & \approx & x \\
\mathsf{B}\langle\alpha_1, \alpha_2, \alpha_3\rangle\, x\, y\, z & \approx & x\, (y\, z) & & &
\end{array}$$

An applicative first-order interpretation $\mathcal{I}$ is a *combinatory interpretation* if $\mathcal{I} \models CA$. Along with the refined concept of an interpretation, there is a refined concept of entailment. A clause set N combinatory entails a clause set N' ($N \models_{\mathsf{SKI}} N'$), if every combinatory model of N is a model of N'.

The next lemma demonstrates that combinatory interpretations interpret the translations of λ-expressions correctly.

Lemma 1. *Let $\mathcal{I}_{\mathsf{fo}}$ be a combinatory interpretation. Then, for any λ-expression $\lambda x.\, t$ and valuation function ξ, $\mathcal{E}([\![(\!|\, \lambda x.\, t \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) = [\![(\!|\, t \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}$*

Proof. The proof is by induction on the size of t. If $t = x$, then:

$$\mathcal{E}([\![(\!|\, \lambda x : \tau.\, x \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) = \mathcal{E}([\![\mathsf{I}\langle\tau\rangle]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) \overset{*}{=} \mathsf{a} = [\![x]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} = [\![(\!|\, x \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}$$

The step labeled (*) is justified as follows. Since $\mathcal{I}_{\mathsf{fo}}$ models the combinator axioms $\mathsf{I}\langle\sigma\rangle\, x \approx x$ is true in $\mathcal{I}_{\mathsf{fo}}$ for all valuation functions. In particular, it is true for the valuation function ξ' defined such that $\xi'(x) = \mathsf{a}$ and $\xi'(\sigma) = [\![\tau]\!]^{\xi}_{\mathcal{I}^{\mathsf{ty}}_{\mathsf{fo}}}$. By the definition of the truth of a literal, this is the same as stating that $[\![\mathsf{I}\langle\sigma\rangle\, x]\!]^{\xi'}_{\mathcal{I}_{\mathsf{fo}}} = [\![x]\!]^{\xi'}_{\mathcal{I}_{\mathsf{fo}}}$. Since $[\![x]\!]^{\xi'}_{\mathcal{I}_{\mathsf{fo}}} = \mathsf{a}$, this equality shows that $\mathcal{E}([\![\mathsf{I}\langle\tau\rangle]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) = \mathcal{E}([\![\mathsf{I}\langle\sigma\rangle]\!]^{\xi'}_{\mathcal{I}_{\mathsf{fo}}})([\![x]\!]^{\xi'}_{\mathcal{I}_{\mathsf{fo}}}) = [\![\mathsf{I}\langle\sigma\rangle\, x]\!]^{\xi'}_{\mathcal{I}_{\mathsf{fo}}} = [\![x]\!]^{\xi'}_{\mathcal{I}_{\mathsf{fo}}} = \mathsf{a}$.

If $t = y : \tau'$ then:

$$\begin{aligned}
\mathcal{E}([\![(\!|\, \lambda x : \tau.\, y \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) &= \mathcal{E}([\![\mathsf{K}\langle\tau',\tau\rangle\, y]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) \\
&= \mathcal{E}(\mathcal{E}([\![\mathsf{K}\langle\tau',\tau\rangle]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})([\![y]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}))(\mathsf{a}) \\
&\overset{*}{=} [\![y]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}} \\
&= [\![y]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} \\
&= [\![(\!|\, y \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}
\end{aligned}$$

If $t = \mathsf{f}\langle\overline{\tau}\rangle : \tau'$ then the proof is almost identical to the previous case. Likewise, if $t = s\, s'$ and x doesn't occur free in t. In the remaining cases, let s be a term of type $\tau_1 \to \tau_2$ and s' a term of type τ_1. If $t = s\, s'$ and x is free only in s, then:

$$\begin{aligned}
\mathcal{E}([\![(\!|\, \lambda x : \tau.\, s\, s' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) &= \mathcal{E}([\![\mathsf{C}\langle\tau,\tau_1,\tau_2\rangle\, (\!|\, \lambda x.\, s \,|\!)\, (\!|\, s' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) \\
&\overset{*}{=} \mathcal{E}(\mathcal{E}([\![(\!|\, \lambda x.\, s \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}))([\![(\!|\, s' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}) \\
&\overset{IH}{=} \mathcal{E}([\![(\!|\, s \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}})([\![(\!|\, s' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}) \\
&= \mathcal{E}([\![(\!|\, s \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}})([\![(\!|\, s' \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}) \\
&= [\![(\!|\, s \,|\!)\, (\!|\, s' \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} \\
&= [\![(\!|\, s\, s' \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}
\end{aligned}$$

If $t = s\, s'$ and x is free only in s', then:

$$\begin{aligned}
\mathcal{E}([\![(\!|\, \lambda x : \tau.\, s\, s' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) &= \mathcal{E}([\![\mathsf{B}\langle\tau,\tau_1,\tau_2\rangle\, (\!|\, s \,|\!)\, (\!|\, \lambda x.\, s' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) \\
&\overset{*}{=} \mathcal{E}([\![(\!|\, s \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathcal{E}([\![(\!|\, \lambda x.\, s' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a})) \\
&\overset{IH}{=} \mathcal{E}([\![(\!|\, s \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})([\![(\!|\, s' \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}})
\end{aligned}$$

$$
\begin{aligned}
&= \mathcal{E}([\![(\!| s |\!)]\!]^{\xi[x\to \mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}})([\![(\!| s' |\!)]\!]^{\xi[x\to \mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}) \\
&= [\![(\!| s \; s' |\!)]\!]^{\xi[x\to \mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} \\
&= [\![(\!| s \; s' |\!)]\!]^{\xi[x\to \mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}
\end{aligned}
$$

The case where x is free in s and s' is very similar to the previous two cases, but requires a double use of the induction hypothesis. In all cases the step marked with (*) can be justified in a similar manner to the first case. In the final case where $t = \lambda y.\, t'$, we have :

$$
\begin{aligned}
\mathcal{E}([\![(\!| \lambda x.\, \lambda y.\, t' |\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) &= \mathcal{E}([\![(\!| \lambda x.\, (\!| \lambda y.\, t' |\!) |\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})(\mathsf{a}) \\
&\overset{\dagger}{=} [\![(\!| (\!| \lambda y.\, t' |\!) |\!)]\!]^{\xi[x\to \mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} \\
&\overset{\ddagger}{=} [\![(\!| \lambda y.\, t' |\!)]\!]^{\xi[x\to \mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} \\
&= [\![(\!| t |\!)]\!]^{\xi[x\to \mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}
\end{aligned}
$$

The step marked (†) is justified since $(\!| \lambda y.\, t' |\!)$ cannot be a λ-expression and therefore one of the previous cases must apply. The step marked (‡) follows from the idempotency of $(\!|\ |\!)$. □

Theorem 1. *For a higher-order clause set N, N is satisfiable ($\mathcal{I}_{\mathsf{ho}} \models N$ for some proper higher-order interpretation $\mathcal{I}_{\mathsf{ho}}$) if and only if $(\!| N |\!)$ is satisfiable by an extensional combinatory interpretation ($\mathcal{I}_{\mathsf{fo}} \models (\!| N |\!)$ for some extensional combinatory interpretation $\mathcal{I}_{\mathsf{fo}}$). The two directions of this theorem provide the soundness and completeness of the $(\!|\ |\!)$ translation.*

Proof. Let N be a higher-order clause set and $\mathcal{I}_{\mathsf{ho}} = (\mathcal{U}_{\mathsf{ho}}, \mathcal{J}^{\mathsf{ho}}_{\mathsf{ty}} \mathcal{J}_{\mathsf{ho}}, \mathcal{L})$ a proper higher-order interpretation such that $\mathcal{I}_{\mathsf{ho}} \models N$. I define an extensional combinatory interpretation $\mathcal{I}_{\mathsf{fo}} = (\mathcal{U}_{\mathsf{fo}}, \mathcal{J}^{\mathsf{fo}}_{\mathsf{ty}}, \mathcal{J}_{\mathsf{fo}}, \mathcal{E})$ such that $\mathcal{I}_{\mathsf{fo}} \models (\!| N |\!)$. Let $\mathcal{U}_{\mathsf{fo}} = \mathcal{U}_{\mathsf{ho}}$, $\mathcal{J}^{\mathsf{fo}}_{\mathsf{ty}} = \mathcal{J}^{\mathsf{ho}}_{\mathsf{ty}}$ and $\mathcal{J}_{\mathsf{fo}} = \mathcal{J}_{\mathsf{ho}}$. Define $\mathcal{E}(\mathsf{a})(\mathsf{b})$ to be $\mathsf{a}(\mathsf{b})$. Since for all $\mathcal{U}_1, \mathcal{U}_2 \in \mathcal{U}$, $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{U}_1, \mathcal{U}_2)$ is a subspace of $\mathcal{U}_1^{\mathcal{U}_2}$, this definition is well formed. To prove that $\mathcal{I}_{\mathsf{fo}} \models (\!| N |\!)$, I prove that for any higher-order term t and evaluation ξ, $[\![t]\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}} = [\![(\!| t |\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}$. If $t = x$, then:

$$
[\![x]\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}} = \xi(x) = [\![x]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}} = [\![(\!| x |\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}
$$

If $t = \mathsf{f}\langle \overline{\tau} \rangle$, then:

$$
[\![\mathsf{f}\langle \overline{\tau} \rangle]\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}} = \mathcal{J}_{\mathsf{ho}}(\mathsf{f}, [\![\overline{\tau}]\!]^{\xi}_{\mathcal{I}^{\mathsf{ho}}_{\mathsf{ty}}}) = \mathcal{J}_{\mathsf{fo}}(\mathsf{f}, [\![\overline{\tau}]\!]^{\xi}_{\mathcal{I}^{\mathsf{fo}}_{\mathsf{ty}}}) = [\![\mathsf{f}\langle \overline{\tau} \rangle]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}} = [\![(\!| \mathsf{f}\langle \overline{\tau} \rangle |\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}
$$

If $t = s\, t'$, then:

$$\begin{aligned}
[\![s\,t']\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}} &= [\![s]\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}}([\![t']\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}}) \\
&\overset{IH}{=} [\![(\!|\, s \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}([\![(\!|\, t' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}) \\
&= \mathcal{E}([\![(\!|\, s \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}})([\![(\!|\, t' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}) \\
&= [\![(\!|\, s \,|\!)\,(\!|\, t' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}} \\
&= [\![(\!|\, s\,t' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}
\end{aligned}$$

If $t = \lambda x.\, t'$. Since $\mathcal{I}_{\mathsf{ho}}$ is a proper interpretation $[\![\lambda x.\, t']\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}}(\mathsf{a}) = [\![t']\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{ho}}}$ for any a. By Lemma 1, we have that $[\![(\!|\, \lambda x.\, t' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}(\mathsf{a}) = [\![(\!|\, t' \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}$. By the induction hypothesis, we have that $[\![(\!|\, t' \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} = [\![t']\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{ho}}}$. By the extensionality of $\mathcal{I}_{\mathsf{fo}}$ it follows that $[\![\lambda x.\, t']\!]^{\xi}_{\mathcal{I}_{\mathsf{ho}}}$ and $[\![(\!|\, \lambda x.\, t' \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}$ must be the same object.

To prove other direction of the theorem, let N be a set of higher-order clauses and $\mathcal{I}_{\mathsf{fo}} = (\mathcal{U}_{\mathsf{fo}}, \mathcal{J}^{\mathsf{fo}}_{\mathsf{ty}}, \mathcal{J}_{\mathsf{fo}}, \mathcal{E})$ an extensional combinatory interpretation such that $\mathcal{I}_{\mathsf{fo}} \models (\!|\, N \,|\!)$. I define a proper higher-order interpretation, $\mathcal{I}_{\mathsf{ho}} = (\mathcal{U}_{\mathsf{ho}}, \mathcal{J}^{\mathsf{ho}}_{\mathsf{ty}}\mathcal{J}_{\mathsf{ho}}, \mathcal{L})$, such that $\mathcal{I}_{\mathsf{ho}} \models N$. Since $\mathcal{I}_{\mathsf{fo}}$ is extensional, $\mathcal{J}^{\mathsf{fo}}_{\mathsf{ty}}(\to)(\mathcal{U}_1, \mathcal{U}_2)$ is a subset of $\mathcal{U}_1^{\mathcal{U}_2}$ for all $\mathcal{U}_1, \mathcal{U}_2 \in \mathcal{U}_{\mathsf{fo}}$. Therefore, $\mathcal{U}_{\mathsf{ho}}$ is defined as $\mathcal{U}_{\mathsf{fo}}$, $\mathcal{J}^{\mathsf{ho}}_{\mathsf{ty}}$ is defined as $\mathcal{J}^{\mathsf{fo}}_{\mathsf{ty}}$ and $\mathcal{J}_{\mathsf{ho}}$ is defined to be $\mathcal{J}_{\mathsf{fo}}$. The λ-designation function $\mathcal{L}$ is defined such that $\mathcal{L}(\xi, \lambda x.\, t) = [\![(\!|\, \lambda x.\, t \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}$. The proof again proceeds by showing that $\mathcal{I}_{\mathsf{fo}}$ and $\mathcal{I}_{\mathsf{ho}}$ interpret terms in the same way. The details are straightforward and therefore omitted.

To show that $\mathcal{I}_{\mathsf{ho}}$ is proper, I need to show that $\mathcal{L}(\xi, \lambda x.\, t)(\mathsf{a}) = [\![t]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{ho}}}$. By the definition of $\mathcal{L}$, $\mathcal{L}(\xi, \lambda x.\, t)(\mathsf{a}) = [\![(\!|\, \lambda x.\, t \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}(\mathsf{a})$. By Lemma 1, $[\![(\!|\, \lambda x.\, t \,|\!)]\!]^{\xi}_{\mathcal{I}_{\mathsf{fo}}}(\mathsf{a}) = [\![(\!|\, t \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}}$. Since $\mathcal{I}_{\mathsf{ho}}$ and $\mathcal{I}_{\mathsf{fo}}$ interpret terms in the same way, $[\![(\!|\, t \,|\!)]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{fo}}} = [\![t]\!]^{\xi[x\to\mathsf{a}]}_{\mathcal{I}_{\mathsf{ho}}}$ □

Theorem 1 is crucial to this thesis. One of the main results of this thesis is a complete superposition calculus for applicative first-order logic in the presence of combinator axioms. Theorem 1 allows me to claim that the calculus is complete for clausal higher-order logic. It has long been part of theorem proving folklore that sound and complete translations exist between higher-order logic and first-order logic. I have been unable to find a proof of this in the literature for a higher-order logic that supports the comprehension axioms. Dowek provides a similar translation and claims that the proof of completeness is easy [53]. Kerber's commonly cited paper [77], proves soundness and completeness, but for a version of HOL that does not have comprehension axioms. Theorem 1 can therefore be taken as a small contribution of this thesis.

2.8 | Term Orders

The concept of *term orders* and *simplification orders* are introduced briefly here. For a more detailed handling of these topics please view [7]. An *order* is an antisymmetric,

transitive relation (on terms). A *strict* order is a antireflexive order. A strict order is a *rewrite order* if it is compatible with contexts and stable under substitutions.

Let $\succ$ be a rewrite order. Compatibility with contexts (also known as monotonicity) means that for all terms s and t such that $s \succ t$ and for all contexts $u[\,]$, $u[s] \succ u[t]$. Stability under substitution means that for all terms s, t such that $s \succ t$, and all substitutions σ, $s\sigma \succ t\sigma$. A rewrite order is a *reduction order* if it is well-founded. The key concept in the context of this thesis is that of a *simplification order*. A simplification order is a rewrite order $\succ$ such that for all terms t and proper subterms t' of t, $t \succ t'$.

An oft-used simplification order is the Knuth-Bendix order. There are many variants of the order including versions that work on applicative first-order terms. Here I present a first-order version based on *untyped* terms. Such an order can easily be used on the polymorphic terms described above by conflating $\mathsf{f}\langle\overline{\tau}_m\rangle(\overline{s}_n)$ with the untyped term $\mathsf{f}(\overline{\tau}_m, \overline{s}_n)$ where types are considered to be terms. Wand provides another method of extending the Knuth-Bendix order to slightly more complex polymorphic terms [125].

Any simplification order can easily be extended to an ordering on literals and clauses by considering literals and clauses as multisets [88]. This is commonly done as follows. Let $\succ$ be an ordering on terms, $ms(t \approx t')$ be the multiset $\{t, t'\}$ and $ms(s \not\approx s')$ the multiset $\{s, s, s', s'\}$. Then, for literals l_1 and l_2, $l_1 \succ l_2$ if $ms(l_1) \succcurlyeq ms(l_2)$ where $\succcurlyeq$ is the multiset extension of $\succ$. Further, a clause C can be considered as a multiset of its literals. For example, $C = t_1 \approx t_2 \vee s_1 \not\approx s_2$ is mapped to the multiset $\{\{t_1, t_2\}, \{s_1, s_1, s_2, s_2\}\}$.[1] Then clauses are compared using the two-fold multiset extension of $\succ$.

Definition 20 (Maximal, Strictly Maximal)**.** A ground literal l is *maximal* in a ground clause $C = l_1 \vee \ldots \vee l_n \vee l$ if $l \succeq l_i$ for $1 \leq i \leq n$. Likewise, l is *strictly maximal* in C if $l \succ l_i$ for $1 \leq i \leq n$. For non-ground literals, the definitions need to be modified to take into account that the literals may be incomparable. A non-ground literal l is maximal in a non-ground clause $C = l_1 \vee \ldots \vee l_n \vee l$ if $l \not\prec l_i$ for $1 \leq i \leq n$. The definition of strictly maximal is obvious.

2.8.1 | First-Order Knuth-Bendix Order (KBO)

Let Σ be a finite signature containing untyped function symbols with associated arities. Let $\succ$ be a total well-founded ordering or *precedence* on Σ. Let w be a function from Σ to $\mathbb{N}$ that denotes the weight of a function symbol and $\mathcal{W}$ a function from $\mathcal{T}_\Sigma(\mathcal{V})$ to $\mathbb{N}$ denoting the weight of a term. Let $\varepsilon \in \mathbb{N}_{>0}$. The weight of a term is defined recursively:

$$\mathcal{W}(\mathsf{c}) = w(\mathsf{c}) \qquad \mathcal{W}(x) = \varepsilon \qquad \mathcal{W}(\mathsf{f}(\overline{s}_n)) = w(f) + \sum_{i=1}^{n} \mathcal{W}(s_i)$$

[1] There are a couple of technical reasons for considering negative literals to be larger than positive literals. Most importantly, the completeness proof of superpositon relies on an order that is total on ground clauses. Therefore, some means of differentiating between literals such as $s \approx t$ and $s \not\approx t$ is necessary.

A weight function $\mathcal{W}$ is called *admissible* with respect to a precedence $\succ$ if and only if:

1. For all constants $\mathsf{c} \in \Sigma^{(0)}$, $\mathcal{W}(\mathsf{c}) \geq \varepsilon$;
2. If f is a unary function symbol such that $\mathcal{W}(\mathsf{f}) = 0$, then for all other symbols $g \in \Sigma$, $f \succ g$.

Standard Knuth-Bendix order $>_{\mathsf{kbo}}$ is defined inductively as follows. Let t and s be terms, $\succ$ a precedence and $\mathcal{W}$ and admissible weight function. Then $t >_{\mathsf{kbo}} s$ if $vars_{\#}(s) \subseteq vars_{\#}(t)$ and:

- $\mathcal{W}(t) > \mathcal{W}(s)$ or $\mathcal{W}(t) = \mathcal{W}(s)$ and any of the following:
  - $t = \mathsf{f}^n(x)$, $s = x$ and $\mathcal{W}(\mathsf{f}) = 0$;
  - $t = \mathsf{f}(\overline{s}_n)$, $s = \mathsf{g}(\overline{r}_m)$ and $\mathsf{f} \succ \mathsf{g}$;
  - $t = \mathsf{f}(\overline{s}_n)$, $s = \mathsf{f}(\overline{r}_n)$ and $\overline{s}_n \gg^{\mathsf{lex}}_{\mathsf{kbo}} \overline{r}_n$.

2.8.2 | Higher-Order Knuth-Bendix Order

Becker et al. [15] introduced a version of the Knuth-Bendix ordering that works on untyped applicative first-order terms (also known as λ-free higher-order terms; hence the name of the ordering and the title of this section). They introduced three versions of the ordering and proved that two are simplification orderings whilst the third is nearly a simplification ordering, but lacks compatibility with contexts. I present a simplified version of one of their orderings, the basic higher-order KBO without argument coefficients. Whilst it cannot be used directly on the typed applicative terms presented above, it can be used on such terms by conflating type and term arguments as described in [19].

Standard first-order KBO first compares the weights of terms, then compares their head-symbols and finally compares arguments recursively. When working with applicative first-order terms, the head symbol may be a variable. To allow the comparison of variable heads, a mapping ghd is introduced that maps variable heads to members of Σ that could possibly instantiate the head. This mapping *respects arities* if for any variable x, all members of $ghd(x)$ have arities greater or equal to that of x. The mapping can be extended to constant heads by taking $ghd(\mathsf{f}) = \{\mathsf{f}\}$. A substitution σ *respects* the mapping ghd, if for all variables x, $ghd(x\sigma) \subseteq ghd(x)$.

Let $\succ$ be a total well-founded ordering or *precedence* on Σ. The precedence $\succ$ is extended to arbitrary heads by defining $\zeta \succ \xi$ iff $\forall f \in ghd(\zeta)$ and $\forall g \in ghd(\xi), f \succ g$. Let w be a function from Σ to $\mathbb{N}$ that denotes the weight of a function symbol and $\mathcal{W}$ a function from $\mathcal{T}_{\Sigma}(\mathcal{V})$ to $\mathbb{N}$ denoting the weight of a term. Let $\varepsilon \in \mathbb{N}_{>0}$. For all constants c, $w(\mathsf{c}) \geq \varepsilon$. The weight of a term is defined recursively:

$$\mathcal{W}(\mathsf{f}) = w(\mathsf{f}) \qquad \mathcal{W}(x) = \varepsilon \qquad \mathcal{W}(s\,t) = \mathcal{W}(s) + \mathcal{W}(t)$$

The graceful higher-order basic Knuth-Bendix order $>_{\mathsf{hb}}$ is defined inductively as follows. Let $t = \zeta\,\overline{s}$ and $s = \xi\,\overline{r}$. Then $t >_{\mathsf{hb}} s$ if $vars_{\#}(s) \subseteq vars_{\#}(t)$ and:

- $\mathcal{W}(t) > \mathcal{W}(s)$ or $\mathcal{W}(t) = \mathcal{W}(s)$ and either of the following are satisfied:
  - $\zeta \succ \xi$;
  - $\zeta = \xi$ and $\overline{s} \gg_{\mathsf{hb}}^{\mathsf{length\_lex}} \overline{r}$.

2.9 | Proof Calculi

A logical inference is written:

$$\frac{C_1 \ \ldots \ C_n}{D}$$

where $\overline{C}_n$ and D are clauses. It is possible to consider inferences that work on formulas, but I do not do so in the rest of this thesis. The clauses $\overline{C}_n$ are known as the *premises* of the inference and the clause D is called the *conclusion*. At times, an inference is written on a single line as $C_1, \ldots, C_n \vdash D$ where $\overline{C}_n$ are again the premises and D is the conclusion.

Let inf be an inference. Then $prems(inf)$ refers to the set of its premise(s) whilst $conc(inf)$ refers to its conclusion. An inference i is *sound* if $prems(i) \models \{conc(i)\}$ where $\models$ is the entailment relation for the logic under consideration. A schema for inferences is known as an *inference rule*. One or more inference rules form an *inference system* or *proof calculus*.

2.10 | First-Order Superposition

A sound and complete inference system for first-order logic with equality is first-order superposition (until Section 2.11, I refer to "first-order superposition" as "superposition"). Roughly, superposition is the replacement of equals with equals. However, uncontrolled replacement leads to an explosion in the size of the clause set. Therefore, superposition contains many refinements such as the use of term orderings, redundancy criteria and related simplification inferences.

Superposition is a refutation based proof method. It attempts to derive a contradiction from a negated conjecture and a set of axioms. Though superposition does not have to work on clauses [61], in practice this is how it is commonly presented and used. The core inferences of the superposition calculus are presented below. The main rule is SUPERPOSITION or SUP which involves rewriting with positive equality literals. The main difference between superposition and paramodulation (unordered rewriting) is the use of a simplification order $\succ$ to restrict the inferences possible. In superposition, the *right* or *main* premise refers to the clause being rewritten.

$$\frac{D = D' \vee t \approx t' \qquad C = C' \vee [\neg]s[u] \approx s'}{(C' \vee D' \vee [\neg]s[t'] \approx s')\sigma}\ \textsc{Sup}$$

with the following side conditions:

1. u is not a variable;
2. $\sigma = mgu(t, u)$;
3. $t\sigma \not\preceq t'\sigma$;
4. $s[u]\sigma \not\preceq s'\sigma$;
5. $C\sigma \not\preceq D\sigma$;
6. $(t \approx t')\sigma$ is strictly maximal in $D\sigma$;
7. $([\neg]\, s[u] \approx s')\sigma$ is maximal in $C\sigma$, and strictly maximal if it is positive.

The other two core rules that form the superposition calculus are the EQRES and EQFACT inferences:

$$\frac{C' \vee u \not\approx u'}{C'\sigma}\ \text{EQRES} \qquad \frac{C' \vee u' \approx v' \vee u \approx v}{(C' \vee v \not\approx v' \vee u \approx v')\sigma}\ \text{EQFACT}$$

For both inferences $\sigma = mgu(u, u')$. For EQRES, $(u \not\approx u')\sigma$ is maximal in the premise. For EQFACT, $u'\sigma \not\preceq v'\sigma$, $u\sigma \not\preceq v\sigma$, and $(u \approx v)\sigma$ is maximal in the premise.

2.10.1 | Redundancy Criteria

Superposition as described above still leads to the generation of a large number of clauses. What we would like would be to delete clauses which can be shown to be in some sense unnecessary. To this end, the concept of *redundancy criteria* was introduced into superposition. A redundancy criterion defines a mapping from sets of clauses to sets of redundant clauses and inferences with respect to an inference system and entailment relation. Intuitively, a clause redundant with respect a set of clauses can be removed without impairing the completeness of the calculus with respect to the entailment relation. Definitions 21 and 23 define a redundancy criterion for superposition and standard first-order entailment based on [124].

Definition 21 (Redundant Clause)**.** Let D be a ground clause and N a set of ground clauses. Then D is redundant with respect to N, if $\{D' \mid (D' \in N) \wedge (D' \prec D)\} \models D$.

Let C be a (possibly non-ground) clause and N a set of (possibly non-ground) clauses. Let $\mathcal{G}_\Sigma(N)$ be the set of all ground instances of clauses in N and let $\sqsupset$ be a well founded ordering on clauses.[2] Then C is *redundant* with respect to N if for every clause $C\theta \in G_\Sigma(C)$ either:

1. $C\theta$ is redundant with respect $\mathcal{G}_\Sigma(N)$;
2. There exists $C' \in N$ such that $C \sqsupset C'$ and $C\theta$ is an instance of C'.

I write $Red_C(N)$ to represent the set of clauses redundant with respect to a clause set N. The redundancy criterion is left implicit in this notation, but should be clear from the context.

[2]The $\sqsupset$ order is known in the terminology of Waldmann et al. as a *tiebreaker* order and bears no connection with the order used to parameterise superposition. In the grounding $\mathcal{G}_\Sigma(N)$ of a set of clauses N, it is possible that a single clause $D \in \mathcal{G}_\Sigma(N)$ is the ground instance of multiple clause $C_1, \ldots, C_n \in N$. The tiebreaker ordering is used to select one of these clauses as the "parent" of D.

Definition 22 (Ground Instance of an Inference)**.** Let inf be an inference $C_1, \ldots, C_n \vdash D$ (n is either 1 or 2 depending on the inference). Let θ be a grounding substitution. If $C_1\theta, \ldots, C_n\theta \vdash D\theta$ is an inference, it is the θ-ground instance of inf, written $inf\theta$.

In order to understand why the above definition has not been written more simply as: for every inference $inf = C_1, \ldots, C_n \vdash D$ and grounding substitution θ, $C_1\theta, \ldots, C_n\theta \vdash D\theta$ is a ground instance of inf, consider the following example:

Example 2. Consider the EQRES inference:

$$\frac{\mathsf{f}(x) \not\approx \mathsf{f}(a) \vee \mathsf{g}(y) \not\approx \mathsf{g}(a)}{\mathsf{f}(x) \not\approx \mathsf{f}(a)} \text{ EQRES}$$

Since the literals $\mathsf{f}(x) \not\approx \mathsf{f}(a)$ and $\mathsf{g}(y) \not\approx \mathsf{g}(a)$ are incomparable, they are both maximal. Let $\theta = \{x \to \mathsf{a}, y \to \mathsf{a}\}$. Assume that $\mathsf{f}(a) \succ \mathsf{g}(b)$. Then $\mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(a) \vee \mathsf{g}(\mathsf{a}) \not\approx \mathsf{g}(a) \vdash \mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(a)$ is not an inference at all since $\mathsf{f}(\mathsf{a}) \not\approx \mathsf{f}(a)$, not $\mathsf{g}(\mathsf{a}) \not\approx \mathsf{g}(a)$, is the maximal literal of the 'premise'.

Definition 23 (Redundant Inference)**.** A ground inference is redundant if any of its premises are redundant, or if its conclusion is entailed by clauses smaller than its maximal premise. A non-ground inference is redundant if all its ground instances are. I write $Red_I(N)$ to represent the set of clauses redundant with respect to a clause set N. Again, the redundancy criterion involved should be clear from the context.

Definition 24 (Saturated Clause Set)**.** A clause set N is saturated with respect to the superposition calculus if all inferences from N are redundant.

2.10.2 | Refutational Completeness

Refutational completeness comes in two versions, static and dynamic. I provide the definitions of both concepts here. The definitions are provided with respect to the superposition calculus and redundancy criterion provided above, but can easily be generalised to an arbitrary calculus and redundancy criterion [123].

Definition 25 (Static Refutational Completeness)**.** Superposition is statically refutationally complete if, for every set of clauses N saturated with respect to superposition, either $\bot \in N$ or N has a model.

Definition 26 (Dynamic Refutational Completeness)**.** A possibly infinite sequence of clause sets $N_1, N_2, \ldots$ is called a *superposition derivation* if:

- For all clauses C in $N_{i+1} \setminus N_i$, C is the conclusion of a superposition inference from clauses in N_i.
- For all clauses C in $N_i \setminus N_{i+1}$, C is redundant with respect to clauses in N_{i+1}.

A clause C is *persistent* in a superposition derivation if there exists an i such that for all $j \geq i$, $C \in N_j$. The set of all persistent clauses is $N^* = \cup_{i\geq 1} \cap_{j\geq i} N_j$. A superposition derivation is *fair* if all inferences from N^* are included in $\cup_{i\geq 1} Red_I(N_i)$.

If for every fair superposition derivation $N_1, N_2 \ldots$, such that N_1 is unsatisfiable, there exists an i such that $\bot \in N_i$, then superposition is dynamically refutationalally complete.

It is dynamic refutational completeness that is of interest, since sequences of clause sets can be used to model a prover run and thus be used to show that a particular prover is complete.[3] In the next section, I provide an overview of the static completeness proof of superposition. Static completeness implies dynamic completeness, provided the redundancy criterion and entailment function meet certain criteria [123, Theorem 17].

I provide an overview of the static completeness proof of superposition as originally given by Bachmair and Ganzinger [9]. The style I present the proof in is close to that of Waldmann [122], but with some modifications.

Let N be a clause set saturated by superposition. The completeness proof considers $\mathcal{G}_\Sigma(N)$, the set of all ground instances of clauses in N, and shows that if $\bot \notin N$, then $\mathcal{G}_\Sigma(N)$ has a model. This model can then be lifted to be a model of N.

The key idea behind Bachmair and Ganzinger's proof is to build a confluent and terminating ground rewrite system R that induces an interpretation $\mathcal{T}_\Sigma(\emptyset)/R$, by defining $\mathcal{T}_\Sigma(\emptyset)/R \models s \approx t$ for all ground equations $s \approx t$ iff $s \downarrow_R t$. Well-founded induction on clauses in $\mathcal{G}_\Sigma(N)$ is then used to prove that $\mathcal{T}_\Sigma(\emptyset)/R$ is a model of $\mathcal{G}_\Sigma(N)$.

Construction of Interpretation

Let N be a clause set saturated by superposition and not containing $\bot$. Assume that all clauses in N are variable disjoint. Let $\mathcal{G}_\Sigma(N)$ be the set of all ground instances of clauses in N. For every clause $C \in \mathcal{G}_\Sigma(N)$, induction on $\succ$ is used to define sets of rewrite rules E_C and R_C. Assume that E_D has been defined for all clauses $D \in \mathcal{G}_\Sigma(N)$ such that $D \prec C$. Then R_C is defined as $\bigcup_{D\prec C} E_D$. The set E_C contains the rewrite rule $s \to t$ if the following conditions are met. Otherwise $E_C = \emptyset$.

(a) $C = C' \vee s \approx t$
(b) $s \approx t$ is strictly maximal in C
(c) $s \succ t$
(d) C is false in R_C
(e) C' is false in $R_C \cup \{s \to t\}$
(f) s is not reducible by any rule in R_C

In this case C is called *productive*. R_∞ is defined as $\bigcup_{C\in\mathcal{G}_\Sigma(N)} E_C$. By an abuse of notation, R_∞ and R_C are used for both the rewrite systems defined above and for the

[3] Most provers work with multiple clause set. This can be modelled using more complex notions of inference and redundancy [123]

interpretations induced by them. I next state some lemmas which are crucial for proving R_∞ is a model of $\mathcal{G}_\Sigma(N)$. Most of the lemmas are stated without proof here, their proofs being widely available in the literature.

Lemma 2. *The rewrite systems R_∞ and R_C are confluent and terminating for all C.*

Lemma 3. *If a clause $D \in \mathcal{G}_\Sigma(N)$ is true in R_D, it is true in $Rinf$ and R_C for all $C \succ D$.*

Lemma 4. *If a clause $D = D' \vee s \approx t \in \mathcal{G}_\Sigma(N)$ is productive, then D' is false and D is true in R_∞ and R_C for all $C \succ D$.*

Lemma 5. *Let $C = C' \vee [\neg]s \approx s'$ and $D = D' \vee t \approx t'$ be members of N and $C\theta$ and $D\theta$ be ground instances of C and D respectively. Assume that there is a ground* SUP *inference possible between $C\theta$ and $D\theta$ with $C\theta$ as the right premise, $t\theta \approx t'\theta$ the superposing equation and some subterm u of $s\theta$ the superposed subterm. Furthermore, assume that u occurs at or below a variable in s. Then the inference is redundant.*

Proof. The SUP inference between $D\theta$ and $C\theta$ must be of the form:

$$\frac{D'\theta \vee t\theta \approx t'\theta \qquad C' \vee [\neg]s\theta[t\theta] \approx s'\theta}{E = C'\theta \vee D'\theta \vee [\neg]s\theta[t'\theta] \approx s'\theta}\ \text{SUP}$$

Let R be an interpretation that models all clauses in $\mathcal{G}_\Sigma(N)$ smaller than $C\theta$. To show that the inference is redundant, I need to show that $R \models E$. By the side conditions of superposition, $D\theta \prec C\theta$ and therefore $R \models D\theta$. If $R \models D'\theta$, $R \models E$ is obvious. Therefore, assume that $R \not\models D'\theta$ and $R \models t\theta \approx t'\theta$. By congruence, it suffices to prove that $R \models C\theta$.

Since the inference occurs at or below a variable, there must exist a variable x in s such that $x\theta|_p = t\theta$ for some position p. Let w be the term obtained by replacing $t\theta$ with $t'\theta$ in $x\theta$. Let θ' be the substitution $\theta[x \to w]$. By the compatibility with contexts of the ordering $\succ$, it follows that $C\theta' \prec C\theta$. Thus, $R \models C\theta'$ and by congruence, $R \models C\theta$. □

Lemma 6 (Lifting Lemma 1)**.** *Every* EQRES*,* EQFACT *inference from clauses in $\mathcal{G}_\Sigma(N)$ is the ground instance of a* EQRES *or* EQFACT *inference from clauses in N.*

Lemma 7 (Lifting Lemma 2)**.** *Let $D = D' \vee t \approx t'$ and $C = C' \vee [\neg]s \approx s'$ be clauses in N. Let $D\theta$ and $C\theta$ be ground instances of D and C respectively. Assume that $s\theta$ has a subterm $u = t\theta$ and that there is a* SUP *inference between $D\theta$ and $C\theta$ with $t\theta \approx t'\theta$ being the superposing equation and u being the superposed subterm. If u does not occur at or below a variable in s, this inference is the ground instance of an inference from D and C.*

The following theorem shows that the interpretation R_∞ is a model of $\mathcal{G}_\Sigma(N)$.

Theorem 2 (Model Construction)**.** *Let $C \in N$ and $C\theta \in \mathcal{G}_\Sigma(N)$, then*

(i) *$E_{C\theta} = \emptyset$ if and only if $R_{C\theta} \models C\theta$;*

(ii) *if $C\theta$ is redundant with respect to $\mathcal{G}_\Sigma(N)$ then $C\theta$ is true in $R_{C\theta}$;*
(iii) *$C\theta$ holds in R_∞ and R_D for all $D \in \mathcal{G}_\Sigma(N), D \succ C\theta$;*

Proof. A partial proof providing the main ideas is provided here. The proof proceeds by well founded induction on the the clause size. It is assumed that (i)–(iii) are true for all clauses in $\mathcal{G}_\Sigma(N)$ smaller than $C\theta$. The proof then shows how this implies that (i)–(iii) are true for $C\theta$. Note that for (i), if $R_{C\theta} \models C\theta$, it is obvious from the construction of $E_{C\theta}$ that $E_{C\theta}$ must be the empty set. So it only remains to prove the $\Longrightarrow$ direction. Moreover, (iii) follows from (i) and Lemmas 3 and 4.

Assume that $C\theta$ is redundant with respect to $\mathcal{G}_\Sigma(N)$. By the definition of redundancy for ground clauses, that means that $C\theta$ is entailed by clauses smaller than itself in $\mathcal{G}_\Sigma(N)$. By the induction hypothesis, these clauses are all true in $\mathcal{G}_\Sigma(N)$. Therefore, $C\theta$ is true in $\mathcal{G}_\Sigma(N)$.

Assume that $C\theta = C'\theta \vee s\theta \not\approx s'\theta$ is a non-redundant clause with $s\theta \not\approx s'\theta$ its maximal literal and $s\theta \succ s'\theta$. If $R_{C\theta} \models s \not\approx s'$, $C\theta$ would be true in $R_{C\theta}$ and there would be nothing to prove. Therefore, assume that $R_{C\theta} \models s \not\approx s'$ doesn't hold and that $s \downarrow_{R_{C\theta}} s'$. That means that $s'\theta$ must be reducible by some rule $t\theta \to t'\theta$ in $R_{C\theta}$. Assume that the rule $t\theta \to t'\theta$ is produced by a clause $D\theta = D'\theta \vee t\theta \approx t'\theta$ which is a ground instance of clause $D = D' \vee t \approx t'$. Since $D\theta$ is productive, part (ii) of the induction hypothesis provides that it is not redundant with respect to $\mathcal{G}_\Sigma(N)$. Consider the SUP inference between $D\theta$ and $C\theta$:

$$\frac{D'\theta \vee t\theta \approx t'\theta \qquad C'\theta \vee [\neg]s\theta[t\theta] \approx s'\theta}{C'\theta \vee D'\theta \vee [\neg]s\theta[t'\theta] \approx s'\theta}\ \text{SUP}$$

I consider two cases depending on whether $t\theta$ occurs in $s\theta$ at or below a variable of s.

Case 1: If $t\theta$ occurs in $s\theta$ at or below a variable of s, then by Lemma 5, the conclusion of this inference is entailed by clauses in $\mathcal{G}_\Sigma(N)$ smaller than $C\theta$ and therefore true in $R_{C\theta}$.

Case 2: If $t\theta$ does not occur in $s\theta$ at or below a variable of s, by Lemma 7, the inference is the ground instance of an inference from C and D. Since N is saturated, the inference from C and D must be redundant. Therefore, the conclusion of its θ-ground instance, $C'\theta \vee D'\theta \vee [\neg]s\theta[t'\theta] \approx s'\theta$ must be entailed by clauses smaller than $C\theta$ and therefore true in $R_{C\theta}$.

In either case the conclusion is true in $R_{C\theta}$. Since $D\theta$ is productive, $D'\theta$ is false in $R_{C\theta}$ by Lemma 4 and $s\theta[t\theta] \approx s'\theta$ is also false $R_{C\theta}$, so it must be the case that $C'\theta$ and thus $C\theta$ are true in $R_{C\theta}$. □

The remaining cases, such as when $C\theta$ has a maximal positive literal, can be dealt with in a like manner. Using Theorem 2, the static refutational completeness of superposition can be shown easily. Using Waldmann et al.'s framework, static refutational completeness can be shown to imply dynamic refutational completeness [124].

2.10.3 | Simplification Inferences

Dynamic refutational completeness shows that clauses can be deleted during proof search without affecting completeness. However, redundancy is a semantic concept that is not, in general, decidable. In some circumstances redundancy of a clause can be shown, leading to the concept of simplification inferences. A simplification inference is an inference that renders one or more of its premises redundant in the presence of its conclusion. An inference that is not simplifying is known as a *generating* inference. Common simplification inferences used in conjunction with superposition include demodulation. In the following presentation, $[\![\, C \,]\!]$ is used to denote a premise C that is redundant in the presence of the conclusion and can therefore be removed. Some simplification rules have no conclusions, in which case they are simply the deletion of a redundant clause.

$$\frac{t \approx t' \qquad [\![\, C' \vee [\neg] s[u] \approx s' \,]\!]}{C' \vee [\neg] s[t'\sigma] \approx s'} \textsc{Demod}$$

where $u = t\sigma$ and $t\sigma \succ t'\sigma$. Via congruence, the right premise is entailed by the conclusion and the left premise. Other common rules include duplicate literal removal and tautology deletion:

$$\frac{[\![\, C' \vee s \approx t \vee s \approx t \,]\!]}{C' \vee s \approx t} \textsc{DupLitRem} \qquad \frac{[\![\, C' \vee s \approx s \,]\!]}{} \textsc{TautologyDel}$$

A particularly powerful simplification rule is subsumption.

$$\frac{C \qquad [\![\, C' \vee D' \,]\!]}{} \textsc{Subsumption}$$

where $C' = C\sigma$ for some substitution σ and D' may possibly be empty. To justify the deletion of the right premise, instantiate the well-founded ordering, $\sqsupset$, in the redundancy criterion with the *whole clause subsumption ordering* $\gtrdot$. For clauses C and C', $C' \geq C$ if there exists a substitution σ such that $C' = C\sigma$. The whole clause subsumption ordering, $\gtrdot$, is defined as $\geq \setminus \leq$.

Let θ be a grounding substitution. Assume that D' is not empty. It is clear that for any grounding substitution θ, $C\sigma\theta$ entails and is smaller than $(C' \vee D')\theta$. Thus, C makes $C' \vee D'$ redundant. If D' is empty, then $C' \gtrdot C$ and every ground instance of C' is also a ground instance of C, so by the definition of clause redundancy, $C' = C' \vee D'$ is still redundant.

2.11 | Applicative First-Order Superposition

In this section I discuss a refutationally complete superposition calculus for applicative first-order logic. It is straightforward to show that by considering subterms as defined in Section 2.3 instead of Section 2.1, the suprposition calculus provided above is already refutationally complete for applicative first-order logic. However, a number of term orderings that work on applicative terms lack full compatibility with contexts and are therefore not simplification orderings. An example is Blanchette et al.'s higher-order recursive path order (RPO) [38] which lacks a property known as compatibility with arguments. Let $>_{\mathsf{ho}}$ be the higher-order RPO and let s and s' be terms such that $s >_{\mathsf{ho}} s'$. It is not necessarily the case that $s\,t >_{\mathsf{ho}} s'\,t$. If a term ordering that is not compatible with arguments is used, standard superposition is no longer complete as the following example shows.

Example 3. Consider the unsatisfiable clause set $\{\mathsf{g} \approx \mathsf{f}, \mathsf{g}\,\mathsf{a} \not\approx \mathsf{f}\,\mathsf{a}\}$ where $\mathsf{g} >_{\mathsf{ho}} \mathsf{f}$, but $\mathsf{f}\,\mathsf{a} >_{\mathsf{ho}} \mathsf{g}\,\mathsf{a}$. Clearly, there is no inference possible from the clause set.

Bentkamp et al. have designed a family of calculi that are refutationally complete for applicative first-order logic in the presence of orderings that are not compatible with arguments [19]. The crucial idea behind their calculi is to not superpose at prefix subterms, but only at green subterms. A new inference rule, ARGCONG, is then introduced to "grow" positive equalities to the right sizes. Here, I provide the inference rules for the intensional non-purifying version of their calculi family. The calculus supports *selection of negative literals*, a concept that I investigate more deeply in Chapter 4 of this thesis. A selection function is a function that maps a clause to a subset of its negative literals. In the presence of a selection function and a substitution σ, a literal l is *σ-eligible* in a clause C if it is selected in C, or C has no selected literals and $l\sigma$ is maximal in $C\sigma$. It is strictly σ-eligible if there are no selected literals and $l\sigma$ is strictly maximal. The SUP inference for the *clausal intensional λ-free higher-order superposition* calculus:

$$\frac{D' \vee t \approx t' \qquad C' \vee [\neg] s\langle u\rangle \approx s'}{(C' \vee D' \vee [\neg] s\langle t'\rangle \approx s')\sigma}\ \textsc{Sup}$$

with the following side conditions:

1. The variable condition (below) holds
3. $\sigma = mgu(t, u)$;
4. $t\sigma \not\preceq t'\sigma$;
5. $s\langle u\rangle\sigma \not\preceq s'\sigma$;
6. $C\sigma \not\preceq D\sigma$;
7. $t \approx t'$ is strictly σ-eligible in D;
8. $[\neg]\, s\langle u\rangle \approx s'$ is σ-eligible in C, and strictly σ-eligible if it is positive.

Variable condition: $u \notin \mathcal{V}$ or there exists a grounding substitution θ such that $t\sigma\theta \succ t'\sigma\theta$, but $C\sigma\theta \prec C\{u \to t'\}\sigma\theta$.

The variable condition essentially mandates superposition into variables in cases where, due to the order's non-monotonicity, $C\sigma\theta \prec C\{u \to t'\}\sigma\theta$ thereby precluding the use

of Lemma 5 to rule out the need for the inference. Note that the condition is a semantic condition and cannot be effectively checked. However, it can be approximated.

What is a Variable Condition?

This thesis uses the terminology *variable condition* for two different concepts. Firstly, for a sufficient syntactic condition for non-ground terms to be comparable by a KBO-like ordering. For example, the standard first-order KBO ordering described in Section 2.8.1 has variable condition $vars_{\#}(s) \subseteq vars_{\#}(t)$. Secondly, as a side condition of the superposition rule as above. These two usages are **not related**, but unfortunately the term is used in both senses in the existing literature [7, 19].

In Chapter 3 "variable condition" refers to the syntactic condition for non-ground terms to be comparable via a KBO-like order. In Chapter 4 "variable condition" refers to the side condition of superposition.

The EQRES and EQFACT inferences:

$$\frac{C' \vee u \not\approx u'}{C'\sigma}\ \text{EQRES} \qquad \frac{C' \vee u' \approx v' \vee u \approx v}{(C' \vee v \not\approx v' \vee u \approx v')\sigma}\ \text{EQFACT}$$

For both inferences $\sigma = mgu(u, u')$. For EQRES, $u \not\approx u'$ is σ-eligible in the premise. For EQFACT, $u'\sigma \not\preceq v'\sigma$, $u\sigma \not\preceq v\sigma$, and $u \approx v$ is σ-eligible in the premise. Finally, the ARGCONG inference which facilitates the simulation of superposition at prefix positions.

$$\frac{\begin{array}{l} C' \vee s \approx s' \end{array}}{\begin{array}{l} C'\sigma \vee (s\sigma)x \approx (s'\sigma)x \\ C'\sigma \vee (s\sigma)\overline{x}_2 \approx (s'\sigma)\overline{x}_2 \\ C'\sigma \vee (s\sigma)\overline{x}_3 \approx (s'\sigma)\overline{x}_3 \\ \vdots \end{array}}\ \text{ARGCONG}$$

The literal $s \approx s'$ must be σ-eligible in C. Let s and s' be of type $\alpha_1 \to \cdots \to \alpha_m \to \beta$. If β is not a type variable, then σ is the identity substitution and the inference has m conclusions. Otherwise, if β is a type variable, the inference has an infinite number of conclusions. In conclusions where $n > m$, σ is the substitution that maps β to type $\tau_1 \to \cdots \to \tau_{n-m} \to \beta'$ where β' and each τ_i are fresh type variables. In each conclusion, the x_is are variables fresh for C.

In the remainder of this thesis, when I refer to the clausal λ-free superposition calculus, I refer to one of the four calculi developed by Bentkamp et al. in [19].

2.11.1 | Refutational Completeness

In this section I provide an overview of the refutational completeness proof for the clausal intensional λ-free higher-order superposition calculus. Let N be a set of clauses over

a signature $(\Sigma_{\mathsf{ty}}, \Sigma)$, saturated by the calculus and not containing the empty clause. To prove that N has a model, we would like to use the model existence proof given in Theorem 2. However, this proof makes use of the fact that the subterms available for the candidate rewrite model are the same as those available for the superposition calculus. This no longer holds, since superposition only works at green subterms, whilst the rewrite system can rewrite arbitrary subterms. The authors of [19] resolve this issue, by translating the saturated set N to a monomorphic (non-applicative) first-order logic they call the *floor logic*. Since the floor logic is non-applicative, there are no prefix subterms, and the mismatch between the subterms available to the rewrite model and the calculus is mended. They then apply the standard model building technique to prove the existence of a model for the translated set of clauses and show that this model can be transferred to a model of the original polymorphic applicative clause set. I define the floor logic and the translation $\lfloor \ \rfloor$ of ground applicative first-order terms to floor logic terms.

Definition 27 (Floor Logic, Floor Translation)**.** Let $(\Sigma^{\mathsf{GF}}_{\mathsf{ty}}, \Sigma^{\mathsf{GF}})$ be the signature of the floor logic. $\Sigma^{\mathsf{GF}}_{\mathsf{ty}}$ can be any infinite set of sorts. The $\lfloor \ \rfloor$ translation is an arbitrary bijection between members of $\mathcal{T}\mathit{ya}_{\Sigma_{\mathsf{ty}}}(\emptyset)$ and members of $\mathcal{T}\mathit{ya}_{\Sigma^{\mathsf{GF}}_{\mathsf{ty}}}$.

For each function symbol $\mathsf{f} : \Pi\overline{\alpha}_m.\, \tau_1 \to \cdots \to \tau_n \to \tau$ in Σ, a family of functions symbols $\mathsf{f}_i^{\overline{v}_m} : (\lfloor \tau_1\sigma \rfloor \times \ldots \times \lfloor \tau_i\sigma \rfloor) \to \lfloor (\tau_{i+1} \to \cdots \to \tau_n \to \tau)\sigma \rfloor$, for each tuple of ground types $\overline{v}_m$ and for each i in $1 \leq i \leq n$, are included in Σ^{GF} where σ is the substitution that maps α_1 to v_1, α_2 to v_2 and so on. The translation of terms and literals is as follows:

- $\lfloor \mathsf{f}\langle\overline{\tau}\rangle\, \overline{s}_n \rfloor = \mathsf{f}^{\overline{\tau}}_n(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor)$;
- $\lfloor s \approx t \rfloor = \lfloor s \rfloor \approx \lfloor t \rfloor$

A clause is translated by translating its literals. A clause set is translated by translating its clauses. The crucial feature of the translation is that subterms in the floor logic match exactly with green subterms in the applicative logic thereby repairing the mismatch discussed above.

Example 4. Consider the ground applicative first-order clause set $\{\mathsf{f}\langle\tau\rangle\, \mathsf{a}\, \mathsf{b} \approx \mathsf{g}\langle\tau\rangle\, \mathsf{a} \vee \mathsf{c} \approx \mathsf{a}, \mathsf{f}\langle\tau\rangle \approx \mathsf{h}\}$. The floor of the clause set is $\{\mathsf{f}^{\tau}_2(\mathsf{a}_0, \mathsf{b}_0) \approx \mathsf{g}^{\tau}_1(\mathsf{a}_0) \ \vee \ \mathsf{c}_0 \approx \mathsf{a}_0, \mathsf{f}^{\tau}_0 \approx \mathsf{h}_0\}$.

