Formalization of Resolution Calculus in Isabelle

Anders Schlichtkrull

Kongens Lyngby 2015

Semantic gardener cutting a semantic tree.
Drawing by Inger Schlichtkrull

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of this thesis is to formalize the resolution calculus for first-order logic and prove it sound and complete in the Isabelle proof assistant. The resolution calculus is a successful proof system, i.e. a system that can prove properties about mathematics, computer science, and much more. Two desirable properties for proof systems are soundness and completeness, because they express that every proof in the system is correct and that the system can prove all valid formulas of the given logic. Isabelle is a proof assistant i.e. a computer program that can help its user in conducting proofs, and which can check their correctness. By proving these properties in Isabelle we will gain confidence in their validity.

The thesis formalizes the resolution calculus and its soundness in Isabelle. The soundness proof is thorough and makes explicit the interplay of syntax and semantics. Likewise, the thesis formalizes two major steps of proving the resolution calculus complete, namely König's lemma, and Herbrand's theorem. The the next two major steps of proving completeness are the lifting lemma and completeness itself. The thesis discusses flaws in informal proofs of the lifting lemma from the literature, which make its formalization difficult. With these flaws in mind, the thesis discusses two possibilities for a formalization of the lemma. The thesis also describes thoroughly, albeit informally, how to finish the completeness proof. Finally, the thesis suggests possibilities for future work on the formalization of proof systems.

Summary (Danish)

Målet for denne afhandling er at formalisere resolutionskalkulen for førsteordenslogik samt at bevise at den er sund og fuldstændig i bevisassistenten Isabelle. Resolutionskalkulen er et succesfuldt bevissystem, dvs. et system der kan bevise egenskaber om matematik, informatik og meget mere. To ønskværdige egenskaber for bevissystemer er sundhed og fuldstændighed, fordi de udtrykker at ethvert bevis, som systemet laver, er korrekt, og at systemet kan bevise alle gyldige formler i den givne logik. Isabelle er en bevisassistent, dvs. et computer program der kan hjælpe sin bruger med at føre beviser, og som kan kontrollere deres korrekthed. Ved at bevise sundhed og fuldstændighed vil vi øge vores tiltro til dem.

Afhandlingen formaliserer resolutionskalkulen og dens sundhed i Isabelle. Sundhedsbeviset er grundigt og gør samspillet mellem syntaks og semantik helt tydelig. Ligeledes formaliserer afhandlingen to store skridt mod et bevis af fuldstændighed, nemlig Königs hjælpesætning og Herbrands sætning. De næste to store skridt er løftehjælpesætningen og selve fuldstændigheden. Afhandlingen diskuterer mangler i uformelle beviser fra litteraturen, som gør det svært at formalisere dem. Med disse mangler for øje diskuterer afhandlingen to muligheder for at formalisere hjælpesætningen. Afhandlingen beskriver også grundigt, omend uformelt, hvordan beviset for resolutionskalkulens fuldstændighed kan gøres færdigt. Slutteligt foreslår specialet muligheder for fremtidigt arbejde på formaliseringer af bevissystemer.

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring an M.Sc. in Engineering. The thesis deals with the formalization of the resolution calculus in the Isabelle proof assistant. The thesis is for 30 ECTS and was written in the period from 31 March 2015 to 31 August 2015. Jørgen Villadsen served as supervisor on the thesis, and Jasmin Christian Blanchette served as co-supervisor.

The prerequisites for understanding the thesis are knowledge of set theory as well as mathematical proving and reasoning. It is also advantageous to have knowledge of functional programming, and first-order logic. Knowledge of formalization work and proof assistants is not required.

Lyngby, 31 August 2015

Anders Schlichtkrull

Acknowledgements

I would like to thank my thesis supervisors Jørgen Villadsen and Jasmin Christian Blanchette for their guidance and feedback during the process. Jørgen was my personal tutor on the Honors Program, and introduced me to the world of Isabelle. I was very fortunate to get Jasmin as supervisor for my thesis. His insight in formalizations of proof systems was exceedingly valuable, and so was his thorough comments on my work.

Additionally, I would like to give special thanks to Dmitriy Traytel, who also provided me with much guidance and feedback. For all practical purposes he served as a third supervisor on the thesis, and his input was remarkably helpful.

I would also like to thank Tobias Nipkow, Peter Lammich, and Johannes Hölzl, for their teaching in the excellent course "Semantics" at TUM (Technische Universität München). I was so lucky to attend the course during my semester abroad at the university. The course made me appreciate formal semantics, and taught me the craft of formal theorem proving.

Furthermore, I would like to thank Melvin Fitting for his textbook on logic and automated theorem proving. This book together with Stefan Berghofer's formalization of its completeness proof helped spark my interest in completeness. For this, I would also like to thank Stefan Berghofer. His enumeration of terms is also used in my thesis, and I took much inspiration from his work.

Writing a thesis during the summer holidays can for many reasons be a challenge. I would therefore like to thank my fellow student Andreas Viktor Hess for company, while we were writing our theses.

I would also like to thank my parents Peter and Inger Schlichtkrull for their support. Likewise, I would like to thank my family, my friends, and my fellow residents of the Professor Ostenfeld Dormitory. Finally, I would like to thank my mother Inger Schlichtkrull for the title and colophon page drawings.

Contents

Chapter 1

Introduction

Logic is the study of reasoning. Given some knowledge, we want to study what new knowledge we can derive from it. This has many different applications. For instance, it can be used to reason about a computer program for alerting the crew of an airplane if it is diverging from its expected flight path during a landing. For instance, using logic, one can derive useful properties e.g. that an alert is issued before a collision.

In our everyday life, different people can have different ideas of what they find logical and what they do not, but in mathematics and computer science this is mostly not the case. Most mathematicians, computer scientists, and engineers can agree on which general logical derivations are correct to make. If we take out a set of these and only use these in combination to construct other derivations then we have a proof system. The chosen set is called the rules of the proof system.

Proof systems can be implemented as computer programs called automated theorem provers. The approach of automatic theorem provers is to take sentences as input, and prove them completely automatically. Another approach is proof assistants in which the user guides the computer program towards the proof. We can therefore get a computer to prove properties about an airplane alerting computer program [CM00]. In this way, we increase our confidence in the program.

The resolution calculus is one of the most successful proof systems implemented as an automatic theorem prover. The resolution calculus and an extension called superposition are for instance used in the computer programs E [Sch13], SPASS [WDF+09], and Vampire [RV99]. These provers have participated in CASC, which is the world championship of automated theorem provers. E, SPASS and Vampire have each won in several categories of the competition over the years [SS15, Sut14].

Experience has shown that humans, and even mathematicians, make mistakes when reasoning. A good example of this is the purported proof of the four color conjecture from by Kempe in 1879. The proof convinced many mathematicians, but turned out to be fatally flawed, and relied on an assumption that had counter-examples [Hea80].

In proof systems, larger derivations are build up from the few small rules that we agree on, and so the proof system ensures that the derivation is correct. Therefore, a proof system would not have accepted Kempe's proof. Since the proof was found to have errors, work has been going on to make new and correct proofs of the conjecture. A thorough proof was made in 2006 in the proof system and proof assistant of Coq [Gon08].

However, if we are to trust a proof system, it is important that its rules are actually correct and sound. It is also important that our proof system is sufficiently strong so that it can prove the properties we want to establish. The strongest proof systems are called complete.

The classical approach to show that a system is sound and complete is to make a mathematical proof of this in a natural language such as English. A recent development is the approach of complementing this with the use of proof assistants. Proof assistants can help their users in conducting proofs, and they check the correctness of proofs. The process of specifying some mathematical objects or systems in a proof assistant and proving properties about them is called formalization.

The purpose of this project is to investigate how one can formalize the proof system called the resolution calculus for first-order logic as well as prove its soundness and completeness in the Isabelle proof assistant.

- Chapter 2 introduces the first-order logic, the resolution calculus, the Isabelle proof assistant, and its underlying logic called higher-order logic.
- Chapter 3 analyzes a number of different approaches to formalizing proof systems, as well as proving their soundness and completeness.

- Chapters 4 to 6 present my approach to formalizing the resolution calculus. They describe the definitions and the proofs as well as how they are formalized. Moreover, they explain the choices I had to make and suggests alternatives.
  - Chapter 4 formalizes background theory.
  - Chapter 5 formalizes the resolution calculus and its soundness.
  - Chapter 6 formalizes König's lemma and Herbrand's Theorem. It then explains how these results can be used to prove completeness.
- Chapter 7 shows a number of examples of derivations in the resolution calculus. These examples are also formalized.
- Chapter 8 discusses the strengths and weaknesses of my approach to the formalization. It also discusses different possibilities for future work on the formalization.
- Chapter 9 concludes on my findings and suggests further work.

Chapter 2

Preliminaries and Theory Background

This chapter explains the theory of first order logic and the Isabelle proof assistant that is necessary to understand this thesis. It explains first-order logic, the resolution calculus, the Isabelle proof assistant, and the higher order logic of Isabelle.

2.1 First-order Logic

Logic is the study of reasoning. Given some knowledge, we want to consider what new knowledge we can derive from it. An example is the following:

We know that

- The pyramids were built by Egyptians or the pyramids were built by aliens.
- The pyramids were not built by aliens.

and it follows that

- The pyramids were built by Egyptians.

This example shows that we can reason logically about sentences in English. However, instead of considering logic about sentences English we now look at another language called propositional logic. Sentences in this language are called formulas. Propositional logic has several advantages over English. The most important is that has a simpler syntax and semantics, but is still quite expressive. The language consists of propositions and logical connectives. An example of a proposition is "The pyramids were built by Egyptians". In propositional logic we represent propositions with letters, so instead of "The pyramids were built by Egyptians" we could just write E. Likewise we could chose that "the pyramids were built by aliens" is denoted by A. Making such a choice is called an interpretation. Logical connectives then bind the propositions together. For instance "The pyramids were built by Egyptians or the pyramids were built by aliens" can be written $E \vee A$ where $\vee$ is the connective that means "or". Another connective is $\neg$, which gives the negation of a proposition, so "The pyramids were not built by aliens" is written $\neg A$. Let us write the example above in the language of propositional logic.

We know that

- $E \vee A$
- $\neg A$

and thus it follows that

- E

By looking at this example we can realize that in this case, it does actually not matter what E and A means. We can choose another interpretation where their meaning is something completely different, but if $E \vee A$ is true and $\neg A$ is true then we can conclude that E is also true. Therefore, we say that E follows logically from $E \vee A$ and $\neg A$. This is written as $E \vee A, \neg A \models E$.

This is an important point in propositional logic. Propositional symbols such as E and A have no preconceived meaning. If we want to give them a meaning, we need to provide an interpretation that says for each propositional symbol what its meaning is.

Let us look at another example:

We know that

- If it is raining on the priest then it is dripping on the perish clerk.
- It is raining on the priest.

and can thus it follows that

- It is dripping on the perish clerk.

We now let P denote "It is raining on the priest" and C denote "It is dripping on the perish clerk". We can now use the $\rightarrow$ connective to express "If it is raining on the priest then it is dripping on the perish clerk" as $P \rightarrow C$. Therefore, the example is as follows:

We know that

- $P \rightarrow C$
- P

and thus it follows that

- C

This derivation is again independent of our interpretation, and therefore we again say that C is a logical consequence of $P \rightarrow C$ and P, i.e. $P \rightarrow C, P \models C$.

A last example of logical consequence is the following:

We know that

- All men are mortal.
- Socrates is a man.

and thus it follows that:

- Socrates is mortal.

We could try express it as $A, B \models C$, but this is wrong since we can let A denote $2+2=4$ and B denote $2 \cdot 2 = 4$ and C denote $2-2=4$. Even though it is true that $2+2=4$ and $2 \cdot 2 = 4$ it is not true that $2-2=4$. We need a language that can capture the inner meaning of a proposition like "All men are mortal", namely what it tells us about some individuals.

Therefore, we introduce the language of first-order logic (FOL). In this language we can express many thing such as for example a mathematical inequality "$x > y$". We write it as $g(x, y)$. Here, g is called a predicate and it says something about the variables x and y. We can also represent expressions like "$x + \pi > x$ for any x" as $\forall x.\ g(p(x, \pi), x)$. Here $\forall$ is called the universal quantifier and expresses "for all x". We can let p denote the function that gives the sum of its arguments, and π denote 3.14159.... Thus, we have predicates, functions and constants in the language of first-order logic. Additionally we can use the logical connectives from the propositional logic. The variables, functions and constants can symbolize anything - not just numbers. This is enough to express the above example:

We know that

- $\forall x.\ m(x) \rightarrow d(x)$
- $m(s)$

and thus it follows that:

- $d(s)$

Here $m(x)$ denotes "x is a man" and $d(x)$ denotes "x is mortal" and s denotes "Socrates". We expressed "All men are mortal" as "For any individual, if it is a man it is mortal" which is $\forall x.\ m(x) \rightarrow d(x)$ in first-order logic. "Socrates is a man" is $m(s)$ and likewise "Socrates is mortal" is $d(s)$. Like in the previous example, we can realize that it actually does not matter what the meaning of m, d and s are. As long as $\forall x.\ m(x) \rightarrow d(x)$ and $m(s)$ hold we can still conclude that $d(s)$. So we know that $d(s)$ follows logically from $\forall x.\ m(x) \rightarrow d(x)$ and $m(s)$, which is written $\forall x.\ m(x) \rightarrow d(x), m(s) \models d(s)$.

Like in propositional logic, it is again an important point that the predicate symbols, constant symbols and function symbols have no preconceived meaning. To give them meaning we must provide a denotation of the predicate symbols that tells us what it means. Likewise, we need a denotation of constant symbols and a denotation of function symbols. These denotations together are called

an interpretation because they interpret the different symbol to give us their meaning.

2.2 Syntax

The above section gives the intuition about what we can express in first-order logic, and what the meaning of a formula in the language is. We have for instance seen that $\vee$ corresponds to "or" in English and $\rightarrow$ corresponds to "if" and "then" in English. However, even simple words like "or" and "if" can mean slightly different things in different contexts. For instance, if I tell my father that I want a bicycle or a Gameboy for my birthday, then I actually wish for both of them. At the other hand, if I tell a bartender that I would like a beer or a cider then I only want one of them, and not both.

To remedy this we now express in a mathematically precise way the syntax and the semantics of first-order logic. The syntax defines what we can write in the language, and the semantics defines the meaning of those things we can write.

Firstly, first-order logic consists of terms. We have already seen some terms such as x, y, s, π and more interestingly $p(x, y)$. In fact, a term could be much more complicated such as $f(g(x, y), h(i(x), z))$.

Definition 2.1 *A term is either*

- *A variable, x (or y, z, etc.), where x is a variable symbols.*
- *A function, e.g. f (or g, h, etc.) applied to a list of terms $t_1, ..., t_n$ that is: $f(t_1, ..., t_n)$ where f is a function symbol. In this thesis we represent symbols by letters or strings of letters. A constant is a function applied to the empty list of terms $f()$ and is often written without the parentheses.*

Secondly, first-order logic contains formulas. We have seen several formulas such as $\forall x.\ m(x) \rightarrow d(x)$ as well as $m(s)$ and $d(s)$. The formulas can contain logical connectives such as $\wedge$. There are a few more than we have seen to far. They can be summarized in a table:

Connective	Short name	Full name	Subformulas
$\wedge$	"and"	conjunction	conjuncts
$\rightarrow$	"if ... then ..." "only if"	implication	
$\vee$	"or"	disjunction	disjuncts
$\leftrightarrow$	"if and only if" "iff"	biimplication	
$\neg$	"not"	negation	
$\bot$		falsity	
$\top$		truth	

We are now ready to define formulas.

Definition 2.2 *A formula is either*

- *A predicate p applied to a list of terms $t_1, ..., t_n$ that is: $p(t_1, ..., t_n)$ where p is a predicate symbol. This is called an atomic formula.*
- *The universal quantification of a formula F: $\forall x.\ F$ where x is a variable. Any occurrence of x in F is said to be bound by the quantifier, unless it is bound by another quantifier inside of F.*
- *The existential quantification of a formula F: $\exists x.\ F$ where x is a variable. Existential quantifiers bind occurrences of variables just like universal quantifiers.*
- *The negation of a formula F: $\neg F$.*
- *The conjunction of two formulas F_1 and F_2: $F_1 \wedge F_2$.*
- *The disjunction of two formulas F_1 and F_2: $F_1 \vee F_2$.*
- *The implication from a formula F_1 to a formula F_2: $F_1 \rightarrow F_2$.*
- *The biimplication between a formula F_1 and a formula F_2: $F_1 \leftrightarrow F_2$.*
- *Falsum: $\bot$.*
- *Truth: $\top$.*

2.3 Semantics

Now that we have defined what we can write in FOL, that is, the syntax of FOL, we are ready to define precisely what the meaning terms and formulas is, that is, the semantics of FOL.

2.3.1 Interpretations

The meaning of a term is some object or element from the universe u of elements about which we wish to express something. u is a set of elements and could for instance be the set of natural numbers, the set of all C programs or the set of streets in Copenhagen. As said, the function symbols do not have any particular meaning. Therefore, we need to specify a denotation of the function symbols. A denotation of function symbols is a map that takes a function symbol and a list of elements of the universe and then returns an element of the universe.

Definition 2.3 *A function denotation is a map of type function symbol* $\Rightarrow$ *u list* $\Rightarrow$ *u.*

Likewise, we need a variable denotation.

Definition 2.4 *A variable denotation is a map from variable symbols to elements of the universe: variable symbol* $\Rightarrow$ *u.*

Lastly, we introduce predicate denotations to describe the meaning of predicates. They are maps from predicate symbols and lists of elements of the universe into the type bool consisting of true and false. The idea is that this map defines for which lists of elements the predicate is true, and for which it is false.

Definition 2.5 *A predicate denotation is a map of type predicate symbol* $\Rightarrow$ *u list* $\Rightarrow$ *u.*

As said, a predicate denotation together with a variable denotation is called an interpretation.

2.3.2 Terms

Now we are ready to give meaning to terms by specifying a semantics. We do this by introducing the semantic brackets, $[\![$ and $]\!]$, which recursively calculate the meaning of the enclosed term. In other words, they evaluate the term.

Definition 2.6 *Given a variable denotation E and a function denotation F, we can evaluate terms as follows:*

- $[\![x]\!]^E_F = E(x)$ *if x is a variable symbol.*

- $[\![f(t_1, ..., t_n)]\!]^E_F = F(f, [\ [\![t_1]\!]^E_F, ..., [\![t_n]\!]^E_F\ ])$

Consider for instance the function denotation F_{nat} which maps the symbol *zero* to the natural number 0, *one* to the natural number 1, the symbol s to the function that adds 1 to a number, *add* to addition and *mul* to multiplication.

Consider also the variable denotation E_{nat} that maps x to 26 and y to five.

We evaluate $add(mul(y, y), one()))$:

$$
\begin{aligned}
&[\![add(mul(y, y), one()))]\!]^{E_{nat}}_{F_{nat}} \\
&= F_{nat}\ add\ [\![(mul(y, y))]\!]^{E_{nat}}_{F_{nat}}\ [\![one()]\!]^{E_{nat}}_{F_{nat}} \\
&= [\![mul(y, y)]\!]^{E_{nat}}_{F_{nat}} + [\![one()]\!]^{E_{nat}}_{F_{nat}} \\
&= F_{nat}\ mul\ ([\![y]\!]^{E_{nat}}_{F_{nat}}, [\![y]\!]^{E_{nat}}_{F_{nat}}) + F_{nat}\ one\ () \\
&= ([\![y]\!]^{E_{nat}}_{F_{nat}} \cdot [\![y]\!]^{E_{nat}}_{F_{nat}}) + 1 \\
&= (E_{nat}\ y \cdot E_{nat}\ y) + 1 \\
&= (5 \cdot 5) + 1 \\
&= 26
\end{aligned}
$$

2.3.3 Formulas

Likewise, we can define the semantics for formulas. We extends the semantic brackets to cover also formulas. A formula expresses some claim that is either true or false. Thus, when we evaluate a formula we get a value of type bool.

Firstly, we need to have a definition of what the logical connectives mean. This is always fixed and can be described by the following tables:

A	B	$[\![\wedge]\!](A, B)$
true	true	true
true	false	false
false	true	false
false	false	false

A	B	$[\![\rightarrow]\!](A, B)$
true	true	true
true	false	false
false	true	true
false	false	true

A	B	$[\![\vee]\!](A,B)$
true	true	true
true	false	true
false	true	true
false	false	false

A	B	$[\![\leftrightarrow]\!](A,B)$
true	true	true
true	false	false
false	true	false
false	false	true

A	$[\![\neg]\!](A)$
true	false
false	true

$[\![\bot]\!]$
false

$[\![\top]\!]$
true

So if we want to know the value of $[\![\wedge]\!](A,B)$ when A is true and B is false, we can read it in the second row of the first table. It is false.

Sometimes we want to change a variable denotation E. We do this by writing $E[x \leftarrow e]$, which gives the same value as E for any variable symbol except for x, which gives e.

Definition 2.7 *Given a variable denotation E, a function denotation F, and a predicate denotation G, we can evaluate the truth value of a formula as follows:*

- $[\![p(t_1,...,t_n)]\!]^E_{(F,G)} = P(p, [\ [\![t_1]\!]^E_F, ..., [\![t_n]\!]^E_F\ ])$
- $[\![\neg P]\!]^E_{(F,G)} = [\![\neg]\!]([\![P]\!]^E_{(F,G)})$
- $[\![\bot]\!]^E_{(F,G)} = [\![\bot]\!]$
- $[\![\top]\!]^E_{(F,G)} = [\![\top]\!]$
- $[\![P \wedge Q]\!]^E_{(F,G)} = [\![\wedge]\!]([\![P]\!]^E_{(F,G)}, [\![Q]\!]^E_{(F,G)})$ *where $\wedge$ is any one of the binary logical connectives defined in the tables.*
- $[\![\forall x.\ P]\!]^E_{(F,G)} = \begin{cases} \textit{true} & \textit{if for all } e \in u : [\![P]\!]^{E[x\leftarrow u]}_{(F,G)} \textit{ evaluates to true} \\ \textit{false} & \textit{otherwise} \end{cases}$
- $[\![\exists x.\ P]\!]^E_{(F,G)} = \begin{cases} \textit{true} & \textit{if for some } e \in u : [\![P]\!]^{E[x\leftarrow u]}_{(F,G)} \textit{ evaluates to true} \\ \textit{false} & \textit{otherwise} \end{cases}$

We can now define what it means for a function and predicate denotation to satisfy or falsify a formula:

Definition 2.8 *An interpretation (F,G) is said to satisfy a formula A if for any variable denotation E we have that $[\![P]\!]^E_{(F,G)}$ evaluates to true. The satisfying interpretation is called a model. Conversely, if it evaluates to false then it is said to falsify the formula.*

We can also make it more clear what we mean when we talk of logical consequence and validity:

Definition 2.9 *If any model of $B_1, ..., B_n$ is also a model of A then we say that A is a logical consequence of $B_1, ..., B_n$.*

It is written $B_1, ..., B_n \models A$

Definition 2.10 *A formula A is said to be valid if any interpretation is a model for A.*

It is written $\models A$

Likewise, we define equality and equisatisfiability which are concepts that come in handy when we want to rewrite formulas.

Definition 2.11 *A formula A and a formula B are said to be equivalent if any model for A is also a model for B, and vice-versa.*

It is written $A \equiv B$

Definition 2.12 *A formula A and a formula B are said to be equisatisfiable if they have the property that:*

A is satisfiable if and only if B is satisfiable.

2.4 Proof Systems

A proof system is a formal and mechanical system for deriving formulas. Proof systems can consists of rules. A rule consists of a list of formulas called premises and a formula called the conclusion. The idea of the rule is that if we derive or assume the premises of a rule, then we can also derive its conclusion. A good example of a proof system is the natural deduction system. Here are some of its rules:

AndE1	AndE2	AndI	OrI1	OrI2
$\frac{A \wedge B}{A}$	$\frac{A \wedge B}{B}$	$\frac{A \quad B}{A \wedge B}$	$\frac{A}{A \vee B}$	$\frac{B}{A \vee B}$

To prove a formula we can combine these rules to build a proof tree. We see the premises above the line and the conclusion below the line. We can for instance prove that $(P(c) \vee Q(d)) \wedge (Q(d) \vee P(c))$ follows from $P(c) \wedge Q(d)$.

We see that $P(c) \wedge Q(d)$ matches with the premise of AndE1 if we replace A with $P(c)$ and B with $Q(c)$. Therefore, this is the beginning of our proof:

$$\frac{P(c) \wedge Q(d)}{P(c)}$$

We can now continue from $P(c)$ using OrI1. Here we replace A with $Q(d)$ and B with $P(c)$.

$$\frac{\dfrac{P(c) \wedge Q(d)}{P(c)}}{P(c) \vee Q(d)}$$

We have now a proof from $P(c) \wedge Q(d)$ of $P(c) \vee Q(d)$. Likewise we can construct a proof from $P(c) \wedge Q(d)$ of $Q(d) \vee P(c)$:

$$\frac{\dfrac{P(c) \wedge Q(d)}{P(c)}}{Q(d) \vee P(c)}$$

Using AndI we can finish our proof:

$$\frac{\dfrac{P(c) \wedge Q(d)}{P(c)}}{P(c) \vee Q(d)} \qquad \frac{\dfrac{P(c) \wedge Q(d)}{P(c)}}{Q(d) \vee P(c)}$$
$$\overline{(P(c) \vee Q(d)) \wedge (Q(d) \vee P(c))}$$

That we can prove $(P(c) \vee Q(d)) \wedge (Q(d) \vee P(c))$ from $P(c) \wedge Q(d)$ in our proof system is written $P(c) \vee Q(d) \vdash (P(c) \vee Q(d)) \wedge (Q(d) \vee P(c))$. In general, if we can prove B from assumptions $A_1, ..., A_n$ in our proof system, then we write $A_1, ..., A_n \vdash B$. If we can prove B without any assumptions, we write $\vdash B$.

Over the years, many different proof systems have been developed each with their own advantages. Some are easy to use, others are easy to understand, others are easy to reason about, and so on. The resolution calculus is a proof system that has been successfully implemented as automated theorem provers.

2.5 Soundness and Completeness

When we have a proof system, we want to ensure that the proofs it constructs are actually correct. More formally we want to ensure that if we can prove B from $A_1, ..., A_n$ then B also follows logically from $A_1, ..., A_n$. In other words, we want the proof system to be sound:

Definition 2.13 *A proof system is sound when for any formulas $A_1, ..., A_n$ and B we have that*

$$\text{if } A_1, ..., A_n \vdash B \text{ then also } A_1, ..., A_n \models B$$

Furthermore, we want the proof system to be strong, such that it can perform proofs of many formulas. The best we can hope for is that we can prove any valid formula. A system that can do that is called complete.

Definition 2.14 *A proof system is complete when for any formulas $A_1, ..., A_n$ and B we have that*

$$\text{if } A_1, ..., A_n \models B \text{ then also } A_1, ..., A_n \vdash B$$

2.6 Prenex Conjunctive Normal Form

First-order logic consists of several logical connectives and quantifiers that can be nested however we like. Therefore, it easy for us to express ourselves in the language. On the other hand, when we want to reason about the logic all the connectives and the nested structure means that there are many cases to consider. Therefore, we restrict the first-order language to formulas in Prenex Conjunctive Normal Form (PCNF). Formulas in this form use only a few connectives and do not have deep nestings. Any first-order formula can be rewritten to an equivalent PCNF formula. Formulas in PCNF consist of literals and clauses.

Definition 2.15 *A literal is an atomic formula or the negation of an atomic formula. An atomic formula is called a positive literal, and the negation a negative literal.*

Thus $p(x, f(c))$ is a positive literal and $\neg q(x)$ is a negative literal.

Definition 2.16 *The complement literal of the atomic formula $p(t_1, ..., t_n)$ is $\neg p(t_1, ..., t_n)$ and vice versa. Together the two formulas are called a complementary pair. If l is a literal, its complement can be written l^c.*

Therefore, $p(x, f(c))$ is the complement of $\neg p(x, f(c))$, and vice versa

Definition 2.17 *A formula is said to be in Conjunctive Normal Form if it is a conjunction of disjunctions of literals.*

Thus, the following is a CNF:

$$(p(f(x, y), c)) \vee q(g(c))) \wedge \neg q(f(y))$$

Definition 2.18 *A formula is said to be in Prenex Conjunctive Normal Form, if it has the form*

$$Q_1\ x_1.\ \ldots\ Q_n\ x_n.\ M$$

where $Q_1 \ldots Q_n$ each are either $\forall$ or $\exists$ and M is in Conjunctive Normal Form.

The resolution calculus additionally requires that Q_1 ... Q_n all are $\forall$. Any formula in first-order logic can be rewritten to an equisatisfiable formula in PCNF where Q_1 ... Q_n all are $\forall$.

For instance, the formula

$$((\forall x.\ p(x) \rightarrow q(x)) \wedge (\forall y.\ p(y))) \rightarrow \forall z.\ q(z)$$

has the equivalent PNCF formula

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

and the equisatisfiable PCNF formula

$$\forall z.\ (p(a) \vee \neg p(b) \vee q(z)) \wedge (\neg q(a) \vee \neg p(b) \vee q(z))$$

with only universal quantifiers.

2.7 Clausal Forms

To make it simpler for us to manipulate the formulas, we prefer to represent them as sets. To do this we take a formula $\forall x_1.$ ... $\forall x_n.\ M$ where M is in CNF and let each of the disjunctions in F be represented by the set of literals of which it consists.

Definition 2.19 *A clause is a set of literals. It represents the disjunction of these literals.*

Since a clause represents a disjunction, it is satisfied by an interpretation if for any variable denotation some literal in the clause is satisfied. A special case is the empty clause $\{\}$. By this definition it is always false, i.e., it is unsatisfiable, and so represents a contradiction.

We can then collect all the clauses of M in a set of clauses.

Definition 2.20 *A clausal form is a disjunction of clauses. It represents the conjunction of those clauses.*

For instance $(p(f(x,y),c)) \vee q(g(c))) \wedge \neg q(f(Y))$ becomes $\{p(f(x,y),c), q(g(c))\}$ and $\{q(g(c))\}$, and so the corresponding set of clauses turns out to be the set $\{\{p(f(x,y),c), q(g(c))\}, \{q(g(c))\}\}$. It is implicit that all the clauses are universally quantified.

2.8 Substitutions

Another important notion in resolution is substitution.

Definition 2.21 *A substitution σ is a map from variables to terms.*

To apply a substitution σ to a term t, we simultaneously replace each occurrence of any variable x with $\sigma(x)$. The result is written $t\{\sigma\}$. This notation is a bit different from the literature, which does not use the curly brackets and just writes $t\sigma$. We choose this alternative notation because it looks somewhat similar and can be formalized in Isabelle. Another alternative would be to use $\cdot$ or some other infix operator.

For instance if $\sigma = \{x \mapsto f(c), y \mapsto g(x,y)\}$ and $t = f(x,y)$ then $t\{\sigma\} = f(f(c), f(x,y))$.

Definition 2.22 *t is an instance of t' if there exists some substitution σ such that $t = t'\sigma$.*

For example $f(f(c), f(x,y))$ is an instance of $f(x,y)$.

Applying substitutions to literals, clauses and sets of clauses is completely analogously defined. We can also define substitution for formulas, but then we only replace variables that are not bound.

Substitutions can also be composed.

Definition 2.23 *The composition $\sigma \cdot \theta$ of one substitution σ and another θ is the substitution that first applies σ to a variable and then applies θ to the resulting term: $(\theta \cdot \sigma)(x) = (\sigma(x))\{\theta\}$*

Unifiers are a central concept of resolution:

Definition 2.24 *A unifier for a set of terms is a substitution that makes all the terms equal to each other.*

The definition for literals, clauses and sets of clauses is completely analogous.

Most general unifiers is another central concept:

Definition 2.25 *A most general unifier (mgu) σ for a set ts of terms is a unifier from which we can construct any other unifier for ts, by composition with some other substitution.*

In other words, let us say σ is an mgu of the set ts. Then for any unifier θ of ts there exists a substitution s such that $\sigma \cdot s = \theta$.

The definition for literals, clauses and sets of clauses is completely analogous.

2.9 Resolution

We are now ready to explain the resolution calculus. There are many variants of this proof system, but they are all based on the following simple rule:

$$\frac{C_1 \vee l \qquad C_2 \vee \neg l}{C_1 \vee C_2}$$

The idea of the rule is that we know that l is either true or false. In the first case we can from $C_2 \vee \neg l$ conclude C_2 and thus also $C_1 \vee C_2$. In the second case we can from $C_1 \vee l$ conclude that C_1 and thus also $C_1 \vee C_2$. So in either case we can conclude $C_1 \vee C_2$.

We now switch from writing the rule with logical formulas to writing it in the set notation of clauses that we saw in the previous section. While the notation is

not as easily readable as that of formulas, it gives us the freedom to manipulate the clauses with the mathematical set operators. The simple resolution rule can now be written

$$\frac{C_1 \qquad C_2}{(C_1 - \{l_1\}) \cup (C_2 - \{l_2\})} \begin{array}{l} l_1 \in C_1 \\ l_2 \in C_2 \\ l_1 = l_2^c \end{array}$$

In first-order logic, we also need rules that take into consideration the variables and terms. A simple rule that does this is the binary resolution rule.

$$\frac{C_1 \qquad C_2}{(C_1\{\sigma\} - \{l_1\{\sigma\}\}) \cup (C_2\{\sigma\} - \{l_2\{\sigma\}\})} \begin{array}{l} C_1 \text{ and } C_2 \text{ share no variables,} \\ l_1 \in C_1, l_2 \in C_2, \\ \sigma \text{ is an mgu for } \{l_1, l_2^c\} \end{array}$$

The rule mirrors the simple resolution rule. The main difference is that l_1 and l_2^c are unified with the mgu σ which is also applied to the clauses C_1 and C_2 when the unified literal is removed.

We need to take three steps to get from this rule to a complete proof system for first-order logic. The first step is to remedy the problem that the rule requires that the two clauses have no variables in common. We fix this by allowing the calculus to rename the variables of clauses such that they have no variables in common. This is called standardizing apart. The second step is to remedy the problem that we can only do unification of a complementary pair with one literal in each clause. It turns out that it is sometimes necessary to unify whole sets of literals. In chapter 3, we analyze two ways of doing this. The third step is to use the resolution calculus as a refutation proof system. This means that we prove the validity of a formula by proving that its negation is unsatisfiable. Since the negation is false under all interpretations, the formula must be true under all interpretations, i.e. valid. The procedure to prove a formula F valid is:

1. Negate F to get $\neg F$.
2. Rewrite $\neg F$ to the equisatisfiable set of clauses cs.
3. Use the rules of a sound resolution calculus to prove the empty clause $\{\}$ from cs.

$\{\}$ is unsatisfiable. Since the resolution calculus is sound, we get from 3. that also cs must be unsatisfiable. Since cs and $\neg F$ are both equisatisfiable, also $\neg F$ must be unsatisfiable. Then F must be valid.

Let us try to prove $(\neg P(c) \wedge \neg Q(d)) \vee Q(d) \vee (P(c) \wedge \neg Q(d))$

As described above we negate the formula: $\neg((\neg P(c) \wedge \neg Q(d)) \vee Q(d) \vee (P(c) \wedge \neg Q(d)))$, and then we turn it into an equisatisfiable set of clauses, so that we can use the resolution procedure. The set is: $\{\{P(c), Q(d)\}, \{\neg Q(d)\}, \{\neg P(c), Q(d)\}\}$. We can now construct a proof tree:

$$\dfrac{\dfrac{\{P(c), Q(d)\} \quad \{\neg Q(d)\}}{\{P(c)\}} \qquad \dfrac{\{\neg P(c), Q(d)\} \quad \{\neg Q(d)\}}{\{\neg P(c)\}}}{\{\}}$$

In this example, the mgu we used was the empty substitution.

