Normalization of First-Order Logic Proofs in Isabelle

Bo Kjellerup [1]
E-mail: kokdg@diku.dk

March 1, 1999

[1]M.Sc. student at DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø

Abstract

Types are important in functional programming. First order logic can be viewed as, using the Curry-Howard isomorphism, a very strict type theory. Proven first order logic propositions are theorems. Hereby theorems are types and, by the Curry-Howard isomorphism, a "linearized proof" of a theorem can be viewed as a program having the theorem as its type.

A proof of a first order logic theorem can be created automatically using some strategy for proof search. A generated proof can be huge[1]. Such a generated proof of the theorem can be reduced to be a minimal proof using techniques adopted from the theory of λ-calculus.

Using this technique a program can be specified as a first order logic theorem, and the automatically derived program hereof can be made efficient. Hereby proof search tools implementing the proof reduction can be used to help writing efficient programs.

To show how this will be done in an environment that can be used for proof search, the proof reduction is implemented in the Isabelle theorem prover. Furthermore, to show the proof reduction can be used to make useful programs, the proof reduction is used on examples having inductive datatypes using primitive recursion.

The main goal of this thesis is to describe first order logic and develop the necessary theory for reasoning about first order logic proofs such that the following can be implemented:

- An implementation of a system for creating first order logic proofs that can be reasoned about.
- An implementation of a reasoning system for first order logic proofs such that proofs can be reduced to some desired form.
- An implementation of a tactic that does reduction of a proof to some desired form automatically.

All 3 goals have been achieved. Automatic tactics reducing proof terms to normal form respectively weak head normal form are created. These two tactics are compared and it turns out that reducing proofs to normal form are much more time consuming than reducing to weak head normal form, but normal form proofs may contain fewer terms than weak head normal form programs, and may be more readable for a human because normal form proofs do not contain redexes.

[1] By Murphy's law, they *will* be huge

Contents

List of Figures

Chapter 1

Introduction

This master's thesis in computer science concerns first order logic and program reduction. The aim of this project is to develop and describe a tool that, given a specification of an algorithm, can create a program that implements the specified algorithm. The program shall fulfil some requirements specifying that

- The program must be as small as possible for the given algorithm, where small is defined such that the run-time of the program is minimised.
- The program must be follow the algorithm. If the algorithm is not fast, the program will not be fast.
- The program implementing the algorithm must be correct.

A process leading to such a program given a specification of an algorithm can be the following:

- Prove the algorithm, perhaps using some automated proving tool.
- Reduce the generated proof to be as small as possible.
- Extract a program out of the reduced proof.

Automated proving and program extraction are a well studied areas. This paper concentrates on implementing a language for specifying algorithms and reducing proofs.

Using the above method for deriving programs will give a fast program for a given algorithm. Most important, the program is proven to be error-free if the algorithm is error-free.

1.1 Why This Project

In fall 1995, a Ph.D. course at Department of Computer Science at University of Copenhagen focused on logics and defining programming languages using natural deduction style logic. At one of the last lectures of the course, a presentation was given on how proofs in some natural deduction style logic can be viewed as programs.

The viewing of a proof as a program seemed to be worth studying since with automatic proving and thereby automatic program creation, there would be no need for programmers. Only specifiers

creating program specifications would be needed. If a specification is formally specified, which is a necessary requirement if a specification must be evaluable, it could be called a program in a higher order program language. Hereby some natural deduction style logics turns out to be programming languages.

To avoid that this thesis will be too big, the focus is on first order logic. To avoid too many human errors in the proofs, an automated proving and proof search tool named "Isabelle" is used for implementations.

1.2 Prerequisites for The Reader

The reader of this thesis should be familiar with the simply typed λ-calculus, functional programming languages and the propositional calculus.

To reach a broader circle of readers this thesis is (tried) written in English.

This thesis is based on the principle that if an explanation of a subject needs an example then the subject is not explained good enough, therefore there are no redundant examples, only examples presenting intuitive meanings of definitions. As well the aim of this thesis is to write something filled with quality and not with quantity; therefore the text may be somewhat compact.

1.3 About Logic, Proofs and Programs

Algorithms for programs can either be stated informally or formally. Informal statements can hardly be dealt with by computers since it is difficult to give a formal description of informality, and computers cannot handle informality. Thereby a formal language for algorithms can be useful. Such a formal language can be a logic. Then the algorithm may be stated as a logic proposition.

A logic proposition can be proven to be valid; a proven proposition is a theorem. Given the right preconditions, a proof of a theorem can be contracted to a program matching a type of some functional programming language.

1.3.1 Logic, Propositions and Proofs

Logic is a formalism for reasoning. An instance hereof is a logic. A logic can be expressed as a formal theory. Specifying the logic as a formal theory helps implement the logic as a set of formal statements.

A proof of a proposition can be specified in numerous ways. As text, inference trees, linearised inference trees, etc. Such a proof contains a sequence of propositions where each proposition is either an axiom or a direct consequence of applying an inference rule on some preceding propositions of the sequence; the last proposition in the sequence is the proven sequence.

Often a proof of a proposition is discarded when the proposition is proven to be a theorem since then the proof will not be needed anymore. This tradition of answering "yes" or "no" to the question whether a proposition is true or not belongs to Tarski [6, pp. 4-5]. In the long view it is not very satisfying only to know that a proposition *can* be proven since it is much more interesting to know *how* the proposition is proven.

1.3.2 Normalizing Proofs

A proof of a proposition contains information about *how* the proposition is proven, whereas the theorem only states that the proposition *can* be proven. When a theorem has to be re-stated, it is convenient to have a proof of the theorem. Because of the time spent on re-stating, it is an advantage if the proof is small.

Re-stating theorems can be needed where an implementation of a formal theory and additional theorems must be moved from one system to another, eventually just re-stated on the same system.

Proofs often have a lot of redundant information since people (and, especially, automata) often make a set of proof steps and inverts the steps afterwards. Removing the redundancy makes the proofs smaller such that the proofs will use less resources. Often a proof is created using already proven theorems, so a proof can be created by concatenating proposition sequences proving the used theorems. Such a proof made by concatenating proposition sequences can have a lot of redundancy and may be reduced a lot, even so much that the new proof is a lot smaller than the theorems used for the proofs.

The process of removing redundancy in proofs is called *normalizing*.

1.3.3 Contracting Proofs to Programs

By the Curry-Howard isomorphism, a proof of a theorem can be viewed as a program, where the theorem can be viewed as the type of the program. Proofs have more information than programs. The additional information of a proof can be thrown away and the remaining parts form a program. If a program must be extracted from a proof, the proof must be represented in a manner such that the same terms are used both in the proof and the program, also the proof terms must reflect the semantics for the program terms.

Finding information parts that can be thrown away, program extraction, is closely studied in [1]. A short example on how to create a proof and extract a program from the proof is in [8].

By tracing a proof normalization, much more information can be obtained about a proof and a program's behaviour, both for

- the trained user, who want to see how a program for an algorithm is constructed and extracted based on an automatic generated proof,
- an untrained user that just want to see the evaluation steps of a program,
- proof (and program) reduction tracing for finding errors in that makes a theorem erroneous.

The program semantics implemented in this thesis is typed λ-calculus.