The proof works by building a model of $\lfloor \mathcal{G}_\Sigma(N) \rfloor$, then lifting this to a model of $\mathcal{G}_\Sigma(N)$ and then finally to a model of N. Since $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ is a standard first-order clause set, techniques very similar to those discussed in Section 2.10.2, can be used to develop a model for it. I use a very similar method to prove refutational completeness for the clausal combinatory superposition calculus in Chapter 4. Note that the presentation of the completeness proof presented here follows Bentkamp et al.'s original presentation [19]. In an upcoming submission, they have updated the presentation to make use of Waldmann et al's saturation framework [20].

2.12 | Combinatory Logic

Combinatory logic is applicative first-order logic where certain function symbols, known as *combinators*, have a special semantics. A combinator is a higher-order function that duplicates, deletes or permutes its arguments. Many different combinators have been defined for various purposes. A commonly used set of combinators is the $\{\mathsf{S}, \mathsf{B}, \mathsf{C}, \mathsf{K}, \mathsf{I}\}$ set, popularised by Turner [115]. The semantics of each of these combinators is provided by its governing equation given in Section 2.7. In this section, I provide the details of combinatory logic required to understand the remainder of the thesis. For a more in depth treatment of the topic please view Hindley and Seldin's excellent work [69]. In the context of this thesis, *combinator* refers to a member of the set $\{\mathsf{S}, \mathsf{B}, \mathsf{C}, \mathsf{K}, \mathsf{I}\}$. There is nothing special about this set, and indeed any function expressible with the above set can be expressed with the set $\{\mathsf{S}, \mathsf{K}\}$. However, by using the B-, C- and K-combinators, some functions can be expressed via exponentially smaller terms [86].

Remark 10. The type arguments of a function symbol are dropped from the notation at times. This is commonly done with combinator symbols.

Definition 28 (Combinator Axiom, Extended Combinator Axiom)**.** By *combinator axiom*, I refer to one of the five defining equations provided in Section 2.7. Let $l \approx r$ be a combinator axiom. Then for every tuple of fresh variables $\overline{x}$, including the empty tuple, $l\,\overline{x} \approx r\,\overline{x}$ is an *extended combinator axiom*.

Definition 29 (Weak Reduction)**.** Weak reduction is the combinatory analogue of β-reduction. Formally, let $l \approx r$ be a combinator axiom. A term t *weak-reduces* to a term t' in one step (denoted $t \longrightarrow_w t'$) if $t = u[s]_p$ and there exists a substitution σ such that $l\sigma = s$ and $t' = u[r\sigma]_p$. The term $l\sigma$ in t is called a *weak redex* or just redex. By $\longrightarrow_w^*$, the reflexive transitive closure of $\longrightarrow_w$ is denoted. If a term t weak-reduces to a term t' in n steps, I write $t \longrightarrow_w^n t'$. Further, if there exists a weak reduction path from a term t of length n, I write that $t \in n_w$. Weak reduction is terminating and confluent as proved in [69]. By $(t)\downarrow^w$, I denote the term formed from t by contracting its leftmost redex. By $(t)\downarrow_w$, the unique weak normal form of t is denoted.

Example 5. In the term $\mathsf{f}\,(\mathsf{I}\langle\tau\rangle\,\mathsf{a})$, the subterm $\mathsf{I}\langle\tau\rangle\,\mathsf{a}$ is a weak redex that reduces to a. In the term $t = \mathsf{C}\langle\tau_1\,\tau_2, \tau_3\rangle\,\mathsf{a}\,\mathsf{b}\,\mathsf{c}\,\mathsf{d}$, the subterm $t|_{\epsilon.1} = \mathsf{C}\langle\tau_1\,\tau_2, \tau_3\rangle\,\mathsf{a}\,\mathsf{b}\,\mathsf{c}$ is a weak redex and $(t)\downarrow^w = \mathsf{a}\,\mathsf{c}\,\mathsf{b}\,\mathsf{d}$. Finally, in the term $\mathsf{S}\langle\tau_1\,\tau_2, \tau_3\rangle\,\mathsf{a}\,\mathsf{b}$ there is no weak redex.

In the λ-calculus, every λ-expression applied to a term forms a β-redex. As the final example above shows, a combinator applied to term(s) does not necessarily form a weak redex. It is often important to differentiate between a combinator that is applied to sufficient arguments to form a redex and one that is not. In what follows, the symbol $\mathcal{C}_{\mathsf{any}}$ denotes a member of $\{\mathsf{S}, \mathsf{C}, \mathsf{B}, \mathsf{K}, \mathsf{I}\}$, whilst $\mathcal{C}_{\mathsf{3}}$ denotes a member of $\{\mathsf{S}, \mathsf{C}, \mathsf{B}\}$. These

symbols are only used when the combinator is at the head of a redex. A combinator that is at the head of a redex is said to be *fully applied*. The symbol $\mathcal{C}_{\mathsf{any}}$ refers to an arbitrary combinator that is not fully applied whilst the symbol $\mathcal{C}_{\mathsf{any}}$ refers to a combinator regardless of its position. The symbols ζ, ξ range over all non-application terms.

Definition 30 (Stable Subterm)**.** I define a subset of the green subterms called *stable* subterms. Let $\mathsf{LPP}(t, p)$ (LPP stands for Longest Proper Prefix) be a partial function that takes a term t and a position p and returns the longest proper prefix p' of p such that $head(t|_{p'})$ is not a partially applied combinator if such a position exists. For a position $p \in pos(t)$, p is a stable position in t if $\mathsf{LPP}(t, p)$ is not defined or $head(t|_{\mathsf{LPP}(t,p)})$ is not a variable or combinator. A *stable subterm* is a subterm occurring at a stable position and is denoted $t\langle\langle u \rangle\rangle_p$. A subterm that is not stable is known as an *unstable* subterm.

Definition 31 (Stable Context)**.** A term with a hole at a stable position, denoted $t\langle\langle\ \rangle\rangle_p$, is called a *stable context*.

Example 6. The subterm a is not stable in $\mathsf{f}\,(\mathbf{S}\,\mathsf{a}\,\mathsf{b}\,\mathsf{c})$, $\mathbf{S}\,(\mathbf{S}\,\mathsf{a})\,\mathsf{b}\,\mathsf{c}$ (in both cases, $head(t|_{\mathsf{LPP}(t,p)}) = \mathbf{S}$) and $\mathsf{a}\,\mathsf{c}$ (a is not a green subterm), but is in $\mathsf{g}\,\mathsf{a}\,\mathsf{b}$ and $\mathsf{f}\,(\mathbf{S}\,\mathsf{a})\,\mathsf{b}$.

Lemma 8. *Let $t = s\langle\langle\ \rangle\rangle$ be a stable context with multiple holes. If $t \to_w t'$ then t' is also a stable context.*

Proof. It suffices to prove the theorem for the case $t = \mathcal{C}_{\mathsf{any}}\,\overline{s}_n$ and $t' = (t)\downarrow^w$. By the definition of stable position, no s_i can be a hole. Furthermore, for a hole that is a subterm of some s_j at position p, there exists a p' such that $\mathsf{LPP}(s_j, p) = p'$ and $head(s_j|_{p'})$ is not a variable or combinator. Using this and the fact that weak reduction only permutes, duplicated or deletes arguments, we can conclude that all holes in t' are stable as well. □

The length of the longest weak reduction from a term t is denoted $\|t\|$. This measure is one of the crucial features of the ordering investigated in Chapter 3. To show that the measure $\|\|$ is computable I provide a maximal weak reduction strategy and prove its maximality. The strategy is in a sense equivalent to Barendregt's 'perpetual strategy' in the λ-calculus [14]. My proof of its maximality follows the style of Van Raamsdonk et al. [117] in their proof of the maximality of a particular β-reduction strategy.

Lemma 9 (Fundamental Lemma of Maximality)**.** $\|\mathcal{C}_{\mathsf{any}}\,\overline{s}_n\| = \|(\mathcal{C}_{\mathsf{any}}\,\overline{s}_n)\downarrow^w\| + 1 + isK(\mathcal{C}_{\mathsf{any}}) \times \|s_2\|$ *where* $isK(\mathcal{C}_{\mathsf{any}}) = 1$ *if* $\mathcal{C}_{\mathsf{any}} = \mathbf{K}$ *and is* 0 *otherwise.*

Proof. The proof of this lemma can be found in Appendix A. □

Finally, I provide a particular weak reduction strategy and prove its maximality. The strategy is later used to prove a property by induction on the longest weak reduction length from terms (Lemma 17).

Lemma 10. *Define a map F_∞ from $\mathcal{T}_\Sigma(\mathcal{V})$ to $\mathcal{T}_\Sigma(\mathcal{V})$ as follows:*

$$\begin{aligned}
F_\infty(s) &= s \quad \textit{if } \|s\| = 0 \\
F_\infty(\zeta\,\overline{s}_n) &= \zeta\, s_1 \ldots s_{i-1}\, F_\infty(s_i)\, s_{i+1} \ldots s_n \\
&\quad \textit{where } \|s_j\| = 0 \textit{ for } 1 \leq j < i \\
&\quad \textit{and } \zeta \textit{ is not the head of a weak redex} \\
F_\infty(\boldsymbol{\mathcal{C}_3}\,\overline{s}_n) &= (\boldsymbol{\mathcal{C}_3}\,\overline{s}_n)\!\downarrow^w \\
F_\infty(\mathbf{I}\,\overline{s}_n) &= \overline{s}_n \\
F_\infty(\mathbf{K}\,\overline{s}_n) &= \begin{cases} s_1\, s_3 \ldots s_n & \textit{if } \|s_2\| = 0 \\ \mathbf{K}\, s_1\, F_\infty(s_2) \ldots s_n & \textit{otherwise} \end{cases}
\end{aligned}$$

The reduction strategy F_∞ is maximal.

Proof. By utilising Lemma 2.14 of [117], we have that F_∞ is maximal iff for all $m \geq 1$ and any term s, $s \in m_w \implies F_\infty(s) \in (m-1)_w$. I proceed by induction on s.

If $s = \mathsf{f}\langle\overline{\tau}\rangle\,\overline{s}\,u\,\overline{r}_n$ or $t = x\,\overline{s}\,u\,\overline{r}_n$ or $t = \boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}\,u\,\overline{r}_n$ where all members of $\overline{s}$ are in normal form, $\|u\| > 0$, $u \in m^0_w, r_1 \in m^1_w \ldots r_n \in m^n_w$ and $m = m^0 + \cdots + m^n$, then $F_\infty(s) = \zeta\,\overline{s}\,F_\infty(u)\,\overline{r}_n$. By the induction hypothesis $F_\infty(u) \in (m^0 - 1)_w$. Thus, $F_\infty(s) = \zeta\,\overline{s}\,F_\infty(u)\,\overline{r}_n \in (m-1)_w$.

If $t = \boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}_n$, the proof splits into two:

- If $\boldsymbol{\mathcal{C}_{\mathsf{any}}} \neq \mathbf{K}$ or $\|s_2\| = 0$, then $F_\infty(s) = (\boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}_n)\!\downarrow^w$. By the fundamental lemma of maximality, $\|(\boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}_n)\!\downarrow^w\| + 1 = \|\boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}_n\| \geq m$. Thus, $\|(\boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}_n)\!\downarrow^w\| \geq m-1$ and $F_\infty(s) \in (m-1)_w$.
- If $\boldsymbol{\mathcal{C}_{\mathsf{any}}} = \mathbf{K}$ and $\|s_2\| > 0$, then by the fundamental lemma of maximality we have that $\|(\boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}_n)\!\downarrow^w\| + \|s_2\| + 1 = \|\boldsymbol{\mathcal{C}_{\mathsf{any}}}\,\overline{s}_n\| \geq m$. By the induction hypothesis we have that $\|F_\infty(s_2)\| \geq \|s_2\| - 1$. Thus

$$\begin{aligned}
\|F_\infty(s)\| &= \|(s)\!\downarrow^w\| + \|F_\infty(s_2)\| + 1 \\
&\geq \|(s)\!\downarrow^w\| + \|s_2\| \\
&= \|s\| - 1 \\
&\geq m - 1
\end{aligned}$$

and so $F_\infty(s) \in (m-1)_w$. □

Chapter 3

A Modified Knuth-Bendix Order

> Mathematics is a game played according to certain simple rules with meaningless marks on paper.
>
> *David Hilbert*

Using the translation from higher-order logic to applicative first-order logic presented in the previous chapter, Bentkamp et al.'s extensional calculus is a complete proof method for higher-order logic. However, there is a severe drawback to translating higher-order problems to applicative first-order logic, and then carrying out proof search with an existing calculus. Consider standard superposition parameterised by the higher-order KBO described in Section 2.8.2. Such a calculus is complete for applicative first-order logic without ArgCong or the requirement to superpose at variables, since the higher-order KBO described is compatible with all contexts including arguments. However, the higher-order KBO is unable to orient the **S**-combinator axiom left to right, or indeed the axiom for any combinator that duplicates an argument. The left and the right hand side of such an axiom will either be incomparable by $>_{\mathsf{hb}}$, or the right hand side will be greater. To see why the left hand side cannot be larger than the right, it suffices to notice that the variable z appears once in $\mathbf{S}\,x\,y\,z$, but twice in the right hand side $x\,z\,(y\,z)$. Therefore, the condition $vars_{\#}(\mathbf{S}\,x\,y\,z) \subseteq vars_{\#}(x\,z\,(y\,z))$ is false. The result is that superposition inferences can take place onto the right hand side of the **S**-axiom. This can occur when the left premise is another combinator axiom or even the **S**-axiom itself. Figure 3.1 shows all the consequences of superposition with the **S**-axiom as both premises. Each conclusion shown in Figure 3.1 also contains variables that occur more often on the right hand side than the left hand side. Therefore, these conclusions can superpose with each other and with the original **S**-axiom. So far, I have not even considered the remaining combinator axioms which can superpose onto the right hand side of the **S**-axiom, or any of the conclusions shown in Figure 3.1.

The issue that I am highlighting here is that in the presence of term orders that are

$$\frac{\mathbf{S}\,x\,y\,z \approx x\,z\,(y\,z) \qquad \mathbf{S}\,x'\,y'\,z' \approx x'\,z'\,(y'\,z')}{\mathbf{S}\,(\mathbf{S}\,x\,y)\,y'\,z' \approx x\,z'\,(y\,z')\,(y'\,z')}$$

$$\frac{\mathbf{S}\,x\,y\,z \approx x\,z\,(y\,z) \qquad \mathbf{S}\,x'\,y'\,z' \approx x'\,z'\,(y'\,z')}{\mathbf{S}\,x'\,(\mathbf{S}\,x\,y)\,z' \approx x'\,z'\,(x\,z'\,(y\,z'))}$$

$$\frac{\mathbf{S}\,x\,y\,z \approx x\,z\,(y\,z) \qquad \mathbf{S}\,x'\,y'\,z' \approx x'\,z'\,(y'\,z')}{\mathbf{S}\,(\mathbf{S}\,x)\,y'\,z' \approx x\,(y'\,z')\,(z'\,(y'\,z'))}$$

Figure 3.1: Self superposition with **S**-axiom. The subterms highlighted in red are the superposed subterms. The variables of the left and right premises have been renamed apart to avoid confusion.

commonly used to parameterise superposition, superposition amongst the combinator axioms is highly explosive and can quickly swamp the search space. There is also another issue here. We would like a higher-order superposition calculus to be *graceful* in the sense that it behaves exactly as first-order superposition on a first-order problem. That is clearly not the case here, since the axioms superpose amongst themselves even when the problem is first-order and consequences of the axioms are irrelevant to finding a proof.

A solution to this issue would be to parameterise superposition with a simplification ordering that orients all combinator axioms left to right. Since superposition does not take place at variables, there could then be no inferences amongst the axioms. Not only would this curb the explosiveness described above, it would also lead to gracefulness, since when dealing with a first-order problem, the combinator axioms would play no part in proof search. In the next section, I explain the difficulties involved in deriving a simplification order with the desired property, and why I hypothesise that no such order exists. The remainder of the chapter introduces an order that is 'nearly' a simplification order and possesses the desired property.

3.1 | The Challenge

The challenge then, is to find a simplication order $\succ$, such that for every ground instance of a combinator axiom $l \approx r$, $l \succ r$. Concentration on ground instances suffices since, by the stability under substitution property of a simplification order, non-ground instances will be oriented as well. For a simplification ordering that orients combinator equations, it must necessarily be the case that for terms t_1 and t_2 such that $t_1 \rightarrow_w t_2$, t_1 is greater than t_2. This follows from compatibility with contexts. I call this property *compatibility with weak reduction* and an ordering that possesses it *weak-compatible*. An obvious method for designing a weak-compatible order is to modify an existing order.

There exists a wide range of first-order and higher-order simplification and reduction

orders such as the well known Knuth-Bendix and recursive path orders [51, 78]. A recent order that generalises both of these is the weighted path order [130]. Jouannaud and Rubio introduced a higher-order version of the RPO called HORPO [74]. HORPO is compatible with β-reduction which suggests that without much difficulty it could be modified to be compatible with weak reduction. However, the ordering does not enjoy the subterm property, nor is it transitive. Likewise is the case for orderings based on HORPO such as the computability path ordering [39] and the iterative HOIPO of Kop and van Raamsdonk [79]. More recently, a pair of orderings for λ-free higher-order terms have been developed [15, 38]. These orderings lack a specific monotonicity property, but this does not prevent their use in superposition [19]. Unfortunately, none of these orderings orient all grounds instances of combinator axioms, as the following examples illustrate.

Example 7. Consider the basic higher-order KBO as presented in Chapter 2. A simple idea is to give each combinator symbol some large weight, perhaps even a transfinite weight. Unfortunately, this does not suffice for orienting all ground instances. Consider the following ground instance of the $\mathbf{S}$-axiom: $\mathbf{S}\langle\tau_1, \tau_2, \tau_3\rangle\, t_1\, t_2\, t_3 \;\approx\; t_1\, t_3\, (t_2\, t_3)$. The ground term t_3 may itself contain $\mathbf{S}$-combinators and as it appears twice on the right hand side, there is no way to guarantee that the right hand side will weigh less than the left.

Example 8. Another possibility is to utilise Becker et al.'s graceful higher-order KBO with argument coefficients. Here, the idea is to give the third argument of the $\mathbf{S}$-symbol a coefficient of 2, to balance it appearing twice on the right. However, this too does not suffice for orienting all ground instances. Consider the following instance of the $\mathbf{S}$-axiom (type arguments ignored): $\mathbf{S}\ (\mathbf{S}\ \mathsf{a}\ \mathsf{b})\ \mathsf{c}\ t \approx \mathbf{S}\ \mathsf{a}\ \mathsf{b}\ t\ (\mathsf{c}\ t)$.

The weight of the left hand side is $(2 \times w(\mathbf{S})) + w(a) + w(b) + w(c) + (2 \times \mathcal{W}(t))$ whilst the weight of the right hand side is $w(\mathbf{S}) + w(a) + w(b) + w(c) + (3 \times \mathcal{W}(t))$. Thus, in all cases where $\mathcal{W}(t) > w(\mathbf{S})$, the axiom will be oriented right to left. It is possible to make the weight of t arbitrarily large, so there must exist some t such that $\mathcal{W}(t) > w(\mathbf{S})$.

Whilst the examples clarify some of the issues involved in deriving a simplification ordering compatible with weak reduction, they don't rule out the existence of such an ordering. I have been unable to derive a weak-compatible simplification ordering after extensive efforts. Moreover, experts in the field consider it unlikely that a weak-compatible simplification ordering exists. In private communication with Jean-Pierre Jouannaud, he commented "compatibility with beta-reduction. I doubt it exists". In the absence of a weak-compatible simplification ordering, the next best thing is a weak-compatible order that possesses all the qualities of a simplification ordering except compatibility with contexts. Such orderings have already been used to parameterise paramodulation calculi [40].

3.2 | The Combinatory Compatible KBO

The combinator orienting KBO that is the focus of this chapter is weak-compatible. This is achieved by first comparing terms by the length of the longest weak reduction from the term, and then using $>_{\mathsf{hb}}$, Becker et al's basic higher-order KBO [15]. This simple approach runs into problems with regards to stability under substitution, a crucial feature for any ordering used in superposition.

Consider the terms $t = \mathsf{f}\, x\, \mathsf{a}$ and $s = x\, \mathsf{b}$. As the length of the maximum reduction from both terms is 0, the terms would be compared using $>_{\mathsf{hb}}$ resulting in $t \succ s$ as $\mathcal{W}(t) > \mathcal{W}(s)$. Now, consider the substitution $\theta = \{x \to \mathbf{I}\}$. Then, $\|s\theta\| = 1$ whilst $\|t\theta\| = 0$ resulting in $s\theta \succ t\theta$.

The easiest and most general way of obtaining an order which is stable under substitution is to restrict the definition of the combinator orienting KBO to ground terms and then *semantically lift* it to non-ground terms by defining $s \succ t$ for non-ground terms iff $s\theta \succ t\theta$ for all grounding substitutions θ. However, the semantic lifting of the ground order is non-computable, hence useless for practical purposes. I therefore provide two approaches to achieving an ordering that can compare non-ground terms and is stable under substitution, both of which approximate the semantic lifting. Both require some conditions on the forms of terms that can be compared. The first is simpler, but more conservative than the second.

First, in the spirit of Bentkamp et al. [21], I provide a translation that replaces "problematic" subterms of the terms to be compared with fresh variables. With this approach, the simple variable condition of the standard KBO, $vars_{\#}(s) \subseteq vars_{\#}(t)$, ensures stability. However, this approach is over-constrained and prevents the comparison of terms such as $t = x\, \mathsf{a}$ and $s = x\, \mathsf{b}$ despite the fact that for all substitutions θ, $\|t\theta\| = \|s\theta\|$. Therefore, I present a second approach wherein no replacement of subterms occurs. This comes at the expense of a far more complex variable condition. Roughly, the condition stipulates that two terms are comparable if and only if the variables and relevant combinators are in identical positions in each.

Approach 1

Because the $>_{\mathsf{hb}}$ ordering is not defined over typed terms, type arguments are replaced by equivalent term arguments before comparison. Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be an arbitrary typed applicative signature. Let $\mathcal{V}_{\mathsf{ty}}$ be a set of type variables and $\mathcal{V}$ a set of typed term variables. Let Σ^{u} be an untyped applicative signature such that $\Sigma^{\mathsf{u}} = \Sigma$, and $\mathcal{V}^{\mathsf{u}}$ a set of untyped variables such that $\mathcal{V}^{\mathsf{u}} = \mathcal{V} \cup \{x_\alpha \,|\, \alpha \in \mathcal{V}_{\mathsf{ty}}\} \cup \{x_t \,|\, t \in \mathcal{T}_\Sigma(\mathcal{V})\}$ The translation $(\![\ ]\!)$ translates members of $\mathcal{T}ya_{\Sigma_{\mathsf{ty}}}(\mathcal{V}_{\mathsf{ty}})$ and $\mathcal{T}_\Sigma(\mathcal{V})$ to members of $\mathcal{T}_{\Sigma^{\mathsf{u}}}(\mathcal{V}^{\mathsf{u}})$. Before providing the translation, I define precisely the subterms that require replacing by variables.

Definition 32 (Type-1 term)**.** Consider a term t of the form $\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s}$. If there exists a position p such that $t|_p$ is a term variable, then t is a type-1 term.

Definition 33 (Type-2 term)**.** A term $x\,\overline{s}_n$ where $n > 0$ is a type-2 term.

The translation from polymorphic types and terms to untyped terms with problematic subterms replaced:

- If τ is a type variable α, then $(\![\,\tau\,]\!) = x_\alpha$;
- If $\tau = \kappa\,(\![\,\overline{\tau}_n\,]\!)$, then $(\![\,\tau\,]\!) = \kappa(\overline{\tau}_n)$;
- If t is a term variable x, then $(\![\,t\,]\!) = x$;
- If t is a a type-1 or type-2 term, then $(\![\,t\,]\!) = x_t$;
- If $t = \mathsf{f}\langle\overline{\tau}_n\rangle$, then $(\![\,t\,]\!) = \mathsf{f}\,(\![\,\overline{\tau}_n\,]\!)$;
- If $t = t_1\,t_2$, then $(\![\,t\,]\!) = (\![\,t_1\,]\!)(\![\,t_2\,]\!)$;

An untyped term t weak-reduces to an untyped term t' in one step if $t = u[s]_p$ and there exists a combinator axiom $l \approx r$ and substitution σ such that $(\![\,l\,]\!)\sigma = s$ and $t' = u[(\![\,r\,]\!)\sigma]_p$. The definitions of *weak redex*, *head symbol* and other terminologies for untyped terms are the obvious ones. The aim of the order is to orient combinator equations by comparing terms based of their maximal reduction lengths. However, it is not the typed terms that are compared, but their untyped translations. To ensure that this is not problematic, the following lemma is proved.

Lemma 11. *For all term ground polymorphic terms t_1 and t_2, it is the case that* $t_1 \longrightarrow_w t_2 \iff (\![\,t_1\,]\!) \longrightarrow_w (\![\,t_2\,]\!)$.

Proof. The $\Longrightarrow$ direction is proved by induction on t_1. If the reduction occurs at ϵ, then t_1 is of the form $\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}_n\rangle\,\overline{s}_n$. I prove that the lemma holds if $\mathcal{C}_{\mathsf{any}} = \mathsf{S}$. The other cases are similar. If $t_1 = \mathsf{S}\langle\tau_1,\tau_2,\tau_3\rangle\,\overline{s}_n$, then $(\![\,t_1\,]\!) = \mathsf{S}\,\tau_1\,\tau_2\,\tau_3\,(\![\,s_1\,]\!)(\![\,s_2\,]\!)(\![\,s_3\,]\!)\,(\![\,s_4\ldots s_n\,]\!) \longrightarrow_w (\![\,s_1\,]\!)(\![\,s_3\,]\!)((\![\,s_2\,]\!)(\![\,s_3\,]\!))(\![\,s_4\ldots s_n\,]\!) = (\![\,s_1\,s_3\,(s_2\,s_3)\;\;s_4\ldots s_n\,]\!) = (\![\,t_2\,]\!)$. Now assume that the reduction does not occur at ϵ. In this case, $t_1 = \xi\,\overline{s}_n$, $s_i \longrightarrow_w s_i'$ and $t_2 = \xi\,s_1\ldots s_{i-1}\,s_i'\,s_{i+1}\ldots s_n$. By the induction hypothesis, $(\![\,s_i\,]\!) \longrightarrow_w (\![\,s_i'\,]\!)$. Thus, $(\![\,t_1\,]\!) = (\![\,\xi\,]\!)(\![\,\overline{s}_n\,]\!) \longrightarrow_w (\![\,\xi\,]\!)(\![\,\overline{s}_{i-1}\,]\!)(\![\,s_i'\,]\!)(\![\,s_{i+1}\ldots s_n\,]\!) = (\![\,t_2\,]\!)$.

The $\Longleftarrow$ direction can be proved in a nearly identical manner. □

Corollary 1. *A straightforward corollary of the above lemma is that for all term-ground polymorphic terms t,* $\|t\| = \|(\![\,t\,]\!)\|$.

The combinator orienting Knuth-Bendix order (approach 1) $>_{\mathsf{ski1}}$ is defined as follows. For terms t and s, let $t' = (\![\,t\,]\!)$ and $s' = (\![\,s\,]\!)$. Then $t >_{\mathsf{ski1}} s$ if $vars_\#(s') \subseteq vars_\#(t')$ and:

R1 $\|t'\| > \|s'\|$ or,
R2 $\|t'\| = \|s'\|$ and $t' >_{\mathsf{hb}} s'$.

Approach 2

Using approach 1, terms $t = y\,\mathsf{a}$ and $s = y\,\mathsf{b}$ are incomparable. Both are type-2 terms and therefore $(\!|\, t \,|\!) = x_t$ and $(\!|\, s \,|\!) = x_s$. The variable condition obviously fails to hold between x_t and x_s. Therefore, I consider another approach which does not replace subterms with fresh variables. I introduce a new translation $[\![\;\;]\!]$ from $\mathcal{T}_\Sigma(\mathcal{V})$ to untyped terms that merely replaces type arguments with equivalent term arguments and does not affect term arguments at all. The simpler translation comes at the cost of a more complex variable condition. Before the revised variable definition can be provided, some further terminology requires introduction.

Definition 34 (Variable-like, V-like)**.** A non-ground term of the form $x\,\overline{r}_n$ (n can be 0) or $\mathcal{C}_{\mathsf{any}}\,\overline{r}$ is known as as *variable-like* or *v-like*.

Definition 35 ($safe_{ground}$)**.** For untyped terms t_1 and t_2, $safe_{ground}(t_1, t_2)$ holds if t_1 and t_2 are ground, both have non-combinator heads and $\|t_1\| \geq \|t_2\|$

Definition 36 (*safe*)**.** Let t_1 and t_2 be untyped terms. The predicate $safe(t_1, t_2)$ is defined inductively:

- If $safe_{ground}(t_1, t_2)$ then $safe(t_1, t_2)$;
- If t_1 and t_2 are the same variable, then $safe(t_1, t_2)$;
- If t_1 and t_2 are the same combinator, then $safe(t_1, t_2)$;
- If $t_1 = s_1\,s_2$, $t_2 = r_1\,r_2$, $safe(s_1, r_1)$ and $safe(s_2, r_2)$, then $safe(t_1, t_2)$.

Example 9. We have $safe(x\,\mathsf{a}, x\,\mathsf{b})$ since $safe(x, x)$ and $safe_{ground}(\mathsf{a}, \mathsf{b})$. Likewise, we have $safe(x\,(\mathsf{B}\,y)\,\mathsf{h}, x\,(\mathsf{B}\,y)\,\mathsf{g})$. However, $safe(\mathsf{C}\,\mathsf{a}\,y, \mathsf{C}\,y\,\mathsf{a})$ does not hold since $safe(\mathsf{a}, y)$ does not. Similarly, $safe(\mathsf{C}\,\mathsf{b}, \mathsf{S}\,\mathsf{b})$ does not hold as $safe(\mathsf{C}, \mathsf{S})$ does not.

Remark 11. I do not claim that *safe* is the most general syntactic condition that can be placed on terms to ensure stability. It appears to be a natural definition, but it is possible that broader conditions may exist.

Lemma 12. *For an untyped term s, $safe(s, s)$.*

Proof. The proof is by induction on $|s|$. If $s = x$ or $s = \mathcal{C}_{\mathsf{any}}$ the lemma holds by definition. If $s = \mathsf{f}$, then $safe_{ground}(\mathsf{f}, \mathsf{f})$ and thus $safe(s, s)$. Finally, if $s = s_1\,s_2$, the induction hypothesis provides $safe(s_1, s_1)$ and $safe(s_2, s_2)$ and thus $safe(s, s)$. □

Definition 37 (Top-level)**.** In the term $s = u\langle t \rangle_p$, t is a *top-level* subterm of s, if for every proper prefix p' of p, $head(s|_{p'})$ is not a variable or fully applied combinator.

Example 10. The top-level subterms of the term $\mathsf{f}\,(x\,\mathsf{a}\,\mathsf{b})$ are the term itself and $x\,\mathsf{a}\,\mathsf{b}$. All subterms of the term $\mathsf{C}\,(\mathsf{g}\,\mathsf{a})\,(\mathsf{g}\,\mathsf{c})$ are top-level, whilst the only top-level subterm of the term $\mathsf{S}\,\mathsf{a}\,\mathsf{b}\,y$ is the term itself.

Variable Condition: Let t and s be untyped terms and let $s = u\langle\overline{s}_n\rangle$, where $\{s_1, \ldots, s_n\}$ is the multiset of all top-level, v-like, subterms in s. Then $var\_cond(t, s)$ holds if $t = u'\langle\overline{v}_n\rangle$, for some green context u' and top-level subterms $\overline{v}$, and both the following hold:

- $\|u'\| \geq \|u\|$;
- For $1 \leq i \leq n$, *safe*(v_i, s_i).

Example 11. If $t = \mathsf{f}\, y\, (x\, \mathsf{a})$ and $s = \mathsf{g}\, (x\, \mathsf{b})$, then $var\_cond(t, s)$ holds. In this case, $t = \mathsf{f}\, y\, \langle x\, \mathsf{a}\rangle$, $s = \mathsf{g}\, \langle x\, \mathsf{b}\rangle$, $\|\mathsf{f}\, y\| \geq \|\mathsf{g}\|$ and *safe*$(x\, \mathsf{b}, x\, \mathsf{a})$. However, the variable condition does not hold in either direction if $t = \mathsf{f}\, y\, (x\, \mathsf{a})$ and $s = \mathsf{g}\, (x\, (\mathsf{I}\, \mathsf{b}))$ since *safe*$(x\, \mathsf{a}, x\, (\mathsf{I}\, \mathsf{b}))$ is not fulfilled.

I now define the combinator orienting Knuth-Bendix order (approach 2) $>_{\mathsf{ski}}$. For terms t and s, let $t' = [\![\, t\,]\!]$ and $s' = [\![\, s'\,]\!]$. Then $t >_{\mathsf{ski}} s$ if $var\_cond(t', s')$ and:

R1 $\|t'\| > \|s'\|$ or,
R2 $\|t'\| = \|s'\|$ and $t' >_{\mathsf{hb}} s'$.

Lemma 13. *For all ground instances of combinator axioms $l \approx r$, we have $l >_{\mathsf{ski}} r$.*

Proof. For all ground instances of the axioms $l \approx r$ we have $\|[\![\, l\,]\!]\| > \|[\![\, r\,]\!]\|$ via Corollary 1. The lemma follows by an application of R1. □

It should be noted that for non-ground instances of an axiom $l \approx r$, we do **not** necessarily have $l >_{\mathsf{ski}} r$ since l and r may be incomparable. This is no problem since the definition of $>_{\mathsf{ski}}$ can easily be amended to have $l >_{\mathsf{ski}} r$ by definition if $l \approx r$ is an instance of an axiom. Lemma 13 ensures that stability under substitution would not be affected by such an amendment.

3.3 | Properties

Various properties of the order $>_{\mathsf{ski}}$ are proved here. The proofs can easily be modified to hold for the less powerful $>_{\mathsf{ski1}}$ ordering. In general, for an ordering to parameterise a superposition calculus, it needs to be a *simplification* ordering [88]. That is, superposition is parameterised by a binary relation that is irreflexive, transitive, well-founded, total and compatible with contexts on ground terms. The only requirement of the relation on non-ground terms is that it is stable under grounding substitutions. Compatibility with contexts can be relaxed at the cost of extra inferences [19, 30, 40]. In this section, I prove that the $>_{\mathsf{ski}}$ ordering possesses most of the above mentioned properties. A desirable property to have in my case is coincidence with first-order KBO, since without this, a superposition calculus parameterised by $>_{\mathsf{ski}}$ would not behave on first-order problems as standard first-order superposition would.

Theorem 3 (Irreflexivity)**.** *For all terms s, it is not the case that $s >_{\mathsf{ski}} s$.*

Proof. Let $s' = [\![s]\!]$. It is obvious that $\|s'\| = \|s'\|$. Therefore $s >_{\mathsf{ski}} s$ can only be derived by rule R2. However, this is precluded by the irreflexivity of $>_{\mathsf{hb}}$. □

Theorem 4 (Transitivity)**.** *For ground terms s, t and u, if $s >_{\mathsf{ski}} t$ and $t >_{\mathsf{ski}} u$ then $s >_{\mathsf{ski}} u$.*

Proof. If $\|s'\| > \|t'\|$ or $\|t'\| > \|u'\|$ then $\|s'\| > \|u'\|$ and $s >_{\mathsf{ski}} u$ follows by an application of rule R1. Therefore, suppose that $\|s'\| = \|t'\| = \|u'\|$. Then it must be the case that $s' >_{\mathsf{hb}} t'$ and $t' >_{\mathsf{hb}} u'$. It follows from the transitivity of $>_{\mathsf{hb}}$ that $s' >_{\mathsf{hb}} u'$ and thus $s >_{\mathsf{ski}} u$. □

Theorem 5 (Ground Totality)**.** *Let s and t be ground terms that are not syntactically equal. Then either $s >_{\mathsf{ski}} t$ or $t >_{\mathsf{ski}} s$.*

Proof. Let $s' = [\![s]\!]$ and $t' = [\![t]\!]$. If $\|s'\| \neq \|t'\|$ then by R1 either $s >_{\mathsf{ski}} t$ or $t >_{\mathsf{ski}} s$. Otherwise, s' and t' are compared using $>_{\mathsf{hb}}$ and either $t' >_{\mathsf{hb}} s'$ or $s' >_{\mathsf{hb}} t'$ holds by the ground totality of $>_{\mathsf{hb}}$ and the injectivity of $[\![\ ]\!]$. □

Theorem 6 (Subterm Property for Ground Terms)**.** *If t and s are ground and t is a proper subterm of s then $s >_{\mathsf{ski}} t$.*

Proof. Let $s' = [\![s]\!]$ and $t' = [\![t]\!]$. Since t is a subterm of s, t' is a subterm of s' and $\|s'\| \geq \|t'\|$ because any weak reduction in t' is also a weak reduction in s'. If $\|s'\| > \|t'\|$, the theorem follows by an application of R1. Otherwise s' and t' are compared using $>_{\mathsf{hb}}$ and $s' >_{\mathsf{hb}} t'$ holds by the subterm property of $>_{\mathsf{hb}}$. Thus $s >_{\mathsf{ski}} t$. □

Next, a series of lemmas are proved that are utilised in the proof of the ordering's compatibility with contexts and stability under substitution. I prove two monotonicity properties Theorems 7 and 8. Both hold for non-ground terms, but to show this, it is required to show that the variable condition holds between terms $u[t]$ and $u[s]$ for t and s such that $t >_{\mathsf{ski}} s$. To avoid this complication, I prove the Lemmas for ground terms which suffices for my purposes. To avoid clutter, assume that terms mentioned in the statement of Lemmas 14 - 26 and Corollary 3 are all untyped, formed by translating polymorphic terms.

Lemma 14. $\|\zeta\, \overline{s}_n\| = \sum_{i=1}^{n} \|s_i\|$ *if ζ is not a fully applied combinator.*

Proof. Trivial. □

Lemma 15. *Let $\overline{s}_n$ be terms such that for each s_i, $head(s_i) \notin \{\mathsf{I}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{S}\}$. Let $\overline{r}_n$ be terms with the same property. Moreover, let $\|s_i\| \geq \|r_i\|$ for $1 \leq i \leq n$. Let $s = u[\overline{s}_n]$ and $r = u[\overline{r}_n]$ where each s_i and r_i is at position p_i in s and r. If the F_∞ redex in s is within s_i for some i, then the F_∞ redex in r is within r_i unless r_i is in normal form.*

Proof. Proof is by induction on $|s| + |r|$. If u has a hole at head position, then $s = \mathsf{f}\,\overline{w}_m\,\overline{s'}_{m''}$ and $r = \mathsf{g}\,\overline{v}_{m'}\,\overline{r'}_{m''}$ where $s_j = \mathsf{f}\,\overline{w}_m$ and $r_j = \mathsf{g}\,\overline{v}_{m'}$ for some $j \in \{1, \ldots, n\}$. Without loss of generality, let $j = 1$. Assume that the F_∞ redex of s is in s_1. Further, assume that $\|r_1\| > 0$. Then, for some $i \in \{1 \ldots m'\}$, it must be the case that $\|v_i\| > 0$. Let j be the smallest index such that $\|v_j\| > 0$. Then by the definition of F_∞, $F_\infty(r) = \mathsf{g}\,v_1 \ldots v_{j-1}\,F_\infty(v_j)\,v_{j+1} \ldots v_{m'}\,\overline{r'}$ and the F_∞ redex of r is in r_1.

Suppose that the F_∞ redex of s is not in s_1. This can only be the case if $\|s_1\| = 0$ in which case $\|r_1\| = 0$ as well. In this case, by the definition of F_∞, $F_\infty(s) = \mathsf{f}\,\overline{w}_m\,s'_1 \ldots s'_{i-1}\,F_\infty(s'_i)\,s'_{i+1} \ldots s'_{m''}$ where $\|s'_j\| = 0$ for $1 \leq j < i$. Without loss of generality, assume that the F_∞ redex of s'_i occurs inside s_i. Then r_i must be a subterm of r'_i. Assume that $\|r_i\| > 0$ and thus $\|r'_i\| > 0$. Since for all i, s'_i and r'_i only differ at positions where one contains a s_j and the other contains a r_j and $\|s_i\| \geq \|r_i\|$ for $1 \leq i \leq m''$, we have that $\|s'_j\| = 0$ implies $\|r'_j\| = 0$. Thus, using the definition of F_∞, $F_\infty(s') = \mathsf{g}\,\overline{v}_{m'}\,r'_1 \ldots r'_{i-1}\,F_\infty(r'_i)\,r'_{i+1} \ldots r'_{m''}$. The induction hypothesis can be applied to s'_i and r'_i to conclude that the F_∞ redex of r'_i occurs inside r_i. The lemma follows immediately.

If u does not have a hole at its head, then $s = \zeta\,\overline{w}_m$ and $r = \zeta\,\overline{v}_m$ where ζ is not a fully applied combinator other than K (if it was, the F_∞ redex would be at the head).

If ζ is not a combinator, the proof follows by a similar induction to above. Therefore, assume that $\zeta = \mathsf{K}$. It must be the case that $\|s_2\| > 0$ otherwise the F_∞ redex in s would be at the head and not within a s_i. By the definition of F_∞, $F_\infty(s) = \mathsf{K}\,w_1\,F_\infty(w_2)\,w_3 \ldots w_n$. Let the F_∞ redex of w_2 occur inside s_j. Then r_j is a subterm of v_2. If $\|r_j\| > 0$ then $\|v_2\| > 0$ and $F_\infty(r) = \mathsf{K}\,v_1\,F_\infty(v_2)\,v_3 \ldots v_n$. By the induction hypothesis, the F_∞ redex of v_2 occurs in r_j. □

Lemma 16. *Let $\overline{s}_n$ be terms such that for $1 \leq i \leq n$, $head(s_i) \notin \{\mathsf{I}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{S}\}$. Then for all contexts $u[\,]_n$, if $u[\overline{s}_n] \longrightarrow_w u'$ then either:*

1. *$\exists i.\, u' = u[s_1, \ldots, \hat{s}_i, \ldots, s_n]$ where $s_i \longrightarrow_w \hat{s}_i$ or*
2. *$u' = \hat{u}\{x_1 \to s_1, \ldots, x_n \to s_n\}$ where $u[x_1, \ldots, x_n] \longrightarrow_w \hat{u}$*

Proof. Let $s = u[\overline{s}_n]$ and let $p_1, \ldots p_n$ be the positions of $\overline{s}_n$ in s. Since s is reducible, there must exist a p such that $s|_p$ is a redex.

If $p > p_i$ for some i, there exists a $p' \neq \epsilon$ such that $p = p_i p'$. Then, $s|_{p_i} = s_i[\mathcal{C}_{\mathsf{any}}\,\overline{r}_n]_{p'} \longrightarrow_w s_i[(\mathcal{C}_{\mathsf{any}}\,\overline{r}_n)\downarrow^w]_{p'}$. Let $\hat{s}_i = s_i[(\mathcal{C}_{\mathsf{any}}\,\overline{r}_n)\downarrow^w]_{p'}$. We thus have that $s_i \longrightarrow_w \hat{s}_i$ and thus $u[s_1, \ldots, s_i, \ldots, s_n] \longrightarrow_w u[s_1, \ldots, \hat{s}_i, \ldots, s_n]$.

It cannot be the case that $p = p_i$ for any i because $head(s_i)$ is not a combinator for any s_i. In the case where $p < p_i$ or $p \parallel p_i$ for all i, we have that $u[\overline{s}_n] = (u[\overline{x}_n])\sigma$ and $u[\overline{x}_n]|_p$ is a redex where $\sigma = \{\overline{x}_n \to \overline{s}_n\}$. Let $\hat{u}$ be formed from $u[\overline{x}_n]$ by reducing its

redex at p. Then:

$$\begin{aligned} s = u[\overline{s}_n] &= (u[\overline{x}_n])\sigma \\ &\longrightarrow_w \hat{u}\sigma \\ &= \hat{u}\{x_1 \to s_1 \ldots x_n \to s_n\} \end{aligned}$$

□

Lemma 17. *Let $\overline{s}_n$ be terms such that for each s_i, $head(s_i) \notin \{\mathsf{I}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{S}\}$. Let $\overline{r}_n$ be terms with the same property. Then:*

1. *If $\|s_i\| = \|r_i\|$ for $1 \leq i \leq n$, then $\|u[\overline{s}_n]\| = \|u[\overline{r}_n]\|$ for all n holed contexts u.*
2. *If $\|s_j\| > \|r_j\|$ for some $j \in \{1, \ldots, n\}$ and $\|s_i\| \geq \|r_i\|$ for $i \neq j$, then $\|u[\overline{s}_n]\| > \|u[\overline{r}_n]\|$ for all n holed contexts u.*

Proof. Let $p_1, \ldots, p_n$ be the positions of the holes in u and let $s = u[\overline{s}_n]$ and $r = u[\overline{r}_n]$. Proof is by induction on $\|s\| + \|r\|$. I prove part (1) first:

Assume that $\|u[\overline{s}_n]\| = 0$. Then $\|s_i\| = 0$ for $1 \leq i \leq n$. Now assume that $\|u[\overline{r}_n]\| \neq 0$. Then there must exist some position p such that $r|_p$ is a redex. We have that $p \neq p_i$ for all p_i as $head(r_i) \notin \{\mathsf{I}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{S}\}$. Assume $p > p_i$ for some p_i. But then, $\|r_i\| > 0$ which contradicts the fact that $\|s_i\| = \|r_i\|$ for all i. Therefore, for all p_i either $p < p_i$ or $p \parallel p_i$. But then, if $r|_p$ is a redex, so must $s|_p$ be, contradicting the fact that $\|u[\overline{s}_n]\| = 0$. Thus, I conclude that $\|u[\overline{r}_n]\| = 0$.

Assume that $\|u[\overline{s}_n]\| > 0$. Let $u' = F_\infty(s)$. By Lemma 16 either $u' = u[s_1, \ldots, \hat{s_i}, \ldots, s_n]$ where $s_i \longrightarrow_w \hat{s_i}$ for $1 \leq i \leq n$ or $u' = \hat{u}\{\overline{x}_n \to \overline{s}_n\}$ where $u[\overline{x}_n] \longrightarrow_w \hat{u}$. In the first case, by Lemma 15 and $\|s_i\| = \|r_i\|$ we have $F_\infty(r) = u'' = u[r_1, \ldots, \hat{r_i}, \ldots, r_n]$ where $r_i \longrightarrow_w \hat{r_i}$. By the induction hypothesis $\|u'\| = \|u''\|$ and thus $\|s\| = \|r\|$. In the second case, $F_\infty(r) = u'' = \hat{u}\{\overline{x}_n \to \overline{r}_n\}$ where $u[\overline{x}_n] \longrightarrow_w \hat{u}$. Again, the induction hypothesis can be used to show $\|u'\| = \|u''\|$ and the theorem follows.

I now prove part (2); $\|u[\overline{s}_n]\|$ must be greater than 0. Again, let $u' = F_\infty(s)$ and $u'' = F_\infty(r)$. If $u' = u[s_1, \ldots, \hat{s_i}, \ldots, s_n]$ and $\|r_i\| \neq 0$, then by Lemma 15 $u'' = u[r_1, \ldots, \hat{r_i}, \ldots, r_n]$ where $r_i \longrightarrow_w \hat{r_i}$ and the lemma follows by the induction hypothesis.

If $\|r_i\| = 0$, consider terms u' and r. If $\|\hat{s_i}\| > 0$ or $\|s_j\| > \|r_j\|$ for some $j \neq i$, then the induction hypothesis can be used to show $\|u'\| > \|r\|$ and therefore $\|s\| = \|u'\| + 1 > \|r\|$. Otherwise, $\|s_j\| = \|r_j\|$ for all $j \neq i$ and $\|\hat{s_i}\| = 0 = \|r_i\|$. Part 1 of this lemma can be used to show that $\|u'\| = \|r\|$ and thus $\|s\| = \|u'\| + 1 > \|r\|$. If $u' = \hat{u}\{\overline{x}_n \to \overline{s}_n\}$, then $u'' = \hat{u}\{\overline{x}_n \to \overline{r}_n\}$ and the lemma follows by the induction hypothesis. □

Theorem 7 (Compatibility with Contexts)**.** *For ground terms s and t, such that $head(s)$, $head(t) \notin \{\mathsf{I}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{S}\}$, and $s >_{\mathsf{ski}} t$, then $u[s] >_{\mathsf{ski}} u[t]$ for all ground contexts $u[\,]$.*

Proof. Let $s' = [\![s]\!]$, $t' = [\![t]\!]$ and $u' = [\![u]\!]$. By Lemma 17 Part 2, we have that if $\|s'\| > \|t'\|$, then $\|u'[s']\| > \|u'[t']\|$. Thus, if $s >_{\mathsf{ski}} t$ was derived by R1, $u[s] >_{\mathsf{ski}} u[t]$

follows by R1. Otherwise, $s >_{\mathsf{ski}} t$ is derived by R2 and $\|s'\| = \|t'\|$. By Lemma 17 Part 1, $\|u'[s']\| = \|u'[t']\|$ follows. Thus, $u'[s']$ is compared with $u'[t']$ by R2 and $u[s] >_{\mathsf{ski}} u[t]$ by the compatibility with contexts of $>_{\mathsf{hb}}$. □

Corollary 2 (Compatibility with Arguments)**.** *If $s >_{\mathsf{ski}} t$ and $head(s)$ and $head(t)$ are not combinators then $s\,u >_{\mathsf{ski}} t\,u$.*

Proof. This is just a special case of Theorem 7. □

Lemma 18. $\|s\| > \|t\| \implies \|u\langle\langle s\rangle\rangle\| > \|u\langle\langle t\rangle\rangle\|$ *and* $\|s\| = \|t\| \implies \|u\langle\langle s\rangle\rangle\| = \|u\langle\langle t\rangle\rangle\|$.

Proof. Proceed by induction on the size of the context u. If u is the empty context, both parts of the theorem hold trivially.

The inductive case is proved for the first implication of the lemma first. If u is not the empty context, $u\langle\langle s\rangle\rangle$ is of the form $u'\langle\langle \zeta\, t_1 \ldots t_{i-1}, s, t_{i+1} \ldots t_n\rangle\rangle$. By the definition of a stable subterm ζ cannot be a fully applied combinator and thus by Lemma 14 we have:

$$\begin{aligned} \|\zeta\, t_1 \ldots t_{i-1}, s, t_{i+1} \ldots t_n\| &= \sum_{\substack{j=1 \\ j\neq i}}^{n} \|t_j\| + \|s\| \\ &> \sum_{\substack{j=1 \\ j\neq i}}^{n} \|t_j\| + \|t\| \\ &= \|\zeta\, t_1 \ldots t_{i-1}, t, t_{i+1} \ldots t_n\| \end{aligned}$$

If ζ is not a combinator, then $\|u'\langle\langle \zeta\, t_1 \ldots t_{i-1}, s, t_{i+1} \ldots t_n\rangle\rangle\| > \|u'\langle\langle \zeta\ t_1 \ldots t_{i-1}, t, t_{i+1} \ldots t_n\rangle\rangle\|$ follows from Lemma 17 Part 2. Otherwise, ζ is a partially applied combinator and u' is a smaller stable context than u. The induction hypothesis can be used to conclude that $\|u'\langle\langle \zeta\, t_1 \ldots t_{i-1}, s, t_{i+1} \ldots t_n\rangle\rangle\| > \|u'\langle\langle \zeta\, t_1 \ldots t_{i-1}, t, t_{i+1} \ldots t_n\rangle\rangle\|$ and thus that $\|u\langle\langle s\rangle\rangle\| > \|u\langle\langle t\rangle\rangle\|$. The proof of the inductive case for the second implication of the lemma is almost identical. □

Theorem 8 (Compatibility with Stable Contexts)**.** *For all stable ground contexts $u\langle\langle\ \rangle\rangle$ and ground terms s and t, if $s >_{\mathsf{ski}} t$ then $u\langle\langle s\rangle\rangle >_{\mathsf{ski}} u\langle\langle t\rangle\rangle$.*

Proof. If $\|s\| > \|t\|$ then by Lemma 18, $\|u\langle\langle s\rangle\rangle\| > \|u\langle\langle t\rangle\rangle\|$ holds and then by an application of R1 we have $u\langle\langle s\rangle\rangle >_{\mathsf{ski}} u\langle\langle t\rangle\rangle$. Otherwise, if $\|s\| = \|t\|$, then by Lemma 18 we have that $\|u\langle\langle s\rangle\rangle\| = \|u\langle\langle t\rangle\rangle\|$. Thus $u\langle\langle s\rangle\rangle$ and $u\langle\langle t\rangle\rangle$ are compared using $>_{\mathsf{hb}}$. By the compatibility with contexts of $>_{\mathsf{hb}}$, $[\![\, u\langle\langle s\rangle\rangle \,]\!] >_{\mathsf{hb}} [\![\, u\langle\langle t\rangle\rangle \,]\!]$ holds and then by an application of R2 $u\langle\langle s\rangle\rangle >_{\mathsf{ski}} u\langle\langle t\rangle\rangle$ is true. □

Rather than proving stability under all substitutions, I prove the slightly restricted property of stability under grounding substitutions. This property is simpler to prove since it does not require proving that the variable condition holds after the application of

a substitution. Furthermore, it suffices for my purpose of using the order to parameterise superposition.

