Formalized First-Order Logic

Andreas Halkjær From

Kongens Lyngby 2017

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

The goal of this thesis is to formalize first-order logic, specifically the natural deduction proof system NaDeA, in the proof assistant Isabelle. The syntax and semantics are formalized as Isabelle data types and functions and the inference rules of NaDeA are defined as an inductive set.

The soundness of these inference rules is formally verified, ensuring only valid formulas can be derived in the proof system.

A textbook proof of completeness for sentences in natural deduction using abstract consistency properties is explained before this proof is formalized. This formalization is based on existing work, but modernized for this thesis. It is described in depth and a version of the completeness result using any countably infinite domain is developed by proving that the semantics respect the bijection between this and the domain of Herbrand terms.

Next the problem of open formulas is discussed and a solution provided by an extension to NaDeA. This extension allows the derivation of the original formula from its universal closure, enabling us to close the formula, apply the original completeness result and derive the original one in the extended system. Assumptions are handled by turning them into implications and back again, a technique that requires a proof of weakening to be formalized first. It is unlikely that this extension is actually necessary for completeness for open formulas, but it makes the otherwise subtle interaction between substitution and de Bruijn indices more manageable.

Finally insights gained while working with the formalization and extending it are shared, in the hope that it may help other formal verification efforts.

Preface

This thesis is submitted in partial fulfillment of the requirements for acquiring a BSc in Engineering (Software Technology). The thesis is for 15 ECTS and deals with the formalization of soundness and completeness proofs for natural deduction in the Isabelle proof assistant.

I have previously taken the course 02156 Logical Systems and Logic Programming on first-order logic and Prolog, where I also worked with NaDeA. Furthermore I have taken the courses 02157 Functional Programming and 02257 Applied Functional Programming, both on F#. I have previous experience with Isabelle through the special course "A Simple Prover with a Formalization in Isabelle" (based on a sequent calculus and thus quite different from NaDeA).

I would like to thank my supervisor Jørgen Villadsen for his guidance, constant encouragement and eye for detail, and for teaching me logic and Isabelle in the first place. I would also like to thank my co-supervisors Anders Schlichtkrull and John Bruntse Larsen for useful critiques on this document and insights into the problem in general.

I am grateful to Stefan Berghofer for taking the time to answer some of the questions that arose during this work.

I want to thank Thomas Søren Henney for his friendship, advice and academic support during the past three years as well as Magnus Brandt-Møller and Jacob E. Overgaard for making the writing process less lonely. A special thanks goes to my family for their support during this project.

Andreas Halkjær From

Contents

Chapter 1

Introduction

This introduction is divided into three sections. First the aim and scope of the project is presented along with important concepts from logic and an introduction to formalization. Next the contributions made by this work are explained and this project's relation to existing work noted. Finally an outline for the rest of the thesis is given.

1.1 Aim and Scope

The aim of the project is to formalize soundness and completeness proofs for a proof system in first-order logic. To do this these concepts must first be understood and the next few subsections aim to provide exactly this understanding. The scope of the project is limited to an existing proof system, NaDeA, and the proofs are based on existing formalized proofs. These proofs have been updated as part of this work, and an extension of the completeness proof has been developed. The details of this are explained in the next section.

1.1.1 Logic

The main part of the title of this thesis is logic. The study of logic has interested mankind at least since Aristotle's syllogisms in ancient Greece: "All men are mortal. Socrates is a man. Therefore Socrates is mortal." Syllogisms deal with the process of deriving new knowledge from existing facts, something known as inference. Another important aspect of logic is semantics, or, the meaning of things. Logic is a tool that allows us to be absolutely precise about inference and semantics, ensuring that the conclusions we make from valid premises are themselves valid, regardless of any specific meaning attributed to either premises or conclusions. With that said, the purpose of this introduction is not absolute precision, but to provide intuition for the rest of the chapters where the definitions are formalized. The following descriptions follow the textbook *Mathematic Logic for Computer Science* by Ben-Ari [Ben12] to some degree.

One type of logic is propositional logic which deals with statements that are true or false, such as "it is raining" or "the moon is made of green cheese." These statements, encoded as proposition symbols, can be combined into formulas using so-called logical connectives like negation, conjunction and implication [Ben12, def. 2.1]. This allows us to form sentences like "it is raining and the moon is made of green cheese." Precise definitions of these logical connectives were given by George Boole in 1847 with his development of boolean algebra and today the connectives are also known as boolean operators, even though we use a different notation than Boole did [Bur14]. Natural language can lead to ambiguity, so when working in propositional logic we use the syntax described below instead. Capital letters are used to represent arbitrary formulas here and henceforth. A formula in propositional logic is then either:

- A proposition symbol, $p, q, r, \ldots, \top$ or $\bot$.
- A negation, $\neg A$.
- A conjunction, $A \wedge B$, disjunction, $A \vee B$, or implication, $A \to B$.

Other logical connectives exist, but these can be derived from the ones above. Our example may then be encoded as $p \wedge q$ where p stands for "it is raining" and q for "the moon is made of green cheese." The next step is semantics, the meaning of formulas. For this we first need an assignment, σ, of truth values to the proposition symbols, e.g. $\sigma(p) = T$ and $\sigma(q) = F$ where T represents truth and F represents falsehood. Next the truth value, $v_\sigma(A)$ of a formula A under assignment σ is determined inductively as follows where "iff" is short for "if and only if":

- $v_\sigma(\top) = T$ and $v_\sigma(\bot) = F$.
- $v_\sigma(p) = \sigma(p)$ when p is a proposition symbol other than $\top$ and $\bot$.
- $v_\sigma(\neg A) = T$ iff $v_\sigma(A) = F$.
- $v_\sigma(A \wedge B) = T$ iff $v_\sigma(A) = T$ and $v_\sigma(B) = T$.
- $v_\sigma(A \vee B) = T$ iff $v_\sigma(A) = T$ or $v_\sigma(B) = T$.
- $v_\sigma(A \to B) = F$ if $v_\sigma(A) = T$ but $v_\sigma(B) = F$, and T otherwise.

A formula is said to be true if its truth value is T and false if its truth value is F.

1.1.2 First-Order Logic

Propositional logic is not powerful enough to express what we really mean when we say "all men are mortal"; in propositional logic the statement is simply true or false and cannot be used to gain knowledge of any particular man. This is remedied by moving to the next part of the title, first-order logic. First-order logic was described by Gottlob Frege in his *Begriffsschrift* from 1879 where he gives precise definitions of of terms, the universal quantifier and predicates [Zal17]. First, a term is either:

- A variable $x, y, z, \ldots$
- A function symbol $f, g, h, \ldots$ applied to a list of terms. Each function symbol has an associated *arity* which determines the length of the list of terms they can be applied to. Functions of no arguments, of arity zero, are called constants and are often named c.

Given terms we can now define formulas in first-order logic as either:

- $\top$ or $\bot$.
- A negated formula or two formulas connected by a logical connective as in propositional logic.
- A predicate symbol $p, q, r, \ldots$ applied to a list of terms. Like function symbols, predicates also have an associated arity. Predicates of arity zero correspond to proposition symbols in propositional logic.

- A universally quantified formula $\forall x.A$, where x is a variable name.
- An existentially quantified formula $\exists x.A$ where x is a variable name.

In the last two cases we say that the quantifier binds the variable x in the formula A. Equivalently a quantifier may be referred to as a binder. When a formula only contains bound variables it is said to be closed. Closed formulas are also called sentences. Conversely we call unbound variables free and the formulas they occur in open.

To give a semantics to formulas in first-order logics we need to specify a domain, $\mathcal{D}$, which is a non-empty set of values a variable can take on. A variable assignment, also known as an environment, maps variables to elements of $\mathcal{D}$. We also need an assignment of the function symbols to functions on the domain, $\mathcal{F}$. If for instance the domain is the natural numbers and f is a function symbol of arity two, f may be assigned the meaning of addition. Constants are simply assigned values from the domain. Finally we need an assignment, $\mathcal{G}$, from predicate symbols and associated lists of terms to truth values. Taking again the domain of natural numbers, a predicate p of arity two may be given the meaning of equality. The three of these, $\mathcal{D}$, $\mathcal{F}$ and $\mathcal{G}$ constitute an interpretation [Ben12, def. 9.3]. The set of formulas expressible under an interpretation may be referred to as the language. Given an interpretation and a variable assignment, we can give a semantics first for terms then for formulas. Terms interpret to members of $\mathcal{D}$:

- $v_\sigma(x) = \sigma(x)$.
- $v_\sigma(f(t_1, \ldots, t_n)) = (\mathcal{F}(f))(v_\sigma(t_1), \ldots, v_\sigma(t_n))$. That is, the value of a function symbol, f, applied to a list of terms, is given by recursively interpreting the list of terms and applying the result of looking up f in $\mathcal{F}$ to the resulting list of values.

The truth value $v_\sigma(A)$ of a formula A under assignment σ is again defined inductively. The notation $\sigma[x \leftarrow d]$ is short-hand for the function which maps x to d and every other input y to $\sigma(y)$.

- $v_\sigma(\top) = T$ and $v_\sigma(\bot) = F$
- The logical connectives have the same meaning as in propositional logic.
- $v_\sigma(p(t_1, \ldots, t_n)) = (\mathcal{G}(p))(v_\sigma(t_1), \ldots, v_\sigma(t_n))$.
- $v_\sigma(\forall x.A) = T$ iff $v_{\sigma[x \leftarrow d]}(A) = T$ for all $d \in \mathcal{D}$.

- $v_\sigma(\exists x.A) = T$ iff $v_{\sigma[x \leftarrow d]}(A) = T$ for some $d \in \mathcal{D}$.

A formula is satisfiable if there exists an interpretation and environment under which it interprets to T and valid if it does so under all interpretations and environments. A satisfying interpretation is called a model. This definition deviates from Ben-Ari where satisfiability and validity are only defined for closed formulas [Ben12, def 7.23] whose truth value is independent of the initial environment.

As an example, the statement "If every person that is not rich has a rich father, then some rich person must have a rich grandfather." can be encoded in first-order logic as:

$$\forall x.(\neg r(x) \to r(f(x))) \to \exists x.(r(x) \land r(f(f(x))))$$

where the domain is people and $r(x) = T$ iff x is rich and $f(x)$ is the father of x, making $f(f(x))$ the grandfather. It turns out that this example is actually valid.

Other logics exist and are used for certain purposes, but first-order logic is in many ways the primary logic used today. It also forms the basis of the higher-order logic used in the Isabelle/HOL proof assistant used in this project.

1.1.3 Proof Systems

To prove that a formula is valid, we might use a proof system to derive it. A proof system is defined as a set of axioms, formulas which we assert can be derived, and an inductive definition of inference rules for deriving more formulas from existing ones. For instance we might encode the inference rule modus ponens which says that if $P \to Q$ can be derived and P can, then we can derive Q. We say that a formula is derivable in a proof system if there exists a chain of inferences starting (or ending) at axioms, which produces the formula. Extending this, a formula may be derivable from some assumptions if the formula can be derived assuming these assumptions as axioms.

Natural deduction is a proof system developed especially by Gerhard Gentzen in 1932 which emphasizes inference rules that are very close to human reasoning [Pla16]. As such, natural deduction has become widespread in both theory and practice; much metatheory about it has been developed and both Isabelle and the Coq proof assistant use this kind of proof system internally. Natural deduction is explained in more depth in chapter 3.

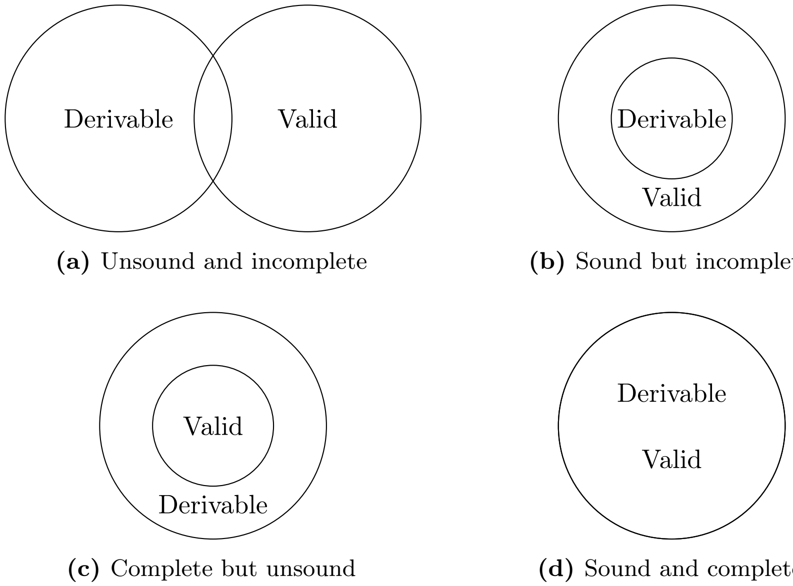

Figure 1.1: The four possible combinations of soundness and completeness over the space of formulas.

Two important properties of proof systems are soundness and completeness. Soundness states that only valid formulas can be derived in the system. It is a correctness property enabling us to trust the proofs we make with the system. Completeness states that all valid formulas are derivable in the system. This obviously makes the proof system a lot more useful than an incomplete one. The relation between these two properties and the space of valid and derivable formulas is depicted in figure 1.1. As depicted, exactly the valid formulas can be derived in a sound and complete proof system.

1.1.4 Formalization

First-order logic is relevant for software development because it provides a precise language that can be used when proving properties about programs. The inferences used in a proof, whether these are specified by the user or found automatically, can be checked by a computer to ensure that they are applied correctly according to the proof system.

However this only checks that the proof system is used correctly, not that the resulting proof itself is valid; for that we need a proof of soundness. It would be a shame to derive a proof in such a system, only to find out later that its soundness proof was flawed. This is especially a problem if the proof system has been used in the verification of mission-critical software: The software may fail, even though it was "proven" correct, because the soundness proof had a flaw! Alternatively, time may be wasted trying to prove something in the belief that the proof system is complete, but that proof might as well be flawed.

A prominent example of a flawed proof is one by Kempe in 1879 of the four color conjecture (now theorem). This proof was believed until 1890 where Heawood found a counter example to one of Kempe's assumptions [Hea80]. Kempe's fallacious proof, along with any other fallacious proof, cannot be formalized in a correct proof assistant and this is where the last part of the title comes in.

When we talk about formalization of a proof, we mean that there is a mechanical process for determining its correctness [Har08]: A computer can check it for us. The machine has no intuition for what should or should not hold, so blind spots like Kempe's are avoided. Formalization of a proof in a state-of-the art proof assistant like Isabelle thus gives us very high confidence that the proof is correct.

Isabelle is an interactive proof assistant for formalizing mathematics and formally verifying the correctness of software. The most commonly used instance of Isabelle is Isabelle/HOL that uses higher-order logic as the basis for its proofs. In this thesis, I will refer to Isabelle/HOL simply as Isabelle. HOL can be thought of as first-order logic extended with data types and a simple (non-dependent) type theory. Isabelle also supports recursion over these data types, pattern matching, inductive definitions and more. Various methods of proof search are available to handle intermediary bookkeeping, allowing Isabelle proofs to somewhat resemble pencil-and-paper proofs in terms of the number of details given. These proofs can be developed in the Isabelle/jEdit editor that continuously checks their correctness and allows rich semantic markup. Isabelle can be downloaded at:

`http://isabelle.in.tum.de`

The issue of trusting the code of the proof assistant is mitigated by hand-verification of the assistant's small kernel through which every proof must go, or by automatically translating proofs between different systems [Hal08]. By formalizing the soundness and completeness proofs in Isabelle we can therefore be almost absolutely certain that they are correct.

There is a research environment around the formalization of logics in Isabelle called IsaFoL [IsaFoL] which this work is a part of. The focus of the IsaFoL project is on formalizing modern research in automated reasoning used e.g. for program verification.

It is worth noting that the highest level of assurance is assigned by the Common Criteria standard to systems that have been formally verified as well as tested [Cri12]. In conclusion, its application to program verification makes formalized first-order logic worth studying.

1.2 Contributions

The main contributions of this thesis are a proof of completeness for closed formulas for the natural deduction proof system NaDeA formalized in Isabelle and an extension of this proof to open formulas by the addition of an inference rule dubbed *Subtle*. Two versions of *Subtle* are given, one needed for the general case of a formula being a consequence of a list of assumptions and a simpler one for completeness of formulas valid by themselves. The soundness of NaDeA was already established [VJS17], but the proof was reworked during this project and extended to cover the extra rule.

NaDeA has been used at DTU for teaching purposes, lately in combination with a system called ProofJudge which allows instructors to specify proofs that the students must complete and hand in. Instructors can then give feedback on the students' proofs [Vil15]. These proofs can be developed online at:

`https://nadea.compute.dtu.dk`

The main result of this contribution can be seen in the Isabelle code below, that is checked by Isabelle itself when generating the LaTeX, cf. section 1.3 below. The keyword *abbreviation* introduces a syntactic abbreviation that is unfolded by the parser, while *proposition* is used before a statement and its proof.

abbreviation ‹*valid p* ≡ ∀ (*e* :: *nat* ⇒ *nat*) *f g*. *semantics e f g p*›

proposition ‹*valid p* ⟹ *semantics e f g p*›
  using *completeness-star soundness-star* **by** *blast*

proposition ‹*OK p* [] = *valid p*› **if** ‹*sentence p*›
  using *completeness soundness that* **by** *fast*

abbreviation ‹*check p* ≡ *OK-star p* []›

proposition ‹*check* = *valid*›
  using *completeness-star soundness-star* **by** *fast*

First the abbreviation *valid p* is introduced which hides three universally quantified variables: e is the environment, while f and g correspond to $\mathcal{F}$ and $\mathcal{G}$ respectively. The function *semantics* corresponds to v above, it takes e, f and g and determines the truth value of the formula p. Thus by universally quantifying e, f and g we have stated almost exactly what we mean when we say a formula is valid, namely that it is true in every interpretation and environment. I say almost because the domain has been fixed to the natural numbers since domains are formalized as types in Isabelle and we cannot quantify over types with the universal quantifier. The proposition on the next line proves that this does not matter, as any *valid* formula is valid in the broader sense with e unrestrained.

The next proposition states that we can derive exactly the valid formulas in the original NaDeA system, *OK*, *if* these are closed. This is proven *using* the soundness and completeness proofs discussed later, *by* the proof method *fast*. The abbreviation following that checks if a formula can be derived in the simplest extension of NaDeA, *OK_star*. Finally the last proposition states that *check* and *valid* are extensionally equal. This means that they agree on every input and thus that we can derive exactly the valid formulas, even if they are open.

The completeness proof is based on the one described by Melvin Fitting in the book *First-Order Logic and Automated Theorem Proving* [Fit96]. A formalization of this and a natural deduction proof system was already formalized in Isabelle by Stefan Berghofer under the name FOL-Fitting [Ber07a]. My formalization is based on this, meaning that, disregarding my extensions, the lemmas and proofs are roughly the same. Mine have however been modernized, as described below. The NaDeA system is simpler than the one formalized by Berghofer: Negation and truth are not built-in, the supporting functions are different and NaDeA has only 14 inference rules against the 17 in FOL-Fitting. While this gives us fewer rules to prove sound, there are also are fewer rules available for completeness. One might imagine that some proof, while possible in FOL-Fitting, is impossible in NaDeA because of a missing rule. Luckily, this is not the case. Berghofer's completeness proof assumes that the given sentence is true in all interpretations with Herbrand terms and shows that it can be derived. Any generally valid sentence must then also be derivable. The completeness proof for NaDeA makes this explicit by assuming validity in any countably infinite domain.

Berghofer's formalization is in the old procedural apply-style of Isabelle while mine uses the newer declarative proof language Isar [Wen99]. Proofs in the old style manipulate the goal through application of various rules whose effect is hidden from the user, until a state is reached which either holds trivially or can be shown using proof search. The declarative style instead states facts explicitly in the source, using proof search to establish them from the previous ones. This improves the presentation of the proof making it more readable and avoids the need for various technical details needed in apply-style.

1.3 Overview

The original NaDeA formalization is available and my extended version are available, as `NaDeA.thy` and `NaDeA_C.thy` respectively, at:

https://github.com/logic-tools/nadea/tree/master/Isabelle

Excerpts are reprinted in this document. Every one of these excerpts except a few type declarations is extracted as LaTeX directly from the formalization. During this process, Isabelle verifies that everything is correct. Since no *sorry* or *oops* commands, which let you skip proofs, appear in the formalization, this means that every formalized proof in this thesis truly has been formally verified.

The generated LaTeX typesets Isabelle commands in bold. When referred to in the text these are written in italics instead as this is less visually obtrusive.

The rest of this thesis is organized as follows.

- Chapter 2 introduces the Isabelle proof assistant via small examples and verification of a functional implementation of the quicksort algorithm. The proof language is explained along with different proof methods.
- Chapter 3 explains natural deduction proofs as they are typically presented in a textbook. Example proofs are given in this textbook style.
- Chapter 4 and 5 formalize the syntax, semantics and inference rules of NaDeA in Isabelle and presents formalized versions of the previous example proofs. Especially the use of de Bruijn indices is discussed.
- Chapter 6 formalizes the soundness proof along with the necessary auxiliary lemmas. It also gives a small consistency corollary.
- Chapter 7 describes the completeness proof given in Fitting's book.
- Chapter 8 formalizes this completeness proof for NaDeA, and proves a version of it that assumes validity in any countably infinite domain. A formalization of the Löwenheim-Skolem theorem is also given.
- Chapter 9 describes my work to extend the completeness proof to cover open formulas by extending NaDeA with a sound inference rule. Several steps are required to do this, and the challenges of each step are covered.
- Chapter 10 concludes the project and discusses some of the gained insight.

Chapter 2

Formalizations in Isabelle

This chapter aims to give a general introduction to formalizations in Isabelle through small examples and to introduce the features used in the coming chapters. It is based in part on *The Isabelle/Isar Reference Manual* [Wen16b].

2.1 Numbers and Lists

Data types in Isabelle resemble those in Standard ML and are introduced by a similar declaration. For instance the natural numbers:

datatype *mynat* = *Zero* | *Succ mynat*

Or we can represent lists using a type variable which is applied in postfix notation:

datatype *'a mylist* = *Nil* | *Cons 'a* ⟨*'a mylist*⟩

Isabelle will automatically prove various properties about these data types for us, which are then available for proofs about them. We can also write functions over data types and there are several ways of declaring these. Primitive recursive functions where recursive calls are only allowed directly on constructor arguments are declared with *primrec* as follows:

primrec *plus* :: ‹*mynat* ⇒ *mynat* ⇒ *mynat*› **where**
  ‹*plus Zero m* = *m*› |
  ‹*plus* (*Succ n*) *m* = *Succ* (*plus n m*)›

After *primrec* we give the name of the function, here *plus*, and its type after a double colon. This declaration terminates with the keyword *where* and the next lines are the clauses of the function, one for each constructor of the data type. The type as well as the clauses are enclosed in brackets separating the HOL-specific types and terms from the outer Isabelle syntax [Wen16b].

Analogously to *plus* we can define functions for the length of a list and the result of appending two lists:

primrec *length* :: ‹*'a mylist* ⇒ *mynat*› **where**
  ‹*length Nil* = *Zero*› |
  ‹*length* (*Cons x xs*) = *Succ* (*length xs*)›

primrec *append* :: ‹*'a mylist* ⇒ *'a mylist* ⇒ *'a mylist*› **where**
  ‹*append Nil ys* = *ys*› |
  ‹*append* (*Cons x xs*) *ys* = *Cons x* (*append xs ys*)›

Given these declarations we are now in a position to prove our first theorem. We will prove that the length of one list appended to another is equal to the sum of the lengths of the original lists. To do this we start by declaring the theorem we want to prove and possibly give it a name, here *length-append*:

theorem *length-append*:
  ‹*length* (*append xs ys*) = *plus* (*length xs*) (*length ys*)›

The variables *xs* and *ys* are automatically universally quantified. Next we need to decide how to prove the given theorem. In this case we will use induction over the first list, *xs*. This will split the goal into two cases, one for each constructor, that we then need to prove. The case for *Nil* is proven thus:

proof (*induct xs*)
  case *Nil*
  show *?case*
    by *simp*
next

The *proof* command initiates the structured proof using the chosen method; a direct proof is done by using a hyphen instead of *(induct xs)*. We specify which case we are proving using *case Nil* and the intent to *show* it is declared on the next line. Here *?case* is a syntactic abbreviation introduced by Isabelle to stand for the goal:

$$length\ (append\ Nil\ ys) = plus\ (length\ Nil)\ (length\ ys)$$

This is the original goal with *xs* replaced by *Nil* as we are in the base case of the induction. Syntactic abbreviations like *?case* are prefixed with a question mark and unfolded by the parser. They may be introduced by the user using either the *is* or *let* commands which will appear later.

The case is proven *by simp*, the Isabelle simplifier that tries to rewrite the given terms, to unify them, here succesfully. This is done using their definitions and any available facts about them. We use the *next* command to signify that this case is proven and we are ready to move on to the next:

  case (*Cons x xs*)
  then show *?case*
    by *simp*
qed

Here *case (Cons x xs)* introduces names for the constructor arguments in this case along with an assumption of the induction hypothesis:

$$length\ (append\ xs\ ys) = plus\ (length\ xs)\ (length\ ys)$$

By using the command *then* before the *show*, we make this assumption available to the coming proof method. Again the case can be solved by the simplifier, and as there are no more cases to prove, the proof is concluded with *qed*.