1.3.4 Advantages and Disadvantages of Machine Constructed Programs

One of the biggest advantages of a machine constructed program is that the program is one hundred percent error free since it is not only a program but also a proof of the program's specifications. If some error should come up during execution of the program, there is an error in the specification (on the assumption that the implementation of the program constructor is proven to be error free). Hereby program errors are moved up a level toward the human by which program errors should be easier to find.

The biggest disadvantage of machine constructed programs is that they may not be as efficient as if they were constructed using some intelligence. Especially if the programs are constructed without normalizing. Even with normalizing the machine constructed programs can be slowly since the algorithm stated is the one implemented.

1.3.5 Proof Search System Isabelle

One of many proof search systems is "Isabelle". Isabelle implements a meta logic in which different object logics can be specified as formal theories. Different tactics for proving proofs are implemented, as well as tacticals for combining tactics.

Isabelle implements many object logics, among these are first order logic, Zermelo-Fraenkel set theory, sequent calculus and logic for computable functions. The implementation of first order logic will be used for inspiration.

1.4 Case Studies

Several minor case studies are included in this thesis. Each logic theory studied is implemented in Isabelle as an object logic and examples are given on how to create proofs using the implementation of the object logic.

A case study is given where a big example is given to show how to build an object logic implementing inference rules for permutations of lists and rules for sorted lists. The case study specifies a proposition describing a sorted list, the proposition is proven and the proof of the proposition is normalized. The proof turns out to be a proof that can be extracted to a program implementing the insertion sort algorithm.

1.5 Contents of This Thesis

This thesis is in two parts, one including theory, one including implementation. The theory part highly focuses on logics and the theory on proving and proof normalization. The implementation part includes case studies implementing object logics.

1.5.1 Theory

The theory part of this thesis elaborates in chapter 2 on what logic is in general, how to reason about a logic, and first order logic. Proofs for first order logic propositions are linearized in chapter 3, including discussions on the Curry-Howard isomorphism, Heyting interpretation, proof terms and proof equivalence.

Proof reduction is discussed in chapter 4, based on techniques adopted from λ-calculus. A logic for proof reduction is developed. Strategies on proof reduction are discussed. For being able to create programs using more than just the pure λ-calculus, inductive datatypes based on the Peano axioms and recursive programs are defined in chapter 5.

1.5.2 Implementation

The proof search and theorem proving system Isabelle is discussed in chapter 6 and each of the object logics discussed theoretically are implemented and demonstrated using lots of examples. A SML module implementing a functor returning tactics, normalizers, for proof reduction using a reduction strategy is implemented in chapter 7. Eventually a case study on insertion sort in chapter 8 demonstrates the usability of the theory applied on a real-life example.

1.6 Acknowledgements

John Hatcliff, assistant professor at Department of Computer Science at Oklahoma State University, teached a course on first order logic when being a visiting assistant professor at the Department of Computer Science at the University of Copenhagen. The course introduced me to a very interesting area of logics. A presentation of the course can be found at

`http://www.cs.okstate.edu/~hatclif/CD/syllabus.html`

Furthermore John Hatcliff was advisor on the first small part of this thesis. Lawrence Paulson, University of Cambridge Computer Laboratory, and Frank Pfenning, Computer Science Department of Carnegie Mellon University, answered some of my questions.

A big thank to my thesis advisor, Fritz Henglein, Department of Computer Science, University of Copenhagen, for a lot of good advice.

Rune Fog Hansen deserves a big thank for sitting looking very interested answering some of my questions without knowing what I was talking about. Jakob Grue Simonsen deserves a big thank for telling me that I'm clever and for proof reading my text. Peter Møller Neergård deserves thanks for proof reading my text. My wife deserves a big thank for ... just being there making me an easy victim of natural seduction. Sidsel, my daughter, needs a big thank for just being there (and living her first 3 months, being so cute, without having colic).

1.7 Background Literature

A good general introduction to logic is [9]. First order logic and proofs for first order logic are described in [6]. Also [16] is a very good introduction to first-order logic described using the natural deductive method of reasoning. The Isabelle system is introduced in [13].

Notice that there has not been much background literature in use. The intention of this thesis is to present some scientific work done by me not resume what other people have done, except for chapter 2 and 3 presenting the knowledge I plan to use. I have tried to remember to cite results in the papers mentioned in the bibliography whenever the results are introduced or used in this thesis but this may have failed once or twice. If I have used another persons work without mentioning it, which I hope I don't have, it is an error.

Part I

Theory

Chapter 2

First Order Logic

First order logic deserves special attention since first order logic can be the basis for constructing programs[1].

In this chapter logic in general (with special attention to mathematical logic) is described. Different methods for reasoning are analysed to build a basis for choosing a reasoning method that is formal enough to be implemented in a programming language, and also suitable for proof construction.

The concepts of logic in a general, reasoning methods, formal theories and first order logic are discussed below, such that a formal theory describing first order logic can be created. Proofs of first order logic propositions are discussed to examine how to express a proof of a first order logic proposition such that the construction of the proof can be done using the chosen method of reasoning.

2.1 Logic

A popular definition of logic is that it is the analysis of methods of reasoning [9]. Methods of reasoning concerns examining how a sentence in some language (being natural or formal) is formulated and constructed to express the meaning of the sentence. Logic studies the scheme of the sentence, not the meaning of the sentence, to improve the ability to extract the meaning of the sentence (which is not always possible, especially not if the sentence is not understandable, or not well-formed).

Logic can be expressed to cover special domains. Hereby a domain has its own reasoning system. If the reasoning system is well defined, it is a logic. Such a logic can be the law of traffic regulation for some local domain of areas defined to be roads, pavements, bike tracks, etc.

A regulation in a logic may be specified using words in some natural language to express the intention of the regulation, i.e. "it is not allowed to ride faster than road conditions allow". A regulation in a logic is often called *a rule*. Whether the intentions of a regulation is met is often discussed in court by judges and lawyers since it is a matter of interpretation. In other words, informality can be expensive.

Since intentions of a logic for use in science should not be a matter of interpretation, specifying a logic formally is an advantage.

[1] Other logics could be used as well, but first order logic is an adequately small logic and most algorithms can be expressed using first order logic.

2.1.1 Formal Theories

A formal theory specifies a logic formally. I.e. a formal theory specifying that one thing follows from another may state that

- a sentence A_j is true if A_i is true, $A_i \rightarrow A_j$, where $A_k, k \in \mathbb{N}$ are variables,
- that sentences, also called well-formed formulas, in the formal theory can only be variables or $S_i \rightarrow S_j$ where S_i and S_j are sentences,
- a sentence of the form $S_i \rightarrow (S_i \rightarrow S_j) \rightarrow S_j$ is always true (and then an axiom).

Definition 2.1.1 *A set of symbols, a set of well-formed formulas, a set of axioms and a set of relations among well-formed formulas defines a formal theory $\mathcal{T}$ [9] if the following is fulfilled:*

- *The set of symbols in $\mathcal{T}$ is countable.*
- *A subset of expressions in $\mathcal{T}$ is called well-formed formulas.*
- *A set of the well-formed formulas is called the axioms of $\mathcal{T}$.*
- *A finite set of relations among well-formed formulas is included in $\mathcal{T}$, where each relation has a fixed positive arity. The relations are called inference rules.*

The set of symbols in a formal theory $\mathcal{T}$ can be *statement letters*, *connectives* etc., depending on the aim of the logic. An expression is a sequence of symbols.