Lemma 19. *For a single hole context $u\langle\,\rangle$ such that the hole does not occur below a fully applied combinator and any term t, $\|u\langle t\rangle\| = \|u\langle\,\rangle\| + \|t\|$.*

Proof. Proceed by induction on the size of u. If u is the empty context the theorem follows trivially. Therefore, assume that $u = \zeta\, t_1 \ldots t_{i-1}\, u'\langle\,\rangle\, t_{i+1} \ldots t_n$ where ζ is not a fully applied combinator. By Lemma 14, $\|u\langle\,\rangle\| = \sum_{i=1}^{n} \|t_i\| + \|u'\langle\,\rangle\|$. Because u' is a smaller context than u, the induction hypothesis can be used to show $\|u'\langle t\rangle\| = \|u'\langle\,\rangle\| + \|t\|$. Thus:

$$\begin{aligned} \|u\langle t\rangle\| &= \sum_{i=1}^{n} \|t_i\| + \|u'\langle t\rangle\| \\ &= \sum_{i=1}^{n} \|t_i\| + \|u'\langle\,\rangle\| + \|t\| \\ &= \|u\langle\,\rangle\| + \|t\| \end{aligned}$$

proving the theorem. □

Lemma 20. *If safe(t_1, t_2), then $t_1 = \mathcal{C}_{\mathsf{any}}\, \overline{s}_n$ if and only if $t_2 = \mathcal{C}_{\mathsf{any}}\, \overline{r}_n$. Moreover, for $1 \leq i \leq n$, safe$(s_i, r_i,)$*

Proof. I prove the $\Longrightarrow$ direction by induction on n. If $n = 0$, then t_1 is a combinator and for *safe*(t_1, t_2) to hold, t_2 must be the same combinator.

Assume that $t_1 = \mathcal{C}_{\mathsf{any}}\, \overline{s}_{n>0}$. Let $v_1 = \mathcal{C}_{\mathsf{any}}\, \overline{s}_{n-1}$ and $v_2 = s_n$. Since *safe*(t_1, t_2), it must be that $t_2 = w_1\, w_2$ and *safe*(v_1, w_1), *safe*(v_2, w_2). By the induction hypothesis, $w_1 = \mathcal{C}_{\mathsf{any}}\, \overline{r}_{n-1}$ with *safe*(s_i, r_i) for $1 \leq i \leq n-1$. Thus, by setting $r_n = w_2$, $t_2 = \mathcal{C}_{\mathsf{any}}\, \overline{r}_n$ and *safe*(s_i, r_i) for $1 \leq i \leq n$.

The argument works the same in reverse proving the $\Longleftarrow$ direction of the lemma. □

Lemma 21. *Let $t_1 = \mathcal{C}_{\mathsf{any}}\, \overline{s}_n$, $t_2 = \mathcal{C}_{\mathsf{any}}\, \overline{r}_n$ and safe(t_1, t_2). Then safe$((t_1)\downarrow^w, (t_2)\downarrow^w)$.*

Proof. By the previous lemma, we have *safe*(s_i, r_i) for $1 \leq i \leq n$.

If $\mathcal{C}_{\mathsf{any}} = \mathsf{I}$, $(t_1)\downarrow^w = \overline{s}_n$ and $(t_2)\downarrow^w = \overline{r}_n$. Since *safe*$(s_i, r_i)$ for $1 \leq i \leq n$, from the definition of *safe*, *safe*$((t_1)\downarrow^w, (t_2)\downarrow^w)$ follows.

If $\mathcal{C}_{\mathsf{any}} = \mathsf{K}$, $(t_1)\downarrow^w = s_1\, s_3 \ldots s_n$ and $(t_2)\downarrow^w = r_1\, r_3 \ldots r_n$. Since *safe*$(s_1, r_1)$ and *safe*(s_i, r_i) for $3 \leq i \leq n$, *safe*$((t_1)\downarrow^w, (t_2)\downarrow^w)$ follows from the definition of *safe*.

If $\mathcal{C}_{\mathsf{any}} = \mathsf{S}$, $(t_1)\downarrow^w = s_1\, s_3\, (s_2\, s_3)\, s_4 \ldots s_n$ and $(t_2)\downarrow^w = r_1\, r_3\, (r_2\, r_3)\, r_4 \ldots r_n$. Since *safe*$(s_2, r_2)$ and *safe*(s_3, r_3), we have *safe*$(s_2\, s_3, r_2\, r_3)$. Further, we have *safe*(s_1, r_1) and *safe*(s_i, r_i) for $4 \leq i \leq n$. Thus, *safe*$((t_1)\downarrow^w, (t_2)\downarrow^w)$ follows from the definition of *safe*.

I omit the proofs of $\mathcal{C}_{\mathsf{any}} = \mathsf{B}$ and $\mathcal{C}_{\mathsf{any}} = \mathsf{C}$ as these are very similar to the cases provided. □

Lemma 22. *If* $\mathit{safe}(t_1, t_2)$ *then* $\|t_1\| \geq \|t_2\|$.

Proof. If $t_1 = x = t_2$ or $t_1 = \mathcal{C}_{\mathsf{any}} = t_2$, this is obvious. If $\mathit{safe}_{\mathit{ground}}(t_1, t_2)$, then $\|t_1\| \geq \|t_2\|$ follows by definition. Otherwise, $t_1 = s_1\, s_2$, $t_2 = r_1\, r_2$, $\mathit{safe}(s_1, r_1)$ and $\mathit{safe}(s_2, r_2)$.

If $\mathit{head}(t_1)$ is not a fully applied combinator, then neither is $\mathit{head}(t_2)$ by Lemma 20 and we proceed by induction on $|t_1| + |t_2|$. By the induction hypothesis, $\|s_1\| \geq \|r_1\|$ and $\|s_2\| \geq \|r_2\|$. Thus, using Lemma 14 we can conclude $\|t_1\| \geq \|t_2\|$.

If $\mathit{head}(t_1)$ is a fully applied combinator, then $t_1 = \mathcal{C}_{\mathsf{any}}\, \overline{w}_n$, $t_2 = \mathcal{C}_{\mathsf{any}}\, \overline{v}_n$ and $\mathit{safe}(w_i, v_i)$ by Lemma 20. We proceed by induction on $\|t_1\| + \|t_2\|$.

If $\mathcal{C}_{\mathsf{any}} = \mathsf{K}$, by the fundamental lemma of maximality, $\|t_1\| = \|w_1\, w_3 \ldots w_n\| + \|w_2\| + 1$ and $\|t_2\| = \|v_1\, v_3 \ldots v_n\| + \|v_2\| + 1$. By Lemma 21, $\mathit{safe}(w_1\, w_3 \ldots w_n, v_1\, v_3 \ldots v_n)$ holds. Using the induction hypothesis, we have that $\|w_1\, w_3 \ldots w_n\| \geq \|v_1\, v_3 \ldots v_n\|$ and $\|w_2\| \geq \|v_2\|$. Thus, $\|t_1\| \geq \|t_2\|$.

If $\mathcal{C}_{\mathsf{any}}$ is any other combinator, by the fundamental lemma of maximality, $\|t_1\| = \|(t_1)\!\downarrow^w\| + 1$ and $\|t_2\| = \|(t_2)\!\downarrow^w\| + 1$. By Lemma 21, $\mathit{safe}((t_1)\!\downarrow^w, (t_2)\!\downarrow^w)$ holds and thus the induction hypothesis can be used to show $\|(t_1)\!\downarrow^w\| \geq \|(t_2)\!\downarrow^w\|$. From this, we conclude $\|t_1\| \geq \|t_2\|$. □

Lemma 23. *Let* $t_1 = \overline{s}_n$ *and* $t_2 = \overline{r}_n$ *be terms such that* $\mathit{safe}(s_i, r_i)$ *for* $1 \leq i \leq n$ *and* $\mathit{head}(t_1)$, $\mathit{head}(t_2)$ *are not fully applied combinators. Then* $\|t_1\| > \|t_2\|$ *if and only if there exists an* $i \in \{1, \ldots, n\}$ *such that* $\|s_i\| > \|r_i\|$.

Proof. I prove the $\Longrightarrow$ direction first. By the previous lemma, we have that $\|s_i\| \geq \|r_i\|$ for $1 \leq i \leq n$. I show that it cannot be the case that $\|s_i\| = \|r_i\|$ for $1 \leq i \leq n$.

Assume that $\|s_i\| = \|r_i\|$ for $1 \leq i \leq n$. Let $s_1 = \zeta\, \overline{w}_m$ and $r_1 = \xi\, \overline{v}_{m'}$. Neither ζ nor ξ can be the head of a redex and therefore, via Lemma 14 and the assumption that $\|s_1\| = \|r_1\|$, we can conclude that $\sum_{i=1}^{m} w_i = \sum_{i=1}^{m'} v_i$. Using Lemma 14 again, we have that $\|t_1\| = \sum_{i=1}^{m} w_i + \sum_{i=2}^{n} s_i = \sum_{i=1}^{m'} v_i + \sum_{i=2}^{n} s_i = \sum_{i=1}^{m'} v_i + \sum_{i=2}^{n} r_i = \|t_2\|$ contradicting $\|t_1\| > \|t_2\|$.

The $\Longleftarrow$ direction of the proof follows a similar pattern. The complete proof can be found in Appendix A. □

Lemma 24. *Let* $t_1 = \mathcal{C}_{\mathsf{any}}\, \overline{s}_n$ *and* $t_2 = \mathcal{C}_{\mathsf{any}}\, \overline{r}_n$ *be terms such that* $\mathit{safe}(t_1, t_2)$. *Then* $\|t_1\| > \|t_2\|$ *if and only if* $\|s_i\| > \|r_i\|$ *for some* $i \in \{1, \ldots, n\}$

Proof. I prove the $\Longrightarrow$ direction first. Assume that $\|t_1\| > \|t_2\|$ holds. The proof proceeds by induction on $\|t_1\| + \|t_2\|$.

If $\mathcal{C}_{\mathsf{any}} = \mathsf{K}$, by the fundamental lemma of maximality, $\|t_1\| = \|s_1\, s_3 \ldots s_n\| + \|s_2\| + 1$ and $\|t_2\| = \|r_1\, r_3 \ldots r_n\| + \|r_2\| + 1$. Let $t_1' = s_1\, s_3 \ldots s_n$ and $t_2' = r_1\, r_3 \ldots r_n$. The following equation holds:

$$\|t_1'\| + \|s_2\| + 1 > \|t_2'\| + \|r_2\| + 1 \tag{3.1}$$

By Lemma 21, we have $\mathit{safe}(t_1', t_2')$. By Lemma 20, we have $\mathit{safe}(s_2, r_2)$. Thus, by Lemma 22, $\|t_1'\| \geq \|t_2'\|$ and $\|s_2\| \geq \|r_2\|$. For Equation 3.1 to hold, either $\|t_1'\| > \|t_2'\|$ or $\|s_2\| > \|r_2\|$ must be the case. If $\|s_2\| > \|r_2\|$, we are done. Therefore, assume that $\|s_2\| = \|r_2\|$ and $\|t_1'\| > \|t_2'\|$.

- If $\mathit{head}(t_1')$ and $\mathit{head}(t_2')$ are not fully applied combinators, by Lemma 23, either $\|s_1\| > \|r_1\|$ or $\|s_i\| > \|r_i\|$ for $i \in \{3 \ldots n\}$ and we are done.
- If $\mathit{head}(t_1')$ and $\mathit{head}(t_2')$ are fully applied combinators, then $t_1' = \mathcal{C}_{\mathsf{any}}\, \overline{w}_m\, s_3 \ldots s_n$ and $t_2' = \mathcal{C}_{\mathsf{any}}\, \overline{v}_m\, r_3 \ldots r_n$. Via the induction hypothesis, we have that $\|w_i\| > \|v_i\|$ for some $i \in \{1, \ldots, m\}$ or $\|s_i\| > \|r_i\|$ for some $i \in \{3, \ldots, n\}$. In the latter case, we are done. Therefore, assume $\|w_i\| > \|v_i\|$ for some $i \in \{1, \ldots, m\}$. If $\mathcal{C}_{\mathsf{any}}\, \overline{w}_m$ forms a weak redex, the $\Longleftarrow$ direction of the induction hypothesis gives us $\|s_1\| = \|\mathcal{C}_{\mathsf{any}}\, \overline{w}_m\| > \|\mathcal{C}_{\mathsf{any}}\, \overline{v}_m\| = \|r_1\|$. Otherwise, Lemma 23 provides the same result.

The $\mathcal{C}_{\mathsf{any}} \neq \mathsf{K}$ case can be found in the complete proof in Appendix A as can the $\Longleftarrow$ direction of the proof. □

Combining the above two lemmas provides the following corollary.

Corollary 3. *Let $t_1 = \overline{s}_n$ and $t_2 = \overline{r}_n$ be terms such that $\mathit{safe}(s_i, r_i)$ for $1 \leq i \leq n$. Then $\|t_1\| > \|t_2\|$ if and only if $\|s_i\| > \|r_i\|$ for some $i \in \{1, \ldots, n\}$.*

Lemma 25. *For terms t_1 and t_2 such that $\mathit{safe}(t_1, t_2)$, then $\mathit{safe}(t_1\sigma, t_2\sigma)$ for all substitutions σ.*

Proof. If $t_1 = \mathcal{C}_{\mathsf{any}} = t_2$, the lemma is trivial. If $t_1 = x = t_2$, then $\mathit{safe}(t_1\sigma, t_2\sigma)$ follows from Lemma 12. If $\mathit{safe}_{\mathit{ground}}(t_1, t_2)$ then t_1 and t_2 are ground and thus $t_1\sigma = t_1$ and $t_2\sigma = t_2$ and so $\mathit{safe}(t_1\sigma, t_2\sigma)$.

Otherwise, $t_1 = s_1\, s_2$, $t_2 = r_1\, r_2$, $\mathit{safe}(s_1, r_1)$ and $\mathit{safe}(s_2, r_2)$. We proceed by induction on $|t_1| + |t_2|$. Using the induction hypothesis, we have that $\mathit{safe}(s_1\sigma, r_1\sigma)$ and $\mathit{safe}(s_2\sigma, r_2\sigma)$. Thus, $\mathit{safe}(t_1\sigma, t_2\sigma)$. □

Lemma 26. *For terms t_1 and t_2 such that $\mathit{safe}(t_1, t_2)$ and $\|t_1\| > \|t_2\|$, then $\|t_1\sigma\| > \|t_2\sigma\|$ for all substitutions σ.*

Proof. If $\mathit{safe}_{\mathit{ground}}(t_1, t_2)$, then t_1 and t_2 are ground and $\|t_1\| > \|t_2\|$ implies $\|t_1\sigma\| > \|t_2\sigma\|$. Otherwise, $t_1 = s_1\, s_2$, $t_2 = r_1\, r_2$, $\mathit{safe}(s_1, r_1)$ and $\mathit{safe}(s_2, r_2)$.

I proceed by induction on $|t_1| + |t_2|$.

By Corollary 3, either $\|s_1\| > \|r_1\|$ or $\|s_2\| > \|r_2\|$. Without loss of generality, assume $\|s_2\| > \|r_2\|$. By the induction hypothesis, we have $\|s_2\sigma\| > \|r_2\sigma\|$. By Lemma 25, we have $\mathit{safe}(s_1\sigma, r_1\sigma)$ and $\mathit{safe}(s_2\sigma, r_2\sigma)$. Using Corollary 3, we conclude $\|s_1\, s_2\| > \|r_1\, r_2\|$. □

Lemma 27. *Let t be a polymorphic term and σ be a substitution. I define a new substitution ρ such that the domain of ρ is $dom(\sigma)$. Define $y\rho = [\![y\sigma]\!]$. For all terms t, $[\![t\sigma]\!] = [\![t]\!]\rho$.*

Proof. Via a straightforward induction on t. □

Theorem 9 (Stability under Substitution)**.** *If $s >_{\mathsf{ski}} t$ then $s\theta >_{\mathsf{ski}} t\theta$ for all grounding substitutions θ.*

Proof. Let $s' = [\![s]\!]$ and $t' = [\![t]\!]$. Let ρ be defined as per Lemma 27. Since $var\_cond(s', t')$, we have that $t' = u\langle \overline{r}_n \rangle$, where $\{r_1, \ldots, r_n\}$ is the set of all top-level, v-like subterms in t', and that $s' = u'\langle \overline{v}_n \rangle$ for some green context u' and top-level subterms $\overline{v}$. Further, both the following hold:

- $\|u'\| \geq \|u\|$;
- For $1 \leq i \leq n$, *safe*$(v_i, r_i,)$.

First, I show that if R1 was used to derive $s >_{\mathsf{ski}} t$ and thus $\|s'\| > \|t'\|$, then $\|s'\rho\| > \|t'\rho\|$ and thus $s\theta >_{\mathsf{ski}} t\theta$ because $[\![s\theta]\!] = s'\rho$ and $[\![t\theta]\!] = t'\rho$. Either $\|u'\| > \|u\|$ or $\|v_i\| > \|r_i\|$ for some i must hold, otherwise by an n-fold application of Lemma 19 $\|t'\| = \|s'\|$ contradicting the assumption. I need to show that $\|s'\rho\| = \|u'\rho\,\langle \overline{v}_n\rho \rangle\| > \|u\rho\,\langle \overline{r}_n\rho \rangle\| = \|t'\rho\|$.

Assume that $\|u'\| > \|u\|$. Since u is ground, we have that $\|u'\rho\| \geq \|u'\| > \|u\| = \|u\rho\|$. By Lemma 25, we have *safe*$(v_i\rho, r_i\rho)$ for $1 \leq i \leq n$, and thus by Lemma 22, $\|v_i\rho\| \geq \|r_i\rho\|$. Thus, via multiple applications of Lemma 19:

$$\begin{aligned} \|s'\rho\| &= \|u'\rho\| + \sum_{i=1}^{n} \|v_i\rho\| \\ &> \|u\| + \sum_{i=1}^{n} \|v_i\rho\| \\ &\geq \|u\| + \sum_{i=1}^{n} \|r_i\rho\| \\ &= \|t'\rho\| \end{aligned}$$

Assume that $\|u'\| = \|u\|$ and that for some j such that $1 \leq j \leq n$, $\|v_j\| > \|r_j\|$. By Lemma 26 it follows that $\|v_j\rho\| > \|r_j\rho\|$. By Lemma 22, $\|v_i\rho\| \geq \|r_i\rho\|$ for $i \neq j$. Thus, via multiple applications of Lemma 19:

$$\|s'\rho\| = \|u'\rho\| + \sum_{i=1}^{n} \|v_i\rho\|$$

$$\begin{aligned} &\geq && \|u\| + \sum_{i=1}^{n} \|v_i\rho\| \\ &> && \|u\| + \sum_{i=1}^{n} \|r_i\rho\| \\ &= && \|t'\rho\| \end{aligned}$$

On the other hand, if $\|s'\| = \|t'\|$, then R2 was used to derive $s >_{\mathsf{ski}} t$. Using Lemmas 25 and 22, we have that $\|s'\rho\| = \|u'\rho\langle\overline{v}_n\rho\rangle\| \geq \|u\rho\langle\overline{r}_n\rho\rangle\| = \|t'\rho\|$. If $\|s'\rho\| > \|t'\rho\|$, then obviously $s\theta >_{\mathsf{ski}} t\theta$. Therefore, assume that $\|s'\rho\| = \|t'\rho\|$ in which case $s'\rho$ and $r'\rho$ are compared using $>_{\mathsf{hb}}$ and $s'\rho >_{\mathsf{hb}} r'\rho$ follows from the stability under substitution of $>_{\mathsf{hb}}$. □

Theorem 10 (Well-foundedness)**.** *There exists no infinite descending chain of comparisons* $s_1 >_{\mathsf{ski}} s_2 >_{\mathsf{ski}} s_3 \cdots$.

Proof. Assume that such a chain exists. For each $s_i >_{\mathsf{ski}} s_{i+1}$ derived by R1, we have that $\|s_i\| > \|s_{i+1}\|$. For each $s_i >_{\mathsf{ski}} s_{i+1}$ derived by R2, we have that $\|s_i\| = \|s_{i+1}\|$. Therefore, the number of times $s_i >_{\mathsf{ski}} s_{i+1}$ by R1 in the infinite chain must be finite and there must exist some m such that for all $n > m$, $s_n >_{\mathsf{ski}} s_{n+1}$ by R2. Therefore, there exists an infinite sequence of $>_{\mathsf{hb}}$ comparisons $[\![s_m]\!] >_{\mathsf{hb}} [\![s_{m+1}]\!] >_{\mathsf{hb}} [\![s_{m+2}]\!] \cdots$. This contradicts the well-foundedness of $>_{\mathsf{hb}}$. □

Theorem 11 (Coincidence with First-Order KBO)**.** *Let* $>_{\mathsf{fo}}$ *be the first-oder KBO as described by Becker et al. in [15]. Assume that* $>_{\mathsf{ski}}$ *and* $>_{\mathsf{fo}}$ *are parameterised by the same precedence* $\succ$ *and that* $>_{\mathsf{fo}}$ *always compares tuples using the lexicographic extension operator. Then* $>_{\mathsf{ski}}$ *and* $>_{\mathsf{fo}}$ *always agree on first-order terms.*

Proof. Let $t' = [\![t]\!]$ and $s' = [\![s]\!]$. Since s and t are first-order, $\|s'\| = 0$ and $\|t'\| = 0$. Thus, s' and t' will always be compared by $>_{\mathsf{hb}}$. Since $>_{\mathsf{hb}}$ coincides with $>_{\mathsf{fo}}$ on first-order terms, so does $>_{\mathsf{ski}}$. □

The $>_{\mathsf{ski}}$ ordering presented here is able to compare non-ground terms that cannot be compared by any ordering used to parameterise Bentkamp et al.'s λ-superposition calculus [21]. They define terms to be β-equivalence classes. Non-ground terms are compared using a quasiorder, $\succsim$, such that $t \succsim s$ iff for all grounding substitutions θ, $t\theta \succeq s\theta$. Consider terms $t = x\ \mathsf{a}\ \mathsf{b}$ and $s = x\ \mathsf{b}\ \mathsf{a}$ and grounding substitutions $\theta_1 = x \to \lambda x\ y\ .\mathsf{f}\ y\ x$ and $\theta_2 = x \to \lambda x\ y\ .\mathsf{f}\ x\ y$. By ground totality of $\succ$ it must be the case that either $\mathsf{f}\ \mathsf{a}\ \mathsf{b} \succ \mathsf{f}\ \mathsf{b}\ \mathsf{a}$ or $\mathsf{f}\ \mathsf{b}\ \mathsf{a} \succ \mathsf{f}\ \mathsf{a}\ \mathsf{b}$. Without loss of generality assume the first. Then, neither $t \succsim s$ nor $s \succsim t$ since $t\theta_1 = \mathsf{f}\ \mathsf{b}\ \mathsf{a} \prec \mathsf{f}\ \mathsf{a}\ \mathsf{b} = s\theta_1$ and $t\theta_2 = \mathsf{f}\ \mathsf{a}\ \mathsf{b} \succ \mathsf{f}\ \mathsf{b}\ \mathsf{a} = s\theta_2$. However, *safe*$(s, t)$ (and *safe*(t, s)) hold. Thus, t and s are comparable using $>_{\mathsf{ski}}$.

Chapter 4

Combinatory Superposition

> ... it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other: Let us calculate.
>
> *Gottfried Leibniz*

In this chapter, I show how the $>_{\mathsf{ski}}$ ordering developed in the previous chapter can be used to parameterise the superposition calculus. Recall that the ordering is not compatible with arguments and not compatible with unstable contexts. As explained in Chapter 2, Bentkamp et al. have designed a set of calculi complete for applicative first-order logic in the presence of an ordering that is not compatible with arguments. Incompatibility with unstable contexts brings further issues, as is highlighted in the following example.

Example 12. Consider the unsatisfiable clause set below where $\mathsf{f}\,\mathsf{e} >_{\mathsf{ski}} \mathbf{C}\,\mathsf{c}\,\mathsf{d}$:

$$C_1 = \mathbf{C}\,\mathsf{c}\,\mathsf{d} \approx \mathsf{f}\,\mathsf{e} \qquad C_2 = \mathbf{S}\,(\mathbf{C}\,\mathsf{c}\,\mathsf{d})\,\mathsf{a}\,\mathsf{b} \not\approx \mathbf{S}\,(\mathsf{f}\,\mathsf{e})\,\mathsf{a}\,\mathsf{b}$$

The hole in the context $\mathbf{S}\,[\,]\,\mathsf{a}\,\mathsf{b}$ is at an unstable position and in this example, we have $\mathbf{S}\,(\mathbf{C}\,\mathsf{c}\,\mathsf{d})\,\mathsf{a}\,\mathsf{b} >_{\mathsf{ski}} \mathbf{S}\,(\mathsf{f}\,\mathsf{e})\,\mathsf{a}\,\mathsf{b}$ preventing a proof from being found.

The solution to the issue demonstrated in the above example, is to only superpose at positions above fully applied combinators, and rely on rewriting with the combinator axioms to provide access to subterms occurring below fully applied combinators. In the above example, a double rewrite with the $\mathbf{S}$-axiom on C_2 results in the clause $C_3 = \mathbf{C}\,\mathsf{c}\,\mathsf{d}\,\mathsf{b}\,(\mathsf{a}\,\mathsf{b}) \not\approx \mathsf{f}\,\mathsf{e}\,\mathsf{b}\,(\mathsf{a}\,\mathsf{b})$. An ArgCong inference on C_1 results in the clause $C_4 = \mathbf{C}\,\mathsf{c}\,\mathsf{d}\,x_1\,x_2 \approx \mathsf{f}\,\mathsf{e}\,x_1\,x_2$ which can be used to rewrite C_3 and achieve the desired contradiction.

However, there is an added complication. Superposition underneath variables is ruled out in clausal λ-free superposition by utilising the term order's compatibility with green contexts. As stable contexts are a subset of green contexts, I am unable to rule out the need for superposition underneath variables using the same methods as Bentkamp et al.

Instead, I provide two solutions. In one, I add an extra inference rule to those of λ-free superposition to deal with superposition underneath variables.[1] In the second, the concept of *selection* of negative literals is avoided. I explain how this allows me to avoid superposition underneath variables in the completeness proof.

4.1 | The Calculi

The calculi are closely modeled on the intensional version of the clausal λ-free superposition calculus. The extensionality axiom can be added if extensionality is required. The main difference between my calculi and clausal λ-free superposition is that superposition inferences are not allowed beneath fully applied combinators. The calculi are called *clausal combinatory superposition select* and *clausal combinatory superposition no-select*. Sometimes I drop the 'select' from the name of the calculus that utilises selection. The word 'clausal' indicates that the calculi do not support reasoning about Booleans or choice. I mentioned above that the calculi do not superpose underneath fully applied combinators. The following definition defines the subterms that the calculi *do* superpose at.

Definition 38 (First-order Subterms)**.** The set of first-order subterms are defined inductively as follows. Every term s is a first-order subterm of itself. For a term s of the form $\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s}_n$ or $\mathsf{f}\langle\overline{\tau}\rangle\,\overline{s}_n$ or $x\,\overline{s}_n$, the first-order subterms of each s_i are first-order subterms of s. The same notations used for green subterms and contexts are utilised for first-order subterms and contexts. Confusion is avoided since green subterms are never used in this chapter.

4.1.1 | Term Order

The calculi must be parameterised by a strict partial order $\succ$ that is well-founded, total on ground terms, stable under substitutions and has the subterm property and which orients all instances of combinator axioms left to right. It is an open problem whether a simplification ordering enjoying this last property exists, but it appears unlikely. However, for completeness, compatibility with stable contexts suffices. The $>_{\mathsf{ski}}$ ordering introduced in [31] can orient all instances of combinator axioms left to right and is compatible with stable contexts. It is *not* compatible with arbitrary contexts. For terms t_1 and t_2 such that $t_1 >_{\mathsf{ski}} t_2$, it is not necessarily the case that $t_1\,u >_{\mathsf{ski}} t_2\,u$ or that $\mathsf{S}\,t_1\,a\,b >_{\mathsf{ski}} \mathsf{S}\,t_2\,a\,b$. I show that by not superposing underneath fully applied combinators and carrying out

[1] A reviewer of the paper on which this chapter is based suggested that this is merely pushing the problem from one place (explosiveness of combinator axioms) to another (explosiveness of superposition below variables). It should be noted that superposition under variables only takes place when a problem is higher-order. First-order problems contain no unstable subterms. On the other hand, axioms can combine explosively even for essentially first-order problems.

some restricted superposition beneath variables, this lack of compatibility with arbitrary contexts can be circumvented and does not lead to a loss of completeness. In a number of places in the completeness proof, the following conditions on the ordering (satisfied by the $>_{\mathsf{ski}}$ ordering) are assumed. It may be possible to relax the conditions at the expense of an increased number of inferences. Let t and t' be terms.

P1 If $t \longrightarrow_w t'$, then $t \succ t'$;
P2 If $t \succ t'$ and $head(t')$ is not a combinator or variable, $u[t] \succ u[t']$;
P3 **Non-selecting only:** If $\|t\| > \|t'\|$, then $t \succ t'$.

The ordering $\succ$ is extended to literals and clauses using the multiset extension as explained in [88].

4.1.2 | Inference Rules

Clausal combinatory superposition select is further parameterised by a selection function that maps a clause to a subset of its negative literals. Due to the requirements of the completeness proof, if a term $t = x\,\overline{s}_{n>0}$ is a maximal term in a clause C, then a literal containing x as a first-order subterm may not be selected. A selection function meeting this criterion is called *admissible*. A literal l is σ-*eligible* in a clause C if it is selected or there are no selected literals in C and $l\sigma$ is maximal in $C\sigma$. If σ is the identity substitution it is left implicit. In the latter case, it is *strictly eligible* if it is strictly maximal. A variable x has a *bad* occurrence in a clause C if it occurs in C at an unstable position. Occurrences of x in C at stable positions are *good*.

Conventions:

Often a clause is written with a single distinguished literal such as $C' \vee t \approx t'$. In this case:

1. The distinguished literal is always σ-eligible for some σ.
2. The name of the clause is assumed to be the name of the remainder without the dash.
3. If the clause is involved in an inference, the distinguished literal is the literal that takes part.

Positive and negative superposition:

$$\frac{D' \vee t \approx t' \qquad C' \vee [\neg]s\langle u\rangle \approx s'}{(C' \vee D' \vee [\neg]s\langle t'\rangle \approx s')\sigma}\ \textsc{Sup}$$

with the following side conditions:

1. The variable conditions (given below) hold;
2. C is not an extended combinator axiom;
3. $\sigma = mgu(t, u)$;
4. $t\sigma \not\preceq t'\sigma$;
5. $s\langle u\rangle\sigma \not\preceq s'\sigma$;
6. $C\sigma \not\preceq D\sigma$ or D is an instance of an extended combinator axiom;
7. $t \approx t'$ is strictly σ-eligible in D;
8. $[\neg]\, s\langle u\rangle \approx s'$ is σ-eligible in C, and strictly σ-eligible if it is positive.

Definition 39. Let $l = \mathcal{C}_{\mathsf{any}}\langle\overline{\alpha}\rangle\,\overline{x}_n$ and $l \approx r$ be an extended combinator axiom. A term $\zeta\,\overline{u}_m$ is *compatible* with $l \approx r$ if $\mathcal{C}_{\mathsf{any}} = \mathsf{I}$ and $m = n$ or if $\mathcal{C}_{\mathsf{any}} = \mathsf{K}$ and $m \geq n - 1$ or if $\mathcal{C}_{\mathsf{any}} \in \{\mathsf{B}, \mathsf{C}, \mathsf{S}\}$ and $m \geq n - 2$.

Variable condition 1: If $u = x\,\overline{s}_n$ and D is an extended combinator axiom, then D and u must be compatible.
Variable condition 2: For the selecting calculus $u \notin \mathcal{V}$. For the non-selecting calculus, if $u \in \mathcal{V}$ then u must have another occurrence bad in C, t' must have a variable or combinator head and D cannot be an instance of an extended combinator axiom.

Selecting only: Because the term ordering $\succ$ is not compatible with unstable contexts, there are instances where the selecting calculus must perform superposition beneath variables. The SUBVARSUP rule deals with this.

$$\frac{D' \vee t \approx t' \qquad C' \vee [\neg]s\langle y\,\overline{u}_n\rangle \approx s'}{(C' \vee D' \vee [\neg]s\langle z\,t'\,\overline{u}_n\rangle \approx s')\sigma}\ \textsc{SubVarSup}$$

with the following side conditions in addition to conditions 4 – 8 of SUP:

1. y has another occurrence bad in C;
2. z is a fresh variable;
3. $\sigma = \{y \to z\,t\}$;
4. t' has a variable or combinator head;
5. $n \leq 1$;
6. D is not an extended combinator axiom.

Remark 12. The SUBVARSUP rule and the FLUIDSUP rule of [21] bear an obvious similarity. Their purpose is the same, to simulate superposition underneath variables in certain circumstances. However, due to differences in logics and term orders, the side conditions of the rules are rather different.

The EQRES and EQFACT inferences:

$$\frac{C' \vee u \not\approx u'}{C'\sigma}\ \textsc{EqRes} \qquad\qquad \frac{C' \vee u' \approx v' \vee u \approx v}{(C' \vee v \not\approx v' \vee u \approx v')\sigma}\ \textsc{EqFact}$$

For both inferences $\sigma = mgu(u, u')$. For EQRES, $u \not\approx u'$ is σ-eligible in the premise. For EQFACT, $u'\sigma \not\preceq v'\sigma$, $u\sigma \not\preceq v\sigma$, and $u \approx v$ is σ-eligible in the premise.

In essence, the ARGCONG inference allows superposition to take place at prefix positions by 'growing' equalities to the necessary size.

$$\frac{C' \vee s \approx s'}{\begin{array}{l} C'\sigma \vee (s\sigma)\, x \approx (s'\sigma)\, x \\ C'\sigma \vee (s\sigma)\, \overline{x}_2 \approx (s'\sigma)\, \overline{x}_2 \\ C'\sigma \vee (s\sigma)\, \overline{x}_3 \approx (s'\sigma)\, \overline{x}_3 \\ \vdots \end{array}} \text{ ARGCONG}$$

The literal $s \approx s'$ must be σ-eligible in C. Let s and s' be of type $\alpha_1 \to \cdots \to \alpha_m \to \beta$. If β is not a type variable, then σ is the identity substitution and the inference has m conclusions. Otherwise, if β is a type variable, the inference has an infinite number of conclusions. In conclusions where $n > m$ variables are appended, σ is the substitution that maps β to type $\tau_1 \to \cdots \to \tau_{n-m} \to \beta'$ where β' and each τ_i are fresh type variables. In each conclusion, the x_is are variables fresh for C. Note that an ARGCONG inference on a combinator axiom results in an extended combinator axiom.

4.1.3 | Extensionality

Clausal combinatory superposition can be either intensional or extensional. If a conjecture is proved by the intensional version of the calculus, it means that the conjecture holds in all models of the axioms. On the other hand, if a conjecture is proved by the extensional version, it means that the conjecture holds in all extensional models (as defined above). Practically, some domains naturally lend themselves to intensional reasoning whilst others to extensional. For example, when reasoning about programs, we may expect to treat different programs as different entities even if they always produce the same output when provided the same input. For the calculus to be extensional, we provide two possibilities. The first is to add a polymorphic extensionality axiom. Let diff be a polymorphic symbol of type $\Pi\alpha_1, \alpha_2.\, (\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_2) \to \alpha_1$. Then the extensionality axiom can be given as:

$$x\,(\mathsf{diff}\langle\alpha_1, \alpha_2\rangle\; x\; y) \;\not\approx\; y\,(\mathsf{diff}\langle\alpha_1, \alpha_2\rangle\; x\; y) \;\vee\; x \approx y$$

However, adding the extensionality axiom to a clause set can be explosive and is not graceful. By any common ordering, the negative literal will be the larger literal and therefore the literal involved in inferences. As it is not of functional type, it can unify with terms of atomic type including first-order terms.

In order to circumvent this issue, I developed another method of dealing with extensionality. Unification is replaced by unification with abstraction. During the unification procedure, no attempt is made to unify pairs consisting of terms of functional or variable type. Instead, if the remaining unification pairs can be solved successfully, such pairs are added to the resulting clause as negative constraint literals. This process works in conjunction with the negative extensionality rule presented below.

Algorithm 1 Unification algorithm with constraints

function $\mathsf{mgu}_{\mathsf{Abs}}(l, r)$
  let $\mathcal{P}$ be a set of unification pairs; $\mathcal{P} := \{\langle l, r\rangle\}$, $\mathcal{D}$ be a set of disequalities; $\mathcal{D} := \emptyset$
  let θ be a substitution; $\theta := \{\}$
  loop
    if $\mathcal{P}$ is empty **then**
      Return (θ, D), where D is the disjunction of literals in $\mathcal{D}$
    Select a pair $\langle s, t\rangle$ in $\mathcal{P}$ and remove it from $\mathcal{P}$
    if s coincides with t **then** do nothing
    else if s is a variable and s does not occur in t **then** $\theta := \theta \circ \{s \mapsto t\}$; $\mathcal{P} := \mathcal{P}\{s \mapsto t\}$
    else if s is a variable and s occurs in t **then** fail
    else if t is a variable **then** $\mathcal{P} := \mathcal{P} \cup \{\langle t, s\rangle\}$
    else if s and t have functional or variable type **then** $\mathcal{D} := \mathcal{D} \cup \{s \not\approx t\}$
    else if s and t have different head symbols **then** fail
    else if $s = \mathsf{f}\, s_1 \ldots s_n$ and $t = \mathsf{f}\, t_1 \ldots t_n$ for some f **then**
      $\mathcal{P} := \mathcal{P} \cup \{\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle\}$

$$\frac{C' \vee s \not\approx s'}{(C' \vee s\,(\mathsf{sk}\langle\overline{\alpha}\rangle\,\overline{x}) \not\approx s'\,(\mathsf{sk}\langle\overline{\alpha}\rangle\,\overline{x}))\sigma}\ \textsc{NegExt}$$

where $s \not\approx s'$ is σ-eligible in the premise, $\overline{\alpha}$ and $\overline{x}$ are the type and term variable of the literal $s \not\approx s'$. The substitution σ is defined exactly as was done for the ArgCong inference.

Example 13. To motivate this second approach to extensionality consider the following example.

$$\mathsf{g}\,x \approx \mathsf{f}\,x \qquad \mathsf{h}\,\mathsf{g} \not\approx \mathsf{h}\,\mathsf{f}$$

Equality resolution with abstraction on the second clause produces the clause $\mathsf{g} \not\approx \mathsf{f}$. A NegExt inference on this clause results in $\mathsf{g}\,\mathsf{sk} \not\approx \mathsf{f}\,\mathsf{sk}$ which can superpose with $\mathsf{g}\,x \approx \mathsf{f}\,x$ to produce $\bot$.

The unification with abstraction procedure used here is very similar to that introduced in [94]. Pseudocode for the algorithm can be found in Algorithm 1. The inference rules other than ArgCong and SubVarSup must be modified to utilise unification with abstraction rather than standard unification. I present the updated superposition rule. The remaining rules can be modified along similar lines.

$$\frac{C_1 \vee t \approx t' \qquad C_2 \vee [\neg]s\langle u\rangle \approx s'}{(C_1 \vee C_2 \vee D \vee [\neg]s\langle t'\rangle \approx s')\sigma}\ \textsc{Sup-wA}$$

where D is the possibly empty set of negative literals returned by unification. Sup-wA shares all the side conditions of Sup given above. This method of dealing with extensionality is not complete as shown by the following counterexample.[2]

[2]Thanks to Visa Nummelin for noticing the incompleteness and for providing the example.

Example 14. Consider the following set of unsatisfiable clauses. The larger side of each literal is shown in bold.

$$\mathsf{f}\, x \approx \mathbf{g}\, \boldsymbol{x} \qquad \mathbf{k}\, \mathbf{g} \approx \mathsf{k}\, \mathsf{h} \qquad \mathsf{k}\, \mathsf{f} \not\approx \mathbf{k}\, \mathbf{h}$$

It is necessarily the case that $\mathsf{k}\, \mathsf{f} \prec \mathsf{k}\, \mathsf{h} \prec \mathsf{k}\, \mathsf{g}$ if the weight of f, h and g are all the same, g has the highest precedence amongst the three and f the lowest. In this case, the only non-redundant inference available is an equality factoring with abstraction on the clause $\mathsf{k}\, \mathsf{f} \not\approx \mathbf{k}\, \mathbf{h}$ resulting in the clause $\mathsf{f} \not\approx \mathbf{h}$. A negative extensionality inference can be carried out on this clause to produce $\mathsf{f}\, \mathsf{sk} \not\approx \mathbf{h}\, \mathbf{sk}$. On addition of this clause, the clause set is saturated and no further inferences can be performed.

4.2 | Examples

I provide a few examples to illustrate how the calcului works. Some of the examples utilised come from Bentkamp et al.'s paper [21] in order to allow a comparison of the two methods. In all the examples, it is assumed that the clause set has been enriched with the combinator axioms.

Example 15. Consider the unsatisfiable clause:

$$x\, \mathsf{a}\, \mathsf{b} \not\approx x\, \mathsf{b}\, \mathsf{a}$$

Superposing onto the left-hand side with the extended $\mathbf{K}$ axiom $\mathbf{K}\, x_1\, x_2\, x_3 \approx x_1\, x_3$ results in the clause $x_1\, \mathsf{b} \not\approx x_1\, \mathsf{a}$. Superposing onto the left-hand side of this clause, this time with the standard $\mathbf{K}$ axiom adds the clause $x \not\approx x$ from which $\bot$ is derived by an EQRES inference.

Example 16. Consider the unsatisfiable clause set where $\mathsf{f}\, \mathsf{a} \succ \mathsf{c}$:

$$\mathsf{f}\, \mathsf{a} \approx \mathsf{c} \qquad \mathsf{h}\, (y\, \mathsf{b})(y\, \mathsf{a}) \not\approx \mathsf{h}\, (\mathsf{g}(\mathsf{f}\, \mathsf{b}))(\mathsf{g}\, \mathsf{c})$$

A SUP inference between the $\mathbf{B}$ axiom $\mathbf{B}\, x_1\, x_2\, x_3 \approx x_1\, (x_2\, x_3)$ and the subterm $y\, \mathsf{b}$ of the second clause adds the clause $\mathsf{h}(x_1(x_2\, \mathsf{b}))(x_1(x_2\, \mathsf{a})) \not\approx \mathsf{h}(\mathsf{g}(\mathsf{f}\, \mathsf{b}))(\mathsf{g}\, \mathsf{c})$ to the set. By superposing onto the subterm $x_2\, \mathsf{a}$ of this clause with the equation $\mathsf{f}\, \mathsf{a} \approx \mathsf{c}$, we derive the clause $\mathsf{h}(x_1(\mathsf{f}\, \mathsf{b}))(x_1\, \mathsf{c}) \not\approx \mathsf{h}(\mathsf{g}(\mathsf{f}\, \mathsf{b}))(\mathsf{g}\, \mathsf{c})$ from which $\bot$ can be derived by an EQRES inference.

Example 17. Consider the unsatisfiable clause set where $\mathsf{f}\, \mathsf{a} \succ \mathsf{c}$. This example is the combinatory equivalent of Bentkamp et al.'s Example 6.

$$\mathsf{f}\, \mathsf{a} \approx \mathsf{c} \qquad \mathsf{h}\, (y\, (\mathbf{B}\, \mathsf{g}\, \mathsf{f})\, \mathsf{a})\, y \not\approx \mathsf{h}\, (\mathsf{g}\, \mathsf{c})\, \mathbf{I}$$

A SUP inference between the extended I axiom $\mathsf{I}\, x_1\, x_2 \approx x_1\, x_2$ and the subterm $y\,(\mathsf{B}\,\mathsf{g}\,\mathsf{f})\,\mathsf{a}$ of the second clause adds the clause $\mathsf{h}\,(\mathsf{B}\,\mathsf{g}\,\mathsf{f}\,\mathsf{a})\,\mathsf{I} \not\approx \mathsf{h}\,(\mathsf{g}\,\mathsf{c})\,\mathsf{I}$ to the set. Superposing onto the subterm $\mathsf{B}\,\mathsf{g}\,\mathsf{f}\,\mathsf{a}$ of this clause with the B axiom results in the clause $\mathsf{h}\,(\mathsf{g}\,(\mathsf{f}\,\mathsf{a}))\,\mathsf{I} \not\approx \mathsf{h}\,(\mathsf{g}\,\mathsf{c})\,\mathsf{I}$. Superposition onto the subterm $\mathsf{f}\,\mathsf{a}$ with the first clause of the original set gives $\mathsf{h}\,(\mathsf{g}\,\mathsf{c}))\,\mathsf{I} \not\approx \mathsf{h}\,(\mathsf{g}\,\mathsf{c})\,\mathsf{I}$ from which $\bot$ can be derived via EQRES.

Note that in Examples 16 and 17, no use is made of SUBVARSUP even though the analogous FLUIDSUP rule is required in Bentkamp et al.'s calculus. I have been unable to develop an example wherein the selecting calculus requires the SUBVARSUP rule even though it is required for the completeness result in Section 4.4.

4.3 | Redundancy Criterion

In Section 4.4, I prove that the calculus is refutationally complete. The proof follows that of Bachmair and Ganzinger's original proof of the completeness of superposition [9], but is presented in the style of Bentkamp et al. [19] and Waldmann [122]. As is normal with such proofs, it utilises the concept of *redundancy* to reduce the number of clauses that must be considered in the induction step during the model construction process (view Section 4.4, Lemma 43, part (ii)). In the normal definition of redundancy, a clause is redundant if entailed by smaller clauses (see Chapter 2, Definition 21). In the calculus I presented, this would render the conclusion of every ARGCONG inference redundant, since it is larger than its premise and entailed by its premise. Therefore, following Bentkamp et al. [19], I define redundancy using a monomorphic logic known as the *floor logic*. A clause is defined to be redundant if its translation to the floor logic is redundant.

For the remainder of this chapter, let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be an arbitrary but fixed applicative first-order signature. Any clauses or clause sets mentioned in the remainder of the chapter are assumed to be over this signature. The logic induced by this signature is known as the *ceiling logic*. I define an encoding $\lfloor\ \rfloor$, of ground ceiling types $\mathcal{T}y_{\Sigma_{\mathsf{ty}}}(\emptyset)$ and terms $\mathcal{T}_{\Sigma}(\emptyset)$, into monomorphic floor types and terms with $\lceil\ \rceil$ as its inverse. The encoding works by indexing each symbol with its type arguments and argument number. For example, $\lfloor \mathsf{f} \rfloor = \mathsf{f}_0$, $\lfloor \mathsf{f}\langle\overline{\tau}\rangle\mathsf{a} \rfloor = \mathsf{f}_1^{\overline{\tau}}(\mathsf{a}_0)$. I formally define the floor logic and the encoding.

Definition 40 (Floor Logic)**.** Let $(\Sigma_{\mathsf{ty}}^{\mathsf{GF}}, \Sigma^{\mathsf{GF}})$ be the signature of the floor logic. $\Sigma_{\mathsf{ty}}^{\mathsf{GF}}$ can be any infinite set of sorts. The $\lfloor\ \rfloor$ translation is an arbitrary bijection between members of $\mathcal{T}ya_{\Sigma_{\mathsf{ty}}}(\emptyset)$ and members of $\mathcal{T}ya_{\Sigma_{\mathsf{ty}}^{\mathsf{GF}}}$.

For each function symbol $\mathsf{f} : \Pi\overline{\alpha}_m.\, \tau_1 \to \cdots \to \tau_n \to \tau$ in Σ, a family of functions symbols $\mathsf{f}_i^{\overline{v}_m} : (\lfloor \tau_1\sigma \rfloor \times \ldots \times \lfloor \tau_i\sigma \rfloor) \to \lfloor (\tau_{i+1} \to \cdots \to \tau_n \to \tau)\sigma \rfloor$, for each tuple of ground types $\overline{v}_m$ and for each i in $1 \leq i \leq n$, are included in Σ^{GF} where σ is the substitution that maps α_1 to v_1, α_2 to v_2 and so on. For each term $\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s} \in \mathcal{T}_{\Sigma}(\emptyset)$ of type τ, a constant $\mathsf{s}_{(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s})}$ of type $\lfloor\tau\rfloor$ is added to Σ^{GF}. The translation of terms, literals and clauses is defined inductively:

- $\lfloor \mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s}\rfloor = \mathsf{s}_{(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s})}$;
- $\lfloor \mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s}_n\rfloor = \mathcal{C}_n^{\overline{\tau}}(\lfloor s_1\rfloor,\ldots,\lfloor s_n\rfloor)$;
- $\lfloor \mathsf{f}\langle\overline{\tau}\rangle\,\overline{s}_n\rfloor = \mathsf{f}_n^{\overline{\tau}}(\lfloor s_1\rfloor,\ldots,\lfloor s_n\rfloor)$;
- $\lfloor s \approx t\rfloor = \lfloor s\rfloor \approx \lfloor t\rfloor$;
- $\lfloor C = l_1 \vee \ldots \vee l_n\rfloor = \lfloor l_1\rfloor \vee \ldots \vee \lfloor l_n\rfloor$.

Example 18. Consider the applicative signature:

$$\Sigma_{\mathsf{ty}} = \{\rightarrow, i\}$$
$$\Sigma = \{\mathsf{f} : i \rightarrow i, \mathsf{a} : i, \mathbf{B} : \Pi\alpha\beta\gamma.\,(\alpha\rightarrow\gamma)\rightarrow(\beta\rightarrow\alpha)\rightarrow\beta\rightarrow\gamma\}$$

Then $\Sigma^{\mathsf{GF}} = \{\mathsf{f}_0 : \lfloor i\rightarrow i\rfloor, \mathsf{f}_1 : (\lfloor i\rfloor)\rightarrow\lfloor i\rfloor, \mathsf{a}_0 : \lfloor i\rfloor, \mathbf{B}_0^{\tau_1,\tau_2,\tau_3} : \lfloor(\tau_1\rightarrow\tau_3)\rightarrow(\tau_2\rightarrow\tau_1)\rightarrow\tau_2\rightarrow\tau_3\rfloor,\ldots\}$. The member $\mathbf{B}_0^{\tau_1,\tau_2,\tau_3}$ is not actually a single symbol, but rather an infinite set of symbols one for each triple in $\mathcal{T}\mathit{ya}_{\Sigma_{\mathsf{ty}}}(\emptyset)$. Using the above signatures, I provide some examples of the translation at work:

$$\lfloor \mathsf{f}\;\mathsf{a}\rfloor = \mathsf{f}_1(\mathsf{a}_0)$$
$$\lfloor \mathbf{B}\langle i,i,i\rangle\;\mathsf{f}\rfloor = \mathbf{B}_1^{i,i,i}(\mathsf{f}_0)$$
$$\lfloor \mathsf{f}\;(\mathbf{B}\langle i,i,i\rangle\;\mathsf{f}\;\mathsf{f}\;\mathsf{a})\rfloor = \mathsf{f}_1(\mathsf{s}_{(\mathbf{B}\langle i,i,i\rangle\;\mathsf{f}\;\mathsf{f}\;\mathsf{a})})$$

The function $\lceil\;\rceil$ is used to compare floor terms. More precisely, for floor logic terms t and t', $t \succ t'$ if $\lceil t\rceil \succ \lceil t'\rceil$. Let $\succ$ be an ordering on ground ceiling terms that is well-founded, total on ground terms, possesses the subterm property and is compatible with stable contexts. In this case, it is straightforward to show that the order $\succ$ on floor terms is compatible with all contexts, well-founded, total on ground terms and has the subterm property.

The encoding serves a dual purpose. Firstly, as redundancy is defined with respect to the floor logic, it prevents the conclusion of all ArgCong inferences from being redundant. Secondly, subterms in the floor logic correspond to first-order subterms in the ceiling logic. This is of critical importance in the completeness proof.

Let N be a set of ground floor clauses and C a ground floor clause. Let ECA be the set of all extended combinator axioms. Then C is redundant with respect to N if:

$$C \notin \lfloor\mathcal{G}_\Sigma(ECA)\rfloor \text{ and } \{D \in N \mid D \prec C\} \cup \lfloor\mathcal{G}_\Sigma(ECA)\rfloor \models C$$

In the above definition, $\models$ refers to the entailment function of the floor logic. Let N a set of ground ceiling clauses and C a ground ceiling clause. Then C is redundant with respect to N if $\lfloor C\rfloor$ is redundant with respect to $\lfloor N\rfloor$.

An arbitrary ceiling clause C is redundant with respect to a set of ceiling clauses N if

every clause $D \in G_\Sigma(C)$ is redundant with respect to $\mathcal{G}_\Sigma(N)$.