As both of these cases can be proven by the simplifier, we may use the following syntax to prove the theorem more succinctly:

theorem ‹*length* (*append xs ys*) = *plus* (*length xs*) (*length ys*)›
  by (*induct xs*) *simp-all*

Here the *by* command takes two proof methods, the first sets up the induction and the second, *simp-all*, solves the resulting cases using the simplifier.

It is worth noting that the namespaces of functions and theorems are separate, so a theorem may be called the same as a function.

If we cannot or do not want to prove the subgoals immediately, we may introduce intermediary facts in a proof using *have*. This is shown in the following first part of a proof of the associativity of *plus*:

lemma ‹*plus x* (*plus y z*) = *plus* (*plus x y*) *z*›
proof (*induct x*)
  case (*Succ x*)
  have ‹*plus* (*Succ x*) (*plus y z*) = *Succ* (*plus x* (*plus y z*))›
    by *simp*

The *have* command can be prefixed with several keywords, e.g. *then* as used before *show* above and *moreover* which will be used later. Below we prefix it with *also*:

  also have ‹... = *Succ* (*plus* (*plus x y*) *z*)›
    using *Succ* **by** *simp*
  also have ‹... = *plus* (*Succ* (*plus x y*)) *z*›
    by *simp*
  also have ‹... = *plus* (*plus* (*Succ x*) *y*) *z*›
    by *simp*
  finally show *?case* **.**
qed *simp*

This chains together a series of equalities with *finally* referring back to all of them [Wen16b, p. 39]. The ellipsis refers to the right-hand side of the previous result. This allows us to do a gradual rewrite of the the left-hand side of the statement to match the right-hand side. Here the single period proof method only succeeds if the two terms unify directly. The *simp* after the final *qed* applies to any unsolved cases, here *Zero*.

Non-recursive function can be introduced as definitions in the following way:

definition *double* :: ‹*mynat* ⇒ *mynat*› **where**
‹*double n* = *plus n n*›

This adds a layer of indirection that can be unfolded to reveal the definition:

lemma ‹*length* (*append xs xs*) = *double* (*length xs*)›
unfolding *double-def* **by** (*simp add*: *length-append*)

Here we are adding the lemma *length-append* to the simplifier, telling it that it can use it as a rewrite rule. An alternative to this is declaring the lemma with the [*simp*] attribute, which adds it globally.

2.2 Proof Methods

The *simp*, *simp-all* and period are far from the only available proof methods [Wen16b, p. 232].

The following lemma cannot be solved by *simp-all* and uses *auto* instead, which combines the simplifier with classical reasoning.

lemma ‹(*xs* = *append xs ys*) = (*ys* = *Nil*)›
by (*induct xs*) *auto*

When *auto* is not strong enough, *force* may be used, which performs a “rather exhaustive search” using “many fancy proof tools” [Wen16b, p. 232]. As an example *force* is powerful enough to automatically prove the following formulation of Cantor’s diagonal argument that there are infinite sets which cannot be put into one-to-one correspondence with the natural numbers [Wen16a]:

theorem *Cantor*: ‹$\nexists f$ :: *nat* ⇒ *nat set*. $\forall A.\ \exists x.\ f\ x = A$›
by *force*

Though it should be noted that we gain no insight from such an automated proof.

blast is an integrated classical tableau prover that is written to be very fast but does not make use of simplification. The rich grandfather example from the introduction can be proven directly in Isabelle with this proof method:

lemma $\langle(\forall x.\ (\neg\ r(x) \longrightarrow r(f(x)))) \longrightarrow (\exists x.\ (r(x) \wedge r(f(f(x)))))\rangle$
  by *blast*

The proof method *fast* uses sequent-style proving and a breadth-first search strategy where *blast* uses a more general strategy, but can be slower than *fast*. With *fast* we may prove that if we have a set of lists which are constructed by appending some list with itself, then any list we pick will have even length. The assumptions are formulated using the symbol for higher-order implication, $\Longrightarrow$:

lemma $\langle\forall\, xs \in A.\ \exists\, ys.\ xs = \mathit{append}\ ys\ ys \Longrightarrow us \in A \Longrightarrow$
  $\exists\, n.\ \mathit{length}\ us = \mathit{plus}\ n\ n\rangle$
  using *length-append* **by** *fast*

Here we are *using* the previous lemma *length-append*. This is a more general method than *then* to make a previous result available to the proof method. The above may also be proven with the *fastforce* method which is essentially like *fast* but with access to the simplifier. While *blast* and *fast* use classical reasoning, the method *iprover* uses only intuitionistic logic.

Finally the following chapters will make use of *metis*, an integrated theorem prover for first-order logic that implements ordered paramodulation, an advanced form of resolution [Wen16b, p. 292]. An example use of *metis* is given below where we prove that if a number acts as the identity for *plus*, it must be zero:

lemma $\langle\forall\, x.\ \mathit{plus}\ x\ y = x \Longrightarrow y = \mathit{Zero}\rangle$
  using $\mathit{plus.simps}(\mathit{1})$ **by** *metis*

These proof methods allow us to take steps in the proof of a natural size, comparative to what we would do on paper. The computer can handle all the details allowing us to focus on the big picture and no matter which method, internal or external, is used for finding a proof, this proof passes through Isabelle's small core of primitives ensuring its correctness.

2.3 Quicksort

With the basics covered, we now turn to the built-in list data type and look at a complete Isabelle theory with a verification of quicksort. The theory starts with a declaration of its name and any imports. Here we will need support for multisets:

theory *QuickSort* **imports** *~~/src/HOL/Library/Multiset* **begin**

We are going to verify the following implementation of quicksort where we use the first element of the list as the pivot element, partition in smaller and larger halves, recursively sort these and append the results. The element type *'a* is required to form a linear order so that we can compare elements of it:

fun $quicksort$ :: ⟨$('a{::}linorder)\ list \Rightarrow 'a\ list$⟩ **where**
⟨$quicksort\ [] = []$⟩ |
⟨$quicksort\ (x\ \#\ xs) =$
$(let\ (as,zs) = partition\ (op \geq x)\ xs$
$in\ quicksort\ as\ @\ x\ \#\ quicksort\ zs)$⟩

This function is not primitive recursive and is therefore declared with the *fun* keyword. Recursive functions in Isabelle must terminate and this can be proven automatically or manually. Since Isabelle knows that the *partition* function does not return longer lists than its input, it is able to prove the termination of *quicksort* automatically.

2.3.1 Permutation

The first thing we will prove is that the sorted list is a permutation of the original list. This is formulated using multisets as follows:

lemma *quicksort-permutes* [*simp*]:
⟨$mset\ (quicksort\ xs) = mset\ xs$⟩

We will prove this lemma by induction over the recursive calls made by the algorithm. Therefore we specify a custom induction rule when starting the proof:

```
proof (induct xs rule: quicksort.induct)
  case 1
  show ?case by simp
next
```

The above also proves the base case, corresponding to the first clause of *quicksort*, using the simplifier. The first half of the next case is more interesting:

```
  case (2 x xs)
  moreover obtain as zs where ‹(as,zs) = partition (op ≥ x) xs›
    by simp
```

First we obtain names, x and xs, for the arguments of the constructor. Then we *obtain* names, *as* and *zs*, for the result of the partitioning done by the algorithm, allowing us to state properties of them. The keyword *moreover* means that we are accumulating these facts behind the scenes — the induction hypothesis and the origin of *as* and *zs*. Using *moreover* we can avoid having to come up with names for all the intermediary facts making the presentation of the proof cleaner. This style is used a lot in the remaining chapters for this reason. Next we will prove that *xs* as a multiset is exactly the union of the multisets of *as* and *zs*.

```
  moreover from this have ‹mset as + mset zs = mset xs›
    by (induct xs arbitrary: as zs) simp-all
  ultimately show ?case
    by simp
qed
```

We do this by induction over *xs* allowing *arbitrary* lists to stand in for *as* and *zs* when applying the induction hypothesis; this allows *simp-all* to finish the proof. The keywords *from this* give the proof method access to the previous fact like with *then*, but unlike *then* are allowed after *moreover*. Any previously established fact can take the place of *this*. Alternatively we could write *using calculation(3)* before *by*, which would refer to the third fact in the chain of *moreover*s. The keyword *ultimately* is another way of accessing the *calculation* and terminates the chain. In this case it is appropriate because we are finished.

As a corollary we will prove the weaker result that the sets of the sorted and original list are equal. This is done automatically with *metis* using the above result and a fact about the equality of sets and multisets:

```
corollary set-quicksort [simp]:
  ‹set (quicksort xs) = set xs›
  using quicksort-permutes set-mset-mset by metis
```

This proof can be found using Isabelle's *sledgehammer* tool that uses various external solvers to essentially do *proof search* search. This is a very convenient tool as it saves us from having to look through the lemmas in the multiset library for anything appropriate, when it can be found automatically.

2.3.2 Sorting

Now we are in a position to prove that quicksort actually sorts its argument list. Isabelle has a built-in function *sorted* that checks that a given list is sorted. The base case is trivial:

```
lemma quicksort-sorts [simp]: ‹sorted (quicksort xs)›
proof (induct xs rule: quicksort.induct)
  case 1
  show ?case by simp
next
```

The recursive case is more interesting. Again we obtain names for the lists created by the call to *partition*, this time naming the fact *:

```
  case (2 x xs)
  obtain as zs where *: ‹(as,zs) = partition (op ≥ x) xs›
    by simp
  then have ‹∀ a ∈ set as. ∀ z ∈ set (x # zs). a ≤ z›
    using order-trans set-ConsD le-cases partition-P by metis
  then have
    ‹∀ a ∈ set (quicksort as). ∀ z ∈ set (x # quicksort zs). a ≤ z›
    by simp
```

Also above, we state that every element in *as* is smaller than or equal to the pivot and the elements of *zs*. And because of the corollary above with the [*simp*] attribute, we can extend this to the results of the recursive calls in the final line above. A few more lines conclude the proof:

```
    then have ‹sorted (quicksort as @ x # quicksort zs)›
      using * 2 set-quicksort sorted-append sorted-Cons le-cases partition-P
      by metis
    then show ?case
      using * by simp
qed
```

Here we use lemmas from the standard library about when a list is sorted to prove that the result of appending the recursive calls and the pivot is sorted, knowing by the induction hypothesis that the recursive calls are sorted. Finally we use the origin of *as* and *zs* to prove that *quicksort (x # xs)* is sorted.

Given the above lemmas we can prove the following theorem that our *quicksort* acts exactly like the built-in *sort* function:

```
theorem sort-quicksort: ‹sort = quicksort›
  using properties-for-sort by (rule ext) simp-all
```

We are using the *properties-for-sort* lemma which states the following:

$$mset\ ?ys = mset\ ?xs \Longrightarrow sorted\ ?ys \Longrightarrow sort\ ?xs = ?ys$$

The premises match *quicksort-permutes* and *quicksort-sorts* perfectly allowing us to conclude *quicksort xs = sort xs*. To turn this into the proof *quicksort = sort* we apply the rule *ext* which states:

$$(\bigwedge x.\ ?f\, x = ?g\ x) \Longrightarrow ?f = ?g$$

Namely that if two functions give the same results for every input then we can conclude that the two functions are themselves equal. The application of *rule* as the initial proof method rewrites the goal using its argument.

Finally we can end the theory having verified that quicksort is functionally equivalent to the built-in sort.

```
end
```

CHAPTER 3

Proofs in Natural Deduction

3.1 Natural Deduction in a Textbook

To understand the formalization of proofs in natural deduction, it is instructive first to consider how they are done in a textbook, here *Logic in Computer Science — Modelling and Reasoning about Systems* by Huth and Ryan [HR04].

3.1.1 On Substitution

Before looking at the inference rules we need to understand the concept of substitution, as this is central to the treatment of quantifiers in natural deduction. The following definition for substitution is given in the considered textbook [HR04, p. 105 top]:

> Given a variable x, a term t and a formula ϕ we define $\phi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

A definition for what it means that "t must be free for x in ϕ" follows shortly after [HR04, p. 106 top].

> Given a term t, a variable x and a formula ϕ, we say that t is free for x in ϕ if no free x leaf in ϕ occurs in the scope of $\forall y$ or $\exists y$ for any variable y occurring in t.

Here the syntax tree of the formula is considered, explaining the use of the term *leaf.* The following quote [HR04, p. 106 bottom] emphasizes the side conditions:

> It might be helpful to compare "t is free for x in ϕ" with a precondition of calling a procedure for substitution. If you are asked to compute $\phi[t/x]$ in your exercises or exams, then that is what you should do; but any reasonable implementation of substitution used in a theorem prover would have to check whether t is free for x in ϕ and, if not, rename some variables with fresh ones to avoid the undesirable capture of variables.

As we will see, these complications are made explicit in the formalization by simple functional programs.

3.1.2 Natural Deduction Rules

Next follows the natural deduction rules as described in the literature [HR04]. The first 9 are rules for classical propositional logic and the last 4 are for first-order logic. Intuitionistic logic can be obtained by omitting the rule *PBC* (proof by contradiction, called "Boole" later) and adding the $\perp$-elimination rule (also known as the rule of explosion) [Sel89].

Besides *PBC* which is a little special, the rules act as either introduction (I) or elimination (E) rules for the logical connectives and quantifiers. The way to read rules like this is that, having derived the formulas above the line we may derive the one below the line. The rules are as follows with names given to the right of the line:

$$\dfrac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\ PBC \qquad \dfrac{\phi \qquad \phi\to\psi}{\psi}\ \to E \qquad \dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi\to\psi}\ \to I$$

$$\dfrac{\phi\vee\psi \qquad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \qquad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\ \vee E \qquad \dfrac{\phi}{\phi\vee\psi}\ \vee I_1 \qquad \dfrac{\psi}{\phi\vee\psi}\ \vee I_2$$

$$\dfrac{\phi\wedge\psi}{\phi}\ \wedge E_1 \qquad \dfrac{\phi\wedge\psi}{\psi}\ \wedge E_2 \qquad \dfrac{\phi \qquad \psi}{\phi\wedge\psi}\ \wedge I$$

$$\dfrac{\exists x\,\phi \qquad \boxed{\begin{array}{cc}x_0 & \phi\,[x_0/x]\\ & \vdots\\ & \chi\end{array}}}{\chi}\ \exists E \qquad \dfrac{\phi\,[t/x]}{\exists x\,\phi}\ \exists I$$

$$\dfrac{\forall x\,\phi}{\phi\,[t/x]}\ \forall E \qquad \dfrac{\boxed{\begin{array}{cc}x_0 & \\ & \vdots\\ & \phi\,[x_0/x]\end{array}}}{\forall x\,\phi}\ \forall I$$

With the following side conditions to rules for quantifiers:

- $\exists E$: x_0 does not occur outside its box (and therefore not in χ).
- $\exists I$: t must be free for x in ϕ.
- $\forall E$: t must be free for x in ϕ.
- $\forall I$: x_0 is a new variable which does not occur outside its box.

Consider for instance the elimination rule for disjunction, $\vee E$. It includes three premises, first that we know either ϕ or ψ (or both) can be derived, $\phi \vee \psi$. Second and third that assuming ϕ respectively ψ holds we can derive χ. Knowing these three things, the rule allows us to derive χ. This makes sense intuitively; if ϕ holds, we can use the second premise to prove χ, while if ψ holds we can use the third. If both do we may use either. In all cases we have proven χ justifying the soundness of this rule.

Consider now the elimination rule for the universal quantifier, $\forall E$, as an example of how substitution might go wrong without the side condition. We might, hypothetically, have a proof of $\forall x.\exists y.P(x, y)$. If we apply the rule ignoring the side condition, we might get $\exists y.P(y, y)$. But this is clearly not the same, as the x and y may always be distinct in the first formula.

In addition to the rules above, the textbook formulation requires a special copy rule [HR04, p. 20] described below. The copy rule is not needed in the formalization due to the way it manages a list of assumptions.

> A final rule is required in order to allow us to conclude a box with a formula which has already appeared earlier in the proof. [...] The copy rule entitles us to copy formulas that appeared before, unless they depend on temporary assumptions whose box has already been closed.

As it can be seen, there are no rules for truth or negation, but the following equivalences can be used:

$$\begin{aligned} \top &\equiv \bot \to \bot \\ \neg A &\equiv A \to \bot \end{aligned}$$

3.2 Example Proofs

Let us construct some proofs using natural deduction to get a feel for the rules.

3.2.1 Modus Tollens

As a first example, we may prove the modus tollens principle which has no quantifiers. Modus tollens states that if A implies B and B does not hold, then A

cannot hold (for if A did hold then so would B because of the implication, but this is a contradiction). It can be encoded in first-order logic as $(A \to B) \land \neg B \to \neg A$ or, using the above equivalence to avoid negation, as:

$$(A \to B) \land (B \to \bot) \to (A \to \bot)$$

Instead of the boxes to represent assumptions, the more compact turnstile notation with assumptions to the left of $\vdash$ and the conclusion to the right will be used. This is similar to what is done in the formalization. Futhermore the assumptions will be abbreviated with $\ldots$ when they do not change between steps.

A natural deduction proof starts from its conclusion:

$$\vdash (A \to B) \land (B \to \bot) \to (A \to \bot)$$

Now we look at the rules and see which ones have something similar as their conclusion. Given that the outermost logical connective is an implication, the proof will start with an implication introduction:

$$\dfrac{(A \to B) \land (B \to \bot) \vdash A \to \bot}{\vdash (A \to B) \land (B \to \bot) \to (A \to \bot)} \to I$$

We still have an implication outermost, so another implication introduction seems like a good choice:

$$\dfrac{\dfrac{A, (A \to B) \land (B \to \bot) \vdash \bot}{(A \to B) \land (B \to \bot) \vdash A \to \bot} \to I}{\vdash (A \to B) \land (B \to \bot) \to (A \to \bot)} \to I$$

Now we need to prove $\bot$ and we see that we have $B \to \bot$ stuffed away in the assumptions, so let use $\to E$ to try and utilise that. This leaves a choice of what to instantiate ψ with, as it only occurs above the line. Given that we want to use $B \to \bot$ we set $\psi = B$. This adds two new premises to prove:

$$\dfrac{\dfrac{\dfrac{\ldots \vdash B \to \bot \qquad \ldots \vdash B}{A, (A \to B) \land (B \to \bot) \vdash \bot} \to E}{(A \to B) \land (B \to \bot) \vdash A \to \bot} \to I}{\vdash (A \to B) \land (B \to \bot) \to (A \to \bot)} \to I$$

The left one can be discharged by the $\land E2$ rule and an appeal to the assumptions:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots \vdash (A \to B) \land (B \to \bot)}{\dots \vdash B \to \bot}\land E2 \qquad A, (A \to B) \land (B \to \bot) \vdash B}{A, (A \to B) \land (B \to \bot) \vdash \bot} \to E}{(A \to B) \land (B \to \bot) \vdash A \to \bot} \to I}{\vdash (A \to B) \land (B \to \bot) \to (A \to \bot)} \to I$$

The right one is proven similarly, leaving us with the following final proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots \vdash (A \to B) \land (B \to \bot)}{\dots \vdash B \to \bot}\land E2 \qquad \dfrac{\dfrac{\dfrac{\dots \vdash (A \to B) \land (B \to \bot)}{\dots \vdash A \to B}\land E1 \qquad \dots \vdash A}{\dots \vdash B} \to E}{}}{A, (A \to B) \land (B \to \bot) \vdash \bot} \to E}{(A \to B) \land (B \to \bot) \vdash A \to \bot} \to I}{\vdash (A \to B) \land (B \to \bot) \to (A \to \bot)} \to I$$

3.2.2 Socrates is Mortal

Next to exercise a quantifier rule and substitution let us prove Aristotle's syllogism encoded in first-order logic:

$$(\forall x.h(x) \to m(x)) \land h(s) \to m(s)$$

Here $h(x)$ can be read as x *is human*, $m(x)$ as x *is mortal* and s as Socrates. So if everyone who is human is also mortal and Socrates is human then Socrates must be mortal. Again the proof starts with an implication introduction:

$$\dfrac{(\forall x.h(x) \to m(x)) \land h(s) \vdash m(s)}{\vdash (\forall x.h(x) \to m(x)) \land h(s) \to m(s)} \to I$$

Next we see that the implication in our assumptions could probably be used, so we do an implication elimination giving us two new premises to prove:

$$\dfrac{\dfrac{\dots \vdash h(s) \to m(s) \qquad \dots \vdash h(s)}{(\forall x.h(x) \to m(x)) \land h(s) \vdash m(s)} \to E}{\vdash (\forall x.h(x) \to m(x)) \land h(s) \to m(s)} \to I$$

The right one can be derived from the assumptions using $\wedge E2$ as previously, but the left one requires the use of a previously unused rule, $\forall E$:

$$\cfrac{\cfrac{\cfrac{\ldots \vdash (\forall x.h(x) \to m(x)) \wedge h(s)}{\ldots \vdash \forall x.h(x) \to m(x)}\wedge E1}{\ldots \vdash h(s) \to m(s)}\forall E \qquad \cfrac{\ldots \vdash (\forall x.h(x) \to m(x)) \wedge h(s)}{\ldots \vdash h(s)}\wedge E2}{\cfrac{(\forall x.h(x) \to m(x)) \wedge h(s) \vdash m(s)}{\vdash (\forall x.h(x) \to m(x)) \wedge h(s) \to m(s)} \to I} \to E$$

The side condition for $\forall E$ says that s must be free for x in $\forall x.h(x) \to m(x)$ but this is evidently the case as s does not appear in the formula at all. Thus we can derive $(h(x) \to m(x))[s/x]$ which simplifies to $h(s) \to m(s)$ and the proof is complete after obtaining this from the assumptions.

Thus we have seen how to decompose a statement into obviously true cases using fairly natural rules.

Chapter 4

Formalizing Syntax and Semantics

To work with first-order logic and natural deduction in Isabelle, we first need to formalize the syntax and semantics of formulas.

4.1 Syntax

Let us consider first how to define the syntax so we can work with it in Isabelle.

4.1.1 Terms

We use a type synonym *id* for strings as these are used as identifiers for function and predicate symbols. Next the terms are defined. The functions are straightforward but the variables use de Bruijn-indexing which needs an explanation.

type-synonym $id = \langle char\ list \rangle$

datatype $tm = Var\ nat \mid Fun\ id\ \langle tm\ list \rangle$

4.1.1.1 De Bruijn Indices

Instead of referring to variables with a name, x, y, etc. as we do on paper, the formalization uses natural numbers, 0, 1, and so on. The number specifies how many quantifiers you need to cross to get to the one which bound the variable. For instance, the formula $\forall x. \exists y. P(x, f(y))$ becomes $\forall \exists P(1, f(0))$ using de Bruijn indices: The x has one quantifier between its use and the quantifier binding it so its index is 1, and equivalently y becomes 0. This representation is an advantage when doing substitutions which will be discussed in section 5.1.2. Another advantage is that formulas which are equivalent up to a change of variable names, like $\forall x. P(x)$ and $\forall y. P(y)$, are now represented equally, $\forall P(0)$, so we can compare them using just structural equality [Bru72].

4.1.2 Formulas

Given the terms, the syntax of formulas is easily defined:

datatype *fm* = *Falsity* | *Pre id* ⟨*tm list*⟩ |
Imp fm fm | *Dis fm fm* | *Con fm fm* | *Exi fm* | *Uni fm*

There is a constant, *Falsity*, for $\bot$, and a constructor, *Pre*, for predicates that takes an identifier and a list of terms. Furthermore each of the logical connectives we need get a binary constructor: *Imp* for $\rightarrow$, *Dis* for $\vee$ and *Con* for $\wedge$. Finally each of the quantifiers are given a unary constructor: *Exi* for existential quantification, $\exists$, and *Uni* for universal quantification, $\forall$. Note that because of the use of de Bruijn indices, the quantifiers need only take the quantified formula as argument; the variable is bound implicitly.

4.2 Semantics

Given the syntax, the semantics can now be defined.

4.2.1 Terms

The semantics of terms is defined first, as that of formulas depend on it. Two mutually, primitive recursive functions are defined, one for a single term, *semantics-*

term, and one for a list of terms, *semantics-list*:

primrec
semantics-term :: ‹$(nat \Rightarrow {'a}) \Rightarrow (id \Rightarrow {'a}\ list \Rightarrow {'a}) \Rightarrow tm \Rightarrow {'a}$› **and**
semantics-list :: ‹$(nat \Rightarrow {'a}) \Rightarrow (id \Rightarrow {'a}\ list \Rightarrow {'a}) \Rightarrow tm\ list \Rightarrow {'a}\ list$›
where
‹*semantics-term* $e\ f\ (Var\ n) = e\ n$› |
‹*semantics-term* $e\ f\ (Fun\ i\ l) = f\ i$ (*semantics-list* $e\ f\ l$)› |
‹*semantics-list* $e\ f\ [] = []$› |
‹*semantics-list* $e\ f\ (t\ \#\ l)$ = *semantics-term* $e\ f\ t\ \#$ *semantics-list* $e\ f\ l$›

It it instructive to look at the types, noting that *'a* corresponds to the domain $\mathcal{D}$ so a value of type *'a* corresponds to an element of $\mathcal{D}$. The first argument, e, corresponds to the environment and maps variables encoded as natural numbers to values of type *'a*. The second argument, *f*, corresponds to $\mathcal{F}$. As such it goes from an identifier, the function symbol, and a list of terms to *'a*.

A variable is looked up in the environment as evident in the first clause. For a function symbol, the list of terms is evaluated and the result is looked up in f along with the identifier. The function *semantics-list* is a specialization of *map*.