Another way of writing a resolution proof is to write it as a derivation. A derivation is a sequence $cs_1, ..., cs_n$ of clausal forms where the step from one clausal cs_i form to the next cs_{i+1} in the sequence is performed by adding the resolvent of two clauses of cs_i. We can write such a derivation as a number of lines. The first cs_1 lines consist of the clauses of the initial clausal form. The next lines are then the clauses that were added by performing the steps. The proof tree we saw before is presented as a derivation here:

1. $\{\neg P(c), Q(d)\}$
2. $\{\neg Q(d)\}$
3. $\{P(c), Q(d)\}$
4. $\{\neg P(c)\}$
5. $\{P(c)\}$
6. $\{\}$

Since we are using the resolution calculus as a refutation proof system, we are now actually proving sets of clauses unsatisfiable. Therefore, we can define a suitable notation of completeness.

Definition 2.26 (Refutational Completeness)
If the set of clauses cs is unsatisfiable then we can prove the empty clause from cs, i.e. $cs \vdash \{\}$

2.10 Isabelle

Isabelle is a proof assistant. A proof assistant is a computer program that can help its user in conducting proofs of theorems in mathematics, logic and computer science. For instance, Isabelle has been used to prove Pythagoras' theorem [Cha15], to prove Gödel's incompleteness theorems [Pau13], to verify an operating system kernel [KEH+09] and to prove Dijkstra's algorithm correct [NL12].

To write a proof in Isabelle the user writes it in a language called Isar. Isar is similar to English and is thus readable by human beings, but it is also readable by Isabelle. Therefore, Isabelle can help its user in several ways. Most importantly, it checks that the proof is correct, and can even conduct parts of the proof automatically. Isabelle checks the correctness by using a logic. The most popular logic for Isabelle is called higher-order logic (HOL).

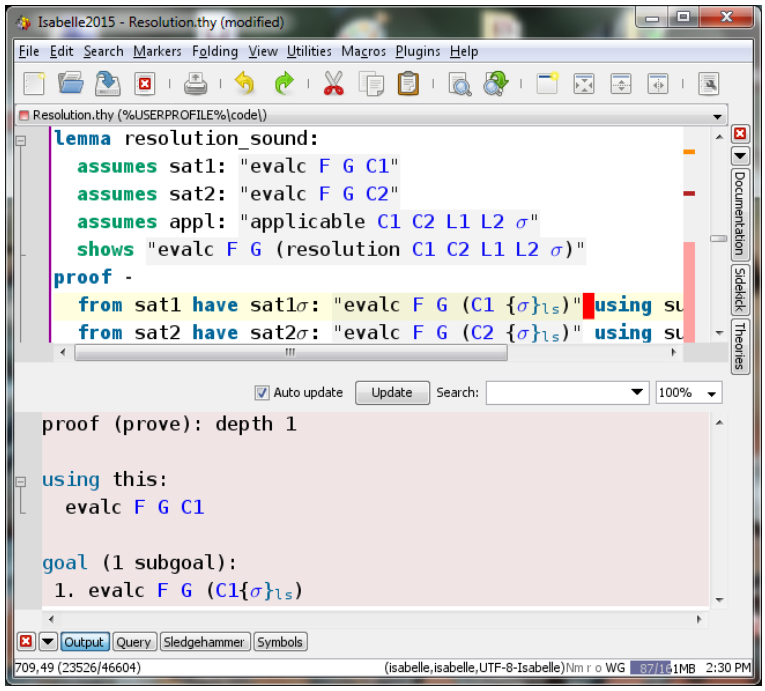

Figure 2.1: Isabelle/jEdit in action.

2.11 HOL

Higher-order logic is the logic used by several proof assistants, including Isabelle. HOL can be seen as a combination of functional programming and logic, because it contains concepts from both worlds.

From functional programming, HOL gets its recursive functions, lambda functions, higher-order functions, pattern matching, and more. HOL also has a type system similar to those known from typed functional languages such as F#, Haskell, and Standard ML. From now on, we use the word function instead of map, following the terminology of functional programming instead of that of mathematics. In HOL, a predicate is simply a function with return type *bool*.

From logic, HOL gets its logical connectives, terms, variables, quantifiers etc. They are similar to those we have seen for first-order logic. A key difference between HOL and FOL is that in HOL we can also quantify over functions, predicates and sets of individuals instead of just over individuals as in FOL.

Chapter 3

Analysis of the Problem

There are several variants of the resolution calculus, and several different approaches to proving its soundness and completeness.

This chapter analyzes several different variants of the resolution calculus from the literature. It looks at some different completeness proofs, and analyzes how well suited they are for a formalization. It concludes on which ideas I will base my formalization.

3.1 Resolution Calculus in the Literature

The variation between different resolution calculi is not only in the notation used to write the system, but also in which rules are in the system. Fitting presents three different resolution calculi that we will consider here [Fit96].

3.1.1 Binary Resolution with Factoring

We have already seen the binary resolution rule. This rule is usually used together with another rule, called the factoring rule, to form *binary resolution with*

factoring. We repeat the definition of the binary resolution rule, and introduce the factoring rule.

Definition 3.1 (Binary Resolution Rule)

$$\frac{C_1 \qquad C_2}{(C_1\{\sigma\} - \{l_1\{\sigma\}\}) \cup (C_2\{\sigma\} - \{l_2\{\sigma\}\})} \begin{array}{l} C_1 \text{ and } C_2 \text{ share no variables,} \\ l_1 \in C_1, l_2 \in C_2, \\ \sigma \text{ is an mgu for } \{l_1, l_2^c\} \end{array}$$

Definition 3.2 (Factoring Rule)

$$\frac{C}{C\{\sigma\}} \ \sigma \text{ is an mgu for } L \subseteq C$$

In some presentations the two rules can be used together in any order to construct proofs. Another way to define resolution is to let the resolvent be the binary resolvent of the factors of two clauses.

An advantage of this system is that there are two rules, each of which serve their own purpose. The factoring rule lets us unify any subset of a clause, and the resolution rule lets us resolve clauses, such that we can eventually derive the empty clause. Splitting these functions in to two rules arguably makes the presentation simpler.

3.1.2 General Resolution

General resolution is another resolution calculus[1].

Definition 3.3 (General Resolution Rule)

$$\frac{C_1 \qquad C_2}{(C_1\{\sigma\} - L_1\{\sigma\}) \cup (C_2\{\sigma\} - L_2\{\sigma\})} \begin{array}{l} C_1 \text{ and } C_2 \text{ have no variables in common,} \\ L_1 \subseteq C_1, L_2 \subseteq C_2, \\ \sigma \text{ mgu for } L_1 \cup L_2^c \end{array}$$

This rule looks very similar to the binary resolution rule, but notice that L_1 and L_2 are no longer literals, but subsets of literals in C_1 and C_2 respectively. Therefore, we can see the general resolution rule as a combination of factoring and binary resolution. The difference is that the factoring rule allowed us to do unification of any subset of literals, while the binary resolution rule only allows unification of the literals that we remove.

[1]Fitting actually calls it the General Literal Resolution Rule.

This resolution calculus is appealing because it consists only of one rule. This means that one is never in doubt about which rule to use when doing resolution proof, one only has to worry about choosing the right instantiation.

3.1.3 Resolution Suited for Hand Calculation

Fitting introduces a resolution that is suited for hand calculation.[2] The resolution is quite different from the others. Instead of having a preprocessing step where formulas are rewritten to clauses, the system works directly on formulas, but contains rules that, when applied break the formulas down to clauses. Furthermore, this system does not use most general unifiers. Instead, it requires us to guess with which terms the variables must be substituted. Therefore, the system is not well suited as the basis of an automatic theorem prover. For this reason, Fitting shows how it can be modified to be the general resolution or binary resolution.

3.1.4 Other Variants of the Resolution Calculus

We have now seen three variants of the resolution calculus. However, one can even find variants of these variants. For instance, some resolution systems apply the mgu after the literals have been removed.

There are also variants that change or restrict the rules considerably such as ordered resolution. The idea is to make the search space smaller by restricting when the resolution rule can be applied [GLJ13]. Furthermore, superposition is an extension of the resolution calculus, that also restricts the search space, and does equational reasoning [KNV+13].

3.2 Soundness and Completeness Proofs

We can prove soundness by showing that if the premises of any rule in the system are satisfied by an interpretation then so is the conclusion. Then the rules all preserve satisfiability and therefore the whole system must do the same. This can be proven by induction. As such, the overall strategy of a soundness proof

[2] In the book this is just called first-order resolution, but that name describes all the presented resolution calculi.

is simple and clear, and therefore the challenge lies in the details of the proof and its formalization.

It is more complicated to prove completeness because here the overall strategy is less clear. For any unsatisfiable clausal form we need to combine rule applications in some way to get a derivation of the empty clause. There are several approaches to this. This section explains three different approaches. It does not go in to all details, but explains the overall ideas thoroughly.

3.2.1 Semantic Trees

The approach used in among others Ben-Ari [BA12] and Chang and Lee [CL73] is semantic trees. We introduce an enumeration $A_0, A_1, A_2, \ldots$ of atomic ground formulas i.e. an infinite sequence that contains all atomic formulas that do not contain variables. The enumeration could for instance be $a(), b(), a(a()), \ldots$. For a given clausal form, the enumeration is restricted to the atoms that can be built from the function and predicate symbols that occur in the clausal form.

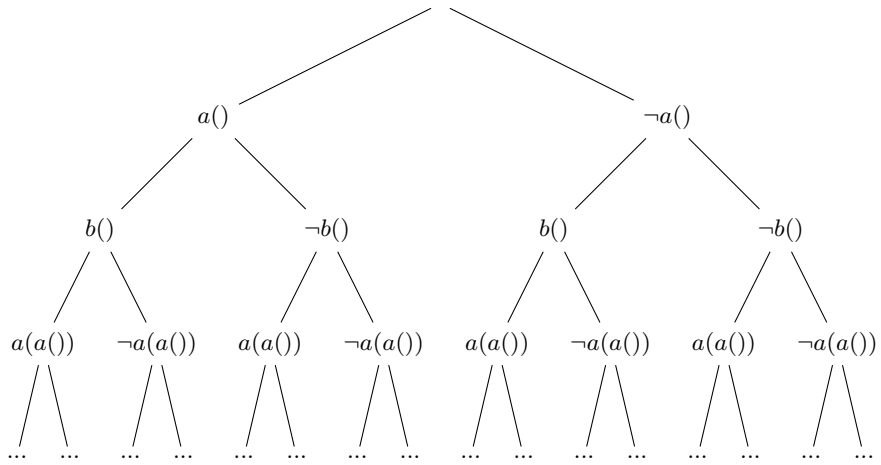

Figure 3.1: A semantic tree.

3.2.1.1 Definition

A semantic tree is a possibly infinite binary tree in which all nodes are labeled with a literal. Each level corresponds to an atomic formula. More precisely, on

the i'th level, all literals are either A_i or its negation. When we go left in a branching of the tree we come to an atom, and when we go right, we come to the negation of an atom. More precisely all left children are positive literals and all right children are negative literals. To sum up, the left children on level i of the tree are labeled with atomic formula A_i and the right children on level i are labeled with its negation $\neg A_i$. Figure 3.1 shows a semantic tree.

3.2.1.2 Partial interpretations

We can see paths in semantic trees as interpretations. In these interpretations the universe consists of the ground terms. This means that the semantic universe consists of syntactical elements, namely the ground terms. We define a denotation of functions such that ground terms evaluate to themselves. This is called the Herbrand function denotation. To define our predicate denotation, we collect all the labels of the nodes on the path into a set L. The predicate denotation G then interprets a predicate symbol and a list of elements by seeing if the corresponding atom or its negation is on the branch. More precisely:

$$G\ p\ [t_1, ..., t_n] = \begin{cases} \text{true} & \text{if } p(t_1, ..., t_n) \in L \\ \text{false} & \text{if } \neg p(t_1, ..., t_n) \in L \end{cases}$$

This is called a partial interpretation because it is only defined for the atoms on the path. For instance, the path in fig. 3.2 corresponds to the set $\{a(), \neg b()\}$, and thus to the partial interpretation $\{a() \mapsto True, b() \mapsto False\}$.

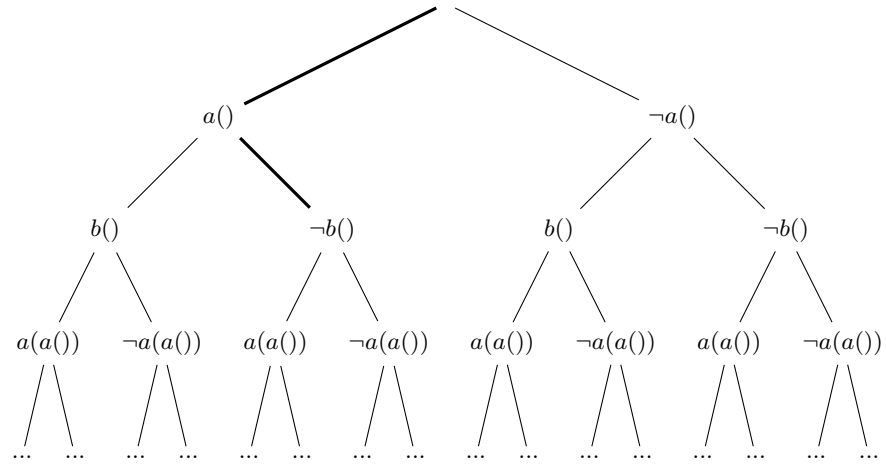

Figure 3.2: A path in a semantic tree corresponds to a partial interpretation.

3.2.1.3 Herbrand's Lemma and Closed Semantic Trees

Herbrand's lemma is a famous result. It says that if a clausal form Cs is unsatisfiable, then there exists a finite semantic tree with the following property: The branches and only the branches each falsify some instance of a clause in Cs. Such a tree is called a finite closed semantic tree for Cs.

Consider the clausal form $Cs = \{\{\neg b(), \neg a()\}, \{a(x)\}, \{a()\}, \{\neg a(), b(), \neg a(a())\}, \{\neg a(), b()\}, \{\neg a()\}\}$. It has the closed semantic tree presented in fig. 3.3.

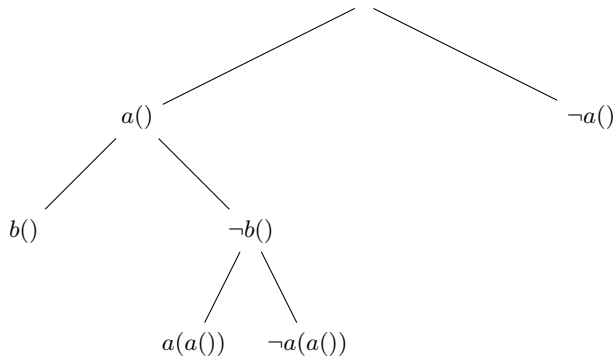

Figure 3.3: A closed semantic tree.

3.2.1.4 The Lifting Lemma

The lifting lemma can take a resolution step from ground clauses and lift it to a resolution step of first-order clauses with variables. That is, if C_1' and C_2' are instances of C_1 and C_2, and if C_1' and C_2' have resolvent C', then C_1 and C_2 have a resolvent C of which C' is an instance. This is illustrated in fig. 3.4.

For example, $C_1 = \{a(x)\}$ has instance $C_1' = \{a(a())\}$, and $C_2 = \{\neg a(), b(), \neg a(a())\}$ has instance $C_2' = \{\neg a(), b(), \neg a(a())\}$, and these instances have the resolvent $C' = \{\neg a(), b()\}$. We find that C' is an instance of $C = \{\neg a(), b()\}$, which is a resolvent of C_1 and C_2. This example was very simple, and there are much more complicated ones, with multiple occurrences of different variables.

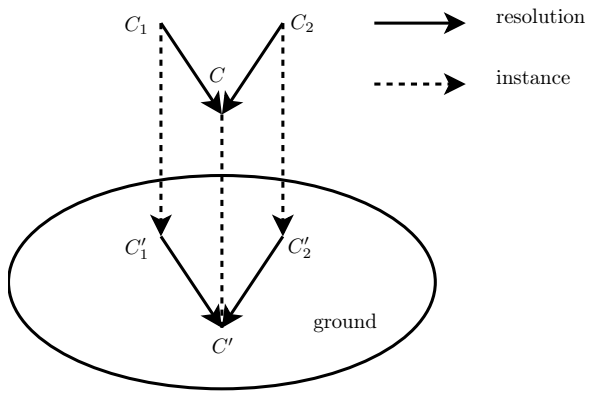

Figure 3.4: The lifting lemma.

3.2.1.5 Completeness

We now have the tools needed to prove completeness. Therefore, we consider an unsatisfiable clausal form Cs. We use Herbrand's lemma to construct a finite closed semantic tree. Next, we find a node N in the tree whose children are two branch nodes and call their labels A and $\neg A$ respectively. We let their paths to the root be B, $B_1 = B \cup \{A\}$, and $B_2 = B \cup \{\neg A\}$, respectively. Since B is not a branch of the closed semantic tree, it does not falsify a clause. But since $B \cup \{A\}$ is a branch, it must falsify some instance C_1' of a clause C_1 in Cs. It must have been making the union of $\{A\}$ and B that made all the literals in C_1' false, and thus C_1' must contain $\neg A$, and all its other literals were already falsified by B. The same is true for B_2 and a corresponding C_2' and C_2. If we look at the resolvent $C' = (C_1' - \{\neg A\}) \cup (C_2' - \{A\})$ of C_1' and C_2', then we see that it contains literals that are all falsified by B, because we removed the two that were not. Using the lifting lemma, can resolve C_1 and C_2 to get a resolvent C which has instance C'. If we remove N_1 and N_2 from the tree, but add C to Cs, then the modified tree is a closed semantic tree for the modified Cs, because B falsifies C' which is an instance of C, and all other branches are already closed.

We can keep doing this until the tree is cut down completely. It turns out that in this last step, the empty clause is derived. Thus, we have shown that for the unsatisfiable set of clauses indeed we can derive the empty clause.

We perform this process for our example in fig. 3.5.

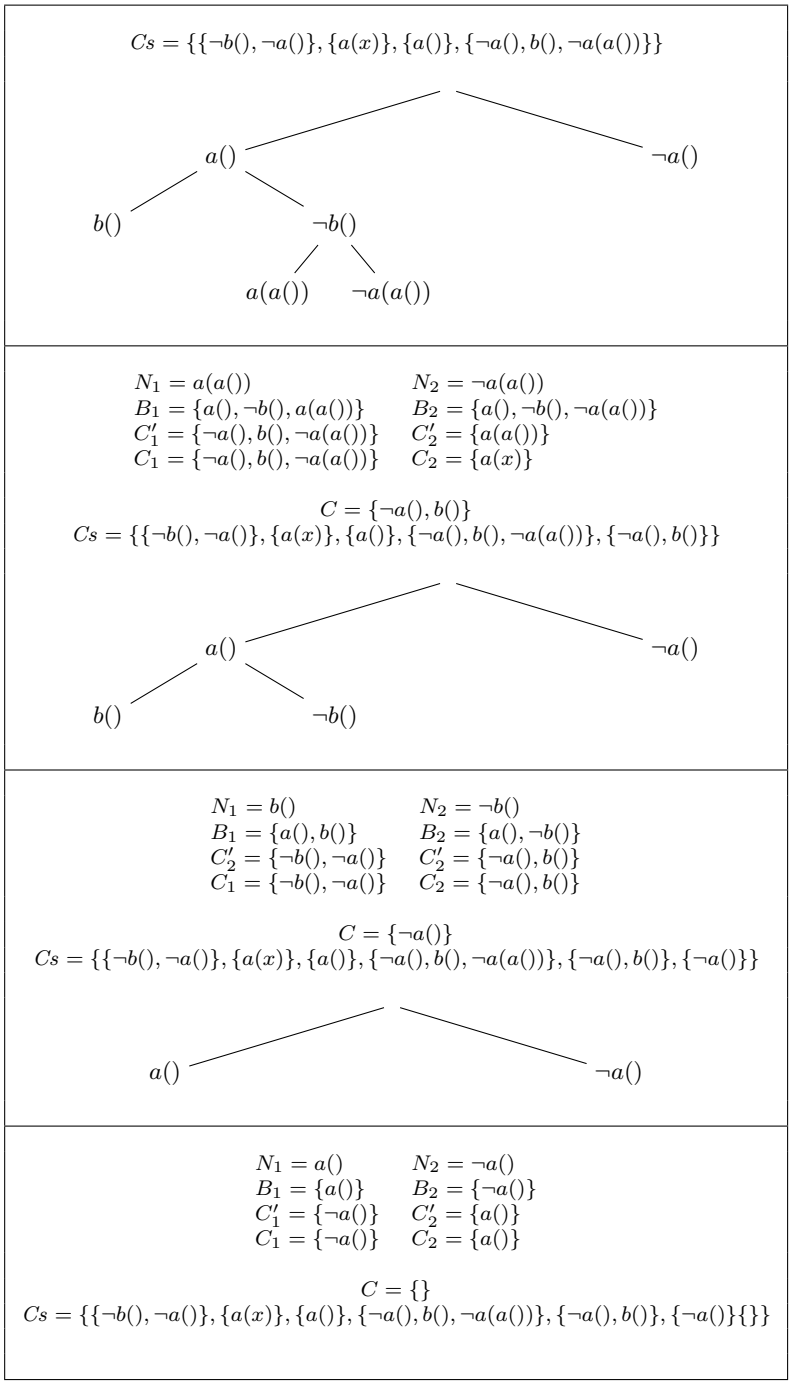

Figure 3.5: Cutting down a semantic tree to derive the empty clause following our proof.

3.2.1.6 Thoughts on the proof

This proof is in many ways very intuitive, because it uses trees, which are known from many different areas of informatics. Furthermore, the trees give us a very simple and graphical way to think about the interpretations, and we can apply results from graph theory. Completeness is a statement about interpretations, and they are represented directly in the semantic tree. Lastly, the presentation of semantic trees by Chang and Lee [CL73] is quite thorough, and goes in to many details, which can help make the formalization process go smoothly.

3.2.2 Consistency Properties

As said, Fitting [Fit96] introduces a resolution suited for hand calculation. To prove its completeness, he uses the following reformulation of completeness:

Property 1 *If $\bot$ cannot be proven from a set of formulas S, then S has a model.*

Therefore, he considers any set of formulas S from which $\bot$ cannot be derived. He then proves that S is a member of what he calls a consistency property. The famous "Model Existence Theorem" tells us that any member of a consistency property is satisfiable.

That is proven by showing that S can be expanded to a Hintikka set H. We show its definition:

Definition 3.4 *We first introduce the α and β formulas and their corresponding subformulas α_1, $\alpha 2$, β_1, and β_2:*

α	α_1	α_2
$X \wedge Y$	X	Y
$\neg(X \vee Y)$	$\neg X$	$\neg Y$
$\neg(X \rightarrow Y)$	X	$\neg Y$
$X \leftrightarrow Y$	$X \leftarrow Y$	$Y \leftarrow Y$

β	β_1	β_2
$\neg(X \wedge Y)$	X	Y
$X \vee Y$	X	Y
$(X \rightarrow Y)$	$\neg X$	Y
$\neg(X \leftrightarrow Y)$	$\neg(X \leftarrow Y)$	$\neg(Y \leftarrow Y)$

γ	$\gamma(t)$
$\forall x.F$	$F\ \{x \leftarrow t\}$
$\neg\exists x.F$	$\neg F\ \{x \leftarrow t\}$

δ	$\delta(t)$
$\neg\forall x.F$	$\neg F\ \{x \leftarrow t\}$
$\exists x.F$	$F\ \{x \leftarrow t\}$

Definition 3.5 *A Hintikka set is a set with the following properties:*

1. *H does not contain both an atomic formula A and its negation* $\neg A$.

2a. *H does not contain* $\bot$.

2b. *H does not contain* $\neg\top$.

3. *If H contains a double negation* $\neg\neg Z$ *of a formula then H also contains* Z.

4. *If H contains an alpha formula* α *then H also contains* α_1 *and* α_2.

5. *If H contains a beta formula* β *then H also contain* β_1 *or* β_2.

6. *If H contains a gamma formula* γ *then H contains* $\gamma(t)$ *for every closed term t of the language.*

7. *If H contains a delta formula* δ *then H contains* $\gamma(t)$ *for some closed term of the language.*

Then it is shown that any Hintikka set has a model. The model uses the Herbrand function denotation and the predicate denotation G is then defined:

$$G\ P\ [t_1, ..., t_n] = \begin{cases} \text{true} & \text{if } P(t_1, ..., t_n) \in H \\ \text{false} & \text{otherwise} \end{cases}$$

It is then easy to show that this interpretation satisfies H, and thus also the subset S. The proof is by induction. Fitting also shows how to lift the proof to general resolution and binary resolution with factoring using a lifting lemma.

The main appeal of this approach is that a lot of the formalization work has been done beforehand. Berghofer's formalization in Isabelle of a natural deduction proof system [Ber07] uses the framework of consistency properties to prove soundness and completeness. Therefore, one could formalize the resolution system from Fittings book and prove it complete by applying the lemmas formalized by Berghofer. The challenging part would be to extend the proof to work for also general resolution or binary resolution with factoring.

3.2.3 Unified Completeness

A third approach is presented in Blanchette, Popescu, and Traytel's completeness proof for a Genzen proof system [BPT14b].

The idea here is that we first prove that our proof system is persistent, meaning that if at one step of our derivation, we can apply a rule, then we are able to do the same in all the following steps. If we can prove this, then it has been shown that our proof system also has the following *abstract completeness property*:

Property 2 (Abstract Completeness)
For any set of clauses, either there is a proof of the empty clause or it has an infinite model path.

A model path is defined as a derivation on which any rule that is ever enabled is eventually used. The idea is that the model path tries every single possibility of applying rules to derive the empty clause, but still goes on forever.

The next step is to prove that a model path corresponds to a model for the given clauses, because then we can lift the abstract completeness property to:

Property 3 *For any set of clauses, either there is a proof of the empty clause or the set of clauses have a model.*

This is equivalent to refutational completeness.

To obtain the completeness result we thus have to prove the system persistent, and to prove the correspondence between model paths and models. The resolution calculus is persistent, since it keeps adding clauses to a set without ever removing them. Thus, a rule that could be applied in any step can also be applied in the following steps. However, it is not easy to prove the correspondence between model paths and models.

An opportunity to do so is presented by Bachmair and Ganzinger [BG01] in a setup that is similar to the unified completeness. It is done by introducing the notion of partial interpretations below clauses and candidate models. The idea of these is that they are attempts at constructing models. To define them an ordering $\succ$ on ground terms, literals, and clauses is introduced. Then the partial interpretation I_C *below* a clause is introduced. I_C is an interpretation that satisfies all the clauses that are below C in the ordering $\succ$ of clauses. Its definition is not so intuitive, so it is not be presented here.

We also define I^C, the partial interpretation *at* C, which satisfies the same clauses as I_C as well as C itself. Lastly, we define the candidate model I_{Cs} for a set of clauses Cs. It is constructed using the partial interpretations of its clauses. The idea of a candidate model is that if a set of clauses is satisfiable, then the candidate model is a model, i.e. a witness of the satisfiablity. We therefore let Cs be the set of the clauses on our model path, and then I_{Cs} is candidate model

for Cs. We now define a counter-example to be a clause in Cs that is falsified by I_{Cs}. It is proven that if Cs were to contain a counter-example, then it would also contain the empty clause. The argument is that then the empty clause could be derived using the counter-example, and since the model path takes every opportunity to derive things, it would indeed be derived. However, the model path does by definition not contain the empty clause (since then it would not be infinite), and thus it cannot contain a counter-example. Therefore, I_{Cs} satisfies all clauses and is thus a model.

Bachmair and Ganzinger shows this result for a proof system called ground resolution, but also indicate how this result can be lifted to the resolution calculus for first-order logic using a lemma called the lifting lemma.

The unified completeness comes with a formalization in Isabelle [BPT14a]. Therefore, like the consistency properties, this approach is appealing, because some of the work is already done. Furthermore, it is quite elegant that we take the infinite derivation and from that build a model more or less directly. As said, it should be easy to prove resolution persistent. However, we still need to make a formalization of the candidate models. This could prove to be difficult, because the ordering on formulas is not very intuitive. Additionally, we have to find out how to lift the result from ground resolution to a resolution for first-order logic.

3.3 Other Considerations

For a resolution prover to be useful, we need to implement an algorithm to translate formulas of the first-order logic to clauses. This is useful because then we are be able to prove formulas correct. Most, if not all, resolution based automatic theorem provers include such machinery, and it is a challenge in itself to write an efficient rewriting algorithm.

To make an automatic theorem prover, we need to find a way to make the proof system executable. It is possible to extract Standard ML code from an Isabelle theory, to get an executable program. This approach has been used to make a verified automatic theorem prover [Rid04]. It does, however, require quite some work to go from an abstract definition of a proof system to one that is executable.

3.4 Other Presentations

There are several other interesting presentations of the resolution calculus than those discussed above. One is by Robinson, the original inventor of the resolution calculus [Rob79]. Another is a book by Leitsch dedicated entirely to resolution [Lei97]. Furthermore, Bachmair and Ganzinger have written an advanced chapter on resolution [BG01].

3.5 The Approach of This Project

In this project, I have chosen to formalize general resolution. My motivation is that it is well suited for automation because of its use of most general unifiers. This mean that my formalization eventually could serve as the basis of an automatic theorem prover. Furthermore, a proof of the completeness of this rule can easily be adapted to a proof of the completeness of binary resolution with factoring [Fit96].

I have chosen to make a formalization of the completeness using the semantic tree approach. My assessment is that this approach is most likely to be successful. Semantic trees can be drawn as graphs on a paper, which often helps with intuition. This choice also gives me a better understanding of the process of formalizing completeness, since I cannot cut corners by using the already developed frameworks for consistency properties [Ber07] and unified completeness [BPT14b]. Furthermore, it also makes the project more interesting seen from a research perspective, because in this way my project contributes a formalization of not only resolution but also semantic trees. I mostly base my formalization of semantic trees on the presentations of Ben-Ari [BA12] and of Chang and Lee [CL73].

Instead of looking at how to rewrite general formulas to clauses and at making the system into an automatic theorem prover, the focus is on formalizing resolution, soundness and completeness. This in itself is both interesting and challenging.

Chapter 4

Formalization: Logical Background

This chapter formalizes most of the logical background from chapter 2 up to and including substitution. The chapter shows formalized definitions, lemmas, and theorems of the logical background. The chapter introduces concepts from Isabelle, Isar, and HOL as they are used in the formalization process.

4.1 Terms

We wish to formalize terms. Therefore, we first need to formalize variable symbols, function symbols, and predicate symbols. We represent them by strings, introducing the types *var-sym*, *fun-sym*, and *pred-sym* as synonyms for the *string* type.

type-synonym *var-sym* = *string*
type-synonym *fun-sym* = *string*
type-synonym *pred-sym* = *string*

A more abstract approach is to represent them by type variables, which can then later be instantiated by a concrete type. On paper, strings are, however, almost always used, and a concrete type makes later definitions simpler since we do not have to carry the type variables around. Another advantage is that the type of strings is countably infinite which means that the strings can be enumerated. Both concrete types [Rid04] and type variables [Ber07] have been used in other formalizations.

We now formalize the terms of first-order logic as a type *fterm*. For this purpose we use datatypes, which is a well-known concept from typed functional programming.

```
datatype fterm =
  Fun fun-sym fterm list
| Var var-sym
```

This mirrors our definition from chapter 2, which stated that a term is either a variable or a function symbol applied to a list of terms.

Here are some examples of terms:

```
value Var ''x''
value Fun ''one'' []
value Fun ''mul'' [Var ''y'', Var ''y'']
value Fun ''add'' [Fun ''mul'' [Var ''y'', Var ''y''], Fun ''one'' []]
```

In the syntax of chapter 2 they respectively correspond to $x, one(), mul(y, y)$, and $add(mul(y, y), one())$.

Since that chapter described syntax, it was clear that a variable x was different from a function application $f(c(), d())$ and therefore we did not state it explicitly. Fortunately, datatypes have the same property. Terms made with different constructors are by definition also different.

4.2 Literals

We can likewise define literals as a datatype with one constructor *Pos* for positive literals and another *Neg* for negative literals:

```
datatype 't literal =
  Pos pred-sym 't list
| Neg pred-sym 't list
```

The datatype is parametrized with a type variable *'t*. We can instantiate it with *fterm* to get the type *fterm literal*, and then the symmetry to chapter 2 is clear: A literal consists of a predicate symbol and a list of terms, and it is positive or negative. The reason we use the type variable is that we in chapter 6 define Herbrand terms as a type *hterm*, and then Herbrand literals are simply *hterm literal*s. It is also convenient to have selector functions *get-pred*, which given a literal gives us its predicate symbol, and *get-terms*, which gives us the list of terms. Furthermore, it is convenient to have a predicate *is-pos*, which tells us if a literal is positive. We can get Isabelle to create these functions as follows:

```
datatype 't literal =
  is-pos: Pos (get-pred: pred-sym) (get-terms: 't list)
| Neg (get-pred: pred-sym) (get-terms: 't list)
```

An alternative would have to formalize a datatype that covers all the logical formulas. However, in chapter 3 we chose not to formalize the rewriting from formulas to clauses, and thus we only need a representation of literals. There are however formalizations the formulas of first-order logic available [Ber07, Rid04, MR04].

Here are some examples of literals:

```
value Pos ''greater'' [Var ''x'', Var ''y'']
value Neg ''less'' [Var ''x'', Var ''y'']
value Pos ''less'' [Var ''x'', Var ''y'']
value Pos ''equals''
      [Fun ''add''[Fun ''mul''[Var ''y'',Var ''y''], Fun ''one''[]],Var ''x'']
```

The literals correspond in our syntax from 2 to $greater(x, y)$, $\neg less(x, y)$, $less(x, y)$, and $equals(add(mul(x, x), one()), y)$.

It is easy to define a function that gives us the complement of a literal. Isabelle function definitions look similar to those of typed functional programming.

```
fun complement :: 't literal ⇒ 't literal where
  complement (Pos P ts) = Neg P ts
| complement (Neg P ts) = Pos P ts
```

We can use a notation similar to that of chapter 2 to make lemmas and theorems easier to read. Luckily, Isabelle allows us to do that with its mixfix notation. We therefore instead define the complement as:

fun *complement* :: *'t literal* ⇒ *'t literal* (-^{c} [*300*] *300*) **where**
  $(Pos\ P\ ts)^c = Neg\ P\ ts$
| $(Neg\ P\ ts)^c = Pos\ P\ ts$

This allows us to write l^c instead of *complement l*.

If we take the double complement $(l^c)^c$ of a literal, we obviously get the literal l. We prove this small lemma in Isabelle:

lemma *cancel-comp1*: $(l^c)^c = l$ **by** (*cases l*) *auto*

Here, we named the lemma *cancel-comp1* and stated it as $(l^c)^c = l$. We have made Isabelle prove it by writing **by** (*cases l*) *auto*. Here (*cases l*) *auto* tells Isabelle how it should conduct the proof. Writing *cases l* splits the lemma in to two cases - one where l is a positive literal and one where it is negative. Then it leaves it up to us to prove the cases separately. We use *auto* to prove them automatically. Writing *auto* invokes a proof method that does primarily simplifications, but also some other forms of reasoning. This concludes the proof in Isabelle.