Definition 2.1.2 *If an expression is a well-formed formula, by some definition of well-form, then it is a* proposition.

Often there is an effective method to judge whether an expression is a well-formed formula or not, simply by checking whether the expression fits a given grammar defining well-formedness in $\mathcal{T}$.

Definition 2.1.3 *If a proposition can be proven, given some definition of a proof, using only rules of a formal theory $\mathcal{T}$, proposition P is a theorem in $\mathcal{T}$.*

Theorems can be used as rules for proving proofs.

Some extra information can be necessary for proving a given proposition. These informations are called hypotheses.

Definition 2.1.4 *A hypothesis of a well-formed formula P is a well-formed formula that is regarded as being an axiom.*

If a theorem proves a proposition using a hypothesis the hypothesis must be mentioned in the theorem, else the theorem is not valid.

2.1.2 Logical Connectives

In most logics axioms and relations are aimed at finding the denotation of a proposition. The denotation of a proposition is either that the proposition is valid or that the proposition is non-valid. By convention a valid proposition is "true" and a non-valid proposition is "false". Often "true" is represented by $\top$ and "false" is represented by $\bot$. If the denotation of a proposition is "false" then the proposition states something absurd, therefore $\bot$ is often called *absurdity.*

The set of symbols in a formal theory representing a logic often includes statement letters, function symbols, variable symbols and also connectives to combine other symbols due to the definition of a well-formed formula in the given theory. Often used connectives are $\rightarrow$, $\neg$ and $\forall$ with the following semantics[2]:

Definition 2.1.5 *The semantics of the connectives* $\rightarrow$, $\neg$ *and* $\forall$ *are*

Implication $\rightarrow$	$P \rightarrow Q \equiv \bot$ *iff* $P \equiv \top$ *and* $Q \equiv \bot$, *else* $P \rightarrow Q \equiv \top$
Negation $\neg$	$\neg P \equiv \top$ *iff* $P \equiv \bot$, *else* $\neg P \equiv \bot$
Universal quantification $\forall$	$\forall x.P(x) \equiv \top$ *iff* $P(t) \equiv \top$ *for all* $t \in Dom(P)$, *else* $\forall x.P(x) \equiv \bot$

If symbols $\top$ and $\bot$ are used in a formal theory they must be defined as statement letters; with $\bot$ negation can be expressed as $\neg P \equiv P \rightarrow \bot$.

Based on the above connectives other connectives can be defined, i.e. as done in [9]:

Definition 2.1.6 *The semantics of the connectives* $\wedge$, $\vee$, $\exists$ *and* $\leftrightarrow$ *are*

Conjunction $\wedge$	$P \wedge Q \equiv \neg(P \rightarrow \neg Q)$
Disjunction $\vee$	$P \vee Q \equiv (\neg P) \rightarrow Q$
Biimplication $\leftrightarrow$	$P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$
Existential quantification $\exists$	$\exists x.P(x) \equiv \neg\forall x.\neg P(x)$

Connective $\neg$ has the lovest precedence followed by (in order) $\wedge$, $\vee$, $\forall$, $\exists$, $\rightarrow$, and $\leftrightarrow$ has the highest precedence.

2.2 Reasoning

Reasoning methods are methods for connecting conclusions with premises. Often a method describes constructing a conclusion for a given set of premises. Most every-day reasoning methods are stated informally (or not stated at all[3]) of which most of them can be stated formally.

Reasoning methods that are specially relevant for first order logic are deduction methods since first order logic takes basis on a set of facts. Deduction methods are reasoning methods for deriving some conclusions based on premises that can be either facts or conclusions of other deductions, such that a conclusion is drawn only from facts.

[2]The symbol "$\equiv$" represents in $A \equiv Z$ that Z and A has the same denotation.

[3]If a specific method for reasoning cannot be stated for drawing a conclusion, the reasoning method is called "intuition". If a method of reasoning is not believed, the reasoning method is called "pure luck".

2.2.1 Reasoning in General

Given a set of premises it is often a problem to find conclusions that can be drawn from the premises. Often more than one conclusion can be drawn from a given set of premises. As well, there can be more than one set of premises leading to a given conclusion.

Forward Reasoning and Backward Reasoning

A good method for *forward reasoning* about mathematics, and thereby also programs, is a method that for each set of premises and a given rule can give exactly one conclusion that can be derived from the set of premises, if the conclusion can be derived from the premises. Thereby, a good method for forward reasoning has to be a function. As well, a good method for *backward reasoning* is a method that that for a given conclusion and a given rule, with which the conclusion is deduced, gives exactly one set of premises that with the rule leads to the conclusion, if the conclusion is deducible with the rule. Thereby, a good method for backward reasoning has to be a function.

Forward reasoning is used when a proof is proved by taking basis in a set of axioms and deducing the conclusion hereof, and therefore often called *solving*. Backward reasoning is used when a proposition is proved by repeatedly splitting the proposition in smaller propositions ending with axioms or hypotheses, and therefore often called *resolving*. Forward reasoning and backward reasoning can with advantage often be combined.

Judgements and Inference Rules

A judgement in a logic is a statement that expresses that by some assumptions a proposition is valid.

Notation 2.2.1 *A judgement is stated as*

$$A_1, \ldots, A_n \vdash P$$

where $A_1, \ldots, A_n$ are assumptions and P is the proposition.

Only if the judgement is conclusion of a proof or is an axiom of the logic the judgement is valid. In logics where hypotheses are allowed a judgement is valid if it is an hypothesis.

The property of a logic that some premises leads directly to some conclusion is called an inference rule.

Notation 2.2.2 *An inference rule is a connection among judgements. An inference rule is stated as*

$$\frac{J_1 \quad \cdots \quad J_n}{J_c}\ \textit{property name}$$

where $J_1, \ldots, J_n$ are the premise judgements and J_c is the conclusion judgement of the inference rule. If judgements $J_1, \ldots, J_n$ are proven valid then J_c is valid.

Deductions as Inference Trees

A deduction in a logic of a judgement J can be expressed as an inference tree. An inference tree is either an inference rule, and then the premises of the inference rule are denoted as leaves and the

conclusion of the inference rule is denoted as root, or the result of unifying a leave of an inference tree with the root of another inference tree. Axioms in inference trees are regarded as inference rules with 0 premises. An example of an inference tree are

$$\dfrac{\dfrac{\dots \quad \begin{array}{c} A_2 \\ \mathcal{D} \\ P_3 \end{array} \quad \dots}{P_2}\,\mathsf{N}_2 \qquad \dots \qquad \dfrac{\dots \quad A_1 \quad \dots}{P_1}\,\mathsf{N}_3}{J}\,\mathsf{N}_1$$

where

- A_1, A_2 are among the axioms or hypotheses,
- J is the judgement derived using the deduction shown by the tree,
- P_1, P_2, P_3 are derived judgements,
- inference rules $\mathsf{N}_1, \mathsf{N}_2, \mathsf{N}_3$ are used in the process of deriving J.