4.2.2 Formulas

Given the semantics of terms, that of formulas is defined below. Isabelle's own boolean values, *True* and *False* are used for the truth values T and F respectively.

primrec
semantics :: ‹$(nat \Rightarrow {'a}) \Rightarrow (id \Rightarrow {'a}\ list \Rightarrow {'a}) \Rightarrow (id \Rightarrow {'a}\ list \Rightarrow bool)$
$\Rightarrow fm \Rightarrow bool$› **where**
‹*semantics* $e\ f\ g$ *Falsity* = *False*› |
‹*semantics* $e\ f\ g\ (Pre\ i\ l) = g\ i$ (*semantics-list* $e\ f\ l$)› |
‹*semantics* $e\ f\ g\ (Imp\ p\ q)$ =
(*if semantics* $e\ f\ g\ p$ *then semantics* $e\ f\ g\ q$ *else True*)› |
‹*semantics* $e\ f\ g\ (Dis\ p\ q)$ =
(*if semantics* $e\ f\ g\ p$ *then True else semantics* $e\ f\ g\ q$)› |
‹*semantics* $e\ f\ g\ (Con\ p\ q)$ =
(*if semantics* $e\ f\ g\ p$ *then semantics* $e\ f\ g\ q$ *else False*)› |
‹*semantics* $e\ f\ g\ (Exi\ p)$ =
($\exists x.$ *semantics* ($\lambda n.$ *if* $n = 0$ *then* x *else* $e\ (n - 1)$) $f\ g\ p$)› |
‹*semantics* $e\ f\ g\ (Uni\ p)$ =
($\forall x.$ *semantics* ($\lambda n.$ *if* $n = 0$ *then* x *else* $e\ (n - 1)$) $f\ g\ p$)›

The first clause evaluates *Falsity* to *False* and the second clause looks up a predicate in g similarly to the evaluation of function symbols. The next three clauses specify the meaning of the logical connectives using *if-then-else*. It is possible to use Isabelle's own logical connectives instead for a somewhat more direct encoding of the semantics, but this is arguably harder for students coming from normal programming languages to understand.

The final two clauses use Isabelle's own quantifiers. These have the same meaning as discussed previously, namely that the containing formula must hold for some, respectively, all x. There may be infinitely many values of type *'a*, so we cannot naively check them all as we would have to do in a normal programming language. Thus access to these quantifiers in Isabelle itself is essential for a proper encoding of the semantics of first-order logic.

In the quantifier cases, the $\sigma[x \leftarrow d]$ notation used in the introduction is encoded using *if-then-else*. This encoding is a consequence of the de Bruijn indexing. Consider the concrete formula $\forall_a.\forall_b.\exists_c.P(2,1,0)$ where the quantifiers are named for convenience. Then when evaluating $\exists_c.P(2,1,0)$, e associates 0 with $\forall_b$ and 1 with $\forall_a$ as we have not crossed the existential quantifier yet. But when evaluating $P(2,1,0)$, e should associate 0 with $\exists_c$ instead, 1 with $\forall_b$ and 2 with $\forall_a$; everything has been shifted up. To emulate this, in the last two clauses, a different environment than the one given as argument is used in the recursive calls. In the new environment, variable 0 maps directly to the quantified x as wanted. For a variable $n \neq 0$, $n - 1$ is looked up in the old environment to account for quantifier, effectively doing the shifting by decrementing the variable.

Chapter 5

Formalizing Natural Deduction

We need to write a few functional programs before we can formalize the rules of natural deduction. These are developed first, then the formalized rules are given and finally the chapter concludes with some examples of proofs within Isabelle.

5.1 Utilities

Besides the ones described below, a function *member* with the suggested meaning is defined. Note that this benefits from the use of de Bruijn indices, as described in chapter 4, because formulas can be compared structurally.

5.1.1 New Constants

When eliminating the existential and introducing the universal quantifier, the side conditions state that the used constant must be new. To enforce this, the functions of the following types are defined to check that an identifier name is new:

new-term :: ⟨*id* ⇒ *tm* ⇒ *bool*⟩ *and*
new-list :: ⟨*id* ⇒ *tm list* ⇒ *bool*⟩
new :: ⟨*id* ⇒ *fm* ⇒ *bool*⟩
news :: ⟨*id* ⇒ *fm list* ⇒ *bool*⟩

The only interesting case is in *new-term* where the passed in identifier c is compared to the i of the inspected *Fun i l*: If they are equal, false is returned, otherwise *new-list c l* is called to ensure c is also new in l. The rest are straightforward primitive recursions over either lists or formulas.

5.1.2 Substitution

The invention of de Bruijn indexes was to make substitutions (in the lambda calculus) simpler [Bru72], but the details can still be somewhat tricky. The substitution defined here is meant to be used specifically after removing a quantifier, as in the natural deduction rules.

Consider the formula $\forall\exists P(0,1)$ where we want to specialize the outer quantified variable to some term t using the $\forall E$ rule. The quantified variable has index 0 by definition, so the substitution becomes $(\exists P(0,1))[t/0]$. It is tempting to reduce this to $\exists(P(0,1)[t/0])$ and then $\exists P(t,1)$, replacing 0 by t, but this would be wrong. Variable 0 in $\forall\exists P(0,1)$ refers to the existential, not the universal quantifier. Instead, we need to increment the variable we are substituting for when crossing a quantifier. This gives us the correct sequence:

$$(\exists P(0,1))[t/0] \rightsquigarrow \exists(P(0,1)[t/1]) \rightsquigarrow \exists P(0,t)$$

Unfortunately this is not the only complication. Imagine that t contains a variable, e.g. $t = f(0)$ referring to the nearest quantifier, and observe what happens when we do the substitution again:

$$(\exists P(0,1))[f(0)/0] \rightsquigarrow \exists(P(0,1)[f(0)/1]) \rightsquigarrow \exists P(0,f(0))$$

Now the 0 refers to the existential quantifier, but we meant it to refer to the one beyond that, variable 0 in the outer environment. Therefore when crossing a quantifier we need to increment not only the variable we are substituting for, but also the variables in the term we are inserting:

$$(\exists P(0,1))[f(0)/0] \rightsquigarrow \exists(P(0,1)[f(1)/1]) \rightsquigarrow \exists P(0,f(1))$$

This is done with the following two functions:

```
primrec
  inc-term :: ‹tm ⇒ tm› and
  inc-list :: ‹tm list ⇒ tm list› where
  ‹inc-term (Var n) = Var (n + 1)› |
  ‹inc-term (Fun i l) = Fun i (inc-list l)› |
  ‹inc-list [] = []› |
  ‹inc-list (t # l) = inc-term t # inc-list l›
```

Alas we have forgotten something. The substitution is performed after removing a quantifier, but there might be variables that pointed beyond that quantifier; now these all point one level too far! The function that does substitution on terms handles this by comparing the variable we substitute for (corresponding to the number of quantifiers crossed) with the encountered variable and acting accordingly:

```
primrec
  sub-term :: ‹nat ⇒ tm ⇒ tm ⇒ tm› and
  sub-list :: ‹nat ⇒ tm ⇒ tm list ⇒ tm list› where
  ‹sub-term v s (Var n) =
    (if n < v then Var n else if n = v then s else Var (n − 1))› |
  ‹sub-term v s (Fun i l) = Fun i (sub-list v s l)› |
  ‹sub-list v s [] = []› |
  ‹sub-list v s (t # l) = sub-term v s t # sub-list v s l›
```

Finally all the bits can be put together to define substitution on formulas:

```
primrec sub :: ‹nat ⇒ tm ⇒ fm ⇒ fm› where
  ‹sub v s Falsity = Falsity› |
  ‹sub v s (Pre i l) = Pre i (sub-list v s l)› |
  ‹sub v s (Imp p q) = Imp (sub v s p) (sub v s q)› |
  ‹sub v s (Dis p q) = Dis (sub v s p) (sub v s q)› |
  ‹sub v s (Con p q) = Con (sub v s p) (sub v s q)› |
  ‹sub v s (Exi p) = Exi (sub (v + 1) (inc-term s) p)› |
  ‹sub v s (Uni p) = Uni (sub (v + 1) (inc-term s) p)›
```

5.2 Formalized Rules

The full set of rules is given below as an inductive definition. *OK p z* means that the formula p can be derived from the list of assumptions z, resembling the turnstile notation introduced in chapter 3, which would be $z \vdash p$.

```
inductive OK :: ‹fm ⇒ fm list ⇒ bool› where
  Assume: ‹member p z ⟹ OK p z› |
  Boole: ‹OK Falsity ((Imp p Falsity) # z) ⟹ OK p z› |
  Imp-E: ‹OK (Imp p q) z ⟹ OK p z ⟹ OK q z› |
  Imp-I: ‹OK q (p # z) ⟹ OK (Imp p q) z› |
  Dis-E: ‹OK (Dis p q) z ⟹ OK r (p # z) ⟹ OK r (q # z) ⟹ OK
r z› |
  Dis-I1: ‹OK p z ⟹ OK (Dis p q) z› |
  Dis-I2: ‹OK q z ⟹ OK (Dis p q) z› |
  Con-E1: ‹OK (Con p q) z ⟹ OK p z› |
  Con-E2: ‹OK (Con p q) z ⟹ OK q z› |
  Con-I: ‹OK p z ⟹ OK q z ⟹ OK (Con p q) z› |
  Exi-E: ‹OK (Exi p) z ⟹ OK q ((sub 0 (Fun c []) p) # z) ⟹
    news c (p # q # z) ⟹ OK q z› |
  Exi-I: ‹OK (sub 0 t p) z ⟹ OK (Exi p) z› |
  Uni-E: ‹OK (Uni p) z ⟹ OK (sub 0 t p) z› |
  Uni-I: ‹OK (sub 0 (Fun c []) p) z ⟹ news c (p # z) ⟹ OK (Uni p)
z›
```

Worthy of mention is the *Assume* rule allowing us to conclude any formula in the assumptions and obviating the need for a special copy rule. Each rule is effectively a function from premises and side conditions to a conclusion. Take for instance the *Exi-E* rule. By providing a proof, *OK (Exi p) z*, of formula *Exi p* from assumptions z, a proof of q with access to $p[c/0]$ as an additional assumption, and a proof, *news c (p # q # z)*, that c is new to both formulas and assumptions, we can obtain a proof, *OK q z*, that q can be derived from z.

5.3 Example Proofs

With the rules formalized in Isabelle it is now possible to prove some formulas within the formalization.

5.3.1 Reflexivity

Below is a proof of $p \to p$ using the declarative proof style.

lemma ⟨*OK* (*Imp* (*Pre* ''p'' []) (*Pre* ''p'' [])) []⟩
proof −
  have ⟨*OK* (*Pre* ''p'' []) [(*Pre* ''p'' [])]⟩ **by** (*rule Assume*) *simp*
  then show ⟨*OK* (*Imp* (*Pre* ''p'' []) (*Pre* ''p'' [])) []⟩ **by** (*rule Imp-I*)
qed

The proof visually resembles the textbook proof with the proven formula last, along with the inference rule $\to I$/*Imp-I*, and the premise on the line above:

$$\frac{p \vdash p}{\vdash p \to p} \to I$$

5.3.2 Modus Tollens

In the declarative style, the axioms appear before the conclusions. For longer proofs the procedural proof style can be more applicable because it allows us to work from the conclusion and back to the axioms more easily. The proof of modus tollens from section 3.2.1 might look like the following using this style.

lemma *modus-tollens*: ⟨*OK* (*Imp*
  (*Con* (*Imp* (*Pre* ''p'' []) (*Pre* ''q'' [])) (*Imp* (*Pre* ''q'' []) *Falsity*))
  (*Imp* (*Pre* ''p'' []) *Falsity*)) []⟩
  apply (*rule Imp-I*)
  apply (*rule Imp-I*)
  apply (*rule Imp-E*)
   apply (*rule Con-E2*)
   apply (*rule Assume*)
   apply *simp*
  apply (*rule Imp-E*)
   apply (*rule Con-E1*)
   apply (*rule Assume*)
   apply *simp*
  apply (*rule Assume*)
  apply *simp*
  done

Here the intermediate steps are not visible as with the declarative approach, but after applying a rule, Isabelle automatically introduces the required premises as new subgoals that can be proven by further application of rules. The indentation of the *apply* command matches the number of subgoals. In this style the premises of a rule are proven after they are used, resembling how the proof is prepared. The formalized proof uses specific predicates p and q instead of arbitrary formulas A and B, but since the proof does not rely on p and q being predicates, treating them like arbitrary formulas, the proofs can be considered equivalent.

5.3.3 Socrates is Mortal

An example that provides a few more complications when formalized in the procedural style is Aristotle's syllogism about Socrates. The textbook proof was given in section 3.2.2 and the formalized proof can be seen below.

```
lemma Socrates-is-mortal: ‹OK (Imp
  (Con (Uni (Imp (Pre ''h'' [Var 0]) (Pre ''m'' [Var 0])))
      (Pre ''h'' [Fun ''s'' []]))
  (Pre ''m'' [Fun ''s'' []])) []›
  apply (rule Imp-I)
  apply (rule Imp-E [where p=‹Pre ''h'' [Fun ''s'' []]›])
   apply (subgoal-tac ‹OK (sub 0 (Fun ''s'' [])
     (Imp (Pre ''h'' [Var 0]) (Pre ''m'' [Var 0]))) -›)
    apply simp
   apply (rule Uni-E)
   apply (rule Con-E1)
   apply (rule Assume)
   apply simp
  apply (rule Con-E2)
  apply (rule Assume)
  apply simp
  done
```

Two things are worth noting. First the application of the *Imp-E* rule comes before the premises, so Isabelle is unable to tell that formula p should be *h(s)*, only that the two new subgoals should be that p implies the current goal and that p itself can be proven. To resolve this issue, p is specified explicitly when applying the rule using *[where p=...]*.

Second the *Uni-E* rule expects the goal to be of the form *OK (sub 0 t p) z*, in our case

OK (sub 0 (Fun ''s'' []) (Imp (Pre ''h'' [Var 0]) (Pre ''m'' [Var 0]))) -

but what the state actually is at this point in the proof is

OK (Imp (Pre ''h'' [Fun ''s'' []]) (Pre ''m'' [Fun ''s'' []])) -

An underscore is used which Isabelle renders as a hyphen but more importantly fills in automatically with the correct list of assumptions. We can recognize that by performing the substitution in the first goal we reach the second, so the two are equivalent, but the system does not do this automatically. Therefore the first goal is introduced as a subgoal using *subgoal-tac*, effectively rewriting the goal. To be able to do this however, we need to prove that the first formula implies the second, so that by proving the first we have actually proven the second; this is easily done using the simplifier. After discharging that goal the remaining goal has the correct form and the *Uni-E* rule can be applied.

Thus with a little work the textbook examples can be formalized and checked by Isabelle. This ensures that the rules are applied correctly and no subgoals are forgotten, ensuring that the proof is reduced down to axioms.

Chapter 6

Formalizing Soundness

With the syntax, semantics and inference rules formalized, we are now in a position to formally prove the soundness of the rules. That is, that the rules can only be used to derive valid formulas. The proof will be by induction over the inference rules. The base case for the induction will be the *Assume* rule which is not premised on any other proofs and the rest of the rules are proven as part of the induction step. Thus we can assume that the premises they rely on are valid formulas and we need to show that the formula derived by the rule using these premises is also valid. If these things hold the induction principle states that we can only derive valid formulas and Isabelle is able to apply this reasoning for us.

6.1 Lemmas

Before the main theorem we need to prove some auxiliary lemmas to help us. This decomposition has the additional benefit of making it easier to understand and to maintain the proofs because each proof in itself can be shorter.

6.1.1 Built-In Logical Connectives

While the semantics are formalized using *if-then-else*, Isabelle is better at reasoning about the equivalent logical connectives directly. Therefore an equivalence between the two is proven and added to the simplifier:

```
lemma symbols [simp]:
  ‹(if p then q else True) = (p ⟶ q)›
  ‹(if p then True else q) = (p ∨ q)›
  ‹(if p then q else False) = (p ∧ q)›
  by simp-all
```

Adding a lemma like this to the simplifier is an obvious choice as it would only clutter the rest of the proofs if we were to add it explicitly every time.

6.1.2 Environment Extension

Furthermore the extension of the environment used for the semantics of quantifiers, as explained in chapter 4, is declared as its own function, *put*, and the equivalence added to Isabelle's simplifier. This makes it easier to prove lemmas about *put* later.

```
fun put :: ‹(nat ⇒ 'a) ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a› where
  ‹put e v x = (λn. if n < v then e n else if n = v then x else e (n − 1))›
```

```
lemma simps [simp]:
  ‹semantics e f g (Exi p) = (∃x. semantics (put e 0 x) f g p)›
  ‹semantics e f g (Uni p) = (∀x. semantics (put e 0 x) f g p)›
  by simp-all
```

Two such lemmas are the following. The first, *increment*, describes a relation between the semantic *put*, and the syntactic *inc-term* and *inc-list* functions:

```
lemma increment:
  ‹semantics-term (put e 0 x) f (inc-term t) = semantics-term e f t›
  ‹semantics-list (put e 0 x) f (inc-list l) = semantics-list e f l›
  by (induct t and l rule: semantics-term.induct semantics-list.induct)
simp-all
```

Looking just at the top part, the left hand side has an incremented term, *inc-term t*, where every variable has been incremented, and an environment, *put e 0 x*, where some term x has been *put* at index 0 and the rest of the indices are shifted one up compared to e. The lemma states that the semantics of the incremented term in the shifted environment is the same as the semantics of the original term in the original environment. The reasoning being that increments and shifts align perfectly. This is proven by mutual induction over the mutually recursive calls made by *inc-term* and *inc-list* and the four resulting subgoals are proven automatically by the simplifier.

Moreover a commutation property of *put* is proven automatically by Isabelle:

lemma *commute*: ‹*put* (*put e v x*) *0 y* = *put* (*put e 0 y*) (*v* + *1*) *x*›
 by *fastforce*

It states that *put*ting a new value x at index v and then another value y at index 0 is equivalent to first *put*ting y at 0 and then x at $v+1$, where v is incremented to account for the already *put y*.

6.1.3 New Constants

The following lemma relates the *new*, *news* and *list-all* functions and is proven by structural induction over the list of formulas z. If c is new for every element of z then *news c z* and vice versa.

lemma *allnew* [*simp*]: ‹*list-all* (*new c*) *z* = *news c z*›
 by (*induct z*) *simp-all*

If a constant does not appear in a term/formula, it should make no difference for the semantics what value the constant has in the interpretation. This is demonstrated by the following two lemmas.

lemma *map′* [*simp*]:
 ‹*new-term n t* ⟹
  semantics-term e (*f*(*n* := *x*)) *t* = *semantics-term e f t*›
 ‹*new-list n l* ⟹
  semantics-list e (*f*(*n* := *x*)) *l* = *semantics-list e f l*›
 by (*induct t* **and** *l rule*: *semantics-term.induct semantics-list.induct*)
auto

lemma *map* [*simp*]:
⟨*new n p* $\Longrightarrow$ *semantics e* $(f(n := x))$ *g p* = *semantics e f g p*⟩
by (*induct p arbitrary*: *e*) *simp-all*

Here *f(n := x)* means the same as the $f[n \leftarrow x]$ notation used in the introduction. That is, the same function as *f*, except for input n where the value is now x. Since n is new in t, l and p respectively, the two sides of the equal sign are equal in all three cases. For *map* the induction is over arbitrary e because, as discussed previously regarding the semantics, the recursive call in the quantifier cases is made on an updated environment.

It is useful to extend the *map* lemma to a list of formulas. This is done in two steps. First as *allmap'* using *list-all* and *new* and by induction over the list of formulas. Since the simplifier has access to *map* the two induction cases are automatically proven.

lemma *allmap'* [*simp*]: ⟨*list-all* (λp. *new c p*) *z* $\Longrightarrow$
list-all (*semantics e* $(f(c := m))$ *g*) *z* = *list-all* (*semantics e f g*) *z*⟩
by (*induct z*) *simp-all*

Second as *allmap* which uses *news* directly to specify the freshness constraint. This is proven automatically because *allnew*, which expresses exactly that the premise of *allmap'* and *allmap* are equivalent, was added to the simplifier.

lemma *allmap* [*simp*]: ⟨*news c z* $\Longrightarrow$
list-all (*semantics e* $(f(c := m))$ *g*) *z* = *list-all* (*semantics e f g*) *z*⟩
by *simp*

6.1.4 Substitution

Finally we need a substitution lemma before tackling the main soundness proof. For terms and lists of terms it is proven similarly to some of the other lemmas:

lemma *substitute'* [*simp*]:
⟨*semantics-term e f* (*sub-term v s t*) =
semantics-term (*put e v* (*semantics-term e f s*)) *f t*⟩
⟨*semantics-list e f* (*sub-list v s l*) =
semantics-list (*put e v* (*semantics-term e f s*)) *f l*⟩
by (*induct t* **and** *l rule*: *semantics-term.induct semantics-list.induct*)
simp-all

This relates the syntactic variables and semantic environments, and expresses that the semantics of a term $t[s/v]$, where s is substituted for variable v, is exactly the same as the semantics of the original term t in the same environment except *semantics-term e f s* has been *put* at index v. This formalizes the notion we have of the correspondence between substitution and the environment: The value of a variable is looked up in the environment, so we can also just substitute it with that value beforehand and vice versa. Extending this to formulas is done by induction over the formula and requires a little bit of manual proving in the quantifier cases. The induction is over arbitrary e, v and t so the induction hypothesis can be applied at different values of these. These cases are similar, so only the existential case is shown below. The proof is done by rewriting the left-hand side of the equal sign into the right.

```
lemma substitute [simp]:
  ‹semantics e f g (sub v t p) =
   semantics (put e v (semantics-term e f t)) f g p›
proof (induct p arbitrary: e v t)
  case (Exi p)
  have ‹semantics e f g (sub v t (Exi p)) =
      (∃x. semantics (put e 0 x) f g (sub (v + 1) (inc-term t) p))›
    by simp
  also have ‹... = (∃x. semantics (put (put e 0 x) (v + 1)
      (semantics-term (put e 0 x) f (inc-term t))) f g p)›
    using Exi by simp
  also have ‹... =
      (∃x. semantics (put (put e v (semantics-term e f t)) 0 x) f g p)›
    using commute increment(1) by metis
  finally show ?case
    by simp
```

The first *have* is derived from the definition of *semantics* and *sub*. The next *have* is obtained using the induction hypothesis with $e = put\ e\ 0\ x$, $v = v + 1$ and $t = inc\text{-}term\ t$. It is now apparent that *put e 0 x* cancels out with *inc-term* using *increment*. The third *have* does this rewriting and also rewrites the *put*s using the *commute* lemma. This final line matches the definition of *semantics* in the *Exi* case so from there the case can be proven by the simplifier.

This is the only place the *increment* and *commute* lemmas are used which is why I have not added them to the simplifier. These lemmas enable slightly large steps to be taken and it is helpful when reading the proof to see immediately what justifies the step.

It is noted elsewhere that there are numerous subtleties in the use of de Bruijn indices with regards to the substitution lemma [BU07]. These subtleties are formalized here in terms of the *commute* and *increment* lemmas and any others are handled automatically by Isabelle's simplifier. If one really wants to understand this proof, these subtleties may pose a problem, and using a nominal approach [BU07] with disciplined names over raw indices may be a better fit.

6.2 Soundness

Soundness is proven in two steps, first as the lemma *soundness'* that allows arbitrary assumptions which are all assumed valid and then as the theorem *soundness* with no assumptions which follows easily from this. First of all a lemma is introduced that makes proofs using the *Assume* rule easier going forward.

```
lemma member-set [simp]: ‹p ∈ set z = member p z›
  by (induct z) simp-all
```

The lemma relates the *member* function with the built-in set membership in Isabelle and is proven by structural induction on the list. This also allows the *Assume* case in the soundness proof to be discharged automatically by the simplifier. In fact all except the *Exi-E* and *Uni-I* cases are proven automatically by the simplifier. The *Exi-E* case is proven thusly:

```
lemma soundness':
  ‹OK p z ⟹ list-all (semantics e f g) z ⟹ semantics e f g p›
proof (induct p z arbitrary: f rule: OK.induct)
  case (Exi-E p z q c)
  then obtain x where ‹semantics (put e 0 x) f g p›
    by auto
  then have ‹semantics (put e 0 x) (f(c := λw. x)) g p›
    using ‹news c (p # q # z)› by simp
  then have ‹semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)›
    by simp
  then have
    ‹list-all (semantics e (f(c := λw. x)) g) (sub 0 (Fun c []) p # z)›
    using Exi-E by simp
  then have ‹semantics e (f(c := λw. x)) g q›
    using Exi-E by blast
  then show ‹semantics e f g q›
    using ‹news c (p # q # z)› by simp
next
```

Reading from the top and down, the proof proceeds as follows. According to the induction hypothesis of the *Exi-E p z q c* case, an x exists which, when *put* at index 0 in the environment, makes the formula p true. This is obtained and because the constant c is fresh by the side condition of the rule, p also holds for $f(c := \lambda w.x)$ by the *map* lemma. Next the *substitute* lemma allows us to substitute c, which evaluates to x, into p instead of using the extended environment. By assumption every formula in the list of assumptions z holds, so the fourth *have* follows easily. Having shown this, we can apply the induction hypothesis and *have ‹semantics e (f(c := λw.x)) g q›*. Now there is just one complication: the f is extended compared to the goal but because c is new, the *map* lemma can be used again to conclude the case.