There are many other proof methods including *simp*, *blast*, and *metis*. They each have their strengths and weaknesses. It is not necessary to know in details how they work, but it is an advantage to have an intuition about when to use the different methods. Here is a short explanation of my intuition for the different proof methods. The pure simplification component of *auto* is *simp*. It knows about the Isabelle library from a large number of simplification rules. Sometimes the reasoning of auto means that it works itself into a corner, where *simp* instead would have been able to succeed by using only simplification rules. On the other hand, *simp* sometimes gives up because it does not do the reasoning that auto does. The proof method *blast* is efficient at solving problems that are, in nature, first-order problems. Furthermore, it often chooses the right instantiations of the lemmas with which we provide it. The proof method *metis* is a resolution prover, and is sometimes able to do the proofs where the other methods give up. Furthermore, Isabelle has a built-in tool called Sledgehammer that applies automatic theorem provers such as E, SPASS, and Vampire as well as satisfiability-modulo-theories such as CVC4 and Z3 to prove theorems or statements automatically [Bla15]. The tool often suggests that we use *metis* together with an appropriate set of lemmas.

We do another proof of a simple property of literals:

lemma *cancel-comp2*:
  assumes *asm*: $l_1{}^c = l_2{}^c$
  shows $l_1 = l_2$
proof −
  from *asm* **have** $(l_1{}^c)^c = (l_2{}^c)^c$ **by** *auto*
  then have $l_1 = (l_2{}^c)^c$ **using** *cancel-comp1*[*of* l_1] **by** *auto*
  then show *?thesis* **using** *cancel-comp1*[*of* l_2] **by** *auto*
qed

This lemma states that assuming $l_1{}^c = l_2{}^c$ we have the thesis $l_1 = l_2$. Therefore, we are proving that if the complements of two literals are the same, then so are the literals. By writing **proof**, we tell Isabelle that we want to prove this ourselves using however many steps we want or we have to. The "−" tells Isabelle that we want to make a direct proof. Our proof starts from the assumption and shows from it that $(l_1{}^c)^c = (l_2{}^c)^c$ using the auto proof method. From this we show $l_1 = (l_2{}^c)^c$ by canceling out the two applications of *complement* on the left hand side. Writing **using** *cancel-comp1*[*of* l_1] *by auto* means that we use *auto* to show this proof line and that it can use the lemma *cancel-comp1*[*of* l_1]. The lemma *cancel-comp1*[*of* l_1] is the same as *cancel-comp1*, except that l is instantiated with l_1. Then we show our thesis using *cancel-comp2*[*of* l_2] to cancel out the applications of *complement* on the right hand side. In Isabelle the thesis automatically gets the short name *?thesis*. Writing **qed** indicates that the proof is done.

This style of writing a proof is similar to how one could write a proof in a natural language. Words like **from, have, then, using,** and **show** mean (at least almost) the same as they do in English. In longer Isabelle proofs, the similarity to natural language is even clearer.

Another easy proof is that of the existence of a complement

lemma *comp-exi1*: $\exists\, l'.\ l' = l^c$ **by** (*cases* l) *auto*

Note that $\exists$ is *not* the existential quantifier for the first-order logic we consider in the thesis. It is instead the existential quantifier of the HOL of Isabelle. In English we could express the lemma as "any literal has a complement", but since we are writing in the HOL of Isabelle we must use its existential quantifier. The same is the case when we see the $\forall$ quantifier, and the connectives $\neg$, $\longrightarrow$, $\longleftrightarrow$, $\vee$, and $\wedge$. A connective we did not see in first-order logic was $\Longrightarrow$. In Isabelle, this

arrow separates an assumption from a conclusion. The statements $A \Longrightarrow B$ and $A \longrightarrow B$ are logically equivalent. It is often more appropriate to write $A \Longrightarrow B$, but $A \longrightarrow B$ can be nested inside any logical construction. We will not dwell more on the difference, which can be studied elsewhere [NK14].

It takes a bit more effort to show the following similar lemma:

lemma *comp-exi2*: $\exists\, l.\ l' = l^c$
proof
  show $l' = (l'^c)^c$ **using** *cancel-comp1*[*of* l'] **by** *auto*
qed

Again, we tell Isabelle that we want to prove the theorem in some steps, but this time we do not use the $-$. This means that Isabelle choses an appropriate rule for us to prove the lemma. Since the lemma is an existential quantification, Isabelle chooses the rule $P\,x \Longrightarrow \exists x.\, P\,x$. The rule says that we can prove $\exists x.\, P\,x$ by proving $P\,x$ for some x. In the proof of *comp-exi2* the $P\,x$ corresponds to $(l' = x^c)$. The x for which we prove it, is l'^c.

We also prove

lemma *comp-swap*: ${l_1}^c = l_2 \longleftrightarrow l_1 = {l_2}^c$
proof $-$
  have ${l_1}^c = l_2 \Longrightarrow l_1 = {l_2}^c$ **using** *cancel-comp1*[*of* l_1] **by** *auto*
  moreover
  have $l_1 = {l_2}^c \Longrightarrow {l_1}^c = l_2$ **using** *cancel-comp1* **by** *auto*
  ultimately
  show *?thesis* **by** *auto*
qed

Here we see a new concept, namely the "... **moreover** ... **ultimately**" construction. In this construction the last statement before "**moreover**" and the last statement before "**ultimately**" are collected and provided to the application of auto on the statement following "**ultimately**". If we wanted to collect more statements, we could have used additional "**moreover**" commands.

4.3 Clauses

We now formalize clauses. Sets are already defined in Isabelle, and so it is easy to define clauses as a synonym for sets of literals:

type-synonym *'t clause* = *'t literal set*

The clauses we consider are always finite, but a set can actually be infinite. Therefore, another possibility would have been to use the type *fset* of finite sets. Furthermore, some presentations of the resolution calculus use multisets. We use sets because their notation in Isabelle is similar to the well-known mathematical notation. The only major difference from mathematical notation is that Isabelle uses a dot instead of the bar when writing for instance the set of all elements with a certain property $\{x.\ P(x)\}$. The proof methods of Isabelle reason efficiently about sets.

We extend the concept of complement to sets of literals, i.e. clauses.

abbreviation *complementls* :: *'t literal set* ⇒ *'t literal set* ($-^C$ [*300*] *300*) **where**
  L^C ≡ *complement* ‘ *L*

The superscript C used here is capital. We use *abbreviation* to define *complementls*. This seems similar to **fun**, and is used in much the same way. However, contrary to **fun**, it does not introduce a function in the logic. Instead Isabelle introduces *complementls L* as a syntactical shorthand for *complement* ‘ *L*.

The operator ‘ is a higher-order map function. It applies *complement* to every literal in L. In other words, L^C is the image of L under the function *complement*.

4.4 Collecting Variables

In chapter 2 we looked at occurrences of variables and we saw that the general resolution rule required that its clauses did not share any variables. We therefore create functions that collect the variables that occur in terms, lists of terms, literals, and sets of literals. On terms we define it recursively:

fun *varst* :: *fterm* ⇒ *var-sym set*
and *varsts* :: *fterm list* ⇒ *var-sym set* **where**
  varst (*Var x*) = {*x*}
| *varst* (*Fun f ts*) = *varsts ts*
| *varsts* [] = {}
| *varsts* (*t* # *ts*) = (*varst t*) ∪ (*varsts ts*)

The # is the cons operator with takes element t and puts it at the beginning of list *ts*. Note that Isabelle functions are required to be be terminating. For many functions Isabelle can automatically check that this is the case [NK14].

On literals we just look in the term:

```
definition varsl :: fterm literal ⇒ var-sym set where
  varsl l = varsts (get-terms l)
```

Here we used **definition** instead of **fun** to define a non-recursive function. In contrary to **fun**, definition does not add simplification rules to the proof methods of Isabelle. Therefore, we have to unfold the definition manually when necessary. We can use **definition** when we want to reason about a function by using its properties instead of having Isabelle unfold its definition automatically.

On sets of literals we take the union over the literals in the clause to collect their variables:

```
definition varsls :: fterm literal set ⇒ var-sym set where
  varsls L ≡ ⋃ l∈L. varsl l
```

4.5 Ground

A ground term is a term in which no variables occur. As we noted in the analysis, the semantic trees are labeled with ground literals, so this concept must also be formalized. We can easily make a recursive predicate that checks if a term does not contains variables:

```
fun ground :: fterm ⇒ bool where
  ground (Var x) ⟷ False
| ground (Fun f ts) ⟷ (∀ t ∈ set ts. ground t)

abbreviation grounds :: fterm list ⇒ bool where
  grounds ts ≡ (∀ t ∈ set ts. ground t)
```

Since *ground* and *grounds* return booleans, we use $\longleftrightarrow$ in the definition instead of $=$. We then extend the definition to literals and sets of literals. For terms, we simply check the list of terms:

abbreviation *groundl* :: *fterm literal* ⇒ *bool* **where**
groundl l ≡ *grounds* (*get-terms l*)

And for sets of literals, we check all the contained literals:

abbreviation *groundls* :: *fterm clause* ⇒ *bool* **where**
groundls L ≡ ∀ *l* ∈ *L*. *groundl l*

4.6 Semantics

We are ready to introduce the semantics of terms, literals, and clauses. Firstly, we need to define function denotations, predicate denotations, and variable denotations:

type-synonym *'u fun-denot* = *fun-sym* ⇒ *'u list* ⇒ *'u*
type-synonym *'u pred-denot* = *pred-sym* ⇒ *'u list* ⇒ *bool*
type-synonym *'u var-denot* = *var-sym* ⇒ *'u*

Here, we have let the universe be represented by a type variable *'u*. This mean that we are able to use our logic to reason about elements of any type by instantiating *'u* with the desired type.

If we for instance want to reason about the natural numbers, we can instantiate *'u* with the type *nat*, and reason about functions and predicates on natural numbers. We can do that by defining a *nat fun-denot*, a *nat pred-denot*, and a *nat var-denot*.

fun F_{nat} :: *nat fun-denot* **where**
F_{nat} *f* [*n*,*m*] =
(*if f* = *''add''* *then n* + *m else*
if f = *''mul''* *then n* ∗ *m else 0*)
| F_{nat} *f* [] =
(*if f* = *''one''* *then 1 else*
if f = *''zero''* *then 0 else 0*)
| F_{nat} *f us* = *0*

fun G_{nat} :: *nat pred-denot* **where**
G_{nat} *p* [*x*,*y*] =

```
    (if p = ''less'' ∧ x < y then True else
     if p = ''greater'' ∧ x > y then True else
     if p = ''equals'' ∧ x = y then True else False)
| G_nat p us = False
```

```
fun E_nat :: nat var-denot where
  E_nat x =
    (if x = ''x'' then 26 else
     if x = ''y'' then 5 else 0)
```

Alternatively, we could have fixed a specific type, or we could have defined an abstract type for the purpose of representing the universe. There are examples of using a type variable [Ber07] and of using an abstract type [MR04, Rid04] in formalizations. We choose to use a type variable, because it allows our logic to reason about anything that can be formalized as a type.

We now formalize the semantics of terms:

```
fun evalt :: 'u var-denot ⇒ 'u fun-denot ⇒ fterm ⇒ 'u where
  evalt E F (Var x) = E x
| evalt E F (Fun f ts) = F f (map (evalt E F) ts)
```

We use the higher order function *map* to apply the function *evalt E F* to *ts*, i.e. *map* (*eval E F*) $[t_1, \ldots, t_n] = [\text{eval E F } t_1, \ldots, \text{eval E F } t_n]$.

As an example we can try to evaluate the terms, function denotations and variable denotations we used as examples.

```
lemma evalt E_nat F_nat (Var ''x'') = 26
  by auto
lemma evalt E_nat F_nat (Fun ''one'' []) = 1
  by auto
lemma evalt E_nat F_nat (Fun ''mul'' [Var ''y'', Var ''y'']) = 25
  by auto
lemma
  evalt E_nat F_nat (Fun ''add'' [Fun ''mul'' [Var ''y'', Var ''y''], Fun ''one'' []]) = 26
  by auto
```

We often need to apply the evaluation of terms to lists of terms:

```
abbreviation evalts :: 'u var-denot ⇒ 'u fun-denot ⇒ fterm list ⇒ 'u list where
  evalts E F ts ≡ map (evalt E F) ts
```

The next step is to evaluate literals. This is done by evaluating the list of terms, and using the predicate denotation to get the meaning of the predicate symbol:

```
fun evall :: 'u var-denot ⇒ 'u fun-denot ⇒ 'u pred-denot ⇒ fterm literal ⇒ bool
where
  evall E F G (Pos p ts) ⟷ (G p (evalts E F ts))
| evall E F G (Neg p ts) ⟷ ¬(G p (evalts E F ts))
```

We present some examples:

```
lemma evall E_nat F_nat G_nat (Pos ''greater'' [Var ''x'', Var ''y'']) = True
  by auto
lemma evall E_nat F_nat G_nat (Neg ''less'' [Var ''x'', Var ''y'']) = True
  by auto
lemma evall E_nat F_nat G_nat (Pos ''less'' [Var ''x'', Var ''y'']) = False
  by auto
lemma evall E_nat F_nat G_nat
      (Pos ''equals''
        [Fun ''add'' [Fun ''mul'' [Var ''y'',Var ''y''],Fun ''one'' []]
        ,Var ''x'']
      ) = True
  by auto
```

In chapter 2 we noticed that a clause is satisfied by an interpretation (F, G), if, for any variable denotation, some literal in the clause is satisfied by (F, G). We use this to define the semantics of clauses.

```
definition evalc :: 'u fun-denot ⇒ 'u pred-denot ⇒ fterm clause ⇒ bool where
  evalc F G C ⟷ (∀ E. ∃ l ∈ C. evall E F G l)
```

Clausal forms represent conjunctions, c.f. chapter 2. Therefore, a clausal form *Cs* is satisfied by an interpretation if it satisfies all the clauses in *Cs*.

```
definition evalcs :: 'u fun-denot ⇒ 'u pred-denot ⇒ fterm clause set ⇒ bool where
  evalcs F G Cs ⟷ (∀ C ∈ Cs. evalc F G C)
```

Since ground terms do not contain variables their evaluation is independent of the variable denotation. Therefore, given an interpretation, a ground term evaluates to the same even under two different variable denotations.

lemma *ground-var-denott*: *ground t* $\Longrightarrow$ (*evalt E F t* = *evalt E′ F t*)

We prove this using structural induction.

PROOF. First we have the case where *t* is a variable *Var x*. In this case we assume *ground* (*Var x*) and then have to show *evalt E F* (*Var x*) = *evalt E′ F* (*Var x*). This is easy, because *ground* (*Var x*) is a contradiction since variables are not ground, and thus we can conclude anything, in particular *evalt E F t* = *evalt E' F t*.

Next, we have the case where *t* is a function application *Fun f ts*. Our induction hypothesis is that the lemma holds for any subterm of *Fun t ts*, i.e.,

$$t \in set\ ts \Longrightarrow ground\ t \Longrightarrow (evalt\ E\ F\ t = evalt\ E'\ F\ t)$$

We now assume *ground* (*Fun t ts)* and have to show *evalt E F* (*Fun t ts*) = *evalt E' F* (*Fun t ts*). Consider any subterm *t* of *ts*. Since *Fun f ts* is ground, so must the subterm *t* be. We then use the induction hypothesis, and conclude *evalt E F t* = *evalt E' F t*. Since any subterm of *ts* evaluates to the same under both *E* and *E′*, we conclude that so does *ts*: *evalts E F ts* = *evalts E' F ts*. By the definition of *evalt* we conclude that also *Fun t ts* evaluates to the same under both variable denotations: *evalt E F* (*Fun t ts*) = *evalt E' F* (*Fun t ts*)

Q. E. D.

We prove this formally in Isabelle:

```
lemma ground-var-denott: ground t ⟹ (evalt E F t = evalt E′ F t)
proof (induction t)
  case (Var x)
  then have False by auto
  then show ?case by auto
next
  case (Fun f ts)
  then have ∀ t ∈ set ts. ground t by auto
  then have ∀ t ∈ set ts. evalt E F t = evalt E′ F t using Fun by auto
  then have evalts E F ts = evalts E′ F ts by auto
  then have F f (map (evalt E F) ts) = F f (map (evalt E′ F) ts) by metis
  then show ?case by simp
qed
```

The proof is very similar to the informal proof that we just finished. Firstly, we state **proof** (*induction t*) which means that the proof is by induction on *t*. We start by proving the first case for variables by stating **case** (*Var x*). The expression **case** (*Var x*) refers to the assumption that *Var x* is ground. Therefore, we can use it to prove *False*, a contradiction. From the contradiction we prove *?case*, which refers to the desired conclusion for the current case, i.e. *evalt E F* (*Var x*) = *evalt E' F* (*Var x*).

We continue with the next case by writing **next** and stating **case** (*Fun f ts*). Here, **case** (*Fun f ts*) contains the induction hypothesis and the assumption *ground* (*Fun t ts*). We use the assumption to prove that any subterm is ground. Then we apply the induction hypothesis by writing **using** *Fun* which refers to **case** (*Fun f ts*). Hereafter, we take additional steps following the informal proof and finally derive the desired conclusion *?case* for this case which is *evalt E F* (*Fun f ts*) = *evalt E' F* (*Fun f ts*).

4.7 Substitutions

Substitutions are maps from variables to terms, and thus we define them as such.

type-synonym *substitution* = *var-sym* ⇒ *fterm*

In chapter 2 we described substitution as the simultaneous replacement of variables occurring in a term. In our formalization we define this much more precisely by writing a recursive function that performs the substitution.

fun *sub* :: *fterm* ⇒ *substitution* ⇒ *fterm* (-{-}$_t$ [*300*,*0*] *300*) **where**
  (*Var x*){σ}$_t$ = σ *x*
| (*Fun f ts*){σ}$_t$ = *Fun f* (*map* (λ*t*. *t* {σ}$_t$) *ts*)

For variables, we apply the substitution directly. For function applications we map the lambda-function λ*t*. *t*{σ}$_t$ on to *ts*. The lambda-function does substitution with σ on any *term* to which it is applied. Our notation is the same as the one introduces in chapter 2, except that it uses a subscript *t* to indicate that this is substitution on a *term*.

We also define substitution for lists of terms, literals, and lists of literals. For lists we map the substitution on to the list:

abbreviation *subs* :: *fterm list* ⇒ *substitution* ⇒ *fterm list* (-{-}$_{ts}$ [*300*,*0*] *300*)
where
  $ts\{\sigma\}_{ts} \equiv (map\ (\lambda t.\ t\ \{\sigma\}_t)\ ts)$

For literals we apply the substitution to the list of terms:

fun *subl* :: *fterm literal* ⇒ *substitution* ⇒ *fterm literal* (-{-}$_l$ [*300*,*0*] *300*) **where**
  $(Pos\ p\ ts)\{\sigma\}_l = Pos\ p\ (ts\{\sigma\}_{ts})$
| $(Neg\ p\ ts)\{\sigma\}_l = Neg\ p\ (ts\{\sigma\}_{ts})$

For sets of literals we use the ‘ function again:

abbreviation *subls* :: *fterm literal set* ⇒ *substitution* ⇒ *fterm literal set* (-{-}$_{ls}$ [*300*,*0*] *300*) **where**
  $L\ \{\sigma\}_{ls} \equiv (\lambda l.\ l\ \{\sigma\}_l)\ `\ L$

Furthermore, we define instances. We use the existential quantifier of HOL:

definition *instance-oft* :: *fterm* ⇒ *fterm* ⇒ *bool* **where**
  instance-oft $t_1\ t_2 \longleftrightarrow (\exists \sigma.\ t_1 = t_2\{\sigma\}_t)$

definition *instance-ofts* :: *fterm list* ⇒ *fterm list* ⇒ *bool* **where**
  instance-ofts $ts_1\ ts_2 \longleftrightarrow (\exists \sigma.\ ts_1 = ts_2\{\sigma\}_{ts})$

definition *instance-ofl* :: *fterm literal* ⇒ *fterm literal* ⇒ *bool* **where**
  instance-ofl $l_1\ l_2 \longleftrightarrow (\exists \sigma.\ l_1 = l_2\{\sigma\}_l)$

We also define a special kind of substitution called variable renaming. Such substitutions can be used for standardizing clauses apart. A variable renaming simply renames the variables as the name implies.

definition *var-renaming* :: *substitution* ⇒ *bool* **where**
  var-renaming $\sigma \longleftrightarrow (\forall x.\ \exists y.\ \sigma\ x = Var\ y)$

It is formalized as a substitution in which any variable symbol x is mapped to some variable $Var\ y$. A more restrictive way of defining it would be to require

that this mapping should also be a bijection. Then one could not make a renaming that renames two different variables to the same name. We, however, choose the simpler definition.

A special substitution is the empty substitution ε which maps any variable symbol to the corresponding variable term. This is the same as what the constructor *Var* does, and so we just use that:

abbreviation ε :: *substitution* **where**
  $\varepsilon \equiv Var$

4.7.1 Composition

We formalize the composition of substitutions:

definition *composition* :: *substitution* $\Rightarrow$ *substitution* $\Rightarrow$ *substitution* (**infixl** $\cdot$ *55*)
where
  $(\sigma_1 \cdot \sigma_2)\ x = (\sigma_1\ x)\{\sigma_2\}_t$

It is defined as an infix operator which means that we can write $\sigma \cdot \theta$ instead of *composition* σ θ.

The definition follows that from chapter 2, which said that the composition of one substitution σ and another θ corresponds to first applying σ to a variable and then applying θ to the resulting term.

We can prove that this holds not only for a variable, but also for terms. This can be proven automatically in Isabelle using structural induction on the term:

lemma *composition-conseq2t*: $t\{\sigma_1\}_t\{\sigma_2\}_t = t\{\sigma_1 \cdot \sigma_2\}_t$
proof (*induction t*)
  case (*Var x*)
  have (*Var x*)$\{\sigma_1\}_t\{\sigma_2\}_t = (\sigma_1\ x)\{\sigma_2\}_t$ **by** *simp*
  also have ... $= (\sigma_1 \cdot \sigma_2)\ x$ **unfolding** *composition-def* **by** *simp*
  finally show *?case* **by** *auto*
next
  case (*Fun t ts*)
  then show *?case* **unfolding** *composition-def* **by** *auto*
qed

In the variable case we use the "**also** ... **finally**" construction to make a proof of $(Var\ x)\{\sigma_1\}_t\{\sigma_2\}_t = (\sigma_1 \cdot \sigma_2)\ x$ from left to right. The idea is that ... refers to the right hand side of the line above. We can have as many lines of "**also**" as we want. Furthermore, *unfolding composition-def* instructs Isabelle to find any occurrences of *composition* and replace them with their definition before the following proof method, *simp*, is applied.

Furthermore, composition is associative:

lemma *composition-assoc*: $\sigma_1 \cdot (\sigma_2 \cdot \sigma_3) = (\sigma_1 \cdot \sigma_2) \cdot \sigma_3$
proof
fix x
show $(\sigma_1 \cdot (\sigma_2 \cdot \sigma_3))\ x = ((\sigma_1 \cdot \sigma_2) \cdot \sigma_3)\ x$ **unfolding** *composition-def* **using** *composition-conseq2t* **by** *simp*
qed

Since we wrote **proof**, and not **proof** −, Isabelle choses an appropriate rule for us to make the proof. The rule is $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow f = g$. Therefore, we fix an arbitrary x and show that $(\sigma_1 \cdot (\sigma_2 \cdot \sigma_3))x = ((\sigma_1 \cdot \sigma_2) \cdot \sigma_3)x$.

The *instance-oft* relation is transitive, i.e., if t_1 is an instance of t_2 which is an instance of t_3 then t_1 is an instance of t_3.

Proof. Since t_1 is an instance of t_2 we can obtain a substitution σ_{12} such that $ts_1 = ts_2\{\sigma_{12}\}_{ts}$. Likewise, since t_2 is an instance of t_3 we can obtain a substitution σ_{23} such that $ts_2 = ts_3\{\sigma_{23}\}_{ts}$. By combining these equations we get $ts_1 = (ts_3\{\sigma_{23}\}_{ts})\{\sigma_{12}\}_{ts}$ and thus also $ts_1 = ts_3\{\sigma_{23} \cdot \sigma_{12}\}_{ts}$ using *composition-conseq2ts*. Therefore, ts_1 is an instance of ts_3. Here is the formalization of the proof:

lemma *instance-ofts-trans* :
assumes ts_{12}: *instance-ofts* ts_1 ts_2
assumes ts_{23}: *instance-ofts* ts_2 ts_3
shows *instance-ofts* ts_1 ts_3
proof −
from ts_{12} **obtain** σ_{12} **where** $ts_1 = ts_2\ \{\sigma_{12}\}_{ts}$
unfolding *instance-ofts-def* **by** *auto*
moreover
from ts_{23} **obtain** σ_{23} **where** $ts_2 = ts_3\ \{\sigma_{23}\}_{ts}$
unfolding *instance-ofts-def* **by** *auto*
ultimately
have $ts_1 = ts_3\ \{\sigma_{23}\}_{ts}\ \{\sigma_{12}\}_{ts}$ **by** *auto*
then have $ts_1 = ts_3\ \{\sigma_{23} \cdot \sigma_{12}\}_{ts}$ **using** *composition-conseq2ts* **by** *simp*

then show *?thesis* **unfolding** *instance-ofts-def* **by** *auto*
qed

It clearly mirrors the natural language proof using also the terminology *obtain* to give names to the substitutions that come from the existential claims of the definition of *instance-of*.

4.7.2 Unifiers

We now turn to unifiers. Recall that a unifier for a set of terms is a substitution that makes all the terms equal to each other.

definition *unifiert* :: *substitution* ⇒ *fterm set* ⇒ *bool* **where**
unifiert σ *ts* ⟷ $(\exists\, t'.\ \forall\, t \in ts.\ t\{\sigma\}_t = t')$

We have formalized this by saying that if σ is a unifier for *ts* then there is some literal t' which all literals in $t\{\sigma\}_t$ are equal to.

A set of literals can also have a unifier:

definition *unifierls* :: *substitution* ⇒ *fterm literal set* ⇒ *bool* **where**
unifierls σ *L* ⟷ $(\exists\, l'.\ \forall\, l \in L.\ l\{\sigma\}_l = l')$

An alternative definition for unifiers of non-empty set is that when we apply the unifier to a set, the set becomes a singleton. We prove that this works as a definition by proving its equivalence to *unifierls*. First, we prove

lemma *unif-sub*:
assumes *unif*: *unifierls* σ *L*
assumes *nonempty*: $L \neq \{\}$
shows $\exists\, l.$ *subls L* $\sigma = \{$*subl l* $\sigma\}$
proof −
from *nonempty* **obtain** *l* **where** $l \in L$ **by** *auto*
from *unif this* **have** $L\ \{\sigma\}_{ls} = \{l\ \{\sigma\}_l\}$ **unfolding** *unifierls-def* **by** *auto*
then show *?thesis* **by** *auto*
qed

Then we prove the equivalence:

lemma *unifierls-def2*:
  assumes *L-elem*: $L \neq \{\}$
  shows *unifierls* σ $L \longleftrightarrow (\exists\, l.\ L\ \{\sigma\}_{ls} = \{l\})$
proof
  assume *unif*: *unifierls* σ L
  from *L-elem* **obtain** l **where** $l \in L$ **by** *auto*
  then have $L\ \{\sigma\}_{ls} = \{l\ \{\sigma\}_l\}$ **using** *unif* **unfolding** *unifierls-def* **by** *auto*
  then show $\exists\, l.\ L\{\sigma\}_{ls} = \{l\}$ **by** *auto*
next
  assume $\exists\, l.\ L\ \{\sigma\}_{ls} = \{l\}$
  then obtain l **where** $L\ \{\sigma\}_{ls} = \{l\}$ **by** *auto*
  then have $\forall\, l' \in L.\ l'\{\sigma\}_l = l$ **by** *auto*
  then show *unifierls* σ L **unfolding** *unifierls-def* **by** *auto*
qed

Isabelle again chooses an appropriate rule when we write **proof**. To prove $A \longleftrightarrow B$ we must first prove $A \Longrightarrow B$ and then $B \Longrightarrow A$.

Since a unifier makes all literals in a set into the same literal, it does the same for a subset, and is thus also a unifier for the subset:

lemma *unifier-sub1*: *unifierls* σ $L \Longrightarrow L' \subseteq L \Longrightarrow$ *unifierls* σ L'
  unfolding *unifierls-def* **by** *auto*

Lastly, we define most general unifiers:

definition *mgut* :: *substitution* $\Rightarrow$ *fterm set* $\Rightarrow$ *bool* **where**
  mgut σ *fs* $\longleftrightarrow$ *unifiert* σ *fs* $\wedge$ $(\forall\, u.$ *unifiert* u *fs* $\longrightarrow (\exists\, i.\ u = \sigma \cdot i))$

definition *mguls* :: *substitution* $\Rightarrow$ *fterm literal set* $\Rightarrow$ *bool* **where**
  mguls σ $L \longleftrightarrow$ *unifierls* σ $L \wedge (\forall\, u.$ *unifierls* u $L \longrightarrow (\exists\, i.\ u = \sigma \cdot i))$

The definition states that σ should be a unifier, and that any other unifier u can be made from it using composition with some other substitution i.

Chapter 5

Formalization: Resolution Calculus and Soundness

In this chapter we formalize the resolution calculus and prove its soundness. The chapter explains the Isabelle proofs but does not show them since they are more complicated than those of chapter 4. For the proof code in Isabelle, we refer to appendix C.

5.1 The Resolution Calculus

We recall the general resolution rule.

$$\frac{C_1 \qquad C_2}{(C_1\{\sigma\}_{ls} - L_1\{\sigma\}_{ls}) \cup (C_2\{\sigma\}_{ls} - L_2\{\sigma\}_{ls})} \begin{array}{l} C_1 \text{ and } C_2 \text{ share no variables,} \\ L_1 \subseteq C_1, L_2 \subseteq C_2, \\ \sigma \text{ mgu for } L_1 \cup {L_2}^c \end{array}$$

We first formalize its side condition as a predicate *applicable* that takes two clauses, two subsets of those clauses, and a substitution, and checks whether the side conditions hold:

definition *applicable* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *bool* **where**
applicable C_1 C_2 L_1 L_2 σ ⟷
$C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
∧ *varsls* $C_1 \cap$ *varsls* $C_2 = \{\}$
∧ $L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
∧ *mguls* σ $(L_1 \cup {L_2}^C)$

Note that we added the extra condition that L_1, L_2, C_1, and C_2 should be non-empty, because if C_1 or C_2 is empty, then we have already derived the empty clause, and if L_1 or L_2 is empty, then we will not remove literals from both clauses.

We formalize the resolution rule as a function that takes C_1, C_2, L_1, L_2, and σ and gives us the corresponding resolvent.

definition *resolution* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *fterm clause* **where**
resolution C_1 C_2 L_1 L_2 $\sigma = (C_1\ \{\sigma\}_{ls} - L_1\ \{\sigma\}_{ls}) \cup (C_2\ \{\sigma\}_{ls} - L_2\ \{\sigma\}_{ls})$

We also define the steps that we use to form derivations. It is defined as a predicate *resolution-step*. Here, *resolution-step cs cs'* means that we can perform a step from *cs* to *cs*′:

inductive *resolution-step* :: *fterm clause set* ⇒ *fterm clause set* ⇒ *bool* **where**
resolution-rule:
$Cs_1 \in Cs \Longrightarrow Cs_2 \in Cs \Longrightarrow$ *applicable* C_1 C_2 L_1 L_2 $\sigma \Longrightarrow$
resolution-step Cs $(Cs \cup \{$*resolution* C_1 C_2 L_1 L_2 $\sigma\})$
| *standardize-apart*:
$C \in Cs \Longrightarrow$ *var-renaming* $\sigma \Longrightarrow$ *resolution-step* Cs $(Cs \cup \{C\ \{\sigma\}_{ls}\})$

It is defined as an inductive definition that consists of two rules. The word *inductive* means that the only resolution steps are the ones constructed with these rules [NK14]. The first rule states that if we can apply resolution to two clauses then we can add one of their resolvents in a step. The second rule states that we may add a renamed clause in a step. We can use this to standardize clauses apart.

Now that we have defined the individual steps, we can define resolution derivations as the predicate *resolution-deriv*.

definition *resolution-deriv* :: *fterm clause set* ⇒ *fterm clause set* ⇒ *bool* **where**
resolution-deriv = *star resolution-step*

We use the *star* predicate from the Isabelle distribution, which extends a predicate to its reflexive transitive closure, i.e. allows us to apply *resolution-step* zero or more times to construct a derivation.

5.2 Soundness of the Resolution Rule

Now that we have defined our resolution calculus, we are ready to prove it sound. We will prove soundness by showing that if the premises of the resolution rule are satisfied in an interpretation, when so is its conclusion. Then the soundness of the resolution calculus follows by induction.

To prove the resolution rule sound, we first prove that it is sound to apply a most general unifier to a clause. Next we prove that a very simple version of resolution is sound. We combine these results to prove that the resolution rule is sound.

5.2.1 Soundness of Substitution

Instead of proving directly that it is sound to apply a most general unifier, we prove the more general result, that it is sound to apply a substitution. We state this lemma:

lemma *subst-sound*:
assumes *asm*: *evalc F G C*
shows *evalc F G* (*C* $\{\sigma\}_{ls}$)

The lemma states that if C is satisfied by the interpretation (F, G), then so is $C\{\sigma\}_{ls}$. Let us think about why this is reasonable. The variables in a clause C are universally quantified. Therefore, we can substitute them with any term, and the result is still satisfied.

However, the above explanation is not formal enough. The problem is that we mix up the syntactic terms with the semantic elements. That the variables in C are universally quantified means that no matter which elements of the universe

u' they point to in a variable denotation E, the clause C still evaluates to true. This does not say anything about substitutions.

Therefore it is tempting to discard the above argument, but fortunately, there is a correspondence between variable denotations and substitutions. In a way, they both specify the meaning of variables. The variable denotations do it on the semantic level by assigning elements to variables, while the substitutions do it on the syntactic level by assigning terms to variables.

We illustrate this correspondence by showing that we can transform a substitution to a variable denotation. Let us look at a substitution $\sigma = \{x_1 \mapsto t_1, x_2 \mapsto t_2, ..., \}$. If we already know a variable denotation E and a function denotation F, then we can evaluate t_1, t_2, ... , i.e. we already know their meaning. This transforms σ to an environment $E' = \{x_1 \mapsto \mathit{evalt}\ E\ F\ t_1,\ x_2 \mapsto \mathit{evalt}\ E\ F\ t_2, ...\}$. We call this transformation the evaluation of a substitution and define it in Isabelle.

fun *evalsub* :: *'u fun-denot* ⇒ *'u var-denot* ⇒ *substitution* ⇒ *'u var-denot* **where**
evalsub F E σ = (*evalt E F*) ∘ *σ*

The definition uses the well-known function composition operator ∘ which construct the function that first applies σ and then *eval E F*.

When we want the meaning of $t\{\sigma\}_t$ under an environment E, we first apply the substitution. At each former occurrence of a variable x_i in t, there is now a term t_i, which will be evaluated under E. However, for all such terms, we could also also have evaluated them beforehand, by evaluating the substitution. We formalize this in the following lemma:

lemma *substitutiont*: *evalt E F* (*t* {*σ*}$_t$) = *evalt* (*evalsub F E σ*) *F t*

The lemma is known in the literature as the substitution lemma [EFT96]. We also prove the lemma for lists of terms and for literals:

lemma *substitutionts*: *evalts E F* (*ts* {*σ*}$_{ts}$) = *evalts* (*evalsub F E σ*) *F ts*

lemma *substitutionl*: *evall E F G* (*l* {*σ*}$_l$) ⟷ *evall* (*evalsub F E σ*) *F G l*

Evaluation of substitutions was the concept that was missing for us to prove substitution sound.