The derivation of P_3 from A_2 is shown as deduction $\mathcal{D}$. Deduction $\mathcal{D}$ is a short-hand for a part of an inference tree.

Definition 2.2.3 *An inference tree in a formal theory $\mathcal{T}$ is a proof tree if*

- *all inference rules used to construct the inference tree are rules of $\mathcal{T}$ or theorems derived in $\mathcal{T}$,*
- *the leaves of the inference tree all are axioms, hypotheses or theorems derived in $\mathcal{T}$.*

Inference trees are usable in for resolving a proof. The root of an inference tree holds a number of subtrees. Each subtree can be resolved to their subtrees whereby a proof is created backwards using the inference tree's structure.

2.2.2 Sequents

Sequents are judgements that allow reasoning about more than one conclusion at a time.

Definition 2.2.4 *A sequent is a judgement*

$$\underline{A} \vdash \underline{B}$$

where $\underline{A} = A_1, \dots, A_m$, $\underline{B} = B_1, \dots, B_n$ are sets of propositions.

A sequent $\underline{A} \vdash \underline{B}$ is naively interpreted by $\bigwedge_i A_i \Rightarrow \bigvee_j B_j$. The naive interpretation gives

$$\begin{array}{lll} \emptyset \vdash \underline{B} & \text{is} & (\top \Rightarrow \bigvee_j B_j) \equiv \bigvee_j B_j \\ \underline{A} \vdash \emptyset & \text{is} & (\bigwedge_i A_i \Rightarrow \bot) \equiv \neg \bigwedge_i A_i \\ \emptyset \vdash \emptyset & \text{is} & (\top \Rightarrow \bot) \equiv \text{contradiction} \end{array}$$

Structural inference rules, which are rules reasoning about the structure of judgements and not constructing or destructing propositions, allows commuting elements in $\underline{A}$ or $\underline{B}$, adding new elements to $\underline{A}$ or $\underline{B}$ and removing one of more equal elements in $\underline{A}$ or $\underline{B}$. As a structural axiom the identity axiom $P \vdash P$ is used for starting proofs.

Logical inference rules, which are rules that constructs propositions in judgements, are constructed to be symmetric, which is that the same action happens both at the assumption part of a judgement and at the proposition part of the judgement. I.e. for a connective $\bullet$ there must be rules

$$\frac{\underline{A}, P_1, \ldots, P_n \vdash \underline{B}}{\underline{A}, \bullet(P_1, \ldots, P_n) \vdash \underline{B}} \quad \text{and} \quad \frac{\underline{A} \vdash P_1, \ldots, P_n, \underline{B}}{\underline{A} \vdash \bullet(P_1, \ldots, P_n), \underline{B}}$$

Connectives can only be used to construct propositions, using logical inference rules, such that a proposition cannot be destructed. The sequent calculus is hereby used to prove propositions by constructing equal assumptions and conclusions by logical inference rules and discard a set of equal assumptions and conclusions using structural inference rules.

Using sequents gives much freedom in handling a logic's rules for proving a theorem. The judgements express knowledge about both the assumptions and the propositions of a judgement whereby it is possible to apply inference rules on both assumptions and propositions. Reasoning about a logic using sequents gives much freedom since many possible proofs prove the same theorem whereby a single erroneously proof step cannot damage a deduction.

A logic using sequents, the sequent calculus, is due to Gentzen and described in detail in [6].

2.2.3 Natural Deduction

Natural deduction was invented by Prawitz [6] and is a reasoning method for deducing proofs in some kind of natural way.

Definition 2.2.5 *A natural deduction style judgement is of the form*

$$P$$

where the judgement cannot have any assumptions.

Natural deductive reasoning allows two methods for deducing propositions. The methods are

- accepting a proposition as being proved using a hypothesis or axiom as deduction of the proposition, or
- deducing a proposition from other deduced propositions using a logical inference rule.

Inference rules in a natural deductive logic are highly symmetrical. For each connective there is at least one introduction rule introducing a connective to a proposition, which is to use the connective to compound one or more premises to form a conclusion. Also there is at least one elimination rule eliminating a connective from a premise, which is to take a compound proposition apart and get one of the compounded propositions as the conclusion.

Definition 2.2.6 *An introduction rule introducing connective $\bullet$ of arity n is a rule*

$$\frac{P_1 \quad \ldots \quad P_n}{\bullet(P_1, \ldots, P_n)} \bullet\mathsf{I}$$

Definition 2.2.7 *An elimination rule eliminating connective $\bullet$ of arity n is a rule*

$$\frac{\bullet(P_1, \ldots, P_n)}{P_i}\ \bullet\mathsf{E}_i$$

The combination of an introduction rule and an elimination rule gives the inference tree

$$\dfrac{\dfrac{\ldots \quad P_i \quad \ldots}{\bullet(\ldots, P_i, \ldots)}\ \bullet\mathsf{I}}{P_i}\ \bullet\mathsf{E}_i$$

which vanishes to the identity inference tree. In chapters 3 and 4 this connection between introduction and elimination turns out to be very important.

Hypotheses of a proposition can either be alive or dead. When a hypothesis is introduced it is regarded as being alive and belonging to the proposition deduced by the hypotheses. When a proposition is deduced using an inference rule the hypotheses of propositions used as premises of the inference rule are regarded as belonging to the proposition deduced by the inference rule.

Definition 2.2.8 *Let C be deduced by deduction*

$$\frac{\begin{array}{ccc}\vdots & & \vdots \\ P_1 & \cdots & P_n\end{array}}{C}$$

and let H_i be the parcel[4] *of hypotheses for P_i, $1 \leq i \leq n$. Then the parcel of hypotheses for C is $\bigoplus_{1 \leq i \leq n} H_i$.*

Notation 2.2.9 *Discharing a hypothesis H to obtain a deduction C is denoted as $[H]\cdots C$ in inference rules.*

As an example,

$$\frac{\begin{array}{c}[H] \\ \vdots \\ C\end{array}}{C \star H}$$

To be able to discharge hypothesis H for obtaining C the hypothesis has to be alive and belonging to C.

Discharinging a hypothesis in a proof tree is denoted by stating where the hypothesis is introduced and where the hypothesis is discharged.

Notation 2.2.10 *Let C be derived by a rule discharging a hypothesis H. The inference tree representing the deduction is then stated as*

$$\frac{\begin{array}{ccc}\cdots & \begin{array}{c}\alpha \\ H \\ \mathcal{D} \\ P\end{array} & \cdots\end{array}}{C}\ \alpha \triangleleft H$$

[4] A parcel acts like a set, except that the same element can occur more than once in a parcel. Let P_1 and P_2 be parcels, then $P_1 \oplus P_2$ is the parcel of elements that belongs to P_1 or P_2, with each element occurring as many times as it occurs in P_1 and P_2 altogether.

All alive hypotheses used in a natural deduction proof has to be a part of the theorem proven by the deduction, as stated in definition 2.1.4. The proof

$$\frac{P_1 \quad P_2}{\bullet(P_1, P_2)}\ \bullet\mathsf{I}$$

proves theorem

$$\bullet(P_1, P_2) \text{ if } P_1 \text{ and } P_2$$

since the hypotheses P_1 and P_2 are not discharged whereas the proof tree

$$\frac{\begin{array}{c}\alpha\\ H\end{array}}{H \star H}\ \alpha \triangleleft H$$

proves theorem

$$H \star H$$

since the hypothesis introduced is discharged.