The structure of the *Uni-I* case is similar with the exception that instead of obtaining a specific x and using this, we universally quantify it and prove that p holds for all x allowing us in the end to conclude *semantics e f g (Uni p)*.

```
    case (Uni-I c p z)
    then have ‹∀x. list-all (semantics e (f(c := λw. x)) g) z›
      by simp
    then have ‹∀x. semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)›
      using Uni-I by blast
    then have ‹∀x. semantics (put e 0 x) (f(c := λw. x)) g p›
      by simp
    then have ‹∀x. semantics (put e 0 x) f g p›
      using ‹news c (p # z)› by simp
    then show ‹semantics e f g (Uni p)›
      by simp
  qed (auto simp: list-all-iff)
```

From this lemma the *soundness* theorem follows directly:

```
  theorem soundness: ‹OK p [] ⟹ semantics e f g p›
    by (simp add: soundness')
```

With this proof formalized we know with very high certainty that the proof system, as formalized, is sound and that it can only be used to derive valid formulas.

One might choose to prove more of the cases in *soundness'* explicitly for pedagogical reasons, but having a short proof without unnecessary details is also an advantage in this regard.

6.2.1 A Consistency Corollary

Given soundness we can prove a consistency corollary about the proof system. This states that *something, but not everything can be proved*, where what can be proved is $A \to A$ and what cannot be proved is *Falsity*:

corollary ⟨∃ *p*. *OK p* []⟩ ⟨∃ *p*. ¬ *OK p* []⟩
proof –
  have ⟨*OK* (*Imp p p*) []⟩ **for** *p*
    by (*rule Imp-I*, *rule Assume*, *simp*)
  then show ⟨∃ *p*. *OK p* []⟩
    by *iprover*
next
  have ⟨¬ *semantics* (*e* :: *nat* ⇒ *unit*) *f g Falsity*⟩ **for** *e f g*
    by *simp*
  then show ⟨∃ *p*. ¬ *OK p* []⟩
    using *soundness* **by** *iprover*
qed

The *for* syntax here is another way to do universal quantification.

Chapter 7

Outline of Completeness Proof

The completeness proof given by Fitting in *First-Order Logic and Automated Theorem Proving* is explained below along with definitions of the necessary set theoretical concepts. Fitting's description is brief, so the following description also builds on Berghofer's formalization. Furthermore Fitting describes the following concepts first for propositional logic and then extends them to first-order logic, where I will present them for the latter directly. This presentation should aid the understanding of the formalization in the next chapter. Only closed formulas are considered in Fitting's proof, below and in the next chapter. Open formulas are discussed in chapter 9.

Following Fitting, the term *parameter* will be used as a synonym for constant and function symbols, primarily those that are introduced for metatheoretical purposes. Only non-empty domains and non-empty sets of constants are considered. Everything in this chapter is formalized in the next.

7.1 The Big Picture

The main part of the proof is the model existence theorem. This describes the consistency properties necessary for a set of formulas to have a model [Fit96, p. 59]. Consistency here means that no contradiction can be derived from the formulas. To reach this theorem, the notion of an abstract consistency property is developed and the initial notion is extended to an alternate consistency property of finite character. Given a set of formulas that live up to the requirements of this consistency property, a maximal set of formulas can be obtained by extending it repeatedly. This maximal set will by construction be a Hintikka set, which basically means that every formula in the set can be derived from formulas also in the set or that it is a term whose negation is not in the set, making it satisfiable without contradictions. Hintikka's lemma states that every Hintikka set is satisfiable (in a Herbrand model) [Fit96, prop. 5.6.2] which concludes the theorem.

Next it is shown that the set of all sets of formulas that cannot be used to derive falsehood is consistent. This is simpler to show for the first consistency property than for an alternate one of finite character, which is why the former is extended to the latter abstractly.

Finally completeness follows by contraposition. Using contraposition we can construct a *single* model for the negated formula instead of having to show directly that the formula is satisfied by *every* interpretation.

The rest of this chapter explains these concepts in more detail.

7.2 Types of Formulas

The proof is given for arbitrary formulas of different types: α, β, γ and δ.

α and β formulas are those formed by a binary connective, possibly prefixed by a negation, i.e. $A \circ B$ and $\neg(A \circ B)$ where $\circ$ is an arbitrary connective. Two components are defined for each formula type, α_1 and α_2 for α formulas and β_1 and β_2 for β formulas. α formulas are conjunctive which means that both components must be true for the formula to be true [Fit96, prop. 2.6.1]:

$$v_\sigma(\alpha) \equiv v_\sigma(\alpha_1) = T \text{ and } v_\sigma(\alpha_2) = T$$

β formulas are disjunctive which means that only one of the components need to be true for the formula to be true [Fit96, prop. 2.6.1]:

$$v_\sigma(\beta) \equiv v_\sigma(\beta_1) = T \text{ or } v_\sigma(\beta_2) = T$$

The components are given in table 7.1 [Fit96, table 2.2].

Conjunctive			Disjunctive		
α	α_1	α_2	β	β_1	β_2
$A \wedge B$	A	B	$\neg(A \wedge B)$	$\neg A$	$\neg B$
$\neg(A \vee B)$	$\neg A$	$\neg B$	$A \vee B$	A	B
$\neg(A \to B)$	A	$\neg B$	$A \to B$	$\neg A$	B

Table 7.1: Conjunctive and disjunctive formulas

The last two types of formula are those with quantifiers. γ formulas act universally while δ formulas act existentially.

$$v_\sigma(\gamma) \equiv v_\sigma(\gamma(t)) = T \text{ for all closed terms } t$$
$$v_\sigma(\delta) \equiv v_\sigma(\delta(t)) = T \text{ for any closed term } t$$

These are summed up in table 7.2 [Fit96, table 5.1]. In this case an *instance* of the formula is defined for each term t instead of its components. As an example $\neg(\exists x.A(x))$ acts universally because it says something about all x, namely that A is true for none of them.

Universal		Existential	
γ	$\gamma(t)$	δ	$\delta(t)$
$\forall x.A(x)$	$A[x/t]$	$\exists x.A(x)$	$A[x/t]$
$\neg(\exists x.A(x))$	$\neg A[x/t]$	$\neg(\forall x.A(x))$	$\neg A[x/t]$

Table 7.2: Universal and existential formulas

7.3 Consistency Properties

As mentioned, a set of formulas is considered consistent if no contradiction can be derived from it. A precise definition is given below. Let C be a collection of sets of formulas and let S be an arbitrary member of C. For C to be a consistency property the following conditions should be met [Fit96, def. 3.6.1, def. 5.8.1]:

1. For any predicate p applied to a list of terms $t_1, t_2, \ldots, t_n$, at most one of $p(t_1, t_2, \ldots, t_n)$ and $\neg p(t_1, t_2, \ldots, t_n)$ should be in S.
2. $\perp \notin S$ and $\neg\top \notin S$.
3. If $\neg\neg Z \in S$ then $S \cup \{Z\} \in C$.
4. If $\alpha \in S$ then $S \cup \{\alpha_1, \alpha_2\} \in C$.
5. If $\beta \in S$ then $S \cup \{\beta_1\} \in C$ or $S \cup \{\beta_2\} \in C$.
6. If $\gamma \in S$ then $S \cup \{\gamma(t)\} \in C$ for every closed term t.
7. If $\delta \in S$ then $S \cup \{\delta(p)\} \in C$ for some parameter p.

7.3.1 Alternate Consistency Property

The consistency property C can be be transformed into an alternate consistency property C^+ where condition 7 above is replaced with [Fit96, def. 5.8.3]:

7'. If $\gamma \in S$ then $S \cup \{\gamma(p)\} \in C$ for every parameter p new to S.

Using this definition we can instantiate a δ formula with any new parameter and the resulting set of formulas is still consistent. On the other hand if no parameters are new, the formula cannot be instantiated at all, where the previous definition guaranteed the existence of at least one usable parameter.

The members of C^+ are all the sets of formulas S such that $S\pi \in C$ for some parameter substitution π [Fit96, p. 131, top]. A parameter substitution is a renaming of parameters by a function. This way all the δ instances are added because there exists parameter substitutions mapping any unused parameter back to the fixed p.

7.3.2 Closure under Subsets

We can close a consistency property C under subsets and it will remain a consistency property. C is closed under subsets if for every $S \in C$ any subset $S' \subseteq S$ is also in C, $S' \in C$. Fitting gives no proof of this but the reasoning is intuitive: If a contradiction could be obtained by adding S' to C, it could already be obtained using S, so the result of closing C must still be consistent. If C is closed under subsets so will C^+ be, as any subset of a set is also added by the above construction.

7.3.3 Finite Character

An alternate consistency property C^+ closed under subsets can be extended to one, C^*, of finite character. That C^* is of finite character is a stronger property than it just being closed under subsets: C^* must be closed under subsets *and* for every set S where every finite subset $S' \subseteq S$ is a member of C^*, S must also be a member of C^*, $S \in C^*$. Thus C^* is obtained from C^+ by adding every set S where every finite subset of S is in C^+.

Fitting gives no proof for the correctness of this extension but the intuition is given in the following. Assume a contradiction can be derived from one of the added sets, S. This derivation terminates so it must use a finite subset of the formulas of S. But then there exists a contradictory finite subset of S that could not have been a part of C^+ so S could not have been added when forming C^* and the assumption that a contradiction can be derived cannot hold.

7.4 Maximal Consistent Sets

We will now see how to extend a formula $S \in C^*$ so that it is maximal in C^*. That S is maximal in C^* means that it is not a subset of any other set in C^* To do this we first need to understand the concept of chains.

7.4.1 Chains

A chain of sets $S_1, S_2, S_3, \ldots$ is a total ordering of those sets under some relation, here subsets, such that $S_1 \subseteq S_2 \subseteq S_3 \subseteq \ldots$. The least upper bound, $\bigcup_i S_i$ of a chain of sets, $S_i \in C^*$, is itself a member of C^*:

C^* is of finite character so if every finite subset of $\bigcup_i S_i$ is a member of C^*, so is $\bigcup_i S_i$ by construction. Consider an arbitrary subset $\{A_1, \ldots, A_k\} \subseteq \bigcup_i S_i$. These sets are part of a subset chain, so one of them, say S_n, will be a superset of all of them. By definition we know that $S_n \in C^*$ and since C^* is subset closed it follows that $\{A_1, \ldots, A_k\} \in C^*$ which is what we needed to show. Thus $\bigcup_i S_i \in C^*$ [Fit96, p. 61, top].

7.4.2 Extension

Given a set of formulas S, that is a member of C^*, we want to extend S so that it is maximal in C^*. To do this we go from an enumeration of all the sentences of the language, $X_1, X_2, X_3, \ldots$, to a sequence of members of C^*, $S_1, S_2, S_3, \ldots$. Each S_i leaves unused an infinite number of parameters. Since S is given to us fresh it contains no parameters. The initial element of the sequence is

$$S_1 = S$$

A subsequent element S_{n+1} is now defined from S_n as follows [Fit96, p. 131]:

$$S_{n+1} = \begin{cases} S_n & \text{if } S_n \cup \{X_n\} \notin C^* \\ S_n \cup \{X_n\} & \text{if } S_n \cup \{X_n\} \in C^* \text{ and } X_n \neq \delta \\ S_n \cup \{X_n\} \cup \delta(p) & \text{if } S_n \cup \{X_n\} \in C^* \text{ and } X_n = \delta \end{cases}$$

In the last case infinitely many parameters are new to $S_n \cup \{X_n\}$ and only one is used, leaving infinitely many unused in S_{n+1} as well.

For each S_n, $S_n \in C^*$ and $S_n \subseteq S_{n+1}$ by construction. Their union, $H = \bigcup_i S_i$, (the choice of name will become apparent shortly) has two important properties [Fit96, p. 132, top]:

1. $H \in C^*$ because each element S_n is finite and in C^* by construction and C^* is of finite character.
2. H is maximal in C^* because if any set $K \in C^*$ was a superset of it, then for some formula X_n, we have $X_n \in K$ but $X_n \notin H$. Since $X_n \notin H$, it follows from the construction of H that $S_n \cup \{X_n\} \notin C^*$. Then $S_n \cup \{X_n\} \subseteq K$ because $S_n \in H$, $X_n \in K$ and H is a subset of K. And since C^* is subset closed, $S_n \cup \{X_n\} \in C^*$ but this is a contradiction.

7.5 Hintikka's Lemma

Before reaching the model existence theorem, we need to consider Hintikka sets and Hintikka's lemma.

7.5.1 Hintikka Sets

The conditions for a set H of formulas to be a Hintikka set resemble those for a set of set of formulas to be a consistency property. They are given below [Fit96, def. 3.5.1, def. 5.6.1].

1. For any predicate p applied to a list of terms $t_1, t_2, \ldots, t_n$, at most one of $p(t_1, t_2, \ldots, t_n)$ and $\neg p(t_1, t_2, \ldots, t_n)$ should be in H.
2. $\bot \notin H$ and $\neg\top \notin H$.
3. If $\neg\neg Z \in H$ then $Z \in H$.
4. If $\alpha \in S$ then $\alpha_1 \in H$ and $\alpha_2 \in H$.
5. If $\beta \in H$ then $\beta_1 \in H$ or $\beta_2 \in H$.
6. If $\gamma \in H$ then $\gamma(t) \in H$ for every closed term t.
7. If $\delta \in H$ then $\delta(t) \in H$ for some closed term t.

An example Hintikka set is $\{(\exists x.p(x)) \wedge q, \exists x.p(x), q, p(c)\}$ where p is some predicate of arity 1, q a predicate of arity 0 and c some constant.

7.5.2 Herbrand Models

To understand Hintikka's lemma we need the concept of a Herbrand model. In a Herbrand model, the domain is exactly the closed terms of the language and every closed term is interpreted as itself [Fit96, def. 5.4.1]. Terms in Herbrand models are called *Herbrand terms.* Since Herbrand terms are closed by definition, no variables can occur in them. This means that any term t, open or closed, interpreted in a Herbrand model with variable assignment e, will equal itself where e is used as a substitution instead: Function symbols interpret to themselves because evaluating their list of terms will close them and this evaluation will have the same effect on variables as substitution [Fit96, def. 5.4.2]. The same property extends to formulas [Fit96, def. 5.4.3].

7.5.3 The Lemma

Now Hintikka's lemma states that if H is a Hintikka set, then H is satisfiable in a Herbrand model [Fit96, prop. 5.6.2]. Fitting outlines the following proof [Fit96, pp. 127–128] giving only the non-negated predicate case and the γ case.

First the model M is constructed by letting the domain be every closed term of the language and letting closed terms interpret to themselves as prescribed. A predicate $p(t_1, \ldots, t_n)$ is interpreted as true iff $p(t_1, \ldots, t_n) \in H$.

Consider a sentence $X \in H$; we want to show that M satisfies X. Fitting uses structural induction to show this, where in the quantifier cases the induction hypothesis applies to instantiations of the formula.

Suppose $X = \top$ then M satisfies it trivially. Supposing $X = \bot$ contradicts $X \in H$ so the case holds vacuously.

Suppose $X = p(t_1, \ldots, t_n)$, then every term $t_1, \ldots, t_n$ must be closed since X is a sentence with no quantifiers. Therefore the terms $t_1, \ldots, t_n$ interpret to themselves by the property of the Herbrand model and $p(t_1, \ldots, t_n)$ interpreted is simply $p(t_1, \ldots, t_n)$. We know by assumption that $X \in H$ and this is the condition for a predicate to be true, so M satisfies X in this case.

Suppose $X = \neg p(t_1, \ldots, t_n)$. Then by definition of Hintikka sets $p(t_1, \ldots, t_n) \notin H$, which means $p(t_1, \ldots, t_n)$ is interpreted as false by M so its negation is true as needed.

Suppose X is an α formula. Then the induction hypothesis applies to its components, α_1 and α_2. By definition of Hintikka sets these are both members of H, so by the induction hypothesis M satisfies them. Thus X is satisfied by M, by the semantics of α formulas.

Suppose X is a β formula. Then the induction hypothesis applies to its components, β_1 and β_2. By definition of Hintikka sets at least one of these is a member of H, and is satisfied by M by the induction hypothesis. Thus X is satisfied by M, by the semantics of β formulas.

Suppose X is a γ formula. Then for every closed term t, $X(t) \in H$ by definition of Hintikka sets and because $X(t)$ is an instantiation of X the induction hypothesis applies. Thus M satisfies $X(t)$ for every closed term t. Since the domain is exactly the set of closed terms, M satisfies X by semantics of γ formulas.

Suppose finally that X is a δ formula. Then there exists some closed t for which $X(t) \in H$ by definition of Hintikka sets and $X(t)$ is an instantiation of X so M satisfies $X(t)$. The existence of one such term is enough to satisfy a δ formula, so M satisfies X.

Thus M satisfies every sentence in H and since we only assumed that M is a Herbrand model and that H is a Hintikka set, this concludes the proof.

7.6 Model Existence Theorem

All of the above can now be combined into the model existence theorem. Given a set of formulas S, that is a member of a consistency property C, we want to construct a model for S. First C is extended to C^* where S is also a member of C^* by construction. Now $H = \bigcup_i S_i$ as constructed previously extends S and is a Hintikka set. This follows directly from the construction of C^* and H and the fact that H is maximal in C^* [Fit96, p. 132, top].

From this third property it follows that H is satisfiable in a Herbrand model and since S is a subset of H so is it. Thus there exists a model for S which concludes the theorem.

7.7 Completeness

The final thing we need for completeness is to show that the set of sets of formulas from which we cannot derive a contradiction, is a consistency property. This is shown in the next chapter by going through the conditions one at a time.

Given this and the model existence theorem, the following trick is employed to show completeness [Ber07a]. Consider an arbitrary valid formula p that we want to show is derivable. The proof uses the contraposition principle:

$$A \to B \equiv \neg B \to \neg A$$

Thus we show that if the formula cannot be derived then it is invalid, and this is equivalent to the statement that if it is valid then it can be derived. Assuming p cannot be derived means that the set $\{\neg p\}$ is consistent, for p cannot be derived so no contradiction is possible. Then by the model existence theorem a model can be obtained for $\neg p$. But then p cannot be valid because if it was, an interpretation would exists which satisfies both p and $\neg p$ and this is impossible. Thus it follows that any valid formula can be derived and the proof system is complete.

This completeness proof has the advantage that we only need to show very little about the particular proof system, namely its consistency. The rest of the proof is developed abstractly.

Chapter 8

Formalizing Completeness

This chapter extends the formalization of the syntax, semantics and soundness of natural deduction with the completeness proof given in the previous chapter. To aid the representation only excerpts are given here, with the proof available in its entirety online as mentioned previously.

8.1 Consistency Properties

We need to develop some utility functions and lemmas before we can express the textbook proof.

The formalized proof uses the syntax of formulas directly instead of the α, β, δ and γ types used in the textbook proof. This is arguably simpler but it also makes for some redundancy in declarations and proofs as evidenced by the formalized version of a consistency property below.

```
definition consistency :: ‹fm set set ⇒ bool› where
  ‹consistency C = (∀ S. S ∈ C ⟶
    (∀ p ts. ¬ (Pre p ts ∈ S ∧ Neg (Pre p ts) ∈ S)) ∧
    Falsity ∉ S ∧
```

$(\forall Z.\ Neg\ (Neg\ Z) \in S \longrightarrow S \cup \{Z\} \in C) \wedge$
$(\forall A\ B.\ Con\ A\ B \in S \longrightarrow S \cup \{A,\ B\} \in C) \wedge$
$(\forall A\ B.\ Neg\ (Dis\ A\ B) \in S \longrightarrow S \cup \{Neg\ A,\ Neg\ B\} \in C) \wedge$
$(\forall A\ B.\ Dis\ A\ B \in S \longrightarrow S \cup \{A\} \in C \vee S \cup \{B\} \in C) \wedge$
$(\forall A\ B.\ Neg\ (Con\ A\ B) \in S \longrightarrow$
$\quad S \cup \{Neg\ A\} \in C \vee S \cup \{Neg\ B\} \in C) \wedge$
$(\forall A\ B.\ Imp\ A\ B \in S \longrightarrow S \cup \{Neg\ A\} \in C \vee S \cup \{B\} \in C) \wedge$
$(\forall A\ B.\ Neg\ (Imp\ A\ B) \in S \longrightarrow S \cup \{A,\ Neg\ B\} \in C) \wedge$
$(\forall P\ t.\ closed\text{-}term\ 0\ t \longrightarrow Uni\ P \in S \longrightarrow S \cup \{sub\ 0\ t\ P\} \in C) \wedge$
$(\forall P\ t.\ closed\text{-}term\ 0\ t \longrightarrow Neg\ (Exi\ P) \in S \longrightarrow$
$\quad S \cup \{Neg\ (sub\ 0\ t\ P)\} \in C) \wedge$
$(\forall P.\ Exi\ P \in S \longrightarrow (\exists x.\ S \cup \{sub\ 0\ (Fun\ x\ [])\ P\} \in C)) \wedge$
$(\forall P.\ Neg\ (Uni\ P) \in S \longrightarrow$
$\quad (\exists x.\ S \cup \{Neg\ (sub\ 0\ (Fun\ x\ [])\ P)\} \in C)))\rangle$

This expresses the consistency property as a function from sets of sets of formulas to true or false. Each condition is expressed as an implication and they are all joined by conjunctions. The definition relies on the function *closed-term* which checks that a term is closed assuming a certain number of quantifiers have been passed — *closed* is similar but for formulas — and the following abbreviation:

abbreviation $Neg :: \langle fm \Rightarrow fm\rangle$ **where** $\langle Neg\ p \equiv Imp\ p\ Falsity\rangle$

8.1.1 Alternate Consistency Property

The declaration *alt-consistency* checks if a given set is an alternate consistency property and is obtained by replacing the last two lines above with the following:

$(\forall P\ x.\ (\forall a \in S.\ x \notin params\ a) \longrightarrow Exi\ P \in S \longrightarrow$
$\quad S \cup \{sub\ 0\ (Fun\ x\ [])\ P\} \in C) \wedge$
$(\forall P\ x.\ (\forall a \in S.\ x \notin params\ a) \longrightarrow Neg\ (Uni\ P) \in S \longrightarrow$
$\quad S \cup \{Neg\ (sub\ 0\ (Fun\ x\ [])\ P)\} \in C)$

As in the textbook proof this replaces the requirement for the existence of a specific parameter with a condition on all new parameters. The function *params* takes a formula and returns a set of all the identifiers that has been used as function symbols (and thus constant/parameter symbols) in that formula. This is used directly here instead of *news* because S is a set and *news* works on lists.

The construction of an alternate consistency property below matches the textbook description. Here $\{S.\ P\ S\}$ is set-builder notation meaning *the set of all elements S that satisfy the predicate P.*

definition *mk-alt-consistency* :: ‹*fm set set* ⇒ *fm set set*› **where**
‹*mk-alt-consistency* $C = \{S.\ \exists f.\ psubst\ f\ `\ S \in C\}$›

The *psubst* function has the following type, mapping the supplied function over every function symbol.

psubst :: ‹(*id* ⇒ *id*) ⇒ *fm* ⇒ *fm*›

We need to prove that this construction actually satisfies the conditions. This is done in the following theorem.

theorem *alt-consistency*:
assumes *conc*: ‹*consistency* C›
shows ‹*alt-consistency* (*mk-alt-consistency* C)› (**is** ‹*alt-consistency* ?*C'*›)

We *assume* that C is a consistency property and name this fact *conc*. Using this we *show* that the construction is an alternate consistency property and name the constructed set *?C'* (corresponding to C^+) for easier reference in the proof.

The proof starts by unfolding the definition of an alternative consistency property and *intro*ducing each conjunction as a new goal allowing us to prove them separately. To show that the conditions hold for all $S' \in ?C'$, we *fix* a specific one assuming only $S \in ?C'$. Furthermore we *obtain* the specific f that puts S' in the original C and call this mapped S', $?S$.

unfolding *alt-consistency-def*
proof (*intro allI impI conjI*)
fix S'

assume ‹$S' \in$?*C'*›
then obtain f **where** *sc*: ‹*psubst* f ‘ $S' \in C$› (**is** ‹?*S* $\in C$›)
unfolding *mk-alt-consistency-def* **by** *blast*

Let us look at the proof of the first condition, that both a predicate and its negation cannot be in S'.

fix *p ts*
show ‹¬ (*Pre p ts* $\in S'$ ∧ *Neg* (*Pre p ts*) $\in S'$)›
proof

assume $*$: ‹*Pre p ts* $\in S'$ $\wedge$ *Neg* (*Pre p ts*) $\in S'$›
then have ‹*psubst f* (*Pre p ts*) $\in$ *?S*›
 by *blast*
then have ‹*Pre p* (*psubst-list f ts*) $\in$ *?S*›
 by *simp*
then have ‹*Neg* (*Pre p* (*psubst-list f ts*)) $\notin$ *?S*›
 using *conc sc* **by** (*simp add*: *consistency-def*)
then have ‹*Neg* (*Pre p ts*) $\notin S'$›
 by *force*
then show *False*
 using $*$ **by** *blast*
qed

It is a proof of a negation, so we assume the un-negated proposition, *, and derive falsehood. First we apply the parameter substitution f to the original positive formula in S' and the result must then be in $?S$ by its construction. Parameter substitution of a predicate is done by applying the substitution to the list of terms which allows us to push the substitution under the *Pre* constructor. With that done we can conclude in the third *have* that the negation of the substituted formula cannot be in $?S$ using the original consistency conditions on C and the fact that $?S \in C$. But then the negation of the original formula cannot be in S' and this contradicts the assumption allowing us to derive *False* as needed.

The $\perp$, negation and α, β and γ cases are similar as these conditions are all equal between the two types of consistency property. Looking at one of the δ cases is more interesting. The first half is shown below.