PROOF. We assume that C is satisfied by interpretation (F, G). We have to prove that $C\{\sigma\}_{ls}$ is satisfied by the same interpretation, i.e. that under any variable denotation E, we can find a literal in $C\{\sigma\}_{ls}$ that is satisfied. Therefore, we choose an arbitrary variable denotation E and fix it. From our assumption we know that C is true in (F, G) under any variable denotation. In particular, it is true under the variable denotation $evalsub\ F\ E\ \sigma$ that we get by evaluating σ. Therefore, there is some literal l, in C that is true under (F, G) and $evalsub\ F\ E\ \sigma$. But then, using the substitution lemma, we know that $l\{\sigma\}_l$ must be true under (F, G) and our initial environment E. Thus there is a literal $l\{\sigma\}_l$ in $C\{\sigma\}_{ls}$, that is true under (F, G) and E, which means that $C\{\sigma\}_{ls}$ itself is true under (F, G) and E. Since E was chosen arbitrarily, $C\{\sigma\}_{ls}$ is true under any interpretation, i.e. $C\{\sigma\}_{ls}$ is satisfied by (F, G). Thus, we have proven instantiation sound. Q. E. D.

The Isabelle proof is a straightforward formalization of these arguments.

5.2.2 Soundness of Simple Resolution

We now, as promised, turn to proving the simple resolution rule from chapter 2 sound. We recall the rule:

Definition 5.1 (Simple Resolution)

$$\frac{C_1 \qquad C_2}{(C_1 - \{l_1\}) \cup (C_2 - \{l_2\})} \begin{array}{l} l_1 \in C_1 \\ l_2 \in C_2 \\ l_1 = {l_2}^c \end{array}$$

And state its soundness:

lemma *simple-resolution-sound*:
assumes $C_1 sat$: *evalc F G* C_1
assumes $C_2 sat$: *evalc F G* C_2
assumes $l_1 inc_1$: $l_1 \in C_1$
assumes $l_2 inc_2$: $l_2 \in C_2$
assumes *Comp*: ${l_1}^c = l_2$
shows *evalc F G* $((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))$

The two first assumptions are the assumptions of the soundness property, and the three next are the side conditions of the rule. In chapter 2 we had an explanation of the rule, and our soundness proof is similar to this explanation.

PROOF. We assume that C_1 and C_2 are satisfied by interpretation (F, G). Since l_1 and l_2 are a complementary pair, either l_1 or l_2 is satisfied by (F, G). Assume that l_1 is the one that is satisfied. Since l_2 is its complement, it is not satisfied. But since $l_2 \in C_2$ and C_2 is satisfied, there must be some other literal l_2' in C_2 that is satisfied. This literal is also in $(C_1 - \{l_1\}) \cup (C_2 - \{l_2\})$, and so it is also satisfied.

If we instead assume that l_2 was satisfied by (F, G), the argument would be entirely analogous, and so, in any case $(C_1 - \{l_1\}) \cup (C_2 - \{l_2\})$ is satisfied. Q. E. D.

The formalization of this proof is also straightforward.

5.2.3 Combining the Rules

We now prove resolution sound by combining the simple resolution rule and substitution. The soundness lemma is formalized as

lemma *resolution-sound*:
  assumes sat_1: *evalc F G* C_1
  assumes sat_2: *evalc F G* C_2
  assumes *appl*: *applicable* C_1 C_2 L_1 L_2 σ
  shows *evalc F G* (*resolution* C_1 C_2 L_1 L_2 σ)

We recall the definition of *applicable* that occurs in the third assumption.

definition *applicable* **where**
  applicable C_1 C_2 L_1 L_2 $\sigma \longleftrightarrow$
      $C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
    $\wedge$ *varsls* $C_1 \cap$ *varsls* $C_2 = \{\}$
    $\wedge$ $L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
    $\wedge$ *mguls* σ $(L_1 \cup ({L_2}^C))$

PROOF. We consider some literal $l_1 \in L_1$. Since σ is an mgu for $L_1 \cup {L_2}^c$, it is also a unifier for L_1. This means that all the literals in $L_1\{\sigma\}$ are the same i.e. they are $l_1\{\sigma\}$. In other words, $L_1\{\sigma\}_{ls} = \{l_1\{\sigma\}_l\}$. Symmetrically we can consider some literal $l_2 \in L_2$ and conclude that $L_2\{\sigma\}_{ls} = \{l_2\{\sigma\}_l\}$. Since $l_1 \in L_1 \subseteq C_1$, we conclude that $l_1\{\sigma\}_l \in C_1\{\sigma\}_{ls}$ and we can likewise

conclude that $l_2\{\sigma\}_l \in C_2\{\sigma\}_{ls}$. Since $l_1 \in L_1 \cup {L_2}^c$ and ${l_2}^c \in L_1 \cup {L_2}^c$ also $l_1\{\sigma\}_l \in (L_1 \cup {L_2}^c)\{\sigma\}_{ls}$ and ${l_2}^c\{\sigma\}_l \in (L_1 \cup {L_2}^c)\{\sigma\}_{ls}$. Since σ unifies $(L_1 \cup {L_2}^c)$ it must also have unified l_1 and ${l_2}^c$. Thus, $l_1\{\sigma\}_l = {l_2}^c\{\sigma\}_l$.

We now construct the resolution rule from the simple resolution rule and substitution:

$$\frac{\dfrac{C_1}{C_1\{\sigma\}_{ls}} \qquad \dfrac{C_2}{C_2\{\sigma\}_{ls}} \qquad \begin{array}{l} l_1\{\sigma\}_l \in C_1\{\sigma\}_{ls} \\ l_2\{\sigma\}_l \in C_2\{\sigma\}_{ls} \\ l_1\{\sigma\}_l = (l_2\{\sigma\}_l)^c \end{array}}{(C_1\{\sigma\}_{ls} - \{l_1\{\sigma\}_l\}) \cup (C_2\{\sigma\}_{ls} - \{l_2\{\sigma\}_l\})}}{(C_1\{\sigma\}_{ls} - L_1\{\sigma\}_{ls}) \cup (C_2\{\sigma\}_{ls} - L_2\{\sigma\}_{ls})}$$

We constructed the proof using substitutions first, to get from C_1 to $C_1\{\sigma\}_{ls}$ and from C_2 to $C_2\{\sigma\}_{ls}$. Then we used the simple resolution rule to derive $(C_1\{\sigma\}_{ls} - \{l_1\{\sigma\}_l\}) \cup (C_2\{\sigma\}_{ls} - \{l_2\{\sigma\}_l\})$ as we had already proven the side conditions for this instance of the rule. In the last derivation step we used the equations $L_1\{\sigma\}_{ls} = \{l_1\{\sigma\}_l\}$ and $L_2\{\sigma\}_{ls} = \{l_2\{\sigma\}_l\}$. Since we could build the resolution rule from two sound rules, it must be sound by construction. Q. E. D.

In the formalization of the proof, we also use the applicability to prove the side conditions of the simple resolution rule. However, we do not have an explicit construction of the resolution rule, from the two other rules, but rather use *inst-sound* and *resolution-sound* directly and follow the same steps as used in the construction.

5.2.4 Applicability

By inspecting the soundness proof, we notice that we did not use all the side conditions. We did not use that C_1 and C_2 have no variables in common. Also, we did not use the fact that σ is a most general unifier for $L_1 \cup {L_2}^c$, only that it is a unifier. Therefore, one can wonder why the side conditions are there at all. After all, they could only make it more difficult to prove completeness, since they exclude possibly fruitful rule applications. However, by excluding these applications, we can make our search for the empty clause more directed. If we can still prove completeness we have not really lost anything.

5.3 Soundness of Resolution Derivations

We now show that a step in a resolution derivation is sound. There are two cases to consider, namely the one where we apply the resolution rule, and the one where we rename variables to standardize clauses apart. We have already proven the resolution rule sound so this case follows easily. When we standardize apart we apply a substitution, and since we proved substitution sound this case also follows easily.

lemma *sound-step*: *resolution-step Cs Cs'* $\Longrightarrow$ *evalcs F G Cs* $\Longrightarrow$ *evalcs F G Cs'*

We can extend the result to a whole derivation. The proof is by induction. Isabelle provides an induction rule called *star.induct* for the *star* function that we used to define derivations.

lemma *sound-derivation*:
resolution-deriv Cs Cs' $\Longrightarrow$ *evalcs F G Cs* $\Longrightarrow$ *evalcs F G Cs'*

Chapter 6

Formalization: Completeness

This chapter formalizes the completeness of the resolution calculus. The chapter uses the semantic tree approach from chapter 3. Therefore, it consists of a formal definition of Herbrand terms, enumerations, and semantic trees. Hereafter it presents formal proofs of two major steps towards completeness, namely König's lemma, and Herbrand's theorem. It also shows the formal statements of the lifting lemma and the completeness theorem, and proves the completeness theorem thoroughly albeit informally.

6.1 Herbrand Terms

We now formalize Herbrand interpretions. These interpretations have two important properties. The first is that their universe consists of herbrand terms. The type of Herbrand terms is similar to that of first-order terms except that it does not contain variables:

datatype *hterm* = *HFun fun-sym hterm list*

Therefore, Herbrand terms correspond to ground terms. We call atoms Herbrand atoms, if they consist of Herbrand terms, i.e. are literals of type *hterm literal*. These atoms correspond to ground atoms.

We illustrate the correspondence between ground terms and Herbrand terms by introducing functions that convert between *hterm*s that are ground, and *hterm*s:

```
primrec fterm-of-hterm :: hterm ⇒ fterm
and fterms-of-hterms :: hterm list ⇒ fterm list where
  fterm-of-hterm (HFun p ts) = Fun p (fterms-of-hterms ts)
| fterms-of-hterms [] = []
| fterms-of-hterms (t#ts) = fterm-of-hterm t # fterms-of-hterms ts
```

```
primrec hterm-of-fterm :: fterm ⇒ hterm
and hterms-of-fterms :: fterm list ⇒ hterm list where
  hterm-of-fterm (Fun p ts) = HFun p (hterms-of-fterms ts)
| hterms-of-fterms [] = []
| hterms-of-fterms (t#ts) = hterm-of-fterm t # hterms-of-fterms ts
```

The functions are defined using mutual recursion. They only use primitive recursion which is why we can define them using **primrec**, but we could also have used **fun**.

The second important property of herbrand interpretations is that their function denotation is the constructor *HFun*. We call it the Herbrand function denotation. The constructor *HFun* has type *fun-sym* ⇒ *hterm list* ⇒ *hterm* which is the same as *hterm fun-denot*, and hence its universe is indeed the herbrand universe. It is a natural function denotation to define, since it simply packs up its input in an *hterm*. When we evaluate a ground term, we really just apply the function denotation recursively to a term, and therefore, evaluating a ground term under *HFun* constructs the conversion of the term to an *hterm*.

We can prove this formally in Isabelle by induction:

```
lemma eval-ground: ground t ⟹ (evalt E HFun t) = hterm-of-fterm t grounds ts
⟹ (evalts E HFun ts) = hterms-of-fterms ts
```

6.2 Enumerations

To put labels on the different levels of the semantic tree, we need an enumeration of Herbrand atoms, that is, an infinite sequence $A_0, A_1, A_2, \ldots$ of ground atoms, that contains all herbrand atoms.

Two types in Isabelle are very well suited for representing enumerations. One is *hterm literal stream*, and the other is *nat* ⇒ *hterm literal*.

The *hterm stream* is a type that represents infinite lists of hterms. Like *list*, it comes with a cons operation ##, a head function *shd* and a tail function *stl*. The *stream* library comes with functions that can be used to build up streams from other streams. Therefore, we could first build a stream of strings, then use that to build a stream of terms, and lastly use that to build a stream of atoms. This is an easy way to build an enumeration from the ground up.

The function type *nat* ⇒ *hterm* does not have the nice functions to build up streams from each other. However, an enumeration of Herbrand terms has already been formalized by Berghofer [Ber07]. He does it by defining a *nat* ⇒ *hterm* function called *hterm-diag* whose domain consists of all *hterm*s. Therefore, we choose to adapt his enumeration and extend it to enumerations of Herbrand atoms, and ground atoms.

We present our enumeration of herbrand terms:

```
definition diag-hatom :: nat ⇒ hterm literal where
  diag-hatom a ≡
    (let (p,ts) = diag a in
      (Pos (diag-string p) (diag-list diag-hterm ts))
    )
```

We do not go in to details, but notice that it combines three other enumerations: *diag*, *diag-string*, and *diag-list*.

We also present the inverse function, which given an atom, returns the corresponding number:

```
definition undiag-hatom :: hterm literal ⇒ nat where
   undiag-hatom a ≡ undiag (undiag-string (get-pred a), undiag-list undiag-hterm
(get-terms a))
```

Lastly, we present the corresponding functions for ground atoms. We reuse the above functions by converting the ground atoms to herbrand atoms:

definition *diag-fatom* :: *nat* ⇒ *fterm literal* **where**
  diag-fatom n = *fatom-of-hatom* (*diag-hatom n*)

definition *undiag-fatom* :: *fterm literal* ⇒ *nat* **where**
  undiag-fatom t = *undiag-hatom* (*hatom-of-fatom t*)

We notice that the enumeration does not consider whether the literal is positive or negative. This means that we can call *undiag-hatom* on a negative literal and get the number of the complement atom.

The definitions of all the enumerations and all the lemmas about them are included in appendix A.

6.3 Semantic Trees and Partial Interpretations

The core of our completeness proof is to take a semantic tree and to cut it down to the root by looking at the partial interpretations it contains. We therefore also need to represent partial interpretations. In the literature, semantic trees are usually represented as labeled binary trees. In a well-formed semantic tree, the label of the left children on the *i*'th level is *diag-fatom i*, and the label of the right children on the *i*'th level is *complementary* (*diag-fatom i*). A path in the tree can then be converted to a partial interpretation by collecting the labels on the path in a set of literals that represents the partial predicate denotation. We presented these definitions in chapter 3, and they could be implemented directly in Isabelle.

We choose another approach that illustrates the correspondence between paths and partial interpretations more clearly. We instead represent the semantic trees as unlabeled binary trees.

datatype *tree* =
  Leaf
| *Branch* (*ltree*: *tree*) (*rtree*: *tree*)

We do not need the labels, since we can always calculate them from the position of an unlabeled node. This representation has the advantage that we do not

need to worry about well-formedness, since the label on the nodes only exist implicitly. This removes junk trees that are not semantic such as the one in which all labels are *Atom "p"* [].

Paths in a binary tree are modeled as lists of elements of the type *dir*.

datatype *dir* = *Left* | *Right*

A path corresponds to the process of walking from the root, by iterating over the list and walking left or right in the binary tree.

For representing partial predicate denotations, we choose the type *bool list* where *True* on the *i*'th position means that *diag-fatom i* is true in the partial predicate denotation and *False* means that *diag-fatom i* is false. In other words, the partial predicate denotation assigns truth-values to the ground atoms.

type-synonym *partial-pred-denot* = *bool list*

Instead of using defining *dir* as the above datatype we can define *Left* and *Right* as synonyms of *True* and *False*. Then we see that the paths in the binary tree *are* actually partial predicate denotations.

type-synonym *dir* = *bool*
definition *Left* :: *bool* **where** *Left* = *True*
definition *Right* :: *bool* **where** *Right* = *False*

This gives us a simple definition of paths and this definition also represents partial interpretations. The equality of paths and partial interpretations serves as a pedagogical point and gives us the advantage that we do not need lemmas that describe the correspondence between these concepts.

We choose to represent trees as a datatype. This means that we have defined the trees as being of finite size. However, semantic trees can also be infinite. In Isabelle we can easily expand the definition to infinite trees by using instead a codatatype:

codatatype *tree* =
  Leaf
| *Branch* (*ltree*: *tree*) (*rtree*: *tree*)

This codatatype represents both finite and infinite trees, in other words, possibly infinite trees. The *stream* type is another example of a codatatype. The codatatypes are a recent addition to Isabelle. Proving properties about codatatypes is a skill in itself that I assess would take me considerable time to understand in depth and learn how to use. Therefore, we chose another approach.

We represent finite trees by the tree datatype, and possibly infinite trees as the type *dir list set*. A *dir list set* represents a tree by consisting of all the paths that are in the tree. For a tree to be well-formed, any prefix of a path in the tree must also be a path in the tree. We formalize this property:

abbreviation *wf-tree* :: *dir list set* ⇒ *bool* **where**
wf-tree $T \equiv (\forall\, ds\ d.\ (ds\ @\ d) \in T \longrightarrow ds \in T)$

We likewise need a representation for infinite paths in the trees. This is another chance to use *stream*s. Instead we choose to use functions of type *nat* ⇒ *dir list*. The idea is that can represent an infinite path by a function f, where $f\ n$ gives of the prefix of the path with length n. We call such a function a list chain, and define it formally:

abbreviation *list-chain* :: (*nat* ⇒ *'a list*) ⇒ *bool* **where**
list-chain $f \equiv (f\ 0 = []) \wedge (\forall\, n.\ \exists\, a.\ f\ (Suc\ n) = (f\ n)\ @\ [a])$

We will often consider subtrees rooted in some node of a tree. We define this concept:

fun *subtree* :: *dir list set* ⇒ *dir list* ⇒ *dir list set* **where**
subtree $T\ r = \{ds \in T.\ \exists\, ds'.\ ds = r\ @\ ds'\}$

The definition takes the path to a node, and then returns the set of paths that pass through this node, i.e. have it as a prefix.

6.4 König's Lemma

Our first step in cutting down semantic trees is König's lemma. The lemma states that any infinite binary tree contains an infinite path. The contrapositive

of this is that a binary tree in which all paths are finite is also finite. Therefore, we can use the theorem to cut trees down to finite size.

We first prove that one of the subtrees T_1 and T_2 of any infinite tree T is also infinite.

lemma *inf-subs*:
assumes *inf*: ¬*finite*(*subtree T ds*)
shows ¬*finite*(*subtree T* (*ds* @ [*Left*])) ∨ ¬*finite*(*subtree T* (*ds* @ [*Right*]))

PROOF. We prove it in its contrapositive formulation: If both the subtrees T_1 and T_2 of a tree T are finite then so is T.

The theorem is now obvious, since T consists of the finite number of notes in T_1 plus the finite number of notes in T_2 plus the root note of T. This is clearly finite. Q. E. D.

We now present König's lemma and its proof in natural language.

Theorem 6.1 *Any infinite tree T contains an infinite path.*

PROOF. We construct the path as follows. Since T is finite, we know from the previous lemma that either the left subtree T_1 or the right subtree T_2 is infinite. If T_1 is infinite, the first step of the infinite path is *Left* and then we continue this process on T_1 to get the following steps. If T_1 is not infinite, then it was T_2 which was infinite and then we let *Right* be the first step in the path and then continue this process on T_2 to get the following steps in infinite the path. Q. E. D.

The formalization of the lemma is

lemma *konig*:
assumes *inf*: ¬*finite T*
assumes *wellformed*: *wf-tree T*
shows ∃ *c*. *list-chain c* ∧ (∀ *n*. (*c n*) ∈ *T*)

The assumptions state that T is an infinite and well-formed tree. The thesis is that there is some list-chain, all of whose prefixes are in T, i.e., there is an infinite path in T.

To formalize this proof we need a way to pick the infinite subtrees. For this purpose, we define the function *?nextnode*. We can call it on an node that is the root of an infinite subtree, and then it returns the path to the left subtree if it is infinite and otherwise the path to the right subtree, which must then be infinite.

```
let ?nextnode =
  λds. (if ¬finite(subtree T (ds @ [Left])) then ds @ [Left] else ds @ [Right])
```

The function is defined inside the proof as a schematic variable. This is somewhat similar to the abbreviations, since it just defines *?nextnode* as a syntactical shorthand for its definition.

We also define the function *buildchain*. Given a natural number n and a function *next* of type *dir list* ⇒ *dir list* it starts from [] and then applies *next* to it n times.

```
fun buildchain :: (dir list ⇒ dir list) ⇒ nat ⇒ dir list where
  buildchain next 0 = []
| buildchain next (Suc n) = next (buildchain next n)
```

This is enough for us to construct our list-chain:

```
let ?c = buildchain ?nextnode
```

Of course, we need to show that it is actually a list-chain. This can be shown with *auto*. It is also easy to realize, since *?nextnode* always appends an element to the end of the previous list.

We also need to show that for any n, the path ?c n is in T. We prove a generalization of this by induction on the natural number n. The generalization is that ?c n is in the tree, *and* that the subtree rooted in ?c n is infinite. We need to make this generalization because *?nextnode* only gives us a new infinite subtree, if we call it on the root of an infinite subtree. Therefore, in our induction, we need to carry this information in our induction hypothesis.

It is easy to prove for the base case of 0. Since T is infinite, it must contain some path *ds*, and since the tree is well-formed, it must also contain the prefix

[], but [] is actually $?c\ 0$. The subtree rooted in $?c\ 0 = []$ is actually the whole tree, and it is infinite by assumption.

We also prove the induction case, i.e. prove the theorem for $n+1$. The induction hypothesis is that that $?c\ n$ is in T, *and* that the subtree rooted in $?c\ n$ is infinite. We must prove the same property for $n + 1$. By the induction hypothesis, the subtree rooted in $?c\ n$ is infinite. Therefore we can apply *?nextnode* to get a new infinite subtree *?nextnode* $(?c\ n)$. We recall that by definition of $?c$, and of *buildchain*, we know that $?c\ n$ consists of n applications of *?nextnode*. Thus, *?nextnode* $(?c\ n)$ must be $n+1$ applications of *?nextnode*, i.e., $?c\ (n+1)$. So the subtree roted in $?c\ (n+1)$ is infinite. This also means that there must be some path *ds* in the subtree, and $?c\ (n+1)$ must surely be a prefix of this. Therefore, since the T well-formed, $?c\ (n+1)$ must also be in T.

This concludes our formalization of König's lemma.

6.5 Semantics of Partial Predicate Denotations

Let us introduce the semantics for partial predicate denotations. We first introduce their semantics for literals. Our semantics is optimistic with regard to satisfying literals. If the partial predicate denotation does not contain enough information about a literal to actively falsify it, then it does not falsify it, but satisfies it.

A partial predicate denotation assigns truth-values to ground atoms. It falsifies an atom if it assigns *False* to an instance of the atom. Likewise, it falsifies a negative literal if it assigns *True* to an instance of the complementary atom.

```
fun falsifiesl :: partial-pred-denot ⇒ fterm literal ⇒ bool where
  falsifiesl G (Pos p ts) =
     (∃ i ts'.
      i < length G
      ∧ G ! i = False
      ∧ diag-fatom i = Pos p ts'
      ∧ instance-ofts ts' ts)
| falsifiesl G (Neg p ts) =
     (∃ i ts'.
      i < length G
      ∧ G ! i = True
      ∧ diag-fatom i = Pos p ts'
      ∧ instance-ofts ts' ts)
```

For a positive literal *Pos p ts*, it is done by looking if there is an entry in G containing an atom that is an instance of the literal, and is assigned to *False*. The definition for negative literals mirrors this.

A partial predicate denotation G is said to satisfy a literal if it does not falsify it. In other words, an atom is satisfied if all instances of it in G are assigned to *True*. Likewise, a negative literal is satisfied if all instances of it in G are assigned to *False*.

A partial predicate denotation falsifies a clause by falsifying some literal in the clause, and it falsifies a clausal form by falsifying some clause in the clausal form:

abbreviation *falsifiesc* :: *partial-pred-denot* ⇒ *fterm clause* ⇒ *bool* **where**
falsifiesc G C ≡ (∀ *l* ∈ *C*. *falsifiesl G l*)

abbreviation *falsifiescs* :: *partial-pred-denot* ⇒ *fterm clause set* ⇒ *bool* **where**
falsifiescs G Cs ≡ (∃ *C* ∈ *Cs*. *falsifiesc G C*)

We can use this to formalize closed and open branches. To do that, we must firstly formalize branches.

inductive *branch* :: *dir list* ⇒ *tree* ⇒ *bool* **where**
branch [] *Leaf*
| *branch ds l* ⟹ *branch* (*Left* # *ds*) (*Branch l r*)
| *branch ds r* ⟹ *branch* (*Right* # *ds*) (*Branch l r*)

The definition says that the empty list is a subtree of a leaf, and that if the tail of a list is a branch of an immediate left subtree, then if we add *Left* to the beginning we have a branch of the whole tree, and similarly for the right subtree. Since it is inductive, it also states that these are the only branches.

We have again used an inductive predicate. A closed branch for a clausal form is a branch that falsifies the clausal form. An open branch is a branch that does not.

abbreviation *closed-branch* :: *partial-pred-denot* ⇒ *tree* ⇒ *fterm clause set* ⇒ *bool* **where**
closed-branch G T Cs ≡ *branch G T* ∧ *falsifiescs G Cs*

abbreviation *open-branch* :: *partial-pred-denot* ⇒ *tree* ⇒ *fterm clause set* ⇒ *bool*
where
open-branch G T Cs ≡ *branch G T* ∧ ¬*falsifiescs G Cs*

A closed tree is a finite tree in which the branches and only the branches are closed:

fun *closed-tree* :: *tree* ⇒ *fterm clause set* ⇒ *bool* **where**
closed-tree T Cs ⟷ *anybranch T* (λ*b*. *closed-branch b T Cs*)
∧ *anyinternal T* (λ*p*. ¬*falsifiescs p Cs*)

6.6 Herbrand's Theorem

Herbrand's theorem comes in several versions [CL73]. We prove the following version:

Theorem 6.2 *If a clausal form Cs is unsatisfiable then there exists a finite and closed semantic tree for Cs.*

We first prove the theorem in the following slightly different and contrapositive formulation:

Lemma 6.3 *If all finite semantic trees of a clausal form Cs have an open branch, then Cs has a model.*

6.6.1 Building a Model

We will use König's lemma to find an infinite path G, all of whose prefixes are partial predicate denotations that satisfy *Cs*. The infinite path G can be seen as an infinite partial predicate denotation. This is the same as a predicate denotation because it assigns a truth value to any predicate symbol and list of elements of the Herbrand universe.

Before we find it, we show that it also satisfies *Cs*. It might seem obvious, but it does require substantial work that illustrates the interplay of syntax and semantics similarly to our soundness proof.

Lemma 6.4 *If G is an infinite path and all its prefixes satisfy a clausal form Cs, then G itself also satisfies Cs.*

Proof. We fix an arbitrary variable denotation E and a clause C in Cs, to show that any clause in Cs is true under any variable denotation.

Since our universe is Herbrand terms, the variable denotation E has type *var-sym* $\Rightarrow$ *hterm*. Therefore, we can easily create a function *sub-of-denot* that turns it into a substitution σ, by converting its domain from *hterms* to ground terms. This mirrors our soundness proof, where we converted substitution to variable denotations.

fun *sub-of-denot* :: *hterm var-denot* $\Rightarrow$ *substitution* **where**
sub-of-denot E = *fterm-of-hterm* $\circ$ E

Using the substitution σ on our clause, we get a new clause $C\{\sigma\}_{ls}$ with the following properties:

a. $C\{\sigma\}_{ls}$ is ground. This is because the domain of σ consists of ground terms.

b. Under E and *HFun*, both C and $C\{\sigma\}_{ls}$ are satisfied and falsified by the same predicate denotations. The reason is this: We consider a variable x that occurs in C. The variable will be evaluated to an Herbrand term h. If we look at $C\{\sigma\}_{ls}$, the variable is first replaced by h's conversion to a term. Then this term is evaluated by *HFun* and thus converts to h again. So in both cases x evaluates to h.

c. If $C\{\sigma\}_{ls}$ is falsified by a partial predicate denotation G', then so is C. That a literal $l\{\sigma\}_l$ in $C\{\sigma\}_{ls}$ is falsified by G', means that G' falsifies some ground instance of $l\{\sigma\}_l$. This ground instance is also an instance of l, and so, also l in C is also falsified.

d. If C is satisfied by a partial predicate denotation, then so is $C\{\sigma\}_{ls}$. This is the contrapositive of item c.

Thus, by item b, to prove that C is satisfied by G, it suffices to show that G satisfies $C\{\sigma\}_{ls}$. Take a prefix G' of the full predicate denotation G, such that G' is large enough that it covers all literals in $C\{\sigma\}_{ls}$. It satisfies C (by our assumption) and thus also $C\{\sigma\}_{ls}$ (by item d). So must G since it is an extension of G', and extending it only adds assignments of *True* and *False* to irrelevant literals that are not in $C\{\sigma\}_{ls}$. Q. E. D.

In the above proof, we looked at an infinite path G that we had gotten from König's lemma, and argued that it was a predicate denotation. But we remember that our formalization of König's lemma actually gives us a *list-chain* of paths.

Therefore, we make a function that converts a list-chain to a predicate denotation.

```
abbreviation extend :: (nat ⇒ partial-pred-denot) ⇒ hterm pred-denot where
  extend f P ts ≡ (
    let n = undiag-hatom (Pos P ts) in
      f (Suc n) ! n
    )
```

We call the function extend, because *extend f* can be seen as an extension of any *f n*. When we give a predicate symbol *p* and an hterm list *ts* to *extend f*, then it finds the number *n* of the atom *Pos p ts*, and then it finds a prefix *f* (*Suc n*) in that is long enough to contain *n*. Then it looks up whether the *n'th* entry assigns *Pos p ts* to *True* or *False*.

With this cleared up we present our formalization of this lemma:

```
lemma extend-preserves-model:
  assumes f-chain: list-chain (f :: nat ⇒ partial-pred-denot)
  assumes n-max: ∀ l∈C. undiag-fatom l ≤ n
  assumes C-ground: groundls C
  assumes C-false: ¬evalc HFun (extend f) C
  shows falsifiesc (f (Suc n)) C
```

6.6.2 Proving Herbrand's Theorem

We are now ready to prove Herbrand's theorem by finding our infinite path.

Proof. We build a semantic tree T from a clausal form *Cs*. It consists of all paths that, when seen as a partial predicate denotations, satisfiy *Cs*.

We now show that this tree is infinite by generating infinitely many different paths that are contained in this tree: We consider some other semantic tree that is finite. From our assumption we know that this tree has an open branch. This is true no matter what the depth of that tree is. So we can build a semantic tree of depth 1, to get an open branch of length 1. Since it is open, it does not falsify any ground instance of any clause. Therefore this path is also a path in

T. We can do the same for a semantic tree of depth 2, and one of depth 3, and so on. Thus the tree is infinite.

Now we can use König's lemma to obtain an infinite path G in T. All its prefixes are satisfying partial predicate denotation, and we already showed that this implies that G also satisfies C. Q. E. D.

We formalize the lemma and its proof in Isabelle.

theorem *herbrand'*:
  assumes *openb*: $\forall\, T.\ \exists\, G.$ *open-branch G T Cs*
  assumes *finite-cs*: *finite Cs* $\forall\, C \in Cs.$ *finite C*
  shows $\exists\, G.$ *evalcs HFun G Cs*

This was our second formulation of Herbrand's lemma. We would also like to prove the first one. We first take the contra-positive of *herbrand'*:

theorem *herbrand'-contra*:
  assumes *finite-cs*: *finite Cs* $\forall\, C \in Cs.$ *finite C*
  assumes *unsat*: $\forall\, G.\ \neg$*evalcs HFun G Cs*
  shows $\exists\, T.\ \forall\, G.$ *branch G T* $\longrightarrow$ *closed-branch G T Cs*

We see that we have a tree in which all the branches are closed. However, we wanted exactly the branches to be closed. We can obtain such a tree by cutting branches off as soon as they are long enough to falsify a clause.

This cutting of branches is formalized by introducing tree operations that can remove nodes, and proving a quite substantial amount of theorems about them. They are not presented here because they do not give much logical insight. Instead we present the formalization of Herbrand's lemma and refer to appendices B and C.

theorem *herbrand*:
  assumes *unsat*: $\forall\, G.\ \neg$ *evalcs HFun G Cs*
  assumes *finite-cs*: *finite Cs* $\forall\, C \in Cs.$ *finite C*
  shows $\exists\, T.$ *closed-tree T Cs*

The proof simply combines *herbrand'-contra* with the tree operations.

6.7 Lifting Lemma

Our next step is to prove the lifting lemma. We first state it:

lemma *lifting*:
  assumes *appart*: $varsc\ c \cap varsc\ d = \{\}$
  assumes $inst_1$: $instance\text{-}ofc\ c'\ c$
  assumes $inst_2$: $instance\text{-}ofc\ d'\ d$
  assumes *appl*: $applicable\ c'\ d'\ l'\ m'\ \sigma$
  shows $\exists\, l\ m\ \tau.\ applicable\ c\ d\ l\ m\ \tau \wedge$
    $instance\text{-}ofc\ (resolution\ c'\ d'\ l'\ m'\ \sigma)\ (resolution\ c\ d\ l\ m\ \tau)$

This thesis does not contain a formalized proof of this lemma, because it turned out to be much more challenging than expected. The main reason was that several proofs of the lemma from the literature were flawed. In chapter 8, we will discuss this challenge, as well as possibilities for overcoming it. For now, we turn to the completeness proof.

6.8 Completeness

Assuming a proof of the lifting lemma, we have all the theory necessary to prove the resolution calculus complete. We state the completeness theorem:

theorem *completeness*:
  assumes *finite-cs*: $finite\ Cs \wedge C \in Cs.\ finite\ C$
  assumes *unsat*: $\forall F\ G.\ \neg evalcs\ F\ G\ Cs$
  shows $\exists\, Cs'.\ resolution\text{-}deriv\ Cs\ Cs' \wedge \{\} \in Cs'$

I used quite a lot of time trying to make a formalized proof of the lifting lemma. Therefore, I did not have time to also formalize a proof of the completeness itself. Furthermore, the correctness of such a proof would depend entirely on the correctness of the lifting lemma. We will therefore instead look at an informal, but thorough, proof of the theorem, which can form the basis of its formalization. The proof follows the one from chapter 3, but uses the definitions of the Isabelle formalization.

Proof. Since our clausal form Cs is unsatisfiable, it must by Herbrand's lemma have a finite closed semantic tree. Our proof is by induction on the size of this

tree T and for arbitrary clausal forms. We therefore restate the completeness lemma to assume the existence of the finite closed semantic tree, such that it will be part of our induction hypothesis.

theorem *completeness'*:
assumes *finite-cs*: *finite Cs* $\forall$ *C*$\in$*Cs*. *finite C*
shows *closed-tree T Cs* $\Longrightarrow$ $\exists$ *Cs'*. *resolution-deriv Cs Cs'* $\wedge$ {} $\in$ *Cs'*

If the tree has size 1, then it must consist only of a leaf node. The empty list [] is a branch in this tree. By definition of closed semantic trees, we know that there must be some clause $C \in Cs$ that [] falsifies, i.e. it has to falsify all of the literals in C. The only clause that [] can falsify is {} because any other clause contains a literal that [] does not falsify. Thus {} $\in Cs$ and we have hence derived the empty clause.

We also prove it for the induction case of trees of size $n > 1$. Our induction hypothesis is that we can derive the empty clause from any clausal form Cs' with a closed semantic tree T' of size smaller than T.