The set of all clauses redundant with respect to a set of clauses N is written $Red_C(N)$. I refine Definition 22 to fit the current context.

Definition 41 (Ground Instance of an Inference)**.** Let inf be a non-ARGCONG inference $C_1, \ldots, C_n \vdash C$. Let θ be a grounding substitution. (**Selecting only:** for $1 \leq i \leq n$, assign the selected literals of $C_i\theta$ to correspond with the selected literals of C_i.) For any inference $inf' = C_1\theta, \ldots, C_n\theta \vdash E$ such that $C\theta \rightarrow_w^* E$, inf' is the θ-ground instance of inf.

Let N be a set of ground ceiling clauses and ECA be the set of all extended combinator axioms. Let $inf = C_1, \ldots, C_n\, C \vdash E$ be a ground inference other than ARGCONG with maximal premise C. Then inf is redundant with respect to N if any of its premises are in $Red_C(N)$ or:

$$\{D \in \lfloor N \rfloor \mid D \prec \lfloor C \rfloor\} \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models \lfloor E \rfloor$$

A non-ground non-ARGCONG inference is redundant with respect to a set of non-ground clauses N, if all its ground instances are redundant with respect to $\mathcal{G}_\Sigma(N)$. The set of all inferences redundant with respect to a set of clauses N is written $Red_I(N)$.

An ARGCONG inference is redundant with respect to a set of clauses N if its premise is redundant with respect to N, or its conclusion is in $N \cup Red_C(N)$.

Definition 42. A set N is saturated up to redundancy if for every inference inf with premises in N, $inf \in Red_I(N)$.

I prove some properties of the redundancy criterion and the grounding definition that are necessary for subsequently linking static to dynamic completeness.

Lemma 28. *Let G_{inf} be the set of all ground instances of a non-*ARGCONG *inference inf. Then $G_{inf} \subseteq Red_I(G_\Sigma(conc(inf)))$.*

Proof. I need to show that for any inference $inf' \in G_{inf}$, $inf' \in Red_I(G_\Sigma(conc(inf)))$. Let $inf = C_1, \ldots, C_n \vdash C$ and $inf' = D_1, \ldots D_n \vdash D$ where C_n and D_n are the maximal premises of inf and inf' respectively. By Definition 41, we have that there exists a ground substitution θ, such that $C_i\theta = D_i$ for $1 \leq i \leq n$ and $C\theta \rightarrow_w^* D$.

To show that $inf' \in Red_I(G_\Sigma(C))$, I need to show that $\{E \in \lfloor \mathcal{G}_\Sigma(C) \rfloor \mid E \prec \lfloor D_n \rfloor\} \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models \lfloor D \rfloor$. This is true since $\lfloor C\theta \rfloor \in \lfloor G_\Sigma(C) \rfloor$, $\lfloor C\theta \rfloor \prec \lfloor D_n \rfloor$ and $\lfloor C\theta \rfloor \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models \lfloor D \rfloor$. The last claim follows from the fact that $C\theta \rightarrow_w^* D$. □

Lemma 29. *Let C be a ground floor clause redundant with respect to a set of ground floor clauses N. Then C is entailed by clauses $C_1, \ldots, C_n \in N \setminus Red_C(N)$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$.*

Proof. Since C is redundant with respect to N, there exists clauses $C_1, \ldots, C_k$ in N such that $C_1, \ldots, C_k \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models C$ and $C \succ C_i$ for $1 \leq i \leq k$. Let $\succ_{mul}$ be the multiset extension of $\succ$. Furthermore, let $N' = \{C_1, \ldots, C_n\}$ be the minimal $\succ_{mul}$ subset of N such that $N' \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ entails C. I show that each C_i, for $1 \leq i \leq n$, is non-redundant with respect to N. If $C_i \in \lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ then this follows by definition. For some $C_i \notin \lfloor \mathcal{G}_\Sigma(ECA) \rfloor$, assume that it is redundant with respect N and that there exists clauses $D_1, \ldots, D_m$ such that $D_1, \ldots, D_m \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models C_i$ and $C_i \succ D_j$ for $1 \leq j \leq m$. But then $N' \setminus C_i \cup \{D_1, \ldots, D_m\} \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models \lfloor C \rfloor$ and $N' \succ_{mul} N' \setminus C_i \cup \{D_1, \ldots, D_m\}$ contradicting the minimality of N'. □

4.4 | Refutational Completeness

Let N be a set of clauses containing the five combinator axioms and saturated up to redundancy by the clausal combinatory superposition calculi parameterised by an admissible ordering (**selecting only:** an an admissible selection function sel). Let $\gg$ be some arbitrary well-founded order on ceiling clauses. I define $\mathcal{G}^{-1}$ to be a function that maps each clause in $\mathcal{G}_\Sigma(N)$ to the $\gg$-maximal clause in N that it is an instance of.

Selecting only: In the upcoming proof, it is important that for every clause $D \in \mathcal{G}_\Sigma(N)$, there is a clause $C \in N$ such that the selected literals of D and C correspond. Let $gsel$ be a selection function on clauses in $\mathcal{G}_\Sigma(N)$ such that for every clause $C\theta \in \mathcal{G}_\Sigma(N)$ with $\mathcal{G}^{-1}(C\theta) = C$, $gsel(C\theta) = sel(C)\theta$. By definition, the $gsel$ selected literals of a clause $D \in \mathcal{G}_\Sigma(N)$ correspond to the sel selected literals of the clause $C = \mathcal{G}^{-1}(D) \in N$. For the remainder of this section, it is assumed that clauses in $\mathcal{G}_\Sigma(N)$ have their selected literals chosen by $gsel$.

By the fact that N is saturated up to redundancy and by the definitions of clause redundancy and inference redundancy for ARGCONG inferences, it follows that the following extended combinator axioms are in N for all $n \in \mathbb{N}$.

$$\mathsf{I}\, x\, \overline{x}_n \approx x\, \overline{x}_n$$
$$\mathsf{K}\, x\, y\, \overline{x}_n \approx x\, \overline{x}_n$$
$$\mathsf{B}\, x\, y\, z\, \overline{x}_n \approx x\, (y\, z)\, \overline{x}_n$$
$$\mathsf{C}\, x\, y\, z\, \overline{x}_n \approx x\, z\, y\, \overline{x}_n$$
$$\mathsf{S}\, x\, y\, z\, \overline{x}_n \approx x\, z\, (y\, z)\, \overline{x}_n$$

Thus, $ECA \subseteq N$ where ECA is the set consisting of all extended combinator axioms. Following from this, $\mathcal{G}_\Sigma(ECA) \subseteq \mathcal{G}_\Sigma(N)$. As per Bentkamp et al. [19], I build a model for $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ which can then be lifted to a model of $\mathcal{G}_\Sigma(N)$ and N. I define a term-

rewriting system R_∞ and use it to construct an interpretation of $\lfloor \mathcal{G}_\Sigma(N) \rfloor$. I then use induction to prove that this interpretation is a model of $\lfloor \mathcal{G}_\Sigma(N) \rfloor$.

4.4.1 | Candidate Interpretation

I define the set of rewrite rules R_{ECA} as $\{l \to r \mid l \approx r \in \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \text{ and } l \succ r\}$. By the condition that the term order orient all ground instances of combinator axioms left to right, we have that R_{ECA} is the floor of the ground of the set of all extended combinator axioms turned into left to right rewrite rules.

Lemma 30. *Let $R_{ECA} = R'_{ECA} \cup \{l \to r\}$. Then, l is not reducible by any rule in R'_{ECA}.*

Proof. Let $l = \lfloor \mathcal{C}_{\mathsf{any}} \langle \overline{\tau} \rangle \, \overline{s} \rfloor$ and $l' = \lfloor \mathcal{C}_{\mathsf{any}} \langle \overline{\tau'} \rangle \, \overline{s'} \rfloor$ be the left hand sides of two different members of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. By the definition of the floor translation $\lfloor \ \rfloor$, we have that $\lfloor \mathcal{C}_{\mathsf{any}} \langle \overline{\tau} \rangle \, \overline{s} \rfloor = \mathsf{s}_{(\mathcal{C}_{\mathsf{any}} \langle \overline{\tau} \rangle \, \overline{s})}$ and $\lfloor \mathcal{C}_{\mathsf{any}} \langle \overline{\tau'} \rangle \, \overline{s'} \rfloor = \mathsf{s}_{(\mathcal{C}_{\mathsf{any}} \langle \overline{\tau'} \rangle \, \overline{s'})}$. Since $\mathsf{s}_{(\mathcal{C}_{\mathsf{any}} \langle \overline{\tau} \rangle \, \overline{s})} \neq \mathsf{s}_{(\mathcal{C}_{\mathsf{any}} \langle \overline{\tau'} \rangle \, \overline{s'})}$, this proves the lemma. □

For every clause $C \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$, induction on $\succ$ is used to define sets of rewrite rules E_C and R_C. Assume that E_D has been defined for all clauses $D \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$ such that $D \prec C$. Then R_C is defined as $R_{ECA} \cup (\bigcup_{D \prec C} E_D)$. The set E_C contains the rewrite rule $s \to t$ if the following conditions are met. Otherwise $E_C = \emptyset$.

(a) $C = C' \vee s \approx t$
(b) $s \approx t$ is strictly maximal in C
(c) $s \succ t$
(d) C is false in R_C
(e) C' is false in $R_C \cup \{s \to t\}$
(f) s is not reducible by any rule in R_C

In this case C is called *productive*. R_∞ is defined as $R_{ECA} \cup (\bigcup_{C \in \lfloor \mathcal{G}_\Sigma(N) \rfloor} E_C)$. Note that due to the definition of R_C and condition (d) an extended combinator axiom is never productive. I define a first-order interpretation $\mathcal{T}_\Sigma(\emptyset)/R$ from a rewrite system R as follows. For an equation $t \approx t'$, $\mathcal{T}_\Sigma(\emptyset)/R \models t \approx t'$ if and only if $t \leftrightarrow_R^* t'$. More formally, $\mathcal{T}_\Sigma(\emptyset)/R$ is the monomorphic interpretation $(\mathcal{I}_{\mathsf{ty}}, \mathcal{J})$ where $\mathcal{I}_{\mathsf{ty}} = (\mathcal{U}, \mathcal{J}_{\mathsf{ty}})$, $\mathcal{U}$ consists of universes $\mathcal{U}_\tau$, each of which contains R-equivalence classes of terms of type τ. The function $\mathcal{J}_{\mathsf{ty}}$ maps each type τ, to $\mathcal{U}_\tau$. By an abuse of notation, I use R to refer to both a rewrite system and the interpretation that it induces. The interpretation induced by a rewrite system R is term-generated in that for each universe element a there exists a ground term t such that $[\![t]\!]_R = \mathsf{a}$.

Lemma 31. *The rewrite systems R_C and R_∞ are confluent and terminating.*

Proof. Condition (c) ensures that for every rule $s \to t$ in R_C or R_∞ we have $s \succ t$. By the well-foundedness of $\succ$ we have that R_C and R_∞ must be terminating.

By Lemma 30 there are no critical pairs between rules in R_{ECA}. Using this fact and condition (f), we have that there are no critical pairs between rules in R_C and R_∞. Absence of critical pairs implies local confluence. Local confluence and termination implies confluence. □

Lemma 32. *If a clause D is true in R_D then it is true in R_C for all $C \succ D$ and in R_∞.*

Proof. As per Waldmann's proof [122]. □

Lemma 33. *If a clause $D = D' \vee s \approx t$ is productive then D' is false and $s \approx t$ is true in R_C and R_∞ for all $C \succ D$.*

Proof. As per Waldmann's proof [122]. □

Lemma 34. *Every member of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ is true in R_C for all $C \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$ and in R_∞.*

Proof. By construction of R_{ECA}, all members of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ are true in R_{ECA}. Thus, by definition of R_C all members of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ are true in R_C. □

The major component of the completeness proof is in demonstrating that R_∞ is a model of $\mathcal{G}_\Sigma(N)$. In order to show this, it is necessary that every ground inference with premises in $\mathcal{G}_\Sigma(N)$ is either the ground instance of an inference with premises in N (is *liftable*) or that it is contained in $Red_I(\mathcal{G}_\Sigma(N))$.

4.4.2 | Liftable Inferences

Lemma 35 (Lifting non-SUP inferences)**.** *Let $C\theta \in \mathcal{G}_\Sigma(N)$, and let $\mathcal{G}^{-1}(C\theta) = C$. Then, every* EQRES*,* EQFACT *and ground instance of an* ARGCONG *inference from $C\theta$ is the ground instance of a corresponding inference from C.*

Proof. The proof is identical to that in [19] ignoring purification. □

Definition 43. Let $D\theta = D'\theta \vee t\theta \approx t'\theta$ and $C\theta$ be clauses in $\mathcal{G}_\Sigma(N)$ with $\mathcal{G}^{-1}(C\theta) = C$ and $\mathcal{G}^{-1}(D\theta) = D$. A ground SUP inference with $D\theta$ as the left premise and $C\theta$ as the right is *liftable* in any of three cases:

1. The superposed subterm in $C\theta$ is not at or below a variable in C and variable condition 1 holds between C and D
2. The superposed subterm in $C\theta$ is at a variable y in C which has another occurrence bad in C, $head(t'\theta)$ is a combinator, and D is not an extended combinator axiom.
3. **Selecting only:** The superposed subterm in $C\theta$ is below a variable y in C which has another occurrence bad in C, $head(y\theta)$ and $head(t'\theta)$ are combinators and D is not an extended combinator axiom.

Next, I show that every liftable SUP inferences with premises in $\mathcal{G}_\Sigma(N)$ is the ground instance of a SUP (**selecting only:** or SUBVARSUP) inference with premises in N. One of the tricky aspects of the proof is to show that for the selecting calculus, if the inference occurs beneath a variable, the z variable of the SUBVARSUP inference can be used to create a context that weak-reduces to an arbitrary context surrounding a term t. The translation function $(\!|\ |\!)$, from λ-terms to applicative first-order terms defined in section 2.7, is useful here. I prove the property that for λ-free terms t_1 and t_2, $(\!|\, \lambda x.t_1 \,|\!)\, t_2 \rightarrow_w^* t_1\{x \rightarrow t_2\}$.

Lemma 36. *Let t_1 and t_2 be applicative first-order terms. Then $(\!|\, \lambda x.t_1 \,|\!)\, t_2 \rightarrow_w^* t_1\{x \rightarrow t_2\}$.*

Proof. The proof is by induction on t_1. If $t_1 = x$, then:

$$(\!|\, \lambda x.\, x \,|\!)\, t_2 = \mathsf{I}\, t_2 \rightarrow_w t_2 = x\{x \rightarrow t_2\}$$

If t_1 is a term which does not contain x, then:

$$(\!|\, \lambda x.\, t_1 \,|\!)\, t_2 = \mathsf{K}\, t_1\, t_2 \rightarrow_w t_1 = t_1\{x \rightarrow t_2\}$$

If $t_1 = s\, t$ and the above case doesn't apply, then either x occurs in s or in t or in both. I provide only the case where x occurs in s, the other two cases being similar.

$$(\!|\, \lambda x.\, s\, t \,|\!)\, t_2 = \mathsf{C}\, (\!|\, \lambda x.\, s \,|\!)\, t\, t_2 \rightarrow_w (\!|\, \lambda x.\, s \,|\!)\, t_2\, t \overset{IH}{\rightarrow}_w^* s\{x \rightarrow t_2\}\, t = (s\, t)\{x \rightarrow t_2\} \quad \square$$

Lemma 37 (Lifting SUP inferences)**.** *Let $D\theta$ and $C\theta$ be members of $\mathcal{G}_\Sigma(N)$ such that $\mathcal{G}^{-1}(D\theta) = D$ and $\mathcal{G}^{-1}(C\theta) = C$. Then any liftable* SUP *inference with $C\theta$ as its right premise and $D\theta$ as its left premise is the θ-ground instance of an inference from C and D.*

Proof. I assume that $D = D' \vee t \approx t'$ and that $C = C' \vee [\neg]s \approx s'$ and that the ground inference has the form:

$$\frac{D'\theta \vee t\theta \approx t'\theta \qquad C'\theta \vee [\neg]s\theta\langle t\theta\rangle|_p \approx s'\theta}{C'\theta \vee D'\theta \vee [\neg]s\theta\langle t'\theta\rangle|_p \approx s'\theta}\ \text{SUP}$$

where $t\theta \approx t'\theta$ is strictly eligible, $[\neg]s\theta \approx s'\theta$ is strictly eligible if positive and eligible if negative, $C\theta \not\preceq D\theta$ or $D\theta \in \mathcal{G}_\Sigma(ECA)$, $t\theta \not\preceq t'\theta$ and $s\theta \not\preceq s'\theta$. The proof is broken into two cases.

Case 1: $t\theta$ is not beneath (**selecting only:** at or beneath) a variable in C. In this case, p must be a position of s. Let $u = s|_p$. We have that θ is a unifier of u and t and therefore, there must exist an idempotent $mgu\ \sigma$ of t and u. The inference conditions can be lifted. $t\theta \not\preceq t'\theta$ implies $t \not\preceq t'$ and $s\theta \not\preceq s'\theta$ implies $s \not\preceq s'$. Moreover $t\theta \approx t'\theta$ being strictly eligible in $D\theta$ (**selecting only:** with respect to $gsel$) implies that $t \approx t'$ is strictly

σ-eligible in D (**selecting only:** with respect to sel) and $[\neg]s\theta \approx s'\theta$ being (strictly) σ-eligible in $C\theta$ (**selecting only:** with respect to $gsel$) implies that $[\neg]s \approx s'$ is (strictly) σ-eligible in C (**selecting only:** with respect to sel). If u is not a variable, liftability, we have that variable condition 1 holds between D and C. Furthermore, we require that either D is an instance of a member of ECA or that $C \not\preceq D$. $D\theta \in \mathcal{G}_\Sigma(ECA)$ implies $D \in ECA$. $D\theta \notin \mathcal{G}_\Sigma(ECA)$ implies that $C\theta \not\preceq D\theta$ which implies $C \not\preceq D$.

Non-selecting only: If u is a variable, liftability implies that it has another occurrence bad in C, $head(t'\theta)$ being a combinator implies $head(t')$ is a variable or combinator, and $D\theta \notin \mathcal{G}_\Sigma(ECA)$ implies D is not an instance of an extended axiom and so variable condition 2 holds.

Therefore, there is the following SUP inference between D and C:

$$\frac{D' \vee t \approx t' \qquad C' \vee [\neg]s\theta\langle t\rangle|_p \approx s'}{(C' \vee D' \vee [\neg]s\langle t'\rangle|_p \approx s')\sigma}\ \textsc{Sup}$$

We have $(C' \vee D' \vee [\neg]s\langle t'\rangle|_p \approx s')\sigma\theta = (C' \vee D' \vee [\neg]s\langle t'\rangle|_p \approx s')\theta = C'\theta \vee D'\theta \vee [\neg]s\theta\langle t'\theta\rangle|_p \approx s'\theta$ by the idempotency of σ proving that the conclusion of the ground inference is the θ-ground instance of the conclusion of the non-ground inference.

Case 2 (selecting only): $t\theta$ is at or beneath a variable in C which has another instance bad in C. In this case, there must exist positions p' and p'' such that $p = p'.p''$ and $s|_{p'} = y\,\overline{u}_n$. Let $u = s|_{p'}$. As per case 1, (strict) eligibility of the ground literals implies (strict) eligibility of the non-ground literals. Further $t'\theta$ having a combinator head implies that t' has a variable or combinator head.

If the inference is beneath a variable in C, then by liftability, $y\theta = \mathcal{C}_{\mathsf{any}}\,\overline{v}_{m>0}$. If $u = y\,\overline{u}_{n>1}$, then $u\theta = \mathcal{C}_{\mathsf{any}}\,\overline{v}_m(\overline{u}_n)\theta$. Thus, $t\theta$ which is a proper subterm of $u\theta$ would be beneath a fully applied combinator which is impossible. Therefore, we have that $n \leq 1$. If the inference is at a variable in C, then we must have $n = 0$ otherwise $y\theta = t\theta$ would not be a first-order subterm.

Thus in both cases $n \leq 1$ and there is a SUBVARSUP inference between C and D:

$$\frac{D' \vee t \approx t' \qquad C' \vee [\neg]s\theta\langle y\,\overline{u}_n\rangle|_{p'} \approx s'}{(C' \vee D' \vee [\neg]s\langle z\,t'\,\overline{u}_n\rangle|_{p'} \approx s')\{y \to z\,t\}}\ \textsc{SubVarSup}$$

where z is a fresh variable. Define the substitution θ' that maps z to $(\!|\,\lambda x.(y\theta)\langle x\rangle|_{p''}\,|\!)$ and all other variables w to $w\theta$. Since z is fresh, $C\theta = C\theta'$ and $D\theta = D\theta'$. Further, we have that $(z\,t')\theta' = (\!|\,\lambda x.(y\theta)\langle x\rangle|_{p''}\,|\!)\,t'\theta \to_w^* y\theta\langle t'\theta\rangle|_{p''}$ and $(z\,t)\theta' \to_w^* y\theta\langle t\theta\rangle|_{p''} = y\theta$.

Therefore:

$$\begin{aligned} & (C' \vee D' \vee [\neg]s\langle z\, t'\, \overline{u}_n\rangle|_{p'} \approx s')\{y \to z\, t\}\theta' \\ = \quad & (C' \vee D' \vee [\neg]s\langle (\!| \lambda x.(y\theta)\langle x\rangle|_{p''} |\!)\, t'\, \overline{u}_n\rangle \approx s')\theta[y \to z\theta'\, t\theta] \\ \to_w^* \quad & C'\theta \vee D'\theta \vee [\neg]s\theta\langle t'\theta\rangle|_p \approx s'\theta \end{aligned}$$

proving that θ'-ground instance of the conclusion of the non-ground inference weak-reduces to the conclusion of the ground inference. By the definition of the ground instances of an inference (Definition 41), the ground inference is a θ'-ground instance of the non-ground inference. □

Lemmas 35 and 37 show that every liftable inference with premises in $\mathcal{G}_\Sigma(N)$ is the ground instance of an inference with premises in N.

Lemma 38. *Let inf' be a liftable ground* EQRES, EQFACT *or* SUP *inference with premises in $\mathcal{G}_\Sigma(N)$ and maximal premise C. Then, $\{D \prec C \mid D \in \lfloor \mathcal{G}_\Sigma(N)\rfloor\} \cup \lfloor \mathcal{G}_\Sigma(ECA)\rfloor \models \lfloor conc(inf')\rfloor$. In words, the conclusion of inf' is entailed by the floor of clauses in $\mathcal{G}_\Sigma(N)$ smaller than $\lfloor C\rfloor$ and by clauses in $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$.*

Proof. By Lemmas 35 and 37, inf' is the ground instance of an inference inf with premises in N. Since N is saturated up to redundancy, inf must be in $Red_I(N)$. By definition, $inf' \in Red_I(\mathcal{G}_\Sigma(N))$ and again by definition, $\{D \mid (D \in \lfloor \mathcal{G}_\Sigma(N)\rfloor) \wedge (D \prec \lfloor C\rfloor)\} \cup \lfloor \mathcal{G}_\Sigma(ECA)\rfloor \models \lfloor conc(inf')\rfloor$. □

Lemma 39. *Assume that N has been saturated by the no-select calculus. Let $D \in \mathcal{G}_\Sigma(N)$ be a clause such that $\|D\| > 0$. Then D is redundant with respect to $\mathcal{G}_\Sigma(N)$.*

Proof. Let $C = \mathcal{G}^{-1}(D)$. The proof splits into two depending on whether there is a weak redex in D at or beneath a variable of C or not. Let $D = C\theta$ where θ is a grounding substitution. If there exists a weak redex in D at or beneath a variable of C, it must be the case that for some variable x in C, $\|x\theta\| > 0$. Let θ' be the substitution that maps all variables y to $y\theta$ and maps x to $(x\theta)\downarrow_w$. Then, $C\theta' \prec D$ and $\lfloor D\rfloor$ is entailed by $\lfloor C\theta'\rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$. Therefore, D is redundant with respect to $\mathcal{G}_\Sigma(N)$.

Assume that there exists no weak redex in D at or beneath a variable of C. Let $l = [\neg]s \approx t$ be the maximal literal of D. By property P3 of the ordering, $\|l\| \geq \|l'\|$ for all literals $l' \in D$. Let s be the maximal side of l. Using property P3 again, we have $\|s\| \geq \|t\|$. Thus, using the fact that $\|D\| > 0$, we can conclude $\|s\| > 0$. Since we are considering the no-select calculus, D has no selected literals so l must be eligible in D. Therefore, the following ground inference exists:

$$\frac{\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_n \approx (\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_n)\downarrow^w \qquad D' \vee s = u\langle\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_n\rangle \approx t}{D'' = D' \vee s = u\langle(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_n)\downarrow^w\rangle \approx t}\ \text{SUP}$$

The left premise is the ground instance of some extended combinator axiom $l \approx r$. Since the weak redex in D is not below a variable of C, there must exist a term $u = \zeta\, \overline{r}_m$ in C such that $u\theta = \mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_n$ and $\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_{n-m}$ is not a redex. This last condition holds, since if it were a redex, $\zeta\theta = \mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_{n-m}$ would contradict the fact that no redex occurs in D at or below a variable of C. Thus, variable condition 1 holds between C and $l \approx r$. Therefore, by Lemma 38, $\{E \in \lfloor \mathcal{G}_\Sigma(N) \rfloor \mid E \prec D\} \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models \lfloor D'' \rfloor$. But then, since $\lfloor D \rfloor$ is entailed by $\lfloor \mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_n \approx (\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\, \overline{s}_n)\downarrow^w \rfloor$ and $\lfloor D'' \rfloor$, we have that $\{E \in \lfloor \mathcal{G}_\Sigma(N) \rfloor \mid E \prec D\} \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models \lfloor D \rfloor$ and thus that D is redundant with respect to $\mathcal{G}_\Sigma(N)$. □

4.4.3 | Non-liftable Inferences

In this section I show that all non-liftable inferences with premises in $\mathcal{G}_\Sigma(N)$ are in $Red_I(\mathcal{G}_\Sigma(N))$. The lemma which proves this result is Lemma 42. The lemma identifies the various ways in which a ground inference may not be liftable and then shows that for each, the conclusion is entailed by clauses smaller than itself in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. Lemmas 40 and 41 are helper lemmas. The conclusion of Lemma 40 is stronger than that of Lemma 41, but it has stronger assumptions as well. Next, terminology is defined that is used in the following couple of Lemmas and later on in this thesis. Let R be an interpretation and t and t' be ground ceiling terms. Then, $t \sim_R t'$ stands for $R \models \lfloor t \rfloor \approx \lfloor t' \rfloor$. Where the interpretation is obvious, the subscript is omitted.

Lemma 40. *Let R be an interpretation such that every member of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ holds in R. Let u and u' be ground terms such that $u\,\overline{s} \sim_R u'\,\overline{s}$ for every type correct tuple of ground terms $\overline{s}$. Then for ground terms s and s' such that s and s' only differ at positions where s contains u and s' contains u', we have $s \sim_R s'$.*

Proof. The proof proceeds by showing how equations that are true in R can be used to rewrite s into s'. Since R is a model of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ all ground instances of combinator and extended combinator axioms are true in R.

Let $s_0 = \lfloor s \rfloor$ and $\tilde{s}_0 = \lfloor s' \rfloor$. Terms $s_1, s_2, s_3, \ldots$ are defined inductively as follows: s_{i+1} is formed from s_i by rewriting all subterms of the form $\lfloor u\,\overline{v} \rfloor$ in s_i to $\lfloor u'\,\overline{v} \rfloor$ and then reducing the outermost leftmost weak redex in the resulting term. Terms $\tilde{s}_1, \tilde{s}_2, \tilde{s}_3, \ldots$ are also defined inductively: $\tilde{s}_{i+1}$ is formed from $\tilde{s}_i$ by reducing the left-most outermost weak redex in $\tilde{s}_i$.

The algorithm maintains the invariant that for all i, $\lceil s_i \rceil$ and $\lceil \tilde{s}_i \rceil$ are identical other than at positions where $\lceil s_i \rceil$ contains a u and $\lceil \tilde{s}_i \rceil$ contains a u'. Let $\tilde{s}_*$ be the final term in the $\tilde{s}_n$ series. Such a term must exist as weak reduction is terminating. Let s_* be the equivalent term in s_n series. We show that $s_* = \tilde{s}_*$.

Assume that $s_* \neq \tilde{s}_*$. Consider the outermost position at which $\lceil s_* \rceil$ and $\lceil \tilde{s}_* \rceil$ differ. If this position is not beneath a fully applied combinator in $\lceil s_* \rceil$, then s_* and $\tilde{s}_*$ must con-

tain $\lfloor u\,\overline{v}\rfloor$ and $\lfloor u'\,\overline{v'}\rfloor$ at the corresponding positions. But this is impossible since $\lfloor u\,\overline{v}\rfloor$ would have been rewritten to $\lfloor u'\,\overline{v'}\rfloor$ at the previous step of the algorithm. Therefore, assume that the outermost position at which $\lceil s_*\rceil$ and $\lceil \tilde{s_*}\rceil$ differ is beneath a fully applied combinator in $\lceil s_*\rceil$. Since we are considering the outermost position, this combinator cannot be a part of u. Thus, we must have that the same combinator occurs fully applied in $\lceil \tilde{s}_*\rceil$. This contradicts the fact that $\tilde{s}_*$ is in weak normal form. Therefore the original assumption is false and $s_* = \tilde{s}_*$. □

Lemma 41. *Let R be an interpretation such that every member of $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$ holds in R. Let u and u' be ground terms such that $u \sim_R u'$. Let $s\langle\langle\,\rangle\rangle_n$ be a ground context with n holes at stable positions. Then $s\langle\langle\overline{u}\rangle\rangle_n \sim_R s\langle\langle\overline{u'}\rangle\rangle_n$.*

Proof. The proof proceeds by induction on $\|s\langle\langle\,\rangle\rangle_n\|$. In the base case $\|s\langle\langle\,\rangle\rangle_n\| = 0$ and no instance of u occurs beneath a fully applied combinator. By the definition of stable positions, no instance of u occurs with arguments. Thus, $\lfloor s\langle\langle\overline{u}\rangle\rangle_n\rfloor$ can be rewritten to $\lfloor s\langle\langle\overline{u'}\rangle\rangle_n\rfloor$ directly using the equation $\lfloor u \approx u'\rfloor$.

For the inductive case, we have that $\|s\langle\langle\,\rangle\rangle_n\| > 0$. Therefore, it must be the case that there exists a context r such that $s\langle\langle\,\rangle\rangle_n \rightarrow_w u$. By Lemma 8, all holes in r occur at stable positions and therefore $r = s'\langle\langle\,\rangle\rangle_m$ for some m. Because $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$ holds in R, we have $s\langle\langle\overline{u}\rangle\rangle_n \sim_R s'\langle\langle\overline{u}\rangle\rangle_m$. By the induction hypothesis we have $s'\langle\langle\overline{u}\rangle\rangle_m \sim_R s'\langle\langle\overline{u'}\rangle\rangle_m$ and then using the relevant member of $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$, we have $s'\langle\langle\overline{u'}\rangle\rangle_m \sim_R s\langle\langle\overline{u'}\rangle\rangle_n$. By the transitivity of $\sim_R$, we have $s\langle\langle\overline{u}\rangle\rangle_n \sim_R s\langle\langle\overline{u'}\rangle\rangle_n$. □

Lemma 42. *Let $C\theta, D\theta \in \mathcal{G}_\Sigma(N)$, where $\mathcal{G}^{-1}(C\theta) = C$ and $\mathcal{G}^{-1}(D\theta) = D$. Consider a non-liftable* SUP *inference inf between $C\theta$ and $D\theta$. Assume that $C\theta$ and $D\theta$ are not redundant with respect to $\mathcal{G}_\Sigma(N)$. Then inf is in $Red_I(\mathcal{G}_\Sigma(N))$.*

Proof. I assume that $D = D' \vee t \approx t'$, that $C = C' \vee [\neg]s \approx s'$ and that the inf has the form:

$$\frac{D'\theta \vee t\theta \approx t'\theta \qquad C'\theta \vee [\neg]s\theta\langle t\theta\rangle|_p \approx s'\theta}{C'\theta \vee D'\theta \vee [\neg]s\theta\langle t'\theta\rangle|_p \approx s'\theta}\ \text{SUP}$$

To show that $inf \in Red_I(\mathcal{G}_\Sigma(N))$, I need to show that $\lfloor conc(inf)\rfloor$ is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(N)\rfloor$ smaller than $\lfloor C\theta\rfloor$. Let R be an interpretation such that clauses in $\lfloor \mathcal{G}_\Sigma(N)\rfloor$ smaller than $\lfloor C\theta\rfloor$ and all members of $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$ are true in R. I need to show that $R \models \lfloor conc(inf)\rfloor$. We can assume that $\lfloor D'\theta\rfloor$ is false in R since if it were true $\lfloor conc(inf)\rfloor$ would follow immediately. By the SUP conditions, we have that either $C\theta \succ D\theta$ or $D\theta \in \mathcal{G}_\Sigma(ECA)$ and in both cases $R \models \lfloor t\theta \approx t'\theta\rfloor$. Thus, if it can be shown that $R \models \lfloor C\theta\rfloor$, then $R \models \lfloor conc(inf)\rfloor$ follows by congruence.

Let $ss = t\theta$ be the superposed subterm in $C\theta$. The inference can be non-liftable for one of the following reasons:

1. Variable condition 1 does not hold between C and D;

2. **Selecting only:** ss is below a variable x in C and $x\theta$ has a first-order head;
3. **Non-selecting only:** ss is below a variable in C;
4. ss is at (**selecting only:** or below) a variable in C all of whose other occurrences are good in C;
5. ss is at (**selecting only:** or below) a variable in C and $t'\theta$ has a first-order head;
6. ss is at (**selecting only:** or below) a variable in C and D is an extended combinator axiom.

I fix some terminology common to cases 2–6. Let x be the variable at or beneath which the inference takes place. Then $t\theta$ is a first-order subterm of $x\theta$. Let v be the result of replacing $t\theta$ by $t'\theta$ in $x\theta$ at the relevant position. Let $C' = C\theta[x \to v]$.

Case 1: Variable condition 1 fails to hold because D is an extended combinator axiom not compatible with u. By the definition of compatibility, $u = x\,\overline{s}_m$, $t = \boldsymbol{\mathcal{C}}_{\mathsf{any}}\,\overline{x}_n \approx t'$ and $\boldsymbol{\mathcal{C}}_{\mathsf{any}}\,\overline{x}_{n-m}$ is a weak redex. Thus, $x\theta = (\boldsymbol{\mathcal{C}}_{\mathsf{any}}\,\overline{x}_{n-m})\theta$ is also a weak redex. Let $C'' = C\{x \mapsto ((\boldsymbol{\mathcal{C}}_{\mathsf{any}}\,\overline{x}_{n-m})\theta)\downarrow^w\}$. Because the maximal weak reduction from the largest term in $C\theta$ is greater than the maximal weak reduction from the largest term in $C''\theta$, we have $C\theta \succ C''\theta$ and thus $R \models \lfloor C''\theta \rfloor$. But then, as $C\theta \longrightarrow_w C''\theta$ and all members of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ are true in R, we have $R \models \lfloor C\theta \rfloor$ by congruence.

Case 2 selecting only: The superposed subterm $t\theta$ is beneath a variable x in C and $x\theta$ has a first-order head. We have that $x\theta = (\mathsf{f}\,\overline{s}_n)\langle t\theta \rangle$ and $v = (\mathsf{f}\,\overline{s}_n)\langle t'\theta \rangle$. Thus, $x\theta \succ v$ follows from the ordering's compatibility with stable context. As v has a first-order head and C' is formed from $C\theta$ by replacing $x\theta$ with v, $C\theta \succ C'$ follows from property P2 of the ordering. Hence, $R \models \lfloor C' \rfloor$ and $R \models \lfloor C\theta \rfloor$ via Lemma 40 and congruence.

Case 3 non-selecting only: The superposed subterm $t\theta$ is beneath a variable x in C. By Lemma 39, we have that $\|C\theta\| = 0$. From the existence of the inference between $C\theta$ and $D\theta$, it follows that $t\theta$ must occur at a first-order position in $x\theta$. By the definition of stability, the only first-order subterms that are unstable are those that occur beneath fully applied combinators. Since $\|C\theta\| = 0$, all occurrences of $t\theta$ in $x\theta$ must be stable in $C\theta$. Thus, by compatibility with stable contexts and the fact that $t\theta \succ t'\theta$, we have that $C\theta \succ C'$ and thus by assumption on R, $R \models \lfloor C' \rfloor$. Since $C\theta$ and C' only differ at stable positions where one contains $t\theta$ and the other $t'\theta$, Lemma 41 can be used to show that every literal of $C\theta$ is equivalent in R to the corresponding literal of C'. This implies $R \models \lfloor C\theta \rfloor$ by congruence.

Case 4: The superposed subterm $t\theta$ is at (**selecting only:** or beneath) a variable x in C all of whose other occurences are good in C. From the existence of the inference between $C\theta$ and $D\theta$, it follows that $t\theta$ must occur at a first-order and thus stable position in $x\theta$. Likewise, the existence of the inference implies that the $x\theta$ involved in the inference occurs at a stable position within $s\theta$. For all other instances of $x\theta$, the fact that x is good in C implies that $x\theta$ is stable in $C\theta$. Thus, by compatibility with stable contexts and the

fact that $t\theta \succ t'\theta$, we have that $C\theta \succ C'$ and thus by assumption on R, $R \models \lfloor C' \rfloor$. Since $C\theta$ and C' only differ at stable positions where one contains $t\theta$ and the other $t'\theta$, Lemma 41 can be used to show that every literal of $C\theta$ is equivalent in R to the corresponding literal of C'. This implies $R \models \lfloor C\theta \rfloor$ by congruence.

Case 5: The superposed subterm $t\theta$ is at (**selecting only:** or beneath) a variable x in C and $t'\theta$ has a first-order head. We have $t\theta \succ t'\theta$ and C' can be formed from $C\theta$ by replacing all occurrences of $t\theta$ with $t'\theta$. Thus, $C\theta \succ C'$ follows from property P2 of the ordering. Hence, $R \models \lfloor C' \rfloor$.

Note that for all type correct tuple of ground terms $\overline{u}$, if $t\theta\,\overline{u}$ and $t'\theta\,\overline{u}$ are smaller than the maximal term of $C\theta$ then

$$R \models \lfloor t\theta\,\overline{u} \approx t'\theta\,\overline{u} \rfloor \tag{4.1}$$

This can be shown exactly as Bentkamp et al. do [22, Lemma 19]. If $[\neg]s\theta \approx s'\theta$ is the maximal literal in $C\theta$, consider the clause C'' formed by rewriting $\lfloor t\theta \rfloor$ to $\lfloor t'\theta \rfloor$ wherever possible in $\lfloor C\theta \rfloor$. As $t\theta$ is a first-order subterm of the maximal term of $C\theta$, $s\theta$, we have that every term of C'' is smaller than the maximal term of $\lfloor C\theta \rfloor$. Further, we have that $\lceil C'' \rceil$ and C' differ only at positions where $\lceil C'' \rceil$ contains a $t\theta$ and C' contains a $t'\theta$. I prove that C'' can be rewritten to $\lfloor C' \rfloor$ using equalities true in R. Hence $R \models \lfloor C\theta \rfloor$ via congruence.

Let $l \approx r$ be an arbitrary literal of C'' and $l' \approx r'$ be the corresponding literal in $\lfloor C' \rfloor$. Then, since $\lceil C'' \rceil$ and C' only differ where $\lceil C'' \rceil$ contains a $t\theta$ and C' contains a $t'\theta$, we have that $l = \lfloor k[\overline{t\theta}_n] \rfloor$ and $l' = \lfloor k[\overline{t'\theta}_n] \rfloor$ for some context k. I provide an algorithm for rewriting l into l' using equalities true in R. The same can be done for r and r' proving that the literal $l \approx r$ can be rewritten to the literal $l' \approx r'$. Since the literal was chosen arbitrarily, this proves that every literal of C'' can be rewritten to the corresponding literal of $\lfloor C' \rfloor$ in R.

The algorithm is the same as that in Lemma 40. It is repeated here for clarity. Let $l_0 = \lfloor l \rfloor$ and $\tilde{l}_0 = \lfloor l' \rfloor$. Terms $l_1, l_2, l_3, \ldots$ are defined inductively as follows: l_{i+1} is formed from l_i by rewriting all subterms of the form $\lfloor t\theta\,\overline{u} \rfloor$ in l_i to $\lfloor t'\theta\,\overline{u} \rfloor$ and then reducing the outermost leftmost weak redex in the resulting term. Terms $\tilde{l}_1, \tilde{l}_2, \tilde{l}_3, \ldots$ are also defined inductively: $\tilde{l}_{i+1}$ is formed from $\tilde{l}_i$ by reducing the left-most outermost weak redex in $\tilde{l}_i$.

The argument that the algorithm terminates and that l_* and $\tilde{l}_*$ are syntactically identical is the same as for Lemma 40. The reason Lemma 40 cannot be invoked here is that using Lemma 40 would require that $R \models \lfloor t\theta\,\overline{w} \approx t'\theta\,\overline{w} \rfloor$ for every type correct tuple of terms $\overline{w}$. In the current context, this does not hold. It therefore remains to justify the rewrites from each l_i to l_{i+1} in the above algorithm. Both l_0 and $\tilde{l}_0$ are smaller than $\lfloor s\theta \rfloor$. Reducing a leftmost redex of a term results in a smaller term. Likewise, by property P2 of the ordering, rewriting a term of the form $\lfloor t\theta\,\overline{u} \rfloor$ to one of the form $\lfloor t'\theta\,\overline{u} \rfloor$ results in a

smaller term. Therefore, for all $i > 0$, $l_i \prec l_{i-1} \prec \ldots \prec l_0 \prec \lfloor s\theta \rfloor$ justifying the use of Equation 4.1.

Selecting only: If $[\neg]s\theta \approx s'\theta$ is a selected literal, then by the selection criteria x cannot be the head of the maximal term of C. Therefore, the algorithm provided above can be used to rewrite C'' into $\lfloor C' \rfloor$ proving that $R \models C\theta$ by congruence.

Case 6: The superposed subterm $t\theta$ is at (**selecting only:** or beneath) a variable x in C and $t\theta \approx t'\theta$ is a member of $\mathcal{G}_\Sigma(ECA)$. In this case $C\theta \longrightarrow_w C'$ and thus $C\theta \succ C'$ by property P1 of the ordering. Hence, $R \models \lfloor C' \rfloor$. Since all members of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ are true in R, every side of a literal in $C\theta$ is equal in R to the equivalent term in C' and $R \models \lfloor C\theta \rfloor$ follows by congruence. □

Model Construction

In this section the candidate interpretation R_∞ developed in the previous section is shown to be a model of $\lfloor \mathcal{G}_\Sigma(N) \rfloor$. As per the standard proof for first-order logic, this is done by induction on the clause order $\succ$. For some fixed clause $\lfloor C \rfloor \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$, by Lemma 34, we have that all members of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$ are true in $R_{\lfloor C \rfloor}$. By the induction hypothesis, we have that for all clauses $\lfloor D \rfloor \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than C are true in $R_{\lfloor C \rfloor}$. In the induction step, it is shown that this implies that $\lfloor C \rfloor$ is true in $R_{\lfloor C \rfloor}$.

Lemma 43 (R_∞ is a model)**.** *Let $\lfloor C \rfloor \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$, then*

(i) *$E_{\lfloor C \rfloor} = \emptyset$ if and only if $R_{\lfloor C \rfloor} \models C$;*
(ii) *if C is redundant with respect to $\mathcal{G}_\Sigma(N)$ then $\lfloor C \rfloor$ is true in $R_{\lfloor C \rfloor}$;*
(iii) *$\lfloor C \rfloor$ holds in R_∞ and R_D for all $D \in \lfloor \mathcal{G}_\Sigma(N) \rfloor, D \succ C$;*
(iv) ***Selecting only:*** *if C has selected literals, then $R_{\lfloor C \rfloor} \models \lfloor C \rfloor$;*
(v) *If C is a member of $\mathcal{G}_\Sigma(ECA)$, then $R_{\lfloor C \rfloor} \models \lfloor C \rfloor$.*

Proof. The proof proceeds by induction on ground clauses of the floor logic. I assume that (i) to (v) are satisfied by all clauses $D \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$ such that $\lfloor C \rfloor \succ D$. I prove that (i) to (v) hold for C. The $\Rightarrow$ direction of (i) follows from the construction. Part (iii) follows from (i) by Lemmas 32 and 33. Part (v) is a straightforward extension of Lemma 34. Therefore, it remains to prove the $\Leftarrow$ direction of (i) and (ii) and (iv) for the case where $C \notin \mathcal{G}_\Sigma(ECA)$.

Case 1: C is redundant with respect to $\mathcal{G}_\Sigma(N)$. Then $\lfloor C \rfloor$ is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ that are smaller than it and by members of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. By part (iii) of the induction hypothesis and Lemma 34, these clauses are true in $R_{\lfloor C \rfloor}$ and therefore $\lfloor C \rfloor$ is true in $R_{\lfloor C \rfloor}$.

Case 2: $C = C' \vee s \not\approx s'$ and C is not redundant with respect to $\mathcal{G}_\Sigma(N)$.

Case 2.1: $s = s'$. In this case, there is an EQRES inference from $C\theta$:

$$\frac{C' \vee s \not\approx s'}{C'} \text{ EQRES}$$

By Lemma 38, we have that $\lfloor C' \rfloor$ is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than $\lfloor C \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. By part (iii) of the induction hypothesis and Lemma 34, these clauses are true in $R_{\lfloor C \rfloor}$. Therefore, $\lfloor C' \rfloor$ and thus $\lfloor C \rfloor$ are true in $R_{\lfloor C \rfloor}$.

Case 2.2: $s \succ s'$. If $R_{\lfloor C \rfloor} \models \lfloor s \not\approx s' \rfloor$ then the lemma follows. Therefore, assume that it doesn't hold and $\lfloor s \rfloor \downarrow_{R_{\lfloor C \rfloor}} \lfloor s' \rfloor$. There must exist some rule in $R_{\lfloor C \rfloor}$ which reduces s. Such a rule must have been produced by some clause $\lfloor D \rfloor = \lfloor D' \vee t \approx t' \rfloor$. Then there exists the following ground SUP inference between C and D:

$$\frac{D' \vee t \approx t' \qquad C' \vee s\langle t \rangle \not\approx s'}{C' \vee D' \vee s\langle t' \rangle \not\approx s'} \text{ SUP}$$

By Lemma 42, if the inference is non-liftable it is redundant with respect to $\mathcal{G}_\Sigma(N)$ and therefore the floor of its conclusion is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than $\lfloor C \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. If the inference is liftable, then by Lemma 38, we again have that the floor of the conclusion is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than $\lfloor C \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. By part (iii) of the induction hypothesis and Lemma 34, all these clauses are true in $R_{\lfloor C \rfloor}$. Therefore, $\lfloor C' \vee D' \vee s\langle t' \rangle \not\approx s' \rfloor$ holds in $R_{\lfloor C \rfloor}$. Since $\lfloor D' \rfloor$ is false in $R_{\lfloor C \rfloor}$, either $R_{\lfloor C \rfloor} \models \lfloor C' \rfloor$ or $R_{\lfloor C \rfloor} \models \lfloor s\langle t' \rangle \not\approx s' \rfloor$. In the latter case, $R_{\lfloor C \rfloor} \models \lfloor s\langle t \rangle \not\approx s' \rfloor$ because $\lfloor t \rfloor \to \lfloor t' \rfloor \in R_{\lfloor C \rfloor}$. Thus, in both cases $R_{\lfloor C \rfloor} \models \lfloor C \rfloor$.

Case 3: C is not redundant and contains no negative literals. In this case $C = C' \vee s \approx s'$. If $E_{\lfloor C \rfloor} = \{\lfloor s \rfloor \to \lfloor s' \rfloor\}$ or $R_{\lfloor C \rfloor} \models C'$ or $s = s'$ then $R_{\lfloor C \rfloor} \models C$. Therefore, assume $E_{\lfloor C \rfloor} = \emptyset$, $s \neq s'$ and C' is false in $R_{\lfloor C \rfloor}$.

Case 3.1: $s \approx s'$ is not strictly maximal in C. In this case, C can be written as $C'' \vee t \approx t' \vee s \approx s'$ where $t = s$ and $t' = s'$. Then there is an EQFACT inference from C:

$$\frac{C'' \vee t \approx t' \vee s \approx s'}{C'' \vee t' \not\approx s' \vee t \approx t'} \text{ EQFACT}$$

By Lemma 38, the floor of the conclusion of the inference is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than $\lfloor C \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. Therefore, by part (iii) of the induction hypothesis and Lemma 34, $\lfloor C'' \vee t' \not\approx s' \vee t \approx t' \rfloor$ is true in $R_{\lfloor C \rfloor}$. Since $t' = s'$, $\lfloor t' \not\approx s' \rfloor$ is false in $R_{\lfloor C \rfloor}$. Therefore $\lfloor t \approx t' \rfloor$ is true in $R_{\lfloor C \rfloor}$ and hence $\lfloor C \rfloor$ is true in $R_{\lfloor C \rfloor}$.

Case 3.2: The literal $s \approx s'$ is strictly maximal and $\lfloor s \rfloor$ is reducible by some rule in $R_{\lfloor C \rfloor}$. This rule must be produced by a clause $\lfloor D \rfloor = \lfloor D' \vee t \approx t' \rfloor$. There exists the following ground SUP inference between C and D:

$$\frac{D' \vee t \approx t' \qquad C' \vee s\langle t \rangle \approx s'}{C' \vee D' \vee s\langle t' \rangle \approx s'} \text{ SUP}$$

By Lemma 42, if the inference is non-liftable it is redundant with respect to $\mathcal{G}_\Sigma(N)$ and therefore the floor of its conclusion is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than $\lfloor C \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. If the inference is liftable, then by Lemma 38, we again have that the floor of the conclusion is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than $\lfloor C \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. By part (iii) of the induction hypothesis and Lemma 34, all these clauses are true in $R_{\lfloor C \rfloor}$. Therefore, $\lfloor C' \vee D' \vee s\langle t' \rangle \not\approx s' \rfloor$ holds in $R_{\lfloor C \rfloor}$. Since $\lfloor D' \rfloor$ is false in $R_{\lfloor C \rfloor}$, either $R_{\lfloor C \rfloor} \models \lfloor C' \rfloor$ or $R_{\lfloor C \rfloor} \models \lfloor s\langle t' \rangle \not\approx s' \rfloor$. In the latter case, $R_{\lfloor C \rfloor} \models \lfloor s\langle t \rangle \not\approx s' \rfloor$ because $\lfloor t \rfloor \to \lfloor t' \rfloor \in R_{\lfloor C \rfloor}$. Thus, in both cases $R_{\lfloor C \rfloor} \models \lfloor C \rfloor$.

Case 3.3: $s \approx s'$ is strictly maximal and $\lfloor s \rfloor$ is not reducible by any rule in $R_{\lfloor C \rfloor}$. By assumption we have that $E_{\lfloor C \rfloor} = \emptyset$. Assume that $\lfloor C \rfloor$ is false in $R_{\lfloor C \rfloor}$. By the construction of $E_{\lfloor C \rfloor}$, it must be the case that $\lfloor C' \rfloor$ is true in $R_{\lfloor C \rfloor} \cup \{\lfloor s \rfloor \to \lfloor s' \rfloor\}$. Thus, C' must have the form $C'' \vee t \approx t'$ where the literal $\lfloor t \approx t' \rfloor$ is true in $R_{\lfloor C \rfloor} \cup \{\lfloor s \rfloor \to \lfloor s' \rfloor\}$, but not in $R_{\lfloor C \rfloor}$. By the confluence of $R_{\lfloor C \rfloor}$, this is equivalent to saying $\lfloor t \rfloor \downarrow_{R_{\lfloor C \rfloor} \cup \{\lfloor s \rfloor \to \lfloor s' \rfloor\}} \lfloor t' \rfloor$, but not $\lfloor t \rfloor \downarrow_{R_{\lfloor C \rfloor}} \lfloor t' \rfloor$. Therefore, the rule $\lfloor s \rfloor \to \lfloor s' \rfloor$ must be used at least once in rewriting either $\lfloor t \rfloor$ or $\lfloor t' \rfloor$ to a normal form. As $s \succ s'$, $t \succ t'$ and $s \approx s' \succ t \approx t'$, we have that $s \succ t'$ and $s \succeq t$. But the fact that $\lfloor s \rfloor \to \lfloor s' \rfloor$ is used in the rewrite proof implies that $s = t$ and that the rewrite proof looks like this: $\lfloor t \rfloor \to \lfloor s' \rfloor \to^* u \,{}^*\!\leftarrow \lfloor t' \rfloor$. Hence, we have that $R_{\lfloor C \rfloor} \models \lfloor s' \approx t' \rfloor$. Now consider the following EQFACT inference from C:

$$\frac{C'' \vee t \approx t' \vee s \approx s'}{C'' \vee t' \not\approx s' \vee t \approx t'} \text{ EQFACT}$$

By Lemma 38, the floor of the conclusion of the inference is entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ smaller than $\lfloor C \rfloor$ and clauses in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor$. Therefore, by part (iii) of the induction hypothesis and Lemma 34, $\lfloor C'' \vee t' \not\approx s' \vee t \approx t' \rfloor$ is true in $R_{\lfloor C \rfloor}$. Since $t' = s'$, $\lfloor t' \not\approx s' \rfloor$ is false in $R_{\lfloor C \rfloor}$. Therefore $\lfloor t \approx t' \rfloor$ is true in $R_{\lfloor C \rfloor}$ and hence $\lfloor C \rfloor$ is true in $R_{\lfloor C \rfloor}$. □

4.4.4 | Higher-Order Model Construction

In this section I lift the model R_∞ of $\lfloor \mathcal{G}_\Sigma(N) \rfloor$, to a applicative first-order interpretation $R_\infty^\uparrow$. I then show that $R_\infty^\uparrow$ is a model of $\mathcal{G}_\Sigma(N)$. I use the notation $t \sim t'$ as a shorthand for $t \sim_{R_\infty} t'$ which, as explained, is equivalent to $R_\infty \models \lfloor t \approx t' \rfloor$.