It is important to notice that the natural deduction style and the sequent calculus style are equivalent. If a logic can be expressed using natural deduction style, it can also be expressed using sequent style.

2.2.4 Hilbert Style Reasoning

Most logics can be expressed as a formal theory only with axioms.

Axioms are used for reasoning about a judgement $A_1, \ldots, A_n \vdash P$ to build a sequence of propositions as a proof of P where the last proposition in the sequence is P. Each element of the sequence is either a hypothesis A_i or a consequence of unifying an axiom of the logic with one or more of the preceding propositions in the sequence. With the example formal theory defined in the beginning of section 2.1.1 a proof of Q with hypotheses P and $P \rightarrow Q$ is

$$\langle P,\ P \rightarrow Q,\ Q \rangle$$

Using an axiomatic style reasoning method gives much freedom in proof creation since it opens up for much freedom in how to choose the application order of rules and hypotheses. The above proof could also be written

$$\langle P \rightarrow Q,\ P,\ Q \rangle$$

2.3 Proofs

A proof of a theorem[5] $\Gamma \vdash_{\mathcal{T}} T$ in a logic represented by a formal theory $\mathcal{T}$ is a finite sequence S of n propositions $S_1 \ldots S_n$ such that $S_n = T$. Each proposition S_i is either

- an axiom of $\mathcal{T}$,
- a hypothesis in Γ,
- an immediate consequence of applying an inference rule N of $\mathcal{T}$ with arity m on m propositions $S_{j_1}, \ldots, S_{j_m}$ of S with $j_k < i$.

[5] That $\Gamma \vdash_{\mathcal{T}} T$ is a theorem means that T is a proposition of $\mathcal{T}$ and derivable from hypothesis Γ and the rules and axioms in $\mathcal{T}$.

The aim of a proof of $\Gamma \vdash_{\mathcal{T}} T$ is to state that $\Gamma \vdash_{\mathcal{T}} T$ is provable by specifying why it is provable in such a rigid way that there is no doubt that $\Gamma \vdash_{\mathcal{T}} T$ is provable. As soon as the proof is accepted, it is no longer of any use and can be discarded. Then the fact $\Gamma \vdash_{\mathcal{T}} T$ can be regarded as common knowledge which is also called a theorem.

A theorem can have more than one proof. Proofs of the same theorem can be different, but the proofs are equivalent since they prove the same proof. Therefore a theorem may be regarded as an equivalence class of proofs.

Definition 2.3.1 *If a proof p proves theorem P, then p is a member of equivalence class P,*

$$p \in P$$

2.3.1 Use of Proofs and Theorems

Theorems are constructed to state a result. Either because it is an important result or since the theorem is profitable because it can be used for proving other theorems. As a theorem represents a derivation sequence of a logic, the theorem can be placed in another derivation sequence of the same logic as a short-hand for the theorem's derivation sequence. Thereby a theorem of a logic can serve the same purposes as axioms, hypotheses and inference rules of the logic.

Sometimes a proof sequence is annotated with information or structured for improving readability. The usability of pure proof sequences can be discussed. A proof sequence S may not be easily verifiable, naively or by hand, since nothing states how a proposition $S_i \in S$ is created. If S_i is not an axiom or hypothesis then S_i is (either invalid, and the proof sequence is invalid, or) conclusion of an application of an inference rule on preceding propositions of the proof sequence S.

To decide which propositions and inference rule that deduct the conclusion each possible combination of propositions and inference rules shall be tried. For each inference rule of arity n there is $\prod_{m=0}^{n-1} i-m$ possible proposition sets. This set of combinations can be reduced. Often it is possible to find out which inference rule has a conclusion matching S_i based on the principal connective of S_i and the conclusion of the inference rule, but still the set of combinations are huge. If the propositions in the sequence is annotated with inference rule names and references to relevant propositions in the sequence the complexity of verifying a proof is reduced a lot.

2.4 First Order Logic

The goal of this project is to create programs from specifications. Then there is a need for a logic for specifications. A possible logic could be based on the propositional calculus which as symbols has statement letters and connectives $\rightarrow, \neg$ (plus parentheses). The symbol set of the propositional calculus is not sufficient since the statement letters cannot be parameterised nor is there a connective binding parameters. Hence the propositional calculus cannot be used, whereas the predicate calculus may be used.

Without binders there are too many boundaries on which objects can be used with the calculus. If a logic must reason about equality it has to have parameterised statement letters, else an equality statement letter cannot have any parameters to state equality for.

2.4.1 Predicate Calculus

The predicate calculus is a calculus where the statement letters can be quantified. In a logic with quantifiers variables can be either free or bound. To prevent variable capturing in substitutions sometimes a proposition must be free of a quantifying variable.

Definition 2.4.1 *A term t in a proposition $P(t)$ is free for x if a substitution $P(x)[t/x]$ does not bind any free variables in t.*

The denotation of a proposition P of the predicate calculus is either that P is valid or non-valid.

Definition 2.4.2 *A formal theory $\mathcal{PC}$ for the predicate calculus consists of the following set of symbols:*

Connectives	$\neg,\ \rightarrow,\ \forall$
Constants	$a_i, b_i, c_i, \ldots,\ i \in \mathbb{N}$, *arity 0*
Variables	$x_i, y_i, z_i, \ldots,\ i \in \mathbb{N}$, *arity 0*
Predicates	$P_i, Q_i, R_i, \ldots,\ i \in \mathbb{N}$, *arity* n, $n \in \mathbb{N}_0$
Functions	$f_i, g_i, h_i, \ldots,\ i \in \mathbb{N}$, *arity* n, $n \in \mathbb{N}_0$

The symbols are possible indexed. Constants, variables and function symbols define terms, a generalized sort of values, by

1. *Constants and variables are terms,*
2. *$f(t_1, \ldots, t_n)$ is a term iff f is a function symbol of arity n and $t_{i,i\in\{1,\ldots,n\}}$ is a term.*

which defines all terms. Notice that a constant can be interpreted as a (constant) function of arity 0. Well-formed formulas of the predicate calculus are

3. *$P(t_1, \ldots, t_n)$ is a well-formed formula iff P is a predicate letter of arity n and $t_{i,i\in\{1,\ldots,n\}}$ is a term.*
4. *$\neg P,\ P \rightarrow Q,\ \forall x.P(x)$ are well-formed formulas iff $P,\ Q$ are well-formed formulas and x is a variable.*

which are all well-formed formulas. Well-formed formulas are called propositions.

Logical axioms of $\mathcal{PC}$ are

$$P \rightarrow (Q \rightarrow P) \tag{2.1}$$

$$(P \rightarrow (Q \rightarrow R)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R)) \tag{2.2}$$

$$(\neg Q \rightarrow \neg P) \rightarrow ((\neg Q \rightarrow P) \rightarrow \neg\neg Q) \tag{2.3}$$

$$\forall x.P(x) \rightarrow P(t) \text{ if } t \text{ free for } x \tag{2.4}$$

$$\forall x.(P \rightarrow Q) \rightarrow (P \rightarrow \forall x.Q) \text{ if } x \text{ not free in } P \tag{2.5}$$

where P, Q are propositions, x is a variable and t is a term. The inference rules of $\mathcal{PC}$ are

$$\frac{P \quad P \rightarrow Q}{Q}\ MP \tag{2.6}$$

$$\frac{P}{\forall x.P}\ Gen \tag{2.7}$$

Some well-formed formulas cannot be destructed into smaller well-formed formulas, and are therefore the smallest possible well-formed formulas.