{ fix *P x*
assume ‹$\forall$ *a* $\in S'$. *x* $\notin$ *params a*› **and** ‹*Exi P* $\in S'$›
moreover have ‹*psubst f* (*Exi P*) $\in$ *?S*›
 using *calculation* **by** *blast*
then have ‹$\exists$ *y*. *?S* $\cup$ {*sub 0* (*Fun y* []) (*psubst f P*)} $\in C$›
 using *conc sc* **by** (*simp add*: *consistency-def*)
then obtain *y* **where** ‹*?S* $\cup$ {*sub 0* (*Fun y* []) (*psubst f P*)} $\in C$›
 by *blast*

Here the proof is enclosed in curly braces, a so-called raw proof block. This allows us to reuse the fixed names in proofs of the other conditions without the assumptions applying to those as well. These are discussed further below. The premises of the condition are assumed and we show that the conclusion follows, namely that in $?C'$, $\delta \in S'$ has been extended with all possible instantiations $\delta(x)$ for all free x. This is done by showing that, no matter what the x is we can map it back to y, since it is free in S' and thus *mk-alt-consistency* must have added every instantiation.

This first half of the proof resembles the predicate case in that we look at the substituted formula in $?S$ and use the original consistency conditions to *obtain* the y which must exist since $?S \in C$. This is the y which can be used to instantiate the original formula keeping the result in C.

```
moreover have ‹psubst (f(x := y)) ` S' = ?S›
  using calculation by (simp cong add: image-cong)
then have ‹psubst (f(x := y)) `
    S' ∪ {sub 0 (Fun ((f(x := y)) x) []) (psubst (f(x := y)) P)} ∈ C›
  using calculation by auto
then have ‹∃f. psubst f `
    S' ∪ {sub 0 (Fun (f x) []) (psubst f P)} ∈ C›
  by blast
then show ‹S' ∪ {sub 0 (Fun x []) P} ∈ ?C'›
  unfolding mk-alt-consistency-def by simp }
```

The assumption that the fixed parameter x does not appear in S' allows uses to derive the first *have* above that uses $f(x := y)$ instead of f. This also applies to the extension of $?S$ from the original consistency property in the next *have*. The third *have* existentially quantifies the $f(x := y)$ making it resemble the definition of *mk-alt-consistency*. Thus the final *show* follows directly.

That the constructed consistency property is a subset of the original follows almost directly by using the identity function as f:

```
theorem mk-alt-consistency-subset: ‹C ⊆ mk-alt-consistency C›
  unfolding mk-alt-consistency-def
proof
  fix S
  assume ‹S ∈ C›
  then have ‹psubst id ` S ∈ C›
    by simp
  then have ‹∃f. psubst f ` S ∈ C›
    by blast
  then show ‹S ∈ {S. ∃f. psubst f ` S ∈ C}›
    by simp
qed
```

8.1.1.1 Proof Structure

A slight digression is worth it to discuss the choice of raw proof blocks which are somewhat rare and the structure of the proofs in general. Raw proof blocks are a feature of the declarative Isar language and thus do not appear in Berghofer's proof which uses *apply* commands exclusively. In the *apply* style the cases of a proof are proved in an order determined by their underlying declaration. This is not the case in the declarative style where we may prove cases in any order we choose and what case we are proving is apparent from either a *case* command or a combination of *assume* and *show*. Where relevant I have used this freedom to prove cases in the order of their corresponding type: α, β, γ and finally δ. Thus the formalization follows the proof in the previous chapter more closely.

The use of raw proof blocks is to minimize the overhead of the formalization compared to the textbook proof. With raw proof blocks the structure for each case becomes something like:

```
{ fix X Y
  assume A B C
  — . . .
  show D by blast }
```

This relies on the correct introduction of implications etc. in the initial *proof* command. Alternatively, and perhaps more traditionally, we can state beforehand what case we are proving and then do the introduction, like this:

```
proof –
 show ‹∀ X Y. A ⟶ B ⟶ C ⟶ D›
 proof (intro allI impI)
  fix X Y
  assume A B C
  — . . .
  show D by blast
 qed
```

But the redundancy is readily apparent, the assumptions and conclusion are stated twice and we need to wrap everything in *proof*, *qed*. There are variations on this latter choice but in all cases it is more verbose than the raw proof blocks.

8.1.2 Closure under Subsets

As mentioned earlier, a set of sets is subset closed by extending it with every subset of every member set. This is done below along with checking that a given set is subset closed:

definition *close* :: ⟨*fm set set* ⇒ *fm set set*⟩ **where**
⟨*close* $C = \{S.\ \exists S' \in C.\ S \subseteq S'\}$⟩

definition *subset-closed* :: ⟨*'a set set* ⇒ *bool*⟩ **where**
⟨*subset-closed* $C = (\forall S' \in C.\ \forall S.\ S \subseteq S' \longrightarrow S \in C)$⟩

The proof that a consistency property remains a consistency property when subset closed relies crucially on the following lemma. This states that if a set S extended with an element x is in the original set, C, any subset of S, S', extended with the same element will be in *close* C.

lemma *subset-in-close*:
assumes ⟨$S' \subseteq S$⟩ **and** ⟨$S \cup x \in C$⟩
shows ⟨$S' \cup x \in$ *close* C⟩

Given this we can prove a larger theorem:

theorem *close-consistency*:
assumes *conc*: ⟨*consistency* C⟩
shows ⟨*consistency* (*close* C)⟩

Again each condition is proved separately and since all the cases are similar I will just show the *Con* case here. The proof is done by looking at an arbitrary set $S' \in$ *close* C in relation to a specific $S \in C$ where $S' \subseteq S$ which can be thought of as the set S' originated from.

{ fix A B
assume ⟨*Con* A $B \in S'$⟩
then have ⟨*Con* A $B \in S$⟩
using ⟨$S' \subseteq S$⟩ **by** *blast*
then have ⟨$S \cup \{A, B\} \in C$⟩
using ⟨$S \in C$⟩ *conc* **unfolding** *consistency-def* **by** *simp*
then show ⟨$S' \cup \{A, B\} \in$ *close* C⟩
using ⟨$S' \subseteq S$⟩ *subset-in-close* **by** *blast* **}**

The conjunction is in the smaller set, so it is also in the larger one. Because C is a consistency property we can extend S with the conjunction's components, A and B, and the resulting set is also in C. Finally by definition of *close*, it follows that S' extended with the components must then be in *close* C.

Lastly we can prove that turning a subset closed consistency property into an alternate one preserves its property of being closed under subsets. In fact we prove a more general version of the theorem where we do not even assume that C is a consistency property.

```
theorem mk-alt-consistency-closed:
  assumes ‹subset-closed C›
  shows ‹subset-closed (mk-alt-consistency C)›
  unfolding subset-closed-def
proof (intro ballI allI impI)
  fix S S'
  assume ‹S ∈ mk-alt-consistency C› and ‹S' ⊆ S›
  then obtain f where *: ‹psubst f ` S ∈ C›
    unfolding mk-alt-consistency-def by blast
  moreover have ‹psubst f ` S' ⊆ psubst f ` S›
    using ‹S' ⊆ S› by blast
  ultimately have ‹psubst f ` S' ∈ C›
    using ‹subset-closed C› unfolding subset-closed-def by blast
  then show ‹S' ∈ mk-alt-consistency C›
    unfolding mk-alt-consistency-def by blast
qed
```

In the proof we consider a set $S \in$ *mk-alt-consistency* C and an arbitrary subset of this, $S' \subseteq S$, showing that this subset is also in *mk-alt-consistency* C. This is done by obtaining the parameter substitution f which made S part of *mk-alt-consistency* C in the first place and showing that this would also make S' part of the alternate consistency property, precisely because it is subset closed.

8.1.3 Finite Character

We recall the definition of a set of finite character and its construction:

```
definition finite-char :: ‹'a set set ⇒ bool› where
  ‹finite-char C =
    (∀S. S ∈ C = (∀S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ C))›

definition mk-finite-char :: ‹'a set set ⇒ 'a set set› where
  ‹mk-finite-char C = {S. ∀S'. S' ⊆ S ⟶ finite S' ⟶ S' ∈ C}›

theorem finite-char: ‹finite-char (mk-finite-char C)›
  unfolding finite-char-def mk-finite-char-def by blast
```

The last theorem states that the function matches the specification.

Given these we can now prove that an alternate consistency property extended to one of finite character is still an alternate consistency property. First the theorem is introduced and a couple of useful facts are derived as we have seen previously.

```
theorem finite-alt-consistency:
  assumes altconc: ‹alt-consistency C›
    and ‹subset-closed C›
  shows ‹alt-consistency (mk-finite-char C)›
  unfolding alt-consistency-def
proof (intro allI impI conjI)
  fix S
  assume ‹S ∈ mk-finite-char C›
  then have finc: ‹∀ S' ⊆ S. finite S' ⟶ S' ∈ C›
    unfolding mk-finite-char-def by blast

  have ‹∀ S' ∈ C. ∀ S ⊆ S'. S ∈ C›
    using ‹subset-closed C› unfolding subset-closed-def by blast
  then have sc: ‹∀ S' x. S' ∪ x ∈ C ⟶ (∀ S ⊆ S' ∪ x. S ∈ C)›
    by blast
```

For brevity only the *Exi* case is included here. This is a δ case and the reason that we use an alternate consistency property instead of the original. We are looking at an $S \in$ *mk-finite-char* C where C is a subset closed alternate consistency property and need to show that $S \cup \{\delta(x)\} \in$ *mk-finite-char* C. To do this we look at a finite subset S' of S. Since we are transforming C to be of finite character, it is enough to show that $S' \in C$; the original claim follows by the finite character construction. We start by considering this S' without $\delta(x)$ but extended with δ:

```
  { fix P x
    assume *: ‹Exi P ∈ S› and ‹∀ a ∈ S. x ∉ params a›
    show ‹S ∪ {sub 0 (Fun x []) P} ∈ mk-finite-char C›
      unfolding mk-finite-char-def
    proof (intro allI impI CollectI)
      fix S'
      let ?S' = ‹(S' − {sub 0 (Fun x []) P}) ∪ {Exi P}›

      assume ‹S' ⊆ S ∪ {sub 0 (Fun x []) P}› and ‹finite S'›
      then have ‹?S' ⊆ S›
        using * by blast
      moreover have ‹finite ?S'›
        using ‹finite S'› by blast
```

This $?S'$ is still finite so $?S' \in C$. Moreover, because of the formulation of an alternate consistency property we can assume that the x we use for the instantiation is free in S. Therefore it is also free in the finite subset S', so $?S' \cup \{\delta(x)\} \in C$ because it is consistent:

ultimately have ‹$?S' \in C$›
  using *finc* **by** *blast*
moreover have ‹$\forall a \in ?S'.\ x \notin$ *params* a›
  using ‹$\forall a \in S.\ x \notin$ *params* a› ‹$?S' \subseteq S$› **by** *blast*
ultimately have ‹$?S' \cup \{$*sub 0* (*Fun* x []) $P\} \in C$›
  using *altconc* ‹$\forall a \in S.\ x \notin$ *params* a›
  unfolding *alt-consistency-def* **by** *blast*
then show ‹$S' \in C$›
  using *sc* **by** *blast*
qed }

And this was all we needed to show since $?S' \cup \{\delta(x)\} = S'$. Had we used an ordinary consistency property, we could obtain a suitable x for extending S' but this x might not be useful for extending S since S is bigger. Here we can use the x given for S to extend S' with, obviating the problem.

8.2 Enumerating Data Types

Fitting's proof relies on the ability to enumerate the sentences of the language. Berghofer develops his own machinery using diagonalization to do this [Ber07a]. In the meantime however, a library called *Countable* has been developed for Isabelle which automates this process [KHB16]. By importing this library we can obtain enumeration of terms and formulas using just the following:

instantiation *tm* :: *countable* **begin**
instance by *countable-datatype*
end

instantiation *fm* :: *countable* **begin**
instance by *countable-datatype*
end

This provides the two functions, *to-nat* and *from-nat* and some lemmas about them. Thus the sentence X_n in the textbook proof becomes *from-nat* n in the formalization.

8.3 Maximal Consistent Sets

I give the highlights of this development.

8.3.1 Chains

First we need a definition of chains under the subset relation, where we use a function from the natural numbers to provide each set in the chain:

definition *is-chain* :: ‹$(nat \Rightarrow 'a\ set) \Rightarrow bool$› **where**
‹$is\text{-}chain\ f = (\forall n.\ f\ n \subseteq f\ (Suc\ n))$›

Several lemmas are developed before proving that unions of subset chains from alternate consistency properties of finite character are themselves part of that property. These are omitted for brevity, but the conclusion can be seen below.

theorem *chain-union-closed*:
assumes ‹$finite\text{-}char\ C$› **and** ‹$is\text{-}chain\ f$› **and** ‹$\forall n.\ f\ n \in C$›
shows ‹$(\bigcup n.\ f\ n) \in C$›

8.3.2 Extension

First we define a recursive function *extend* for obtaining a specific element, S_n, in the sequence:

primrec *extend* :: ‹$fm\ set \Rightarrow fm\ set\ set \Rightarrow (nat \Rightarrow fm) \Rightarrow nat \Rightarrow fm\ set$›
where
‹$extend\ S\ C\ f\ 0 = S$› |
‹$extend\ S\ C\ f\ (Suc\ n) = (if\ extend\ S\ C\ f\ n \cup \{f\ n\} \in C$
$then\ (if\ (\exists p.\ f\ n = Exi\ p)$
$then\ extend\ S\ C\ f\ n \cup \{f\ n\} \cup \{sub\ 0$
$(Fun\ (SOME\ k.\ k \notin (\bigcup p \in extend\ S\ C\ f\ n \cup \{f\ n\}.\ params\ p))\ [])$
$(dest\text{-}Exi\ (f\ n))\}$
$else\ if\ (\exists p.\ f\ n = Neg\ (Uni\ p))$
$then\ extend\ S\ C\ f\ n \cup \{f\ n\} \cup \{Neg\ (sub\ 0$
$(Fun\ (SOME\ k.\ k \notin (\bigcup p \in extend\ S\ C\ f\ n \cup \{f\ n\}.\ params\ p))\ [])$
$(dest\text{-}Uni\ (dest\text{-}Neg\ (f\ n))))\}$
$else\ extend\ S\ C\ f\ n \cup \{f\ n\})$
$else\ extend\ S\ C\ f\ n)$›

The f above will be specialized to *from-nat* in the end. The *dest* functions are defined by Berghofer as inexhaustive primitive recursions that return *undefined* on any other input than the intended. I use the following type of abbreviation instead which covers all cases and expresses the pattern matching explicitly with *case* instead of relying on *primrec* that suggests recursion where none is needed:

abbreviation *dest-Uni* :: ‹*fm* ⇒ *fm*› **where**
‹*dest-Uni p* ≡ (*case p of* (*Uni p′*) ⇒ *p′* | *p′* ⇒ *p′*)›

Elements obtained by *extend* form a chain by construction:

theorem *is-chain-extend*: ‹*is-chain* (*extend S C f*)›
by (*simp add*: *is-chain-def*) *blast*

Given this we can easily obtain the union of the entire chain:

definition *Extend* :: ‹*fm set* ⇒ *fm set set* ⇒ (*nat* ⇒ *fm*) ⇒ *fm set*›
where
‹*Extend S C f* = (⋃*n*. *extend S C f n*)›

We need to prove that each element in the sequence leaves infinitely many parameters unused so the sequence can be extended infinitely without causing problems in the δ cases. This is done in the following lemma where we show that for any chosen n there exists a parameter unused by S_n.

lemma *infinite-params-available*:
assumes ‹*infinite* (− (⋃*p* ∈ *S*. *params p*))›
shows ‹∃*x*. *x* ∉ (⋃*p* ∈ *extend S C f n* ∪ {*f n*}. *params p*)›
(**is** ‹- (⋃- ∈ *?S′*. -)›)
proof −
have ‹*infinite* (− (⋃*p* ∈ *?S′*. *params p*))›
using *assms* **by** (*simp add*: *set-inter-compl-diff*)
then obtain *x* **where** ‹*x* ∈ − (⋃*p* ∈ *?S′*. *params p*)›
using *infinite-imp-nonempty* **by** *blast*
then show ‹∃*x*. *x* ∉ (⋃*p* ∈ *?S′*. *params p*)›
by *blast*
qed

This essentially follows from the fact that there are infinitely many unused parameters but each S_n can only use finitely many of them, always leaving some unused.

Looking at the cases in *extend* we need to prove that no matter which is chosen, the resulting element is in C^*. The interesting case is for δ formulas, where the *Exi* case is given below, starting with the declaration of the lemma.

lemma *extend-in-C-Exi*:
  assumes ‹*alt-consistency C*›
    and ‹*infinite* $(- (\bigcup p \in S.\ params\ p))$›
    and ‹*extend S C f n* $\cup$ {*f n*} $\in C$› (**is** ‹*?S′* $\in C$›)
    and ‹$\exists p.\ f\ n = Exi\ p$›
  shows ‹*extend S C f* (*Suc n*) $\in C$›

The assumptions are simply the facts we know by the definition of *extend* when we are in the *Exi* case. The direct proof follows.

proof –
  obtain *p* **where** $*$: ‹$f\ n = Exi\ p$›
    using ‹$\exists p.\ f\ n = Exi\ p$› **by** *blast*

  let *?x* = ‹(*SOME k. k* $\notin (\bigcup p \in ?S'.\ params\ p)$)›

  from ‹*infinite* $(- (\bigcup p \in S.\ params\ p))$›
  have ‹$\exists x.\ x \notin (\bigcup p \in ?S'.\ params\ p)$›
    using *infinite-params-available* **by** *blast*
  then have ‹$?x \notin (\bigcup p \in ?S'.\ params\ p)$›
    using *someI-ex* **by** *metis*
  then have ‹($?S' \cup$ {*sub 0* (*Fun ?x* []) *p*}) $\in C$›
    using $*$ ‹$?S' \in C$› ‹*alt-consistency C*›
    unfolding *alt-consistency-def* **by** *simp*
  then show *?thesis*
    using *assms* $*$ **by** *simp*
qed

We start by obtaining the *p* from $X_n = f\ n = Exi\ p$ and define a parameter *?x* which is free in *p*. The definition of *?x* uses Hilbert's choice operator which selects some element, if one exists, that satisfies the given property. Next we show *?x* actually exists using *infinite-params-available* and thus that it really is free in *p*. From there the proof follows directly from the consistency of *C*.

Given proofs of all the cases in *extend* the proof of the following theorem is trivial by induction on *n* and omitted:

theorem *extend-in-C*: ‹*alt-consistency C* $\Longrightarrow S \in C \Longrightarrow$
    infinite $(- (\bigcup p \in S.\ params\ p)) \Longrightarrow$ *extend S C f n* $\in C$›

This extends to the main theorem, that the union of the chain of all S_n is in C^*.

theorem *Extend-in-C*: ⟨*alt-consistency* $C \Longrightarrow$ *finite-char* $C \Longrightarrow$
$S \in C \Longrightarrow$ *infinite* $(- (\bigcup p \in S.\ params\ p)) \Longrightarrow$ *Extend* $S\ C\ f \in C$⟩
using *chain-union-closed is-chain-extend extend-in-C*
unfolding *Extend-def* **by** *blast*

8.3.3 Maximality

Finally we just need to show that the obtained set is maximal. For this we first need to define maximality:

definition *maximal* :: ⟨$'a$ *set* $\Rightarrow$ $'a$ *set set* $\Rightarrow$ *bool*⟩ **where**
⟨*maximal* $S\ C = (\forall S' \in C.\ S \subseteq S' \longrightarrow S = S')$⟩

I will omit the formalized proof of maximality. It follows the one given in section 7.4.2. The conclusion is given below.

theorem *Extend-maximal*:
assumes ⟨$\forall y :: fm.\ \exists n.\ y = f\ n$⟩ **and** ⟨*finite-char* C⟩
shows ⟨*maximal* (*Extend* $S\ C\ f$) C⟩

8.4 Hintikka Sets

Hintikka sets, like consistency properties, are formalized by a boolean function on sets:

definition *hintikka* :: ⟨*fm set* $\Rightarrow$ *bool*⟩ **where**
⟨*hintikka* H =
$((\forall p\ ts.\ \neg\ (Pre\ p\ ts \in H \land Neg\ (Pre\ p\ ts) \in H)) \land$
$Falsity \notin H \land$
$(\forall Z.\ Neg\ (Neg\ Z) \in H \longrightarrow Z \in H) \land$
$(\forall A\ B.\ Con\ A\ B \in H \longrightarrow A \in H \land B \in H) \land$
$(\forall A\ B.\ Neg\ (Dis\ A\ B) \in H \longrightarrow Neg\ A \in H \land Neg\ B \in H) \land$
$(\forall A\ B.\ Dis\ A\ B \in H \longrightarrow A \in H \lor B \in H) \land$
$(\forall A\ B.\ Neg\ (Con\ A\ B) \in H \longrightarrow Neg\ A \in H \lor Neg\ B \in H) \land$
$(\forall A\ B.\ Imp\ A\ B \in H \longrightarrow Neg\ A \in H \lor B \in H) \land$

$(\forall A\ B.\ Neg\ (Imp\ A\ B) \in H \longrightarrow A \in H \wedge Neg\ B \in H) \wedge$
$(\forall P\ t.\ closed\text{-}term\ 0\ t \longrightarrow Uni\ P \in H \longrightarrow sub\ 0\ t\ P \in H) \wedge$
$(\forall P\ t.\ closed\text{-}term\ 0\ t \longrightarrow Neg\ (Exi\ P) \in H \longrightarrow$
$\quad Neg\ (sub\ 0\ t\ P) \in H) \wedge$
$(\forall P.\ Exi\ P \in H \longrightarrow (\exists t.\ closed\text{-}term\ 0\ t \wedge sub\ 0\ t\ P \in H)) \wedge$
$(\forall P.\ Neg\ (Uni\ P) \in H \longrightarrow$
$\quad (\exists t.\ closed\text{-}term\ 0\ t \wedge Neg\ (sub\ 0\ t\ P) \in H)))\rangle$

8.4.1 Herbrand Terms

A separate term data type without variables is used for Herbrand terms to ensure by construction that they are closed:

datatype *htm* = *HFun id* ⟨*htm list*⟩

We also need functions for turning Herbrand terms into regular terms.

tm-of-htm :: ⟨*htm* ⇒ *tm*⟩ *and*
tms-of-htms :: ⟨*htm list* ⇒ *tm list*⟩

When a term is closed, its semantics in a Herbrand model is equal to itself.

lemma *herbrand-semantics* [*simp*]:
⟨*closed-term 0 t* ⟹ *tm-of-htm* (*semantics-term e HFun t*) = *t*⟩
⟨*closed-list 0 l* ⟹ *tms-of-htms* (*semantics-list e HFun l*) = *l*⟩
by (*induct t* **and** *l rule*: *closed-term.induct closed-list.induct*) *simp-all*

The way we formalize a Herbrand model is by using *HFun* as $\mathcal{F}$, so every closed term is turned into the equivalent Herbrand term. For $\mathcal{G}$ we will use a lambda function which converts the Herbrand terms in the predicate back to the equivalent regular terms before looking the predicate up in H.

Any term originating from a Herbrand term becomes that Herbrand term again when evaluated in a Herbrand model.

lemma *herbrand-semantics′* [*simp*]:
⟨*semantics-term e HFun* (*tm-of-htm ht*) = *ht*⟩
⟨*semantics-list e HFun* (*tms-of-htms hts*) = *hts*⟩
by (*induct ht* **and** *hts rule*: *tm-of-htm.induct tms-of-htms.induct*) *simp-all*

8.4.2 The Lemma

We will prove that any closed formula from a Hintikka set is true in a Herbrand model. This however, does not provide us with a strong enough induction hypothesis e.g. for implication so we will extend the claim to also say that any negated closed formula in a Hintikka set is true in a Herbrand model.

```
theorem hintikka-model:
  assumes hin: ‹hintikka H›
  shows ‹(p ∈ H ⟶ closed 0 p ⟶
    semantics e HFun (λi l. Pre i (tms-of-htms l) ∈ H) p) ∧
     (Neg p ∈ H ⟶ closed 0 p ⟶
    semantics e HFun (λi l. Pre i (tms-of-htms l) ∈ H) (Neg p))›
```

The proof given in section 7.5.3 is by structural induction on the formula. In the case of quantifiers, e.g. $\forall x.A$, we use the induction hypotheses on instances of the quantified formula, e.g. $A[c/0]$. Isabelle's standard induction on data types only allows the induction hypothesis to be applied to direct constructor arguments, i.e A. Therefore we will instead use well-founded induction over the size of the data type, defined as the number of logical connectives and quantifiers used in it. Since $A[c/0]$ has the same size as A this solves our problem:

```
proof (rule wf-induct[where a=p and r=‹measure size-formulas›])
  show ‹wf (measure size-formulas)›
    by blast
next
  let ?semantics = ‹semantics e HFun (λi l. Pre i (tms-of-htms l) ∈ H)›

  fix x
  assume wf: ‹∀ y. (y, x) ∈ measure size-formulas ⟶
      (y ∈ H ⟶ closed 0 y ⟶ ?semantics y) ∧
  (Neg y ∈ H ⟶ closed 0 y ⟶ ?semantics (Neg y))›

  show ‹(x ∈ H ⟶ closed 0 x ⟶ ?semantics x) ∧
    (Neg x ∈ H ⟶ closed 0 x ⟶ ?semantics (Neg x))›
  proof (cases x)
```

It makes the start of the proof a bit more verbose than when using *induct* directly, but it is a small price to pay for the ability to choose our own induction measure.