We must prove that we can also derive the empty clause from Cs using our semantic tree T. We find a node N in T, whose children N_1 and N_2 are leafs. The existence of this node can be proven by induction on trees. Now, we consider the path B from the root to N, the branch B_1 from the root to N_1, and the branch B_2 from the root to N_2. By the definition of semantic trees, we know that $B_1 = B$ @ [*True*] and $B_2 = B$ @ [*False*]. Furthermore, we know that there exists some clause C_1 that has a ground instance C_1' that is falsified by B_1. Likewise there is a C_2 and a C_2' corresponding to B_2. Since C_1' is falsified by B @ [*True*], all the literals in C_1' must be falsified by this list. But they are not all falsified by B, since it is not a branch node. Therefore there must be a literal $l_1 \in C_1$ that is falsified by B @ [*True*], but not by B. This literal must have number $|B| + 1$ in our enumeration, since it is the only one that is falsified by B @ [*True*] but not B. It must also be negative. Additionally, all the other literals in C_1' must be falsified by B, since they are falsified by B_1, but not l_1'. Likewise C_2' must contain l_2' which is the positive literal with number $|B|+1$. Therefore C_1' and C_2' have a clashing pair of literals, and we can perform resolution to get a resolvent C'. The resolvent C' must be falsified by B since resolving C_1' and C_2' removes the only two of their literals that were not falsified by B. By the lifting lemma, we can obtain the resolvent C of C_1 and C_2 with instance C'. We create a new tree T' which is T with N_1 and N_2 removed, and a new clause set $Cs' = Cs \cup \{C\}$. Since C' is falsified by the branch B of T', and all other branches of B' are already closed, we can cut T' down to a closed tree for $Cs \cup \{C\}$ using the tree operations for that we used in Herbrand's theorem to cut branches.

By the induction hypothesis we can derive the empty clause from Cs'. This gives us the following derivation of the empty clause from Cs: From Cs we derive $Cs' = Cs \cup \{C\}$ using the resolution rule. Then from that, we derive the empty clause. Q. E. D.

Chapter 7

Examples

In this section we show some formalizations of resolution derivations in our resolution calculus. We start with the derivation we presented in chapter 2. This derivation works on ground clauses. Therefore, the complementary sets of literals each contain one element. This means that any substitution is a unifier, including the empty substitution. Furthermore, the empty substitution is an mgu, since we can construct any substitution from it.

theorem *unifier-single*: *unifierls* σ $\{l\}$

theorem *empty-mgu*: *unifierls* ε $L \Longrightarrow$ *mguls* ε L

We define the literals of the clauses.

definition *PP* :: *fterm literal* **where**
PP = *Pos* ''*P*'' [*Fun* ''*c*'' []]

definition *PQ* :: *fterm literal* **where**
PQ = *Pos* ''*Q*'' [*Fun* ''*d*'' []]

definition *NP* :: *fterm literal* **where**

NP = *Neg* ''*P*'' [*Fun* ''*c*'' []]

definition *NQ* :: *fterm literal* **where**
NQ = *Neg* ''*Q*'' [*Fun* ''*d*'' []]

The first letter, P or N indicates if the literal is positive or negative, and the next letter P or Q, indicates if the predicate symbol is P or Q. We present the formalization of our derivation:

lemma *resolution-example1*:
∃ *Cs*. *resolution-deriv* {{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*}}
{{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*},{*NP*},{*PP*},{}}

We see that the empty clause can indeed be derived. I have not included the proof, but it essentially consists of spelling out each derivation step as a proof statement in Isabelle. It is included in appendix D. The full derivation is then proven using the simplification rules of *resolution-deriv*.

We can also prove the example we looked at in chapter 3 as an example of how the completeness proof works. Again we define the literals first:

definition *Pa* :: *fterm literal* **where**
Pa = *Pos* ''*a*'' []

definition *Na* :: *fterm literal* **where**
Na = *Neg* ''*a*'' []

definition *Pb* :: *fterm literal* **where**
Pb = *Pos* ''*b*'' []

definition *Nb* :: *fterm literal* **where**
Nb = *Neg* ''*b*'' []

definition *Paa* :: *fterm literal* **where**
Paa = *Pos* ''*a*'' [*Fun* ''*a*'' []]

definition *Naa* :: *fterm literal* **where**
Naa = *Neg* ''*a*'' [*Fun* ''*a*'' []]

definition *Pax* :: *fterm literal* **where**
Pax = *Pos* ''*a*'' [*Var* ''*x*'']

definition *Nax* :: *fterm literal* **where**
Nax = *Neg* ''*a*'' [*Var* ''*x*'']

We also need a most general unifier for the terms $a(a())$ and $a(x)$:

definition *mguPaaPax* :: *substitution* **where**
mguPaaPax = (λx. *if* x = ''x'' *then Fun* ''a'' [] *else Var* x)

lemma *mguPaaPax-mgu*: *mguls mguPaaPax* {*Paa*,*Pax*}

It takes considerable work to prove that it is an mgu. For the formalized proof, we refer to appendix C. With a formalization of a unification algorithm, we could just calculate an mgu instead of explicitly proving its existence.

We can now perform the derivation:

lemma *resolution-example2*:
$\exists$ *Cs*. *resolution-deriv* {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}
{{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*},{*Na*,*Pb*},{*Na*},{}}

Finally, using our soundness lemma, we show that since we can derive the empty clause in the examples, the clausal forms are indeed unsatisfiable. We first prove that if we can derive the empty clause from a clausal form, then it is satisfiable:

lemma *ref-sound*:
assumes *deriv*: *resolution-deriv Cs Cs'* $\wedge$ {} $\in$ *Cs'*
shows $\neg$*evalcs F G Cs*

We then apply this theorem to our examples:

lemma *resolution-example1-sem*: $\neg$*evalcs F G* {{*NP*, *PQ*}, {*NQ*}, {*PP*, *PQ*}}

lemma *resolution-example2-sem*: $\neg$*evalcs F G* {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}

Chapter 8

Discussion

This chapter discusses the formalization of the resolution calculus.

- It discusses proofs of the lifting lemma from the literature.
- It suggests solutions obtaining a formalized proof of the lifting lemma.
- It discusses the relation the formalization of the resolution calculus to automated theorem provers.
- It discusses the lessons I have learned during the formalization process.
- It discusses the thesis itself.

8.1 Proving the Lifting Lemma

The formalization does not yet have a formal proof of completeness. To obtain a proof, there are two hurdles to overcome. First, we must prove the lifting lemma, and then we must use it to prove completeness. In chapter 6, we already had a thorough proof of completeness that could be formalized, and therefore, the larger challenge is formalizing a proof of the lifting lemma.

Formalizing a proof of the lifting lemma turned out to be more of a challenge than I expected. The reason is that the presentations I studied turned out not to be well suited for a formalization, because they had some flaws that could not easily be resolved. In this section, we will study a proof from the literature. We observe some flaws and discuss how to resolve them.

8.1.1 A Proof From the Literature

We will study a proof of the lifting lemma from Ben-Ari's book *Mathematical Logic for Computer Science* [BA12]. We will see why the proof can not easily be formalized, and describe some flaws and imprecisions. The notation and definitions are similar to ours, except that substitutions are sets of key-value pairs writen $k \leftarrow v$, where k is a key and v a value. Thus they can be combined with the $\cup$ operator. A substitution is well-formed if it is finite and maps each key to exactly one value. The notation for applying substitutions is $l\sigma$ and that for composition is $\sigma_1\sigma_2$

Theorem 10.33 (Lifting Lemma) *Let C_1', C_2' be ground instances of C_1, C_2, respectively. Let C' be a ground resolvent of C_1' and C_2'. Then there is a resolvent C of C_1 and C_2 such that C' is a ground instance of C.*

Proof First, standardize apart so that the names of the variables in C_1 are different from those in C_2. Let $l \in C_1', l^c \in C'$ be the clashing literals in the ground resolution.

Since C_1' is an instance of C_1 and $l \in C_1'$, there must be a set of literals $L_1 \subseteq C_1$ such that l is an instance of each literal in L_1. Similarly there must be a set of literals $L_2 \subseteq C_2$ such that l^c is an instance of each literal in L_2.

The last paragraph is problematic, since with this formulation we can just let $L_1 = \{\}$ and then it contains nothing that can be resolved. I suggest that we instead obtain a substitution θ such that $C_1\theta = C_1'$ and $C_2\theta = C_2'$. Then we let L_1 be the set of all literals $l_1 \in C_1$ where $l_1\theta = l$, and let L_2 be the set of all literals $l_2 \in C_2$ where $l_2\theta = l$. In this way we catch exactly the clashing sets of literals that are to be removed.

Let λ_1 and λ_2 [be] mgu's for L_1 and L_2, respectively, and let $\lambda = \lambda_1 \cup \lambda_2$. λ is a well-formed substitution since L_1 and L_2 have no variables in common.

This argument strictly does not hold. Here is a counter-example: Let $L_1 = \{c\}$ and $L_2 = \{c\}$. L_1 and L_2 have no variables in common. An mgu for L_1 is $u_1 = \{x \leftarrow y, y \leftarrow x\}$. An mgu for L_2 is $u_2 = \{x \leftarrow z, z \leftarrow x\}$. Then consider the union $u_1 \cup u_2 = \{x \leftarrow y, y \leftarrow x, x \leftarrow z, z \leftarrow x\}$. It is not well formed since x is mapped to both y and z.

I think we can remedy this by requiring that we require λ_1 to not substitute any variable that is not in L_1, and likewise for L_2. In fact, the mgu's produced by unification algorithms have this property. This flaw is minor, but for a formalization even such small flaws are not allowed.

> By construction, $L_1\lambda$ and $L_2\lambda$ are sets which contain a single literal each. These literals have clashing ground instances, so they have a[n] mgu σ.

The argument is presumably that if they have clashing ground instances, then they have a unifier and then they have an mgu. Here is another minor flaw: The two literals $f(x)$ and $f(f(x))$ both have ground instance $f(f(c))$, but they cannot be unified. This can also be remedied by adding the above restrictions on λ_1 and λ_2.

Furthermore, this argument is left to the reader: The reason they have clashing ground instances. Here is the argument: θ unifies L_1 to a ground instance $L_1\theta$, so since λ_1 is mgu: $\theta = \lambda_1 u_1$ for some u_1. Thus $L_1\theta = L_1(\lambda_1 u_1) = (L_1\lambda_1)u_1 = (L_1\lambda)u_1$. Likewise for $L_2\lambda$.

> Since $L_i \subseteq C_i$, we have $L_i\lambda \subseteq C_i\lambda$. Therefore, $C_1\lambda$ and $C_2\lambda$ are clauses that can be made to clash under the mgu σ. It follows that they can be resolved to obtain clause C:
>
> $$C = ((C_1\lambda)\sigma - (L_1\lambda)\sigma) \cup ((C_2\lambda)\sigma - (L_2\lambda)\sigma).$$
>
> By the associativity of substitution (Theorem 10.10):
>
> $$C = (C_1(\lambda\sigma) - L_1(\lambda\sigma)) \cup (C2(\lambda\sigma) - (L2(\lambda\sigma)).$$
>
> C is a resolvent of C_1 and C_2 provided that $\lambda\sigma$ is an mgu of L_1 and L_2^c.
>
> But λ is already reduced to equations of the form $x \leftarrow t$ for distinct variables x and σ is constructed to be an mgu, so $\lambda\sigma$ is a reduced set of equations, all of which are necessary to unify L_1 and L_2^c. Hence $\lambda\sigma$ is an mgu.

The last two sentences can be difficult to understand. For instance, it is not entirely clear what the author means by a reduced set of equations or how we know that they are necessary to unify L_1 and L_2^c. A more thorough treatment is necessary for a formalization.

> Since C_1' and C_2' are ground instances of C_1 and C_2:
>
> $$C_1' = C_1\theta_1 = C_1\lambda\sigma\theta_1' \quad C_2' = C_2\theta_2 = C_2\lambda\sigma\theta_2'$$
>
> for some substitutions θ_1,θ_2,θ_1',θ_2'. Let $\theta' = \theta_1' \cup \theta_2'$.

Substitutions θ_1 and θ_2 are introduced, corresponding to the θ that I suggested we introduced earlier.

We should also argue that θ' is well formed. To do this we need to make some of the same considerations as for when we made λ. We also have to think about how λ and σ were defined.

> Then $C' = C\theta'$ and C' is a ground instance of C.

Here it is in my opinion not clear why $C' = C\theta'$. I have (unsuccessfully) tried to come up with a proof for it. My attempt follows here:

We want to have that $C_1' = C_1\lambda\sigma\theta_1' = C_1\lambda\sigma\theta'$. This should follow from the variable distinctness of θ_1' and θ_2' as well as from the variable distinctness of $C_1\lambda\sigma$ and $C_2\lambda\sigma$. Likewise we argue that $C_2' = C_2\lambda\sigma\theta_2' = C_2\lambda\sigma\theta'$.

Then:

$$\begin{aligned} C' &= (C_1' - \{l\}) \cup (C_2' - \{l^c\}) \\ &= (C_1\lambda\sigma\theta' - L_1\lambda\sigma\theta') \cup (C_2\lambda\sigma\theta' - L_2\lambda\sigma\theta') \\ &= (C_1\lambda\sigma - L_1\lambda\sigma)\theta' \cup (C_2\lambda\sigma\theta' - L_2\lambda\sigma)\theta' \\ &= ((C_1\lambda\sigma - L_1\lambda\sigma) \cup (C_2\lambda\sigma\theta' - L_2\lambda\sigma))\theta' \\ &= C\theta' \end{aligned}$$

To prove this I used that $(A \cup B)\sigma = A\sigma \cup B\sigma$, which holds in general, but also $(A - B)\sigma = A\sigma - B\sigma$ which does *not* hold in general. For instance $(\{Z, X\} - \{X\})\{Z \leftarrow X\} = \{X\}$, but $\{Z, X\}\{Z \leftarrow X\} - \{X\}\{Z \leftarrow X\} = \{\}$. Thus my proof attempt is unsuccessful.

To sum up, the proof has several flaws and imprecisions. They can be summarized in five bullet points:

- L_1 and L_2 are not properly defined.
- It is not enough that C_1 and C_2 are standardized apart. We also need to make sure that many of the substitutions of the proof preserve this property.
- Union of substitutions is not necessarily well formed under the given conditions.
- From the text it is not clear why $\lambda\sigma$ is an mgu of L_1 and L_2^c.
- It is not clear why $C' = C\theta'$.

I have pointed to solutions to these flaws and imprecisions except for the last one, which I am not sure how to address. In appendix E, I repeat the exercise for another proof, and present a counter-example by Leitsch [Lei89] to that proof.

If we want to prove the lifting lemma, then we must either fix the flaws in the proof somehow, or we must come up with a new proof entirely.

8.1.2 Another Resolution Calculus

Another approach is to use a resolution calculus that has a simpler proof of the lifting lemma. One such resolution calculus is presented by Leitsch [Lei97]. We look at its definition using the notation of this thesis. It is an example of binary resolution with factoring, but the binary resolution rule is a little different from the one we presented.

Definition 8.1 (Leitsch's Binary Resolution Rule)

$$\frac{C_1 \qquad C_2}{((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))\{\sigma\}} \quad \begin{array}{l} C_1 \textit{ and } C_2 \textit{ have no variables in common,} \\ l_1 \in C_1, l_2 \in C_2, \\ \sigma \textit{ is an mgu for } \{l_1, l_2^c\} \end{array}$$

The difference from our binary resolution rule is that the literals are removed before the mgu is applied, instead of the other way round. Letisch presents a simple proof of the lifting lemma for this resolution calculus. One of its advantages is that it does not have the problems of the above proof because it avoids the wrong equation $(A - B)\{\sigma\} = A\{\sigma\} - B\{\sigma\}$ completely. Other authors use similar resolution calculi, including Robinson, who invented the resolution calculus.

8.1.3 The Unification Algorithm

Another challenge in formalizing the lifting lemma is that its proof assumes the existence of most general unifiers. That is, if we have a unifier for a set, then we also have a most general unifier for the set. The standard way to prove this theorem is by constructing a unification algorithm and proving it correct. A unification algorithm is an algorithm that explicitly constructs the most general unifier.

A shortcut to obtain this goal would be to adapt the unification algorithm of another formalization such as that of IsaFoR [TS09].

8.1.4 Recommended Approach

We have identified two approaches to proving the lifting lemma. One is to repair the proof for our current definition, or making a new proof. The other is to choose a definition of resolution that removes literals before applying the mgu.

To achieve the goal of obtaining a formalized complete resolution calculus, I recommend the second approach. It has the clear advantage of having a thorough paper proof to follow. The disadvantage is that we need to prove soundness again for the new system. However, soundness proofs are simpler than completeness proofs, and our current proof can hopefully be adapted without too much effort.

Formalizing the lifting lemma for our current definition of resolution could also be an interesting result, because it would confirm that such systems are complete and it would give us a way to repair the proof from Ben-Ari's textbook.

8.2 Formalizing a Logical System in a Logical System

I have proven the resolution system sound in the logical system of Isabelle's HOL. Therefore, one can argue that the soundness of my resolution calculus depends on the soundness on Isabelle. Thus, one could argue, I should also prove Isabelle sound. To do this I would need a stronger proof system than Isabelle since Gödel's incompleteness theorem tells us that a sound proof system

cannot prove itself sound. But then we must again consider the soundness of that stronger logical system. Therefore, one can argue, that we have not really learned much by doing this exercise.

However, in our introduction we looked at a proof system such as Isabelle as a collection of arguments that we agree on are true, and from which we construct our proof. In this view, our formalization is at least as good as an informal proof. An informal proof will prove the soundness using a number of mathematical arguments. Our proof does the same, except that we restrict ourselves to a subset that we agree on beforehand. This also means that we cannot appeal to the sometimes faulty human intuition, and that a computer can check if the proof indeed only uses the argument we agreed on. Giving a theorem a proof that a human can read and understand, and a proof that a computer can check increases our confidence in the theorem.

Furthermore, there is good empirical evidence that Isabelle is sound. It is used by many researchers to formalize many different disciplines of mathematics, computer science, logic, and more. If an unsoundness does occur it can potentially be fixed without affecting the theorems proven thus far. Finally, the logic of proof assistants can also be verified formally [Har06].

8.3 Automatic Theorem Proving

As mentioned in the introduction, the resolution calculus has been implemented successfully in automatic theorem provers. It is worth discussing what the implications of this formalization of the resolution calculus has on these provers.

Firstly, the formalization increases confidence in the background theory of these provers. It also provides through proofs of soundness and completeness, which can give a better understanding of these theorems. However, the best of the automatic theorem provers have additional rules in addition to resolution, such as superposition rules. These rules are not part of this formalization.

Furthermore, the systems are concrete implementations in a programming language of a more abstract proof system. This means that even if the full abstract system is proven sound and complete, this does not necessarily mean that the concrete implementation is sound and complete. One way to overcome this obstacle is to create a verified prover.

Isabelle can generate code in the Standard ML programming language from a formalization. To do this, however, the formalization needs to be specified in a

way that makes it executable. Our formalization is not at that stage, since we, for instance, have not formalized a unification algorithm. The approach has, however, been used to create a verified prover for another prover [Rid04].

8.4 Societal Perspective

As we have seen, this thesis can be seen as a first step towards a verified automatic prover based on resolution. Provers are highly advanced software and their formalization will show that verified software is an obtainable goal. Software plays a very important role in our society, as increasingly many tools and devices contain some kind of software. Today the correctness of software is mostly tested by providing it with input and seeing if it behaves as expected. However, it is almost always impossible to test all possible inputs, which means that there might be some case in which the program does not behave as expected. A good example is the sorting algorithm of the Java library which was thoroughly tested, but still contained a bug [dGRdB+15].

Verifying the software formally means that we logically prove the absence of bugs. Such proofs cover all possible cases, because they explain why the software works on an arbitrary input. This is especially important as software becomes a larger and larger part of our lives. A good example is the verified airplane alerting algorithm [CM00] that we considered in chapter 1, because software bugs can have serious consequences in the case of air transport. Thus, the verification of software is important in itself. However, if proof systems are to play an important role in software verification, it is also important that we gain high confidence in their soundness. This thesis is a step in that direction.

8.5 Lessons Learned

I have learned a lot during the process of formalizing resolution. Firstly, I have learned that it is important to prepare well. During the first phase of the process I studied the literature quite thoroughly and was an advantage when I had to formalize lemmas and theorem, because I understood them well and got to know them without thinking about formalization details. This meant that when I formalized the proofs of König's lemma and Herbrand's theorem, I had already overcome the challenge of understanding the proofs, and thus could focus on formalization.

Another lesson that I have learned is that the formalization process can uncover flaws in proofs. When I initially read the proof of the lifting lemma, I thought I understood it. However, the formalization process made me discover its flaws. This mirrors the experience of others.

Additionally, I have learned that the formalization process can lead to more precise proofs, and help us understand theorems better. A good example of this is my soundness proof. Many of the proofs I read in the literature only glanced over this property. The formalization, however, makes the correspondence between syntax and semantics very clear by introducing the evaluation of substitutions and proving the substitution lemma formally. The separation of syntax and semantics is central to logics, and the formalization makes this very explicit.

8.6 Reflections on the Thesis

During the process of writing this thesis, I have intended that my thesis should be more than an account of my formalization. Therefore, I have an analysis chapter in which I analyze different opportunities for defining the resolution calculus and proving it complete, as well as their merits and weaknesses. I also have this discussion chapter, in which I discuss my results and what further work can be done. Additionally, the formalization chapters also contain analyses, reflections, and discussion about the different choices I had to make in addition to the account of the formalization. There is thus a balance between the different taxonomic levels in the thesis. I could have made this balance more clear by isolating analysis, formalization, and discussions, completely, and in this was have shorter formalization chapters that were purely accounts of the formalization. On the other hand, in the current presentation, the reflections on the formalization occur in the context in which they came about. This has the advantage that it makes this context very clear.

Chapter 9

Conclusions

This chapter concludes on the results of the thesis by determining to which degree the goals of the thesis were achieved. Furthermore, it looks at a number of opportunities for further work based on the thesis.

9.1 Results

The thesis has three goals, namely to make an Isabelle formalization of

1. the resolution calculus
2. its soundness
3. its completeness

The thesis contains a formalization of the syntactic objects of terms, literals, and clauses. From this it formalizes the resolution calculus. Thus, the thesis clearly fulfilled item 1.

The thesis also fulfills item 2, by formally proving the resolution calculus sound. The formal soundness proof contributes with a proof with a clear separation of syntax and semantics, and explicitly shows their interplay.

The thesis partially fulfills item 3. While it does not formally prove resolution complete, it does take important and significant steps towards this goal. First, it formally proves König's lemma and Herbrand's theorem. Second, it finds weaknesses in informal proofs of the lifting lemma and identifies opportunities to overcome this problem. Finally, it proves completeness thoroughly, but informally.

9.2 Contribution

The formalization of the resolution calculus is part of the ongoing effort of formalizing results of mathematics and computer science in proof assistants. There are already formalizations of several different proof systems and properties of different logics. Resolution is a central proof system in the study of logic, and especially in the area of automated theorem proving. Its formalization is therefore a desirable step towards the formalization of computer science.

9.3 Future Work

There are many opportunities for future work based on this thesis. The first and most obvious one is to finish the formalization of the completeness proof. The best opportunity to do this was presented in chapter 8 – namely to use another resolution calculus, that has a more thorough proof of the lifting lemma.

Another opportunity for future work is to try to prove formally that our current resolution calculus fulfills the lifting lemma. It is possible that there are thorough proofs of this somewhere in the literature, and it is also possible that we can repair the mistakes of the proofs we looked at.

Yet another opportunity is to create a verified automatic theorem prover from our formalization. The next important step in this direction is to formalize a unification algorithm. In chapter 8, we suggested adapting an already finished formalization. It could, however, also be an interesting result to formalize some other unification algorithm. The most efficient automatic theorem provers are based on the logical systems of the resolution calculus and superposition.

Therefore, it also could be interesting to work towards a formalization of superposition.

In chapter 3 we looked at two other opportunities for proving the resolution calculus complete. It would be interesting to pursue these opportunities also. Both of them build on formalization work that has already been done, and on advanced literature.

Appendix A

Formalization Code: TermsAndLiterals.thy

The code in appendix A.2 is a redistribution of a formalization by Berghofer [Ber07] with modifications by the author of this thesis. It is released under the BSD software license in appendix A.1.

A.1 BSD Software License


```
Copyright (c) 2004, Gerwin Klein, Tobias Nipkow, Lawrence C. Paulson
Copyright (c) 2004, contributing authors
                    (see author notice in individual files)

All rights reserved.

All files in the Archive of Formal Proofs that are unmarked or marked
with 'License: BSD' are released under the following license. Files
marked with 'License: LGPL' are released under the terms detailed in
LICENSE.LGPL


Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:
```

Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer. Redistributions
in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the documentation and/or
other materials provided with the distribution. Neither the name of
the Archive of Formal Proofs nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A.2 Terms and Literals

Author: Stefan Berghofer, TU Muenchen, 2003

Author: Anders Schlichtkrull, DTU, 2015

theory *TermsAndLiterals* **imports** *Main* **begin**
type-synonym *var-sym* = *string*
type-synonym *fun-sym* = *string*
type-synonym *pred-sym* = *string*
datatype *fterm* =
  Fun fun-sym fterm list
| *Var var-sym*

fun *ground* :: *fterm* ⇒ *bool* **where**
  ground (*Var x*) ⟷ *False*
| *ground* (*Fun f ts*) ⟷ (∀ *t* ∈ *set ts*. *ground t*)

abbreviation *grounds* :: *fterm list* ⇒ *bool* **where**
  grounds ts ≡ (∀ *t* ∈ *set ts*. *ground t*)
datatype *hterm* = *HFun fun-sym hterm list*
datatype *'t literal* =
  is-pos: *Pos* (*get-pred*: *pred-sym*) (*get-terms*: *'t list*)
| *Neg* (*get-pred*: *pred-sym*) (*get-terms*: *'t list*)

A.2.1 Enumerating datatypes

A.2.1.1 Enumerating Pairs of Natural Numbers

```
primrec diag :: nat ⇒ (nat × nat)
where
  diag 0 = (0, 0)
| diag (Suc n) =
    (let (x, y) = diag n
     in case y of
        0 ⇒ (0, Suc x)
      | Suc y ⇒ (Suc x, y))


theorem diag-le1: fst (diag (Suc n)) < Suc n
  by (induct n) (simp-all add: Let-def split-def split add: nat.split)


theorem diag-le2: snd (diag (Suc (Suc n))) < Suc (Suc n)
  apply (induct n)
  apply (simp-all add: Let-def split-def split add: nat.split nat.split-asm)
  apply (rule impI)
  apply (case-tac n)
  apply simp
  apply hypsubst
  apply (rule diag-le1)
  done


theorem diag-le3: fst (diag n) = Suc x ⟹ snd (diag n) < n
  apply (case-tac n)
  apply simp
  apply (case-tac nat)
  apply (simp add: Let-def)
  apply hypsubst
  apply (rule diag-le2)
  done


theorem diag-le4: fst (diag n) = Suc x ⟹ x < n
  apply (case-tac n)
  apply simp
  apply (case-tac nat)
  apply (simp add: Let-def)
  apply hypsubst-thin
  apply (drule sym)
  apply (drule ord-eq-less-trans)
  apply (rule diag-le1)
```

```
  apply simp
  done

function undiag :: nat × nat ⇒ nat
where
  undiag (0, 0) = 0
| undiag (0, Suc y) = Suc (undiag (y, 0))
| undiag (Suc x, y) = Suc (undiag (x, Suc y))
  by pat-completeness auto

termination
  by (relation measure (λ(x, y). ((x + y) * (x + y + 1)) div 2 + x)) auto


theorem diag-undiag [simp]: diag (undiag (x, y)) = (x, y)
  by (rule undiag.induct) (simp add: Let-def)+
```

A.2.1.2 Enumerating Trees

```
datatype btree = Leaf nat | Branch btree btree

function diag-btree :: nat ⇒ btree
where
  diag-btree n = (case fst (diag n) of
      0 ⇒ Leaf (snd (diag n))
    | Suc x ⇒ Branch (diag-btree x) (diag-btree (snd (diag n))))
  by auto

termination
  by (relation measure (λx. x)) (auto intro: diag-le3 diag-le4)

primrec undiag-btree :: btree ⇒ nat
where
  undiag-btree (Leaf n) = undiag (0, n)
| undiag-btree (Branch t1 t2) =
    undiag (Suc (undiag-btree t1), undiag-btree t2)


theorem diag-undiag-btree [simp]: diag-btree (undiag-btree t) = t
  by (induct t) (simp-all add: Let-def)

declare diag-btree.simps [simp del] undiag-btree.simps [simp del]
```

A.2.1.3 Enumerating Lists

```
fun list-of-btree :: (nat ⇒ 'a) ⇒ btree ⇒ 'a list
where
```

```
  list-of-btree f (Leaf x) = []
| list-of-btree f (Branch (Leaf n) t) = f n # list-of-btree f t

primrec btree-of-list :: ('a ⇒ nat) ⇒ 'a list ⇒ btree
where
  btree-of-list f [] = Leaf 0
| btree-of-list f (x # xs) = Branch (Leaf (f x)) (btree-of-list f xs)


definition diag-list :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list where
  diag-list f n = list-of-btree f (diag-btree n)


definition undiag-list :: ('a ⇒ nat) ⇒ 'a list ⇒ nat where
  undiag-list f xs = undiag-btree (btree-of-list f xs)


theorem diag-undiag-list [simp]:
  (⋀x. d (u x) = x) ⟹ diag-list d (undiag-list u xs) = xs
  by (induct xs) (simp-all add: diag-list-def undiag-list-def)
```

A.2.1.4 Enumerating hterms

```
fun term-of-btree :: (nat ⇒ string) ⇒ btree ⇒ hterm
and term-list-of-btree :: (nat ⇒ string) ⇒ btree ⇒ hterm list
where
  term-of-btree f (Leaf m) = HFun (f m) []
| term-of-btree f (Branch (Leaf m) t) =
    HFun (f m) (term-list-of-btree f t)
| term-list-of-btree f (Leaf m) = []
| term-list-of-btree f (Branch t1 t2) =
    term-of-btree f t1 # term-list-of-btree f t2

primrec btree-of-term :: (string ⇒ nat) ⇒ hterm ⇒ btree
   and btree-of-term-list :: (string ⇒ nat) ⇒ hterm list ⇒ btree
where
  btree-of-term f (HFun m ts) = (if ts=[] then Leaf (f m) else Branch (Leaf (f m))
(btree-of-term-list f ts))
| btree-of-term-list f [] = Leaf 0
| btree-of-term-list f (t # ts) = Branch (btree-of-term f t) (btree-of-term-list f ts)


theorem term-btree: assumes du: ⋀x. d (u x) = x
  shows term-of-btree d (btree-of-term u t) = t
  and term-list-of-btree d (btree-of-term-list u ts) = ts
  by (induct t and ts rule: btree-of-term.induct btree-of-term-list.induct) (simp-all
add: du)
```

definition *diag-term* :: (*nat* ⇒ *string*) ⇒ *nat* ⇒ *hterm* **where**
diag-term f n = *term-of-btree f* (*diag-btree n*)

definition *undiag-term* :: (*string* ⇒ *nat*) ⇒ *hterm* ⇒ *nat* **where**
undiag-term f t = *undiag-btree* (*btree-of-term f t*)

theorem *diag-undiag-term* [*simp*]:
(⋀*x*. *d* (*u x*) = *x*) ⟹ *diag-term d* (*undiag-term u t*) = *t*
by (*simp add*: *diag-term-def undiag-term-def term-btree*)

A.2.1.5 Enumerating chars

definition *diag-char* :: *nat* ⇒ *char* **where**
diag-char == *char-of-nat*

definition *undiag-char* :: *char* ⇒ *nat* **where**
undiag-char == *nat-of-char*

theorem *diag-undiag-char* [*simp*]: *diag-char* (*undiag-char c*) = *c*
unfolding *diag-char-def undiag-char-def* **by** *auto*

A.2.1.6 Enumerating strings

definition *diag-string* :: *nat* ⇒ *string* **where**
diag-string ≡ *diag-list diag-char*

definition *undiag-string* :: *string* ⇒ *nat* **where**
undiag-string ≡ *undiag-list undiag-char*

theorem *diag-undiag-string* [*simp*]:
diag-string (*undiag-string s*) = *s*
unfolding *diag-string-def undiag-string-def* **by** *auto*

A.2.1.7 Really enumerating hterms

definition *diag-hterm* :: *nat* ⇒ *hterm* **where**
diag-hterm ≡ *diag-term diag-string*

definition *undiag-hterm* :: *hterm* ⇒ *nat* **where**
undiag-hterm ≡ *undiag-term undiag-string*

theorem *diag-undiag-hterm*[*simp*]:
diag-hterm (*undiag-hterm t*) = *t*
unfolding *diag-hterm-def undiag-hterm-def* **by** *auto*

A.2.1.8 Enumerating hatoms

definition *undiag-hatom* :: *hterm literal* ⇒ *nat* **where**
undiag-hatom a ≡ *undiag* (*undiag-string* (*get-pred a*), *undiag-list undiag-hterm* (*get-terms a*))

definition *diag-hatom* :: *nat* ⇒ *hterm literal* **where**
diag-hatom a ≡
(*let* (*p*,*ts*) = *diag a in*
(*Pos* (*diag-string p*) (*diag-list diag-hterm ts*))
)

theorem *diag-undiag-hatom*[*simp*]:
is-pos a ⟹ *diag-hatom* (*undiag-hatom a*) = *a*
unfolding *diag-hatom-def undiag-hatom-def* **by** *auto*

A.2.1.9 Enumerating ground terms

primrec *fterm-of-hterm* :: *hterm* ⇒ *fterm*
and *fterms-of-hterms* :: *hterm list* ⇒ *fterm list* **where**
fterm-of-hterm (*HFun p ts*) = *Fun p* (*fterms-of-hterms ts*)
| *fterms-of-hterms* [] = []
| *fterms-of-hterms* (*t*#*ts*) = *fterm-of-hterm t* # *fterms-of-hterms ts*
primrec *hterm-of-fterm* :: *fterm* ⇒ *hterm*
and *hterms-of-fterms* :: *fterm list* ⇒ *hterm list* **where**
hterm-of-fterm (*Fun p ts*) = *HFun p* (*hterms-of-fterms ts*)
| *hterms-of-fterms* [] = []
| *hterms-of-fterms* (*t*#*ts*) = *hterm-of-fterm t* # *hterms-of-fterms ts*

theorem [*simp*]: *hterm-of-fterm* (*fterm-of-hterm t*) = *t*
hterms-of-fterms (*fterms-of-hterms ts*) = *ts*
by (*induct t* **and** *ts rule*: *fterm-of-hterm.induct fterms-of-hterms.induct*) *auto*

theorem [*simp*]: *ground t* ⟹ *fterm-of-hterm* (*hterm-of-fterm t*) = *t*
grounds ts ⟹*fterms-of-hterms* (*hterms-of-fterms ts*) = *ts*
by (*induct t* **and** *ts rule*: *hterm-of-fterm.induct hterms-of-fterms.induct*) *auto*

definition *diag-fterm* :: *nat* ⇒ *fterm* **where**
diag-fterm n = *fterm-of-hterm* (*diag-hterm n*)

definition *undiag-fterm* :: *fterm* ⇒ *nat* **where**
undiag-fterm t = *undiag-hterm* (*hterm-of-fterm t*)

theorem *diag-undiag-fterm*: *ground t* ⟹ *diag-fterm* (*undiag-fterm t*) = *t*
unfolding *diag-fterm-def undiag-fterm-def* **by** *auto*