Lemma 44. *Let t and t' be ground ceiling terms such that $\lfloor t \rfloor \to \lfloor t' \rfloor$ is a rule in R_∞. Then, for all type correct tuple of terms $\overline{u}$, $t\,\overline{u} \sim t'\,\overline{u}$.*

Proof. The rule $\lfloor t \rfloor \to \lfloor t' \rfloor$ must stem from a productive clause of the form $\lfloor C\theta \rfloor \approx \lfloor C'\theta \vee t_1\theta \approx t_2\theta \rfloor$ where $t_1\theta \approx t$ and $t_2\theta \approx t'$. By the definition of a productive clause and part (iv) of Lemma 43, $t \approx t'$ is strictly eligible in $C\theta$ and therefore $t_1 \approx t_2$ is strictly eligible in C. Further, t and t' are of functional type, so t_1 and t_2 must be of functional

or polymorphic type. Thus, there is an ARGCONG inference from C with conclusions $(C' \vee t_1\,\overline{x}_n \approx t_2\,\overline{x}_n)\sigma$ for all possible n. Let these conclusions be called $E_1 \ldots E_n$.

By part (ii) of Lemma 43, $C\theta$ is not in $Red_C(\mathcal{G}_\Sigma(N))$ and therefore C is not in $Red_C(N)$. Thus, E_i is either in N or $Red_C(N)$ for $i \leq i \leq n$. The ground instance of $\lfloor E_i \rfloor$ for any i, $\lfloor C'\theta \vee t\,\overline{u}_i \approx t'\,\overline{u}_i \rfloor$ is thus either in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$ or entailed by clauses in $\lfloor \mathcal{G}_\Sigma(N) \rfloor$. Therefore it is true in R_∞. By Lemma 33, we have that $\lfloor C'\theta \rfloor$ is false in R_∞ which implies that $\lfloor t\,\overline{u}_i \approx t'\,\overline{u}_i \rfloor$ must be true. □

Lemma 45. *Let t and t' be ground ceiling terms such that $t \sim t'$ in a single step, but neither $\lfloor t \rfloor \to \lfloor t' \rfloor$ nor $\lfloor t' \rfloor \to \lfloor t \rfloor$ is a rule of R_∞. Let u be a ceiling ground term of the relevant type. Then, $t\,u \sim t'\,u$.*

Proof. It must be the case that there is a single position at which $\lfloor t \rfloor$ and $\lfloor t' \rfloor$ differ. Let v be the subterm in t at this position and v' be the subterm in t' at the same position. It must also be the case that $\lfloor v \rfloor \to \lfloor v' \rfloor$ or $\lfloor v' \rfloor \to \lfloor v \rfloor$ is a rule in R_∞. Without loss of generality, assume that it is the first. By Lemma 44, we have that $v\,\overline{u} \sim v'\,\overline{u}$ for every type-correct tuple of terms u. Now, Lemma 40 can be invoked with $R = R_\infty$ to show that $t\,u \sim t'\,u$ since $t\,u$ and $t'\,u$ only differ at a position where one contains v and the other v'. □

Lemma 46. *For ground ceiling terms, t, t' and u, if $t \sim t'$ then $t\,u \sim t'\,u$.*

Proof. The proof proceeds by induction on the number of rewrite steps between $\lfloor t \rfloor$ and $\lfloor t' \rfloor$. If $t = t'$ then the Lemma follows trivially. Let the number of rewrite steps between $\lfloor t \rfloor$ and $\lfloor t' \rfloor$ be n. Let t'' be the term such that $\lfloor t' \rfloor$ rewrites to $\lfloor t'' \rfloor$ in $n-1$ steps and $\lfloor t'' \rfloor$ rewrites to $\lfloor t \rfloor$ in a single step. By the induction hypothesis we have that $t'\,u \sim t''\,u$. Thus, if it can be shown that $t''\,u \sim t\,u$ the Lemma follows by the transitivity of $\sim$. $t''\,u \sim t\,u$ follows from either Lemma 44 or 45 depending on whether the rewrite between t'' and t takes place at the top level or not completing the proof. □

Lemma 47. *For ground ceiling terms t, t', u, u', if $t \sim t'$ and $u \sim u'$, then $t\,u \sim t'\,u'$ if neither t nor t' is of the form $\boldsymbol{\mathcal{C}_3}\,\overline{s}_{n>1}$, $\mathbf{K}\,\overline{s}_{n>0}$ or $\mathbf{I}\,\overline{s}_n$.*

Proof. By Lemma 46, we have that $t'\,u' \sim t\,u'$, so if it can be shown that $t\,u' \sim t\,u$ the Lemma follows immediately by the transitivity of $\sim$. We have that $\lfloor t \rfloor = \zeta_n(\overline{s}_n)$. By the condition on the form of t and t', $t\,u'$ cannot have a fully applied combinator as its head symbol. Therefore, $\lfloor t\,u' \rfloor = \zeta_{n+1}(\overline{s}_n, u')$. It is obvious that any rewrite steps from $\lfloor u' \rfloor$ can be carried out from $\lfloor t\,u' \rfloor$ and therefore $t\,u' \sim t\,u$. □

Lemma 48. *For ground ceiling terms u and u' such that $u \sim u'$, $\boldsymbol{\mathcal{C}_3}\,t_1\,t_2\,u \sim \boldsymbol{\mathcal{C}_3}\,t_1\,t_2\,u'$ and $\mathbf{K}\,t\,u \sim \mathbf{K}\,t\,u'$ and $\mathbf{I}\,u \sim \mathbf{I}\,u'$.*

Proof. By multiple applications of Lemma 46, we have that $\lfloor u\,\overline{s}\rfloor \approx \lfloor u'\,\overline{s}\rfloor$ holds in R_∞ for all type correct tuple of terms $\overline{s}$. We also have that every member of $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$ holds in R_∞. Thus, the lemma follows by an appeal to Lemma 40 with $R = R_\infty$. □

Lemma 49. *For all ground ceiling terms t, t', u and u', if $t \sim t'$ and $u \sim u'$, then $t\,u \sim t'\,u'$.*

Proof. Proof is by induction on $\|t\| + \|u\| + \|t'\| + \|u'\|$. the base case splits into two cases.

Case 1: Neither t nor t' is of the form $\boldsymbol{\mathcal{C}_3}\,t_1\,t_2$, $\mathbf{K}\,t$ or $\mathbf{I}$. Then the proof follows by an application of Lemma 47.
Case 2: One or both of t and t' are of the form $\boldsymbol{\mathcal{C}_3}\,t_1\,t_2$, $\mathbf{K}\,t$ or $\mathbf{I}$. Without loss of generality, assume that t is of the form $\boldsymbol{\mathcal{C}_3}\,t_1\,t_2$. By Lemma 46, $t'\,u' \sim t\,u'$. Thus, if it can be proven that $t\,u' = \boldsymbol{\mathcal{C}_3}\,t_1\,t_2\,u' \sim \boldsymbol{\mathcal{C}_3}\,t_1\,t_2\,u = t\,u$, the theorem follows by the transitivity of $\sim$. By Lemma 48, $\boldsymbol{\mathcal{C}_3}\,t_1\,t_2\,u' \sim \boldsymbol{\mathcal{C}_3}\,t_1\,t_2\,u$ completing the base case.

For the inductive case, one or more of $\|t\|, \|u\|, \|t'\|$ or $\|u'\|$ is greater than 0. We show that the theorem holds for the first two cases. The latter two can be proved in a like manner.

Case 1: $\|t\| > 0$. Let $t'' = (t)\downarrow^w$. Since R_∞ is a model of $\lfloor \mathcal{G}_\Sigma(N)\rfloor$, it also models $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$. Thus, $t \sim t''$ and by Lemma 46, $t\,u \sim t''\,u$. Since $\|t''\| < \|t\|$, the induction hypothesis can be used to conclude that $t''\,u \sim t'\,u'$. By the transitivity of $\sim$, $t\,u \sim t'\,u'$ follows.
Case 2: $\|u\| > 0$. Since R_∞ models $\lfloor \mathcal{G}_\Sigma(ECA)\rfloor$, we have $u \sim u_2$ where $u_2 = (u)\downarrow^w$. By the induction hypothesis, we have $t\,u_2 \sim t'\,u'$. Either Lemma 47 or Lemma 48 is applicable to prove $t\,u \sim t\,u_2$. The theorem follows by the transitivity of $\sim$. □

Definition 44. I define an interpretation $R^\uparrow_\infty = (\mathcal{I}^\uparrow_{\mathsf{ty}}, \mathcal{E}^\uparrow, \mathcal{J}^\uparrow)$ in the ceiling logic as follows. Let $R_\infty = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J})$ and $\mathcal{I}_{\mathsf{ty}} = (\mathcal{U}, \mathcal{J}_{\mathsf{ty}})$. Let $\mathcal{U}^\uparrow_\tau = \mathcal{U}_{\lfloor\tau\rfloor}$ and $\mathcal{J}^\uparrow_{\mathsf{ty}}(\kappa)(\mathcal{U}_{\tau_1}, \ldots, \mathcal{U}_{\tau_n}) = \mathcal{J}_{\mathsf{ty}}(\lfloor\kappa(\tau_1, \ldots, \tau_n)\rfloor)$. Furthermore, let $\mathcal{J}^\uparrow(\mathsf{f}, \mathcal{U}_{\tau_1}, \ldots, \mathcal{U}_{\tau_n}) = \mathcal{J}(\mathsf{f}_0^{\overline{\tau}_n})$. Let $\mathsf{a} \in \mathcal{U}_{\tau_1\to\tau_2}$ and $\mathsf{b} \in \mathcal{U}_{\tau_1}$. I define $\mathcal{E}^\uparrow$ as:

$$\mathcal{E}^\uparrow_{\mathcal{U}_{\tau_1},\mathcal{U}_{\tau_2}}(\mathsf{a})(\mathsf{b}) = [\![\lfloor s\,t\rfloor]\!]^\xi_{R_\infty}$$

Where s is a ground term of type $\tau_1 \to \tau_2$ and t is a ground term of type τ_1, $[\![\lfloor s\rfloor]\!]^\xi_{R_\infty} = \mathsf{a}$ and $[\![\lfloor t\rfloor]\!]^\xi_{R_\infty} = \mathsf{b}$. The existence of the terms s and t is guaranteed by the fact that R_∞ is term-generated.

This interpretation is well defined if the definition of $\mathcal{E}^\uparrow$ does not depend on the choice of the ground terms s and t. To show this, I assume that there exists other ground ceiling terms s' and t' such that $[\![\lfloor s'\rfloor]\!]^\xi_{R_\infty} = \mathsf{a}$ and $[\![\lfloor t'\rfloor]\!]^\xi_{R_\infty} = \mathsf{b}$. By Lemma 49, it follows from

$[\![\lfloor s \rfloor]\!]^{\xi}_{R^{\uparrow}_{\infty}} = [\![\lfloor s' \rfloor]\!]^{\xi}_{R^{\uparrow}_{\infty}}$ and $[\![\lfloor t \rfloor]\!]^{\xi}_{R^{\uparrow}_{\infty}} = [\![\lfloor t' \rfloor]\!]^{\xi}_{R^{\uparrow}_{\infty}}$ that

$$[\![\lfloor s\, t \rfloor]\!]^{\xi}_{R^{\uparrow}_{\infty}} = [\![\lfloor s'\, t' \rfloor]\!]^{\xi}_{R^{\uparrow}_{\infty}}$$

indicating that the definition of $\mathcal{E}^{\uparrow}$ is independent of the choice of s and t.

Lemma 50 (Substitution Lemma)**.** *For all ceiling logic terms t and types τ and for all grounding substitutions ρ, $[\![t\rho]\!]_{R^{\uparrow}_{\infty}} = [\![t]\!]^{\xi}_{R^{\uparrow}_{\infty}}$ and $[\![\tau\rho]\!]_{R^{\uparrow}_{\infty}} = [\![\tau]\!]^{\xi}_{R^{\uparrow}_{\infty}}$ if $\xi(\alpha) = [\![\alpha\rho]\!]_{R^{\uparrow}_{\infty}}$ for all α and $\xi(x) = [\![x\rho]\!]_{R^{\uparrow}_{\infty}}$ for all x.*

Proof. I first prove the lemma for types by induction on the structure of τ. If τ is a variable σ, then $[\![\sigma]\!]^{\xi}_{R^{\uparrow}_{\infty}} = \xi(\sigma) = [\![\sigma\rho]\!]_{R^{\uparrow}_{\infty}}$. If $\tau = \kappa(\overline{\tau}_n)$, then:

$$[\![\tau]\!]^{\xi}_{R^{\uparrow}_{\infty}} = \mathcal{J}^{\uparrow}_{\mathsf{ty}}(\kappa)([\![\tau_1]\!]^{\xi}_{R^{\uparrow}_{\infty}}, \ldots, [\![\tau_n]\!]^{\xi}_{R^{\uparrow}_{\infty}}) \overset{IH}{=} \mathcal{J}^{\uparrow}_{\mathsf{ty}}(\kappa)([\![\tau_1\rho]\!]_{R^{\uparrow}_{\infty}}, \ldots, [\![\tau_n\rho]\!]_{R^{\uparrow}_{\infty}}) = [\![\kappa(\overline{\tau})\rho]\!]_{R^{\uparrow}_{\infty}}$$

Now I prove the lemma for terms by induction on the structure of t. If t is a variable x, then $[\![x]\!]^{\xi}_{R^{\uparrow}_{\infty}} = \xi(x) = [\![x\rho]\!]_{R^{\uparrow}_{\infty}}$. If t is of the form $\mathsf{f}\langle\overline{\tau}\rangle$, then:

$$[\![t]\!]^{\xi}_{R^{\uparrow}_{\infty}} = \mathcal{J}^{\uparrow}(\mathsf{f}, [\![\overline{\tau}]\!]^{\xi}_{R^{\uparrow}_{\infty}}) = \mathcal{J}^{\uparrow}(\mathsf{f}, [\![\overline{\tau}\rho]\!]_{R^{\uparrow}_{\infty}}) = [\![\mathsf{f}\langle\overline{\tau}\rangle\rho]\!]_{R^{\uparrow}_{\infty}}$$

Finally, if t is an application of the form $t_1\, t_2$, where $t_1 : \tau_1 \to \tau_2$ and $t_2 : \tau_1$, then:

$$\begin{aligned}
[\![t_1\, t_2]\!]^{\xi}_{R^{\uparrow}_{\infty}} &= \mathcal{E}^{\uparrow}_{\mathcal{U}_{\tau_1}, \mathcal{U}_{\tau_2}}([\![t_1]\!]^{\xi}_{R^{\uparrow}_{\infty}})([\![t_2]\!]^{\xi}_{R^{\uparrow}_{\infty}}) \\
&\overset{\text{IH}}{=} \mathcal{E}^{\uparrow}_{\mathcal{U}_{\tau_1}, \mathcal{U}_{\tau_2}}([\![t_1\rho]\!]_{R^{\uparrow}_{\infty}})([\![t_2\rho]\!]_{R^{\uparrow}_{\infty}}) \\
&= [\![t_1\rho\, t_2\rho]\!]_{R^{\uparrow}_{\infty}} \\
&= [\![(t_1\, t_2)\rho]\!]_{R^{\uparrow}_{\infty}}
\end{aligned}$$

□

Lemma 51 (Model transfer to ceiling logic)**.** *$R^{\uparrow}_{\infty}$ is a term-generated model of $\mathcal{G}_{\Sigma}(N)$.*

Proof. By induction on the structure of ground terms t of the ceiling logic, it is shown that $[\![t]\!]^{\xi}_{R^{\uparrow}_{\infty}} = [\![\lfloor t \rfloor]\!]^{\xi}_{R_{\infty}}$. Let t be a ground ceiling term. If t is of the form $\mathsf{f}\langle\overline{\tau}\rangle$, then $[\![t]\!]^{\xi}_{R^{\uparrow}_{\infty}} = \mathcal{J}^{\uparrow}(\mathsf{f}, [\![\overline{\tau}]\!]_{R^{\uparrow}_{\infty}}) = \mathcal{J}(\mathsf{f}^{\overline{\tau}}_0) = [\![\lfloor t \rfloor]\!]^{\xi}_{R_{\infty}}$

If t is an application $t = t_1\, t_2$, where t_1 is of type $\tau_1 \to \tau_2$, then we have:

$$\begin{aligned}
[\![t_1\, t_2]\!]^{\xi}_{R^{\uparrow}_{\infty}} &= \mathcal{E}^{\uparrow}_{\mathcal{U}_{\tau_1}, \mathcal{U}_{\tau_2}}([\![t_1]\!]^{\xi}_{R^{\uparrow}_{\infty}})([\![t_2]\!]^{\xi}_{R^{\uparrow}_{\infty}}) \\
&\overset{\text{IH}}{=} \mathcal{E}^{\uparrow}_{\mathcal{U}_{\tau_1}, \mathcal{U}_{\tau_2}}([\![\lfloor t_1 \rfloor]\!]^{\xi}_{R_{\infty}})([\![\lfloor t_2 \rfloor]\!]^{\xi}_{R_{\infty}}) \\
&= [\![\lfloor t_1\, t_2 \rfloor]\!]^{\xi}_{R_{\infty}} && \text{(By definition of } \mathcal{E}^{\uparrow}\text{)}
\end{aligned}$$

I have shown that $[\![t]\!]^{\xi}_{R^{\uparrow}_{\infty}} = [\![\lfloor t \rfloor]\!]^{\xi}_{R_{\infty}}$ for all ground ceiling logic terms t. It follows that a ground literal $s \approx t$ or the negation of a ground literal $s \not\approx t$ is true in $R^{\uparrow}_{\infty}$ if and

only if $\lfloor s \approx t \rfloor$ or $\lfloor s \not\approx t \rfloor$ is true in R_∞. Hence, a ground clause C is true in $R_\infty^\uparrow$ if and only if $\lfloor C \rfloor$ is true in R_∞.

By Lemma 43, R_∞ is a model of $\lfloor \mathcal{G}_\Sigma(N) \rfloor$, i.e., all clauses $\lfloor C \rfloor \in \lfloor \mathcal{G}_\Sigma(N) \rfloor$ are true in R_∞. Hence, all clauses $C \in \mathcal{G}_\Sigma(N)$ are true in $R_\infty^\uparrow$ and therefore $R_\infty^\uparrow$ is a model of $\mathcal{G}_\Sigma(N)$.

To show that $R_\infty^\uparrow$ is term-generated, let a be an arbitrary member of $\mathcal{U}_\tau^\uparrow$. Since $\mathcal{U}_\tau^\uparrow = \mathcal{U}_{\lfloor \tau \rfloor}$, we have that $\mathsf{a} \in \mathcal{U}_{\lfloor \tau \rfloor}$. Since R_∞ is term-generated, there must exist a floor ground term t such that $[\![t]\!]_{R_\infty}^\xi = \mathsf{a}$. We have $[\![\lceil t \rceil]\!]_{R_\infty^\uparrow}^\xi = [\![t]\!]_{R_\infty}^\xi = \mathsf{a}$, since I have just shown $[\![u]\!]_{R_\infty^\uparrow}^\xi = [\![\lfloor u \rfloor]\!]_{R_\infty}^\xi$ for all ground ceiling terms u. Hence, $R_\infty^\uparrow$ is term-generated. □

Lemma 52 (Model transfer)**.** *Let N be a set of ceiling clauses. For a model $\mathcal{I}$, such that $\mathcal{I} \models \mathcal{G}_\Sigma(N)$, $\mathcal{I} \models N$.*

Proof. From the interpretation $\mathcal{I}$, a term-generated interpretation $\mathcal{I}'$ can be formed such that $\mathcal{I}' \models \mathcal{G}_\Sigma(N)$. This can be achieved by removing from $\mathcal{I}$ all universes which are not the denotation of any type in $\mathcal{T}ya_{\Sigma_{\mathsf{ty}}}(\emptyset)$ and by removing, from the remaining universes, all members which are not the denotation of a term in $\mathcal{T}_\Sigma(\emptyset)$.

I need to show that for all clauses $C \in N$, C holds in $\mathcal{I}'$ for all ξ. Since $\mathcal{I}'$ is term-generated, we have that for all variables x in C, there exists a ground term s_x such that $\xi(x) = [\![s_x]\!]_{R_\infty^\uparrow}$. Let ρ be a substitution that maps each term variable x in C to s_x, and each type variable α in C, such that $\xi(\alpha) = \mathcal{U}_\tau$, to τ. Then for any term variable x in C, $[\![x\rho]\!]_{\mathcal{I}'} = \xi(x)$ and for any type variable α in C, $[\![\alpha\rho]\!]_{\mathcal{I}'} = \xi(\alpha)$. Then by Lemma 50, $[\![C]\!]_{\mathcal{I}'}^\xi = [\![C\rho]\!]_{\mathcal{I}'}$. As $C\rho$ is ground, it is a member of $\mathcal{G}_\Sigma(N)$ and therefore true in $\mathcal{I}'$. Thus so is C. □

Lemma 53 (Model transfer)**.** *$R_\infty^\uparrow$ is a model of N.*

Proof. By Lemma 51 $R_\infty^\uparrow$ is a model of $\mathcal{G}_\Sigma(N)$. By Lemma 52, $R_\infty^\uparrow$ is thus a model of N. □

Corollary 4. *$R_\infty^\uparrow$ is a combinatory model.*

Proof. Since N contains the combinator axioms and $R_\infty^\uparrow$ models N, it must model the combinator axioms. It is thus a combinatory model. □

Lemma 54 (Extensional model)**.** *If the extensionality axiom is present in N, then $R_\infty^\uparrow$ is an extensional model.*

Proof. The proof is the same as that of Bentkamp et al. [19] □

Now, static refutational completeness can be proven.

Theorem 12. *For a set of clauses N saturated to redundancy by clausal combinatory superposition (select or no-select), N has a model iff it does not contain $\bot$. Moreover, if N contains the extensionality axiom, the model is extensional.*

4.4.5 | Dynamic Refutational Completeness

From static refutation completeness, I use Lemma 6 of Waldmann et al.'s framework [124] to prove dynamic refutational completeness. In order for Lemma 6 to be applicable, I need to prove four properties regarding the redundancy criterion. It is also necessary to show that the inference system is an inference system in the sense of the framework and that the entailment relation is an entailment relation in the sense of the framework. However, these are straightforward, so I do not bother proving them here. Prior to proving that the redundancy criterion is a redundancy criterion in the sense of the framework, I prove a number of useful lemmas.

Lemma 55. *For an applicative interpretation $\mathcal{I}$ and a clause C, if $\mathcal{I} \models C$, then $\mathcal{I} \models C\theta$ for every ground substitution θ.*

Proof. Define a valuation function ξ such that for any term variable x in C, $\xi(x) = [\![x\theta]\!]_{\mathcal{I}}$ and for any type variable α, $\xi(\alpha) = [\![\alpha\theta]\!]_{\mathcal{I}}$. Then by Lemma, 50, for any subterm t of C, $[\![t]\!]^{\xi}_{\mathcal{I}} = [\![t\theta]\!]_{\mathcal{I}}$. By the definition of truth of a clause, this implies that $\mathcal{I} \models C\theta$ if and only if C is true in $\mathcal{I}$ for ξ. Since C is true in $\mathcal{I}$ for all valuation functions, it must be true in $\mathcal{I}$ for ξ and therefore we can conclude $\mathcal{I} \models C\theta$. □

Lemma 56. *Given a combinatory interpretation $\mathcal{I}$, there exists a monomorphic interpretation $\mathcal{I}'$ such that for all ground clauses C, $\mathcal{I} \models C$ iff $\mathcal{I}' \models \lfloor \mathcal{G}_{\Sigma}(ECA) \rfloor \cup \lfloor C \rfloor$.*

Proof. Let $\mathcal{I} = (\mathcal{U}, \mathcal{J}_{\mathsf{ty}}, \mathcal{J}, \mathcal{E})$. Let $\mathcal{I}' = (\mathcal{U}^{\downarrow}, \mathcal{J}^{\downarrow}_{\mathsf{ty}}, \mathcal{J}^{\downarrow})$. In order to define $\mathcal{U}^{\downarrow}$, let $\mathcal{U}^{\downarrow}_{\tau} = [\![\lceil \tau \rceil]\!]_{\mathcal{I}_{\mathsf{ty}}}$. Let $\mathcal{J}^{\downarrow}_{\mathsf{ty}}(\tau) = \mathcal{U}^{\downarrow}_{\tau}$. Finally to define $\mathcal{J}^{\downarrow}$, let $\overline{\mathsf{a}}_n$ be a tuple of universe elements:

$$\begin{aligned}
\mathcal{J}^{\downarrow}(\mathsf{s}_{(\boldsymbol{\mathcal{C}}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s})}) &= [\![\boldsymbol{\mathcal{C}}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s}]\!]_{\mathcal{I}} \\
\mathcal{J}^{\downarrow}(\mathsf{f}^{\overline{\tau}}_{n})(\overline{\mathsf{a}}_n) &= [\![\mathsf{f}\langle\overline{\tau}\rangle\,\overline{x}_n]\!]^{\{\overline{x}\to\overline{\mathsf{a}}\}}_{\mathcal{I}} \\
\mathcal{J}^{\downarrow}(\boldsymbol{\mathcal{C}}^{\overline{\tau}}_{n})(\overline{\mathsf{a}}_n) &= [\![\boldsymbol{\mathcal{C}}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{x}_n]\!]^{\{\overline{x}\to\overline{\mathsf{a}}\}}_{\mathcal{I}}
\end{aligned}$$

To show that the truth value of C in $\mathcal{I}$ is the same as that of $\lfloor C \rfloor$ in $\mathcal{I}'$ it suffices to show that for any term $t \in \mathcal{T}_{\Sigma}(\emptyset)$, $[\![t]\!]_{\mathcal{I}} = [\![\lfloor t \rfloor]\!]_{\mathcal{I}'}$. I do this by induction on the structure of t. If t is a weak redex, then:

$$[\![\lfloor t \rfloor]\!]_{\mathcal{I}'} = [\![\lfloor \boldsymbol{\mathcal{C}}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s} \rfloor]\!]_{\mathcal{I}'} = [\![\mathsf{s}_{(\boldsymbol{\mathcal{C}}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s})}]\!]_{\mathcal{I}'} = [\![\boldsymbol{\mathcal{C}}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s}]\!]_{\mathcal{I}} = [\![t]\!]_{\mathcal{I}}$$

If t is of the form $\mathsf{f}\langle\overline{\tau}\rangle\,\overline{s}_n$ or $\boldsymbol{\mathcal{C}}_{\mathsf{any}}\langle\overline{\tau}\rangle\,\overline{s}$, then:

$$\begin{aligned}
[\![\lfloor t \rfloor]\!]_{\mathcal{I}'} &= \mathcal{J}^{\downarrow}(\mathsf{f}^{\overline{\tau}}_{n})([\![\lfloor s_1 \rfloor]\!]_{\mathcal{I}'}, \ldots, [\![\lfloor s_n \rfloor]\!]_{\mathcal{I}'}) \\
&\overset{IH}{=} \mathcal{J}^{\downarrow}(\mathsf{f}^{\overline{\tau}}_{n})([\![s_1]\!]_{\mathcal{I}}, \ldots, [\![s_n]\!]_{\mathcal{I}}) \\
&= [\![\mathsf{f}\langle\overline{\tau}\rangle\,\overline{x}_n]\!]^{\{x_1\to[\![s_1]\!]_{\mathcal{I}}, x_2\to[\![s_2]\!]_{\mathcal{I}}, \ldots\}}_{\mathcal{I}}
\end{aligned}$$

$$= [\![\mathsf{f}\langle\overline{\tau}\rangle\ \overline{s}_n]\!]_{\mathcal{I}} \qquad \text{(By Lemma 50)}$$
$$= [\![t]\!]_{\mathcal{I}}$$

Finally, I prove that for any A in $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor, \mathcal{I}' \models A$. A must be of the form $\mathsf{s}_{(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})} \approx \lfloor (\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})\downarrow^w \rfloor$. By the previous part of the proof, we have that $[\![\mathsf{s}_{(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})}]\!]_{\mathcal{I}'} = [\![\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s}]\!]_{\mathcal{I}}$ and $[\![\lfloor (\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})\downarrow^w \rfloor]\!]_{\mathcal{I}'} = [\![(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})\downarrow^w]\!]_{\mathcal{I}}$. Since $\mathcal{I}$ is a combinatory interpretation, we have that $[\![\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s}]\!]_{\mathcal{I}} = [\![(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})\downarrow^w]\!]_{\mathcal{I}}$. Thus, we can conclude that $[\![\mathsf{s}_{(\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})}]\!]_{\mathcal{I}'} = [\![\lfloor (\mathcal{C}_{\mathsf{any}}\langle\overline{\tau}\rangle\ \overline{s})\downarrow^w \rfloor]\!]_{\mathcal{I}'}$. □

Lemma 57. *For a set of clauses N, $\mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N)) \subseteq \mathcal{G}_\Sigma(N \setminus Red_C(N))$.*

Proof. Assume that $D \in \mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N))$. There must exist a clause $C \in N$ such that $D \in G_\Sigma(C)$. If $C \in Red_C(N)$, then by definition, $D \in Red_C(\mathcal{G}_\Sigma(N))$ contradicting the original assumption. Therefore, it must be that $C \in N \setminus Red_C(N)$ and thus $D \in G_\Sigma(N \setminus Red_C(N))$. □

I now prove that the redundancy criterion meets requirements (R1) – (R4) of Waldmann et al.'s framework.

Lemma 58. *For a set of clauses N, if $N \models_{\mathsf{SKI}} \bot$, then $N \setminus Red_C(N) \models_{\mathsf{SKI}} \bot$.*

Proof. It suffices to prove that for all sets of clauses N, $N \setminus Red_C(N) \models_{\mathsf{SKI}} N$. Let $\mathcal{I}$ be a combinatory model of $N \setminus Red_C(N)$. By Lemma 55, $\mathcal{I}$ is a model of $\mathcal{G}_\Sigma(N \setminus Red_C(N))$. By Lemma 57, $\mathcal{I}$ is a model of $\mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N))$. By Lemma 56, there exists a monomorphic model $\mathcal{I}'$ such that $\mathcal{I} \models \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \cup \lfloor \mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N)) \rfloor$ and thus $\mathcal{I} \models \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \cup \lfloor \mathcal{G}_\Sigma(N) \rfloor \setminus Red_C(\lfloor \mathcal{G}_\Sigma(N) \rfloor)$. By Lemma 29, $\mathcal{I}'$ is a model of $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor \cup \lfloor \mathcal{G}_\Sigma(N) \rfloor$. By Lemma 56, $\mathcal{I}$ is thus a model of $\mathcal{G}_\Sigma(N)$ and by Lemma 52, a model of N. □

Lemma 59. *For a sets of clauses N and N' such that $N \subseteq N'$, $Red_C(N) \subseteq Red_C(N')$ and $Red_I(N) \subseteq Red_I(N')$*

Proof. For a clause $C \in Red_C(N)$, I need to show $C \in Red_C(N')$. By definition, $C \in Red_C(N)$ implies that for every $D \in \mathcal{G}_\Sigma(C)$, $D \in Red_C(\mathcal{G}_\Sigma(N))$. This in turn implies that $\lfloor D \rfloor \in Red_C(\lfloor \mathcal{G}_\Sigma(N) \rfloor)$. Using Lemma 29, we have $\lfloor D \rfloor \in Red_C(\lfloor \mathcal{G}_\Sigma(N') \rfloor)$ and thus $D \in Red_C(\mathcal{G}_\Sigma(N'))$. Since this is the case for every $D \in \mathcal{G}_\Sigma(C)$, it follows that $C \in Red_C(N')$.

For an inference $inf \in Red_I(N)$, I need to show $inf \in Red_I(N')$. If inf is an ArgCong inference, then $conc(inf) \in N \cup Red_C(N)$. If $conc(inf) \in N$, then $conc(inf) \in N'$ and thus $inf \in Red_I(N')$. Otherwise if $conc(inf) \in Red_C(N)$, then by the first part of this lemma, $conc(inf) \in Red_C(N')$ and thus $inf \in Red_I(N')$.

If inf is not an ARGCONG inference, let G_{inf} be the set of ground instances of inf. By definition, for every $inf' \in G_{inf}$, $inf' \in Red_I(\mathcal{G}_\Sigma(N))$. Let C be the maximal premise of inf' and E its conclusion. By definition, $\lfloor \mathcal{G}_\Sigma(ECA) \rfloor \cup \{\lfloor D \rfloor \in \lfloor N \rfloor \mid \lfloor D \rfloor \prec \lfloor C \rfloor\} \models \lfloor E \rfloor$. Since $\lfloor D \rfloor \in \lfloor N \rfloor$ implies $\lfloor D \rfloor \in \lfloor N' \rfloor$, it follows that $inf' \in Red_I(\mathcal{G}_\Sigma(N'))$. Since this is the case for every $inf' \in G_{inf}$, it follows that $inf \in Red_I(N')$. □

Lemma 60. *For a set of clauses N let $N' \subseteq Red_C(N)$. Then, $Red_C(N) \subseteq Red_C(N \setminus N')$ and $Red_I(N) \subseteq Red_I(N \setminus N')$*

Proof. For a clause $C \in Red_C(N)$, I need to show that $C \in Red_C(N \setminus N')$. By definition, we have that for every clause $D \in G_\Sigma(C)$, $D \in Red_C(\mathcal{G}_\Sigma(N))$. By the definition of redundancy for ground clauses, this entails that $\lfloor D \rfloor \in Red_C(\lfloor \mathcal{G}_\Sigma(N) \rfloor)$. From this, by Lemma 29, we have that $\lfloor D \rfloor \in Red_C(\lfloor \mathcal{G}_\Sigma(N) \rfloor \setminus Red_C(\lfloor \mathcal{G}_\Sigma(N) \rfloor))$. From this, we have that $\lfloor D \rfloor \in Red_C(\lfloor \mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N)) \rfloor)$ and thus $D \in Red_C(\mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N)))$. By Lemma 57, this implies $D \in Red_C(G_\Sigma(N \setminus Red_C(N)))$. Since $N' \subseteq Red_C(N)$, from this follows $D \in Red_C(G_\Sigma(N \setminus N'))$. Since for every $D \in G_\Sigma(C)$, $D \in Red_C(G_\Sigma(N \setminus N'))$, it follows that $C \in Red_C(N \setminus N')$

For an inference $inf \in Red_I(N)$, I need to show $inf \in Red_I(N \setminus N')$. If inf is an ARGCONG inference this is can be shown easily using the previous part of the lemma.

If inf is not an ARGCONG inference, let G_{inf} be the set of ground instances of inf. By definition, for every $inf' \in G_{inf}$, $inf' \in Red_I(\mathcal{G}_\Sigma(N))$. Let C be the maximal premise of inf' and E its conclusion. By Lemma 29, and the definition of inference redundancy, there exists clauses $D_1, \ldots, D_n \in \lfloor \mathcal{G}_\Sigma(N) \rfloor \setminus Red_C(\lfloor \mathcal{G}_\Sigma(N) \rfloor)$ such that each $D_i \prec \lfloor C \rfloor$ and $\{D_1, \ldots, D_n\} \cup \lfloor \mathcal{G}_\Sigma(ECA) \rfloor \models \lfloor E \rfloor$. Since the floor function is a bijection, we have that $D_1, \ldots D_n \in \lfloor \mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N)) \rfloor$. This implies that $inf' \in Red_I(\mathcal{G}_\Sigma(N) \setminus Red_C(\mathcal{G}_\Sigma(N)))$ and thus that $inf \in Red_I(N \setminus Red_C(N))$. Since $N' \subseteq N$, this implies that $inf \in Red_I(N \setminus N')$ □

Lemma 61. *For an inference inf, if $conc(inf) \in N$, then $inf \in Red_I(N)$.*

Proof. If inf is an ARGCONG inference, the lemma holds be definition. Otherwise, by Lemma 28, the set of all ground instances of inf, G_{inf} is a subset of $Red_I(G_\Sigma(conc(inf)))$. Since $conc(inf) \in N$, $Red_I(G_\Sigma(conc(inf))) \subseteq Red_I(G_\Sigma(N))$ by Lemma 59. Thus, all ground instances of inf are redundant with respect to $\mathcal{G}_\Sigma(N)$ and therefore inf is redundant with respect to N. □

With the above four lemmas in place, Lemma 6 of Waldmann et al's framework [124] ensures that dynamic completeness follows from static completeness.

Theorem 13. *For every fair clausal combinatory superposition (select or no-select) derivation $N_1, N_2, \ldots$ such that N_1 is not satisfiable, there exists some i such that $\bot \in N_i$.*

4.5 | Removing Combinator Axioms

Next, I show that it is possible to replace the combinator axioms with a dedicated inference rule. I name the inference NARROW. Unlike the other inference rules, it works at prefix positions. I define *nearly first-order* positions inductively. For any term t, either $t = \zeta\,\overline{s}_n$ where ζ is not a fully applied combinator or $t = \mathcal{C}_{\mathsf{any}}\,\overline{s}_n$. In the first case, the nearly first-order subterms of t are $\zeta\,\overline{s}_i$ for $0 \leq i \leq n$ and all the nearly first-order subterms of the s_i. In the second case, the nearly first-order subterms are $\mathcal{C}_{\mathsf{any}}\,\overline{s}_i$ for $0 \leq i \leq n$. The notation $s\langle\!\langle u \rangle\!\rangle$ is to be read as u is a nearly first-order subterm of s. The NARROW inference:

$$\frac{C' \vee [\neg] s\langle\!\langle u \rangle\!\rangle \approx s'}{(C' \vee [\neg] s\langle\!\langle r \rangle\!\rangle \approx s')\sigma}\ \textsc{Narrow}$$

with the following side conditions:

1. $u \notin \mathcal{V}$
2. Let $l \approx r$ be a combinator axiom. $\sigma = mgu(l, u)$;
3. $s\langle\!\langle u \rangle\!\rangle\sigma \not\preceq s'\sigma$;
4. $[\neg]\, s\langle\!\langle u \rangle\!\rangle \approx s'$ is σ-eligible in C, and strictly σ-eligible if it is positive.

I show that any inference that can be carried out using an extended combinator axiom can be simulated with NARROW proving completeness. It is obvious that an EQRES or EQFACT inference cannot have an extended combinator axiom as its premise. By the SUBVARSUP side conditions, an extended combinator axiom cannot be either of its premises. Thus we only need to show that SUP inferences with extended combinator axioms can be simulated. Note that an extended axiom can only be the left premise of a SUP inference. Consider the following inference:

$$\frac{l \approx r \qquad C' \vee [\neg] s\langle u \rangle|_p \approx s'}{(C' \vee [\neg] s\langle r \rangle \approx s')\sigma}\ \textsc{Sup}$$

Let $l = \mathsf{S}\langle\overline{\alpha}\rangle\,\overline{x}_{n>3}$. By variable condition 1, we have that $u = \zeta\,\overline{s}_m$ where $n \geq m \geq n-2$. If $u = y\,\overline{s}_{n-2}$, then $\sigma = \{y \to \mathsf{S}\langle\overline{\alpha}\rangle\,x_1\,x_2, x_3 \to s_1, \ldots, x_n \to s_{n-2}\}$. In this case $r\sigma = (x_1\,x_3\,(x_2\,x_3)\,x_4 \ldots x_n)\sigma = x_1\,s_1\,(x_2\,s_1)\,s_2 \ldots s_{n-2}$ and the conclusion of the inference is $(C' \vee [\neg] s\langle x_1\,s_1\,(x_2\,s_1)\,s_2 \ldots s_{n-2}\rangle \approx s')\{y \to \mathsf{S}\langle\overline{\alpha}\rangle\,x_1\,x_2\}$. Now consider the following NARROW inference from C at the nearly first-order subterm $y\,t_1$:

$$\frac{C' \vee [\neg] s\langle\langle\!\langle y\,s_1 \rangle\!\rangle\,s_2 \ldots s_n\rangle|_p \approx s'}{(C' \vee [\neg] s\langle x_1\,s_1\,(x_2\,s_1)\,s_2 \ldots s_{n-2}\rangle \approx s')\{y \to \mathsf{S}\langle\overline{\alpha}\rangle\,x_1\,x_2\}}\ \textsc{Narrow}$$

As can be seen, the conclusion of the SUP inference is equivalent to that of the NARROW inference up to variable naming. The same can be shown to be the case where $u = y\,\overline{s}_{n-1}$ or $u = y\,\overline{s}_n$ or $u = \mathsf{S}\langle\overline{\alpha}\rangle\,\overline{s}_n$. Likewise, the same can be shown to hold when the $l \approx r$ is an extended $\mathsf{B}, \mathsf{C}, \mathsf{K}$ or I axiom.

The calculus can be further improved by limiting the NARROW rule to the case where the narrowed subterm u, has a variable head. In the case where u has a combinator head, no variables in u are bound by σ, only variables in the combinator axiom are. Therefore, the SUP inference can be replaced by a DEMOD inference. I provide a special name for a demodulation inference involving combinator axioms, COMBDEMOD. The rule is as follows:

$$\frac{[\![\, C' \vee [\neg] s[\mathcal{C}_{\mathsf{any}} \langle \overline{\tau} \rangle \, \overline{s}] \approx s' \,]\!]}{C' \vee [\neg] s[(\mathcal{C}_{\mathsf{any}} \langle \overline{\tau} \rangle \, \overline{s}) \downarrow^{w}] \approx s'} \text{ COMBDEMOD}$$

The name NARROW is a generic name that does not specify the axiom involved in the inference. In practice it can be useful to differentiate between different cases of the NARROW inference. In particular, the case where the axiom being used is an **S**-, **C**- or **B**-axiom and the term being narrowed is of the form $x\,t$ (a variable with a single argument), behaves rather differently to other cases. Such an inference can introduce polymorphism into a monomorphic problem as the following example demonstrates.

Example 19. Consider the negated conjecture $x\,\mathsf{a} \not\approx \mathsf{a}$, where x is of type $i \to i$. Consider narrowing the term $x\,\mathsf{a}$ with the axiom $\mathbf{S}\langle \overline{\alpha}_3 \rangle \, x_1 \, x_2 \, x_3 \approx x_1 \, x_3 \, (x_2 \, x_3)$. Unifying x_3 with a forces α_1 to be i. Likewise, α_3 is forced to be i. However, there is nothing to constrain the type of α_2. Therefore, the variable x_2 in the conclusion has type $i \to \alpha_2$. The variable x_1 has type $i \to \alpha_2 \to i$.

Due to the above, I assign different names to NARROW inferences based on the axiom involved and the number of arguments the variable head of the narrowed term has. If the axiom involved is the **S**-axiom and the narrowed term is of the form $x\,t$, the inference is called a **S**XX-NARROW. If the narrowed term is of the form $x\,t_1\,t_2$, the inference is called a **S**X-NARROW. If the narrowed term has three arguments, the inference is called **S**-NARROW. The definition of **C**XX-NARROW, **B**XX-NARROW, **B**X-NARROW etc. are analogous.

Chapter 5

Extensions to the Calculi

> Logic is justly considered the basis of all other sciences, even if only for the reason that ... every correct inference proceeds in accordance with its laws.
>
> *Alfred Tarski*

In this chapter, I present various extensions to the clausal combinatory superposition calculi. The chapter does not represent a research contribution of the thesis, since the extensions are in the main part minor variants of existing ideas. The chapter is included as part of the thesis for the purpose of completeness and narrative.

5.1 | Dealing with Booleans

In Chapter 2 the syntax and semantics of *clausal* higher-order and applicative first-order logic were introduced. Of course, the clausal logic is highly restricted by its lack of support for Booleans. The most common semantics for higher-order logic stipulates Booleans and choice [24]. This is also the semantics assumed by the TPTP problem library [112]. In this section I describe how the calculi presented in the previous chapter can be extended to deal with reasoning about Booleans and choice. I first present the syntax and semantics of non-clausal applicative first-order logic.

5.1.1 | The Syntax of Applicative First-Order Logic (Non-clausal)

Type signatures and types are defined as per the clausal first-order case. The definition of a type declaration is as before. It is assumed that any term signature Σ contains the logical connectives $\bot, \top, \wedge, \sqcup, \neg, \Rightarrow$ with their usual types. It is also assumed that Σ contains the quantifiers $\exists, \forall : \Pi\sigma.\,(\sigma \to o) \to o$ and equality $\simeq$: $\Pi\sigma.\,\sigma \to \sigma \to o$. Finally, the presence of a Hilbert choice operator $\varepsilon : \Pi\sigma.\,(\sigma \to o) \to \sigma$ is assumed. Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be

an arbitrary signature and $\mathcal{V}$ a countably infinite set of typed variables. The set of terms $\mathcal{T}_\Sigma(\mathcal{V})$ over the signature $(\Sigma_{\mathsf{ty}}, \Sigma)$ is defined as:

- If $x : \tau \in \mathcal{V}$, then x is a term of type τ.
- Let $\mathsf{f} : \Pi\overline{\alpha}_n.\tau$ be a member of Σ. Let $\overline{\tau}_n$ be a tuple of types. Let σ be the substitution $\{\alpha_1 \to \tau_1, \ldots, \alpha_n \to \tau_n\}$. Then, $\mathsf{f}\langle\overline{\tau}_n\rangle$ is a term of type $\tau\sigma$.
- If t_1 is a term of type $\tau_1 \to \tau_2$ and t_2 is a term of type τ_1 then $t_1\, t_2$ is a term of type τ_2.

Terms of type o are called *formulas*. Formulas are denoted using symbols $f, g, h, \ldots$. Following Vukmirović and Nummelin [120], formulas are categorised as either being *nested* or *top level*. A formula f that occurs in a term t at position p is a top level Boolean if, for every prefix p' of p, $head(t|_{p'})$ is a logical connective. Otherwise f is a nested Boolean.

A literal is an equality between terms written $s \approx t$. A negative literal is denoted $s \not\approx t$. A clause is a finite multiset of literals denoted $l_1 \vee \ldots \vee l_n$ where each l_i is a literal. The syntax contains two symbols for equality with $\approx$ being used for equality at the literal level, and $\simeq$ for equality at the term level. Likewise, disjunction has two symbols, $\vee$ to represent the disjunction between literals of a clause and $\sqcup$ to represent disjunction at the term level.

5.1.2 | The Semantics of Applicative First-Order Logic (Non-clausal)

The definition of a type valuation is as per the clausal case with three additional conditions. Firstly, it is required that $\mathcal{U}$ contains a set $\mathcal{U}_o = \{0, 1\}$. Secondly, it is required that $\mathcal{J}_{\mathsf{ty}}(o) = \mathcal{U}_o$. Lastly, it is required that for every pair of universes $\mathcal{U}_1, \mathcal{U}_2$, $\mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{U}_1, \mathcal{U}_2)$ must be a subset of $\mathcal{U}_1^{\mathcal{U}_2}$.

The definition of an interpretation is as per the clausal applicative first-order case, with some extra conditions required to ensure that the logical connectives are interpreted appropriately. These conditions are as follows:

- $\mathcal{J}(\bot) = 0$;
- $\mathcal{J}(\top) = 1$;
- $\mathcal{J}(\neg)(\mathsf{a}) = 1 - \mathsf{a}$ for all $\mathsf{a} \in \mathcal{U}_o$;
- $\mathcal{J}(\sqcup)(\mathsf{a}, \mathsf{b}) = min(\{\mathsf{a}, \mathsf{b}\})$ for all $\mathsf{a}, \mathsf{b} \in \mathcal{U}_o$;
- $\mathcal{J}(\wedge)(\mathsf{a}, \mathsf{b}) = max(\{\mathsf{a}, \mathsf{b}\})$ for all $\mathsf{a}, \mathsf{b} \in \mathcal{U}_o$;
- $\mathcal{J}(\Rightarrow)(\mathsf{a}, \mathsf{b}) = max(\{1 - \mathsf{a}, \mathsf{b}\})$ for all $\mathsf{a}, \mathsf{b} \in \mathcal{U}_o$;
- $\mathcal{J}(\forall, \mathcal{U}_1)(\mathsf{f}) = min(\{\mathsf{f}(\mathsf{a}) \mid \mathsf{a} \in \mathcal{U}_1\})$ for all $\mathcal{U}_1 \in \mathcal{U}$ and $\mathsf{f} \in \mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{U}_1, \mathcal{U}_o)$;
- $\mathcal{J}(\exists, \mathcal{U}_1)(\mathsf{f}) = max(\{\mathsf{f}(\mathsf{a}) \mid \mathsf{a} \in \mathcal{U}_1\})$ for all $\mathcal{U}_1 \in \mathcal{U}$ and $\mathsf{f} \in \mathcal{J}_{\mathsf{ty}}(\to)(\mathcal{U}_1, \mathcal{U}_o)$;
- $\mathcal{J}(\simeq, \mathcal{U}_1)(\mathsf{a}, \mathsf{b}) = (\mathsf{a} = \mathsf{b})$ for all $\mathsf{a}, \mathsf{b} \in \mathcal{U}_1$;

- $\mathsf{f}(\mathcal{J}(\varepsilon, \mathcal{U}_1)(\mathsf{f})) = max(\{\mathsf{f}(\mathsf{a}) \mid \mathsf{a} \in \mathcal{U}_1\})$ for all $\mathsf{f} \in \mathcal{J}_{\mathsf{ty}}(\rightarrow)(\mathcal{U}_1, \mathcal{U}_o)$;

For an interpretation $\mathcal{I} = (\mathcal{I}_{\mathsf{ty}}, \mathcal{J})$ and a valuation ξ, the denotation of a term is as follows:

- $[\![x]\!]^{\xi}_{\mathcal{I}} = \xi(x)$;
- $[\![\mathsf{f}\langle\overline{\tau}_n\rangle]\!]^{\xi}_{\mathcal{I}} = \mathcal{J}(\mathsf{f}, [\![\tau_1]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}}, \ldots, [\![\tau_n]\!]^{\xi}_{\mathcal{I}_{\mathsf{ty}}})$;
- $[\![s\,t]\!]^{\xi}_{\mathcal{I}} = [\![s]\!]^{\xi}_{\mathcal{I}}([\![t]\!]^{\xi}_{\mathcal{I}})$.

An equation $s \approx t$ is true in an interpretation $\mathcal{I}$ with valuation function ξ if $[\![s]\!]^{\xi}_{\mathcal{I}}$ and $[\![t]\!]^{\xi}_{\mathcal{I}}$ are the same object, and is false otherwise. A disequation $s \not\approx t$ is true if $s \approx t$ is false. A clause is true if one of its literals is true, and a clause set is true if every clause in the set is true. An interpretation $\mathcal{I}$ is a model of a clause C, written $\mathcal{I} \models C$, if C is true in $\mathcal{I}$ for all valuation functions. An interpretation is a model of a clause set if it is a model of each of its clauses.

In the presentation given above, all literals are assumed to be equations. This is no restriction since terms of type o can be converted into literals by setting them equal to $\top$. Some calculi for non-clausal higher-order logic assume a standard representation for literals involving $\top$ and $\bot$. For example, Vukmirović and Nummelin [120] assume that literals of the form $f \approx \bot$ are rewritten to $f \not\approx \top$ and literals of the form $f \not\approx \bot$ are rewritten to $f \approx \top$. I make no such assumptions in the presentation below. However, to avoid having to duplicate inference rules, the following convention is used. If an inference rule involves a literal of the form $f \approx \top$ it should be assumed that it equally applies to a literal of the form $f \not\approx \bot$. Likewise, if an inference rule involves a literal of the form $f \approx \bot$ it should be assumed that the it equally applies to a literal of the form $f \not\approx \top$.