The proof was given in the last chapter so I will just show a single case here. For variety this will be the positive *Imp* case. x below is equal to *Imp A B*, another consequence of using our own induction measure.

```
case (Imp A B)
then show ?thesis proof (intro conjI impI)
  assume ‹x ∈ H› and ‹closed 0 x›
  then have ‹Imp A B ∈ H› and ‹closed 0 (Imp A B)›
    using Imp by simp-all
  then have ‹Neg A ∈ H ∨ B ∈ H›
    using hin unfolding hintikka-def by blast
  then show ‹?semantics x›
    using Imp wf ‹closed 0 (Imp A B)› by force
```

The implication is in H by assumption, so either component *Neg* A or B is also in H by the properties of a Hintikka set. Thus if *Neg* $A \in H$, it evaluates to true by the induction hypothesis, making A false and *Imp* A B true. If on the other hand it is $B \in H$ which holds, then B is true in the Herbrand model and so is *Imp* A B by the semantics no matter the truth value of A.

8.4.3 Maximal Extension is Hintikka

Before we can derive the model existence theorem we need to show that the set produced by *Extend* is in fact a Hintikka set.

```
theorem extend-hintikka:
  assumes ‹S ∈ C›
    and fin-ch: ‹finite-char C›
    and infin-p: ‹infinite (− (⋃p ∈ S. params p))›
    and surj: ‹∀ y. ∃ n. y = f n›
    and altc: ‹alt-consistency C›
  shows ‹hintikka (Extend S C f)› (is ‹hintikka ?H›)
```

The proof is set up as we have seen previously, unfolding the definition of Hintikka sets and introducing each conjunct as a goal to prove. Before proving a case some facts are derived which are useful for several of the cases.

```
  unfolding hintikka-def
proof (intro allI impI conjI)
  have ‹maximal ?H C› and ‹?H ∈ C›
    using Extend-maximal Extend-in-C assms by blast+
```

Everything except the δ cases are similar and trivial. Below is an example of a γ case, namely *Neg (Exi P)*.

{ fix *P t*
assume ‹*Neg* (*Exi P*) $\in$ *?H*› **and** ‹*closed-term 0 t*›
then have ‹*?H* $\cup$ {*Neg* (*sub 0 t P*)} $\in$ *C*›
using ‹*?H* $\in$ *C*› *altc* **unfolding** *alt-consistency-def* **by** *blast*
then show ‹*Neg* (*sub 0 t P*) $\in$ *?H*›
using ‹*maximal ?H C*› **unfolding** *maximal-def* **by** *fast* **}**

We can extend $?H$ with the instantiation because it is a member of the consistency property. Since $?H$ is maximal it must already include the instantiation, which is the condition for it being Hintikka in this case and what we want to show.

The *Exi* case, one of the two δ cases, relies on the omitted *Exi-in-Extend* lemma. It states that the *extend* function adds an instantiation of the δ formula in that case. Given this we can prove that *Extend* also produces a Hintikka set in the *Exi* case. First by obtaining the n which for which X_n is an *Exi* formula and setting up the (closed) term matching the one from the definition of *extend*:

{ fix *P*
assume ‹*Exi P* $\in$ *?H*›
obtain *n* **where** $*$: ‹*Exi P* = *f n*›
using *surj* **by** *blast*

let *?t* = ‹*Fun* (*SOME k*.
k $\notin$ ($\bigcup$ *p* $\in$ *extend S C f n* $\cup$ {*f n*}. *params p*)) []›

have ‹*closed-term 0 ?t*›
by *simp*

Then by noting that the extension of the chain by X_n is in C so using the lemma mentioned above, the instantiation must be in *?H*:

moreover have ‹*extend S C f n* $\cup$ {*f n*} $\subseteq$ *?H*›
using ‹*Exi P* $\in$ *?H*› $*$ *Extend-def* **by** (*simp add*: *UN-upper*)
then have ‹*extend S C f n* $\cup$ {*f n*} $\in$ *C*›
using ‹*?H* $\in$ *C*› *fin-ch finite-char-closed subset-closed-def* **by** *metis*
then have ‹*sub 0 ?t P* $\in$ *?H*›
using $*$ *Exi-in-extend Extend-def* **by** *fast*
ultimately show ‹$\exists$ *t*. *closed-term 0 t* $\wedge$ *sub 0 t P* $\in$ *?H*›
by *blast* **}**

8.5 Model Existence Theorem

To obtain the model existence theorem we first need a lemma that connects all of the above pieces together, going from a set of formulas S in a consistency property C to a superset of S that is Hintikka. The following lemma does this:

lemma *hintikka-Extend-S*:
  assumes ‹*consistency C*› **and** ‹$S \in C$›
    and ‹*infinite* ($-$ ($\bigcup p \in S.$ *params p*))›
  defines ‹$C' \equiv$ *mk-finite-char* (*mk-alt-consistency* (*close C*))›
  shows ‹*hintikka* (*Extend S C′ from-nat*)›

The proof is trivial but lengthy and omitted for brevity. Now the model existence theorem follows directly.

theorem *model-existence*:
  assumes ‹*infinite* ($-$ ($\bigcup p \in S.$ *params p*))›
    and ‹$p \in S$› ‹*closed 0 p*›
    and ‹$S \in C$› ‹*consistency C*›
  defines ‹$C' \equiv$ *mk-finite-char* (*mk-alt-consistency* (*close C*))›
  defines ‹$H \equiv$ *Extend S C′ from-nat*›
  shows ‹*semantics e HFun* (λa *ts. Pre a* (*tms-of-htms ts*) $\in H$) *p*›
  using *assms hintikka-model hintikka-Extend-S Extend-subset* **by** *blast*

This proof of the model existence theorem in Isabelle differs from Berghofer's [Ber07a]. Berghofer proves the *hintikka-Extend-S* lemma directly in the proof of the model existence theorem where I have chosen to split it out. I have chosen to do this because the fact that the maximal extension of S is Hintikka is an important result in itself. Furthermore it makes the proof of the model existence theorem trivial as apparent above. The same decomposition was applied to the *extend-in-C* theorem for the same reasons.

Another difference between Berghofer's formalization and mine is in the formulation of the above theorem. Berghofer writes the definitions of C' and H directly inside the goal where I have given them names with the *defines* construct. This naming is intended to remind the reader what expression constitutes the (alternate) consistency property, the Hintikka set etc. and is something I have tried to introduce all the way through the formalization. It allows reasoning at a higher level when reading the proof, because often the important thing in a proof step is just that e.g. C is a consistency property and now how it was formed.

8.6 Inference Rule Consistency

To apply the model existence theorem we need to prove that the set of sets from which we cannot derive falsehood is a consistency property.

theorem *OK-consistency*:
⟨*consistency* {*set G* |*G*. ¬ (*OK Falsity G*)}⟩

The proof proceeds by unfolding the definition of a consistency property and introducing each condition as a new goal. Because the inference rules use a list G but the consistency property uses sets, we need to consider the set S instead of G directly.

The difficulty of these cases varies significantly. The predicate case below is one of the simpler ones:

```
{ fix i l
  assume ⟨Pre i l ∈ S ∧ Neg (Pre i l) ∈ S⟩
  then have ⟨OK (Pre i l) G⟩ and ⟨OK (Neg (Pre i l)) G⟩
    using Assume ∗ by auto
  then have ⟨OK Falsity G⟩
    using Imp-E by blast
  then show False
    using ⟨¬ (OK Falsity G)⟩ by blast }
```

Instead of proving that the positive and negative predicates are not both in S, we assume that they are and derive falsehood. This allows us to derive both of them using the *Assume* rule and thus show *Falsity* by remembering that the formula *Neg (Pre i l)* is short for *Imp (Pre i l) Falsity* and we have just derived *Pre i l*.

A slightly more interesting but still manageable case is for conjunctions. Given a proof of a conjunction we can obtain proofs of both components:

```
{ fix A B
  assume ⟨Con A B ∈ S⟩
  then have ⟨OK (Con A B) G⟩
    using Assume ∗ by simp
  then have ⟨OK A G⟩ and ⟨OK B G⟩
    using Con-E1 Con-E2 by blast+
```

Now a contradiction is caused when assuming that we can derive *Falsity* by assuming these components, so adding them to the list of assumptions must still be consistent, which concludes the proof:

```
{ assume ‹OK Falsity (A # B # G)›
  then have ‹OK (Neg A) (B # G)›
    using Imp-I by blast
  then have ‹OK (Neg A) G›
    using cut ‹OK B G› by blast
  then have ‹OK Falsity G›
    using Imp-E ‹OK A G› by blast
  then have False
    using ‹¬ (OK Falsity G)› by blast }
then have ‹¬ (OK Falsity (A # B # G))›
  by blast
moreover have ‹S ∪ {A, B} = set (A # B # G)›
  using * by simp
ultimately show ‹S ∪ {A, B} ∈ ?C›
  by blast }
```

This proof uses the following *cut* rule:

```
lemma cut: ‹OK p z ⟹ OK q (p # z) ⟹ OK q z›
  apply (rule Imp-E) apply (rule Imp-I) .
```

This cut rule allows us to get rid of an assumption in a proof if we can derive the assumption on its own and is an entire topic on its own. Worth mentioning is that the rule can be derived internally in natural deduction and thus need not be added explicitly.

8.7 Completeness using Herbrand Terms

We can now prove completeness of the proof system in the domain of Herbrand terms and we will see later why this is sufficient. We want to prove the following:

```
theorem natded-complete:
  assumes ‹closed 0 p› and ‹list-all (closed 0) z›
    and mod: ‹∀ (e :: nat ⇒ htm) f g.
      list-all (semantics e f g) z ⟶ semantics e f g p›
  shows ‹OK p z›
```

We assume that p follows from the assumptions z in all interpretations and call this fact *mod*. The proof is then by contradiction instead of contraposition as in section 7.7, but this is just a technical difference. A few abbreviations are set up for pedagogical reasons and easier reference in the proof:

proof (*rule Boole*, *rule ccontr*)
  fix *e*
  assume ‹¬ (*OK Falsity* (*Neg p* # *z*))›

  let *?S* = ‹*set* (*Neg p* # *z*)›
  let *?C* = ‹{*set G* | *G*. ¬ (*OK Falsity G*)}›
  let *?C′* = ‹*mk-finite-char* (*mk-alt-consistency* (*close ?C*))›
  let *?H* = ‹*Extend ?S ?C′ from-nat*›
  let *?f* = *HFun*
  let *?g* = ‹λ*i l*. *Pre i* (*tms-of-htms l*) ∈ *?H*›

We start by showing that *Neg p* as well as every element in z is true in the Herbrand model using the model existence theorem.

  { fix *x*
    assume ‹*x* ∈ *?S*›
    moreover have ‹*closed 0 x*›
      using ‹*closed 0 p*› ‹*list-all* (*closed 0*) *z*› ‹*x* ∈ *?S*›
      by (*auto simp*: *list-all-iff*)
    moreover have ‹*?S* ∈ *?C*›
      using ‹¬ (*OK Falsity* (*Neg p* # *z*))› **by** *blast*
    moreover have ‹*consistency ?C*›
      using *OK-consistency* **by** *blast*
    moreover have ‹*infinite* (− (⋃*p* ∈ *?S*. *params p*))›
      by (*simp add*: *Compl-eq-Diff-UNIV infinite-UNIV-listI*)
    ultimately have ‹*semantics e ?f ?g x*›
      using *model-existence* **by** *simp* **}**
  then have ‹*semantics e ?f ?g* (*Neg p*)›
    and ‹*list-all* (*semantics e ?f ?g*) *z*›
    unfolding *list-all-def* **by** *fastforce+*

Knowing that every element in z is true allows us to get a model for p with *mod*. But this contradics with the model obtained for *Neg p*, proving the theorem:

  then have ‹*semantics e ?f ?g p*›
    using *mod* **by** *blast*
  then show *False*
    using ‹*semantics e ?f ?g* (*Neg p*)› **by** *simp*
qed

8.8 Completeness in Countably Infinite Domains

That the proof system is complete in the domain of Herbrand terms is a strong result: Any valid formula must be true in all interpretations with the domain of Herbrand terms and can thus be derived in the system. Here we will, for pedagogical reasons, derive a version of the completeness result that applies to any countably infinite domain such as the natural numbers instead of Herbrand terms specifically. This work goes beyond Berghofer's formalization and is based on work by Anders Schlichtkrull on completeness for unordered resolution [Sch17]. Resolution is a vastly different proof system from natural deduction but the same overall strategy can be applied here. This strategy is to prove that there is a bijection between the Herbrand terms and any countably infinite domain and that the semantics respect this bijection. As the former is independent of the proof system, I will focus on the latter here.

8.8.1 Bijective Semantics

The proof relies on three functions for converting environments and interpretations to operate on different types:

definition *e-conv* :: ‹$('a \Rightarrow 'b) \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b)$› **where**
‹*e-conv b-of-a e* $\equiv$ ($\lambda n.$ *b-of-a* (*e n*))›

definition *f-conv* ::
‹$('a \Rightarrow 'b) \Rightarrow (id \Rightarrow 'a\ list \Rightarrow 'a) \Rightarrow (id \Rightarrow 'b\ list \Rightarrow 'b)$› **where**
‹*f-conv b-of-a f* $\equiv$ ($\lambda a\ ts.$ *b-of-a* (*f a* (*map* (*inv b-of-a*) *ts*)))›

definition *g-conv* ::
‹$('a \Rightarrow 'b) \Rightarrow (id \Rightarrow 'a\ list \Rightarrow bool) \Rightarrow (id \Rightarrow 'b\ list \Rightarrow bool)$› **where**
‹*g-conv b-of-a g* $\equiv$ ($\lambda a\ ts.$ *g a* (*map* (*inv b-of-a*) *ts*))›

Before tackling the semantics we need a lemma relating the *put* and *e-conv* functions. This states that putting a converted element into a converted environment is the same as converting the result of putting an the original element into the original environment:

lemma *put-e-conv*:
‹(*put* (*e-conv b-of-a e*) *m* (*b-of-a x*)) = *e-conv b-of-a* (*put e m x*)›
unfolding *e-conv-def* **by** *auto*

Next we need to show that when the conversion is bijective, the semantics of terms and formulas is bijective with respect to the conversion. Bijection essentially means that each element of the first type maps to exactly one element of the second type and vice versa. This is shown for terms first:

```
lemma semantics-bij':
  assumes ‹bij (b-of-a :: 'a ⇒ 'b)›
  shows
    ‹semantics-term (e-conv b-of-a e) (f-conv b-of-a f) p =
     b-of-a (semantics-term e f p)›
    ‹semantics-list (e-conv b-of-a e) (f-conv b-of-a f) l =
     map b-of-a (semantics-list e f l)›
  unfolding e-conv-def f-conv-def using assms
  by (induct p and l rule: semantics-term.induct semantics-list.induct)
(simp-all add: bij-is-inj)
```

The proof is by induction on the recursive calls to *semantics-term* and *semantics-list*. All the cases are proven automatically with *simp-all* after unfolding the definitions and reminding Isabelle that bijective functions are also injective.

Finally we can show that the semantics in the original environment and interpretation is exactly the same as the semantics of the same environment and interpretation converted by a bijective function. This proof is by induction on the type of formula and all cases except predicates and quantifiers are solved automatically. The cases for the quantifiers are symmetrical so I will only show the existential one here.

```
lemma semantics-bij:
  assumes ‹bij (b-of-a :: 'a ⇒ 'b)›
  shows ‹semantics e f g p =
    semantics (e-conv b-of-a e) (f-conv b-of-a f) (g-conv b-of-a g) p›
proof (induct p arbitrary: e f g)
```

The *Pre* case is shown using the *semantics-term-bij* lemma:

```
  case (Pre a l)
  then show ?case
    unfolding g-conv-def using assms
    by (simp add: semantics-bij' bij-is-inj)
```

In the *Exi* case we first use the bijectivity of *b-of-a* to show that the existence of x' which satisfies the formula is the same as the existence of an x which converts to that x' and thus satisfies the formula:

case (*Exi p*)

let *?e* = ‹*e-conv b-of-a e*›
  and *?f* = ‹*f-conv b-of-a f*›
  and *?g* = ‹*g-conv b-of-a g*›

have ‹($\exists\, x'$. *semantics* (*put ?e 0 x'*) *?f ?g p*) =
  ($\exists\, x$. *semantics* (*put ?e 0* (*b-of-a x*)) *?f ?g p*)›
  using *assms* **by** (*metis bij-pointE*)

Then we do a bit of rewriting using the *put-e-conv* lemma to make the induction hypothesis apply. The environment and interpretation is different but that is why we are doing the induction with these as arbitrary. This conclude the proof:

also have ‹... = ($\exists\, x$. *semantics* (*e-conv b-of-a* (*put e 0 x*)) *?f ?g p*)›
  using *put-e-conv* **by** *metis*
finally show *?case*
  using *Exi* **by** *simp*

8.8.2 Completeness

We introduce an abbreviation for sentences, that is, closed formulas:

abbreviation ‹*sentence* $\equiv$ *closed 0*›

Then we can prove completeness of sentences under arbitrary assumptions z and in any countably infinite domain:

lemma *completeness'*:
  assumes ‹*infinite* (*UNIV* :: (*'a* :: *countable*) *set*)›
  and ‹*sentence p*›
  and ‹*list-all sentence z*›
  and ‹$\forall$ (*e* :: *nat* $\Rightarrow$ *'a*) *f g*.
   list-all (*semantics e f g*) *z* $\longrightarrow$ *semantics e f g p*›
  shows ‹*OK p z*›

We are assuming that the formula is true in all interpretations with countably infinite domain *'a* and will use this to show that then this is also the case in the domain of Herbrand terms. Given this we can apply the original completeness result to show that the formula can be derived. The proof is direct and starts by obtaining conversion functions to and from Herbrand terms and setting up environments and interpretations using these:

```
proof –
  have ‹∀(e :: nat ⇒ htm) f g.
    list-all (semantics e f g) z ⟶ semantics e f g p›
  proof (intro allI)
    fix e :: ‹nat ⇒ htm›
      and f :: ‹id ⇒ htm list ⇒ htm›
      and g :: ‹id ⇒ htm list ⇒ bool›

    obtain a-of-htm :: ‹htm ⇒ 'a› where p-a-of-hterm: ‹bij a-of-htm›
      using assms countably-inf-bij infinite-htms by blast

    let ?e = ‹e-conv a-of-htm e›
    let ?f = ‹f-conv a-of-htm f›
    let ?g = ‹g-conv a-of-htm g›
```

Knowing the formula holds under conversions from Herbrand terms to *'a* we can use the bijectivity of the semantics to use Herbrand terms themselves. Using this we can apply the original completeness result and we are done:

```
    have ‹list-all (semantics ?e ?f ?g) z ⟶ semantics ?e ?f ?g p›
      using assms by blast
    then show ‹list-all (semantics e f g) z ⟶ semantics e f g p›
      using p-a-of-hterm semantics-bij by (metis list.pred-cong)
  qed
  then show ?thesis
    using assms natded-complete by blast
qed
```

The result is summed up in the theorem and corollary below.

```
theorem completeness: ‹infinite (UNIV :: ('a :: countable) set) ⟹
    sentence p ⟹ ∀(e :: nat ⇒ 'a) f g. semantics e f g p ⟹ OK p []›
  by (simp add: completeness')

corollary
  ‹sentence p ⟹ ∀(e :: nat ⇒ nat) f g. semantics e f g p ⟹ OK p []›
  using completeness by fast
```

In the original *natded-complete* theorem the domain is fixed to Herbrand terms. These include a constant and function symbol so that domain is infinite. Since this alternate completeness proof relies on a bijection between the countable domain and the domain of Herbrand terms, we need to assume that the countable domain is infinite as well. Finite model theory deals with the case of finite domains and is beyond the scope of this thesis, as is uncountable domains.

8.9 The Löwenheim-Skolem Theorem

The formalization also includes a proof of the Löwenheim-Skolem theorem which could not be ported directly from Berghofer's formalization.

The following quote by Avigad explains the theorem well [Avi06]:

> In modern terms, a first-order "fleeing equation"' is a first-order sentence that is true in every finite model, but not true in every model. Löwenheim's theorem asserts that such a sentence can be falsified in a model whose elements are drawn from a countably infinite domain. Since a sentence is true in a model if and only if its negation is false, we can restate Löwenheim's theorem in its modern form: if a sentence has a model, it has a countable model (that is, one whose domain is finite or countably infinite).

It is the theorem in this modern form that we will prove here. Assuming a set of sentences is satisfiable, we will construct a countable model for the sentences.

8.9.1 Satisfiable Sets are a Consistency Property

The Löwenheim-Skolem theorem is proven by showing that the set of satisfiable sets is a consistency property and then applying the model existence theorem.

We start by proving the former:

theorem *sat-consistency*:
$\langle consistency\ \{S.\ infinite\ (-\ (\bigcup p \in S.\ params\ p)) \land$
$(\exists f.\ \forall p \in S.\ semantics\ e\ f\ g\ p)\}\rangle$
(**is** $\langle consistency\ ?C\rangle$)

The sets, $?S$, we consider leave infinitely many parameters unused and are satisfiable under some function interpretation f. In the proof we look at an arbitrary set S and show that it satisfies the conditions for $?C$ being *consistent*. We obtain the f that satisfies the formulas in S before showing any cases:

```
  unfolding consistency-def
proof (intro allI impI conjI)
  fix S :: ‹fm set›
  assume ‹S ∈ ?C›
  then have inf-params: ‹infinite (− (⋃p ∈ S. params p))›
    and ‹∃f. ∀p ∈ S. semantics e f g p›
    by blast+
  then obtain f where *: ‹∀x ∈ S. semantics e f g x› by blast
```

All the cases except the ones for δ formulas are trivial. One of these, the *Exi* one, is shown below.

```
  { fix P
    assume ‹Exi P ∈ S›
    then obtain y where ‹semantics (put e 0 y) f g P›
      using * by fastforce
    moreover obtain x where **: ‹x ∈ − (⋃p ∈ S. params p)›
      using inf-params infinite-imp-nonempty by blast
    then have ‹x ∉ params P›
      using ‹Exi P ∈ S› by auto
    moreover have ‹∀p ∈ S. semantics e (f(x := λ-. y)) g p›
      using * ** by simp
```

We have obtained y that satisfies the quantified formula and a free parameter x. Mapping x to y preserves the semantics of the formulas in S since x is free in S.

```
    ultimately have ‹∀p ∈ S ∪ {sub 0 (Fun x []) P}.
        semantics e (f(x := λ-. y)) g p›
      by simp
    moreover have
      ‹infinite (− (⋃p ∈ S ∪ {sub 0 (Fun x []) P}. params p))›
      using inf-params by (simp add: set-inter-compl-diff)
    ultimately show ‹∃x. S ∪ {sub 0 (Fun x []) P} ∈ ?C›
      by blast }
```

The entire set $S \cup P[x/0]$ is therefore satisfied under $f[x \leftarrow y]$. Moreover the number of parameters unused by S is still infinite after extending the S with $P[x/0]$. Ultimately the parameter x exists and makes $S \cup P[x/0]$ part of $?C$.

8.9.2 Unused Parameters

To apply the model existence theorem the formula we want a model for needs to leave infinitely many parameters unused. Berghofer ensures this by letting the function symbols be the natural numbers and doubles every identifier in S ensuring every odd number is unused. In NaDeA the identifiers are fixed to be strings so we need to employ a different but similar trick. For this we will use the two following functions:

primrec *double* :: ⟨*'a list* ⇒ *'a list*⟩ **where**
⟨*double* [] = []⟩ |
⟨*double* (*x*#*xs*) = *x* # *x* # *double xs*⟩

fun *undouble* :: ⟨*'a list* ⇒ *'a list*⟩ **where**
⟨*undouble* [] = []⟩ |
⟨*undouble* [*x*] = [*x*]⟩ |
⟨*undouble* (*x*#-#*xs*) = *x* # *undouble xs*⟩

The first function duplicates every character in the function symbol while the latter contracts it again. This duplication leaves every identifier of odd length unused analogously to Berghofer's doubling of numbers.