A.2.1.10 Enumerating ground atoms

fun *fatom-of-hatom* :: *hterm literal* ⇒ *fterm literal* **where**
fatom-of-hatom (*Pos p ts*) = *Pos p* (*fterms-of-hterms ts*)
| *fatom-of-hatom* (*Neg p ts*) = *Neg p* (*fterms-of-hterms ts*)

fun *hatom-of-fatom* :: *fterm literal* ⇒ *hterm literal* **where**
hatom-of-fatom (*Pos p ts*) = *Pos p* (*hterms-of-fterms ts*)
| *hatom-of-fatom* (*Neg p ts*) = *Neg p* (*hterms-of-fterms ts*)

theorem [*simp*]: *hatom-of-fatom* (*fatom-of-hatom* (*Pos p ts*)) = *Pos p ts*
by *auto*

theorem [*simp*]: *grounds ts* ⟹ *fatom-of-hatom* (*hatom-of-fatom* (*Pos p ts*)) = *Pos p ts*
by *auto*

definition *undiag-fatom* :: *fterm literal* ⇒ *nat* **where**
undiag-fatom t = *undiag-hatom* (*hatom-of-fatom t*)

definition *diag-fatom* :: *nat* ⇒ *fterm literal* **where**
diag-fatom n = *fatom-of-hatom* (*diag-hatom n*)

theorem *diag-undiag-fatom*[*simp*]: *grounds ts* ⟹ *diag-fatom* (*undiag-fatom* (*Pos p ts*)) = *Pos p ts*
unfolding *undiag-fatom-def diag-fatom-def* **by** *auto*

end

Appendix B

Formalization Code: Tree.thy

---

theory *Tree* **imports** *Main* **begin**

hide-const (**open**) *Left Right*
type-synonym *dir* = *bool*

definition *Left* :: *bool* **where** *Left* = *True*

definition *Right* :: *bool* **where** *Right* = *False*
declare *Left-def* [*simp*]
declare *Right-def* [*simp*]
datatype *tree* =
  Leaf
| *Branch* (*ltree*: *tree*) (*rtree*: *tree*)

B.1 Paths

inductive *path* :: *dir list* ⇒ *tree* ⇒ *bool* **where**
  path [] *t*
| *path ds l* ⟹ *path* (*Left*#*ds*) (*Branch l r*)
| *path ds r* ⟹ *path* (*Right*#*ds*) (*Branch l r*)

lemma *path-inv-Leaf*: *path p Leaf* $\longleftrightarrow$ $p = []$
apply *auto*
using *path.simps* **apply** *blast*
apply (*simp add*: *path.intros*)
done

lemma *path-inv-Branch-Left*:
path (*Left*#*p*) (*Branch l r*) $\longleftrightarrow$ *path p l*
using *path.intros* **apply** *auto*
using *Left-def Right-def path.cases* **apply** *blast*
done

lemma *path-inv-Branch-Right*:
path (*Right*#*p*) (*Branch l r*) $\longleftrightarrow$ *path p r*
using *path.intros* **apply** *auto*
using *Left-def Right-def path.cases* **apply** *blast*
done

lemma *path-inv-Branch*:
path p (*Branch l r*) $\longleftrightarrow$ (p=[] $\vee$ ($\exists$ *a p'*. p=*a*#*p'*$\wedge$ (*a* $\longrightarrow$ *path p' l*) $\wedge$ ($\neg$*a* $\longrightarrow$ *path p' r*))) (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L* **then show** *?R* **using** *path.simps*[*of p*] **by** *auto*
next
assume *r*: *?R*
then show *?L*
proof
assume $p = []$ **then show** *?L* **using** *path.intros* **by** *auto*
next
assume $\exists$ *a p'*. p=*a*#*p'*$\wedge$ (*a* $\longrightarrow$ *path p' l*) $\wedge$ ($\neg$*a* $\longrightarrow$ *path p' r*)
then obtain *a p'* **where** p=*a*#*p'*$\wedge$ (*a* $\longrightarrow$ *path p' l*) $\wedge$ ($\neg$*a* $\longrightarrow$ *path p' r*) **by** *auto*
then show *?L* **using** *path.intros* **by** (*cases a*) *auto*
qed
qed

B.2 Branches

inductive *branch* :: *dir list* $\Rightarrow$ *tree* $\Rightarrow$ *bool* **where**
branch [] *Leaf*
| *branch ds l* $\Longrightarrow$ *branch* (*Left* # *ds*) (*Branch l r*)

```
| branch ds r ⟹ branch (Right # ds) (Branch l r)

lemma has-branch: ∃ b. branch b T
proof (induction T)
  case (Leaf) then show ?case using branch.intros by auto
next
  case (Branch T₁ T₂)
  then obtain b where branch b T₁ by auto
  then have branch (Left#b) (Branch T₁ T₂) using branch.intros by auto
  then show ?case by auto
qed


lemma branch-inv-Leaf: branch b Leaf ⟷ b = [] using branch.simps by blast


lemma branch-inv-Branch-Left:
  branch (Left#b) (Branch l r) ⟷ branch b l
using branch.intros apply auto
using Left-def Right-def branch.cases apply blast
done


lemma branch-inv-Branch-Right:
  branch (Right#b) (Branch l r) ⟷ branch b r
using branch.intros apply auto
using Left-def branch.cases by blast


lemma branch-inv-Branch:
  branch b (Branch l r) ⟷
     (∃ a b'. b=a#b'∧ (a ⟶ branch b' l) ∧ (¬a ⟶  branch b' r))
using branch.simps[of b] by auto


lemma branch-inv-Leaf2:
  T=Leaf ⟷ (∀ b. branch b T ⟶ b=[])
proof −
  {
    assume T=Leaf
    then have ∀ b. branch b T ⟶ b = [] using branch-inv-Leaf by auto
  }
  moreover
  {
    assume ∀ b. branch b T ⟶ b=[]
    then have ∀ b. branch b T ⟶ ¬(∃ a b'. b = a # b') by auto
    then have ∀ b. branch b T ⟶ ¬(∃ l r. branch b (Branch l r))
```

```
    using branch-inv-Branch by auto
  then have T=Leaf using has-branch[of T]  by (metis branch.simps)
 }
 ultimately show T=Leaf ⟷ (∀ b. branch b T ⟶ b=[]) by auto
qed


lemma branch-is-path:
  branch ds T ⟹ path ds T
proof (induction T arbitrary: ds)
  case Leaf
  then have ds = [] using branch-inv-Leaf by auto
  then show ?case using path.intros by auto
next
  case (Branch T1 T2)
  then obtain a b where ds-p: ds = a # b ∧ (a ⟶ branch b T1) ∧ (¬ a ⟶ branch
b T2) using branch-inv-Branch[of ds] by blast
  then have (a ⟶ path b T1) ∧ (¬a ⟶ path b T2) using Branch by auto
  then show ?case using ds-p path.intros by (cases a) auto
qed
```

B.3 Internal Nodes

```
inductive internal :: dir list ⇒ tree ⇒ bool where
  internal [] (Branch l r)
| internal ds l ⟹ internal (Left#ds) (Branch l r)
| internal ds r ⟹ internal (Right#ds) (Branch l r)


lemma internal-inv-Leaf: ¬internal b Leaf using internal.simps by blast


lemma internal-inv-Branch-Left:
  internal (Left#b) (Branch l r) ⟷ internal b l
apply rule
using internal.intros apply auto
using Left-def Right-def internal.cases apply blast
done


lemma internal-inv-Branch-Right:
  internal (Right#b) (Branch l r) ⟷ internal b r
using internal.intros apply auto
using Left-def Right-def internal.cases apply blast
done
```

lemma *internal-inv-Branch*:
internal p (Branch l r) $\longleftrightarrow$ *(p=[] ∨ (∃ a p'. p=a#p'∧ (a ⟶ internal p' l) ∧ (¬a ⟶ internal p' r)))* (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L* **then show** *?R* **using** *internal.simps*[*of p*] **by** *auto*
next
assume *r*: *?R*
then show *?L*
proof
assume *p* = [] **then show** *?L* **using** *internal.intros* **by** *auto*
next
assume ∃ *a p'*. *p=a#p'∧ (a ⟶ internal p' l) ∧ (¬a ⟶ internal p' r)*
then obtain *a p'* **where** *p=a#p'∧ (a ⟶ internal p' l) ∧ (¬a ⟶ internal p' r)* **by** *auto*
then show *?L* **using** *internal.intros* **by** (*cases a*) *auto*
qed
qed

lemma *internal-is-path*:
internal ds T ⟹ *path ds T*
proof (*induction T arbitrary*: *ds*)
case *Leaf*
then have *False* **using** *internal-inv-Leaf* **by** *auto*
then show *?case* **by** *auto*
next
case (*Branch T₁ T₂*)
then obtain *a b* **where** *ds-p*: *ds=[] ∨ ds = a # b ∧ (a ⟶ internal b T₁) ∧ (¬ a ⟶ internal b T₂)* **using** *internal-inv-Branch* **by** *blast*
then have *ds = [] ∨ (a ⟶ path b T₁) ∧ (¬a ⟶ path b T₂)* **using** *Branch* **by** *auto*
then show *?case* **using** *ds-p path.intros* **by** (*cases a*) *auto*
qed

fun *parent* :: *dir list* ⇒ *dir list* **where**
parent ds = tl ds

abbreviation *prefix* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
prefix a b ≡ ∃ *c*. *a* @ *c* = *b*

abbreviation *pprefix* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
pprefix a b ≡ ∃ *c*. *a* @ *c* = *b* ∧ *a*≠*b*

abbreviation *postfix* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
postfix a b ≡ ∃ *c*. *c* @ *a* = *b*

abbreviation *ppostfix* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
ppostfix a b ≡ ∃ *c*. *c* @ *a* = *b* ∧ *a*≠*b*

B.4 Deleting Nodes

fun *delete* :: *dir list* ⇒ *tree* ⇒ *tree* **where**
delete [] *T* = *Leaf*
| *delete* (*True*#*ds*) (*Branch T*$_1$ *T*$_2$) = *Branch* (*delete ds T*$_1$) *T*$_2$
| *delete* (*False*#*ds*) (*Branch T*$_1$ *T*$_2$) = *Branch T*$_1$ (*delete ds T*$_2$)

fun *cutoff* :: (*dir list* ⇒ *bool*) ⇒ *dir list* ⇒ *tree* ⇒ *tree* **where**
cutoff red ds (*Branch T*$_1$ *T*$_2$) =
(*if red ds then Leaf else Branch* (*cutoff red* (*ds*@[*Left*]) *T*$_1$) (*cutoff red* (*ds*@[*Right*]) *T*$_2$))
| *cutoff red ds Leaf* = *Leaf*

abbreviation *anypath* :: *tree* ⇒ (*dir list* ⇒ *bool*) ⇒ *bool* **where**
anypath T P ≡ ∀ *p*. *path p T* ⟶ *P p*

abbreviation *anybranch* :: *tree* ⇒ (*dir list* ⇒ *bool*) ⇒ *bool* **where**
anybranch T P ≡ ∀ *p*. *branch p T* ⟶ *P p*

abbreviation *anyinternal* :: *tree* ⇒ (*dir list* ⇒ *bool*) ⇒ *bool* **where**
anyinternal T P ≡ ∀ *p*. *internal p T* ⟶ *P p*

lemma *cutoff-branch'*:
anybranch T (λ*b*. *red*(*ds*@*b*)) ⟹ *anybranch* (*cutoff red ds T*) (λ*b*. *red*(*ds*@*b*))
proof (*induction T arbitrary*: *ds*)
case (*Leaf*)
let *?T* = *cutoff red ds Leaf*
{
fix *b*
assume *branch b ?T*

```
    then have branch b Leaf by auto
    then have red(ds@b) using Leaf by auto
  }
  then show ?case by simp
next
  case (Branch T1 T2)
  let ?T = cutoff red ds (Branch T1 T2)
  from Branch have ∀ p. branch (Left#p) (Branch T1 T2) ⟶ red (ds @ (Left#p))
by blast
  then have ∀ p. branch p T1 ⟶ red (ds @ (Left#p)) using branch.intros by auto
   then have anybranch T1 (λp. red ((ds @ [Left]) @ p))  using branch.intros by
auto
  then have aa: anybranch (cutoff red (ds @ [Left]) T1) (λp. red ((ds @ [Left]) @
p))
         using Branch by blast

 from Branch have ∀ p. branch (Right#p) (Branch T1 T2) ⟶ red (ds @ (Right#p))
by blast
  then have ∀ p. branch p T2 ⟶ red (ds @ (Right#p)) using branch.intros by auto
   then have anybranch T2 (λp. red ((ds @ [Right]) @ p))  using branch.intros by
auto
  then have bb: anybranch (cutoff red (ds @ [Right]) T2) (λp. red ((ds @ [Right]) @
p))
         using Branch by blast
  {
    fix b
    assume b-p: branch b ?T
    have red ds ∨ ¬red ds by auto
    then have red(ds@b)
      proof
        assume ds-p: red ds
        then have ?T = Leaf by auto
        then have b = [] using b-p branch-inv-Leaf by auto
        then show red(ds@b) using ds-p by auto
      next
        assume ds-p: ¬red ds
        let ?T1' = cutoff red (ds@[Left])  T1
        let ?T2' = cutoff red (ds@[Right]) T2
        from ds-p have ?T = Branch ?T1' ?T2' by auto
        from this b-p obtain a b' where b = a # b' ∧ (a ⟶ branch b' ?T1') ∧ (¬a
⟶ branch b' ?T2' ) using branch-inv-Branch[of b ?T1' ?T2'] by auto
        then show red(ds@b) using aa bb by (cases a) auto
      qed
  }
  then show ?case by blast
qed
```

lemma *cutoff-branch*: *anybranch T* (λ*p*. *red p*) ⟹ *anybranch* (*cutoff red* [] *T*) (λ*p*. *red p*)
using *cutoff-branch'*[*of T red* []] **by** *auto*

lemma *cutoff-internal'*:
anybranch T (λ*b*. *red*(*ds*@*b*)) ⟹ *anyinternal* (*cutoff red ds T*) (λ*b*. ¬*red*(*ds*@*b*))
proof (*induction T arbitrary*: *ds*)
case (*Leaf*) **then show** *?case* **using** *internal-inv-Leaf* **by** *simp*
next
case (*Branch* T_1 T_2)
let *?T* = *cutoff red ds* (*Branch* T_1 T_2)
from *Branch* **have** ∀ *p*. *branch* (*Left*#*p*) (*Branch* T_1 T_2) ⟶ *red* (*ds* @ (*Left*#*p*)) **by** *blast*
then have ∀ *p*. *branch p* T_1 ⟶ *red* (*ds* @ (*Left*#*p*)) **using** *branch.intros* **by** *auto*
then have *anybranch* T_1 (λ*p*. *red* ((*ds* @ [*Left*]) @ *p*)) **using** *branch.intros* **by** *auto*
then have *aa*: *anyinternal* (*cutoff red* (*ds* @ [*Left*]) T_1) (λ*p*. ¬ *red* ((*ds* @ [*Left*]) @ *p*)) **using** *Branch* **by** *blast*

from *Branch* **have** ∀ *p*. *branch* (*Right*#*p*) (*Branch* T_1 T_2) ⟶ *red* (*ds* @ (*Right*#*p*)) **by** *blast*
then have ∀ *p*. *branch p* T_2 ⟶ *red* (*ds* @ (*Right*#*p*)) **using** *branch.intros* **by** *auto*
then have *anybranch* T_2 (λ*p*. *red* ((*ds* @ [*Right*]) @ *p*)) **using** *branch.intros* **by** *auto*
then have *bb*: *anyinternal* (*cutoff red* (*ds* @ [*Right*]) T_2) (λ*p*. ¬ *red* ((*ds* @ [*Right*]) @ *p*)) **using** *Branch* **by** *blast*
{
fix *p*
assume *b-p*: *internal p ?T*
then have *ds-p*: ¬*red ds* **using** *internal-inv-Leaf internal.intros* **by** *auto*
have *p*=[] ∨ *p*≠[] **by** *auto*
then have ¬*red*(*ds*@*p*)
proof
assume *p*=[] **then show** ¬*red*(*ds*@*p*) **using** *ds-p* **by** *auto*
next
let *?*T_1*'* = *cutoff red* (*ds*@[*Left*]) T_1
let *?*T_2*'* = *cutoff red* (*ds*@[*Right*]) T_2
assume *p*≠[]
moreover
have *?T* = *Branch ?*T_1*' ?*T_2*'* **using** *ds-p* **by** *auto*
ultimately
obtain *a p'* **where** *b-p*: *p* = *a* # *p'* ∧
(*a* ⟶ *internal p'* (*cutoff red* (*ds* @ [*Left*]) T_1)) ∧
(¬ *a* ⟶ *internal p'* (*cutoff red* (*ds* @ [*Right*]) T_2))
using *b-p internal-inv-Branch*[*of p ?*T_1*' ?*T_2*'*] **by** *auto*

```
      then have ¬red(ds @ [a] @ p') using aa bb by (cases a) auto
      then show ¬red(ds @ p) using b-p by simp
    qed
  }
  then show ?case by blast
qed


lemma cutoff-internal: anybranch T red ⟹ anyinternal (cutoff red [] T) (λp. ¬red
p)
  using cutoff-internal'[of T red []] by auto


lemma cutoff-branch-internal:
  anybranch T red ⟹ ∃ T'. anyinternal T' (λp. ¬red p) ∧ anybranch T' (λp. red p)
  using cutoff-internal[of T] cutoff-branch[of T] by blast
```

B.5 Possibly Infinite Trees

```
abbreviation wf-tree :: dir list set ⇒ bool where
  wf-tree T ≡ (∀ ds d. (ds @ d) ∈ T ⟶ ds ∈ T)

fun subtree :: dir list set ⇒ dir list ⇒ dir list set where
  subtree T r = {ds ∈ T. ∃ ds'. ds = r @ ds'}

lemma subtree-pos:
  subtree T ds ⊆ subtree T (ds @ [Left]) ∪ subtree T (ds @ [Right]) ∪ {ds}
proof (rule subsetI; rule Set.UnCI)
  let ?subtree = subtree T
  fix x
  assume asm: x ∈ ?subtree ds
  assume x ∉ {ds}
  then have x ≠ ds by simp
  then have ∃ e d. x = ds @ [d] @ e using asm list.exhaust by auto
   then have (∃ e. x = ds @ [Left] @ e) ∨ (∃ e. x = ds @ [Right] @ e) using
bool.exhaust by auto
  then show x ∈ ?subtree (ds @ [Left]) ∪ ?subtree (ds @ [Right]) using asm by auto
qed
```

B.6 Infinite Paths

```
abbreviation list-chain :: (nat ⇒ 'a list) ⇒ bool where
  list-chain f ≡ (f 0 = []) ∧ (∀ n. ∃ a. f (Suc n) = (f n) @ [a])
```

lemma *chain-length*: *list-chain f* $\Longrightarrow$ *length* (*f n*) = *n*
apply (*induction n*)
apply *auto*
apply (*metis length-append-singleton*)
done

lemma *chain-prefix*: *list-chain f* $\Longrightarrow$ $n_1 \leq n_2 \Longrightarrow \exists\, a.$ (*f* n_1) @ *a* = (*f* n_2)
proof (*induction* n_2)
  case (*Suc* n_2)
  then have $n_1 \leq n_2 \vee n_1$ = *Suc* n_2 **by** *auto*
  then show *?case*
    proof
      assume $n_1 \leq n_2$
      then obtain *a* **where** *a*: *f* n_1 @ *a* = *f* n_2 **using** *Suc* **by** *auto*
      have *b*: $\exists\, b.$ *f* (*Suc* n_2) = *f* n_2 @ [*b*] **using** *Suc* **by** *auto*
      from *a b* **have** $\exists\, b.$ *f* n_1 @ (*a* @ [*b*]) = *f* (*Suc* n_2) **by** *auto*
      then show $\exists\, c.$ *f* n_1 @ *c* = *f* (*Suc* n_2) **by** *blast*
    next
      assume n_1 = *Suc* n_2
      then have *f* n_1 @ [] = *f* (*Suc* n_2) **by** *auto*
      then show $\exists\, a.$ *f* n_1 @ *a* = *f* (*Suc* n_2) **by** *auto*
    qed
qed *auto*

lemma *ith-in-extension*:
  assumes *chain*: *list-chain f*
  assumes *smalli*: *i* < *length* (*f* n_1)
  assumes $n_1 n_2$: $n_1 \leq n_2$
  shows *f* n_1 ! *i* = *f* n_2 ! *i*
proof −
  from *chain* $n_1 n_2$ **have** $\exists\, a.$ *f* n_1 @ *a* = *f* n_2 **using** *chain-prefix* **by** *blast*
  then obtain *a* **where** *a-p*: *f* n_1 @ *a* = *f* n_2 **by** *auto*
  have (*f* n_1 @ *a*) ! *i* = *f* n_1 ! *i* **using** *smalli* **by** (*simp add*: *nth-append*)
  then show *?thesis* **using** *a-p* **by** *auto*
qed

B.7 König's Lemma

lemma *inf-subs*:
  assumes *inf*: ¬*finite*(*subtree T ds*)
  shows ¬*finite*(*subtree T* (*ds* @ [*Left*])) ∨ ¬*finite*(*subtree T* (*ds* @ [*Right*]))
proof −
  let *?subtree* = *subtree T*

```
  {
    assume asms: finite(?subtree(ds @ [Left]))
                 finite(?subtree(ds @ [Right]))
    have ?subtree ds ⊆ ?subtree (ds @ [Left] ) ∪ ?subtree (ds @ [Right]) ∪ {ds}
      using subtree-pos by auto
    then have finite(?subtree (ds)) using asms by (simp add: finite-subset)
  }
  then show ¬finite(?subtree (ds @ [Left])) ∨ ¬finite(?subtree (ds @ [Right])) using
inf by auto
qed

fun buildchain :: (dir list ⇒ dir list) ⇒ nat ⇒ dir list where
  buildchain next 0 = []
| buildchain next (Suc n) = next (buildchain next n)

lemma konig:
  assumes inf: ¬finite T
  assumes wellformed: wf-tree T
  shows ∃ c. list-chain c ∧ (∀ n. (c n) ∈ T)
proof
  let ?subtree = subtree T
  let ?nextnode = λds. (if ¬finite (subtree T (ds @ [Left])) then ds @ [Left] else ds
@ [Right])

  let ?c = buildchain ?nextnode

  have is-chain: list-chain ?c by auto

  from wellformed have prefix: ⋀ds d. (ds @ d) ∈ T ⟹ ds ∈ T by blast

  {
    fix n
    have (?c n) ∈ T ∧ ¬finite (?subtree (?c n))
      proof (induction n)
        case 0
        have ∃ ds. ds ∈ T using inf by (simp add: not-finite-existsD)
        then obtain ds where ds ∈ T by auto
        then have ([]@ds) ∈ T by auto
        then have [] ∈ T using prefix[of []] by auto
        then show ?case using inf by auto
      next
        case (Suc n)
        from Suc have next-in:  (?c n) ∈ T by auto
        from Suc have next-inf: ¬finite (?subtree (?c n)) by auto

        from next-inf have next-next-inf:
           ¬finite (?subtree (?nextnode (?c n)))
```

```
          using inf-subs by auto
      then have ∃ ds. ds ∈ ?subtree (?nextnode (?c n))
        by (simp add: not-finite-existsD)
      then obtain ds where dss: ds ∈ ?subtree (?nextnode (?c n)) by auto
      then have ds ∈ T ∃ suf. ds = (?nextnode (?c n)) @ suf by auto
      then obtain suf where ds ∈ T ∧ ds = (?nextnode (?c n)) @ suf by auto
      then have (?nextnode (?c n)) ∈ T
        using prefix[of ?nextnode (?c n) suf] by auto

      then have (?c (Suc n)) ∈ T by auto
      then show ?case using next-next-inf by auto
    qed
  }
  then show list-chain ?c ∧ (∀ n. (?c n)∈ T)  using is-chain by auto
qed

end
```

Appendix C

Formalization Code: Resolution.thy

---

theory *Resolution* **imports** *TermsAndLiterals Tree* ~~/*src*/*HOL*/*IMP*/*Star* **begin**

hide-const (**open**) *TermsAndLiterals.Leaf TermsAndLiterals.Branch*

C.1 Terms and literals

fun *complement* :: *'t literal* ⇒ *'t literal* (-^{c} [*300*] *300*) **where**
  (*Pos P ts*)^c = *Neg P ts*
| (*Neg P ts*)^c = *Pos P ts*

lemma *cancel-comp1*: (l^c)^c = l **by** (*cases l*) *auto*

lemma *cancel-comp2*:
  assumes *asm*: ${l_1}^c = {l_2}^c$
  shows $l_1 = l_2$
proof −
  from *asm* **have** (${l_1}^c$)^c = (${l_2}^c$)^c **by** *auto*
  then have l_1 = (${l_2}^c$)^c **using** *cancel-comp1*[*of* l_1] **by** *auto*
  then show *?thesis* **using** *cancel-comp1*[*of* l_2] **by** *auto*
qed

lemma *comp-exi1*: ∃ l'. $l' = l^c$ **by** (*cases l*) *auto*

lemma *comp-exi2*: $\exists\, l.\ l' = l^c$
proof
  show $l' = (l'^c)^c$ **using** *cancel-comp1*[*of* l'] **by** *auto*
qed

lemma *comp-swap*: ${l_1}^c = l_2 \longleftrightarrow l_1 = {l_2}^c$
proof −
  have ${l_1}^c = l_2 \Longrightarrow l_1 = {l_2}^c$ **using** *cancel-comp1*[*of* l_1] **by** *auto*
  moreover
  have $l_1 = {l_2}^c \Longrightarrow {l_1}^c = l_2$ **using** *cancel-comp1* **by** *auto*
  ultimately
  show *?thesis* **by** *auto*
qed

C.2 Clauses

type-synonym *'t clause* = *'t literal set*

abbreviation *complementls* :: *'t literal set* ⇒ *'t literal set* (-^{C} [*300*] *300*) **where**
  $L^C \equiv$ *complement* ‘ L

lemma *cancel-compls1*: $(L^C)^C = L$
apply *auto*
apply (*simp add*: *cancel-comp1*)
apply (*metis imageI cancel-comp1*)
done

lemma *cancel-compls2*:
  assumes *asm*: ${L_1}^C = {L_2}^C$
  shows $L_1 = L_2$
proof −
  from *asm* **have** $({L_1}^C)^C = ({L_2}^C)^C$ **by** *auto*
  then show *?thesis* **using** *cancel-compls1*[*of* L_1] *cancel-compls1*[*of* L_2] **by** *simp*
qed

fun *varst* :: *fterm* ⇒ *var-sym set*
and *varsts* :: *fterm list* ⇒ *var-sym set* **where**
  varst (*Var* x) = $\{x\}$
| *varst* (*Fun* f ts) = *varsts* ts
| *varsts* [] = {}
| *varsts* (t # ts) = (*varst* t) ∪ (*varsts* ts)

definition *varsl* :: *fterm literal* ⇒ *var-sym set* **where**
  varsl l = *varsts* (*get-terms* l)

definition *varsls* :: *fterm literal set* ⇒ *var-sym set* **where**

varsls L ≡ ⋃ *l*∈*L*. *varsl l*

abbreviation *groundl* :: *fterm literal* ⇒ *bool* **where**
groundl l ≡ *grounds* (*get-terms l*)

abbreviation *groundls* :: *fterm clause* ⇒ *bool* **where**
groundls L ≡ ∀ *l* ∈ *L*. *groundl l*

lemma *ground-comp*: *groundl* (l^c) ⟷ *groundl l* **by** (*cases l*) *auto*

lemma *ground-compls*: *groundls* (L^C) ⟷ *groundls L* **using** *ground-comp* **by** *auto*

C.3 Semantics

type-synonym *'u fun-denot* = *fun-sym* ⇒ *'u list* ⇒ *'u*
type-synonym *'u pred-denot* = *pred-sym* ⇒ *'u list* ⇒ *bool*
type-synonym *'u var-denot* = *var-sym* ⇒ *'u*

fun *evalt* :: *'u var-denot* ⇒ *'u fun-denot* ⇒ *fterm* ⇒ *'u* **where**
evalt E F (*Var x*) = *E x*
| *evalt E F* (*Fun f ts*) = *F f* (*map* (*evalt E F*) *ts*)

abbreviation *evalts* :: *'u var-denot* ⇒ *'u fun-denot* ⇒ *fterm list* ⇒ *'u list* **where**
evalts E F ts ≡ *map* (*evalt E F*) *ts*

fun *evall* :: *'u var-denot* ⇒ *'u fun-denot* ⇒ *'u pred-denot* ⇒ *fterm literal* ⇒ *bool* **where**
evall E F G (*Pos p ts*) ⟷ (*G p* (*evalts E F ts*))
| *evall E F G* (*Neg p ts*) ⟷ ¬(*G p* (*evalts E F ts*))

definition *evalc* :: *'u fun-denot* ⇒ *'u pred-denot* ⇒ *fterm clause* ⇒ *bool* **where**
evalc F G C ⟷ (∀ *E*. ∃ *l* ∈ *C*. *evall E F G l*)

definition *evalcs* :: *'u fun-denot* ⇒ *'u pred-denot* ⇒ *fterm clause set* ⇒ *bool* **where**
evalcs F G Cs ⟷ (∀ *C* ∈ *Cs*. *evalc F G C*)

definition *validcs* :: *fterm clause set* ⇒ *bool* **where**
validcs Cs ⟷ (∀ *F G*. *evalcs F G Cs*)

C.3.1 Semantics of Ground Terms

lemma *ground-var-denott*: *ground t* ⟹ (*evalt E F t* = *evalt E′ F t*)
proof (*induction t*)
case (*Var x*)
then have *False* **by** *auto*
then show *?case* **by** *auto*

next
  case (*Fun f ts*)
  then have $\forall t \in set\ ts.\ ground\ t$ **by** *auto*
  then have $\forall t \in set\ ts.\ evalt\ E\ F\ t = evalt\ E'\ F\ t$ **using** *Fun* **by** *auto*
  then have $evalts\ E\ F\ ts = evalts\ E'\ F\ ts$ **by** *auto*
  then have $F\ f\ (map\ (evalt\ E\ F)\ ts) = F\ f\ (map\ (evalt\ E'\ F)\ ts)$ **by** *metis*
  then show *?case* **by** *simp*
qed

lemma *ground-var-denotts*: $grounds\ ts \Longrightarrow (evalts\ E\ F\ ts = evalts\ E'\ F\ ts)$
  using *ground-var-denott* **by** (*metis map-eq-conv*)

lemma *ground-var-denot*: $groundl\ l \Longrightarrow (evall\ E\ F\ G\ l = evall\ E'\ F\ G\ l)$
proof (*induction l*)
  case *Pos* **then show** *?case* **using** *ground-var-denotts* **by** (*metis evall.simps*(*1*) *literal.sel*(*3*))
next
  case *Neg* **then show** *?case* **using** *ground-var-denotts* **by** (*metis evall.simps*(*2*) *literal.sel*(*4*))
qed

C.4 Substitutions

type-synonym *substitution* = *var-sym* ⇒ *fterm*

fun *sub* :: *fterm* ⇒ *substitution* ⇒ *fterm* (-{-}$_t$ [*300*,*0*] *300*) **where**
  $(Var\ x)\{\sigma\}_t = \sigma\ x$
| $(Fun\ f\ ts)\{\sigma\}_t = Fun\ f\ (map\ (\lambda t.\ t\ \{\sigma\}_t)\ ts)$

abbreviation *subs* :: *fterm list* ⇒ *substitution* ⇒ *fterm list* (-{-}$_{ts}$ [*300*,*0*] *300*) **where**
  $ts\{\sigma\}_{ts} \equiv (map\ (\lambda t.\ t\ \{\sigma\}_t)\ ts)$

fun *subl* :: *fterm literal* ⇒ *substitution* ⇒ *fterm literal* (-{-}$_l$ [*300*,*0*] *300*) **where**
  $(Pos\ p\ ts)\{\sigma\}_l = Pos\ p\ (ts\{\sigma\}_{ts})$
| $(Neg\ p\ ts)\{\sigma\}_l = Neg\ p\ (ts\{\sigma\}_{ts})$

abbreviation *subls* :: *fterm literal set* ⇒ *substitution* ⇒ *fterm literal set* (-{-}$_{ls}$ [*300*,*0*] *300*) **where**
  $L\ \{\sigma\}_{ls} \equiv (\lambda l.\ l\ \{\sigma\}_l)\ `\ L$

definition *instance-oft* :: *fterm* ⇒ *fterm* ⇒ *bool* **where**
  $instance\text{-}oft\ t_1\ t_2 \longleftrightarrow (\exists \sigma.\ t_1 = t_2\{\sigma\}_t)$

definition *instance-ofts* :: *fterm list* ⇒ *fterm list* ⇒ *bool* **where**
  $instance\text{-}ofts\ ts_1\ ts_2 \longleftrightarrow (\exists \sigma.\ ts_1 = ts_2\{\sigma\}_{ts})$

definition *instance-ofl* :: *fterm literal* ⇒ *fterm literal* ⇒ *bool* **where**
instance-ofl l_1 l_2 ⟷ (∃ σ. l_1 = $l_2\{\sigma\}_l$)

lemma *comp-sub*: (l^c) $\{\sigma\}_l$=$(l$ $\{\sigma\}_l)^c$
by (*cases l*) *auto*

definition *var-renaming* :: *substitution* ⇒ *bool* **where**
var-renaming σ ⟷ (∀ x. ∃ y. σ x = *Var* y)

C.4.1 The Empty Substitution

abbreviation ε :: *substitution* **where**
ε ≡ *Var*

lemma *empty-subt*: $(t$ :: *fterm*$)\{\varepsilon\}_t = t$
apply (*induction t*)
apply (*auto simp add*: *map-idI*)
done

lemma *empty-subts*: $(ts$ :: *fterm list*$)\{\varepsilon\}_{ts} = ts$
using *empty-subt* **by** *auto*

lemma *empty-subl*: $(l$ :: *fterm literal*$)\{\varepsilon\}_l = l$
using *empty-subts* **by** (*cases l*) *auto*

lemma *instance-oft-self*: *instance-oft t t*
unfolding *instance-oft-def*
proof
show $t = t\{\varepsilon\}_t$ **using** *empty-subt* **by** *auto*
qed

lemma *instance-ofts-self*: *instance-ofts ts ts*
unfolding *instance-ofts-def*
proof
show $ts = ts\{\varepsilon\}_{ts}$ **using** *empty-subts* **by** *auto*
qed

lemma *instance-ofl-self*: *instance-ofl l l*
unfolding *instance-ofl-def*
proof
show $l = l\{\varepsilon\}_l$ **using** *empty-subl* **by** *auto*
qed

C.4.2 Substitutions and Ground Terms

lemma *ground-sub*: *ground t* ⟹ t $\{\sigma\}_t = t$

apply (*induction t*)
apply (*auto simp add*: *map-idI*)
done

lemma *ground-subs*: *grounds ts* $\Longrightarrow$ *ts* $\{\sigma\}_{ts} = ts$
using *ground-sub* **by** (*simp add*: *map-idI*)

lemma *groundl-subs*: *groundl l* $\Longrightarrow$ $l\ \{\sigma\}_l = l$
using *ground-subs* **by** (*cases l*) *auto*

lemma *groundls-subls*:
  assumes *ground*: *groundls L*
  shows $L\ \{\sigma\}_{ls} = L$
proof −
  {
    fix *l*
    assume *l-L*: $l \in L$
    then have *groundl l* **using** *ground* **by** *auto*
    then have $l = l\{\sigma\}_l$ **using** *groundl-subs* **by** *auto*
    moreover
    then have $l\ \{\sigma\}_l \in L\ \{\sigma\}_{ls}$ **using** *l-L* **by** *auto*
    ultimately
    have $l \in L\ \{\sigma\}_{ls}$ **by** *auto*
  }
  moreover
  {
    fix *l*
    assume *l-L*: $l \in L\ \{\sigma\}_{ls}$
    then obtain l' **where** l'*-p*: $l' \in L \wedge l'\ \{\sigma\}_l = l$ **by** *auto*
    then have $l' = l$ **using** *ground groundl-subs* **by** *auto*
    from *l-L* l'*-p this* **have** $l \in L$ **by** *auto*
  }
  ultimately show *?thesis* **by** *auto*
qed