5.1.3 | Axiomatisation

One way of reasoning about the logical connectives is to axiomatise them. I present an axiomatisation of the logical connectives below. Note that it is one possible axiomatisation. Others are possible as well and may well be more efficient in practice. The axiomatisation method considers logical connectives to be uninterpreted symbols and adds axioms to ensure the desired semantics.

$$\bot \not\approx \top \qquad x \approx \bot \vee x \approx \top$$

$$x \sqcup y \approx \bot \vee x \approx \top \vee y \approx \top$$
$$x \sqcup y \approx \top \vee x \approx \bot$$
$$x \sqcup y \approx \top \vee y \approx \bot$$

$$x \wedge y \approx \top \vee x \approx \bot \vee y \approx \bot$$
$$x \wedge y \approx \bot \vee x \approx \top$$
$$x \wedge y \approx \bot \vee y \approx \top$$

$$x \Rightarrow y \approx \bot \vee x \approx \bot \vee y \approx \top$$
$$x \Rightarrow y \approx \top \vee x \approx \top$$
$$x \Rightarrow y \approx \top \vee x \approx \bot$$

$$\neg x \approx \bot \vee x \approx \bot \qquad \neg x \approx \top \vee x \approx \top$$

$$\simeq \langle \alpha \rangle\, x\, y \approx \bot \vee x \approx y \qquad \simeq \langle \alpha \rangle\, x\, y \approx \top \vee x \not\approx y$$

$$\forall \langle \alpha \rangle\, x \approx \top \vee x\, (\mathsf{sk}\, x) \approx \bot \qquad \forall \langle \alpha \rangle\, x \approx \bot \vee x\, y \approx \top$$

$$\exists \langle \alpha \rangle\, x \approx \top \vee x\, y \approx \bot \qquad \exists \langle \alpha \rangle\, x \approx \bot \vee x\, (\mathsf{sk}\, x) \approx \top$$

$$y\, x \approx \bot \vee y\, (\varepsilon \langle \alpha \rangle\, y) \approx \top$$

It is relatively easy to demonstrate that by assuming the above axioms in addition to the inference rules of the clausal combinatory superposition calculi, complete non-clausal calculi are derived. Unfortunately, research has shown that axiomatising logical connectives tends to lead to extremely poor prover performance. Therefore, the approach of treating logical constants as logical symbols and adding specific inferences to reason about them is preferable.

5.1.4 | Inference Rules

An effective alternative to axiomatising Booleans is to add inference rules to reason about them. This is very similar to how equality itself is treated by superposition or how superposition can be extended to reason about arbitrary bounded domains [68]. Note however, that whilst the axiomatisation provided above is complete, the inference rules that are to be presented are pragmatic. I provide them without any completeness guarantees. That is not to say that the method of using inferences to deal with Boolean reasoning cannot be made complete. Steen presents a complete calculus for full higher-order logic in his thesis [107]. However, the calculus is an unordered paramodulation calculus without the concept of redundancy. Reasoning about Booleans in the presence of orderings and redundancy criteria is an open area of research.

Kotelnikov et al. introduced the FOOL logic which is essentially first-order logic with support for Booleans, if-then-else constructs and let-in constructs [81]. They provided two approaches to reasoning about Booleans, both complete with respect to their semantics.

The first is motivated by the desire to avoid adding the axiom $x \approx \top \vee x \approx \bot$ to the search space. The authors note that for a term order $\succ$, such that $\top \succ \bot$ and $\top$ and $\bot$ are the smallest members of the signature with respect to $\succ$, the only possible inference involving this axiom is a superposition inference with the axiom as the left premise. Moreover, the axiom cannot superpose with itself. Therefore, it can be replaced by a dedicated inference rule which they call FOOLPARAMODULATION.

$$\frac{C[s]}{C[\top] \vee s \approx \bot} \text{ FOOLPARAMODULATION}$$

Where s is a nested Boolean that is not a variable and is not $\top$ or $\bot$. The second approach of Kotelnikov et al. was to extend the VCNF clausification process [93] to deal with Booleans (and the other constructs supported by FOOL)[82]. The extension to VCNF is called $\text{VCNF}_{\text{FOOL}}$. Due to the clausification techniques employed by $\text{VCNF}_{\text{FOOL}}$, after the clausification process the only Boolean subterms remaining in a clause are variables, and the constants $\top$ and $\bot$. As FOOLPARAMODULATION is not relevant on these, when $\text{VCNF}_{\text{FOOL}}$ is used, no special Boolean reasoning is required at all.

When moving to extensional higher-order logic, preprocessing techniques such as $\text{VCNF}_{\text{FOOL}}$ are very incomplete when used on their own. Extensionality can lead to Boolean terms being created during proof search as the following example demonstrates.

Example 20. Consider the unsatisfiable clause set $\{\mathbf{B}\,\mathbf{I}\,(\wedge\,\bot) \not\approx \mathbf{K}\,\bot\}$. The single literal in the sole member of the set is of type $o \to o$ and there are no Boolean terms other than $\bot$. However, using extensionality, the clause $\mathbf{B}\,\mathbf{I}\,(\wedge\,\bot)\,\mathsf{sk} \not\approx \mathbf{K}\,\bot\,\mathsf{sk}$ can be derived which contains two Boolean terms. Multiple narrow steps result in the clause $\bot \wedge \mathsf{sk} \not\approx \bot$ which is easily refutable.

The example shows that preprocessing of Booleans is not sufficient for higher-order logic and that the interplay between functional extensionality and Boolean reasoning can be quite subtle. As FOOLPARAMODULATION is not a preprocessing technique, but a calculus rule, it is more complete than $\text{VCNF}_{\text{FOOL}}$ in the higher-order setting.

However, due to the more complex nature of higher-order terms, FOOLPARAMODULATION on its own is not sufficient for solving many problems. In the remainder of this section, I discuss various method of extending clausal combinatory superposition to reason about Booleans.

Combinatory Superposition-a

The combinatory superposition-a calculus is an extension of the clausal combinatory superposition calculi. Combinatory superposition-a assumes extensionality, either via the axiom or through the -WA version of the core inference rules. Furthermore, the following inference rules are added.

$$\frac{C\langle s\rangle}{C\langle\top\rangle \vee s \approx \bot}\ \textsc{FoolParamodulation}$$

where s is a nested Boolean subterm that is not a variable and is not $\top$ or $\bot$. The above inference can also be used as a simplification rule, resulting in the CASESSIMP rule introduced by Vukmirović and Nummelin [120].

$$\frac{[\![C\langle s\rangle]\!]}{C\langle\top\rangle \vee s \approx \bot \quad C\langle\bot\rangle \vee s \approx \top}\ \textsc{CasesSimp}$$

A peculiarity of non-clausal higher-order logic is that unification is not sufficient for instantiating higher-order variables. Because variables can represent arbitrary formulas and the semantics assumes the logical constants, it may be necessary to "guess" the logical formula a variable represents. The following example makes this clear.

Example 21. The statement $x\,\mathsf{a}\,\mathsf{a} \approx \top$ is a theorem of higher-order logic since x can be instantiated with $\simeq$ to produce $\mathsf{a} \simeq \mathsf{a} \approx \top$.

To deal with such reasoning most higher-order calculi utilise a rule known variously as primitive instantiation or splitting [23]. A version of the rule suitable for the combinatory setting is provided below. Let R be a set of Boolean terms.

$$\frac{C \vee [\neg]x\,\overline{s} \approx t}{(C \vee [\neg]x\,\overline{s} \approx t)\sigma}\ \textsc{ExtendedNarrow}$$

where σ is the most general unifier of some term $u \in R$ and $x\,\overline{s}$. Four versions of the rule exist. In each version, the set R contains different terms. In version 1, the full version, R contains the terms:

$$\begin{array}{lllll} x \wedge y & x \sqcup y & x \Rightarrow y & \neg x & x \simeq \langle\alpha\rangle\, y \\ \mathbf{K}\,\bot\,x & \mathbf{K}\,\top\,x & \mathbf{B}\,(\mathbf{B}\,\neg) \simeq \langle\alpha\rangle\, x\, y & \forall\langle\alpha\rangle\, x & \exists\langle\alpha\rangle\, x \end{array}$$

In version 2, R contains a reduced set of terms:

$$\begin{array}{lllll} x \wedge y & x \sqcup y & x \Rightarrow y & \neg x & x \simeq \langle\alpha\rangle\, y \\ \mathbf{K}\,\bot\,x & \mathbf{K}\,\top\,x & & & \end{array}$$

In version 3, R contains the terms:

$$\begin{array}{lllll} \neg x & x \simeq \langle\alpha\rangle\, y & \mathbf{K}\,\bot\,x & \mathbf{K}\,\top\,x & \mathbf{B}\,(\mathbf{B}\,\neg) \simeq \langle\alpha\rangle\, x\, y \\ \forall\langle\alpha\rangle\, x & \exists\langle\alpha\rangle\, x & & & \end{array}$$

In version 4, the minimal version, R only contains the terms $\mathbf{K}\,\bot\,x, \mathbf{K}\,\top\,x$ and $\neg x$. In theory, any number of versions of the rule could be developed by varying R. By including more terms, greater completeness can be achieved at the cost of more explosive calculi. Some of the terms are merely included because they can lead to shorter proofs, as demonstrated in the following example.

Example 22. Consider the negated conjecture $x\,\mathsf{a} \not\approx \top$. One way to achieve a contradiction from this clause is to apply an EXTENDEDNARROW inference by unifying $x\,\mathsf{a}$ with the term $\mathbf{K}\top x \in R$.

$$\frac{x\,\mathsf{a} \not\approx \top}{\mathbf{K}\top\,\mathsf{a} \not\approx \top}\ \textsc{ExtendedNarrow}$$

A simple rewrite with the $\mathbf{K}$-axiom on the conclusion results in the trivially refutable $\top \not\approx \top$. However, another way to achieve the contradiction is to *first* carry out a NARROW step.

$$\frac{x\,\mathsf{a} \not\approx \top}{x \not\approx \top}\ \textsc{Narrow}$$

An EXTENDEDNARROW inference can be carried out on the conclusion by unifying x with $\neg y \in R$. The result is the clause $\neg x \not\approx \top$. Clausification on the fly (explained below) results in $x \approx \top$. This clause can superpose with the original clause to again achieve $\top \not\approx \top$.

The name of the EXTENDEDNARROW inference comes from the field of theorem proving modulo. It should be noted that the rule presented above is not quite an extended narrowing rule in the sense understood by Dowek et al. [55]. In the original presentation of extended narrowing, the rule involves unifying with the left hand side of a rewrite rule, rewriting and then clausifying the result if necessary. In my presentation, I separate out the rewriting and clausifying steps in order to provide greater flexibility. Note that in the presentation of the syntax of non-clausal higher-order logic, there is no stipulation that the left or right hand sides of a literal not have logical constants as heads. Indeed, $t_1 \wedge t_2 \approx t_3$ is a perfectly valid literal. Even if no clauses in a problem contain literals of this sort, the EXTENDEDNARROW and FOOLPARAMODULATION rules can result in literals of this form. Therefore, a set of clausification inference rules are required.

$$\frac{[\![f \wedge g \approx \top \vee C]\!]}{f \approx \top \vee C \quad g \approx \top \vee C}\ \mathrm{CNF}_{\wedge} \qquad \frac{[\![\forall\langle\alpha\rangle\, t \approx \top \vee C]\!]}{t\,x \approx \top \vee C}\ \mathrm{CNF}_{\forall}$$

$$\frac{[\![f \sqcup g \approx \top \vee C]\!]}{f \approx \top \vee g \approx \top \vee C}\ \mathrm{CNF}_{\vee} \qquad \frac{[\![\exists\langle\alpha\rangle\, t \approx \top \vee C]\!]}{t\,(\mathsf{sk}\langle\overline{\alpha}\rangle\,\overline{x}) \vee C}\ \mathrm{CNF}_{\exists}$$

$$\frac{[\![f \Rightarrow g \approx \top \vee C]\!]}{f \not\approx \top \vee g \approx \top \vee C}\ \mathrm{CNF}_{\Rightarrow} \qquad \frac{[\![f \simeq g \approx \top \vee C]\!]}{f \approx g \vee C}\ \mathrm{CNF}_{\simeq +}$$

$$\frac{[\![\neg f \approx \top \vee C]\!]}{f \not\approx \top \vee C}\ \mathrm{CNF}_{\neg} \qquad \frac{[\![f \simeq g \approx \bot \vee C]\!]}{f \not\approx g \vee C}\ \mathrm{CNF}_{\simeq -}$$

$$\frac{[\![f \approx g \vee C]\!]}{f \not\approx \top \vee g \approx \top \vee C \quad g \not\approx \top \vee f \approx \top \vee C}\ \mathrm{CNF}_{\approx}$$

I do not present the CNF rules with opposite polarity, but these should be easy enough to visualise. In the $\text{CNF}_\forall$ rule, x is a fresh variable. In the $\text{CNF}_\exists$ rule, sk is a Skolem function and, $\overline{\alpha}$ are the type variables of t and $\overline{x}$ are t's term variables. In an implementation of the calculi, the ordering of these rules with respect to the other simplification rules is a rich area for exploration. If the CNF rules are always carried out before other simplification rules, the EC (eager clausification) rule of Vukmirović and Nummelin is attained. At times it is also helpful to use generating versions of the rules that do not delete the premises.

In order to reason about choice, I use the inference mechanisms introduced by Steen in the Leo-III prover [107]. An inference that removes clauses that define choice operators is added to the calculi.

$$\frac{[\![\, x \approx \bot \vee y\,(\mathsf{f}\,y) \approx \top \,]\!]}{} \text{ ACD}$$

As a side effect of the rule, the constant f is added to a set CO of choice operators. The choice operator ε is always a member of CO. In addition to this rule, a second rule that creates instances of the axiom of choice is utilised.

$$\frac{C\langle x\,t\rangle}{\begin{array}{c}(t\,x \approx \bot \vee t\,(\xi_1\,t) \approx \top)\sigma \\ (t\,x \approx \bot \vee t\,(\xi_2\,t) \approx \top)\sigma \\ (t\,x \approx \bot \vee t\,(\xi_3\,t) \approx \top)\sigma \\ \vdots\end{array}} \text{ CHOICE}$$

The inference has one conclusion for each $\xi_i \in CO$. In each conclusion, σ is the most general unifier of the types of ξ_i and x. There is also a second version of the inference.

$$\frac{C\langle \xi\,t\rangle}{t\,x \approx \bot \vee t\,(\xi\,t) \approx \top} \text{ CHOICE}$$

This version is used for $\xi \in CO$. In many cases, it is possible to simplify nested booleans without the use of FOOLPARAMODULATION. I add a rule BOOLSIMP to the calculi which is very similar to the identically named rule described by Vukmirović and Nummelin [120] and the SIMP rule described by Steen [107]. Let E be the following set of rewrite rules:

$$\begin{array}{cccc}
\bot \wedge x \to \bot & \top \wedge x \to x & x \wedge x \to x & x \wedge \neg x \to \bot \\
\top \sqcup x \to \top & \bot \sqcup x \to x & x \sqcup x \to x & x \sqcup \neg x \to \top \\
\top \Rightarrow x \to \top & x \Rightarrow \top \to x & \top \Rightarrow x \to \top & x \Rightarrow \neg x \to x \\
x \Rightarrow x \to \top & \bot \Rightarrow x \to \neg x & \neg\bot \to \top & \neg\top \to \mathit{bot} \\
\neg\neg x \to x & x \simeq x \to \top & &
\end{array}$$

For the sake of simplicity, I have left out some obvious rules such as $x \wedge \bot \to \bot$. Let $l \to r$ be a member of E. Then, the BOOLSIMP inference is as given below.

$$\frac{[\![C\langle l\sigma\rangle]\!]}{C\langle r\sigma\rangle}\ \textsc{BoolSimp}$$

Here and elsewhere in the chapter, a simplification rule is shown as working on first-order subterms. This need not be the case, and in practice it may well be useful to carry out simplifications beneath fully applied combinators and even at prefix positions.

In higher-order logic, equality can be defined. For example, the formula $\forall x : \alpha, y : \alpha, p : \alpha \to o.\, (p\, x \Rightarrow p\, y) \Rightarrow x \simeq y$ is known as Leibniz equality. It is useful to replace instances of defined equality with native equality since the underlying calculi are superposition calculi that can efficiently reason about equality. Following Steen [107], the following rule is added to the calculi.

$$\frac{C \vee x\, s \approx \bot \vee x\, t \approx \top}{(C \vee s \approx t)\{x \to \simeq\ s\} \quad (C \vee s \approx t)\{x \to \mathbf{B}\neg(\simeq\ t)\}}\ \textsc{ElimLeibniz}$$

Note that this rule does not add any reasoning power unavailable through ExtendedNarrow. However, in some cases it can shorten proofs.

Optional Inference Rules

Vukmirović and Nummelin introduce an interesting set of rules: BoolER, BoolEF+-, BoolEF-+ and BoolEF--. The BoolER inference is as follows:

$$\frac{s \approx s' \vee C'}{C'\sigma}\ \textsc{BoolER}$$

where σ is the unifier of s and $\neg s'$. The rule is equivalent to first rewriting the premise to $s \not\approx \neg s' \vee C'$, then carrying out a standard EqRes inference. The rule works well in the λ-calculus setting where a complex unifier of s and $\neg s'$ can be found in a single step. In the combinatory setting, it may take a number of Narrow steps to find the unifier. Therefore, instead of BoolER, the following inference can be added to the calculi:

$$\frac{s \approx s' \vee C'}{s \not\approx \neg s' \vee C' \quad \neg s \not\approx s' \vee C'}\ \textsc{RwBoolEq}$$

If RwBoolEq is added to the calculi, rules $\text{CNF}_{\approx}$ and $\text{CNF}_{\neg}$ must be turned into generating rules, since otherwise these would undo the action of RwBoolEq. In an implementation, it is often more efficient not to include both conclusions, but only one, based on some heuristic.

Combinatory Superposition-b

The -b version of the calculi extends the core rules to be 'Boolean aware'. In Algorithm 1, I presented an algorithm for partially unifying terms whilst returning a set of constraint literals for subterms that are not amenable to syntactic unification. The algorithm's original purpose was to return constraint literals consisting of pairs of terms of functional or polymorphic type. The idea being that these terms may not be syntactically unifiable whilst being extensionally equivalent. The use of Algorithm 1 in the core inferences, instead of standard syntactic unification, lead to the SUP-WA, EQRES-WA and EQFACT-WA inference rules. The same idea can be extended to deal with Boolean subterms.

This is achieved by changing line 11 of the algorithm from "**else if** s and t have functional or variable type **then** $\mathcal{D} := \mathcal{D} \cup \{s \not\approx t\}$" to "**else if** s and t have functional or variable or Boolean type **then** $\mathcal{D} := \mathcal{D} \cup \{s \not\approx t\}$".

In the presence of FOOLPARAMODULATION and CASESSIMP, the use of unification with abstraction is not strictly required, but can lead to shorter proofs. Consider the following example.

Example 23. This example is a modification of Example 16 in [23]. Consider the unsatisfiable clause set:

$$C_1 = \mathsf{p}\,(\mathsf{red}\,x \wedge \mathsf{ball}\,x) \approx \top \qquad C_2 = \mathsf{p}\,(\mathsf{ball}\,x \wedge\ \mathsf{red}\,x) \approx \bot$$

A SUP-WA inference between clauses C_1 and C_2 results in the clause $C_3 = \top \approx \bot \vee \mathsf{ball}\,x \wedge \mathsf{red}\,x \not\approx \mathsf{red}\,x \wedge \mathsf{ball}\,x$. Removing the trivial inequality $\bot \approx \top$ results in the clause $C_4 = \mathsf{ball}\,x \wedge\ \mathsf{red}\,x \not\approx \mathsf{red}\,x \wedge \mathsf{ball}\,x$. A $\text{CNF}_{\approx}$ inference on C_4 results in clauses:

$$C_5 = \mathsf{ball}\,x \wedge\ \mathsf{red}\,x \approx \top \vee\ \mathsf{red}\,x \wedge \mathsf{ball}\,x \approx \top$$
$$C_6 = \mathsf{ball}\,x \wedge\ \mathsf{red}\,x \approx \bot \vee\ \mathsf{red}\,x \wedge \mathsf{ball}\,x \approx \bot$$

Clausifying C_5 and C_6 along with a few subsumption inferences leads to the easily refutable clause set:

$$C_7 = \mathsf{ball}\,x \approx \bot \vee\ \mathsf{red}\,x \approx \bot$$
$$C_8 = \mathsf{ball}\,x \approx \top$$
$$C_9 = \mathsf{red}\,x \approx \top$$

A proof can be arrived at without using the abstraction rules, but instead using CASES-

SIMP. A double use of CASESSIMP on clauses C_1 and C_2 results in clauses:

$$C_3 = \mathsf{p}\,(\top) \approx \top \vee \mathsf{red}\,x \wedge \mathsf{ball}\,x \approx \bot$$
$$C_4 = \mathsf{p}\,(\bot) \approx \top \vee \mathsf{red}\,x \wedge \mathsf{ball}\,x \approx \top$$
$$C_5 = \mathsf{p}\,(\top) \approx \bot \vee \mathsf{ball}\,x \wedge \mathsf{red}\,x \approx \bot$$
$$C_6 = \mathsf{p}\,(\bot) \approx \bot \vee \mathsf{ball}\,x \wedge \mathsf{red}\,x \approx \top$$

SUP inferences between clauses C_3 and C_5 and C_4 and C_6 result in clauses:

$$C_7 = \top \approx \bot \vee \mathsf{red}\,x \wedge \mathsf{ball}\,x \approx \bot \vee \mathsf{red}\,x \wedge \mathsf{ball}\,x \approx \bot$$
$$C_8 = \top \approx \bot \vee \mathsf{ball}\,x \wedge \mathsf{red}\,x \approx \top \vee \mathsf{ball}\,x \wedge \mathsf{red}\,x \approx \top$$

Clauses C_7 and C_8 simplify to clauses C_5 and C_6 of the first proof. The proof then proceeds as per the first one.

5.2 | Reasoning with Combinators

Despite the fact that combinatory superposition is an advance over standard superposition when dealing with combinators, it is still explosive. In particular the NARROW inference can quickly result in a large number of clauses. Many of these are extensionally equivalent, but cannot be removed by subsumption which utilises non-extensional first-order matching. Consider the following example.

Example 24. let $C_1 = x\,\mathsf{a}\,\mathsf{b} \approx \mathsf{d} \vee \mathsf{f}\,x \approx \mathsf{a}$. A **C**X-NARROW inference on C_1 results in the clause $C_2 = x\,\mathsf{b}\,\mathsf{a} \approx \mathsf{d} \vee \mathsf{f}\,(\mathbf{C}\,x) \approx \mathsf{a}$. A second **C**X-NARROW inference on C_2 results in the clause $C_3 = x\,\mathsf{a}\,\mathsf{b} \approx \mathsf{d} \vee \mathsf{f}\,(\mathbf{C}\,(\mathbf{C}\,x)) \approx \mathsf{a}$. The clauses C_3 and C_1 are equivalent extensionally since $\mathbf{C}\,(\mathbf{C}\,x)$ and x are extensionally equivalent. However, neither subsumes the other.

In order to help curb the explosion of clauses that NARROW can lead to and to deal with the issue highlighted in the previous example, a simplification rule COMBNORM can be added to the calculi. Let R be the following set of rewrite rules. Each rule in R rewrites a combinator term to an extensionally equivalent term.

$$\begin{array}{ccc}
\mathbf{B}\,\mathbf{I} \to \mathbf{I} & \mathbf{B}\,x\,\mathbf{I} \to x & \mathbf{B}\,x\,(\mathbf{K}\,y) \to \mathbf{K}\,(x\,y) \\
\mathbf{S}\,(\mathbf{C}\,\mathbf{K}) \to \mathbf{I} & \mathbf{S}\,(\mathbf{S}\,\mathbf{K}) \to \mathbf{I} & \mathbf{S}\,(\mathbf{B}\,\mathbf{K}\,x) \to \mathbf{K}\,x \\
\mathbf{S}\,\mathbf{K}\,x \to \mathbf{I} & \mathbf{S}\,(\mathbf{B}\,\mathbf{K}\,x)\,y \to x & \mathbf{S}\,(\mathbf{K}\,x)\,y \to \mathbf{B}\,x\,y \\
\mathbf{S}\,x\,(\mathbf{K}\,y) \to \mathbf{C}\,x\,y & \mathbf{C}\,(\mathbf{C}\,x) \to x & \mathbf{C}\,(\mathbf{K}\,x)\,y \to \mathbf{K}\,(x\,y)
\end{array}$$

Let $l \to r$ be a member of R. The COMBNORM inference can then be given as:

$$\frac{[\![\, C[l\sigma] \,]\!]}{C[r\sigma]}\ \textsc{CombNorm}$$

In the previous example, there is a CombNorm inference that replaces C_3 with the clause $C_4 = x\,\mathsf{a}\,\mathsf{b} \approx \mathsf{d} \vee \mathsf{f}\,x \approx \mathsf{a}$ using the equation $\mathbf{C}\,(\mathbf{C}\,x) \to x$. Clauses C_4 and C_1 are syntactically identical and therefore subsumption would remove one of them. The rules included in R above are pragmatic rather than complete. Furthermore, the approach outlined here relies heavily on subsumption to prune the search space. Subsumption is computationally expensive to carry out, so it would be of interest to develop a version of the combinatory calculi that avoid producing the redundant clauses at all. That is future work. Thanks to Petar Vukmirović for suggesting the approach provided here.

5.3 | Examples

I end this chapter with some examples of how the combinatory superposition-a and -b calculi work. The examples are taken from the TPTP problem library.

5.3.1 | SET557^1.p

```
thf(surjectiveCantorThm,conjecture,(
    ~ ( ? [G: $i > $i > $o] :
        ! [F: $i > $o] :
        ? [X: $i] :
          ( ( G @ X )
          = F ) ) )).
```

Figure 5.1: SET557^1.p

Figure 5.1 displays problem SET557^1.p using TPTP syntax. The same problem displayed using the syntax of this thesis; $\neg(\exists x : i \to i \to o.\, \forall y : i \to o.\, \exists z : i.\, x\,z \approx y)$. Negating and clausifying the conjecture results in the clause $C_1 = \mathsf{sk}_1\,(\mathsf{sk}_2\,y) \approx y$, where sk_1 and sk_2 are Skolem constants. An ArgCong inference on C_1 results in $C_2 = \mathsf{sk}_1\,(\mathsf{sk}_2\,y)\,x \approx y\,x$. A version of the calculi not containing RwBoolEq would simplify C_2 into two clauses using $\text{CNF}_{\approx}$. On the other hand, a version containing RwBoolEq would derive the clause $C_3 = \neg(\mathsf{sk}_1\,(\mathsf{sk}_2\,y)\,x) \not\approx y\,x$. From there the proof proceeds as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{C_3 = \neg(\mathsf{sk}_1\,(\mathsf{sk}_2\,y)\,x) \not\approx y\,x}{C_4 = \neg(\mathsf{sk}_1\,(\mathsf{sk}_2\,(\mathbf{B}\,x_1\,x_2))\,x_3) \not\approx x_1\,(x_2\,x_3)}\ \mathbf{B}\text{XX-NARROW}}{C_5 = \neg(\mathsf{sk}_1\,(\mathsf{sk}_2\,(\mathbf{B}\,x_1\,(\mathbf{S}\,x_4\,x_5)))\,x_6) \not\approx x_1\,(x_4\,x_6\,(x_5\,x_6))}\ \mathbf{S}\text{XX-NARROW}}{C_6 = \neg(\mathsf{sk}_1\,(\mathsf{sk}_2\,(\mathbf{B}\,x_1\,(\mathbf{S}\,x_4\,\mathbf{I})))\,x_6) \not\approx x_1\,(x_4\,x_6\,x_6)}\ \mathbf{I}\text{-NARROW}}{\bot}\ \text{EqRes}$$

This example demonstrates how higher-order unification is simulated by narrowing with the combinator equations.

5.3.2 | SYN997^1.p

```
thf(conj,conjecture,(
    ? [E: ( $i > $o ) > $i] :
    ! [P: $i > $o] :
      ( ? [Y: $i] :
          ( P @ Y )
     => ( P @ ( E @ P ) ) ) ) ).
```

Figure 5.2: SYN997^1.p

The problem SYN997^1.p, displayed in Figure 5.2, requires reasoning about choice to solve. In my syntax, the problem is written $\exists x : (i \to o) \to o. \forall y : i \to o. \exists z : i. y\, z \Rightarrow y\,(x\, y)$. The problem posits the existence of a choice function and is straightforward to solve. It is included here to demonstrate reasoning about choice. Negating and clausifying the problem results in two clauses:

$$C1 = \mathsf{sk}_1\, x_1\, (\mathsf{sk}_2\, x_1) \approx \top$$
$$C2 = \mathsf{sk}_1\, x_1\, (x_1\, (\mathsf{sk}_1\, x_1)) \approx \bot$$

Where sk_1 and sk_2 are Skolem functions.The proof then proceeds as follows:

$$\dfrac{\dfrac{\mathsf{sk}_1\, x_1\, (\mathsf{sk}_2\, x_1) \approx \top \qquad \dfrac{\mathsf{sk}_1\, x_1\, (x_1\, (\mathsf{sk}_1\, x_1)) \approx \bot \quad \dfrac{\mathsf{sk}_1\, x_1\, (x_1\, (\mathsf{sk}_1\, x_1)) \approx \bot}{\mathsf{sk}_1\, x_1\, x_2 \approx \bot \vee \mathsf{sk}_1\, x_1\, (\varepsilon\, \mathsf{sk}_1\, x_1) \approx \top}\,\textsc{Choice}}{\bot \approx \top \vee \mathsf{sk}_1\, \varepsilon\, x_1 \approx \bot}\,\textsc{Sup}}{\mathsf{sk}_1\, \varepsilon\, x_1 \approx \bot}\,\textsc{TrivIneqRemoval}}{\dfrac{\bot \approx \top}{\bot}\,\textsc{TrivIneqRemoval}}\,\textsc{Sup}$$

5.3.3 | SYO252^5.p

```
thf(cTHM123B,conjecture,(
    ! [Xp: $o] :
    ? [Xf: $o > $o] :
      ~ ( Xf @ Xp ) ) ).
```

Figure 5.3: SYN997^1.p

Problem SYO252^5.p states that there exists a function from Booleans to Booleans that is always false. This is easy to prove, since it merely requires synthesising the constant function $\lambda x. \bot$. I include it to here to demonstrate extended narrowing. The clausified negated conjecture is $x\, \mathsf{sk} \approx \top$.

$$\dfrac{\dfrac{\dfrac{x\,\mathsf{sk} \approx \top}{\mathbf{K}\,\bot\,\mathsf{sk} \approx \top}\,\textsc{ExtendedNarrow}}{\bot \approx \top}\,\textsc{CombDemod}}{\bot}\,\textsc{TrivIneqRemoval}$$

Chapter 6

Vampire Implementation

> The greatest strategy is doomed if it's implemented badly.
>
> *Attributed to Bernhard Riemann*

The selecting combinatory superposition-a and -b calculi have been implemented in the first-order theorem prover Vampire [83]. As the no-select version of the calculi was a late theoretical addition, it has yet to be implemented. Numerous variants of each of the -a and -b calculi have also been implemented by adding modified versions of the rules and adding options to control the behaviour of the rules. The Vampire theorem prover has been introduced in Section 1.3 of the Introduction. Here, I provide further details relating to the prover before describing the changes made to support reasoning about combinators using the calculi described above. When presented with a problem, Vampire parses the problem into its internal representation, preprocesses the problem and finally carries out proof search using the given clause algorithm. Vampire consists of some roughly 200,000 to 300,000 lines of code. The core data structures of Vampire have undergone a number of revisions and are highly optimized.

6.1 | Preprocessing

Preprocessing takes a parsed problem and converts it to clausal normal form. Historically this was done by repeatedly passing through the clause set and carrying out various operations. Recently, a new method of clausification that works via a single top down pass through the formulas has been implemented in Vampire [93]. In my work, I have not updated this new method at all and therefore do not discuss it further. The main steps in the classical preprocessing procedure are:

- Rectification, which essentially renames variables in a formula such that all variables are distinct.

- Simplifying formulas that contain $\bot$ or $\top$. In essence, this involves carrying out BOOLSIMP during preprocessing.
- Flattening formulas.
- Transforming formulas to equivalence negation normal form (ENNF) and flattening a second time. A formula is in ENNF if it contains no implications and all negations occur before atomic formulas.
- Transforming formulas to negation normal form (NNF) and flattening. A formula is in NNF if it is in ENNF and moreover does not contain bi-implication or exclusive or.[1]
- Skolemization
- Conversion to CNF

Converting to NNF can result in an exponential increase in the number of clauses. Thus, this step can be preceded by a step called *renaming*. Renaming aims to reduce the number of clauses produced by replacing an arbitrary formula with an atomic formula and adding a definition for the newly introduced atomic formula. Different naming schemes exist. Vampire's method of renaming is to set a naming threshold n, and rename a subformula if clausifying its parent would lead to more than n clauses. For example, consider the formula $(f \wedge g) \Leftrightarrow (f' \wedge g')$. Assume a naming threshold of 3. Clausifying the formula $(f \wedge g) \Leftrightarrow (f' \wedge g')$ would lead to 4 clauses and therefore, in the Vampire approach one of its subformulas is renamed. Assume the right subformula is renamed. This leads to two formulas, $(f' \wedge g') \Leftrightarrow \mathsf{p}(\overline{x})$ and $\mathsf{p}(\overline{x}) \Leftrightarrow (f' \wedge g')$ where $\overline{x}$ are the free variables of $(f' \wedge g')$. Renaming as described above can be improved by making it *polarity aware*. This means using a bi-implication in the defining formula only if the formula being defined occurs with both positive and negative polarity. Where this is not the case, an implication in the relevant direction is used instead. Details of polarity aware naming can be found in [126]. In general, deciding when to rename is a difficult problem without a simple solution [126].

There are other optional steps that can be carried out during preprocessing such as the removal of unused function and predicate definitions.

6.2 | Terms, Literals and Clauses in Vampire

During proof search, Vampire only deals with clausified problems. Therefore, as with any first-order theorem prover, the most crucial design decision is how to represent terms, literals and clauses. Figure 6.1 displays a simplified version of Vampire's term representation via a pseudo-UML diagram. A term is represented by the `TermList` class. An instance of the `TermList` class uses a union data structure to store either an integer

[1] Some authors define NNF to include the removal of bi-implications [126, Section 3.2]. Others, such as Andrews [2], do not consider the removal of bi-implications to be part of placing a formula into NNF. With the former definition converting to NNF can result in an exponential increase in formula size, whilst with the latter it cannot.

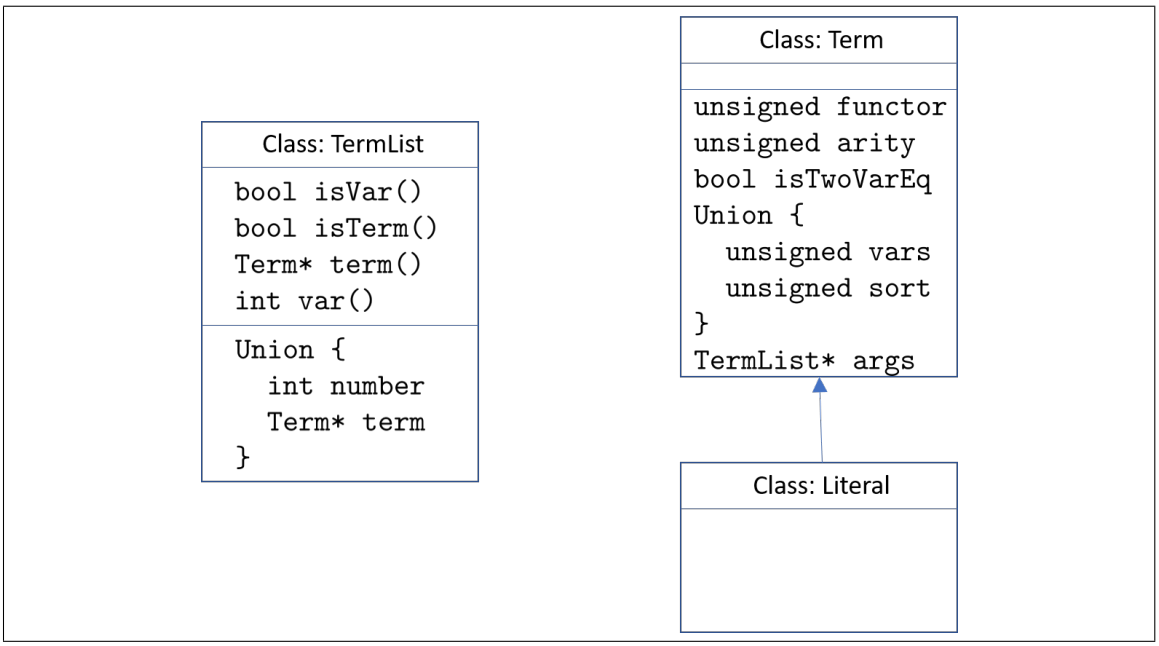

Figure 6.1: Representation of terms in Vampire

representing a variable, or a pointer to an instance of the `Term` class. The `Term` class represents non-variable terms. Since a first-order term is either a variable or a symbol followed by n arguments, the use of a union reduces the space used.

The `Term` class contains a number of attributes. Important amongst these is the `functor` attribute which represents the head symbol of the term. It is an unsigned integer and is linked to the symbol's name via the signature which is to be discussed later. The `arity` field stores the arity of the head symbol. The array `args` stores pointers to `TermList` objects representing the arguments of the head symbol. The array is of length arity $+1$. One member of the `args` array is a dummy `TermList` object used to store meta-information, hence the $+1$. Memory locations for arguments are contiguous and arguments are stored in reverse order with the first argument being stored at `args[arity]`. The meta-data is stored at `args[0]` and acts as an end of arguments marker.

As mentioned above, Vampire's core data structures are highly optimised, particularly in terms of space usage. I provide two examples here. Vampire makes use of the fact that the two least significant digits of a memory pointer are always `00` to differentiate between variables, special variables and pointers to `Terms`. Special variables are used exclusively in substitution trees and need to be disjoint from standard variables.

Figure 6.1 shows that the `Term` class contains a field `isTwoVarEq`. This field is used by the `Literal` class which inherits from `Term`. The only case where an instance of `Literal` needs to store the sort of its arguments is if both arguments are variables. In all other cases, the sort can be worked out. Vampire very cleverly optimises space by using a union to store either the sort of variables for a two variable equality, or the number of variables contained in the term for all other terms and literals.

In Vampire, terms are perfectly shared and immutable. This is achieved by maintaining a hash map of terms. When a term is created a check is performed to see whether it already exists in the hash map. If it does, the existing term is used and the new term is deleted. Otherwise, the new term is inserted into the hash map. The benefit of perfect term sharing is that term comparisons can be achieved by constant time pointer comparisons.

Literals are represented in Vampire as terms with a few separate methods and properties. A clause is essentially an array of pointers to literals. Clauses are immutable, and when a clause is simplified, the old clause is deleted and a new one created.

6.3 | Sorts and Types

Vampire was originally developed as a prover for untyped first-order logic. It was subsequently updated to handle monomorphic first-order logic. Atomic types otherwise known as sorts are represented by unsigned integers. Types are stored as arrays of unsigned integers. Types, like terms, are perfectly shared with identical types being represented by the same object. A signature class links symbols with their types, arities and names. For a function symbol, its type is an array of length 1+ its arity. The unsigned integers represent its argument and return types. For predicate symbols the type is an array of length arity. Recall that in Vampire, symbols are represented as unsigned integers. The core of the `Signature` class is an array of `Symbol` classes with the ith entry of the array containing information about symbol i. This design allows for constant time access to a symbol's type, name etc. Figure 6.2 displays how the term $\mathsf{f}(x_1, \mathsf{a}, x_0, \mathsf{a}, x_0)$ is stored in memory.

Types are mainly used during parsing to ensure well-typedness of the original problem. Most inferences preserve well-typedness, so there is no need to check types during proof search. The exception to this is superposition from a variable wherein it is necessary to check that the variable and the superposed term are of the same sort. It is for this reason that it is necessary to record the sort of variables in two variable equalities as described above.

6.4 | Indexing Data Structures

A term indexing data structure can be defined formally in the manner of Sekar et al. [104]. Let R be a relation over terms and l be a term called the *query term*. Let L be a set of terms. A term index is a data structure designed to facilitate the efficient retrieval of all terms $t \in L$ such that $R(l, t)$. Some typical retrieval conditions in paramodulation based theorem proving are:

- Unifiability, used in superposition;
- Being an instance of (matching), used in demodulation;
- Being a generalization of, used in subsumption.

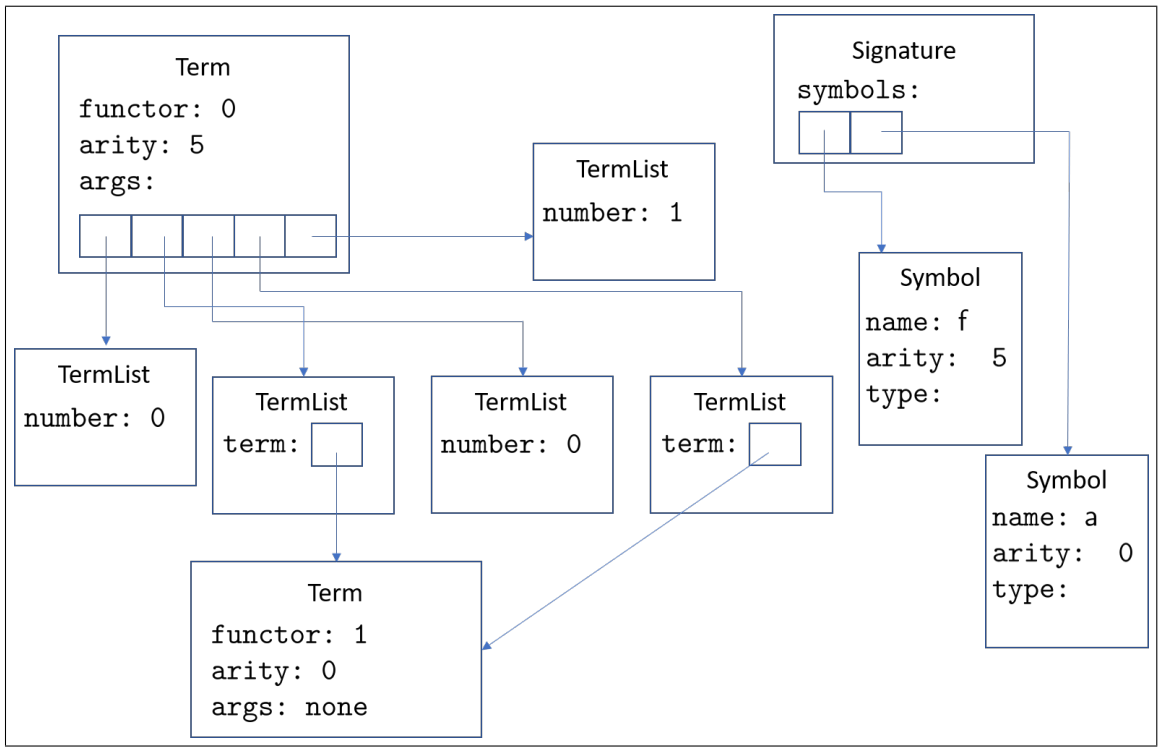

Figure 6.2: How the term $\mathsf{f}(x_1, \mathsf{a}, x_0, \mathsf{a}, x_0)$ is stored in memory by Vampire

Many different indexing structures have been developed with different strengths and weaknesses. Some are better suited to some operations, whilst others are quite general purpose. Some well known indexing data structures are discrimination trees [104], fingerprint indices [102], context trees [60], code trees [95] and substitution trees[64]. Despite the fact that the Vampire prover pioneered the use of code trees, by the time the work under discussion started, substitution trees had become the major indexing mechanism used. Substitution trees are used in Vampire for finding unification partners, generalisations and instances.

In any prover the choice of indexing structure used is crucial. A substitution tree is a tree-like data structure that stores substitutions at nodes. This allows a substitution tree to be more compact than a discrimination tree, which stores symbols at nodes. Vampire uses a particular version of substitution trees known as *linear downward* substitution trees. I describe the indexing data structure here based on the presentation provided in [70] and [64].

6.4.1 | Substitution Trees

Substitution trees are based on the principle of sharing common context between terms stored in the index, as opposed to common prefixes as with discrimination trees. Each node in the tree stores a substitution and by composing all the substitutions in a path from the root to a leaf, a term stored in the index is formed. Substitutions inside the tree are represented using variables known as *special variables*. Let $\mathcal{V}^* = \{*_0, *_1 \ldots\}$ be an infinite

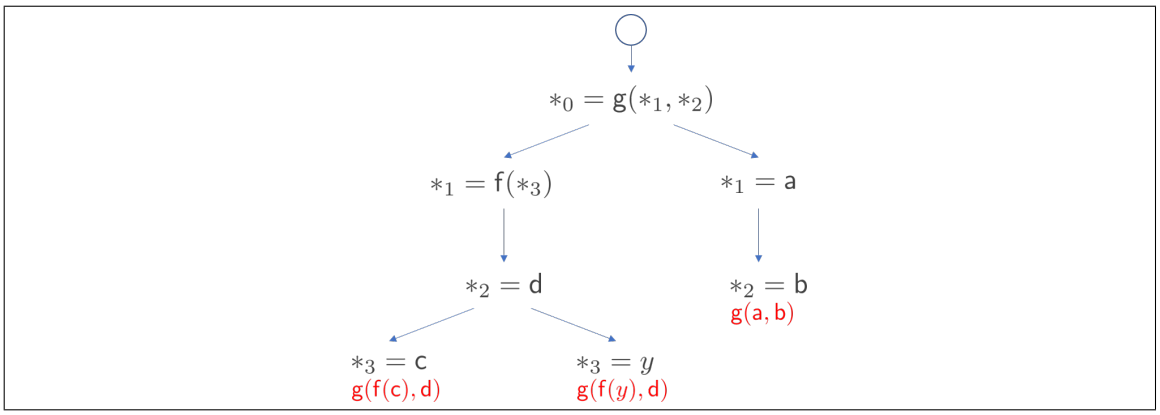

Figure 6.3: A substitution tree

set of special variables disjoint from normal term variables (recall that standard Vampire supports monomorphic FOL and therefore contains no type variables). Figure 6.3 displays a simple substitution tree storing three terms. Following Graf [64], a substitution tree can be defined formally as follows:

Definition 45 (Substitution Tree)**.** A substitution tree is a pair (σ, Σ) where σ is a substitution and Σ is an ordered set of substitution trees. Moreover, the following conditions have to be met:

- A node is either a root $(\{\}, \Sigma \neq \emptyset)$, a leaf $(\sigma \neq \{\}, \emptyset)$, or an intermediate node $(\sigma \neq \{\}, \Sigma \neq \emptyset)$
- Let $(\sigma_1, \Sigma_1), \ldots (\sigma_n, \emptyset)$ be a path from the root in a tree to a leaf. Then it must be that $\mathsf{im}(\sigma_1 \circ \ldots \circ \sigma_n) \subseteq \mathcal{V}$ where $\mathsf{im}(\sigma)$ is the image of σ. In words, this condition states that no special variables can occur in the composition of substitutions from the root to a leaf.
- Let $(\sigma_1, \Sigma_1), \ldots (\sigma_n, \emptyset)$ be a path from the root in a tree to a leaf. Then, for all i and j such that $i < j \leq n$, $\mathsf{dom}(\sigma_i) \cap \mathsf{dom}(\sigma_j) = \emptyset$ where $\mathsf{dom}(\sigma)$ is the domain of σ. In words, all variables are bound at most once in a path from a root to a leaf.

A substitution tree is *linear* if in addition to the conditions listed above it satisfies the property that each special variable appears at most once in the image of a substitution in the tree. For a substitution tree to be *downwards*, it must satisfy two further properties.

- Let $d = (\sigma, \Sigma)$ be a node in a tree. Then, the size of $\mathsf{dom}(\sigma)$ is 1 unless d is the root. Let $(\sigma_1, \Sigma_1), \ldots, (\sigma_1, \Sigma_n)$ be the children of d. Then, $\mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2) = \cdots = \mathsf{dom}(\sigma_n)$. In words, every node binds one variable and the children of a node all bind the same variable called the *selected* variable of the node.
- Let (σ, Σ) be a node in a tree and $(\sigma_1, \Sigma_1), \ldots, (\sigma_1, \Sigma_n)$ its children. Further, let $\sigma_1 = \{*_i \rightarrow t_1\}, \sigma_2 = \{*_i \rightarrow t_2\}, \ldots, \sigma_n = \{*_i \rightarrow t_n\}$. Then, for all i and j such that $i \neq j$, $head(t_i) \neq head(t_j)$. In words, no two children bind the selected variable to terms with the same head symbol.

The easiest method of understanding the definitions above is via examples. In Figure 6.3, if on the right branch $*_2$ was bound before $*_1$, the tree would no longer be downwards since the two children of the root bind different variables. Likewise, if the root stored the substitution $*_0 = \mathsf{g}(*_1, *_1)$ the tree would not be linear since the special variable $*_1$ appears more than once in the image of a substitution. In Figure 6.3 again, for the right branch to represent the term $\mathsf{g}(\mathsf{f}(\mathsf{e}), \mathsf{b})$, it would not be correct to change the right child of the root to $*_1 = \mathsf{f}(\mathsf{e})$. If that was done, both the left and the right child of the root would bind $*_1$ to a term with f as its head. This contradicts the second condition for downwardness.

Insertion and deletion of terms from a downward substitution tree can be carried out without backtracking. As my work does not touch on these operations, I do not bother presenting the algorithms here. However, they can be found in [104]. In order to maximize sharing within a tree, terms are *normalised* before insertion [104]. For example, $\mathsf{f}(x, y)$ and $\mathsf{f}(y, z)$ would both be stored in the tree as $\mathsf{f}(x_1, x_2)$. In Vampire, denormalized versions of terms are stored in each leaf to allow the normalized and denormalized variables to be linked. Leaf nodes can be used to store other data as well, such as the clauses and literals which the indexed terms belong to. Such data is referred to as *leaf data*.

Algorithm 2, for a given query term t, recurses through a tree and, for each term s in the tree such that t and s are unifiable, returns a pair consisting of a unifier and a leaf data. In the algorithm, unfiable is a procedure that takes two terms and and an empty substitution σ. If the two terms are unifiable, the function returns $true$ and stores the mgu in σ. Otherwise it returns $false$. In Vampire, the calculation of the mgu is carried out using a polynomial time variant of Robinson's famous unification algorithm [70]. The overall algorithm is presented using recursion. The Vampire implementation uses iteration instead and uses stacks to control the backtracking. An example best demonstrates how the algorithm works.

Example 25. Consider the tree shown in Figure 6.3. Let the query term be $t = \mathsf{g}(\mathsf{f}(x), \mathsf{d})$. By line 3 of the algorithm, $\sigma = \{*_0 \rightarrow \mathsf{g}(\mathsf{f}(x), \mathsf{d})\}$. Line 8 then attempts to unify $*_0\sigma = \mathsf{g}(\mathsf{f}(x), \mathsf{d})$ with $\mathsf{g}(*_1, *_2)$. The most general unifier produced on line 9 is $\dot{\sigma} = \{*_1 \rightarrow \mathsf{f}(x), *_2 \rightarrow \mathsf{d}\}$. After line 9 is executed, $\sigma = \{*_0 \rightarrow \mathsf{g}(\mathsf{f}(x), \mathsf{d}), *_1 \rightarrow \mathsf{f}(x), *_2 \rightarrow \mathsf{d}\}$. Then follow the recursive calls on line 14. Assume that the left subtree is explored first. Line 8 attempts to unify $*_1\sigma = \mathsf{f}(x)$ with $\mathsf{f}(*_3)$. This succeeds with unifier $\dot{\sigma} = \{x \rightarrow *_3\}$. The algorithm continues in this manner. When an attempt is made to explore the right branch, a failure is encountered since $*_1\sigma = \mathsf{f}(x)$ is not unifiable with a and no further exploration of this subtree takes place.

Algorithm 2 Retrieval of unification partners

```
1:  function search_for_unifiers(t, tree)
2:      let *_0 be the selected variable of the root of tree
3:      let σ := {*_0 → l}
4:      let L be a set of pairs of leaf data and substitution; L := {}
5:      search(σ, tree, L)
6:      Return L

7:  function search(σ, tree, L)
8:      let tree = (σ', Σ)
9:      if σ' = {} (tree is root) then
10:         for tree* ∈ Σ do search(σ, tree*, L)
11:     else
12:         let σ' = {*_i → l}
13:         if unifiable(*_i σ, l σ, σ̇) then
14:             σ := σ ∪ σ̇
15:             if tree is leaf then L := ⟨σ, tree.data⟩ ∪ L
16:             else
17:                 for tree* ∈ Σ do search(σ, tree*, L)
```

6.5 | Given Clause algorithm

Vampire is a saturation based prover implementing the given clause architecture. The architecture was introduced in Chapter 1, but as Vampire implements a unique variant of it, I expand on the overview provided there. As with most saturation based provers, Vampire converts any problem that it is given to clausal normal form. Unlike other provers, Vampire maintains three sets of clauses during proof search. These are the *unprocessed* clauses, the *passive* clauses and the *active clauses*. Initially, all clauses are added to the unprocessed set. Whilst the unprocessed set contains clauses, clauses are removed from the set and various simplification inferences involving the clause are carried out. If the clause cannot be simplified, it is added to passive set. If the clause can be simplified in some way, the simplified conclusion is added back to unprocessed. Once unprocessed is empty, a clause is selected from passive and added to active. Then, all possible inferences between this clause and clauses in active are carried out and the conclusions placed in unprocessed. The algorithm then loops back to the beginning. The place in the loop at which simplification inferences are carried out and the nature of these inferences leads to two versions of the algorithm, the *Otter* loop and the *Discount* loop presented in Algorithms 3 and 4.