Definition 2.4.3 *A well-formed formula without connectives is called an atomic formula.*

Since the predicate calculus described above allow only terms as parameters of predicates, the calculus is of first order.

The first-order predicate calculus is usable for deriving theorems on propositions since, by Gödel's completeness theorem [9], the logically valid propositions are precisely the theorems.

The formal theory is expressed using Hilbert style where as many rules as possible is stated as axioms; rules MP and Gen are not axiomatised since the rules have to be context independent.

The definition of the predicate calculus is basically adopted from [9]. Axiom 2.3 is in [9] stated as $(\neg Q \to \neg P) \to ((\neg Q \to P) \to Q)$ which is altered in the above presentation since only in classical logic it is possible to state that $\neg\neg Q \to Q$ for an arbitrary Q. Since the logic is not classic it is intuitionistic.

Classical Logic

In classical logic there is an equivalence

$$\neg\neg P \equiv P$$

which often is expressed as the rule of the excluded middle

$$P \vee \neg P$$

that a proposition is always either true or false. To to state this equality requires knowledge of everything about all subjects. This can be possible in some areas but certainly not in all areas. The intuitionistic logic does not have the equivalence. Since theory $\mathcal{PC}$ must be as usable as possible it must be intuitionistic.

If the equivalence must be used it will often be a property of the model on which the calculus is applied, whereby the rule of the excluded middle is redundant. Moreover, classical logic can be dangerous; a judge deducing that a person is guilty if it is not correct that it is provable that the person is not guilty risks to commit murder of justice.

A logic including the rule of excluded middle is probably called classic since the use of the rule is often without thinking generalised to be valid for too many domains. Moreover the rule is very old – at least it can be dated to nearly two thousand years ago, in [19, Matthew 12:30]: "He that is not with me is against me"[6] which at that time was unsound; Vikings and many other people had never heard about the cited person and therefore did not care about him.

Predicate Calculus with Equality

The predicate calculus can be extended with equality between variables by adding some logical axioms stating equality.

[6]The sentence can be interpreted to express that people are either with the cited person or not with (i.e. against) the cited person. The use of classical reasoning is wrong. Whether the logical misuse is intentional or not is not important right now. What matters is that the classical reasoning was misused.

Definition 2.4.4 *With x and y being variables, P and "=" being predicates the axioms*

$$\forall x.x = x \tag{2.8}$$
$$x = y \rightarrow (P(x,x) \rightarrow P(x,y)) \tag{2.9}$$

defines "=" as equality between variables.

From the axioms for variable equality some theorems on term equality, reflexivity and commutivity can be derived [9].

Theorem 2.4.5 *In formal theory $\mathcal{PC}$ extended with axioms (2.8, 2.9) the theorems*

$$\vdash t = t \tag{2.10}$$
$$\vdash x = y \rightarrow y = x \tag{2.11}$$
$$\vdash a = b \rightarrow (b = c \rightarrow a = c) \tag{2.12}$$

are provable.

PROOF: The proofs are easy done by using substitution axiom (2.9); for (2.11) substitution $P : (x,y) \mapsto y = x$ is used and for (2.12) substitution $P : (x,y) \mapsto a = x \rightarrow a = y$ is used. GLAD!

With the rules for equality, terms can be compared and substituted in propositions. More important, unique existential quantification can be defined, which is not possible without equality.

Definition 2.4.6 *The connective for unique existence is defined by*

$$\exists!x.P(x) \equiv \exists x.P(x) \wedge \forall x,y.(P(x) \wedge P(y) \rightarrow x = y)$$

2.4.2 Axioms and Theorems Derived from First Order Predicate Calculus

Some theorems on predicate calculus can be proven of which the most important are mentioned in the following. The theorems will be used for creating a representation of a formal theory of first order logic such that a chosen reasoning method can be used for the representation of the theory. The theorems are proved in [9].

Deduction Theorem

The deduction theorem [9, p. 59], with a proof based solely on the predicate calculus axioms and inference rules, gives the inference rule

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \rightarrow Q}$$

if deduction $\begin{matrix}\mathcal{D}\\ \Gamma, P \vdash Q\end{matrix}$ does not catch any P-free variables in Q.

Particularization Rule

Axiom (2.4) can be modified to the *particularization rule* $\Gamma, \forall x.P(x) \vdash P(t)$ which is valid only if t is free for x in $P(x)$.

Existential Rule

As well, using the definition of the $\exists$ connective in section 2.1.2, axiom (2.4) proves rule $P(t,t) \vdash \exists x.P(x,t)$ if t is free for x in $\mathrm{P}(x,t)$ and $P(t,t) = P(x,t)[t/x]$. Having $Q(x) = P(x,t)$ for a given t, the rule can be represented as $Q(t) \vdash \exists x.Q(x)$, which is the *existential rule*.

Rule of Choice

The rule of choice states that

$$\frac{\exists x.P(x)}{P(c)}$$

where

- c is a fresh constant,
- c may not be bound in a derivation $\begin{array}{c} P(c) \\ \mathcal{D} \\ \cdots \end{array}$,
- the proposition proven by the deduction which the rule of choice is part of may not include c,
- x may not be free in $P(c)$,
- $\exists x.P(x)$ is derived in the predicate calculus.

Derived Theorems

Using the predicate calculus and the above theorems gives the following derived theorems:

Negation
$P \vdash \neg\neg P$

Conjunction
$P \wedge Q \vdash P$
$P \wedge Q \vdash Q$
$\neg(P \wedge Q) \vdash \neg P \vee \neg Q$
$P, Q \vdash P \wedge Q$

Disjunction
$P \to R, Q \to R, P \vee Q \vdash R$
$\neg(P \vee Q) \vdash \neg P \wedge \neg Q$
$P \vee Q, \neg P \vdash Q$
$P \vee Q, \neg Q \vdash P$
$P \vdash P \vee Q$
$Q \vdash P \vee Q$

Conditional
$P \to Q, \neg Q \vdash \neg P$
$\neg(P \to Q) \vdash P$
$\neg(P \to Q) \vdash \neg Q$

Biconditional
$P \leftrightarrow Q, P \vdash Q$
$P \leftrightarrow Q, Q \vdash P$
$P \leftrightarrow Q \vdash P \to Q$
$P \leftrightarrow Q \vdash Q \to P$
$P \leftrightarrow Q, \neg P \vdash \neg Q$
$P \leftrightarrow Q, \neg Q \vdash \neg P$

2.4.3 Suitable Reasoning Methods

The rules and the theorems derived from the formal theory of first order predicate calculus in section 2.4.2 can for each connective be paired such that the rules mirrors each other, i.e. is an introduction and an elimination rule, cf. definitions 2.2.6 and 2.2.7. If the rules are turned into inference rules, it is obvious that the rules will be suitable for natural deduction because of the introduction/elimination connection.