That *undouble* cancels *double* is shown by induction over the argument list:

lemma *undouble-double-id* [*simp*]: ⟨*undouble* (*double xs*) = *xs*⟩
by (*induct xs*) *simp-all*

Showing that applying *double* as a parameter substitution has the wanted effect is trickier. For this we will need two lemmas. The first says that the set of doubled lists with an added element in front is infinite:

lemma *infinite-double-Cons*: ⟨*infinite* (*range* (*λxs. a* # *double xs*))⟩
using *undouble-double-id infinite-UNIV-listI*
by (*metis* (*mono-tags*, *lifting*) *finite-imageD inj-onI list.inject*)

The seconds says that a doubled list with an added element in front can never be equal to a doubled list. This is proved by observing that the parity of their lengths must differ:

```
lemma double-Cons-neq: ‹a # (double xs) ≠ double ys›
proof −
  have ‹odd (length (a # double xs))›
    by (induct xs) simp-all
  moreover have ‹even (length (double ys))›
    by (induct ys) simp-all
  ultimately show ?thesis
    by metis
qed
```

Finally we get to the main result which is proven by the rule *infinite-super*. This says that if an infinite set is a subset of another set, then the superset must also be infinite. Thus we use the previous result that the set, T, of doubled lists with an added element in front, is infinite. Moreover we show that given a set, S_{par}, of doubled parameters, the inverse of this is a subset of T. The latter follows from the fact that S_{par} and T cannot have any elements in common:

```
lemma doublep-infinite-params:
  ‹infinite (− (⋃p ∈ psubst double ‘ S. params p))›
proof (rule infinite-super)
  fix a
  show ‹infinite (range (λxs :: id. a # double xs))›
    using infinite-double-Cons by metis
next
  fix a
  show ‹range (λxs. a # double xs) ⊆
      − (⋃p ∈ psubst double ‘ S. params p)›
    using double-Cons-neq by fastforce
qed
```

8.9.3 The Theorem

We now have the machinery necessary to prove the Löwenheim-Skolem theorem. It is formulated as follows:

```
theorem loewenheim-skolem:
  assumes ‹∀p ∈ S. semantics e f g p› ‹∀p ∈ S. closed 0 p›
  defines ‹C ≡ {S. infinite (− (⋃p ∈ S. params p)) ∧
                    (∃f. ∀p ∈ S. semantics e f g p)}›
  defines ‹C′ ≡ mk-finite-char (mk-alt-consistency (close C))›
  defines ‹H ≡ Extend (psubst double ‘ S) C′ from-nat›
  shows ‹∀p ∈ S. semantics e′ (λxs. HFun (double xs))
          (λi l. Pre i (tms-of-htms l) ∈ H) p›
```

We assume that all the sentences in S are satisfiable and show that they are satisfiable in a Herbrand model wherein we have doubled the identifier names. The proof continues by looking at an arbitrary formula $p \in S$ which we will show is satisfiable. First we use the result above to show that doubling the identifiers in S makes it part of the consistency property C:

```
proof (intro ballI impI)
  fix p
  assume ‹p ∈ S›

  let ?g = ‹λi l. Pre i (tms-of-htms l) ∈ H›

  have ‹∀ p ∈ psubst double ‘ S. semantics e (λxs. f (undouble xs)) g p›
    using ‹∀ p ∈ S. semantics e f g p› by (simp add: psubst-semantics)
  then have ‹psubst double ‘ S ∈ C›
    using C-def doublep-infinite-params by blast
```

The rest of the proof is then simply a matter of applying the model existence theorem after proving the necessary prerequisites.

```
  moreover have ‹psubst double p ∈ psubst double ‘ S›
    using ‹p ∈ S› by blast
  moreover have ‹closed 0 (psubst double p)›
    using ‹∀ p ∈ S. closed 0 p› ‹p ∈ S› by simp
  moreover have ‹consistency C›
    using C-def sat-consistency by blast
  ultimately have ‹semantics e' HFun ?g (psubst double p)›
    using C-def C'-def H-def model-existence by simp
  then show ‹semantics e' (λxs. HFun (double xs)) ?g p›
    using psubst-semantics by blast
qed
```

C is consistent, doubling p makes it part of the doubled S which is a member of C, and sets in C leave infinitely many parameters unused by construction. Thus we can obtain a model for doubled p and this is equivalent to a model which doubles p during evaluation. This concludes the proof.

Let us recap what we know at this point. We just proved that if a sentence is satisfiable then it is so in a model with the domain of Herbrand terms. Thus if it is valid then it is true in all models that are Herbrand. In the introduction we proved using the soundness and completeness of the proof system that if a sentence is true in all interpretations with the natural numbers as domain, then it is valid in general — and we might as well have used the domain of Herbrand terms instead of the natural numbers. The other direction follows trivially as explained previously.

Combining these facts we arrive at the Herbrand model theorem: A sentence is valid if and only if it is true in all interpretations that are Herbrand [Fit96, theorem 9.5.4].

Chapter 9

On Open Formulas

Open formulas are strange beasts. The meaning of a variable in a formula is determined by the quantifier binding it: does it represent every element of the domain or just some specific element? So what should we think of a formula like

$$p(x) \to p(y)$$

where x and y are unbound variables, not constants. It is unclear. One solution is to universally close the formula, binding every free variable with a universal quantifier like this:

$$\forall x.\forall y.p(x) \to p(y)$$

We call this the *universal closure* of p and in this interpretation the formula is not valid. On the other hand the following formula is clearly valid no matter how we interpret the free variable, as the right disjunct is always true:

$$p(x) \vee (\bot \to \bot)$$

And we can certainly derive this formula using natural deduction:

$$\dfrac{\dfrac{\bot \vdash \bot}{\vdash \bot \to \bot} \to I}{\vdash p(x) \lor (\bot \to \bot)} \lor I_2$$

We can also derive it in the formalization:

lemma *open-example*:
⟨*OK* (*Dis* (*Pre* ''*p*'' [*Var x*]) (*Imp Falsity Falsity*)) []⟩
apply (*rule Dis-I2*)
apply (*rule Imp-I*)
apply (*rule Assume*)
apply *simp*
done

The index of the variable does not even have to be specified. While the semantics of free variables are unclear in the textbook formulation, it is necessarily specified in the formalization. Here variables are looked up in the environment, e, and as this is represented as a function, from the natural numbers into the chosen domain, its value is defined even for free variables. Thus a proof of *semantics* e f g p where the e is implicitly quantified universally, means that the formula p is true no matter what e maps the variables to. As such they are treated as if p has been universally closed.

In the following I will show how to extend the proof system and remove the restriction that formulas must be closed in the completeness proof. First we will handle the case of formulas that are valid given no assumptions and then see how to apply this result to the case of an arbitrary list of assumptions.

9.1 Assuming Nothing

Let us consider first formulas that should be derivable from the empty list of assumptions.

An initial attempt to show completeness for open formulas could be to extend the proof by Fitting that was explained and formalized in the previous chapters. This is not possible, however, as it relies crucially on Hintikka's lemma. In the predicate case of the proof of Hintikka's lemma we need the fact that the formula is closed, because then its constituent terms are closed and the term interprets to itself in the Herbrand model. Open terms cannot interpret to themselves, as Herbrand terms are closed by definition.

Instead we will take a detour to show completeness of an open formula. To use the existing completeness proof we need to close the formula and ensure it is still valid. But then we obtain a proof of the closed formula and need to show that we can derive the original formula from this. This leaves open the questions of how to do each of these steps.

9.1.1 Strategy

There are two obvious ways to close a formula. One, the universal closure, was presented above. The other consists in substituting every free variable with a fresh constant. I have chosen the first approach, universally closing the formula, as it is easier to keep track of how many quantifiers have been added than to remember which variables were mapped to which constants. In practice, proving that the universal closure closes the formula is also simpler since the formalization checks if a formula is closed by counting the number of passed quantifiers.

I initially attempted to derive the original formula directly from its universal closure, but the following example shows why this is tricky to formalize. Consider the following open formula and its universal closure in de Bruijn notation:

$$p(0,1,2) \rightsquigarrow \forall\forall\forall p(0,1,2)$$

Now the only option we have for eliminating universal quantifiers in the proof system is the *Uni-E* rule. This says that we must substitute something for zero, immediately after removing a quantifier. Doing this with the variables in decreasing order gives the correct result in the end. After the first substitution the variable is pointing way too far:

$$(\forall\forall p(0,1,2))[2/0] \rightsquigarrow \forall((\forall p(0,1,2))[3/1]) \rightsquigarrow \forall\forall(p(0,1,2)[4/2]) \rightsquigarrow \forall\forall p(0,1,4)$$

But because we substitute for zero again, it will be decremented:

$$(\forall p(0,1,4))[1/0] \rightsquigarrow \forall(p(0,1,4)[2/1]) \rightsquigarrow \forall p(0,2,3)$$

And finally we are back at the original formula:

$$p(0,2,3))[0/0] \rightsquigarrow p(0,1,2)$$

This works because while the variable we insert is incremented by each quantifier it passes, those quantifiers are later removed and the variable decremented for each of them as well, returning it to its original value. Formalizing that this works turned out to be very tricky as one has to keep track of this relationship between the variables and number of quantifiers. Instead we will derive a version of the formula where every variable bound by the universal closure is replaced with a fresh constant. Then we can use the newly introduced inference rule to turn these constants back into the original variables. And by doing these substitutions in the proper order we will be able to cancel them out pair-wise, instead of having to reason about the entire chain of substitutions at once, as we have to do to show that the above method works.

De Bruijn indices have been very useful in the other parts of the formalization, because, as noted elsewhere [Ber12; Ber07b], very little background theory needs to be developed for their use. In this case however their subtle interaction with substitution, as also noted in chapter 6, makes the above method hard to formalize. If a nominal approach [BU07] could make a proof of the correctness of the above method trivial, it might be worth switching to, even though it requires more theory to be developed. The following solution is less radical though and was chosen instead for this reason.

9.1.2 Substituting Constants

An interpretation may assign any meaning to a constant in a formula. As such, constants act like universally quantified variables and it should be possible to substitute this constant for any other term in a given proof. The inference rule that we are going to add does this, but is actually a little bit more flexible, allowing us to substitute away compound terms also. For instance given a proof of $p(c)$ we can directly obtain a proof of $p(f(x,y))$ and vice versa. This is implemented almost like variable substitution using functions of these types:

$subc\text{-}term :: \langle id \Rightarrow tm \Rightarrow tm \Rightarrow tm\rangle$ *and*
$subc\text{-}list :: \langle id \Rightarrow tm \Rightarrow tm\ list \Rightarrow tm\ list\rangle$
$subc :: \langle id \Rightarrow tm \Rightarrow fm \Rightarrow fm\rangle$
$subcs :: \langle id \Rightarrow tm \Rightarrow fm\ list \Rightarrow fm\ list\rangle$

The following two clauses are worth examining, namely the case for functions and one of the quantifier cases:

$$\textit{subc-term}\ c\ s\ (\textit{Fun}\ i\ l) = (\textit{if}\ i = c\ \textit{then}\ s\ \textit{else}\ \textit{Fun}\ i\ (\textit{subc-list}\ c\ s\ l))$$
$$\textit{subc}\ c\ s\ (\textit{Exi}\ p) = \textit{Exi}\ (\textit{subc}\ c\ (\textit{inc-term}\ s)\ p)$$

For functions we replace the term if the symbol matches and otherwise we recurse on the function's arguments. For quantifiers we increment the term when recursing on the quantified formula as we do when substituting for variables.

Thus we can define the extended proof system consisting of all the *proper* rules, implication elimination and the new rule dubbed *Subtle*:

inductive OK' :: $\langle \textit{fm} \Rightarrow \textit{fm list} \Rightarrow \textit{bool}\rangle$ **where**
Proper: $\langle OK\ p\ z \Longrightarrow OK'\ p\ z\rangle$ |
Imp-E': $\langle OK'\ (\textit{Imp}\ p\ q)\ z \Longrightarrow OK'\ p\ z \Longrightarrow OK'\ q\ z\rangle$ |
Subtle: $\langle OK'\ p\ z \Longrightarrow \textit{new-term}\ c\ s \Longrightarrow OK'\ (\textit{subc}\ c\ s\ p)\ (\textit{subcs}\ c\ s\ z)\rangle$

We make sure to substitute uniformly among the formula and its assumptions and for technical reasons we require c to be free in the term we insert. This will allow us to prove an essential lemma later. It also means that c will be free in both p and z after the substitution.

In all likelihood this rule or a suitable variant of it can be derived from the existing rules. This was attempted by induction over the inference rules but proved difficult in the *Exi-E* case where the interaction between variable and constant substitution is tricky. The case relies on being able to commute the two substitutions, but the order matters as s may not be closed and it cannot be for our purposes. I have chosen instead to prove the above rule sound and work with this extended proof system.

Note that the extension is very conservative; as soon as we move from *OK* to *OK'* the only inferences we are allowed to apply are implication elimination and constant substitution. None of the other original rules are allowed. As such it is very like that this new rule is not actually necessary for completeness of open formulas, but simply makes it easier to prove it.

9.1.3 Soundness

It is important to ensure that we have not traded soundness for completeness. If we were willing to make that bargain we could get away with a much simpler proof system. Luckily we can rely on the existing soundness proof and only have to prove the new rule sound.

The *substitute* lemma proved the correspondence between the syntactic act of substituting a variable and the semantic act of modifying the environment. We need a similar lemma here showing the correspondence between function symbol substitution and modifying the interpretation. The proof follows the proof of *substitute* and is omitted, as is its extension to lists of assumptions *substitutecs*.

lemma *substitutec*:
  ‹*semantics e* (*f*(*c* := λ-. *semantics-term e f s*)) *g p* =
   semantics e f g (*subc c s p*)›

Given this rule we can now state the soundness of the new system. All the hard work has already been done however; the soundness of the old rules are given by *soundness'* and the soundness of the new rule follows directly from the *substitutec* and *substitutecs* lemmas:

lemma *Subtle-soundness'*:
  ‹*OK' p z* ⟹ *list-all* (*semantics e f g*) *z* ⟹ *semantics e f g p*›
proof (*induct p z arbitrary*: *f rule*: *OK'.induct*)
  case (*Proper p z*)
  then show *?case*
    using *soundness'* **by** *blast*
next
  case (*Subtle p z c s*)
  then show *?case*
    using *substitutec substitutecs* **by** *blast*
qed *simp*

Again we can state this result for the case of no assumptions:

theorem *Subtle-soundness*: ‹*OK' p* [] ⟹ *semantics e f g p*›
  by (*simp add*: *Subtle-soundness'*)

9.1.4 Universal Closure

To recapitulate, we are going to universally close the formula to obtain a proof of it, then derive the same formula using fresh constants instead of the newly closed variables and finally use the new inference rule to derive the original formula.

The following primitive recursive function will do the work of closing a formula:

primrec *put-unis* :: ‹*nat* ⇒ *fm* ⇒ *fm*› **where**
‹*put-unis 0 p* = *p*› |
‹*put-unis* (*Suc m*) *p* = *Uni* (*put-unis m p*)›

A few lemmas about this function will be useful later. First, a variable is incremented accordingly when substituting under a number of quantifiers:

lemma *sub-put-unis* [*simp*]:
‹*sub i* (*Fun c* []) (*put-unis k p*) = *put-unis k* (*sub* (*i* + *k*) (*Fun c* []) *p*)›
by (*induct k arbitrary*: *i*) *simp-all*

Second, if a formula with a number of quantifiers is closed at some level, the formula without those quantifiers is closed a correspondingly higher level:

lemma *closed-put-unis*: ‹*closed m* (*put-unis k p*) = *closed* (*m* + *k*) *p*›
by (*induct k arbitrary*: *m*) *simp-all*

We also need to show that every formula can actually be closed by adding a number of quantifiers in front of it. We could calculate this number explicitly but that is more work than necessary. Instead we simply prove its existence:

lemma *ex-closed*: ‹∃ *m*. *closed m p*›

The lemma is trivial to prove by induction over the formula so I will omit the details; surprisingly Isabelle cannot prove it automatically. With this lemma we can know that every formula has a universal closure:

lemma *ex-closure*: ‹∃ *m*. *sentence* (*put-unis m p*)›
using *ex-closed closed-put-unis* **by** *simp*

Finally, if a formula is valid, any free variable has been looked up in all environments with the formula still being true, so we can universally quantify the formula any number of times and it remains valid:

lemma *valid-put-unis*: ‹∀ (*e* :: *nat* ⇒ *'a*) *f g*. *semantics e f g p* ⟹
semantics (*e* :: *nat* ⇒ *'a*) *f g* (*put-unis m p*)›
by (*induct m arbitrary*: *e*) *simp-all*

9.1.4.1 Constants for Quantifiers

We need a function for substituting the quantifiers with a list of constants:

```
fun consts-for-unis :: ‹fm ⇒ id list ⇒ fm› where
  ‹consts-for-unis (Uni p) (c#cs) =
    consts-for-unis (sub 0 (Fun c []) p) cs› |
  ‹consts-for-unis p - = p›
```

If the formula is quantified and the list has more constants we do the substitution, otherwise we simply return the original formula. The function is designed to mimic the *Uni-E* rule. It is important here to substitute before making the recursive call to make the following proof easier:

```
lemma consts-for-unis: ‹OK (put-unis (length cs) p) [] ⟹
  OK (consts-for-unis (put-unis (length cs) p) cs) []›
proof (induct cs arbitrary: p)
  case Nil
  then show ?case
    by simp
next
  case (Cons c cs)
  then have ‹OK (Uni (put-unis (length cs) p)) []›
    by simp
  then have ‹OK (sub 0 (Fun c []) (put-unis (length cs) p)) []›
    using Uni-E by blast
  then show ?case
    using Cons by simp
qed
```

We are proving that given a proof of some formula with some number of universal quantifiers in front, we can derive a proof of the formula with the universal quantifiers substituted for constants. Only the *Cons* case is interesting. We start by unfolding the assumption one level, putting the *Uni* constructor outermost. Then we eliminate this for the constant c using the *Uni-E* rule. This substitution can go through the call to *put-unis* using the *sub-put-unis* lemma making the induction hypothesis apply. The recursive call in *consts-for-unis* is made on exactly the term this applies to so the simplifier can finish the proof. By recursing before-hand this would not be the case and the lemma would be correspondingly harder to prove.

9.1.5 Variables for Constants

Given some example formula with free variables it has so far gone through the following transformations:

$$\forall p(0,1,3) \rightsquigarrow \forall\forall\forall\forall p(0,1,3) \rightsquigarrow \forall\forall\forall p(0,1,a) \rightsquigarrow \forall\forall p(0,1,a) \rightsquigarrow \forall p(0,c,a)$$

That is, assuming the list of constants used to eliminate the quantifiers was $[a, b, c]$, as the outermost is eliminated first, so that becomes a, while b is lost as there is no variable 2 and 1 is replaced by c. The 0 is bound and thus not replaced. Each of these substitutions were done for the variable 0, but because we passed binders on the way all of the variables were affected.

Now we need to know what to substitute the constants for to obtain the original formula. To do this we count how many added binders we passed during a specific substitution as that corresponds to the number of times 0 was incremented and thus the variable ultimately replaced by the constant. For instance a passed two added binders (the third was eliminated just before the substitution) so we get:

$$(\forall p(0,c,a))[2/a] \rightsquigarrow \forall(p(0,c,a)[3/a]) \rightsquigarrow \forall p(0,c,3)$$

which is correct. The variable to substitute with thus corresponds to the length of the list from the position of the constant and forward, excluding the constant itself as that corresponds to the eliminated binder. This is calculated by the following function while doing the substitution:

```
primrec vars-for-consts :: ‹fm ⇒ id list ⇒ fm› where
  ‹vars-for-consts p [] = p› |
  ‹vars-for-consts p (c # cs) =
    subc c (Var (length cs)) (vars-for-consts p cs)›
```

And using the *Subtle* rule we can easily prove by induction on the list that such a substitution can be derived in the extended proof system:

```
lemma vars-for-consts:
  ‹OK' p [] ⟹ OK' (vars-for-consts p xs) []›
  using Subtle by (induct xs arbitrary: p) fastforce+
```

Finally we obtain the crucial result that substituting the variables for constants for variables returns the original formula, given that the constants are all distinct and do not appear in the given formula:

lemma *vars-for-consts-for-unis*:
‹*closed* (*length cs*) $p \Longrightarrow \forall\, c \in$ *set cs*. $c \notin$ *params* $p \Longrightarrow$ *distinct cs* $\Longrightarrow$
vars-for-consts (*consts-for-unis* (*put-unis* (*length cs*) *p*) *cs*) *cs* = *p*›
using *sub-free-params-all subc-sub* **by** (*induct cs arbitrary*: *p*) *auto*

Since the definitions line up so well the lemma can be proven automatically by Isabelle after instantiating the induction correctly and adding two lemmas. The *sub-free-params-all* lemma states that if a set of constants are free in *p*, they are still free after substituting a different constant into *p*.

Before looking at the *subc-sub* lemma we need to consider the chain of recursive calls made by *vars-for-consts* and *consts-for-unis*. Starting with the latter, we do the substitution before recursing so the first substitution passes through all the binders while the last passes through none. As passing through a binder corresponds to substituting for an incremented variable this is equivalent to the following chain of substitutions by the indicated constants, where m = *length cs*:

$$\mathit{sub}\ 0\ c_{m-1}\ (\mathit{sub}\ 1\ c_{m-2}\ (\ldots\ (\mathit{sub}\ (m\text{-}1)\ c_0\ p)))$$

The recursive call to *vars-for-consts* is made before the substitution and the variable we substitute for is effectively decremented at each call, making for a chain of calls looking like this (including the start of the chain above):

$$\mathit{subc}\ c_0\ (m\text{-}1)\ (\mathit{subc}\ c_1\ (m\text{-}2)\ (\ldots\ (\mathit{subc}\ c_{m-1}\ 0\ (\mathit{sub}\ 0\ c_{m-1}\ \ldots))))$$

It is evident that the two chains of calls line up perfectly when composed. Thus by proving that each pair of calls cancel each other out it follows that the whole chain cancels. This cancelling of pairs is what the *subc-sub* lemma proves:

lemma *subc-sub*: ‹$c \notin$ *params* $p \Longrightarrow$ *closed* (*Suc m*) $p \Longrightarrow$
subc c (*Var m*) (*sub m* (*Fun c* []) *p*) = *p*›
by (*induct p arbitrary*: *m*) *simp-all*

We can assume that the formula is closed at level *m+1* because the preceding substitution inserted the variable *m* and this is the highest variable inserted so far in the chain. From there it follows by induction on the formula that the two calls cancel out and we obtain the original formula.

It is worth reflecting on this proof which is deceptively simple. We could have substituted the constants for variables in any other order and obtained the same result, but then we would not have obtained this "telescoping" chain of calls that cancel each other out. The proof is simple only because the hard work was put into coming up with the right definitions. Even in a system like Isabelle with powerful proof search, good definitions make a world of difference.

9.1.6 Obtaining Fresh Constants

So far we have worked under the assumption that we had suitable fresh and distinct constants available for our operations. To discharge this assumption we need to prove that this is actually the case. For this we have two immediate options: Constructing suitable constants or proving that they must exist. Given the range of lemmas in Isabelle's libraries to draw on, the latter will be easier.

We start by showing that there always exists another fresh constant distinct from a given list. This is done by noting that the set of parameters used by the formula and list of constants is finite as both are themselves finite. Since the identifiers are strings, lists of characters, there are infinitely many of these and thus we can always obtain one fresh to finite set:

lemma *fresh-constant*: ⟨∃ *c*. *c* ∉ *set cs* ∧ *c* ∉ *params p*⟩
proof –
  have ⟨*finite* (*set cs* ∪ *params p*)⟩
    by *simp*
  then show *?thesis*
    by (*metis UnI1 UnI2 ex-new-if-finite infinite-UNIV-listI*)
qed

The constants are used to eliminate the universal quantifiers and we need as many of those as it takes to close the formula. Therefore we need to show that we can obtain just as many constants. Induction on the number of quantifiers needed is a good strategy as we can use the lemma above in the inductive step:

lemma *fresh-constants*:
  assumes ⟨*sentence* (*put-unis m p*)⟩
  shows ⟨∃ *cs*. *length cs* = *m* ∧ (∀ *c* ∈ *set cs*. *c* ∉ *params p*) ∧ *distinct cs*⟩

```
proof (induct m)
  case (Suc m)
  then obtain cs where
    ‹length cs = m ∧ (∀ c ∈ set cs. c ∉ params p) ∧ distinct cs›
    by blast
  moreover obtain c where ‹c ∉ set cs ∧ c ∉ params p›
    using Suc fresh-constant by blast
  ultimately have ‹length (c # cs) = Suc m ∧
      (∀ c ∈ set (c # cs). c ∉ params p) ∧ distinct (c # cs)›
    by simp
  then show ?case
    by blast
qed simp
```

We start by obtaining the m fresh constants given to us by the induction hypothesis. Moreover we use the *fresh-constant* lemma to obtain one more and ultimately we show that adding this to the rest fits the criteria.

Let us immediately put these constants to good use and show that if we can derive the universal closure of p in the original proof system then we can derive p itself in the extended system:

```
lemma remove-unis:
  assumes ‹sentence (put-unis m p)› ‹OK (put-unis m p) []›
  shows ‹OK' p []›
proof −
  obtain cs :: ‹id list› where ‹length cs = m›
    and ∗: ‹distinct cs› and ∗∗: ‹∀ c ∈ set cs. c ∉ params p›
    using assms fresh-constants by blast
  then have ‹OK (consts-for-unis (put-unis (length cs) p) cs) []›
    using assms consts-for-unis by blast
  then have ‹OK' (vars-for-consts (consts-for-unis
      (put-unis (length cs) p) cs) cs) []›
    using Proper vars-for-consts by blast
  moreover have ‹closed (length cs) p›
    using assms ‹length cs = m› closed-put-unis by simp
  ultimately show ‹OK' p []›
    using vars-for-consts-for-unis ∗ ∗∗ by simp
qed
```

We start by obtaining some suitable constants and then deriving the formula where these constants are substituted for the universal closure. Then we switch to the extended proof system to derive the formula with the original variables substituted for the fresh constants. Moreover we note that the original formula is closed at a suitable level and ultimately this allows us to show that we have derived the original formula. Again we see the importance of decomposing a proof into smaller lemmas for readability and even the ability to come up with proofs as we can up the level of abstraction, taking bigger steps with the lemmas.

Now we can close a formula ensuring validity and remove the closure again in the extended system. Thus we are ready to move on to handling assumptions.

9.2 Implications and Assumptions

Intuitively the following equivalence between proofs using assumptions and implications should hold:

$$p_1, p_2, \ldots, p_n \vdash q \equiv \vdash (p_1 \to p_2 \to \ldots \to p_n \to q)$$

We can go from the left one to the right one using the $\to I$ rule, but it is not apparent how to go back again. This section proves how, its essence being the following proof which we shall work towards:

lemma *shift-imp-assum*:
  assumes ‹*OK′* (*Imp p q*) *z*›
  shows ‹*OK′ q* (*p* # *z*)›
proof −
  have ‹*set z* ⊆ *set* (*p* # *z*)›
    by *auto*
  then have ‹*OK′* (*Imp p q*) (*p* # *z*)›
    using *assms weaken-assumptions′* **by** *blast*
  moreover have ‹*OK′ p* (*p* # *z*)›
    using *Proper Assume* **by** *simp*
  ultimately show ‹*OK′ q* (*p* # *z*)›
    using *Imp-E′* **by** *blast*
qed

We start by weakening the proof to assume p as well. Then we can derive a proof of p using the *Assume* rule and use this to eliminate the implication and obtain the wanted proof of q. This is the only place we need the *Imp-E'* rule.