The main difference between the Otter and Discount algorithms is that in the Otter algorithm, simplification inferences are carried out between a clause from the unprocessed set and all clauses in both active *and* passive, whilst in the Discount algorithm only clauses from the active set are used. The Discount algorithm also employs a second round of

Algorithm 3 The Otter loop

```
1:  let N be a set of clauses
2:  let active, passive, unprocessed be sets of clauses
3:  let given, new be clauses
4:  active := ∅
5:  passive := ∅
6:  unprocessed := N
7:  loop
8:      while unprocessed ≠ ∅ do
9:          new = pop(unprocessed)
10:         if new = ⊥ then Return unsat
11:         if retained(new) then
12:             simplify new by clauses in active ∪ passive
13:             if new = ⊥ then Return unsat
14:             if new not simplified then
15:                 simplify clauses in active and passive using new
16:                 move simplified clauses to unprocessed
17:                 add new to passive
18:     if passive = ∅ then Return satisfiable
19:     given := select(passive)
20:     move given from passive to active
21:     unprocessed := infer(given, active)
```

simplification. The retained test on line 11 of the algorithm refers to simplification rules with a single premise such as DUPLITREM. Only if none of these are applicable to a clause does it pass the test.

Vampire implements a third variant of the loop, the so called *limited resource strategy*. This variant is basically the same as the Otter loop, but periodically, the prover estimates which clauses in passive cannot be processed within the time limit and jettisons these.

In all versions of the architecture, clause selection is a crucial feature. The standard method of selecting clauses in Vampire is based on the concepts of age and weight. Age represents how old a clause is whilst weight represents the size of the clause. Vampire maintains two priority queues for clauses in passive based on age and weight. For every m oldest clauses selected from the age queue, n lightest clauses are selected from the weight queue. The values m and n can be varied. More complex clause selection schemes have recently been introduced to Vampire, but do not affect the current work [110].

Finally, Vampire is a portfolio solver. When attempting to find a proof, its most effective method is not to run with a single set of parameters for some time period, but rather to break the time up into shorter slots and in each slot run a set of parameters. A single set of parameters is known as a *strategy*. For example, Vampire runs strategies which utilise the Discount loop and others which use the Otter loop. Discovering useful strategies and evaluating the usefulness of parameters is an area of research by itself [92].

Algorithm 4 The Discount loop

let N be a set of clauses
let $active, passive, unprocessed$ be sets of clauses
let $given, new$ be clauses
$active := \emptyset$
$passive := \emptyset$
$unprocessed := N$
loop
    while $unprocessed \neq \emptyset$ **do**
        $new = \mathsf{pop}(unprocessed)$
        if $new = \bot$ **then Return** unsat
        if $\mathsf{retained}(new)$ **then**
            simplify new by clauses in $active$
            if $new = \bot$ **then Return** unsat
            if new not simplified **then**
                simplify clauses in $active$ using new
                move simplified clauses to unprocessed
                add new to $passive$
    if $passive = \emptyset$ **then Return** satisfiable
    $given := \mathsf{select}(passive)$
    simplify $given$ by clauses in $active$
    if $given = \bot$ **then Return** unsat
    if $given$ not simplified **then**
        simplify clauses in $active$ using $given$
        move simplified clauses to unprocessed
        move $given$ from $passive$ to active
        $unprocessed := \mathsf{infer}(given, active)$

6.6 | Updating Vampire to Support Polymorphism

The first step towards supporting applicative first-order logic was to update the prover to support polymorphism. This work is described in the short paper by Bhayat and Reger [33]. In this section, I summarise the paper. The Vampire source code for all extensions described in this chapter is located in GitHub in the `polymorphic_vampire` branch of the Vampire repository.[2]

To support polymorphism, modifications had to be carried out in three main areas. Firstly, changes had to be made to the concept of types in Vampire. Secondly, some inferences had to be modified slightly and finally preprocessing required consideration. I describe the work undertaken in this order.

Figure 6.4 is an example of a problem expressible in TPTP TF1 syntax [35, 112]. I use this problem to illustrate the implementation. The major change undertaken was to replace atomic types with terms. In monomorphic Vampire each atomic type in the

[2] `https://github.com/vprover/vampire/tree/polymorphic_vampire`

map is a type constructor of arity 2

$\mathsf{lookup} : \Pi\alpha_1, \alpha_2 : (\mathsf{map}(\alpha_1, \alpha_2) \times \alpha_1) \to \alpha_2$
$\mathsf{update} : \Pi\alpha_1, \alpha_2 : (\mathsf{map}(\alpha_1, \alpha_2) \times \alpha_1 \times \alpha_2) \to \mathsf{map}(\alpha_1, \alpha_2)$
Axiom 1:
$\forall \alpha_1 : \mathsf{sup}, \alpha_2 : \mathsf{sup}, x : \mathsf{map}(\alpha_1, \alpha_2), y : \alpha_1, z : \alpha_2.$
$\quad \mathsf{lookup}\langle \alpha_1, \sigma_2 \rangle((\mathsf{update}\langle \alpha_1, \alpha_2 \rangle(x, y, z), y) \approx z$
Axiom 2:
$\forall \alpha_1 : \mathsf{sup}, \alpha_2 : \mathsf{sup}, x : \mathsf{map}(\alpha_1, \alpha_2), y : \alpha_1, z : \alpha_2.$
$\quad \mathsf{update}\langle \alpha_1, \alpha_2 \rangle(\mathsf{update}\langle \alpha_1, \alpha_2 \rangle(x, y, z), y, z) \approx \mathsf{update}\langle \alpha_1, \alpha_2 \rangle(x, y, z)$

Figure 6.4: Sample rank-1 polymorphic problem

input problem is stored as an unsigned integer. In polymorphic first-order logic, atomic types have all the structure of terms. Therefore, it makes sense to replace atomic types with terms. Types then become arrays of terms rather than unsigned integers. A special constant sup, represents the type of all types in Vampire.

Let f be a function symbol of type $\Pi\overline{\alpha}_n.\,(\overline{\tau}_m) \to \tau$. The type of a term of the form $\mathsf{f}\langle\overline{v}_n\rangle(\overline{s}_m)$ can be constructed by substituting α_1 for v_1, α_2 for v_2 and so on in τ. For example, the type of $\mathsf{update}\langle\mathsf{int}, i\rangle(\mathsf{map}, 1, X)$ would be $\mathsf{map}(\mathsf{int}, i)$. For a term $\mathsf{f}\langle\overline{\tau}\rangle(\overline{s})$, the type of the ith term argument can be calculated in the same way. The one problem that arises is with two variable literals such as $x \approx y$. In this case, the type of the terms x and y have to be stored as a separate field in the literal as mentioned previously.

The elegance of treating types as terms can be gauged when attention is turned to unification. Had types and terms been kept separate, unifying terms would have become an involved process requiring the unification of term and type arguments separately. As it is, type unification comes for 'free' with one caveat, as shall be seen. Consider unifying the terms $\mathsf{update}\langle\mathsf{int}, i\rangle(\mathsf{map}, 1, X)$ and $\mathsf{update}\langle y, z\rangle(\mathsf{map}, z', \mathsf{a})$. The existing unification procedure in Vampire can handle this and return the type and term unifier $\{y \to \mathsf{int}, z \to i, z' \to 1, x \to \mathsf{a}\}$. The one hitch occurs when unifying a term with a variable. As variables carry no type information, a second call must be made to the unification procedure to ensure that the type of the variable and the type of the term are unifiable.

As far as changes to inferences are concerned, no updates were required for inferences that do not work at subterms such as resolution or equality factoring. For inferences that work at subterms such as superposition and demodulation, I modified the iterators that return candidate subterms so that they do not return type arguments, as superposition into types is unnecessary. The modifications required to support polymorphism are relatively light. They also (in theory) add no overhead when dealing with monomorphic problems. In this case all types are constants and unifiabilty checking of types in the variable case

degenerates to a syntactic equality check.

Finally, regarding preprocessing, implementing Skolemization posed a subtle issue. A Skolem function must be applied to the free term and *type* variables in its context. For example, the Skolemization of $\forall x : \mathsf{int}, y : sup.\, \exists z : i.\, \mathsf{p}\langle y\rangle(x, z)$ would be $\forall x : \mathsf{int}, y : sup.\, \exists z : i.\, \mathsf{p}\langle y\rangle(x, \mathsf{sk}\langle y\rangle(z))$. This required the updating of the notion of free variable within the code (e.g., when iterating over the free variables of a formula).

6.7 | Supporting Higher-Order Logic

One of the aims of this PhD project was to extend Vampire to support HOL. Building on the support for polymorphism, this was achieved through a number of steps. I updated the parser to parse statements of higher-order logic expressed in TPTP TH0 and TH1 formats [75]. The statements are parsed into what are called *special terms* in Vampire. Prior to my work, these terms were used to represent `if-then-else` statements and other constructs not supported within standard first-order logic. I introduced new special terms to represent λ-expressions and applications. All special terms are translated away during preprocessing. The specials representing λ-expressions are removed during preprocessing using the translation given in Section 2.7 of Chapter 2. Obviously applicative terms, which can have variables at the head and function symbols which are partially applied, are not supported by Vampire's first-order core either. Therefore, I had to decide how to deal with these. The two main possibilities were to either translate applicative terms into first-order terms using the applicative encoding [86], or to update Vampire's core data structures and algorithms to support applicative terms directly, in a manner similar to how this has been done in the E prover [119].

6.7.1 | Applicative Encoding

As the name suggests, the applicative encoding is an encoding of applicative first-order logic into standard polymorphic first-order logic. Types and terms as defined in Section 2.3 of Chapter 2 are encoded into types and terms as defined in Section 2.1 of the same chapter. To avoid confusion, the symbol $\rightarrow$ used to demarcate between the argument and return types in Section 2.1 is replaced by the symbol $>$.

The translation is relatively straightforward. Let $(\Sigma_{\mathsf{ty}}, \Sigma)$ be the signature of an applicative first-order logic. Define a polymorphic FOL signature $(\Sigma_{\mathsf{ty}}^{\mathsf{poly\_fo}}, \Sigma^{\mathsf{poly\_fo}})$ such that $\Sigma_{\mathsf{ty}}^{\mathsf{poly\_fo}} = \Sigma_{\mathsf{ty}}$ and $\Sigma^{\mathsf{poly\_fo}} = \Sigma$. In the polymorphic first-order logic, the type constructor $\rightarrow$ has no special semantics. The applicative type $i \rightarrow i \rightarrow i$ is represented as a polymorphic first-order type $\rightarrow (i, \rightarrow (i, i))$.

To the signature $\Sigma^{\mathsf{poly\_fo}}$, a binary function $\mathsf{app} : \Pi\alpha_1, \alpha_2.\, (\rightarrow (\alpha_1, \alpha_2), \alpha_1) > \alpha_2$ is added. This function is used to simulate application. For example, consider the applicative

term f a where f has type $i \to i$ and a is of type i. This is translated to the first-order term $\mathsf{app}\langle i, i\rangle(\mathsf{f}, \mathsf{a})$. In the encoding, the only symbol that takes term arguments is app. Using the applicative encoding, no changes beyond those described in section 6.6 need to be made to Vampire's type and term representation.

The applicative translation possesses the virtue of simplicity, but also suffers from a number of disadvantages. Chief amongst these is the need to recurse or iterate through the arguments of a term in order to access its head symbol. Other objections that have been raised against the encoding is that the app symbol clutters data structures and causes algorithms to behave in unusual ways. I will return to these objections later.

6.7.2 | Flat Representation

The second method of representing types and terms is to use their flattened form. Thus, $\to$ is not treated as a binary type constructor, but a type constructor that can take an arbitrary number of arguments. Likewise, the term $x\,\mathsf{a}\,\mathsf{b}$ is represented as a head x with two arguments a and b. This is the path that Vukmirović et al. followed when updating the E theorem prover to deal with applicative first-order logic [119]. They updated E's data structures and algorithms to support variables occurring at head position and partial application. This was no easy task since many of the algorithms make assumptions about variables not being applied and about function symbols having a full complement of arguments.

6.7.3 | Vampire's Implementation

In Vampire the challenges in implementing a flat representation are severe. I mention just the most major. Vampire's term representation, as described above, assumes some fixed maximum arity for each function symbol. In applicative first-order logic, this assumption is not valid. A symbol $\mathsf{f} : \Pi\alpha.\,\alpha \to \alpha$ can have an arbitrary number of arguments. It is possible to support this by changing `args` to a dynamic array rather than a fixed length array. However, numerous places in the code assume that the first argument is to be found at `args[arity]` and that it is safe to iterate through the arguments until the end of arguments marker is reached. To update all these places in the code is a mammoth task.

For this reason, in my implementation I decided to use the applicative encoding. Despite this decision, in the remainder of this chapter I present terms using the applicative syntax without the explicit app operator. The use of the applicative encoding has the advantage that most of Vampire's algorithms and data structures required no modification at all. However, it does come with the disadvantages mentioned. I do not believe that these issues are as severe as they are presented to be in some of the literature. Below, I explain why.

- Judicious translation from the applied representation to the flat representation can help mitigate the need to iterate through a term's arguments in order to access the head (or even any argument but the final one). Obviously, translating to a flat representation involves iterating through the term structure, but if it is known in advance that a number of head accesses will be required, this is still an efficiency gain. I have implemented this in Vampire.
- As far as the app symbol cluttering data structures is concerned, the usage of app can at most double the size of terms. Since space is not normally the limiting factor in modern prover performance, I do not feel that this is a crucial problem.
- The final objection raised is that the usage of app causes algorithms to behave unusually. In particular, any algorithm that branches based on the head symbol of a term will perform oddly, since for most terms app is the head symbol. However, this is only the case if app is treated as part of the term rather than as syntactic sugar.[3] In Vampire this is not the case and, for the purpose of the term order in particular, apps are simply ignored. They do not contribute to a term's weight nor are they considered when comparing heads.

6.8 | Implementing the Combinatory Superposition-a and -b Calculi

The first step to implementing the calculi in Vampire involved implementing the ordering introduced in Chapter 3.

6.8.1 | Implementing the Modified KBO

I have implemented the simple but not so powerful $>_{\mathsf{ski1}}$ ordering. It remains to implement the more powerful $>_{\mathsf{ski}}$ ordering. The $>_{\mathsf{ski1}}$ ordering is based on the length of the longest weak reduction from a term. In order to increase the efficiency of calculating this quantity, I implemented the order using caching and lazy evaluation. I added a field `maxRedLen` to the `Term` class. On creation of a term, this field is set to 0 if not easily calculable. When the term ordering needs to know the maximum reduction length from a term, it checks the field to see if the quantity has been set. If it has not been set, the maximum weak reduction is calculated and the `maxRedLen` field is set accordingly for the term in question.

To provide an example of what "easily calculable" means, when inserting a term of the form $\mathsf{f}\, t_1\, t_2$ into the term-sharing data structure, a check is made to see if the maximum reduction lengths of t_1 and t_2 have already been calculated. If they have, then the maximum reduction length of the term being inserted is set to the sum of the maximum reduction lengths of t_1 and t_2. If not, it is left unassigned and only calculated at the time

[3] In which case the first objection does not apply

it is required.

Note that calculating the maximum weak reduction for the purpose of the implementing the ordering is not as onerous as it may seem. Due to the presence of the COMB-DEMOD inference, clauses are stored in weak normal form. Since ordering checks occur, whilst carrying out an inference, *after* the application of unifiers, the only weak redexes present are those introduced by the unifiers.

6.8.2 | Implementing the Inference Rules

Implementing the the core inference rules of the calculus was relatively straightforward. For the SUP inference, the only change required was to modify the iterator that returns possible rewritable subterms of a clause to only return first-order subterms. Otherwise, the inference is identical to its first-order counterpart. EQRES and EQFACT are more or less identical to their first-order counterparts and required little modification. Code had to be written for the new inference rules NARROW, ARGCONG and SUBVARSUP. The code can be found in files `Narrow.cpp`, `SubVarSup.cpp` and `ArgCong.cpp` in the source code. As SUBVARSUP is a variation of SUP, implementing it was relatively straightforward and code could be reused.

ARGCONG

Implementing ARGCONG was more difficult. The ARGCONG inference can potentially have an infinite set of conclusions. This is handled in the Zipperposition prover by maintaining a set of T of scheduled inferences. Rather than carrying out all inferences between the given clause and the clauses in active, Zipperposition adds these inferences to a set of scheduled inferences. Periodically, this set is visited and a fair strategy is used to derive conclusions from the set of scheduled inferences. Such an architecture represents a major change for Vampire's optimised code base. Therefore, instead of implementing the version of ARGCONG given in the calculus, the following version with a single conclusion was implemented.

$$\frac{C' \vee s \approx s'}{C'\sigma \vee (s\sigma)\, x \approx (s'\sigma)\, x}\ \textsc{ArgCong}$$

Obviously, this harms completeness. It remains future work to repair this mismatch between the calculus and implementation.

NARROW and EXTENDEDNARROW

NARROW was implemented as follows. Prior to proof search, Vampire sets up a substitution tree T that indexes the left hand sides of combinator equations. When attempting to carry out a NARROW inference on a clause C, an iterator is used to return all nearly

first-order subterms in C. An attempt is then made to unify these with terms in T and then rewrite them using the relevant axiom. Based on an option `narrow`, Vampire can instantiate T with three sets of terms.

- If `narrow` is set to `all`, T is instantiated to contain the left hand side of the five combinator axioms mentioned previously.
- If `narrow` is set to `SKI`, T is initialised to contain only the terms $\mathbf{S}\,x\,y\,z$, $\mathbf{K}\,x\,y$ and $\mathbf{I}\,x$.
- If `narrow` is set to `SK`, T only contains the terms $\mathbf{S}\,x\,y\,z$, $\mathbf{K}\,x\,y$.

Note that the second and third options do not harm completeness since $\{\mathbf{S}, \mathbf{K}\}$ form a complete set of combinators. Instead, using these smaller sets may help curb explosion of clauses at the expense of missing a short proof. The `narrow` option can also be set to `off` in which case no NARROW inferences are carried out at all.

Aside from the core inferences, the rules dealing with Boolean reasoning have also been implemented in Vampire. The EXTENDEDNARROW rule is implemented in a similar manner to the NARROW rule, highlighting their shared heritage [55]. Prior to proof search, a substitution tree is instantiated with the four sets of terms given in Section 5.1.4 of the previous chapter. The main differences between EXTENDEDNARROW and and NARROW are:

- NARROW works at nearly first-order subterms whilst EXTENDEDNARROW works at the top level of literals.
- With EXTENDEDNARROW, after unifying with a term in the tree, no rewriting takes place. Instead the rewriting is carried out by the CNF inferences.

Which of the four versions of EXTENDEDNARROW is carried out is controlled in Vampire via an option `prim_inst_set`. The four values that the option can take are `all`, `all_but_not_eq`, `false_true_not` and `small_set`. The sets of terms related to each of these values do not quite correspond to the sets provided in the previous chapter.

CNF rules

Probably the most interesting engineering challenge was posed by the clausification inferences. As mentioned in the previous chapter (Section 5.1.4), these inferences can be used as generating inferences, simplification inferences or some can be generating and some simplifying. In Vampire, this behavior is controlled by a flag `cnf_on_the_fly` which can take one of seven values.

- If `cnf_on_the_fly` is set to `eager`, clausification takes place during preprocessing. During proof search, clausification inferences are carried out before all other simplification inferences. With all other settings, no clausification takes place during preprocessing.

- If `cnf_on_the_fly` is set to `lazy_simp`, other simplification inferences such as DEMOD take place prior to clausification inferences.
- If `cnf_on_the_fly` is set to `lazy_gen`, all clausification rules are treated as generating rules.
- If `cnf_on_the_fly` is set to `lazy_not_gen`, the $\text{CNF}_\neg$ inference rule is generating whilst the remaining rule are simplifying. This setting is useful in conjunction with the BOOLEQRW inference.
- If `cnf_on_the_fly` is set to `lazy_not_gen_be_off`, the $\text{CNF}_\neg$ rule is generating whilst the $\text{CNF}_\simeq$ rules are switched off altogether. This setting can help in cases where $\text{CNF}_\simeq$ rules lead to an exponential blowup in clause numbers.
- If `cnf_on_the_fly` is set to `lazy_not_be_gen`, the $\text{CNF}_\neg$ and the $\text{CNF}_\simeq$ rules are generating.
- If `cnf_on_the_fly` set to `off` no clausification takes place during proof search.

As discussed by Reger et al. [93], it can be useful to reuse Skolem functions and names. Vukmirović and Nummelin carry out such reuse in Zipperposition [120]. In Vampire, such reuse was not carried out prior to the start of my work. Consider the formulas $\exists x.f$ and $\exists x.g$. Let $\mathsf{sk}\langle\overline{\alpha}\rangle(\overline{x})$ be the witness for the first formula. If there exists a substitution σ such that $g\sigma = f$, it is beneficial to use $(\mathsf{sk}\langle\overline{\alpha}\rangle(\overline{x}))\sigma$ as the witness for the second formula rather than introduce a new Skolem function, since this reduces the size of the signature.[4] Vampire does not implement this for two reasons.

- Since formulas do not occur in Vampire during proof search (only during preprocessing), they are stored in a naive data structure and are not shared. This makes finding generalisations of a particular formula a relatively difficult and expensive operation.
- To reuse Skolem functions fully, generalisation finding needs to be carried out modulo the commutativity and associativity of $\wedge, \sqcup$. This adds complications to the algorithms.

In my implementation of the $\text{CNF}_\exists$ rule, reuse of Skolems does take place. Prior to the start of proof search, a substitution tree T is set up. When attempting to simplify a clause of the form $\exists\langle\alpha\rangle f \approx \top \vee C$, I check whether there exists a term g in T such that g is a generalisation of f. If such a term exists, g's witness is reused for f. If no such term exists, a new witness is created for f and f is then inserted into the tree. To deal with the commutativity and associativity of $\wedge$ and $\sqcup$, whenever a new term is created with one of these connectives as its head symbol, the arguments are ordered based on some arbitrary total ordering on terms. This solution is not perfect. It may be, for example,

[4] Note that the reuse of Skolems does not come without its own issues. Consider the Skolemization of formulas $\exists x.\mathsf{p}(y, x)$ and $\exists x.\mathsf{p}(\mathsf{a}, x)$. Skolemization of the first formula results in $\mathsf{p}(y, \mathsf{sk}(y))$. If the witness is reused, the Skolemization of the second is $\mathsf{p}(\mathsf{a}, \mathsf{sk}(a))$. Without reuse, the Skolemization is $\mathsf{p}(\mathsf{a}, \mathsf{sk})$ and it can be seen that reuse has lead to an unnecessary argument being passed to the witness.

that for the ordering $\succ$ utilised, $\mathsf{b} \succ \mathsf{a} \succ x$. Thus, $\mathsf{a} \sqcup x$ would not be considered a generalisation of $\mathsf{b} \sqcup \mathsf{a}$. It does however remove the issue of terms that are identical modulo the commutativity and associativity of $\wedge$ and $\sqcup$.

Two important questions relating to the naming formulas are when to name and what to name. These are not as easy to answer as it may look. In Zipperposition, the developers chose the following scheme. A count is kept of how many times each non-atomic formula f occurs as the left or right hand side of a literal. Before applying a CNF rule to a clause $f \approx \top \vee C$, f's count is checked and if it is above the threshold, f is renamed. This scheme suffers from the weakness of not considering instances of f that occur as subformulas. For example, let $g = f \simeq f'$. Consider a clause set N such that $C = g \approx \top \vee C'$ is in N, and f appears as one side of a literal in clauses in N three times. Assume a naming threshold of 4. If a clause containing $f \approx \top$ is selected before C, f will not be renamed since it has fewer than threshold occurrences. On the other hand, if C is selected first, $\text{CNF}_{\simeq}$ produces two new occurrences of f at the top level and f will subsequently be renamed.

Another difficulty lies in cascading the name through the clause set once a formula has been named. One possibility is to not cascade the name at all. After a formula f is named, if a clause of the form $f \approx \top \vee C$ is being processed, in attempting to find named generalisations of f, the previous naming of f itself will be discovered. This relies on the relatively expensive matching algorithm to cascade naming. It would be useful to have a more efficient scheme, one that could potentially work at subterms as well. Due to the above and other difficulties, naming of formulas during proof search, though implemented, is not activated in the latest version of Vampire.

Abstraction rules

Implementing the abstraction version of the core inference rules involved modifying Vampire's substitution tree structure and unification algorithm. The underlying ideas are very similar to those presented in some of my previous work with Reger [29]. Before inserting terms into the substitution tree, all terms of Boolean, functional or polymorphic type are replaced by variables. The variables used are disjoint from both normal type and term variables, and also the special variables used to represent substitutions within a tree. I call these variables *very special variables* and use the symbols $\#_1, \#_2, \ldots$ to represent these variables. For example, the term $\mathsf{f}\,(\mathsf{a} \wedge \mathsf{b})$ is transformed into $\mathsf{f}\,\#_1$ prior to insertion into a substitution tree. Associated with each substitution tree is a bidirectional hashmap from very special variables to the terms they represent. I call the process of replacing the subterms of a term with very special variables *hashing*. The opposite process of dereferencing the very special variables is called *dehashing*. An example of a hashed substitution tree can be found in Figure 6.5. In theory, when all terms in a tree that contain some very special variable $\#_i$ are removed from the tree, the hash map should be updated to remove

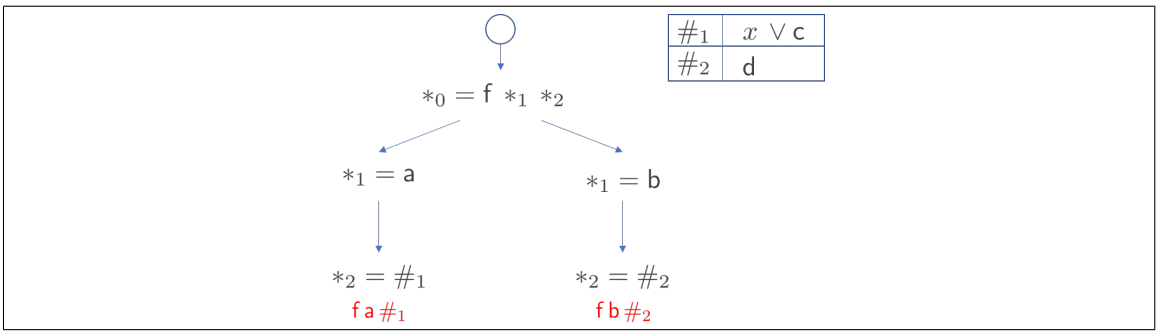

Figure 6.5: Hashed substitution tree

reference to the variable $\#_i$. In practice, doing this complicates the implementation since it necessitates incrementing and decrementing counts related to very special variables as terms are inserted and removed from the tree. In my implementation, I don't bother with this.

The algorithm for finding unification partners (Algorithm 2) is modified to return triples instead of pairs. Each triple contains the unifying substitution, the leaf data and the set of unification constraints. The modified algorithm is presented in Algorithm 5. I updated the variant of Robinson's unification algorithm used by Vampire, to implement unification with abstraction. The modified version is presented in Algorithm 6.

Example 26. Let T be the substitution tree presented in Figure 6.5. Let the query term be $t = \mathsf{f}\,\mathsf{a}\,(\mathsf{c} \sqcup x)$. Hashing the query produces the term $\mathsf{f}\,\mathsf{a}\,\#_3$ and as part of the hashing process, $\#_3$ is added to T's hash map. Function search is called with arguments $\sigma = \{*_0 \to \mathsf{f}\,\mathsf{a}\,\#_3\}$, $tree = T$, $D = \{\}$ and $L = \{\}$. Since $tree$ is the root, a second call is made to the search function, this time with arguments $\sigma = \{*_0 \to \mathsf{f}\,\mathsf{a}\,\#_3\}$, $tree = (\{*_0 \to \mathsf{f}\ *_1\ *_2\}, \Sigma)$, $D = \{\}$ and $L = \{\}$.

Unifying $l\sigma = \mathsf{f}\,*_1\,*_2$ with $*_0\sigma = \mathsf{f}\,\mathsf{a}\,\#_3$ succeeds with $mgu\ \dot{\sigma} = \{*_1 \to \mathsf{a}, *_2 \to \#_3\}$. Two recursive calls to explore the left and the right subtrees are then made, both with $\sigma = \{*_0 \to \mathsf{f}\,\mathsf{a}\,\#_3, *_1 \to \mathsf{a}, *_2 \to \#_3\}$. In the left subtree, the algorithm proceeds by attempting to unify $l\sigma = \mathsf{a}$ with $*_1\sigma = \mathsf{a}$ which succeeds with the empty unifier. There follows a single recursive call that attempts to unify $l\sigma = \#_1$ with $*_2\sigma = \#_3$. This succeeds with the empty unifier and adds the constrain $\mathsf{c} \sqcup x \not\approx x \sqcup \mathsf{c}$ to D. As the algorithm has now succeeded in a leaf node, L is updated with the relevant triple.

In attempting to traverse the right subtree, the algorithm tries to unify $*_1\sigma = \mathsf{a}$ with $l\sigma = \mathsf{b}$ and fails.

As can be seen from the above description and example, very special variables do not really behave like variables at all. They are never bound during unification. Rather, they work as abbreviations or definitions. As such it would make more sense to represent them using constants rather than variables, as was done in my previous work [29]. However, this expands the signature massively and therefore, I take the approach described here. In Vampire, the usage of abstraction rules is controlled by an option `func_ext`.

Algorithm 5 Retieval of unification partners

```
1: function search_for_unifiers(t, tree)
2:     t := hash(t)
3:     let *_0 be the selected variable of the root of tree
4:     let σ := {*_0 → l}
5:     let D be a set of constraints; D := {}
6:     let L be a set of triples of substitution, leaf data and constraints; L := {}
7:     search(σ, tree, L)
8:     Return L

9: function search(σ, tree, D, L)
10:    let tree = (σ', Σ)
11:    if σ' = {} (tree is root) then
12:        for tree* ∈ Σ do search(σ, tree*, L)
13:    else
14:        let σ' = {*_i → l}
15:        if unifiable_with_abstration(*_i σ, lσ, σ̇, D) then
16:            σ := σ ∪ σ̇
17:            if tree is leaf then L := ⟨σ, tree.data, D⟩ ∪ L
18:            else
19:                for tree* ∈ Σ do search(σ, tree*, L, D)
```

- If `func_ext` is set to `abstraction`, the -WA version of the core rules are used. No attempt is made to unify Boolean, functional and polymorphic subterms.
- If `func_ext` is set to `axiom`, the standard version of the core rules are used. The functional extensionality axiom is added to the clause set. Reasoning about Booleans is carried out by FOOLPARAMODULATION or CASESSIMP.
- If `func_ext` is set to `off`, no extensionality reasoning takes place.

6.9 | Higher-Order Schedule

In order to create a higher-order strategy schedule, I used a set of higher-order problems to test the usefulness of particular strategies. The higher-order problems that I mainly used were the TPTP problems [112]. I started by handcrafting a set of higher-order options and adding them to each strategy in Vampire's CASC portfolio of strategies. I ran the resulting schedule across all TPTP higher-order problems and then excised strategies from the schedule that did not contribute any unique solutions. I used the resulting schedule as a basis for further engineering. Call it the *base schedule*. Based on my domain knowledge, I subsequently handcrafted numerous strategies incorporating higher-order options. Each strategy was run over the set of TPTP higher-order problems. If the strategy solved any problems not solved by the base schedule, the base schedule was expanded to incorporate the strategy. Periodically, the base schedule was minimized by running it across all

Algorithm 6 Unification algorithm with constraints

```
1: function unifiable_with_abstration(l, r, σ, D)
2:     let P be a set of unification pairs; P := {⟨l, r⟩}
3:     loop
4:         if P is empty then Return true
5:         Select a pair ⟨s, t⟩ in P and remove it from P
6:         if s coincides with t then do nothing
7:         else if s is a variable and s does not occur in dehashed(t) then σ := σ ∘ {s ↦
   t}; P := P{s ↦ t}
8:         else if s is a variable and s occurs in dehashed(t) then Return false
9:         else if t is a variable then P := P ∪ {⟨t, s⟩}
10:        else if s and t are very special variables then D := D ∪ {s ≉ t}
11:        else if s and t have different head symbols then Return false
12:        else if s = f s_1 ... s_n and t = f t_1 ... t_n for some f then
13:            P := P ∪ {⟨s_1, t_1⟩, ..., ⟨s_n, t_n⟩}
```

problems in the library and removing strategies that did not contribute solutions.

In the remainder of this section, I describe the major new options added to Vampire to support higher-order reasoning. Some of these have already been discussed above in which case they are merely listed.

`injectivity`: this is a binary option that can take values `on` and `off`. If set to `on`, Vampire attempts to recognise injective functions and then postulates the existence of left inverses. It is very similar to the injectivity rule of Steen [107, Section 4.2.5].

`pragmatic`: despite the fact that combinatory superposition is clearly an improvement over standard superposition and **SK**-combinators, it still suffers from clause explosion, particularly when faced with a problem containing many variable heads. In such a case, using the option `pragamatic` can curb the explosion at the expense of completeness. It works by limiting the depth at which the most explosive **S**XX-NARROW, **C**XX-NARROW and **B**XX-NARROW inferences operate. This is achieved by attaching a counter to each clause. For clauses in the input this counter is set to 0. If a clause is the conclusion of a **S**XX-NARROW, **C**XX-NARROW or **B**XX-NARROW inference, its counter is set to be one larger than its parent's counter. If the counter of the conclusion is greater than some value (settable via another option), the conclusion is discarded.

`complex_bool_reasoning`: another binary option, if set to `on` primitive instantiation and reasoning about Leibniz equality take place. If set to `off`, these inferences are switched off. The reasoning behind including this option, is that these inferences can be quite explosive and are not required for many problems.

`cnf_on_the_fly`

`prim_inst_set`

`priority_to_long_reducts`: based on a suggestion by Vukmirović, I added an

option to prefer, during clause selection, clauses which are the outcome of many weak reduction steps. This is a binary option and proved helpful for a few problems.

`choice_ax`: a binary option that can be used to add the choice axiom to the search space. If set to `on`, choice reasoning is switched off. Though included in the option list, no strategies actually make use of the axiom. Similarly, options exist for adding combinator axioms and axiomatising the logical constants. Again, neither of these is used in the higher-order schedule due to their poor performance.

`func_ext`: an option used to control extensionality. It can take three values. If set to `axiom`, the functional extensionality axiom is added to the search space. If set to `abstraction`, the *-wA version of the core rules are used. Finally, if set to `off`, no extensionality reasoning takes place other than that provided via NEGEXT which is always on.

`new_taut_del`: Vampire's existing tautology deletion rule recognises clauses containing literals of the form $t \approx t$ to be tautologies. It also recognises a clause of the form $t \approx s \vee t \not\approx s \vee C$ to be a tautology. However, it does not reason about Booleans and therefore a clause containing the literal $\bot \not\approx \top$, or containing two literals of the form $t \approx \bot$ and $t \approx \top$ is not recognised as a tautology. Rather than re-implementing Vampire's highly optimised existing tautology deletion code, I opted to add a new inference which can recognise and delete tautologies such as the above. This new inference is controlled by the binary option `new_taut_del`. Surprisingly, this option is not always helpful.

In order to run Vampire with its higher-order schedule, `--mode casc_hol` can be appended to the command to run Vampire.

Chapter 7

Evaluation

> Nothing exists until it is measured.
>
> *Niels Bohr*

In this chapter, I present a thorough evaluation of Vampire's polymorphic first-order, applicative first-order and higher-order capabilities. Furthermore, I explore the performance of various options and also the impact of some of the design decisions described in the previous chapter. In order to evaluate Vampire, I used the TPTP benchmark library version 7.3.0 [112]. All experiments were carried out on one the following two clusters:

- The StarExec cluster [109] in which each node is equipped with four Intel Xeon E5-2609 CPUs clocked at 2.40 GHz.
- A cluster located at the University of Manchester in which each node is equipped with sixteen Intel(R) Xeon(R) CPU E5-2620 CPUs clocked at 2.10 GHz and 188 gigabytes of RAM.

I attempt to make all experiments as reproducible as possible. As a part of achieving this aim, I have made the problem sets I use available from `https://github.com/ibnyusuf/thesis_problem_sets`. Where a problem sets consists of a well defined and easily accessible subset of the TPTP library, such as the set of all monomorphic higher-order (TH0) problems, it is not added to the above link.

Two versions of Vampire were used in the experiments. For the experiments reported in Section 7.1, Vampire was built with z3 from commit `f7dbad4dbeab0b97d571d11381c6fbf6a9ae1f22`. For experiments linked to higher-order problem sets, Vampire was built without z3 from commit `ee5856ed446b771d19bbc574cf2fbbba30ef580c`. Both commits are from the `polymorphic_vampire` branch. For first-order problems, Vampire was run with `--mode casc` as its only option. For all other problems, Vampire was run with options `--mode portfolio --schedule casc_hol_2020` unless stated otherwise.

Disclaimer: two of the experiments remarked on below are marked with a *. In these experiments, Vampire is compared against higher-order provers that ran in the 2020

Table 7.1: Number of non-arithmetic first-order (TF1, TF0, FOF, CNF) problems proved

	TF1		TF0, FOF, CNF	
	Solved	Unique	Solved	Unique
Leo-III 1.4	224	10	8,665	100
Vampire 4.4	-	-	**11,338**	**381**
Vampire-poly	**239**	**21**	10,948	92
ZenonModulo 0.4.2	80	1	2,926	6
Zipperposition 1.4	171	0	6144	1

CASC system competition. The other provers are optimised to run on multiple cores whilst Vampire is optimised to run on a single core. I chose to run the experiments using a CPU time limit. This somewhat disadvantages the other provers as they use the allotted time running strategies simultaneously resulting in interference between strategies whilst Vampire runs strategies linearly. However, this seemed fairer than the alternatives of running Vampire in multicore mode, or running the other provers in single core mode.

7.1 | Polymorphic First-Order

In order to evaluate Vampire's performance as a polymorphic first-order prover, I tested it on all 539 TF1 [35] (rank-1 polymorphic) problems in the TPTP library. I compared Vampire's results against those of three other provers able to parse TPTP TF1 syntax that I am aware of, Leo-III 1.4 [108] and ZenonModulo 0.4.2 [50] and Zipperposition 1.4 [46]. The experiment was carried out on the StarExec cluster with a CPU time limit of 300s. Vampire solved 15 more problems than Leo-III and 21 problems that none of the other provers could solve (see Table 7.1), although both Leo-III and ZenonModulo solved problems Vampire was unable to solve. Vampire solves 7 previously unsolved rating 1.00 problems.

I also wanted to ascertain how much overhead had been added for non-polymorphic problems, so I tested the polymorphic version of Vampire, Vampire-poly, against the CASC 2019[1] monomorphic version Vampire 4.4, on the set of all 16,922 monomorphic or untyped first order problems in the TPTP library not containing arithmetic. Note that this simply tests whether Vampire goes from solving a problem to not solving it (or vice versa) and not the time taken to find a solution, i.e., I test the impact on proof search and whether any time overhead takes us past the given time limit.

The results (see Table 7.1) are interesting. For monomorphic first-order problems Vampire 4.4 does indeed outperform its polymorphic sibling. The performance of Vampire-poly lags behind that of Vampire by 390 problems. Two potential causes for this disparity have been identified. Firstly, Vampire 4.4 contains a number of improvements and

[1] `http://tptp.cs.miami.edu/CASC/`

new strategies that have not yet been ported into Vampire-poly. Competition performance suggests that standard Vampire improves over time, so the changes made since Vampire-poly diverged are likely to have aided performance. Secondly, Vampire 4.4 makes use of the VCNF clausification procedure discussed earlier [93]. This procedure has not been updated to deal with polymorphic terms and therefore Vampire-poly only uses the classical clausification routine. Clausification has been shown to have a large impact on proof search. Note that Vampire-poly solves 115 problems unsolved by Vampire 4.4.

7.1.1 | Applicative First-Order

As mentioned above, there are two main methods of representing applicative terms in a prover. Either using a flat representation or using the applicative encoding. When using the applicative encoding, the implementation can choose to either treat the app symbol specially, or deal with it as with any other function symbol. Vampire takes the last option. The Ehoh prover of Vukmirović et al. [119] takes the first option. Vukmirović et al. compare the performance of using a flat representation in Ehoh with using the applicative encoding in E [101]. However, no comparison is performed against using the encoding as syntactic sugar. The question therefore remains open to what extent the encoding harms prover performance. In order to answer this question, I have carried out two experiments.

Firstly, I ran Vampire on a set of first-order problems expressed in both first- and higher-order syntax to gauge the overhead caused by the use of the applicative encoding. I selected 99 first-order set theory problems (TPTP SET domain) and 99 first-order logic calculi problems (TPTP LCL domain) at random from the TPTP library. I then converted these to TPTP higher-order syntax (TH0) using the Isabelle theorem prover and ran Vampire on both the first-order and higher-order versions of the problems. The results of the experiment can be found in table 7.2. Surprisingly, on set theory benchmarks, Vampire actually solved more problems using the higher-order syntax than the first-order. This may be due to the fact that Vampire preprocesses problems in higher-order syntax differently to those in first-order syntax. If a problem is written in higher-order syntax, all symbols are treated as function symbols and literals of the form $\mathsf{p}(\overline{s})$ are stored as equality literals of the form $\mathsf{p}(\overline{s}) \approx \top$. For set theory problems, using applicative form does not appear to have had any negative impact on performance in terms of number of problems solved.

For logic calculi problems, Vampire performs very poorly on the higher-order representations, solving barely half the problems solved with the first-order representation. However, the comparison is not very meaningful since, due to differences in preprocessing, completely different strategy schedules were used.

Figure 7.1 displays the time taken by Vampire to find each proof. Problems that were only solved in one version are not included in the graph. In general, the higher-order

Table 7.2: Number of proofs found by Vampire on FOF and TH0 representations of problems

	FOF Syntax	TH0 Syntax
SET	82	84
LCL	81	43

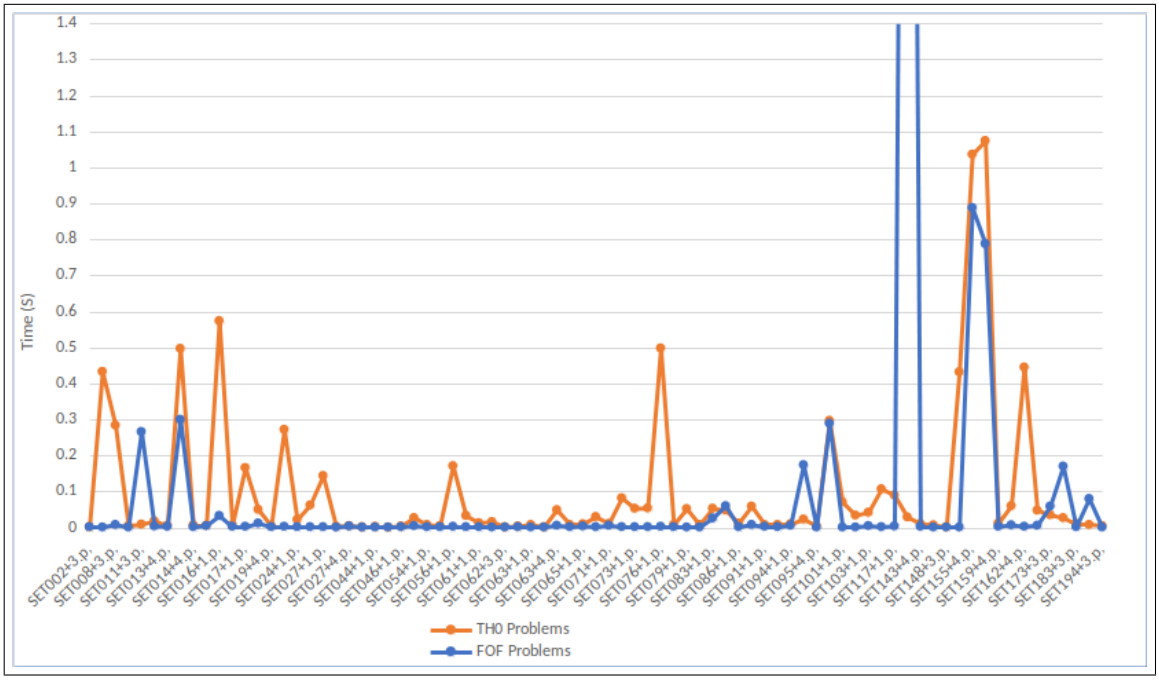

Figure 7.1: Time taken by Vampire to prove SET problems

version of the problems took longer to solve, but this is unlikely to be linked to the usage of applicative terms. If that was the case, we should see a regular time increase for each problem. In some cases, the higher-order format is actually solved far *more* quickly.

In the second experiment, I compared the performance of Vampire against that of Ehoh[2], the only other high performance prover that is sound and complete for applicative first-order logic.[3] For this experiment, Vampire was run with options `--mode casc --lam_free_hol on` which switches on the basic higher-order KBO and ensures that all inferences that are not sound with respect to applicative FOL are deactivated. The only difference to the standard first-order Vampire, was the use of the applicative encoding and basic higher-order KBO. The experiments were carried out over four problem sets. These are:

- The set of of all 955 monomorphic higher-order TPTP problems that are syntactically within the applicative FOL fragment.
- A set consisting of 1253 problems generated by Sledgehammer and translated from

[2]Ehoh used in these experiments was built from commit `https://github.com/eprover/eprover/commit/c1793d5dc7199cc6bad49364fbf5a84435a19dec`

[3]Higher-order provers are complete with respect to the logic, but not sound

higher-order logic to applicative first-order logic using λ-lifting. Each problem contains 32 axioms. I refer to this problem set as SH32L.

- The same as the above set, but the problems were translated using combinators. I refer to this problem set as SH32C.
- The same as the above set, but 512 axioms are included in each problem. I refer to this problem set as SH512C.

Ehoh was run using its autoschedule mode, whilst Vampire ran using its CASC mode. Both provers were provided 60 seconds per problem. The experiments were run on the University of Manchester cluster. The results of the experiment can be found in Table 7.3. In the table, "U" stands for "Uniques". On the TPTP problems, Vampire slightly outperformed Ehoh, whilst on Sledgehammer benchmarks, Ehoh outperformed Vampire. This may be linked to Vampire's use of the applicative encoding, but the difference in proof numbers is small enough to suggest that even if this is the case, the encoding is not prohibitively limiting. In Table 7.4, I display the average running times for Vampire and Ehoh over the problem sets. Vampire and Ehoh are different provers with completely different underlying data structures and algorithms. Therefore, a direct comparison between the running times cannot be made. However, Vampire's shorter solving times again suggest that the applicative encoding is not prohibitively expensive.

Table 7.3: Number of applicative first-order problems proved.

	TPTP		SH32L		SH32C		SH512C	
	Solved	U	Solved	U	Solved	U	Solved	U
Vampire	**693**	**17**	474	4	485	3	637	58
Ehoh	690	14	**489**	**19**	**492**	**10**	**649**	**71**

Table 7.4: Average time (in seconds) taken to solve applicative first-order problems.

	TPTP	SH32L	SH32C	SH512C
Vampire	0.34	0.13	0.14	0.48
Ehoh	0.38	1.01	1.34	4.24

7.1.2 | Clausal Higher-Order*

Vampire targets clausal higher-order logic, so it makes sense to test the prover's capabilities on problems from this domain. In order to do this, I selected two problem sets.

- The set of all 592 monomorphic higher-order (TH0) problems from the TPTP library that do not contain interpreted Booleans or arithmetic. Note that this is a *syntactic* characterisation of the problems. It can be the case that a problem from

this set is satisfiable with respect to clausal semantics, but a theorem with respect to full Henkin semantics. I refer to this problem set as TH0BF.

- A set of 1253 higher-order problems generated by Sledgehammer to target the clausal fragment. Each problem contains 256 axioms. I refer to this problem set as Sh-λ.

I compared the performance of Vampire against state-of-the-art higher-order provers over these sets. The major higher-order provers other than Vampire are currently Zipperposition 2.0 [46], Satallax 3.5 [42], Leo-III 1.5 [108] and the higher-order SMT solver CVC4 1.8 [13]. Zipperposition is based on superposition, Leo-III on incomplete paramodulation and Satallax on a higher-order tableaux calculus. It should be noted that these provers are all more mature than Vampire's higher-order modifications. Besides for CVC4, the remaining provers are all *cooperative* provers in that they make periodic calls to first-order solver(s). For the experiments, I used the CASC 2020 version of each prover. Only Satallax 3.5 is complete for full higher-order logic.

Zipperposition is complete for the clausal fragment, whilst Leo-III and CVC4 have no completeness guarantees. Thus, I only record the number of theorems proved by each system, and discards proofs of non-theoremhood since, for saturation based provers, such proofs require completeness. The experiments were carried out on the StarExec cluster with a (CPU) time limit of 300 seconds. The results can be seen in Table 7.5. Though Vampire is not the leading prover in either set of benchmarks, it is very competitive in both.

Table 7.5: Number of clausal higher-order problems proved.

	TH0BF		Sh-λ	
	Solved	Unique	Solved	Unique
Leo-III 1.5	**480**	**1**	660	6
Satallax 3.5	465	0	556	5
Vampire	477	0	710	5
CVC4 1.8	454	3	674	9
Zipperposition 2.0	478	1	**714**	**13**

7.1.3 | Full Higher-Order*

The higher-order section of the TPTP library consists of monomorphic (TH0) problems and rank-1 polymorphic (TH1) problems. There are 3845 high-order problems altogether in the library. Of these, 3187 are TH0 problems and the remaining 667 are TH1 problems. Of the TH0 problems, 2607 are theorems and the remaining are satisfiable. Of the TH1 problems, 662 are theorems and 5 are satisfiable. Of the higher-order provers, Leo-III and Zipperposition are capable of parsing and reasoning about TH1. I tested

Vampire against the other higher-order solvers across both these sets with a (CPU) time limit of 300 seconds on the StarExec cluster. The results can be seen in Table 7.6. In the TH0 division, Vampire performs well, solving one less problem than the multiple CASC higher-order division winner Satallax, and outperforming the far more developed prover Leo-III. However, it is still some way behind the CASC 2020 winner Zipperposition. In the TH1 division, Vampire is currently the best performing prover.

Table 7.6: Number of higher-order monomorphic and polymorphic problems proved.

	TH0		TH1	
	Solved	Unique	Solved	Unique
Leo-III 1.5	2089	3	199	12
Satallax 3.5	2110	40	-	-
Vampire	2109	12	**224**	**20**
CVC4 1.8	1811	13	-	-
Zipperposition 2.0	**2202**	**47**	212	7

7.1.4 | Evaluating Options

So far, I have discussed the performance of Vampire based on its higher-order schedule. It is interesting to consider the impact of individual options as well. It is a tricky task to decide how to evaluate a single option. I do this by fixing a particular base higher-order strategy. Then, to test a single option, I vary that option on top of the base strategy and compare the results. The base strategy given in Vampire's coded format is:

```
dis+1011_10_add=large:csup=on:inj=on:chr=on:foolp=on:
cnfonf=eager:afr=on:afp=4000:afq=1.0:amm=off:anc=none:
lma=on:nm=64:nwc=4:sac=on:sp=occurrence_75
```

The base strategy runs with the Discount loop, with FOOLPARAMODULATION enabled (rather than CASESSIMP), with injectivity and choice reasoning enabled and with Avatar on, for a period of 7.5 seconds. All evaluations were carried out across the entire higher-order section of the TPTP library. Only the numbers of theorems proved is recorded, since Vampire is incomplete when dealing with full higher-order logic. The results from these experiments can be found in Table 7.7. The number of solutions unique to one particular setting of an option is provided in brackets. The base strategy is repeated for each option. For example the default setting of the option `pragmatic` is `off`. Thus the row in which `pragmatic` is set to `off` corresponds to the base strategy. Due to a certain amount of randomness built into Vampire, the number of problems solved by the base strategy is not the same on all runs.