As well, the rules fits well for a sequent representation because of the symmetry between proposition and assumptions, such that the deduction of a theorem can take place both among the assumptions and the propositions.

The derivations of the first order predicate logic are already mostly axiomatised or can easily be axiomatised, whereby using Hilbert style reasoning can be done. Hereby nearly any reasoning method can be used for reasoning about first order predicate calculus.

2.4.4 Natural Deduction Representation of First Order Logic

A formal theory for first order logic is to be constructed. To decide how to construct the formal theory, a reasoning method shall be chosen. The purpose of this thesis is to create programs by proving propositions whereby the reasoning method shall support this. The natural deduction reasoning method supports Heyting interpretation and the Curry-Howard isomorphism, discussed in chapter 3, whereby the natural deduction reasoning method can be used. Natural deduction gives some of the connective elimination rules a very ugly interpretation whereas the sequent representation of the same rules are beautiful. Sadly, the sequent calculus does not support the Curry-Howard isomorphism [6], then the formal theory for first order logic most be implemented using natural deduction. Since axiomatic reasoning probably will require a certainly complex logic expressing informality and unification Hilbert style reasoning may be a bad choice. Hence the best choice is be natural deduction.

Use of Formal Theory for First Order Logic

The rules of a formal theory of first order logic decides which theorems are provable and how the theorems are provable.

The denotation of the rules decide what is provable. If no rule of the logic denotes that some connective can be either eliminated or introduced, and the connective has to be either eliminated or introduced to prove the proposition, then the proposition cannot be proven in the logic.

The semantics of the rules decide how a theorem is provable, and furthermore decides which reasoning methods can be used in the logic. I.e. the semantics can decide whether forward reasoning or backward reasoning are usable.

The totality of the denotation of the rules of a formal theory for first order logic must implement provability for the same propositions as the first order predicate calculus. The semantics of the rules of the formal theory for first order logic must allow both forward and backward reasoning. In other words the theory shall allow what in section 2.2.1 is called “a good method”.

Inference Rules and Axioms of First Order Logic Theory and the Intuitive Interpretation Hereof

Not all the derived rules for the connectives in section 2.4.2 shall be implemented as inference rules in the formal theory. With the correct choice of a pair of rules for each connective, the rest of the rules can be proved using the derived rules. This does not mean that the theory shall be independent[7] since the theory gets all too clumsy if the only connectives are $\rightarrow$, $\forall$ and $\neg$. Connectives of the formal theory for first order logic also includes $\vee$, $\wedge$ and $\exists$ as defined in section 2.1.2. Since the denotation of a non-valid proposition is $\bot$, absurdity, the $\neg$ connective will give a better meaning in natural deduction by rewriting $\neg P \equiv P \rightarrow \bot$ and add the axiom $\bot \vdash P$.

Natural deductive inference rules can be based on judgements defined in definition 2.2.1.

Definition 2.4.7 *A theorem $A_1, \ldots, A_n \vdash J$ gives reason to an inference rule*

$$\frac{A_1 \quad \cdots \quad A_n}{J}$$

Hereby inference rules are created for derived theorems

$$P \wedge Q \vdash P \qquad P \wedge Q \vdash Q \qquad P, Q \vdash P \wedge Q$$

for connective $\wedge$. The intuition behind these rules are that if two propositions are true, then one of them must be true, and that if two propositions are true then they can be packed together as one true proposition.

Inference rules are created for derived theorems

$$P \rightarrow R, Q \rightarrow R, P \vee Q \vdash R \qquad P \vdash P \vee Q \qquad Q \vdash P \vee Q$$

for connective $\vee$, except that rule $P \rightarrow R, Q \rightarrow R, P \vee Q \vdash R$ does not get $\rightarrow$ connective introduced in the premises but resolved with the deduction theorem since the rule is not dealing about the $\rightarrow$ connective. The intuition behind these rules are that if a proposition can be deduced from each one of two other propositions and at least one of these are true, then the deduced proposition is true, and if a proposition is true then it can be packed with another proposition and at least one of these propositions will be true.

For the $\forall$ connective, the particularization rule is

$$\frac{\forall x.P(x)}{P(t)}$$

where the requirement that t is free for x in $P(x)$ can be discarded if P is α converted before substitution $P(x)[t/x]$. But this interpretation of the particularization rule does not fit with the backward reasoning method since a conclusion $P(t)$ can be derived from both $\forall x.P(x)$ and $\forall x.P(t)$.

[7] No rule in an independent theory can be proved using the other rules of the same theory. Here, independency is interpreted denotationally, such that it will be possible to prove the denotation of a rule of the theory using other rules of the theory.

If the particularization is packed as part of a context

$$\frac{\forall x.P(x) \qquad \begin{array}{c}[P(z)]\\ \Lambda z \quad \vdots \\ R\end{array}}{R}$$

the problem does not come up since no free variables are exhibited in the rule conclusion. The intuition behind this rule is that if a proposition is true for any argument, then the proposition is true no matter which term it is applied on.

The deduction theorem, existential rule and rule of choice requires that variables are not captured, therefore provisos may be added on rules that introduces binders, i.e. that only variables prepared for binding may be bound. To support this a meta binding connective Λ is introduced in some meta logic in which the formal theory is specified (at the moment this meta logic is written English). The generalisation inference rule of the predicate calculus is transformed to

$$\Lambda x \, \frac{P(x)}{\forall y.P(y)}$$

Only variables that are meta level bound can be quantified; the variables are bound, ergo they are not free. The binding of the variable is done when the variable is introduced and the logic level quantification of the bound variable is delayed until the quantification is introduced.

The intuition behind the generalisation rule is that if a proposition is true for an unspecified variable that may be shared with other propositions, then the proposition is also true if the variable is not shared with other propositions.

The modus ponens inference rule can be inherited as is from the predicate calculus. The deduction theorem converts to the inference rule

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \to Q}$$

where the condition that no P-free variables may be captured in Q can be discarded by the convention that only meta bound variables can be captured. Since the premise of the deduction theorem is $\Gamma, P \vdash Q$ assumption P may be a discharged hypothesis used for proving Q. The intuition behind the modus ponens rule is that if A is true and B is true if A is true, then B must be true. The intuition behind the deduction theorem is that if Q can be deduced if P is true, then Q is true if P is true.

The existential rule can be converted to inference rule

$$\frac{P(t)}{\exists x.P(x)}$$

where the requirement that t is free for x in $P(x)$ can be discarded if x is a fresh variable. The intuition behind the rule is that if a proposition is true for a special argument, then there is some argument the proposition is true for.

Harder to convert is the rule of choice since the prerequisites is hardly expressible in the inference rule as it is. First, no constants introduced using the rule of choice may appear in the final theorem. This can only be checked by tracking the fate of constants introduced using the rule of choice which makes it impossible to have the prerequisite as a part of the inference rule. Another

solution is to cover the inference rule as part of a context

$$\frac{\exists x.P(x) \qquad \begin{array}{c}[P(c)]\\ \vdots \\ R\end{array}}{R}$$

and state that if a parameter of a proposition in an inference rule is stated, then the parameter may only be part of other propositions if the parameter is stated. In the above rule, since c is stated as a parameter of P, it may only be a parameter of R if stated, but it is not and therefore c may not be a parameter of R. It is still not assured that c may not be captured and that it is fresh, therefore c, since it is thrown away anyway, can be a meta-quantified variable. This gives inference rule

$$\frac{\exists x.P(x) \qquad \begin{array}{c}[P(z)]\\ \Lambda z \quad \vdots \\ R\end{array}}{R}$$

The intuition for the rule is that if a proposition is true for some argument then the argument can be found.