The technique is simple but the proof that we can weaken the assumptions is delicate. To prove this we first need to prove that we can map an injective function over the parameters in a proof.

9.2.1 Renaming Parameters

An injective function maps distinct elements to distinct values, so by mapping it across the formulas in a proof we are effectively just renaming terms. We prove that this is legal first for *OK* and then its extension *OK'*. The proof is by induction on the rules and the cases are cumbersome but trivial and omitted:

lemma *OK-psubst*:
  ‹*OK p z* ⟹ *inj f* ⟹ *OK* (*psubst f p*) (*map* (*psubst f*) *z*)›

The *Exi-E* and *Uni-I* cases rely on the following proof that a new constant is still new after it and every parameter has been mapped by an injective function:

lemma *news-psubst*:
  ‹*news c z* ⟹ *inj f* ⟹ *news* (*f c*) (*map* (*psubst f*) *z*)›
  by (*induct z*) (*simp-all add*: *inj-image-mem-iff*)

Moving on to the same proof for *OK'*:

lemma *OK'-psubst*:
  ‹*OK' p z* ⟹ *inj f* ⟹ *OK'* (*psubst f p*) (*map* (*psubst f*) *z*)›

The *Proper* case can reuse the proof above and the *Imp-E'* case is trivial. Consider instead the *Subtle* case:

  case (*Subtle p z c s*)
  then have ‹*OK'* (*psubst f p*) (*map* (*psubst f*) *z*)›
    by *blast*
  then have ‹*OK'* (*subc* (*f c*) (*psubst-term f s*) (*psubst f p*))
      (*subcs* (*f c*) (*psubst-term f s*) (*map* (*psubst f*) *z*))›
    using *Subtle OK'.Subtle* **by** (*simp add*: *inj-image-mem-iff*)
  then show *?case*
    using ‹*inj f*› *subc-psubst subcs-psubst* **by** *simp*

The first line is given by the induction hypothesis. Next we *subc* the term *psubst-term f s* for the constant *f c*. This trick allows us to prove the case by applying the following lemma and commuting the two substitutions:

lemma *subc-psubst*: ⟨$inj\ f \Longrightarrow$
$psubst\ f\ (subc\ c\ s\ p) = subc\ (f\ c)\ (psubst\text{-}term\ f\ s)\ (psubst\ f\ p)$⟩
by (*induct p arbitrary*: *s*) *simp-all*

The insight here is that because the function is injective, substituting for $f\ c$ after the mapping affects exactly the same elements as substituting for c before. So when picking the pre-mapped terms above, this is undone by the commutation.

When proving this I noticed that I needed to commute the two calls and thought about what happens to the terms in question when doing this. Knowing this allowed me to choose the right terms above making the proof fairly simple.

9.2.2 Weakening Assumptions

We are going to prove a stronger lemma about weakening assumptions for the original proof system than the extended one. The way we did the extension allows us to do this, making the work worthwhile even outside the context of open formulas.

9.2.2.1 Without rule Subtle

The following lemma states that given a proof from some assumptions, we can add any assumptions we want and even permute all of them and the formula can still be derived:

lemma *weaken-assumptions*: ⟨$OK\ p\ z \Longrightarrow set\ z \subseteq set\ z' \Longrightarrow OK\ p\ z'$⟩
proof (*induct p z arbitrary*: z' *rule*: *OK.induct*)

Again most of the cases are cumbersome but trivial; the ones for *Exi-E* and *Uni-I* are not. We will consider the first one in depth, but omitting the lemmas used along the way. The case is instantiated like this:

case (*Exi-E p z q c*)

We need to prove *OK q z'* and we are given the following facts by the induction where we can instantiate ?-prefixed names as we want:

$OK\ (Exi\ p)\ z$
$set\ z \subseteq set\ ?z' \Longrightarrow OK\ (Exi\ p)\ ?z'$
$OK\ q\ (sub\ 0\ (Fun\ c\ [])\ p\ \#\ z)$
$set\ (sub\ 0\ (Fun\ c\ [])\ p\ \#\ z) \subseteq set\ ?z' \Longrightarrow OK\ q\ ?z'$
$news\ c\ (p\ \#\ q\ \#\ z)$
$set\ z \subseteq set\ z'$

The problem is that to apply the *Exi-E* rule we need a proof that the used constant is free in *z'* but here we only know that *c* is free in the subset *z*. To make matters worse we need to use precisely the given constant *c* to apply the induction hypotheses.

To solve this we will obtain a completely free constant, *fresh*, and substitute any *c* in *z'* with this fresh variable, resulting in the new list of assumptions *?z'*:

obtain *fresh* **where** $*$: $\langle fresh \notin (\bigcup p \in set\ z'.\ params\ p) \cup params\ p \cup params\ q \cup \{c\}\rangle$
  using *finite-params ex-new-if-finite List.finite-set infinite-UNIV-listI*
  by (*metis finite.emptyI finite.insertI finite-UN finite-Un*)

let $?z' = \langle map\ (psubst\ (id(c := fresh)))\ z'\rangle$

Crucially, *z* is also a subset of the new *?z'* since *c* is free in *z*. Thus every common element was unchanged by the parameter substitution. This allows us to derive *Exi p* from *?z'*:

have $\langle c \notin (\bigcup p \in set\ z.\ params\ p)\rangle$
  using *Exi-E news-params* **by** (*simp add*: *list-all-iff*)
then have $\langle set\ z \subseteq set\ ?z'\rangle$
  using *Exi-E psubst-fresh-subset* **by** *metis*
then have $\langle OK\ (Exi\ p)\ ?z'\rangle$
  using *Exi-E* **by** *blast*

Moreover adding the same element to both lists does not change this relation, allowing us to apply another induction hypothesis:

moreover have $\langle set\ (sub\ 0\ (Fun\ c\ [])\ p\ \#\ z) \subseteq$
  $set\ (sub\ 0\ (Fun\ c\ [])\ p\ \#\ ?z')\rangle$
  using $\langle set\ z \subseteq set\ ?z'\rangle$ **by** *auto*
then have $\langle OK\ q\ (sub\ 0\ (Fun\ c\ [])\ p\ \#\ ?z')\rangle$
  using *Exi-E* **by** *blast*

Furthermore since *fresh* was chosen to be distinct from *c*, *c* does not appear at all in *?z'* as it has been substituted away. Thus it is new to all of *p*, *q* and *?z'*:

moreover have ⟨$c \neq$ *fresh*⟩
  using $*$ **by** *blast*
then have $**$: ⟨$\forall p \in$ *set* $?z'$. $c \notin$ *params* p⟩
  using *map-psubst-fresh-free* **by** *simp*
then have ⟨*list-all* (λp. $c \notin$ *params* p) (p # q # $?z'$)⟩
  using *Exi-E* **by** (*simp add*: *list-all-iff*)
then have ⟨*news* c (p # q # $?z'$)⟩
  using *news-params* **by** *blast*

These facts are exactly those we need to apply the *Exi-E* rule, concluding:

ultimately have ⟨*OK* q $?z'$⟩
  using *Exi-E OK.Exi-E* **by** *blast*

But we needed to derive *q* from *z'*, not the modified *?z'*. Fortunately *?z'* is constructed such that we can recover *z*. We cannot simply do this by renaming *fresh* to *c*, however, as that is not an injective operation. Instead we will simultaneously map *c* to *fresh* as well, and utilize the fact that *c* is free in *?z'*. The mapping is justified by the *swap-param* lemma derived from *OK-psubst* and that the substitution cancels is shown in two steps:

then have ⟨*OK* (*psubst* (*id*(*fresh* := c, c := *fresh*)) q)
  (*map* (*psubst* (*id*(*fresh* := c, c := *fresh*))) $?z'$)⟩
  using *swap-param* **by** *blast*
moreover have ⟨*map* (*psubst* (*id*(*fresh* := c))) $?z' = z'$⟩
  using $*$ *map-psubst-fresh-away* **by** *blast*
then have ⟨*map* (*psubst* (*id*(*fresh* := c, c := *fresh*))) $?z' = z'$⟩
  by (*metis* (*mono-tags*, *lifting*) $**$ *map-eq-conv psubst-upd*)

Finally we simply need to show that *q* is unaffected by the substitution because *fresh* was to chosen to be free and we can conclude the case:

moreover have ⟨*psubst* (*id*(*fresh* := c, c := *fresh*)) $q = q$⟩
  using $*$ *Exi-E* **by** *simp*
ultimately show ⟨*OK* q z'⟩
  by *simp*

The need to employ this trick was what prompted the proof of *OK-psubst* which is also an interesting result in itself.

It follows directly from this formulation of weakening that we can permute the assumptions freely:

lemma *permute-assumptions*: ‹$OK\ p\ z \Longrightarrow set\ z = set\ z' \Longrightarrow OK\ p\ z'$›
using *weaken-assumptions* **by** *blast*

9.2.2.2 With rule Subtle

The proof of weakening for the extended proof system uses basically the same trick in the *Subtle* case as that employed above. There is one exception however, causing us to prove the weaker, but still sufficient, lemma below:

lemma *weaken-assumptions'*: ‹$OK'\ p\ z \Longrightarrow OK'\ p\ (q\ \#\ z)$›
proof (*induct p z arbitrary*: *q rule*: *OK'.induct*)

Here we only add a single element at a time, instead of considering the assumptions as sets. This is necessary because the induction in the *Subtle* case cannot tell us that the fixed c is new to z as we were allowed to assume previously. To circumvent this we make use the following functions:

let *?f* = ‹$id(c := fresh)$›
let *?f'* = ‹$id(c := fresh, fresh := c)$›

Note that the goal is to prove *OK' (subc c s p) (subcs c s (q # z))*. The important thing about these functions is then, that mapping them across *subc c s p* (and *subcs c s z*) has no effect because the c has been substituted away and *fresh* has been to chosen to be completely free:

have $**$: ‹*psubst ?f'* (*subc c s p*) = *subc c s p*›
using ‹*new-term c s*› $*$ *params-subc psubst-subc* **by** *simp*

The need for this equality was the motivation to add the *new-term c s* requirement to the *Subtle* rule. Fortunately it is not a very unnatural constraint which is why I decided to add it. Moreover mapping first *?f* then *?f'* across q cancels out:

have ‹*psubst ?f′ (psubst ?f q) = psubst (id(fresh := c)) (psubst ?f q)*›
  using ∗ *psubst-fresh-free psubst-upd*
  by (*metis* (*no-types*, *lifting*) *fun-upd-twist UnI2 insertCI*)
then have ∗∗∗∗: ‹*psubst ?f′ (psubst ?f q) = q*›
  using ∗ *psubst-fresh-away* **by** *fastforce*

Therefore we apply the induction hypothesis at q = *psubst ?f q* and use the *Subtle* rule to obtain the following proof:

have ‹*OK′ (subc c s p) (subc c s (psubst ?f q) # subcs c s z)*›
  using *Subtle OK′.Subtle* **by** *fastforce*

And then we can eliminate the constant substitution because the constant it applies to has been mapped away:

then have ‹*OK′ (subc c s p) (psubst ?f q # subcs c s z)*›
  using ∗ *subc-fresh* **by** *fastforce*
then have ‹*OK′ (psubst ?f′ (subc c s p))*
    (psubst ?f′ (psubst ?f q) # map (psubst ?f′) (subcs c s z))›
  using *OK′-psubst* **by** (*fastforce simp add*: *inj-on-def*)
then show ‹*OK′ (subc c s p) (q # subcs c s z)*›
  using ∗∗ ∗∗∗ ∗∗∗∗ **by** *metis*

The remaining lines use the *OK'-psubst* result to map the injective function *?f'* across the entire proof, allowing us to use the previous results to cancel everything out.

It is really important in these proofs that we have an infinite number of identifiers available so we can always pick a fresh one. Doing so allows us to use this trick of parameter substitution to ensure calls can be commuted or cancelled.

9.2.3 Completeness

Before getting to the completeness proof we need a function for turning assumptions into implications:

primrec *put-imps* :: ‹*fm* ⇒ *fm list* ⇒ *fm*› **where**
  ‹*put-imps p* [] = *p*› |
  ‹*put-imps p* (*q* # *z*) = *Imp q* (*put-imps p z*)›

We also need to show that this preserves semantics:

lemma *semantics-put-imps*:
⟨(*list-all* (*semantics e f g*) *z* ⟶ *semantics e f g p*) =
semantics e f g (*put-imps p z*)⟩
by (*induct z*) *auto*

Now we can use the previous lemma turning a single implication into an assumption, to show that we can convert a chain of implications back into assumptions:

lemma *remove-imps*:
⟨*OK′* (*put-imps p z*) *z′* ⟹ *OK′ p* (*rev z* @ *z′*)⟩
using *shift-imp-assum* **by** (*induct z arbitrary*: *z′*) *simp-all*

Finally we can prove completeness for open formulas in the extended system:

theorem *Subtle-completeness′*:
assumes ⟨*infinite* (*UNIV* :: (*′a* :: *countable*) *set*)⟩
and ⟨∀ (*e* :: *nat* ⇒ *′a*) *f g*.
list-all (*semantics e f g*) *z* ⟶ *semantics e f g p*⟩
shows ⟨*OK′ p z*⟩

We start from the formula with the reverse list of assumptions turned into implications and assert that it is valid and thus has a valid universal closure:

proof −
let *?p* = ⟨*put-imps p* (*rev z*)⟩

have ∗: ⟨∀ (*e* :: *nat* ⇒ *′a*) *f g*. *semantics e f g ?p*⟩
using *assms semantics-put-imps* **by** *fastforce*
obtain *m* **where** ∗∗: ⟨*sentence* (*put-unis m ?p*)⟩
using *ex-closure* **by** *blast*
moreover have ⟨∀ (*e* :: *nat* ⇒ *′a*) *f g*. *semantics e f g* (*put-unis m ?p*)⟩
using ∗ *valid-put-unis* **by** *blast*

Thus we can derive this version of the formula in the original proof system via the original completeness proof. From this we can derive first the unclosed formula in the extended proof system and finally turn the implications back into assumptions giving us our final proof:

```
  ultimately have ‹OK (put-unis m ?p) []›
    using assms completeness by blast
  then have ‹OK' ?p []›
    using ** remove-unis by blast
  then show ‹OK' p z›
    using remove-imps by fastforce
qed
```

Thus we have successfully extended the completeness proof to open formulas by addition of the sound *Subtle* rule. The price for this was an extra inference rule, but as this rule is provably sound it is a small price to pay. Moreover this rule is only needed, along with implication elimination, in a step after the derivation in the original proof system.

We would not need *Subtle* if either it could be derived in the original proof system or the universal closure could be removed in a different way, but, as described, it is unclear how to formalize either of these. We can however, get away with a simpler version of *Subtle* if we do not care about assumptions. This is the topic of the next section.

9.3 A Simpler Rule Subtle

We did not use the *new-term c s* constraint on *Subtle* until the proof of weakening. Nor did we use elimination implication until we needed to turn implications back into assumptions. Furthermore we only operated on empty lists of assumptions when removing the universal closure. So if we do not care about assumptions we can get away with the following simpler extension:

```
inductive OK-star :: ‹fm ⇒ fm list ⇒ bool› where
  Proper: ‹OK p z ⟹ OK-star p z› |
  Subtle: ‹OK-star p [] ⟹ OK-star (subc c s p) []›
```

Here we only allow application of *Subtle* to proofs from no assumptions and we place no restriction on c. Of course this proof system is still sound:

```
theorem soundness-star: ‹OK-star p [] ⟹ semantics e f g p›
  by (simp add: soundness-star')
```

We need to prove new versions of *vars-for-consts* and *remove-unis* that use the new version of *Subtle*:

```
lemma vars-for-consts-star:
  ‹OK-star p [] ⟹ OK-star (vars-for-consts p xs) []›
  using Subtle by (induct xs arbitrary: p) simp-all

lemma remove-unis-star:
  assumes ‹sentence (put-unis m p)› ‹OK (put-unis m p) []›
  shows ‹OK-star p []›
```

Note that *simp-all* is enough to prove *vars-for-consts-star* now; we do not need *fastforce* as we did for the more complex rule.

The completeness proof for *OK-star* is also simpler than the one for *OK'* since we do not have to deal with assumptions:

```
theorem completeness-star:
  assumes ‹infinite (UNIV :: ('a :: countable) set)›
    and ‹∀ (e :: nat ⇒ 'a) f g. semantics e f g p›
  shows ‹OK-star p []›
proof -
  obtain m where *: ‹sentence (put-unis m p)›
    using ex-closure by blast
  moreover have ‹∀ (e :: nat ⇒ 'a) f g. semantics e f g (put-unis m p)›
    using assms valid-put-unis by blast
  ultimately have ‹OK (put-unis m p) []›
    using assms completeness by blast
  then show ‹OK-star p []›
    using * remove-unis-star by blast
qed
```

If one wanted to be absolutely certain that NaDeA is complete for open formulas, either of these versions of *Subtle* could be added depending on whether one cares about assumptions or not. Again, as this extension is very conservative, it suggests that NaDeA is complete on its own and this is simply difficult to prove. This section has provided a formalized alternative to that proof.

Chapter 10

Conclusion

The goal of this thesis was to formalize first-order logic and this goal has certainly been fulfilled. The syntax, semantics soundness and completeness have all been formalized in Isabelle; completeness using a classical proof that applies only to sentences and through my own extension to open formulas.

The next two sections will respectively discuss the obtained results reflecting on insights gained in the process and point out future work.

10.1 Discussion

This thesis has presented the formalized natural deduction proof system NaDeA with a thorough description of its soundness and completeness proofs. As described in the introduction, proof systems and formalizations are useful means for software verification, a field that is becoming more and more important with the increasing use of computers in our daily lives. As such a thorough description of how a textbook proof can be formalized may help in the formalization of other proofs. Anyone seeking to develop their own proof system specifically, may use this work as a foundation for that work.

Chapter 9 presented an extension of the completeness proof to open formulas, which are rarely considered, a common solution being to universally close them before deriving them. That chapter analyzed the problem and explained why one strategy was used over another for the solution. This insight into what makes proofs easier or harder to formalize should also be applicable more widely. For this reason a few more insights are provided here:

When doing induction proofs it is often easier to prove a more general theorem than a more specific one because the induction hypothesis is correspondingly stronger. Specifically, avoiding so-called "magic numbers" that are known from general programming and generalizing them to any number, in combination with Isabelle's *arbitrary* mechanism, can make a hard proof trivial. An example in this work is assuming *closed* m where the m is always instantiated to zero.

Giving long expressions definitions with *defines*, or syntactic abbreviations using the *let* or *is* constructs can provide mental leverage by hiding unimportant details. A familiar name can be given instead of having to unpack a compound expression whose details may not matter. Picking good and consistent names is an important aspect of this, e.g. using C for consistency properties and S for sets of formulas. This also makes the proof more readable and enables a closer resemblance to a possible paper-and-pencil version of it.

In Isabelle facts may be named or referred to by copying them literally. Striking a good balance between these two possibilities was not always easy, but introduced names can help here by making the literal facts shorter. The literal fact $S \in ?C$ can be understood directly while S_in_C may require finding the definition, e.g. to answer what S and C refer to. For long facts that are used a lot, giving them a name can be the only way to ensure the proof is still readable.

In situations where an expression is rewritten the *also* mechanism exposes this clearly, while accumulating facts with *moreover* resembles the thinking behind the proof. On the other hand, one should not go out of one's way to use these to avoid names or literal facts as that will only obscure the proof.

Finally the proof search facilities in Isabelle are very powerful but should be used with consideration. We write proofs in the declarative style such that we may read them again, and if the steps between facts are too large to understand for humans it is useless. Instead the proof should either take smaller steps or the large step be justified by a named lemma that is probably an interesting result in itself. Doing so helped me both to understand and explain the provided completeness proof as well as to develop its extension. Some of these lemmas may be added to the simplifier if the reasoning is obvious but otherwise should be referenced explicitly when used, for the sake of clarity. If the goal is to formalize all proofs one day, and that is a very reasonable goal, we should follow these guidelines to ensure the proofs are not obscured by the formalization.

10.2 Future Work

As noted previously the formalized version of Fitting's proof contains some redundancy compared to the textbook proof. This redundancy occurs because Fitting treats the different types of formulas uniformly in his proofs but the formalized proof is less abstract, handling each syntactic case distinctly.

Trying to classify types of formulas abstractly in the formalization could make the proofs resemble Fitting's more and make for a shorter formalization. The completeness proof might then be instantiated for a concrete syntax and semantics by proving certain properties about these, e.g. consistency of the semantics. This would make it even easier to obtain completeness results for other natural deduction proof systems.

Specifically for NaDeA, an obvious improvement would be to get rid of the *Subtle* rule by either deriving it from the existing rules or by proving that the universal closure can be removed in a different way. This should certainly be possible, but requires reasoning in a different way that is likely harder to formalize.

Bibliography

[Avi06] Jeremy Avigad. "Review of Calixto Badesa, The Birth of Model Theory: Löwenheim's Theorem in the Frame of the Theory of Relatives". In: *The Mathematical Intelligencer* 28.4 (September 2006).

[Ben12] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. 3rd ed. Springer, 2012.

[Ber07a] Stefan Berghofer. "First-Order Logic According to Fitting". In: *Archive of Formal Proofs* (August 2007). `http://isa-afp.org/entries/FOL-Fitting.shtml`, Formal proof development.

[Ber07b] Stefan Berghofer. "POPLmark Challenge Via de Bruijn Indices". In: *Archive of Formal Proofs* (August 2007). `http://isa-afp.org/entries/POPLmark-deBruijn.shtml`, Formal proof development. ISSN: 2150-914x.

[Ber12] Stefan Berghofer. "A Solution to the PoplMark Challenge Using de Bruijn Indices in Isabelle/HOL". In: *Journal of Automated Reasoning* 49.3 (2012), pp. 303–326.

[Bru72] N. G. de Bruijn. "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392.

[BU07] Stefan Berghofer and Christian Urban. "A Head-to-Head Comparison of de Bruijn Indices and Names". In: *Electronic Notes in Theoretical Computer Science* 174.5 (2007), pp. 53–67.

[Bur14] Stanley Burris. "George Boole". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2014. Metaphysics Research Lab, Stanford University, 2014.

[Cri12] Common Criteria. *Common Criteria for Information Technology Security Evaluation*. 2012. Accessed 4 June 2017.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. 2nd ed. Graduate Texts in Computer Science. Springer, 1996.

[Hal08] Thomas C. Hales. “Formal Proof”. In: *Notices of the American Mathematical Society* 55 (2008), pp. 1370–1380.

[Har08] John Harrison. “Formal Proof — Theory and Practice”. In: *Notices of the American Mathematical Society* 55 (2008), pp. 1395–1406.

[Hea80] Percy John Heawood. “Map-Colour Theorem”. In: *The Quarterly Journal of Pure and Applied Mathematics* 24 (1880), pp. 332–338.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science — Modelling and Reasoning about Systems*. 2nd ed. Cambridge University Press, 2004.

[IsaFoL] *IsaFoL: Isabelle Formalization of Logic*. URL: `https://bitbucket.org/isafol/isafol/` Accessed 30 June 2017.

[KHB16] Alexander Krauss, Brian Huffman, and Jasmin Blanchette. *Theory Countable*. 2016. URL: `http://isabelle.in.tum.de/website-Isabelle2016/dist/library/HOL/HOLCF/Countable.html` Accessed 16 June 2017.

[Pla16] Jan von Plato. “The Development of Proof Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University, 2016.

[Sch17] Anders Schlichtkrull. *Formalization of First-Order Unordered Resolution*. April 2017. URL: `https://bitbucket.org/isafol/isafol/src/master/Unordered_Resolution/`.

[Sel89] Jonathan P. Seldin. “Normalization and Excluded Middle. I”. In: *Studia Logica* 48.2 (1989), pp. 193–217.

[Vil15] Jørgen Villadsen. “ProofJudge: Automated Proof Judging Tool for Learning Mathematical Logic”. In: *Exploring Teaching for Active Learning in Engineering Education (ETALEE 2015)*. Copenhagen, Denmark: IUPN - IngeniørUddannelsernes Pædagogiske Netværk, 2015, pp. 141–148.

[VJS17] Jørgen Villadsen, Alexander Birch Jensen, and Anders Schlichtkrull. “NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle”. In: *IfCoLog Journal of Logics and their Applications* 4.1 (2017), pp. 55–82.

[Wen16a] Makarius Wenzel. *Miscellaneous Isabelle/Isar examples*. December 2016. URL: `https://isabelle.in.tum.de/dist/library/HOL/HOL-Isar_Examples/document.pdf` Accessed 19 June 2017.

[Wen16b] Makarius Wenzel. *The Isabelle/Isar Reference Manual.* December 2016. URL: `http://isabelle.in.tum.de/doc/isar-ref.pdf` Accessed 19 June 2017.

[Wen99] Markus Wenzel. "Isar — A Generic Interpretative Approach to Readable Formal Proof Documents". In: *Theorem Proving in Higher Order Logics, 12th International Conference (TPHOLs 1999), Proceedings.* Nice, France, September 1999, pp. 167–184.

[Zal17] Edward N. Zalta. "Gottlob Frege". In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Spring 2017. Metaphysics Research Lab, Stanford University, 2017.