C.4.3 Composition

definition *composition* :: *substitution* ⇒ *substitution* ⇒ *substitution* (**infixl** · 55)
where
  $(\sigma_1 \cdot \sigma_2)\ x = (\sigma_1\ x)\{\sigma_2\}_t$

lemma *composition-conseq2t*: $t\{\sigma_1\}_t\{\sigma_2\}_t = t\{\sigma_1 \cdot \sigma_2\}_t$
proof (*induction t*)
  case (*Var x*)
  have $(Var\ x)\{\sigma_1\}_t\{\sigma_2\}_t = (\sigma_1\ x)\{\sigma_2\}_t$ **by** *simp*
  also have ... $= (\sigma_1 \cdot \sigma_2)\ x$ **unfolding** *composition-def* **by** *simp*
  finally show *?case* **by** *auto*

next
case (*Fun t ts*)
then show *?case* **unfolding** *composition-def* **by** *auto*
qed

lemma *composition-conseq2ts*: $ts\{\sigma_1\}_{ts}\{\sigma_2\}_{ts} = ts\{\sigma_1 \cdot \sigma_2\}_{ts}$
using *composition-conseq2t* **by** *auto*

lemma *composition-conseq2l*: $l\{\sigma_1\}_l\{\sigma_2\}_l = l\{\sigma_1 \cdot \sigma_2\}_l$
using *composition-conseq2t* **by** (*cases l*) *auto*

lemma *composition-assoc*: $\sigma_1 \cdot (\sigma_2 \cdot \sigma_3) = (\sigma_1 \cdot \sigma_2) \cdot \sigma_3$
proof
fix *x*
show $(\sigma_1 \cdot (\sigma_2 \cdot \sigma_3))\ x = ((\sigma_1 \cdot \sigma_2) \cdot \sigma_3)\ x$ **unfolding** *composition-def* **using** *composition-conseq2t* **by** *simp*
qed

lemma *empty-comp1*: $(\sigma \cdot \varepsilon) = \sigma$
proof
fix *x*
show $(\sigma \cdot \varepsilon)\ x = \sigma\ x$ **unfolding** *composition-def* **using** *empty-subt* **by** *auto*
qed

lemma *empty-comp2*: $(\varepsilon \cdot \sigma) = \sigma$
proof
fix *x*
show $(\varepsilon \cdot \sigma)\ x = \sigma\ x$ **unfolding** *composition-def* **by** *simp*
qed

lemma *instance-ofts-trans* :
assumes ts_{12}: *instance-ofts* ts_1 ts_2
assumes ts_{23}: *instance-ofts* ts_2 ts_3
shows *instance-ofts* ts_1 ts_3
proof −
from ts_{12} **obtain** σ_{12} **where** $ts_1 = ts_2\ \{\sigma_{12}\}_{ts}$
unfolding *instance-ofts-def* **by** *auto*
moreover
from ts_{23} **obtain** σ_{23} **where** $ts_2 = ts_3\ \{\sigma_{23}\}_{ts}$
unfolding *instance-ofts-def* **by** *auto*
ultimately
have $ts_1 = ts_3\ \{\sigma_{23}\}_{ts}\ \{\sigma_{12}\}_{ts}$ **by** *auto*
then have $ts_1 = ts_3\ \{\sigma_{23} \cdot \sigma_{12}\}_{ts}$ **using** *composition-conseq2ts* **by** *simp*
then show *?thesis* **unfolding** *instance-ofts-def* **by** *auto*
qed

C.5 Unifiers

definition *unifiert* :: *substitution* ⇒ *fterm set* ⇒ *bool* **where**
unifiert σ *ts* ⟷ (∃ *t'*. ∀ *t* ∈ *ts*. *t*{σ}$_t$ = *t'*)

definition *unifierls* :: *substitution* ⇒ *fterm literal set* ⇒ *bool* **where**
unifierls σ *L* ⟷ (∃ *l'*. ∀ *l* ∈ *L*. *l*{σ}$_l$ = *l'*)

lemma *unif-sub*:
assumes *unif*: *unifierls* σ *L*
assumes *nonempty*: *L* ≠ {}
shows ∃ *l*. *subls L* σ = {*subl l* σ}
proof −
from *nonempty* **obtain** *l* **where** *l* ∈ *L* **by** *auto*
from *unif this* **have** *L* {σ}$_{ls}$ = {*l* {σ}$_l$} **unfolding** *unifierls-def* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *unifierls-def2*:
assumes *L-elem*: *L* ≠ {}
shows *unifierls* σ *L* ⟷ (∃ *l*. *L* {σ}$_{ls}$ ={*l*})
proof
assume *unif*: *unifierls* σ *L*
from *L-elem* **obtain** *l* **where** *l* ∈ *L* **by** *auto*
then have *L* {σ}$_{ls}$ = {*l* {σ}$_l$} **using** *unif* **unfolding** *unifierls-def* **by** *auto*
then show ∃ *l*. *L*{σ}$_{ls}$ = {*l*} **by** *auto*
next
assume ∃ *l*. *L* {σ}$_{ls}$ ={*l*}
then obtain *l* **where** *L* {σ}$_{ls}$ = {*l*} **by** *auto*
then have ∀ *l'* ∈ *L*. *l'*{σ}$_l$ = *l* **by** *auto*
then show *unifierls* σ *L* **unfolding** *unifierls-def* **by** *auto*
qed

lemma *groundls-unif-singleton*:
assumes *groundls*: *groundls L*
assumes *unif*: *unifierls* σ′ *L*
assumes *empt*: *L* ≠ {}
shows ∃ *l*. *L* = {*l*}
proof −
from *unif empt* **have** ∃ *l*. *L* {σ′}$_{ls}$ = {*l*} **using** *unif-sub* **by** *auto*
then show *?thesis* **using** *groundls-subls groundls* **by** *auto*
qed

definition *unifiablet* :: *fterm set* ⇒ *bool* **where**
unifiablet fs ⟷ (∃ σ. *unifiert* σ *fs*)

definition *unifiablels* :: *fterm literal set* ⇒ *bool* **where**
unifiablels L ⟷ (∃ *σ*. *unifierls σ L*)

lemma *unifier-comp*[*simp*]: *unifierls σ* (L^C) ⟷ *unifierls σ L*
proof
assume *unifierls σ* (L^C)
then obtain *l''* **where** *l''-p*: ∀ *l* ∈ L^C. $l\{\sigma\}_l = l''$
unfolding *unifierls-def* **by** *auto*
obtain *l'* **where** $(l')^c = l''$ **using** *comp-exi2*[*of l''*] **by** *auto*
from *this l''-p* **have** *l'-p*:∀ *l* ∈ L^C. $l\{\sigma\}_l = (l')^c$ **by** *auto*
have ∀ *l* ∈ *L*. $l\{\sigma\}_l = l'$
proof
fix *l*
assume *l*∈*L*
then have $l^c \in L^C$ **by** *auto*
then have $(l^c)\{\sigma\}_l = (l')^c$ **using** *l'-p* **by** *auto*
then have $(l\ \{\sigma\}_l)^c = (l')^c$ **by** (*cases l*) *auto*
then show $l\{\sigma\}_l = l'$ **using** *cancel-comp2* **by** *blast*
qed
then show *unifierls σ L* **unfolding** *unifierls-def* **by** *auto*
next
assume *unifierls σ L*
then obtain *l'* **where** *l'-p*: ∀ *l* ∈ *L*. $l\{\sigma\}_l = l'$ **unfolding** *unifierls-def* **by** *auto*
have ∀ *l* ∈ L^C. $l\{\sigma\}_l = (l')^c$
proof
fix *l*
assume *l* ∈ L^C
then have $l^c \in L$ **using** *cancel-comp1* **by** (*metis image-iff*)
then show $l\{\sigma\}_l = (l')^c$ **using** *l'-p comp-sub cancel-comp1* **by** *metis*
qed
then show *unifierls σ* (L^C) **unfolding** *unifierls-def* **by** *auto*
qed

lemma *unifier-sub1*: *unifierls σ L* ⟹ *L'* ⊆ *L* ⟹ *unifierls σ L'*
unfolding *unifierls-def* **by** *auto*

lemma *unifier-sub2*:
assumes *asm*: *unifierls σ* (*L*$_1$ ∪ *L*$_2$)
shows *unifierls σ L*$_1$ ∧ *unifierls σ L*$_2$
proof −
have *L*$_1$ ⊆ (*L*$_1$ ∪ *L*$_2$) ∧ *L*$_2$ ⊆ (*L*$_1$ ∪ *L*$_2$) **by** *simp*
from *this asm* **show** *?thesis* **using** *unifier-sub1* **by** *auto*
qed

C.5.1 Most General Unifiers

definition *mgut* :: *substitution* ⇒ *fterm set* ⇒ *bool* **where**
mgut σ *fs* ⟷ *unifiert* σ *fs* ∧ (∀ *u*. *unifiert u fs* ⟶ (∃ *i*. *u* = σ · *i*))

definition *mguls* :: *substitution* ⇒ *fterm literal set* ⇒ *bool* **where**
mguls σ *L* ⟷ *unifierls* σ *L* ∧ (∀ *u*. *unifierls u L* ⟶ (∃ *i*. *u* = σ · *i*))

C.6 Resolution

definition *applicable* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *bool* **where**
applicable C_1 C_2 L_1 L_2 σ ⟷
$C_1 \neq \{\} \wedge C_2 \neq \{\} \wedge L_1 \neq \{\} \wedge L_2 \neq \{\}$
∧ *varsls* C_1 ∩ *varsls* C_2 = {}
∧ $L_1 \subseteq C_1 \wedge L_2 \subseteq C_2$
∧ *mguls* σ ($L_1 \cup {L_2}^C$)

definition *resolution* :: *fterm clause* ⇒ *fterm clause*
⇒ *fterm literal set* ⇒ *fterm literal set*
⇒ *substitution* ⇒ *fterm clause* **where**
resolution C_1 C_2 L_1 L_2 σ = (C_1 $\{\sigma\}_{ls}$ − L_1 $\{\sigma\}_{ls}$) ∪ (C_2 $\{\sigma\}_{ls}$ − L_2 $\{\sigma\}_{ls}$)

inductive *resolution-step* :: *fterm clause set* ⇒ *fterm clause set* ⇒ *bool* **where**
resolution-rule:
$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow$ *applicable* C_1 C_2 L_1 L_2 σ ⟹
resolution-step Cs (*Cs* ∪ {*resolution* C_1 C_2 L_1 L_2 σ})
| *standardize-apart*:
$C \in Cs \Longrightarrow$ *var-renaming* σ ⟹ *resolution-step Cs* (*Cs* ∪ {*C* $\{\sigma\}_{ls}$})

definition *resolution-deriv* :: *fterm clause set* ⇒ *fterm clause set* ⇒ *bool* **where**
resolution-deriv = *star resolution-step*

lemma *ground-resolution*:
assumes *ground*: *groundls* C_1 ∧ *groundls* C_2
assumes *appl*: *applicable* C_1 C_2 L_1 L_2 σ
shows *resolution* C_1 C_2 L_1 L_2 σ = (C_1 − L_1) ∪ (C_2 − L_2) ∧ (∃ *l*. L_1 = {*l*} ∧ L_2 = $\{l\}^C$)
proof −
from *appl ground* **have** *groundl*: *groundls* L_1 ∧ *groundls* L_2 **unfolding** *applicable-def* **by** *auto*
from *this ground appl* **have** *resl*: (C_1 $\{\sigma\}_{ls}$ − L_1 $\{\sigma\}_{ls}$) ∪ (C_2 $\{\sigma\}_{ls}$ − L_2 $\{\sigma\}_{ls}$) = (C_1 − L_1) ∪ (C_2 − L_2)
using *groundls-subls* **unfolding** *applicable-def* **by** *auto*

from *ground appl* **have** $l_1{}'l_2{}'$*ground*: *groundls* L_1 ∧ *groundls* L_2
unfolding *applicable-def* **by** *auto*

then have *groundls* $(L_1 \cup {L_2}^C)$ **using** *ground-compls* **by** *auto*
moreover
from *appl* **have** *unifierls* σ $(L_1 \cup {L_2}^C)$ **unfolding** *mguls-def applicable-def* **by** *auto*
ultimately
obtain l **where** $L_1 \cup {L_2}^C = \{l\}$
using *appl groundls-unif-singleton*
unfolding *applicable-def* **by** (*metis sup-eq-bot-iff*)
from *appl this* **have** $L_1 = \{l\} \wedge {L_2}^C = \{l\}$ **unfolding** *applicable-def* **by** (*metis image-is-empty singleton-Un-iff*)
then have *l-p*: $L_1 = \{l\} \wedge L_2 = \{l\}^C$ **using** *cancel-compls1*[*of* L_2] **by** *auto*

from *resl l-p* **show** *?thesis* **unfolding** *resolution-def* **by** *auto*
qed

lemma *ground-resolution-ground*:
assumes *asm*: *groundls* $C_1 \wedge$ *groundls* C_2
assumes *appl*: *applicable* C_1 C_2 L_1 L_2 σ
shows *groundls* (*resolution* C_1 C_2 L_1 L_2 σ)
proof −
from *asm appl* **have** *resolution* C_1 C_2 L_1 L_2 $\sigma = (C_1 - L_1) \cup (C_2 - L_2)$ **using** *ground-resolution* **by** *auto*
then show *?thesis* **using** *asm* **by** *auto*
qed

C.7 Soundness

fun *evalsub* :: *'u fun-denot* ⇒ *'u var-denot* ⇒ *substitution* ⇒ *'u var-denot* **where**
evalsub F E σ = (*evalt E F*) ∘ σ

lemma *substitutiont*: *evalt E F* $(t\ \{\sigma\}_t)$ = *evalt* (*evalsub F E* σ) *F t*
apply (*induction t*)
apply *auto*
apply (*metis* (*mono-tags*, *lifting*) *comp-apply map-cong*)
done

lemma *substitutionts*: *evalts E F* $(ts\ \{\sigma\}_{ts})$ = *evalts* (*evalsub F E* σ) *F ts*
using *substitutiont* **by** *auto*

lemma *substitutionl*: *evall E F G* $(l\ \{\sigma\}_l) \longleftrightarrow$ *evall* (*evalsub F E* σ) *F G l*
apply (*induction l*)
using *substitutionts* **apply** (*metis evall.simps*(*1*) *subl.simps*(*1*))
using *substitutionts* **apply** (*metis evall.simps*(*2*) *subl.simps*(*2*))
done

lemma *subst-sound*:
assumes *asm*: *evalc F G C*
shows *evalc F G* (*C* $\{\sigma\}_{ls}$)
proof −
have $\forall E.\ \exists l \in C\ \{\sigma\}_{ls}.$ *evall E F G l*
proof
fix *E*
from *asm* **have** $\exists l \in C.$ *evall* (*evalsub F E* σ) *F G l* **unfolding** *evalc-def* **by** *auto*
then show $\exists l \in C\ \{\sigma\}_{ls}.$ *evall E F G l* **using** *substitutionl*[*of E F G - σ*] **by** *auto*
qed
then show *?thesis* **unfolding** *evalc-def* **by** *auto*
qed

lemma *simple-resolution-sound*:
assumes $C_1 sat$: *evalc F G* C_1
assumes $C_2 sat$: *evalc F G* C_2
assumes $l_1 inc_1$: $l_1 \in C_1$
assumes $l_2 inc_2$: $l_2 \in C_2$
assumes *Comp*: ${l_1}^c = l_2$
shows *evalc F G* $((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))$
proof −
have $\forall E.\ \exists l \in (((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))).$ *evall E F G l*
proof
fix *E*
have *evall E F G* $l_1 \vee$ *evall E F G* l_2 **using** *Comp* **by** (*cases* l_1) *auto*
then show $\exists l \in (((C_1 - \{l_1\}) \cup (C_2 - \{l_2\}))).$ *evall E F G l*
proof
assume *evall E F G* l_1
then have $\neg$*evall E F G* l_2 **using** *Comp* **by** (*cases* l_1) *auto*
then have $\exists {l_2}' \in C_2.\ {l_2}' \neq l_2 \wedge$ *evall E F G* ${l_2}'$ **using** $l_2 inc_2$ $C_2 sat$ **unfolding** *evalc-def* **by** *auto*
then show $\exists l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}).$ *evall E F G l* **by** *auto*
next
assume *evall E F G* l_2
then have $\neg$*evall E F G* l_1 **using** *Comp* **by** (*cases* l_1) *auto*
then have $\exists {l_1}' \in C_1.\ {l_1}' \neq l_1 \wedge$ *evall E F G* ${l_1}'$ **using** $l_1 inc_1$ $C_1 sat$ **unfolding** *evalc-def* **by** *auto*
then show $\exists l \in (C_1 - \{l_1\}) \cup (C_2 - \{l_2\}).$ *evall E F G l* **by** *auto*
qed
qed
then show *?thesis* **unfolding** *evalc-def* **by** *simp*
qed

lemma *resolution-sound*:
assumes sat_1: *evalc F G* C_1

assumes sat_2: *evalc F G* C_2
assumes *appl*: *applicable* C_1 C_2 L_1 L_2 σ
shows *evalc F G* (*resolution* C_1 C_2 L_1 L_2 σ)
proof −
from sat_1 **have** $sat_1\sigma$: *evalc F G* (C_1 $\{\sigma\}_{ls}$) **using** *subst-sound* **by** *blast*
from sat_2 **have** $sat_2\sigma$: *evalc F G* (C_2 $\{\sigma\}_{ls}$) **using** *subst-sound* **by** *blast*

from *appl* **obtain** l_1 **where** l_1-*p*: $l_1 \in L_1$ **unfolding** *applicable-def* **by** *auto*

from l_1-*p appl* **have** $l_1 \in C_1$ **unfolding** *applicable-def* **by** *auto*
then have $inc_1\sigma$: l_1 $\{\sigma\}_l \in C_1$ $\{\sigma\}_{ls}$ **by** *auto*

from l_1-*p* **have** $unified_1$: $l_1 \in (L_1 \cup ({L_2}^C))$ **by** *auto*

from l_1-*p appl* **have** $l_1\sigma isl_1\sigma$: $\{l_1\{\sigma\}_l\} = L_1$ $\{\sigma\}_{ls}$
unfolding *mguls-def unifierls-def applicable-def* **by** *auto*

from *appl* **obtain** l_2 **where** l_2-*p*: $l_2 \in L_2$ **unfolding** *applicable-def* **by** *auto*

from l_2-*p appl* **have** $l_2 \in C_2$ **unfolding** *applicable-def* **by** *auto*
then have $inc_2\sigma$: l_2 $\{\sigma\}_l \in C_2$ $\{\sigma\}_{ls}$ **by** *auto*

from l_2-*p* **have** $unified_2$: ${l_2}^c \in (L_1 \cup ({L_2}^C))$ **by** *auto*

from $unified_1$ $unified_2$ *appl* **have** l_1 $\{\sigma\}_l = ({l_2}^c)\{\sigma\}_l$
unfolding *mguls-def unifierls-def applicable-def* **by** *auto*
then have *comp*: $(l_1$ $\{\sigma\}_l)^c = l_2$ $\{\sigma\}_l$ **using** *comp-sub comp-swap* **by** *auto*

from *appl* **have** *unifierls* σ $({L_2}^C)$
using *unifier-sub2* **unfolding** *mguls-def applicable-def* **by** *blast*
then have *unifierls* σ L_2 **by** *auto*
from *this* l_2-*p* **have** $l_2\sigma isl_2\sigma$: $\{l_2\{\sigma\}_l\} = L_2$ $\{\sigma\}_{ls}$ **unfolding** *unifierls-def* **by** *auto*

from $sat_1\sigma$ $sat_2\sigma$ $inc_1\sigma$ $inc_2\sigma$ *comp* **have** *evalc F G* ($C_1\{\sigma\}_{ls} - \{l_1\{\sigma\}_l\} \cup (C_2\{\sigma\}_{ls} - \{l_2\{\sigma\}_l\})$) **using** *simple-resolution-sound*[*of F G* C_1 $\{\sigma\}_{ls}$ C_2 $\{\sigma\}_{ls}$ l_1 $\{\sigma\}_l$ l_2 $\{\sigma\}_l$]
by *auto*
from *this* $l_1\sigma isl_1\sigma$ $l_2\sigma isl_2\sigma$ **show** *?thesis* **unfolding** *resolution-def* **by** *auto*
qed

lemma *sound-step*: *resolution-step Cs Cs'* $\Longrightarrow$ *evalcs F G Cs* $\Longrightarrow$ *evalcs F G Cs'*
proof (*induction rule*: *resolution-step.induct*)
case (*resolution-rule* C_1 *Cs* C_2 l_1 l_2 σ)
then have *evalc F G* $C_1 \wedge$ *evalc F G* C_2 **unfolding** *evalcs-def* **by** *auto*
then have *evalc F G* (*resolution* C_1 C_2 l_1 l_2 σ)
using *resolution-sound resolution-rule* **by** *auto*
then show *?case* **using** *resolution-rule* **unfolding** *evalcs-def* **by** *auto*

next
case (*standardize-apart C Cs* σ)
then have *evalc F G C* **unfolding** *evalcs-def* **by** *auto*
then have *evalc F G* ($C\{\sigma\}_{ls}$) **using** *subst-sound* **by** *auto*
then show *?case* **using** *standardize-apart* **unfolding** *evalcs-def* **by** *auto*
qed

lemma *sound-derivation*:
resolution-deriv Cs Cs' $\Longrightarrow$ *evalcs F G Cs* $\Longrightarrow$ *evalcs F G Cs'*
unfolding *resolution-deriv-def*
proof (*induction rule*: *star.induct*)
case *refl* **then show** *?case* **by** *auto*
next
case (*step* Cs_1 Cs_2 Cs_3) **then show** *?case* **using** *sound-step* **by** *auto*
qed

C.8 Enumerations

fun *hlit-of-flit* :: *fterm literal* $\Rightarrow$ *hterm literal* **where**
hlit-of-flit (*Pos P ts*) = *Pos P* (*hterms-of-fterms ts*)
| *hlit-of-flit* (*Neg P ts*) = *Neg P* (*hterms-of-fterms ts*)

lemma *undiag-neg*: *undiag-fatom* (*Neg P ts*) = *undiag-fatom* (*Pos P ts*)
unfolding *undiag-fatom-def undiag-hatom-def* **by** *auto*

lemma *undiag-neg2*: *undiag-hatom* (*Neg P ts*) = *undiag-hatom* (*Pos P ts*)
unfolding *undiag-fatom-def undiag-hatom-def* **by** *auto*

lemma *ground-h-undiag*: *groundl l* $\Longrightarrow$ *undiag-hatom* (*hlit-of-flit l*) = *undiag-fatom l*
proof (*induction l*)
case (*Pos P ts*)
then show *?case* **unfolding** *undiag-fatom-def* **by** *auto*
next
case (*Neg P ts*)
then show *?case* **using** *undiag-neg undiag-neg2* **unfolding** *undiag-fatom-def* **by** *auto*
qed

C.9 Herbrand Interpretations

value *HFun*

lemma *hterms-ground*: *ground* (*fterm-of-hterm t*) *grounds* (*fterms-of-hterms ts*)
apply (*induction t* **and** *ts rule*: *fterm-of-hterm.induct fterms-of-hterms.induct*)
apply *auto*
done

lemma *eval-ground*: *ground t* $\Longrightarrow$ (*evalt E HFun t*) = *hterm-of-fterm t grounds ts*
$\Longrightarrow$ (*evalts E HFun ts*) = *hterms-of-fterms ts*
apply (*induction t* **and** *ts rule*: *hterm-of-fterm.induct hterms-of-fterms.induct*)
apply *auto*
done

lemma *evall-grounds*:
assumes *asm*: *grounds ts*
shows *evall E HFun G* (*Pos P ts*) $\longleftrightarrow$ *G P* (*hterms-of-fterms ts*)
proof −
have *evall E HFun G* (*Pos P ts*) = *G P* (*evalts E HFun ts*) **by** *auto*
also have ... = *G P* (*hterms-of-fterms ts*) **using** *asm eval-ground* **by** *metis*
finally show *?thesis* **by** *auto*
qed

C.10 Partial Interpretations

type-synonym *partial-pred-denot* = *bool list*

fun *falsifiesl* :: *partial-pred-denot* ⇒ *fterm literal* ⇒ *bool* **where**
falsifiesl G (*Pos p ts*) =
(∃ *i ts′*.
i < *length G*
∧ *G* ! *i* = *False*
∧ *diag-fatom i* = *Pos p ts′*
∧ *instance-ofts ts′ ts*)
| *falsifiesl G* (*Neg p ts*) =
(∃ *i ts′*.
i < *length G*
∧ *G* ! *i* = *True*
∧ *diag-fatom i* = *Pos p ts′*
∧ *instance-ofts ts′ ts*)

abbreviation *falsifiesc* :: *partial-pred-denot* ⇒ *fterm clause* ⇒ *bool* **where**
falsifiesc G C ≡ (∀ *l* ∈ *C*. *falsifiesl G l*)

abbreviation *falsifiescs* :: *partial-pred-denot* ⇒ *fterm clause set* ⇒ *bool* **where**
falsifiescs G Cs ≡ (∃ *C* ∈ *Cs*. *falsifiesc G C*)

abbreviation *extend* :: (*nat* ⇒ *partial-pred-denot*) ⇒ *hterm pred-denot* **where**

```
  extend f P ts ≡ (
    let n = undiag-hatom (Pos P ts) in
      f (Suc n) ! n
    )

fun sub-of-denot :: hterm var-denot ⇒ substitution where
  sub-of-denot E = fterm-of-hterm ∘ E

lemma ground-sub-of-denott: ground ((t :: fterm) {sub-of-denot E}t)
apply (induction t)
apply (auto simp add: hterms-ground)
done

lemma ground-sub-of-denotts: grounds ((ts :: fterm list) {sub-of-denot E}ts)
apply auto
using ground-sub-of-denott apply simp
done

lemma ground-sub-of-denotl: groundl ((l :: fterm literal) {sub-of-denot E}l)
proof −
  have grounds (subs (get-terms l :: fterm list) (sub-of-denot E))
    using ground-sub-of-denotts by auto
  then show ?thesis by (cases l)  auto
qed

lemma sub-of-denot-equivx: evalt E HFun (sub-of-denot E x) = E x
proof −
  have ground (sub-of-denot E x) using hterms-ground by auto
  then
  have evalt E HFun (sub-of-denot E x) = hterm-of-fterm (sub-of-denot E x)
    using eval-ground(1) by auto
  also have ... = hterm-of-fterm (fterm-of-hterm (E x)) by auto
  also have ... = E x by auto
  finally show ?thesis by auto
qed

lemma sub-of-denot-equivt:
    evalt E HFun (t {sub-of-denot E}t) = evalt E HFun t
apply (induction t)
using sub-of-denot-equivx apply auto
done

lemma sub-of-denot-equivts: evalts E HFun (ts {sub-of-denot E}ts) = evalts E HFun
ts
using sub-of-denot-equivt apply simp
done
```

lemma *sub-of-denot-equivl*: *evall E HFun G* (*l* {*sub-of-denot E*}$_l$) = *evall E HFun G l*
proof (*induction l*)
case (*Pos p ts*)
have *evall E HFun G* ((*Pos p ts*) {*sub-of-denot E*}$_l$) ⟷ *G p* (*evalts E HFun* (*ts* {*sub-of-denot E*}$_{ts}$)) **by** *auto*
also have ... ⟷ *G p* (*evalts E HFun ts*) **using** *sub-of-denot-equivts*[*of E ts*] **by** *metis*
also have ... ⟷ *evall E HFun G* (*Pos p ts*) **by** *simp*
finally
show *?case* **by** *blast*
next
case (*Neg p ts*)
have *evall E HFun G* ((*Neg p ts*) {*sub-of-denot E*}$_l$) ⟷ ¬*G p* (*evalts E HFun* (*ts* {*sub-of-denot E*}$_{ts}$)) **by** *auto*
also have ... ⟷ ¬*G p* (*evalts E HFun ts*) **using** *sub-of-denot-equivts*[*of E ts*] **by** *metis*
also have ... = *evall E HFun G* (*Neg p ts*) **by** *simp*
finally
show *?case* **by** *blast*
qed

lemma *sub-of-denot-equiv-ground'*:
evall E HFun G l = *evall E HFun G* (*l* {*sub-of-denot E*}$_l$) ∧ *groundl* (*l* {*sub-of-denot E*}$_l$)
using *sub-of-denot-equivl ground-sub-of-denotl* **by** *auto*

lemma *partial-equiv-subst'*: *falsifiesl G* ((*l* ::*fterm literal*) {σ}$_l$) ⟹ *falsifiesl G l*
proof (*induction l*)
case (*Pos P ts*)
then have *falsifiesl G* (*Pos P* (*ts*{σ}$_{ts}$)) **by** *auto*
then obtain *i ts'* **where** *i-ts'*:
i < *length G*
∧ *G* ! *i* = *False*
∧ *diag-fatom i* = *Pos P ts'*
∧ *instance-ofts ts'* (*ts* {σ}$_{ts}$) **by** *auto*
moreover
have *instance-ofts* (*ts* {σ}$_{ts}$) *ts* **unfolding** *instance-ofts-def* **by** *auto*
then have *instance-ofts ts' ts* **using** *i-ts' instance-ofts-trans* **by** *auto*
ultimately
have
i < *length G*
∧ *G* ! *i* = *False*
∧ *diag-fatom i* = *Pos P ts'*
∧ *instance-ofts ts' ts* **by** *auto*

```
  then show ?case by auto
next
  case (Neg P ts)
  then have falsifiesl G (Neg P (ts{σ}ts)) by auto
  then obtain i ts' where i-ts':
      i < length G
      ∧ G ! i = True
      ∧ diag-fatom i = Pos P ts'
      ∧ instance-ofts ts' (ts {σ}ts) by auto
  moreover
  have instance-ofts (ts {σ}ts) ts unfolding instance-ofts-def by auto
  then have instance-ofts ts' ts using i-ts' instance-ofts-trans by auto
  ultimately
  have
      i < length G
      ∧ G ! i = True
      ∧ diag-fatom i = Pos P ts'
      ∧ instance-ofts ts' ts by auto
  then show ?case by auto
qed

lemma partial-equiv-subst:
  assumes asm: falsifiesc G ((C :: fterm clause) {σ}ls)
  shows falsifiesc G C
proof
  fix l
  assume l ∈ C
  then have falsifiesl G (l {σ}l) using asm by auto
  then show falsifiesl G l using partial-equiv-subst' by auto
qed



lemma sub-of-denot-equiv-ground:
  ((∃ l ∈ C. evall E HFun G l) ⟷ (∃ l ∈ C {sub-of-denot E}ls. evall E HFun G l))
        ∧ groundls (C {sub-of-denot E}ls)
  using sub-of-denot-equiv-ground' by auto
```

C.10.1 Semantic Trees

```
abbreviation closed-branch :: partial-pred-denot ⇒ tree ⇒ fterm clause set ⇒ bool
where
  closed-branch G T Cs ≡ branch G T ∧ falsifiescs G Cs

abbreviation open-branch :: partial-pred-denot ⇒ tree ⇒ fterm clause set ⇒ bool
where
  open-branch G T Cs ≡ branch G T ∧ ¬falsifiescs G Cs
```

fun *closed-tree* :: *tree* ⇒ *fterm clause set* ⇒ *bool* **where**
closed-tree T Cs ⟷ *anybranch T* (λ*b*. *closed-branch b T Cs*)
∧ *anyinternal T* (λ*p*. ¬*falsifiescs p Cs*)