The results show that eager clausification is generally better than its lazy counterpart. However, true lazy clausification, where clausifying rules are generating, is useful in some cases. Interestingly, not carrying out any NARROW inferences is effective in a large number of cases. Furthermore, narrowing with the full set generally doesn't help. The results suggest that it would be useful to experiment with other combinator sets, including incomplete sets. As highlighted by others, reasoning about extensionality using the extensionality axiom, generally does not perform very well. Surprisingly, deleting extra tautologies can result in proofs being lost.

Table 7.7: Impact of various higher-order parameters

Option	Values	Solved TH1	Solved TH0
cnf_on_the_fly	eager base	128 (5)	1795 (508)
	lazy_simp	127(0)	1312 (0)
	lazy_gen	104 (2)	1036 (4)
	lazy_not_gen	127 (0)	1312 (0)
	lazy_not_gen_be_off	89 (0)	1105 (2)
	lazy_not_be_gen	106 (0)	1221 (1)
	off	82 (0)	175 (1)
narr	all base	128 (0)	1795 (7)
	ski	131 (2)	1819 (3)
	sk	132 (1)	1819 (2)
	off	129 (3)	1791 (25)
pragmatic	on	130 (4)	1803 (15)
	off base	128 (2)	1796 (8)
func_ext	off	141 (5)	1785 (34)
	axiom	127 (2)	1639 (13)
	abstraction base	128 (3)	1796 (42)
piset	all	129 (2)	1801 (2)
	all_but_not_eq base	128 (1)	1798 (2)
	false_true_not	128 (4)	1779 (9)
	small_set	130 (0)	1799 (0)
new_taut_del	on	132 (7)	1815 (36)
	off base	128 (3)	1796 (17)

Arguably, the above method of evaluating options is not ideal since an option may only work well in conjunction with certain settings of other options. Furthermore, a configuration may not lead to many proofs, but it may provide a number of unique proofs making it more valuable in a strategy schedule than other configurations. Therefore, it may be more helpful to evaluate an option by counting the number of problems that require the option to be set to a particular value for a proof to be found. This data is surprisingly difficult to obtain, since the complete option space cannot be searched.

Chapter 8

Conclusion, Philosophical Implications & Further Work

In this thesis, I have presented three contributions to the field of automated reasoning. Firstly, I have have extended the Knuth-Bendix ordering to a weak-compatible ordering that works on applicative terms. The resulting order can compare terms that are incomparable by an ordering that works on weak or β-equivalence classes. This is important, because superposition uses an ordering to restrict inferences. The more terms that are comparable, the fewer inferences that need to be carried out. With higher-order logic, the search space is enormous and any methods to restrict the space without harming completeness are likely to be beneficial.

Secondly, I have used the order to parameterise the superposition calculus. The resulting clausal combinatory superposition calculi are complete for the clausal fragment of higher-order logic. They are based on combinatory terms rather than λ-terms and thus are very similar to standard first-order superposition. The calculi are thus good candidates for extending first-order ATPs to higher-order logic. Supporting higher-order logic brings the language of first-order ATPs closer to that of many ITPs. This in turn removes the need for translating an ITP's logic in order to call automated tools. As such, my work should hopefully improve the success rate of hammers and related tools. I see my work as a small step towards the goals set out in the QED manifesto [116].

Finally, I have implemented the selecting calculus in the Vampire theorem prover and detailed the steps involved. The implementation is immature and there are a large number of obvious and not so obvious areas for improvement. Nevertheless, Vampire higher-order has already displayed promise. Indeed, on polymorphic first-order and higher-order problem sets, Vampire is currently the leading prover (Chapter 7, Tables 7.1 and 7.6). I also hypothesise that Vampire is currently the strongest non-cooperative higher-order prover.

In the introduction, I mentioned the debate about the status of higher-order logic as a

separate logic from first-order. I mentioned that whilst philosophers of mathematics are inclined to class them together, proof-theoreticians have tended to consider them as two separate logics due to the very different nature of their proof calculi. I have demonstrated that, with minor modifications, the leading calculus used for theorem proving in first-order logic, superposition, forms an efficient reasoning method for higher-order logic. I do not claim that my work establishes the non-existence of higher-order logic as a distinct entity, but I do claim that it indicates that the gap between first- and higher-order logic is not as wide as presumed. In this claim, I follow Dowek [53] who pointed out that higher-order logic can fruitfully be treated as a first-order theory. He points out that doing so simplifies some obscure aspects of simple type theory such as its complex Skolem theorem [87].

8.1 | Related Work

In the introduction, I provided an overview of the history of automated reasoning and I will not cover that material here. Reasoning about combinators goes against the current trends in higher-order reasoning, and I am not aware of any other proof calculi for HOL that have been designed around combinatory logic. The deduction modulo framework of Dowek et al. [55] was designed to facilitate efficient reasoning about theories in first-order logic. It can be used to treat combinatory logic as a first-order theory. However, it requires a unification algorithm that works modulo the theory under consideration, in this case combinatory logic. In the absence of an efficient unifcation procedure modulo the combinator axioms, the method is not competitive. The only prover that I am aware of to be based on these ideas received little attention and even less development [44]. Perhaps a more promising approach is to use deduction modulo to reason about a version of higer-order logic based on the calculus of explicit substitutions [1] as described by Dowek et al. [54]. The issues caused by orders not orienting the combinator axioms as desired has long cropped up in the context of translations by hammers from HOL to FOL. Blanchette et al. note the issue with regards to Sledgehammer [37, Section 4]. They suggest a pragmatic solution, force the desired orientation against the term ordering. Clearly, this affects completeness.

The Knuth-Bendix and recursive path orderings have been extended and modified for various purposes. For example, Korovin and Voronkov modified the Knuth-Bendix order to create an AC-compatible ordering [80].[1] Many researchers have explored extending RPO to be AC-compatible [99, 100]. In as yet unpublished work, Bentkamp has developed the embedding path order, a generalisation of RPO to applicative first-order terms [18]. As far as I am aware, there have been no previous attempts to derive a weak-compatible ordering.

There have been a few attempts to modify first-order provers to deal with HOL. Bee-

[1] The word “compatible” is not used in the same sense as I have used the term.

son modified the Otter prover to support a fragment of HOL called λ-logic [16]. In work that I have reported on extensively elsewhere, Zipperposition has been extended to support HOL. Finally, the E prover has been updated to support applicative first-order logic [119]. Vampire is the first high performance first-order prover to be extended to support rank-1 polymorphism and HOL.

8.2 | Future Directions

The work presented in this thesis opens a number of lines of investigation, and I present only the most promising and interesting here. It should be possible to extend the weak-compatible ordering that I have developed into a β-compatible ordering. Indeed, I sketch such an extension in [32]. The sketch can be improved and completed with full proofs.

It remains future work to create a refined implementation of the combinatory superposition calculi. At present, only the simple $>_{\mathsf{ski1}}$ ordering has been implemented. This removes one of the main advantages of combinatory superposition over other methods, namely its more restricted nature due to being able to compare more non-ground terms. It remains to be investigated whether the $>_{\mathsf{ski}}$ ordering can be implemented efficiently and to implement it if it can.

Vampire implements the CASESSIMP and FOOLPARAMODULATION rules naively by replacing all Boolean subterms of a term simultaneously. Vukmirović and Nummelin [120] point out the issues with this method. They propose following a leftmost outermost scheme for replacing Boolean subterms and show that in general, this is beneficial. The Zipperposition prover also implements a simplification rule, PRUNEARG, that can be used to remove some of the arguments of applied variables under certain conditions. As terms with variable heads are candidates for narrowing and therefore disastrous for proof search, I hypothesise that such a rule would be beneficial in Vampire. It should be a relatively straightforward task to implement the no-select calculus. Lastly, it would be of interest to develop a version of the calculi that can utilise selection, but still do without SUBVARSUP.

More broadly, it remains to investigate whether a β-compatible ordering could be used to parameterise superposition and create a hybrid calculus that uses λ-terms, but first-order unification. The indications are that such a calculus is feasible and that it may avoid some of the downsides of combinatory superposition whilst maintaining its attractive qualities. The NARROW rule of combinatory superposition can introduce polymorphism into a monomorphic problem. A λ-version of the calculus should be able to avoid this, just as higher-order unification of monomorphic λ-terms does not require polymorphism whilst its combinatory equivalent does [52]. The Matryoshka team are currently designing an extension to the clausal λ-superposition calculus that is complete for full higher-order logic. Presumably it will be possible to adapt many of their ideas to combinatory superposition and its proposed λ-variant resulting in calculi that are complete for

full HOL.

Support for higher-order logic brings ATPs closer to ITPs based on HOL. However, other features of ITPs such as induction still require support. Furthermore, a gap remains between ATPs and ITPs based on dependent type theory. Czajka and Kaliszyk have developed a hammer for the dependent type theory based ITP Coq [47], but due to the complex nature of the logic, the hammer utilises unsound translations. Extending Vampire and other ATPs to support dependent type theory would improve the situation. Many ATPs do possess some support for induction [46, 65, 125]. However, this remains an ongoing and active area of research, particularly the combining of induction and machine learning.

Finally, for the computer science and mathematics communities to be able to benefit from the improved calculi and provers developed as part of my research and elsewhere, better integration with ITPs is required. Steps in this direction have been undertaken. HOL(y)Hammer is an online service that links ATPs to the HOL Light and HOL4 ITPs [76]. By being cloud based, it avoids some of the hardware limitations that affect other hammers. However, hammers often still call ATPs with sub-optimal strategies. Further, hammers discard ATP proofs and attempt to reconstruct them using trusted methods based on the axioms used by the ATP. This is problematic for higher-order ATPs, since currently the trusted verified provers such as Metis and Meson are first-order. A better approach would be for the ATP community to decide on a common proof language for ATPs that ITPs could then replay. The TESC language is a step in this direction [12].

Bibliography

[1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. “Explicit substitutions”. In: *Journal of functional programming* 1.4 (1991), pp. 375–416 (page 143).

[2] Peter B Andrews. “Theorem proving via general matings”. In: *Journal of the ACM (JACM)* 28.2 (1981), pp. 193–214 (page 113).

[3] Peter B Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. “TPS: A theorem-proving system for classical type theory”. In: *Journal of automated reasoning* 16.3 (1996), pp. 321–353 (page 15).

[4] Peter B. Andrews. “Resolution in type theory”. In: *Journal of Symbolic Logic* 36.3 (1971), pp. 414–432 (page 15).

[5] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. “Uniform derivation of decision procedures by superposition”. In: *CSL*. Vol. 2142. LNCS. Springer, 2001, pp. 513–527 (page 20).

[6] Noran Azmy and Christoph Weidenbach. “Computing tiny clause normal forms”. In: *CADE-24*. Vol. 7898. LNCS. Springer, 2013, pp. 109–125 (pages 18, 25).

[7] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999 (pages 22, 34, 45).

[8] Leo Bachmair and Harald Ganzinger. “On restrictions of ordered paramodulation with simplification”. In: *IJCAR*. Vol. 449. LNCS. Springer, 1990, pp. 427–441 (page 14).

[9] Leo Bachmair and Harald Ganzinger. “Rewrite-based equational theorem proving with selection and simplification”. In: *Journal of Logic and Computation* 4.3 (1994), pp. 217–247 (pages 40, 73).

[10] Leo Bachmair and Harald Ganzinger. “Resolution Theorem Proving”. In: *Handbook of Automated Reasoning*. Ed. by A. Robinson and A. Voronkov. Vol. I. Elsevier Science, 2001. Chap. 2 (page 22).

[11] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. “Superposition with simplification as a decision procedure for the monadic class with equality”. In: *KGC*. Vol. 713. LNCS. Springer, 1993, pp. 83–96 (page 20).

[12] Seulkee Baek. “The TESC proof format for first-order ATPs”. In: *PAAR*. 2020 (page 145).

[13] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. “Extending SMT solvers to higher-order logic”. In: *CADE-27*. Vol. 11716. LNCS. Springer, 2019, pp. 35–54 (pages 15, 139).

[14] Henk P. Barendregt. *The lambda calculus: Its syntax and semantics*. 2nd. Elsevier Science Publishers, 1984 (page 48).

[15] Heiko Becker, Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. “A transfinite Knuth–Bendix order for lambda-free higher-order terms”. In: *CADE*. Vol. 10395. LNCS. Springer, 2017, pp. 432–453 (pages 36, 52, 53, 65).

[16] Michael Beeson. “Lambda logic”. In: *IJCAR*. Vol. 3097. LNCS. Springer, 2004, pp. 460–474 (page 144).

[17] Michael Beeson. “The mechanization of mathematics”. In: *Alan Turing: Life and legacy of a great thinker*. Springer, 2004, pp. 77–134 (page 13).

[18] Alexander Bentkamp. “The embedding path order for lambda-free higher-order terms”. Unpublished paper. `https://matryoshka-project.github.io/pubs/epo_paper.pdf` (page 143).

[19] Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. “Superposition for lambda-free higher-order logic”. In: *IJCAR*. Vol. 1 0900. LNCS. Springer, 2018, pp. 28–46 (pages 28, 36, 44–46, 52, 56, 73, 76, 78, 92).

[20] Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. “Superposition for lambda-free higher-order logic”. Unpublished paper. `http://matryoshka.gforge.inria.fr/pubs/lfhosup_article.pdf` (page 46).

[21] Alexander Bentkamp, Jasmin Christian Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. “Superposition with lambdas”. In: *CADE*. Vol. 11 716. LNCS. Springer, 2019, pp. 55–73 (pages 15, 29, 53, 65, 69, 72).

[22] Alexander Bentkamp, Jasmin Christian Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. *Superposition with lambdas (technical report)*. Technical report. `http://matryoshka.gforge.inria.fr/pubs/lamsup_report.pdf`. 2019 (page 85).

[23] Christoph Benzmüller. “Comparing approaches to resolution based higher-order theorem proving”. In: *Synthese* 133 (2002), pp. 203–335 (pages 103, 107).

[24] Christoph Benzmüller, Chad E Brown, Michael Kohlhase, et al. “Higher-order semantics and extensionality”. In: *Journal of Symbolic Logic* 69 (2004), pp. 1027–1088 (page 98).

[25] Christoph Benzmüller and Michael Kohlhase. “System description: LEO–a higher-order theorem prover”. In: *IJCAR*. Springer, 1998, pp. 139–143 (page 15).

[26] Christoph Benzmüller and Dale Miller. “Automation of higher-order logic”. In: *Handbook of the History of Logic, Volume 9 — Logic and Computation*. Elsevier, 2014 (pages 13, 16).

[27] Christoph Benzmüller, Nik Sultana, Lawrence C Paulson, and Frank Theiß. “The higher-order prover LEO-II”. In: *Journal of Automated Reasoning* 55.4 (2015), pp. 389–404 (page 15).

[28] Ahmed Bhayat and Giles Reger. “Set of support for higher-order reasoning.” In: *PAAR*. 2018 (page 21).

[29] Ahmed Bhayat and Giles Reger. “Restricted combinatory unification”. In: *CADE*. Vol. 11716. LNCS. Springer, 2019, pp. 74–93 (pages 21, 129, 130).

[30] Ahmed Bhayat and Giles Reger. “A combinator-based superposition calculus for higher-order logic”. In: *IJCAR*. Vol. 12166. LNCS. Springer, 2020, pp. 278–296 (pages 21, 56).

[31] Ahmed Bhayat and Giles Reger. “A Knuth-Bendix-like ordering for orienting combinator equations”. In: *IJCAR*. Vol. 12166. LNCS. Springer, 2020, pp. 259–277 (pages 21, 67).

[32] Ahmed Bhayat and Giles Reger. *A Knuth-Bendix-like ordering for orienting combinator Equations (Technical Report)*. Technical report. `https://easychair.org/publications/preprint_open/rXSk`. 2020 (page 144).

[33] Ahmed Bhayat and Giles Reger. “A polymorphic Vampire”. In: *IJCAR*. Vol. 12167. LNCS. Springer, 2020, pp. 361–368 (pages 21, 121).

[34] Wolfgang Bibel. “Early history and perspectives of automated deduction”. In: *KI 2007: Advances in Artificial Intelligence*. Springer, 2007, pp. 2–18 (page 13).

[35] Jasmin Christian Blanchette and Andrei Paskevich. “TFF1: The TPTP typed first-order form with rank-1 polymorphism”. In: *CADE*. Vol. 7898. LNCS. Springer, 2013, pp. 414–420 (pages 121, 135).

[36] Jasmin Christian Blanchette, Nicolas Peltier, and Simon Robillard. “Superposition with datatypes and codatatypes”. In: *IJCAR*. Vol. 10900. LNCS. Springer, 2018, pp. 370–387 (page 20).

[37] Jasmin Christian Blanchette, Andrei Popescu, Daniel Wand, and Weidenbach Christoph. “More SPASS with Isabelle”. In: *ITP*. Vol. 7406. LNCS. Springer, 2012, pp. 345–360 (page 143).

[38] Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. “A lambda-free higher-order recursive path order”. In: *FoSSaCS*. Vol. 10203. LNCS. Springer, 2017, pp. 461–479 (pages 44, 52).

[39] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. “The computability path ordering: The end of a quest”. In: *CSL*. Vol. 5213. LNCS. Springer, 2008, pp. 1–14 (page 52).

[40] Miquel Bofill and Albert Rubio. “Paramodulation with non-monotonic orderings and simplification”. In: *Journal of automated reasoning* 50.1 (2013), pp. 51–98 (pages 52, 56).

[41] Joshua Brakensiek, Marijn Heule, John Mackey, and David Narváez. “The resolution of Keller’s conjecture”. In: *IJCAR*. Vol. 1-2166. LNCS. Springer, 2020, pp. 48–65 (page 14).

[42] Chad E Brown. “Satallax: An automatic higher-order prover”. In: *IJCAR*. Vol. 7364. LNCS. Springer, 2012, pp. 111–117 (page 139).

[43] Chad E Brown. “Reducing higher-order theorem proving to a sequence of SAT problems”. In: *Journal of Automated Reasoning* 51.1 (2013), pp. 57–77 (page 15).

[44] Guillaume Burel. “Embedding deduction modulo into a prover”. In: *International Workshop on Computer Science Logic*. Vol. 6247. LNCS. Springer, 2010, pp. 155–169 (page 143).

[45] Alonzo Church. “A formulation of the simple theory of types”. In: *Journal of symbolic logic* 5.2 (1940), pp. 56–68 (pages 13, 15).

[46] Simon Cruanes. “Superposition with structural induction”. In: *FroCoS*. Vol. 10483. LNCS. Springer, 2017, pp. 172–188 (pages 135, 139, 145).

[47] Łukasz Czajka and Cezary Kaliszyk. “Hammer for Coq: Automation for dependent type theory”. In: *Journal of automated reasoning* 61.1-4 (2018), pp. 423–453 (page 145).

[48] Martin Davis. “The prehistory and early history of automated deduction”. In: *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966* (1983) (page 13).

[49] Martin Davis. “The early history of automated deduction”. In: *Handbook of Automated Reasoning*. Ed. by A. Robinson and A. Voronkov. Vol. I. Elsevier Science, 2001. Chap. 1 (page 13).

[50] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. “Zenon Modulo: When achilles outruns the tortoise using deduction modulo”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Vol. 8312. LNCS. Springer, 2013, pp. 274–290 (page 135).

[51] Nachum Dershowitz. “Termination of rewriting”. In: *Journal of symbolic computation* 3 (1987), pp. 69–115 (page 52).

[52] Daniel J Dougherty. “Higher-order unification via combinators”. In: *Theoretical Computer Science* 114.2 (1993), pp. 273–298 (page 144).

[53] Gilles Dowek. “Skolemization in simple type theory: the logical and the theoretical points of view”. In: *Festschrift in Honour of Peter B. Andrews on his 70th Birthday, Studies in Logic and the Foundations of Mathematics. College Publications* (2008) (pages 34, 143).

[54] Gilles Dowek, Thérese Hardin, and Claude Kirchner. “HOL-$\lambda\sigma$: An intentional first-order expression of higher-order logic”. In: *RTA*. Vol. 1631. LNCS. Springer, 1999, pp. 317–331 (page 143).

[55] Gilles Dowek, Thérese Hardin, and Claude Kirchner. “Theorem proving modulo”. In: *Journal of Automated Reasoning* 31 (2003), pp. 33–72 (pages 104, 127, 143).

[56] Herbert Enderton. “Second-order and higher-order logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2019. Metaphysics Research Lab, Stanford University, 2019 (page 17).

[57] Arnaud Fietzke and Christoph Weidenbach. “Superposition as a decision procedure for timed automata”. In: *Mathematics in Computer Science* 6.4 (2012), pp. 409–425 (page 20).

[58] Gottlob Frege. “Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought”. In: (1879) (page 14).

[59] Harald Ganzinger and Hans De Nivelle. “A superposition decision procedure for the guarded fragment with equality”. In: *LICS*. IEEE, 1999, pp. 295–303 (page 20).

[60] Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. “Context trees”. In: *IJCAR*. Vol. 2083. LNCS. Springer, 2001, pp. 242–256 (page 116).

[61] Harald Ganzinger and Jürgen Stuber. “Superposition with equivalence reasoning and delayed clause normal form transformation”. In: *IJCAR*. Vol. 2741. LNCS. Springer, 2003, pp. 335–349 (page 37).

[62] Thibault Gauthier. “Deep reinforcement learning for synthesizing functions in higher-order logic.” In: *LPAR*. 2020, pp. 230–248 (page 16).

[63] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für mathematik und physik* 38.1 (1931), pp. 173–198 (page 17).

[64] Peter Graf. “Substitution tree indexing”. In: *Rewriting Techniques and Applications*. Vol. 914. LNCS. Springer, 1995, pp. 117–131 (pages 19, 116, 117).

[65] Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. “Induction with generalization in superposition reasoning”. In: *CICM*. Vol. 12236. LNCS. Springer, 2020, pp. 123–137 (page 145).

[66] Jacques Herbrand. “Recherches sur la théorie de la démonstration”. PhD thesis. J. Dziewulski, 1930 (page 13).

[67] Marijn Heule, Manuel Kauers, and Martina Seidl. “Local search for fast matrix multiplication”. In: *SAT*. Vol. 11628. LNCS. Springer, 2019, pp. 155–163 (page 12).

[68] Thomas Hillenbrand and Christoph Weidenbach. “Superposition for bounded domains”. In: *Automated Reasoning and Mathematics*. Springer, 2013, pp. 68–100 (page 101).

[69] J Roger Hindley and Jonathan P Seldin. *Lambda-calculus and combinators, an introduction*. Cambridge University Press, 2008 (pages 30, 47).

[70] Kryštof Hoder and Andrei Voronkov. “Comparing unification algorithms in first-order theorem proving”. In: *Annual Conference on Artificial Intelligence*. Vol. 5803. LNCS. Springer, 2009, pp. 435–443 (pages 19, 116, 118).

[71] Gerard P Hubt. “A mechanization of type theory”. In: *IJCAI*. Morgan Kaufmann Publishers Inc., 1973, pp. 139–146 (page 15).

[72] Gerard P. Huet. “A unification algorithm for typed λ-calculus”. In: *Theoretical Computer Science* 1.1 (1975), pp. 27–57 (page 15).

[73] Don C Jensen and Tomasz Pietrzykowski. “Mechanizing ω-order type theory through unification”. In: *Theoretical Computer Science* 3.2 (1976), pp. 123–171 (page 15).

[74] Jean-Pierre Jouannaud and Albert Rubio. “Polymorphic higher-order recursive path orderings”. In: *Journal of the ACM (JACM)* 54 (2007), pp. 1–48 (page 52).

[75] Cezary Kaliszyk, Geoff Sutcliffe, and Florian Rabe. “TH1: The TPTP typed higher-order form with rank-1 polymorphism.” In: *PAAR*. 2016, pp. 41–55 (page 123).

[76] Cezary Kaliszyk and Josef Urban. “HOL(y)Hammer: Online ATP service for HOL Light”. In: *Mathematics in Computer Science* 9.1 (2015), pp. 5–22 (pages 12, 145).

[77] Manfred Kerber. “Sound and complete translations from sorted higher-order logic into sorted first-order logic”. In: *Proceedings of PRICAI-94, Third Pacific Rim International Conference on Artificial Intelligence*. 1994, pp. 149–154 (page 34).

[78] Donald E Knuth and Peter B Bendix. “Simple word problems in universal algebras”. In: *Automation of Reasoning*. Springer, 1983, pp. 342–376 (page 52).

[79] Cynthia Kop and Femke van Raamsdonk. “A higher-order iterative path ordering”. In: *LPAR*. Vol. 5330. LNCS. Springer, 2008, pp. 697–711 (page 52).

[80] Konstantin Korovin and Andrei Voronkov. “An AC-compatible Knuth-Bendix order”. In: *CADE*. Vol. 2741. LNCS. Springer, 2003, pp. 47–59 (page 143).

[81] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. “The Vampire and the FOOL”. In: *CPP*. Association for Computing Machinery, 2016, pp. 37–48 (pages 19, 101).

[82] Evgenii Kotelnikov, Laura Kovács, Martin Suda, and Andrei Voronkov. “A clausal normal form translation for FOOL.” In: *GCAI*. 2016, pp. 53–71 (pages 19, 25, 102).

[83] Laura Kovács and Andrei Voronkov. “First-order theorem proving and Vampire”. In: *CAV*. Vol. 8044. LNCS. Springer, 2013, pp. 1–35 (pages 12, 18, 22, 112).

[84] Donald MacKenzie. “The automation of proof: A historical and sociological exploration”. In: *IEEE Ann. Hist. Comput.* 17.3 (1995), pp. 7–29 (page 13).

[85] William McCune. “Solution of the Robbins problem”. In: *Journal of Automated Reasoning* 19.3 (1997), pp. 263–276 (page 14).

[86] Jia Meng and Lawrence C Paulson. “Translating higher-order clauses to first-order clauses”. In: *Journal of Automated Reasoning* 40.1 (2008), pp. 35–60 (pages 12, 47, 123).

[87] Dale A Miller. “A compact representation of proofs”. In: *Studia Logica* 46.4 (1987), pp. 347–370 (pages 29, 143).

[88] R. Nieuwenhuis and A. Rubio. “Paramodulation-based theorem proving”. In: *Handbook of Automated Reasoning*. Ed. by A. Robinson and A. Voronkov. Vol. I. Elsevier Science, 2001. Chap. 7, pp. 371–443 (pages 19, 35, 56, 68).

[89] Dag Prawitz, Haå kan Prawitz, and Neri Voghera. “A mechanical proof procedure and its realization in an electronic computer”. In: *Journal of the ACM (JACM)* 7.2 (1960), pp. 102–128 (page 13).

[90] Willard V Quine. *Philosophy of logic*. Harvard University Press, 1986 (page 17).

[91] Michael Rawson, Ahmed Bhayat, and Giles Reger. "Reinforced external guidance for theorem provers". In: *PAAR*. 2020 (page 21).

[92] Giles Reger, Martin Suda, and Andrei Voronkov. "The challenges of evaluating a new feature in Vampire." In: *Vampire Workshop*. 2014, pp. 70–74 (page 120).

[93] Giles Reger, Martin Suda, and Andrei Voronkov. "New techniques in clausal form generation." In: *GCAI* 41 (2016), pp. 11–23 (pages 18, 102, 112, 128, 136).

[94] Giles Reger, Martin Suda, and Andrei Voronkov. "Unification with abstraction and theory instantiation in saturation-based reasoning". In: *TACAS*. Vol. 10805. LNCS. Springer, 2018, pp. 3–22 (page 71).

[95] Alexandre Riazanov and Andrei Voronkov. "Partially adaptive code trees". In: *European Workshop on Logics in Artificial Intelligence*. Vol. 1919. LNCS. Springer, 2000, pp. 209–223 (page 116).

[96] Alexandre Riazanov and Andrei Voronkov. "Splitting without backtracking". In: *IJCAI*. 2001, pp. 611–617 (page 19).

[97] George Robinson and Lawrence Wos. "Paramodulation and theorem proving in first order theories with equality". In: *Machine intelligence* 4 (1969), pp. 133–150 (page 14).

[98] John Alan Robinson. "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41 (page 13).

[99] Albert Rubio. "A fully syntactic AC-RPO". In: *RTA*. Vol. 1631. LNCS. Springer, 1999, pp. 133–147 (page 143).

[100] Albert Rubio and Robert Nieuwenhuis. "A precedence-based total AC-compatible ordering". In: *RTA*. Vol. 690. LNCS. Springer, 1993, pp. 374–388 (page 143).

[101] Stephan Schulz. "E–a brainiac theorem prover". In: *AI Communications* 15.2, 3 (2002), pp. 111–126 (pages 18, 22, 136).

[102] Stephan Schulz. "Fingerprint indexing for paramodulation and rewriting". In: *IJCAR*. Vol. 7364. LNCS. Springer, 2012, pp. 477–483 (page 116).

[103] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. "Faster, higher, stronger: E 2.3". In: *CADE-27*. Vol. 11716. LNCS. Springer, 2019, pp. 495–507 (page 15).

[104] R Sekar, IV Ramakrishnan, and Andrei Voronkov. "Term indexing". In: *Handbook of automated reasoning*. Ed. by A Robinson and A Voronkov. Elsevier Science, 2001, pp. 1853–1964 (pages 115, 116, 118).

[105] Stewart Shapiro. "Second-order logic, foundations, and rules". In: *The Journal of philosophy* 87.5 (1990), pp. 234–261 (pages 16, 17).

[106] Stewart Shapiro. *Foundations without foundationalism: A case for second-order logic*. Vol. 17. Clarendon Press, 1991 (pages 17, 22).

[107] Alexander Steen. "Extensional paramodulation for higher-order logic and its effective implementation Leo-III". PhD thesis. 2018 (pages 101, 105, 106, 132).

[108] Alexander Steen and Christoph Benzmüller. "The higher-order prover Leo-III". In: *IJCAR*. Vol. 10900. LNCS. Springer, 2018, pp. 108–116 (pages 15, 135, 139).

[109] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. "StarExec: A cross-community infrastructure for logic solving". In: *IJCAR*. Vol. 8562. LNCS. Springer, 2014, pp. 367–373 (page 134).

[110] Martin Suda and Bernhard Gleiss. "Layered clause selection for theory reasoning". In: *IJCAR*. Vol. 12166. LNCS. Springer, 2020, pp. 402–409 (page 120).

[111] Nik Sultana, Jasmin Christian Blanchette, and Lawrence C Paulson. "LEO-II and Satallax on the Sledgehammer test bench". In: *Journal of Applied Logic* 11.1 (2013), pp. 91–102 (page 12).

[112] Geoff Sutcliffe. "The TPTP problem library and associated infrastructure". In: *Journal of Automated Reasoning* 43 (2009), p. 337 (pages 98, 121, 131, 134).

[113] Geoff Sutcliffe and Christian Suttner. "The state of CASC". In: *AI Communications* 19.1 (2006), pp. 35–48 (pages 13, 15).

[114] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 18 (page 13).

[115] David A Turner. "A new implementation technique for applicative languages". In: *Software: Practice and Experience* 9.1 (1979), pp. 31–49 (page 47).

[116] Unknown. "The QED Manifesto". In: *CADE*. Vol. 814. LNAI. Springer, 1994, pp. 238–251 (page 142).

[117] F. van Raamsdonk, P. Severi, M.H. Sørensen, and H. Xi. "Perpetual reductions in λ-calculus." In: *Information and Computation* 149 (1999), pp. 173–225 (pages 48, 49).

[118] Andrei Voronkov. "AVATAR: The architecture for first-order theorem provers". In: *CAV*. Vol. 8559. LNCS. Springer, 2014, pp. 696–710 (page 19).

[119] Petar Vukmirović, Jasmin Christian Blanchette, Simon Cruanes, and Stephan Schulz. "Extending a brainiac prover to lambda-free higher-order logic". In: *TACAS*. Vol. 11427. LNCS. Springer, 2019, pp. 192–210 (pages 123, 124, 136, 144).

[120] Petar Vukmirović and Visa Nummelin. "Boolean reasoning in a higher-order superposition prover". In: *PAAR*. 2020 (pages 99, 100, 103, 105, 128, 144).

[121] Uwe Waldmann. "Superposition and chaining for totally ordered divisible abelian groups". In: *IJCAR*. Vol. 2083. LNCS. Springer, 2001, pp. 226–241 (page 20).

[122] Uwe Waldmann. *Automated reasoning II*. Lecture notes. `http://resources.mpi-inf.mpg.de/departments/rg1/teaching/autrea2-ss16/script-current.pdf`. Max-Planck-Institut für Informatik, 2016 (pages 40, 73, 78).

[123] Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. "A comprehensive framework for saturation theorem proving". In: *IJCAR*. Vol. 12166. LNCS. Springer, 2020, pp. 316–334 (pages 39, 40).

[124] Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. *A comprehensive framework for saturation theorem proving (Technical Report)*. Technical report. `http://matryoshka.gforge.inria.fr/pubs/satur_report.pdf`. 2020 (pages 38, 43, 93, 95).

[125] Daniel Wand. "Polymorphic+typeclass superposition". In: *PAAR 2014*. Vol. 31. EPiC Series in Computing. EasyChair, 2015, pp. 105–119 (pages 35, 145).

[126] C Weidenbach and A Nonnengart. In: ed. by A Robinson and A Voronkov. Elsevier Science, 2001. Chap. Small clause normal form (pages 18, 25, 113).

[127] Christoph Weidenbach. "Combining superposition, sorts and splitting". In: ed. by A Robinson and A Voronkov. Vol. 2. Elsevier Science, 2001, pp. 1965–2013 (page 19).

[128] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topić. "Spass Version 2.0". In: *CADE-18*. Vol. 2392. LNCS. Springer, 2002, pp. 275–279 (page 22).

[129] Lawrence Wos, Daniel Carson, and George Robinson. "The unit preference strategy in theorem proving". In: *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*. 1964, pp. 615–621 (page 13).

[130] Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. "A unified ordering for termination proving". In: *Science of Computer Programming* 111 (2015), pp. 110–134 (page 52).

Appendix A

Extended Proofs

Lemma 9 (Fundamental Lemma of Maximality)**.** $\|\mathcal{C}_{\mathsf{any}}\,\overline{s}_n\| = \|(\mathcal{C}_{\mathsf{any}}\,\overline{s}_n)\downarrow^w\| + 1 + isK(\mathcal{C}_{\mathsf{any}}) \times \|s_2\|$ *where* $isK(\mathcal{C}_{\mathsf{any}}) = 1$ *if* $\mathcal{C}_{\mathsf{any}} = \mathsf{K}$ *and is* 0 *otherwise.*

Proof. Assume that $\mathcal{C}_{\mathsf{any}} = \mathsf{K}$. Then any maximal reduction from $\mathsf{K}\,\overline{s}_n$ is of the form:

$$\begin{aligned} \mathsf{K}\,s_1\,s_2\ldots s_n &\longrightarrow_w^m & \mathsf{K}\,s_1'\,s_2'\ldots s_n' \\ &\longrightarrow_w & s_1'\,s_3'\ldots s_n' \\ &\longrightarrow_w^{m'} & r \end{aligned}$$

where $\|r\| = 0$, $s_1 \longrightarrow_w^{m_1} s_1' \ldots s_n \longrightarrow_w^{m_n} s_n'$, $\|s_2\| = m_2$ and $m = m_1 + \cdots + m_n$. Thus, $\|\mathsf{K}\,\overline{s}_n\| = \sum_{i=1}^n m_i + 1 + m'$. There is another method of reducing $\mathsf{K}\,\overline{s}_n$ to r:

$$\begin{aligned} \mathsf{K}\,s_1\,s_2\ldots s_n &\longrightarrow_w^{m_2} & \mathsf{K}\,s_1\,s_2'\ldots s_n \\ &\longrightarrow_w & s_1\,s_3\ldots s_n \\ &\longrightarrow_w^{m-m_2} & s_1'\,s_3'\ldots s_n' \\ &\longrightarrow_w^{m'} & r \end{aligned}$$

As the length of this reduction is the same as the previous reduction, it must be a maximal reduction as well. Therefore we have that:

$$\begin{aligned} \|\mathsf{K}\,s_1\,s_2\ldots s_n\| &= & m + m' + 1 \\ &= & (m - m_2 + m') + m_2 + 1 \\ &= & \|s_1\,s_3\ldots s_n\| + \|s_2\| + 1 \end{aligned}$$

Conversely, assume that $\mathcal{C}_{\mathsf{any}}$ is not K. I prove that the formula holds if $\mathcal{C}_{\mathsf{any}} = \mathsf{S}$. The other cases are similar. If $\mathcal{C}_{\mathsf{any}} = \mathsf{S}$, any maximal reduction from $\mathsf{S}\,\overline{s}_n$ must be of the

form:

$$\begin{aligned} \mathsf{S}\, s_1 \ldots s_n &\longrightarrow_w^m && \mathsf{S}\, s_1' \ldots s_n' \\ &\longrightarrow_w && s_1'\, s_3'\, (s_2'\, s_3')\, s_4' \ldots s_n' \\ &\longrightarrow_w^{m'} && r \end{aligned}$$

where $\|r\| = 0$, $s_1 \longrightarrow_w^{m_1} s_1' \ldots s_n \longrightarrow_w^{m_n} s_n'$ and $m = m_1 + \cdots + m_n$. There is another method of reducing $\mathsf{S}\, \overline{s}_n$ to r:

$$\begin{aligned} \mathsf{S}\, s_1 \ldots s_n &\longrightarrow_w && s_1\, s_3\, (s_2\, s_3)\, s_4 \ldots s_n \\ &\longrightarrow_w^{m+m_3} && s_1'\, s_3'\, (s_2'\, s_3')\, s_4' \ldots t_n \\ &\longrightarrow_w^{m'} && r \end{aligned}$$

Thus, we have that $\|\mathsf{S}\, \overline{s}_n\| = m + m' + 1 \leq m + m_3 + m' + 1 = \|(\mathsf{S}\, \overline{s}_n)\!\downarrow^w\| + 1$. Since $m + m' + 1$ is the length of the maximal reduction, equality must hold. □

Lemma 23. *Let $t_1 = \overline{s}_n$ and $t_2 = \overline{r}_n$ be terms such that safe(s_i, r_i) for $1 \leq i \leq n$ and $head(t_1)$, $head(t_2)$ are not fully applied combinators. Then $\|t_1\| > \|t_2\|$ if and only if there exists an $i \in \{1, \ldots, n\}$ such that $\|s_i\| > \|r_i\|$.*

Proof. I prove the $\Longrightarrow$ direction first. By Lemma 22, we have that $\|s_i\| \geq \|r_i\|$ for $1 \leq i \leq n$. I show that it cannot be the case that $\|s_i\| = \|r_i\|$ for $1 \leq i \leq n$.

Assume that $\|s_i\| = \|r_i\|$ for $1 \leq i \leq n$. Let $s_1 = \zeta\, \overline{w}_m$ and $r_1 = \xi\, \overline{v}_{m'}$. Neither ζ nor ξ can be the head of a redex and therefore, via Lemma 14 and the assumption that $\|s_1\| = \|r_1\|$, we can conclude that $\sum_{i=1}^{m} w_i = \sum_{i=1}^{m'} v_i$. Using Lemma 14 again, we have that $\|t_1\| = \sum_{i=1}^{m} w_i + \sum_{i=2}^{n} s_i = \sum_{i=1}^{m'} v_i + \sum_{i=2}^{n} s_i = \sum_{i=1}^{m'} v_i + \sum_{i=2}^{n} r_i = \|t_2\|$ contradicting $\|t_1\| > \|t_2\|$.

I now prove the $\Longleftarrow$ direction. Assume that for some $j \in \{1, \ldots, n\}$, $\|s_j\| > \|r_j\|$. Let $s_1 = \zeta\, \overline{w}_m$ and $r_1 = \xi\, \overline{v}_{m'}$. Neither ζ nor ξ can be the head of a redex and therefore, via Lemma 14, we have that $\|s_1\| = \sum_{i=1}^{m} w_i$ and $\|r_1\| = \sum_{i=1}^{m'} v_i$. Using Lemma 14 again, we have $\|t_1\| = \sum_{i=1}^{m} w_i + \sum_{i=2}^{n} s_i = \sum_{i=1}^{n} s_i$ and $\|t_2\| = \sum_{i=1}^{m'} v_i + \sum_{i=2}^{n} r_i = \sum_{i=1}^{n} r_i$. By assumption, for some $j \in \{1, \ldots, n\}$, $\|s_j\| > \|r_j\|$. For very $i \neq j$ in $\{1, \ldots, n\}$, we have $\|s_i\| \geq \|r_i\|$. Thus, $\|t_1\| > \|t_2\|$. □

Lemma 24. *Let $t_1 = \mathcal{C}_{\mathsf{any}}\, \overline{s}_n$ and $t_2 = \mathcal{C}_{\mathsf{any}}\, \overline{r}_n$ be terms such that safe(t_1, t_2). Then $\|t_1\| > \|t_2\|$ if and only if $\|s_i\| > \|r_i\|$ for some $i \in \{1, \ldots, n\}$*

Proof. I prove the $\Longrightarrow$ direction first. Assume that $\|t_1\| > \|t_2\|$ holds. The proof proceeds by induction on $\|t_1\| + \|t_2\|$.

- If $\mathcal{C}_{\mathsf{any}} = \mathsf{K}$, by the fundamental lemma of maximality, $\|t_1\| = \|s_1\, s_3 \ldots s_n\| + \|s_2\| + 1$ and $\|t_2\| = \|r_1\, r_3 \ldots r_n\| + \|r_2\| + 1$. Let $t'_1 = s_1\, s_3 \ldots s_n$ and $t'_2 = r_1\, r_3 \ldots r_n$. The following equation holds:

$$\|t'_1\| + \|s_2\| + 1 > \|t'_2\| + \|r_2\| + 1 \tag{A.1}$$

  By Lemma 21, we have $\mathit{safe}(t'_1, t'_2)$. By Lemma 20, we have $\mathit{safe}(s_2, r_2)$. Thus, by Lemma 22, $\|t'_1\| \geq \|t'_2\|$ and $\|s_2\| \geq \|r_2\|$. For Equation A.1 to hold, either $\|t'_1\| > \|t'_2\|$ or $\|s_2\| > \|r_2\|$ must be the case. If $\|s_2\| > \|r_2\|$, we are done. Therefore, assume that $\|s_2\| = \|r_2\|$ and $\|t'_1\| > \|t'_2\|$.

  - If $\mathit{head}(t'_1)$ and $\mathit{head}(t'_2)$ are not fully applied combinators, by Lemma 23, either $\|s_1\| > \|r_1\|$ or $\|s_i\| > \|r_i\|$ for $i \in \{3 \ldots n\}$ and we are done.
  - If $\mathit{head}(t'_1)$ and $\mathit{head}(t'_2)$ are fully applied combinators, then $t'_1 = \mathcal{C}_{\mathsf{any}}\, \overline{w}_m\, s_3 \ldots s_n$ and $t'_2 = \mathcal{C}_{\mathsf{any}}\, \overline{v}_m\, r_3 \ldots r_n$. Via the induction hypothesis, we have that $\|w_i\| > \|v_i\|$ for some $i \in \{1, \ldots, m\}$ or $\|s_i\| > \|r_i\|$ for some $i \in \{3, \ldots, n\}$. In the latter case, we are done. Therefore, assume $\|w_i\| > \|v_i\|$ for some $i \in \{1, \ldots, m\}$. If $\mathcal{C}_{\mathsf{any}}\, \overline{w}_m$ forms a weak redex, the $\Longleftarrow$ direction of the induction hypothesis gives us $\|s_1\| = \|\mathcal{C}_{\mathsf{any}}\, \overline{w}_m\| > \|\mathcal{C}_{\mathsf{any}}\, \overline{v}_m\| = \|r_1\|$. Otherwise, Lemma 23 provides the same result.

- If $\mathcal{C}_{\mathsf{any}} \neq \mathsf{K}$, by the fundamental lemma of maximality, $\|t_1\| = \|t'_1\| + 1$ and $\|t_2\| = \|t'_2\| + 1$ where $t'_1 = \overline{w}_m$, $t'_2 = \overline{v}_m$, $w_1 = s_1$, $v_1 = r_1$ and for $2 \leq i \leq m$ either:
  - $w_i = s_j$ and $v_i = r_j$ for some $j \in \{1, \ldots n\}$;
  - $w_i = s_j\, s_{j+1}$ and $v_i = r_j\, r_{j+1}$ for some $j \in \{1, \ldots n\}$.

  For example, if $\mathcal{C}_{\mathsf{any}} = \mathsf{I}$, $w_i = s_i$ and $v_i = r_i$ for $1 \leq i \leq n$. If $\mathcal{C}_{\mathsf{any}} = \mathsf{S}$, $w_1 = s_1$, $v_1 = r_1$, $w_2 = s_3$, $v_2 = r_3$, $w_3 = s_2\, s_3$ and $v_3 = r_2\, r_3$. Using the assumption $\|t_1\| > \|t_2\|$, we have $\|t'_1\| > \|t'_2\|$.

  - If $\mathit{head}(t'_1)$ and $\mathit{head}(t'_2)$ are not fully applied combinators, by Lemma 23, $\|w_i\| > \|v_i\|$ for some $i \in \{1, \ldots, m\}$. If $w_i = s_j$ and $v_i = r_j$ for some $j \in \{1, \ldots n\}$, we are done. Therefore, assume $w_i = s_j\, s_{j+1}$ and $v_i = r_j\, r_{j+1}$.
    - If $\mathit{head}(s_j\, s_{j+1})$ and $\mathit{head}(r_j\, r_{j+1})$ are not fully applied combinators, by Lemma 23, either $\|s_j\| > \|r_j\|$ or $\|s_{j+1}\| > \|r_{j+1}\|$ and again we are complete.
    - If $\mathit{head}(s_j\, s_{j+1})$ and $\mathit{head}(r_j\, r_{j+1})$ are fully applied combinators, then $s_j\, s_{j+1} = \mathcal{C}_{\mathsf{any}}\, \overline{u}_m\, s_{j+1}$ and $r_j\, r_{j+1} = \mathcal{C}_{\mathsf{any}}\, \overline{q}_m\, r_{j+1}$. Via the induction hypothesis, we have that $\|u_i\| > \|q_i\|$ for some $i \in \{1, \ldots, m\}$ or $\|s_{j+1}\| > \|r_{j+1}\|$. In the latter case, we are done. Therefore, assume $\|u_i\| > \|q_i\|$ for some $i \in \{1, \ldots, m\}$. If $\mathcal{C}_{\mathsf{any}}\, \overline{u}_m$ forms a weak redex, the $\Longleftarrow$ direction of the induction hypothesis gives us $\|s_j\| = \|\mathcal{C}_{\mathsf{any}}\, \overline{u}_m\| > \|\mathcal{C}_{\mathsf{any}}\, \overline{q}_m\| = \|r_j\|$. Otherwise, Lemma 23 provides the same result.

- If $head(t'_1)$ and $head(t'_2)$ are fully applied combinators, a similar line of reasoning completes the proof of this case.

I now prove the $\Longleftarrow$ direction of the lemma. Assume that $\|s_i\| > \|r_i\|$ holds for some $i \in \{1, \ldots, n\}$. The proof proceeds by induction on $\|t_1\| + \|t_2\|$.

- If $\mathcal{C}_{\mathsf{any}} = \mathbf{K}$, by the fundamental lemma of maximality, $\|t_1\| = \|s_1\, s_3 \ldots s_n\| + \|s_2\| + 1$ and $\|t_2\| = \|r_1\, r_3 \ldots r_n\| + \|r_2\| + 1$. Let $t'_1 = s_1\, s_3 \ldots s_n$ and $t'_2 = r_1\, r_3 \ldots r_n$. By Lemma 25, we have $\mathit{safe}(t'_1, t'_2)$ and thus by Lemma 22, $\|t'_1\| \geq \|t'_2\|$. Thus, if $\|s_2\| > \|r_2\|$, we can conclude $\|t_1\| > \|t_2\|$. Assume that $\|s_2\| = \|r_2\|$ and thus that either $\|s_1\| > \|r_1\|$ or $\|s_i\| > \|r_i\|$ for some $i \in \{3, \ldots, n\}$.
  - If $head(t'_1)$ and $head(t'_2)$ are not fully applied combinators, then Lemma 23 entails $\|t'_1\| > \|t'_2\|$ and thus $\|t_1\| > \|t_2\|$.
  - If $head(t'_1)$ and $head(t'_2)$ are not fully applied combinators, $t'_1 = \mathcal{C}_{\mathsf{any}}\, \overline{w}_m\, s_3 \ldots s_n$ and $t'_2 = \mathcal{C}_{\mathsf{any}}\, \overline{w}_m\, r_3 \ldots r_n$. If $\|s_i\| > \|r_i\|$ for $i \in \{3, \ldots, n\}$, the induction hypothesis can be used to conclude $\|t'_1\| > \|t'_2\|$. Therefore, assume that for $3 \leq i \leq n$, $\|s_i\| = \|r_i\|$ and $\|s_1\| > \|t_1\|$. If $\mathcal{C}_{\mathsf{any}}\, \overline{w}_m$ does not form a redex, Lemma 23 implies that $\|w_i\| > \|v_i\|$ for some $i \in \{1, \ldots, m\}$. If $\mathcal{C}_{\mathsf{any}}\, \overline{w}_m$ does form a redex, the $\Longrightarrow$ direction of the induction hypothesis implies the same. Thus, in both cases, the induction hypothesis can again be used to conclude $\|t'_1\| > \|t'_2\|$.
- If $\mathcal{C}_{\mathsf{any}} \neq \mathbf{K}$, by the fundamental lemma of maximality, $\|t_1\| = \|t'_1\| + 1$ and $\|t_2\| = \|t'_2\| + 1$ where $t'_1 = \overline{w}_m$, $t'_2 = \overline{v}_m$, $w_1 = s_1$, $v_1 = r_1$ and for $2 \leq i \leq m$ either:
  - $w_i = s_j$ and $v_i = r_j$ for some $j \in \{1, \ldots n\}$;
  - $w_i = s_j\, s_{j+1}$ and $v_i = r_j\, r_{j+1}$ for some $j \in \{1, \ldots n\}$.

  Moreover, for $1 \leq i \leq n$, there exists a $j \in \{1, \ldots, m\}$ such that $w_j = s_i$ and $v_j = r_i$ or $w_j = s_i\, s_{i+1}$ and $v_j = r_i\, r_{i+1}$ or $w_j = s_{i-1}\, s_i$ and $v_j = r_{i-1}\, r_i$. For example, if $\mathcal{C}_{\mathsf{any}} = \mathbf{B}$, $w_1 = s_1$, $w_2 = s_2\, s_{2+1}$, $w_2 = s_{3-1}\, s_3$ and for $3 < i \leq n$, $w_{i-1} = s_i$. By assumption, $\|s_i\| > \|r_i\|$ for some $i \in \{1, \ldots, n\}$. Let j be the index of one such pair. That is $\|s_j\| > \|r_j\|$
  - If $head(t'_1)$ and $head(t'_2)$ are not fully applied combinators and there exists a $k \in \{1, \ldots, m\}$ such that $w_k = s_j$ and $v_k = r_j$, by Lemma 23, $\|t'_1\| > \|t'_2\|$ and we are done.
    - Assume $w_k = s_{j-1}\, s_j$ and $v_k = r_{j-1}\, r_j$ for some $k \in \{1, \ldots, m\}$. If $head(w_k)$ and $head(v_k)$ are not fully applied combinators, by Lemma 23, $\|s_j\| > \|r_j\|$ implies that $\|w_k\| > \|v_k\|$ and thus $\|t'_1\| > \|t'_2\|$. If $head(w_k)$ and $head(v_k)$ are fully applied combinators, the induction hypothesis implies the same.
    - Assume $w_k = s_j\, s_{j+1}$ and $v_k = r_j\, r_{j+1}$ for some $k \in \{1, \ldots, m\}$.
      - If $head(w_k)$ and $head(v_k)$ are not fully applied combinators, by Lemma 23, $\|s_j\| > \|r_j\|$ implies that $\|w_k\| > \|v_k\|$ and thus $\|t'_1\| > \|t'_2\|$.

- If $head(s_j\ s_{j+1})$ and $head(r_j\ r_{j+1})$ are fully applied combinators, then $s_j\ s_{j+1} = \boldsymbol{\mathcal{C}}_{\mathsf{any}}\ \overline{u}_{m'}\ s_{j+1}$ and $r_j\ r_{j+1} = \boldsymbol{\mathcal{C}}_{\mathsf{any}}\ \overline{q}_{m'}\ r_{j+1}$. If $\boldsymbol{\mathcal{C}}_{\mathsf{any}}\ \overline{u}_{m'}$ forms a weak redex, the $\Longrightarrow$ direction of the induction hypothesis gives us $\|u_l\| > \|q_l\|$ for some $l \in \{1, \ldots, m'\}$. If $\boldsymbol{\mathcal{C}}_{\mathsf{any}}\ \overline{u}_{m'}$ forms a weak redex, Lemma 23 provides the same. Thus the induction hypothesis can be used in the $\Longleftarrow$ direction to conclude $\|w_k\| > \|v_k\|$ and thus $\|t'_1\| > \|t'_2\|$.

- If $head(t'_1)$ and $head(t'_2)$ are fully applied combinators, a similar line of reasoning completes the proof of this case.

□