C.11 Herbrand's Theorem

lemma *maximum*:
assumes *asm*: *finite C*
shows ∃ *n* :: *nat*. ∀ *l* ∈ *C*. *f l* ≤ *n*
proof
from *asm* **show** ∀ *l*∈*C*. *f l* ≤ (*Max* (*f* ‘ *C*)) **by** *auto*
qed

lemma *extend-preserves-model*:
assumes *f-chain*: *list-chain* (*f* :: *nat* ⇒ *partial-pred-denot*)
assumes *n-max*: ∀ *l*∈*C*. *undiag-fatom l* ≤ *n*
assumes *C-ground*: *groundls C*
assumes *C-false*: ¬*evalc HFun* (*extend f*) *C*
shows *falsifiesc* (*f* (*Suc n*)) *C*
proof
let *?F* = *HFun*
let *?G* = *extend f*

fix *l*
assume *asm*: *l*∈*C*
let *?i* = *undiag-fatom l*
from *asm* **have** *i-n*: *?i* ≤ *n* **using** *n-max* **by** *auto*
then have *j-n*: *?i* ≤ *length* (*f n*) **using** *f-chain chain-length*[*of f n*] **by** *auto*

from *C-false* **have** ¬(∀ *E*. ∃ *l* ∈ *C*. *evall E ?F ?G l*) **unfolding** *evalc-def* **by** *auto*
then have ∃ *E*. ∀ *l* ∈ *C*. ¬ *evall E ?F ?G l* **by** *auto*
then have ∀ *E*. ∀ *l* ∈ *C*. ¬ *evall E ?F ?G l* **using** *C-ground ground-var-denot* **by** *blast*
then have *last*: ∀ *E*. ¬ *evall E ?F ?G l* **using** *asm* **by** *blast*

then show *falsifiesl* (*f* (*Suc n*)) *l*
proof (*cases l*)
case (*Pos P ts*)
from *Pos asm C-ground* **have** *ts-ground*: *grounds ts* **by** *auto*
from *Pos asm C-ground* **have** *undiag-l*: *undiag-hatom* (*hlit-of-flit l*) = *?i* **using** *ground-h-undiag* **by** *blast*

from *last* **have** ¬*?G P* (*hterms-of-fterms ts*) **using** *evall-grounds*[*of ts - ?G P*] *ts-ground Pos* **by** *auto*
then have *f* (*Suc ?i*) ! *?i* = *False* **using** *Pos undiag-l* **by** *auto*

```
    moreover
    have f (Suc ?i) ! ?i = f (Suc n) ! ?i
      using f-chain i-n j-n chain-length[of f] ith-in-extension[of f] by simp
    ultimately have f (Suc n) ! ?i = False by auto
    then have
    ?i < length (f (Suc n)) (* j-n *)
    ∧ f (Suc n) ! ?i = False (*last thing *)
    ∧ diag-fatom ?i = Pos P ts (* by definition of ?i *)
    ∧ instance-ofts ts ts
      using
        j-n ts-ground diag-undiag-fatom instance-ofts-self f-chain chain-length[of f]
Pos
      by auto
    then show ?thesis using Pos by auto
  next
    case (Neg P ts)
    from Neg asm C-ground have ts-ground: grounds ts by auto
    from Neg asm C-ground have undiag-l: undiag-hatom (hlit-of-flit l) = ?i using
ground-h-undiag by blast

     from last have ?G P (hterms-of-fterms ts) using evall-grounds[of ts - ?G P]
C-ground asm Neg by auto
    then have f (Suc ?i) ! ?i = True using Neg undiag-neg undiag-l
      by (metis hatom-of-fatom.simps undiag-fatom-def)
    moreover
    have f (Suc ?i) ! ?i = f (Suc n) ! ?i
      using f-chain i-n j-n chain-length[of f] ith-in-extension[of f] by simp
    ultimately have f (Suc n) ! ?i = True by auto
    then have
    ?i < length (f (Suc n)) (* j-n *)
    ∧ f (Suc n) ! ?i = True (*last thing *)
    ∧ diag-fatom ?i = Pos P ts (* by definition of ?i *)
    ∧ instance-ofts ts ts
      using j-n diag-undiag-fatom instance-ofts-self[of ts] f-chain chain-length[of f]
Neg undiag-neg ts-ground
      by auto
    then show ?thesis using Neg by auto
  qed
qed



lemma list-chain-model:
  assumes f-chain: list-chain (f :: nat ⇒ partial-pred-denot)
  assumes model-cs: ∀ n. ¬falsifiescs (f n) Cs
  assumes fin-cs: finite Cs
  assumes fin-c: ∀ C ∈ Cs. finite C
```

```
  shows ∃ G. evalcs HFun G Cs
proof
  let ?F = HFun
  let ?G = extend f

  have ∀ C E. (C ∈ Cs ⟶ (∃ l ∈ C. evall E ?F ?G l))
    proof (rule allI; rule allI; rule impI)
      fix C
      fix E
      assume asm: C ∈ Cs
      let ?σ = sub-of-denot E
      have groundcσ: groundls (C {?σ}ls) using sub-of-denot-equiv-ground by auto
      from fin-c asm have finite (C {?σ}ls) by auto
       then obtain n where largest: ∀ l ∈ (C {?σ}ls). undiag-fatom l ≤ n using
maximum by blast
      from model-cs asm have ¬falsifiesc (f (Suc n)) C by auto
     then have model-c: ¬falsifiesc (f (Suc n)) (C {?σ}ls) using partial-equiv-subst
by blast

      have evalc HFun ?G (C {?σ}ls)
        using groundcσ f-chain largest model-c
              extend-preserves-model[of f C {?σ}ls n] by blast
     then have ∀ E. ∃ l ∈ (C {?σ}ls). evall E ?F ?G l unfolding evalc-def by auto
     then have ∃ l ∈ (C {?σ}ls). evall E ?F ?G l by auto
     then show ∃ l∈C. evall E ?F ?G l using sub-of-denot-equiv-ground by simp

   qed
  then have ∀ C ∈ Cs. ∀ E. ∃ l ∈ C. evall E ?F ?G l by auto
  then have ∀ C ∈ Cs. evalc ?F ?G C unfolding evalc-def by auto
  then show evalcs ?F ?G Cs unfolding evalcs-def by auto
qed


fun deeptree :: nat ⇒ tree where
  deeptree 0 = Leaf
| deeptree (Suc n) = Branch (deeptree n) (deeptree n)

thm extend-preserves-model
thm list-chain-model


lemma branch-length: branch b (deeptree n) ⟹ length b = n
proof (induction n arbitrary: b)
  case 0 then show ?case using branch-inv-Leaf by auto
next
  case (Suc n)
  then have branch b (Branch (deeptree n) (deeptree n)) by auto
```

then obtain $a\ b'$ **where** p: $b{=}a\#b' \wedge$ *branch* b' (*deeptree* n) **using** *branch-inv-Branch*[*of* b] **by** *blast*
then have *length* $b' = n$ **using** *Suc* **by** *auto*
then show *?case* **using** p **by** *auto*
qed

lemma *infinity*:
assumes *bij*: $\forall\, n :: nat.\ undiago\ (diago\ n) = n$
assumes *all-tree*: $\forall\, n :: nat.\ (diago\ n) \in tree$
shows $\neg$*finite tree*
proof −
from *bij all-tree* **have** $\forall\, n.\ n = undiago\ (diago\ n) \wedge (diago\ n) \in tree$ **by** *auto*
then have $\forall\, n.\ \exists\, ds.\ n = undiago\ ds \wedge ds \in tree$ **by** *auto*
then have *undiago* ‘ *tree* = (*UNIV* :: *nat set*) **by** *auto*
then have $\neg$*finite tree***by** (*metis finite-imageI infinite-UNIV-nat*)
then show *?thesis* **by** *auto*
qed

lemma *longer-falsifies′*:
falsifiesl ds l $\Longrightarrow$ *falsifiesl* (*ds*@*d*) *l*
proof (*induction l*)
case (*Pos P ts*)
then obtain *i ts′* **where** *i-ts′*:
i < *length ds*
$\wedge$ *ds* ! *i* = *False*
$\wedge$ *diag-fatom i* = *Pos P ts′*
$\wedge$ *instance-ofts ts′ ts* **by** *auto*
moreover
from *i-ts′* **have** *i* < *length* (*ds*@*d*) **by** *auto*
moreover
from *i-ts′* **have** (*ds*@*d*) ! *i* = *False* **by** (*simp add*: *nth-append*)
ultimately
have
i < *length* (*ds*@*d*)
$\wedge$ (*ds*@*d*) ! *i* = *False*
$\wedge$ *diag-fatom i* = *Pos P ts′*
$\wedge$ *instance-ofts ts′ ts* **by** *auto*
then show *?case* **by** *auto*
next
case (*Neg P ts*)
then obtain *i ts′* **where** *i-ts′*:
i < *length ds*
$\wedge$ *ds* ! *i* = *True*
$\wedge$ *diag-fatom i* = *Pos P ts′*
$\wedge$ *instance-ofts ts′ ts* **by** *auto*

```
  moreover
  from i-ts' have i < length (ds@d) by auto
  moreover
  from i-ts' have (ds@d) ! i = True by (simp add: nth-append)
  ultimately
  have
      i < length (ds@d)
      ∧ (ds@d) ! i = True
      ∧ diag-fatom i = Pos P ts'
      ∧ instance-ofts ts' ts by auto
  then show ?case by auto
qed


lemma longer-falsifies:
  assumes asm: falsifiescs ds Cs
  shows falsifiescs (ds @ d) Cs
proof −
  from asm obtain C where C-p: C ∈ Cs ∧ falsifiesc ds C by auto
  then have ∀ l ∈ C. falsifiesl ds l by auto
  then have ∀ l ∈ C. falsifiesl (ds@d) l using longer-falsifies' by auto
  then have falsifiesc (ds @ d) C by auto
  then show falsifiescs (ds @ d) Cs using C-p by auto
qed

theorem herbrand':
  assumes openb: ∀ T. ∃ G. open-branch G T Cs
  assumes finite-cs: finite Cs ∀ C∈Cs. finite C
  shows ∃ G. evalcs HFun G Cs
proof −

  let ?tree = {G. ¬falsifiescs G Cs}
  let ?undiag = length
  let ?diag = (λl. SOME b. open-branch b (deeptree l) Cs) :: nat ⇒ partial-pred-denot

  from openb have diag-open: ∀ l. open-branch (?diag l) (deeptree l) Cs
    using someI-ex[of %b. open-branch b (deeptree -) Cs] by auto
  then have ∀ n. ?undiag (?diag n) = n using branch-length by auto
  moreover
  have ∀ n. (?diag n) ∈ ?tree using diag-open by auto
  ultimately
  have ¬finite ?tree using infinity[of - λn. SOME b. open-branch b (- n) Cs] by simp

  moreover
  have ∀ ds d. ¬falsifiescs (ds @ d) Cs ⟶ ¬falsifiescs ds Cs
    using longer-falsifies[of Cs] by blast
  then have (∀ ds d. ds @ d ∈ ?tree ⟶ ds ∈ ?tree) by auto
```

ultimately
have $\exists\, c.$ *list-chain* $c \wedge (\forall\, n.\ c\ n \in$ *?tree*$)$ **using** *konig*[*of ?tree*] **by** *blast*
then have $\exists\, G.$ *list-chain* $G \wedge (\forall\, n.\ \neg$ *falsifiescs* $(G\ n)$ *Cs*$)$ **by** *auto*

then show $\exists\, G.$ *evalcs HFun G Cs* **using** *list-chain-model finite-cs* **by** *auto*
qed

theorem *herbrand′-contra*:
assumes *finite-cs*: *finite Cs* $\forall\, C{\in}Cs.$ *finite C*
assumes *unsat*: $\forall\, G.\ \neg$*evalcs HFun G Cs*
shows $\exists\, T.\ \forall\, G.$ *branch G T* $\longrightarrow$ *closed-branch G T Cs*
proof −
from *finite-cs unsat* **have** $\forall\, T.\ \exists\, G.$ *open-branch G T Cs* $\Longrightarrow \exists\, G.$ *evalcs HFun G Cs* **using** *herbrand′* **by** *blast*
then show *?thesis* **using** *unsat* **by** *blast*
qed

theorem *herbrand*:
assumes *unsat*: $\forall\, G.\ \neg$ *evalcs HFun G Cs*
assumes *finite-cs*: *finite Cs* $\forall\, C{\in}Cs.$ *finite C*
shows $\exists\, T.$ *closed-tree T Cs*
proof −
from *unsat finite-cs* **obtain** *T* **where** *anybranch T* $(\lambda b.$ *closed-branch b T Cs*$)$ **using** *herbrand′-contra*[*of Cs*] **by** *blast*
then have $\exists\, T.$ *anybranch T* $(\lambda p.$ *falsifiescs p Cs*$) \wedge$ *anyinternal T* $(\lambda p.\ \neg$ *falsifiescs p Cs*$)$
using *cutoff-branch-internal*[*of T* $(\lambda p.$ *falsifiescs p Cs*$)$] **by** *blast*
then show *?thesis* **by** *auto*
qed

C.12 Lifting Lemma

lemma *lifting*:
assumes *appart*: *varsc c* $\cap$ *varsc d* $= \{\}$
assumes $inst_1$: *instance-ofc* c' *c*
assumes $inst_2$: *instance-ofc* d' *d*
assumes *appl*: *applicable* c' d' l' m' σ
shows $\exists\, l\ m\ \tau.$ *applicable c d l m* $\tau \wedge$
instance-ofc (*resolution* c' d' l' m' σ) (*resolution c d l m* τ)
oops

C.13 Completeness

lemma *falsifiesc* [] *C* $\Longrightarrow$ *C* $= \{\}$

proof −
  { fix *l*
    assume $l \in C$
    moreover
    have ¬*falsifiesl* [] *l* **by** (*cases l*) *auto*
    ultimately have ¬*falsifiesc* [] *C* **by** *auto*
  } then show *falsifiesc* [] *C* $\Longrightarrow$ $C = \{\}$ **by** *auto*
qed

theorem *completeness′*:
  assumes *finite-cs*: *finite Cs* $\forall\, C{\in}Cs.$ *finite C*
  shows *closed-tree T Cs* $\Longrightarrow \exists\, Cs'.$ *resolution-deriv Cs Cs′* $\wedge$ $\{\} \in Cs'$
proof (*induction T arbitrary*: *Cs rule*: *Nat.measure-induct-rule*[*of size*])
  fix *T*::*tree*
  fix *Cs* :: *fterm clause set*
  assume ($\bigwedge T'$ *Cs*. *size T′* < *size T* $\Longrightarrow$
                closed-tree T′ Cs $\Longrightarrow \exists\, Cs'.$ *resolution-deriv Cs Cs′* $\wedge$ $\{\} \in Cs'$)
  assume *closed-tree T Cs*
  have *True* **by** *auto*
  then show $\exists\, Cs'.$ *resolution-deriv Cs Cs′* $\wedge$ $\{\} \in Cs'$ **oops**

theorem *completeness*:
  assumes *finite-cs*: *finite Cs* $\forall\, C{\in}Cs.$ *finite C*
  assumes *unsat*: $\forall\, F\ G.$ ¬*evalcs F G Cs*
  shows $\exists\, Cs'.$ *resolution-deriv Cs Cs′* $\wedge$ $\{\} \in Cs'$
oops

end

Appendix D

Formalization Code: Examples.thy

```
theory Examples imports Resolution begin
value Var ''x''
value Fun ''one'' []
value Fun ''mul'' [Var ''y'',Var ''y'']
value Fun ''add'' [Fun ''mul'' [Var ''y'',Var ''y''], Fun ''one'' []]
value Pos ''greater'' [Var ''x'', Var ''y'']
value Neg ''less'' [Var ''x'', Var ''y'']
value Pos ''less'' [Var ''x'', Var ''y'']
value Pos ''equals''
      [Fun ''add''[Fun ''mul''[Var ''y'',Var ''y''], Fun ''one''[]],Var ''x'']

fun F_nat :: nat fun-denot where
  F_nat f [n,m] =
    (if f = ''add'' then n + m else
     if f = ''mul'' then n * m else 0)
| F_nat f [] =
    (if f = ''one''  then 1 else
     if f = ''zero'' then 0 else 0)
| F_nat f us = 0

fun G_nat :: nat pred-denot where
  G_nat p [x,y] =
    (if p = ''less'' ∧ x < y then True else
```

```
    if p = ''greater'' ∧ x > y then True else
    if p = ''equals'' ∧ x = y then True else False)
| G_nat p us = False

fun E_nat :: nat var-denot where
  E_nat x =
    (if x = ''x'' then 26 else
     if x = ''y'' then 5 else 0)

lemma evalt E_nat F_nat (Var ''x'') = 26
  by auto

lemma evalt E_nat F_nat (Fun ''one'' []) = 1
  by auto

lemma evalt E_nat F_nat (Fun ''mul'' [Var ''y'',Var ''y'']) = 25
  by auto

lemma
  evalt E_nat F_nat (Fun ''add'' [Fun ''mul'' [Var ''y'',Var ''y''], Fun ''one'' []]) = 26
  by auto

lemma evall E_nat F_nat G_nat (Pos ''greater'' [Var ''x'', Var ''y'']) = True
  by auto

lemma evall E_nat F_nat G_nat (Neg ''less'' [Var ''x'', Var ''y'']) = True
  by auto

lemma evall E_nat F_nat G_nat (Pos ''less'' [Var ''x'', Var ''y'']) = False
  by auto

lemma evall E_nat F_nat G_nat
    (Pos ''equals''
      [Fun ''add'' [Fun ''mul'' [Var ''y'',Var ''y''],Fun ''one'' []]
      ,Var ''x'']
    ) = True
  by auto

definition PP :: fterm literal where
  PP = Pos ''P'' [Fun ''c'' []]


definition PQ :: fterm literal where
  PQ = Pos ''Q'' [Fun ''d'' []]


definition NP :: fterm literal where
```

NP = *Neg* ''*P*'' [*Fun* ''*c*'' []]

definition *NQ* :: *fterm literal* **where**
NQ = *Neg* ''*Q*'' [*Fun* ''*d*'' []]

theorem *empty-mgu*: *unifierls* ε *L* $\Longrightarrow$ *mguls* ε *L*
unfolding *unifierls-def mguls-def* **apply** *auto*
apply (*rule-tac x*=*u* **in** *exI*)
using *empty-comp1 empty-comp2* **apply** (*auto*)
done

theorem *unifier-single*: *unifierls* σ {*l*}
unfolding *unifierls-def* **by** *auto*

theorem *resolution-rule'*:
$C_1 \in Cs \Longrightarrow C_2 \in Cs \Longrightarrow$ *applicable* C_1 C_2 L_1 L_2 σ
$\Longrightarrow C =$ {*resolution* C_1 C_2 L_1 L_2 σ}
$\Longrightarrow$ *resolution-step Cs* ($Cs \cup C$)
using *resolution-rule* **by** *auto*

lemma *resolution-example1*:
$\exists$ *Cs*. *resolution-deriv* {{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*}}
{{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*},{*NP*},{*PP*},{}}
proof −
have *resolution-step*
{{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*}}
({{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*}} ∪ {{*NP*}})
apply (*rule resolution-rule'*[*of* {*NP*,*PQ*} - {*NQ*} {*PQ*} {*NQ*} ε])
unfolding *applicable-def varsls-def varsl-def*
NQ-def NP-def PQ-def PP-def resolution-def
using *unifier-single empty-mgu* **apply** (*auto*)
done
then have *resolution-step*
{{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*}}
({{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*},{*NP*}})
by (*simp add*: *insert-commute*)
moreover
have *resolution-step*
{{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*},{*NP*}}
({{*NP*,*PQ*},{*NQ*},{*PP*,*PQ*},{*NP*}} ∪ {{*PP*}})
apply (*rule resolution-rule'*[*of* {*NQ*} - {*PP*,*PQ*} {*NQ*} {*PQ*} ε])
unfolding *applicable-def varsls-def varsl-def*
NQ-def NP-def PQ-def PP-def resolution-def
using *unifier-single empty-mgu* **apply** (*auto*)
done
then have *resolution-step*

```
        {{NP,PQ},{NQ},{PP,PQ},{NP}}
       ({{NP,PQ},{NQ},{PP,PQ},{NP},{PP}})
    by (simp add: insert-commute)
  moreover
  have resolution-step
        {{NP,PQ},{NQ},{PP,PQ},{NP},{PP}}
       ({{NP,PQ},{NQ},{PP,PQ},{NP},{PP}} ∪ {{}})
    apply (rule resolution-rule'[of {NP} - {PP} {NP} {PP} ε])
    unfolding applicable-def varsls-def  varsl-def
              NQ-def NP-def PQ-def PP-def resolution-def
    using unifier-single empty-mgu apply (auto)
    done
  then have resolution-step
        {{NP,PQ},{NQ},{PP,PQ},{NP},{PP}}
       ({{NP,PQ},{NQ},{PP,PQ},{NP},{PP},{}})
    by (simp add: insert-commute)
  ultimately
  have resolution-deriv {{NP,PQ},{NQ},{PP,PQ}}
                        {{NP,PQ},{NQ},{PP,PQ},{NP},{PP},{}}
    unfolding resolution-deriv-def using star.intros[of resolution-step] by auto
  then show ?thesis by auto
qed

definition Pa :: fterm literal where
  Pa = Pos ''a'' []


definition Na :: fterm literal where
  Na = Neg ''a'' []


definition Pb :: fterm literal where
  Pb = Pos ''b'' []


definition Nb :: fterm literal where
  Nb = Neg ''b'' []


definition Paa :: fterm literal where
  Paa = Pos ''a'' [Fun ''a'' []]


definition Naa :: fterm literal where
  Naa = Neg ''a'' [Fun ''a'' []]
```

definition *Pax* :: *fterm literal* **where**
Pax = *Pos* ''*a*'' [*Var* ''*x*'']

definition *Nax* :: *fterm literal* **where**
Nax = *Neg* ''*a*'' [*Var* ''*x*'']

definition *mguPaaPax* :: *substitution* **where**
mguPaaPax = (λ*x*. *if x* = ''*x*'' *then Fun* ''*a*'' [] *else Var x*)

lemma *mguPaaPax-mgu*: *mguls mguPaaPax* {*Paa*,*Pax*}
proof −
let *?σ* = λ*x*. *if x* = ''*x*'' *then Fun* ''*a*'' [] *else Var x*
have *a*: *unifierls* (λ*x*. *if x* = ''*x*'' *then Fun* ''*a*'' [] *else Var x*) {*Paa*,*Pax*} **unfolding** *Paa-def Pax-def unifierls-def* **by** *auto*
have *b*: ∀ *u*. *unifierls u* {*Paa*,*Pax*} ⟶ (∃ *i*. *u* = *?σ* · *i*)
proof (*rule*;*rule*)
fix *u*
assume *unifierls u* {*Paa*,*Pax*}
then have *uuu*: *u* ''*x*'' = *Fun* ''*a*'' [] **unfolding** *unifierls-def Paa-def Pax-def* **by** *auto*
have *?σ* · *u* = *u*
proof
fix *x*
{
assume *x*=''*x*''
moreover
have (*?σ* · *u*) ''*x*'' = *Fun* ''*a*'' [] **unfolding** *composition-def* **by** *auto*
ultimately have (*?σ* · *u*) *x* = *u x* **using** *uuu* **by** *auto*
}
moreover
{
assume *x*≠''*x*''
then have (*?σ* · *u*) *x* = (*ε x*) {*u*}$_t$ **unfolding** *composition-def* **by** *auto*
then have (*?σ* · *u*) *x* = *u x* **by** *auto*
}
ultimately show (*?σ* · *u*) *x* = *u x* **by** *auto*
qed
then have (∃ *i*. *?σ* · *i* = *u*) **by** *auto*
then show (∃ *i*. *u* = *?σ* · *i*) **by** *auto*
qed
from *a b* **show** *?thesis* **unfolding** *mguls-def* **unfolding** *mguPaaPax-def* **by** *auto*
qed

lemma *resolution-example2*:
∃ *Cs*. *resolution-deriv* {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}

$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\},\{\}\}$
proof −
have *resolution-step*
$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}$
$(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\} \cup \{\{Na,Pb\}\})$
apply (*rule resolution-rule'*[*of* {*Pax*} - {*Na*,*Pb*,*Naa*} {*Pax*} {*Naa*} *mguPaaPax*
])
using *mguPaaPax-mgu* **unfolding** *applicable-def varsls-def varsl-def*
Nb-def Na-def Pax-def Pa-def Pb-def Naa-def Paa-def mguPaaPax-def
resolution-def
apply *auto*
done
then have *resolution-step*
$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}$
$(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\})$
by (*simp add*: *insert-commute*)
moreover
have *resolution-step*
$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\}$
$(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\} \cup \{\{Na\}\})$
apply (*rule resolution-rule'*[*of* {*Nb*,*Na*} - {*Na*,*Pb*} {*Nb*} {*Pb*} ε])
unfolding *applicable-def varsls-def varsl-def*
Pb-def Nb-def Na-def PP-def resolution-def
using *unifier-single empty-mgu* **apply** (*auto*)
done
then have *resolution-step*
$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\}\}$
$(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\}\})$
by (*simp add*: *insert-commute*)
moreover
have *resolution-step*
$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\}\}$
$(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\}\} \cup \{\{\}\})$
apply (*rule resolution-rule'*[*of* {*Na*} - {*Pa*} {*Na*} {*Pa*} ε])
unfolding *applicable-def varsls-def varsl-def*
Pa-def Nb-def Na-def PP-def resolution-def
using *unifier-single empty-mgu* **apply** (*auto*)
done
then have *resolution-step*
$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\}\}$
$(\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\},\{\}\})$
by (*simp add*: *insert-commute*)
ultimately
have *resolution-deriv* $\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\}\}$
$\{\{Nb,Na\},\{Pax\},\{Pa\},\{Na,Pb,Naa\},\{Na,Pb\},\{Na\},\{\}\}$
unfolding *resolution-deriv-def* **using** *star.intros*[*of resolution-step*] **by** *auto*
then show *?thesis* **by** *auto*

qed

lemma *ref-sound*:
assumes *deriv*: *resolution-deriv Cs Cs′* ∧ {} ∈ *Cs′*
shows ¬*evalcs F G Cs*
proof −
from *deriv* **have** *evalcs F G Cs* ⟹ *evalcs F G Cs′* **using** *sound-derivation* **by** *auto*
moreover
from *deriv* **have** *evalcs F G Cs′* ⟹ *evalc F G* {} **unfolding** *evalcs-def* **by** *auto*
moreover
then have *evalc F G* {} ⟹ *False* **unfolding** *evalc-def* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *resolution-example1-sem*: ¬*evalcs F G* {{*NP*, *PQ*}, {*NQ*}, {*PP*, *PQ*}}
using *resolution-example1 ref-sound* **by** *auto*

lemma *resolution-example2-sem*: ¬*evalcs F G* {{*Nb*,*Na*},{*Pax*},{*Pa*},{*Na*,*Pb*,*Naa*}}
using *resolution-example2 ref-sound* **by** *auto*

end

Appendix E

Chang and Lee's Lifting Lemma

Here follows an account of flaws and imprecisions in Chang and Lee's proof of the lifting lemma [CL73]. It also presents a counter-example to the proof. Their notation uses $\sim$ to give the complement literal and $\circ$ for composition of substitutions. Like Ben-Ari, they represent substitutions as finite sets of key-value pairs and can thus apply the $\cup$ operator to combine them.

Lemma 5.1 (Lifting Lemma) If C_1' and C_2' are instances of C_1 and C_2, respectively, and if C' is a resolvent of C_1' and C_2', then there is a resolvent C of C_1 and C_2 such that C' is an instance of C.

Proof We rename, if necessary, the variables in C_1 and C_2 so that variables in C_1 are all different from those in C_2. Let L_1' and L_2' be the literals resolved upon, and let

$$C' = (C_1'\gamma - L_1'\gamma) \cup (C_2'\gamma - L_2'\gamma),$$

where γ is a most general unifier of L_1' and $\sim L_2'$.

This is problematic in the binary resolution with factoring of their book. The reason is that in the proof it is assumed that no factoring is done on C_1' and C_2',

but factoring is actually allowed by the resolution calculus of the book. One could argue that it is acceptable, since factoring is just further instantiation and thus we could perhaps just obtain the factored C_1' and C_2'. However, we will see in the counter-example that this handling of factors is problematic.

> Since C_1' and C_2' are instances of C_1 and C_2, respectively, there is a substitution θ such that $C_1' = C_1\theta$ and $C_2' = C_2\theta$. Let $L_i^1, ..., L_i^{r_i}$ be the literals in C_i corresponding to L_i' (i.e., $L_i^1\theta = \cdots = L_i^{r_i}\theta = L_i'$), $i = 1, 2$. If $r_i > 1$, obtain a most general unifier λ_i, for $\{L_i^1, ..., L_i^{r_i}\}$ and let $L_i = L_i^1\lambda_i$, $i = l, 2$. (Note that $L_i^1\lambda_i, ..., L_i^{r_i}\lambda_i$ are the same, since λ_i is a most general unifier.) Then L_i, is a literal in the factor $C_i\lambda_i$ of C_i. If $r_i = 1$, let $\lambda_i = \epsilon$ and $L_i = L_i^1\lambda_i$.
>
> Let $\lambda = \lambda_1 \cup \lambda_2$.

Like in the proof of Ben-Ari, we should make sure that the union $\lambda_1 \cup \lambda_2$ is well-formed. Again we could require additional properties of the mgus λ_1 and λ_2.

> Thus, clearly L_i' is an instance of L_i. Since L_i' and $\sim L_2'$ are unifiable, L_1 and $\sim L_2$ are unifiable.

L_i' should probably have been L_1' in the last sentence.

> Let σ be a most general unifier of L_1 and $\sim L_2$. Let
>
> $$\begin{aligned} C &= ((C_1\lambda)\sigma - L_1\sigma) \cup ((C_2\lambda)\sigma - L_2\sigma) \\ &= ((C_1\lambda)\sigma - (\{L_1^1, ..., L_1^{r_1}\}\lambda)\sigma) \cup ((C_2\lambda)\sigma - (\{L_2^1, ..., L_2^{r_2}\}\lambda)\sigma) \\ &= (C_1(\lambda \circ \sigma) - \{L_1^1, ..., L_1^{r_1}\}(\lambda \circ \sigma)) \cup (C_2(\lambda \circ \sigma) - \{L_2^1, ..., L_2^{r_2}\}(\lambda \circ \sigma)). \end{aligned}$$
>
> C is a resolvent of C_l and C_2. Clearly, C' is an instance of C since
>
> $$\begin{aligned} C' &= (C_1'\gamma - L_1'\gamma) \cup (C_2'\gamma - L_2'\gamma) \\ &= ((C_1\theta)\gamma - (\{L_1^1, ..., L_1^{r_1}\}\theta)\gamma) \cup ((C_2\theta)\gamma - (\{L_2^1, ..., L_2^{r_2}\}\theta)\gamma) \\ &= (C_1(\theta \circ \gamma) - \{L_1^1, ..., L_1^{r_1}\}(\theta \circ \gamma)) \cup (C_2(\theta \circ \gamma) - \{L_2^1, ..., L_2^{r_2}\}(\theta \circ \gamma)) \end{aligned}$$
>
> and $\lambda \circ \sigma$ is more general than $\theta \circ \gamma$. Thus we complete the proof of this lemma.

Realizing that $\lambda \circ \sigma$ is more general than $\theta \circ \gamma$ is left to the reader, but I do not find it obvious.

I do not find it clear that C' is an instance of C. Like the last proof, I have made a unsuccessful proof attempt:

Since $\lambda \circ \sigma$ is more general than $\theta \circ \gamma$ we can obtain substitution u where $\theta \circ \gamma = (\lambda \circ \sigma) \circ u$. Then

$$
\begin{aligned}
C' &= (C_1(\theta \circ \gamma) - \{L_1^1, ..., L_1^{r_1}\}(\theta \circ \gamma)) \cup (C_2(\theta \circ \gamma) - \{L_2^1, ..., L_2^{r_2}\}(\theta \circ \gamma)) \\
&= (C_1((\lambda \circ \sigma) \circ u) - \{L_1^1, ..., L_1^{r_1}\}((\lambda \circ \sigma) \circ u)) \\
&\quad \cup (C_2((\lambda \circ \sigma) \circ u) - \{L_2^1, ..., L_2^{r_2}\}((\lambda \circ \sigma) \circ u)) \\
&= ((C_1(\lambda \circ \sigma))u - (\{L_1^1, ..., L_1^{r_1}\}(\lambda \circ \sigma))u) \\
&\quad \cup ((C_2(\lambda \circ \sigma))u - (\{L_2^1, ..., L_2^{r_2}\}(\lambda \circ \sigma))u) \\
&= (C_1(\lambda \circ \sigma) - \{L_1^1, ..., L_1^{r_1}\}(\lambda \circ \sigma))u \cup (C_2(\lambda \circ \sigma) - \{L_2^1, ..., L_2^{r_2}\}(\lambda \circ \sigma))u \\
&= ((C_1(\lambda \circ \sigma) - \{L_1^1, ..., L_1^{r_1}\}(\lambda \circ \sigma)) \cup (C_2(\lambda \circ \sigma) - \{L_2^1, ..., L_2^{r_2}\}(\lambda \circ \sigma)))u \\
&= Cu
\end{aligned}
$$

But here I used $(A - B)\sigma = A\sigma - B\sigma$ again, which does not hold.

An example

Leitsch found a counter-example to the proof that illustrates its problems with factors [Lei89]. We present this counter-example, and try to follow the proof:

Let $C_1 = \{P(y), P(x)\}$, $C_2 = \{\sim P(a)\}$, $C_1' = \{P(a), P(x)\}$, $C_2 = \{\neg P(a)\}$, $C' = \{\}$. Notice that C' can indeed be obtained as a resolvent of C_1' and C_2' since $\{P(x)\}$ is a factor of C_1' and then when we resolve that with C_2' we get $\{\}$. We let L_1 and L_2 be the literals resolved upon, i.e. $L_1 = P(x)$ and $L_2 = \sim P(a)$. A most general unifier is $\gamma = \{x \leftarrow a\}$. We also obtain the substitution that made the instances of C_1 and C_2 i.e. $\theta = \{y \leftarrow a\}$ and indeed $C_1\theta = C_1'$ and $C_2\theta = C_2'$. We now obtain all the literals in C_1 that turn in to L_1 when θ is applied, that is, only $L_1^1 = P(x)$, likewise for C_2 and L_2 we get only $L_2^1 = \sim P(a)$. Since there was only one of each we get $\lambda_1 = \epsilon$ and $\lambda_2 = \epsilon$ which again means $\lambda = \epsilon$. Now we can get $L_1 = L_1^1\lambda = P(x)$ and $L_2 = L_2^1\lambda = \sim P(a)$. Finally, we obtain an mgu σ of L_1 and $\sim L_2$ namely $\sigma = \{x \leftarrow a\}$. Now we can construct

C:

$$\begin{aligned} C &= (C_1\lambda\sigma - L_1\sigma) \cup (C_2\lambda\sigma - L_2\sigma) \\ &= (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma) \\ &= (\{P(y), P(a)\} - P(a)) \cup (\{\sim P(a)\} - \sim P(a)) \\ &= \{P(y)\} \end{aligned}$$

But clearly, $C' = \{\}$ is not an instance of $C = \{P(y)\}$.

Summary

This proof contains some mistakes and imprecisions that are similar to those of the first one:

- C' may have come from doing factoring on C'_1 or C'_2 before the resolution.
- λ is not necessarily well-formed.
- It is not clear that $\lambda \circ \sigma$ is more general than $\theta \circ \gamma$.
- It is not clear that C' is an instance of C.

Bibliography

[BA12] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 3rd edition, 2012.

[Ber07] Stefan Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, 2007. `http://afp.sf.net/entries/FOL-Fitting.shtml`, Formal proof development.

[BG01] Leo Bachmair and Harald Ganzinger. Chapter 2 - resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 19 – 99. North-Holland, Amsterdam, 2001.

[Bla15] Jasmin Christian Blanchette. Hammering away: A user's guide to Sledgehammer for Isabelle/HOL. May 2015. `https://isabelle.in.tum.de/doc/sledgehammer.pdf`.

[BPT14a] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Abstract completeness. *Archive of Formal Proofs*, April 2014. `http://afp.sf.net/entries/Abstract_Completeness.shtml`, Formal proof development.

[BPT14b] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Unified classical logic completeness - a coinductive pearl. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 46–60. Springer International Publishing, 2014.

[Cha15] Amine Chaieb. The pythagorean theorem. *The Isabelle2015 Library*, May 2015. `https://isabelle.in.tum.de/website-Isabelle2015/dist/library/HOL/HOL-ex/Pythagoras.html`.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1st edition, 1973.

[CM00] Víctor Carreño and César Muñoz. Aircraft trajectory modeling and alerting algorithm verification. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 90–105. Springer Berlin Heidelberg, 2000.

[dGRdB+15] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 273–289, 2015.

[EFT96] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer New York, 1996.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer, 1996.

[GLJ13] Jean Goubault-Larrecq and Jean-Pierre Jouannaud. The blossom of finite semantic trees. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics*, volume 7797 of *Lecture Notes in Computer Science*, pages 90–122. Springer Berlin Heidelberg, 2013.

[Gon08] Georges Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[Har06] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[Hea80] Percy John Heawood. Map-colour theorem. *The quarterly journal of pure and applied mathematics*, 24:332–338, 1880.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[KNV+13] Deepak Kapur, Robert Nieuwenhuis, Andrei Voronkov, Christoph Weidenbach, and Reinhard Wilhelm. Harald Ganzinger's legacy: Contributions to logics and programming. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics*, volume 7797 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2013.

[Lei89] Alexander Leitsch. On different concepts of resolution. *Mathematical Logic Quarterly*, 35(1):71–77, 1989.

[Lei97] Alexander Leitsch. *The Resolution Calculus*. Texts in theoretical computer science. Springer, 1997.

[MR04] James Margetson and Tom Ridge. Completeness theorem. *Archive of Formal Proofs*, September 2004. `http://afp.sf.net/entries/Completeness.shtml`, Formal proof development.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

[NL12] Benedikt Nordhoff and Peter Lammich. Dijkstra's shortest path algorithm. *Archive of Formal Proofs*, January 2012. `http://afp.sf.net/entries/Dijkstra_Shortest_Path.shtml`, Formal proof development.

[Pau13] Lawrence C. Paulson. Gödel's incompleteness theorems. *Archive of Formal Proofs*, November 2013. `http://afp.sf.net/entries/Incompleteness.shtml`, Formal proof development.

[Rid04] Tom Ridge. A mechanically verified, efficient, sound and complete theorem prover for first order logic. *Archive of Formal Proofs*, September 2004. `http://afp.sf.net/entries/Verified-Prover.shtml`, Formal proof development.

[Rob79] J.A. Robinson. *Logic, Form and Function: The Mechanization of Deductive Reasoning*. Artificial Intelligence Series. North-Holland, 1979.

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In *Automated Deduction — CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer Berlin Heidelberg, 1999.

[Sch13] Stephan Schulz. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pages 735–743, 2013.

[SS15] Geoff Sutcliffe and Christian Suttner. The CADE ATP system competition. `http://www.cs.miami.edu/~tptp/CASC/`, August 2015.

[Sut14] Geoff Sutcliffe. The CADE-24 Automated Theorem Proving System Competition – CASC-24. *AI Communications*, 27(4):405–416, 2014.

[TS09] René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer Berlin Heidelberg, 2009.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer Berlin Heidelberg, 2009.