For negation, $\bot$ representing non-validity (also called absurdity) is eliminated by inference rule

$$\frac{\bot}{P}$$

The intuition behind the rule is that if an absurdity can be proven then everything can be proven. There are no introduction rule for absurdity since absurdity must only be introduced as part of assumptions for a given proposition, i.e. as hypothesis. Having a rule introducing absurdity will ruin the logic totally since every proposition then is provable. Not having an introduction rule does not ruin the natural deductive style since $\bot$ is better regarded as a predicate letter than a connective.

A first order logic with equality is more usable than a first order logic without equality. Therefore axioms (2.8) and (2.9) should be a part of the formal theory for first order logic. Since theorem (2.10) has a wider scope than axiom (2.8) this is a better choice as axiom in the formal theory. The intuition behind the axiom is that a term is equal with itself.

Substitution can be represented as inference rule

$$\frac{x = y \qquad P(x,x)}{P(x,y)}$$

but as with the particularization rule conclusion parameters are exhibited which gives problems with backward reasoning. Hence the substitution should be packed in a context rule to yell

$$\frac{x = y \qquad P(x,x) \qquad \begin{array}{c}[P(x,y)]\\ \vdots \\ R\end{array}}{R}$$

The intuition behind the rule is that if two terms are equal and a proposition is true for one of the terms, some occurrences of the term can be replaced with the second term. From the substitution rule the rules for symmetry (2.11) and transitivity (2.12) can be proven.

As with the $\bot$ predicate "=" can be regarded as a binary reserved predicate letter. Hence it does not need introduction and elimination rules, even though (2.10) could be introduction rule and (2.9) could be elimination rule for "=" as connective.

Definition of Formal Theory for First Order Logic with Equation

Definition 2.4.8 *A formal theory $\mathcal{IFOL}$ of first order logic consists of the following set of symbols:*

Symbols *of $\mathcal{IFOL}$ are*

Connectives	$\rightarrow, \forall, \vee, \wedge, \exists$
Constants	$a_i, b_i, c_i, \ldots,\ i \in \mathbb{N}$, *arity 0*
Variables	$x_i, y_i, z_i, \ldots,\ i \in \mathbb{N}$, *arity 0*
Predicates	$P_i, Q_i, R_i, \ldots,\ i \in \mathbb{N}$, *arity* n, $n \in \mathbb{N}_0$, *and special predicates* $\perp$ *of arity 0 and* $=$ *of arity 2.*
Functions	$f_i, g_i, h_i, \ldots,\ i \in \mathbb{N}$, *arity* n, $n \in \mathbb{N}_0$ *The symbols, except connectives, are possible indexed.*

Propositions *are built of connectives and first order predicates parameterised on terms. Constants, variables and function symbols define terms, a generalised sort of values, by*

5_{4^0} *Constants and variables are terms,*

6_{4^0} $f(t_1, \ldots, t_n)$ *is a term iff* f *is a function symbol of arity* n *and* $t_{i,i\in\{1,\ldots,n\}}$ *is a term.*

which defines all terms. Using the above definition of terms, propositions are

7_{4^0} $P(t_1, \ldots, t_n)$ *is a well-formed formula iff* P *is a predicate letter with arity* n *and also* $t_{i,i\in\{1,\ldots,n\}}$ *is a term.*

8_{4^0} $P \rightarrow Q$, $\forall x.P(x)$, $P \vee Q$, $P \wedge Q$, $\exists x.P(x)$ *are well-formed formulas iff* P, Q *are well-formed formulas and* x *is a variable.*

which are all propositions.

Axioms *see figure 2.1.*

Inference rules *see figure 2.1.*

where

- Λ *is a meta level quantifier,*
- *the prerequisite that if a variable is present in one proposition of an inference rule then it can only be present in other propositions of the inference rule if explicitly mentioned in the propositions,*
- *fresh identifiers for quantified variables; a fresh variable cannot appear in any other propositions in a proof than the one where it is introduced and the deductions hereof,*
- α *conversion is used to prevent variable capturing.*

2.5 Proofs in First Order Logic

In first order predicate calculus a proof of a theorem $\Gamma \vdash_{\mathcal{PC}} T$ states that $T \equiv \top$ if, for each $P \in \Gamma$, $P \equiv \top$ (where $\top$ is assumed to be a symbol of the meta logic).

Figure 2.1 The inference rules and axioms of the $\mathcal{IFOL}$ theory.

Axioms

$$\frac{}{a = a}\ \mathsf{refl}$$

Inference rules

$$\frac{\bot}{P}\ \bot\mathsf{E}$$

$$\frac{x = y \quad P(x,x) \quad \begin{array}{c}[P(x,y)]\\ \vdots \\ R\end{array}}{R}\ \mathsf{subst}$$

$$\frac{P \quad Q}{P \wedge Q}\ \wedge\mathsf{I}$$

$$\frac{P \wedge Q}{P}\ \wedge\mathsf{E}_1$$

$$\frac{P \wedge Q}{Q}\ \wedge\mathsf{E}_2$$

$$\frac{P}{P \vee Q}\ \vee\mathsf{I}_1$$

$$\frac{Q}{P \vee Q}\ \vee\mathsf{I}_2$$

$$\frac{P \vee Q \quad \begin{array}{c}[P]\\ \vdots \\ R\end{array} \quad \begin{array}{c}[Q]\\ \vdots \\ R\end{array}}{R}\ \vee\mathsf{E}$$

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \to Q}\ \to\mathsf{I}$$

$$\frac{P \quad P \to Q}{Q}\ \to\mathsf{E}$$

$$\frac{P(x)}{\forall y.P(y)}\ \forall\mathsf{I}\ \Lambda x$$

$$\frac{\forall x.P(x) \quad \begin{array}{c}[P(z)]\\ \vdots \\ R\end{array}}{R}\ \forall\mathsf{E}$$

$$\frac{P(x)}{\exists y.P(y)}\ \exists\mathsf{I}$$

$$\frac{\exists y.P(y) \quad \begin{array}{c}[P(x)]\\ \Lambda x\ \vdots \\ R\end{array}}{R}\ \exists\mathsf{E}$$

When the first order logic is implemented for natural deductive reasoning as in $\mathcal{FOL}$ it becomes much easier to verify, naively or by hand, that a sequence S proves a theorem $\Gamma \vdash_{\mathcal{FOL}} T$, simply by writing the proof as an inference tree. For each $S_i \in S$ not being an axiom or hypothesis it is clear which S_i-preceding propositions are used to prove S_i. Then it is just to discover which inference rule has been used, which makes verification a little easier.

2.6 Summary

This chapter has discussed what logic is and how to represent a logic in a formal manner using the idea of a formal theory. The semantics of the most used connectives of logics have been presented and it was shown how to derive other connectives from a basic set of connectives. In formal theories the derivations are often used since even though the same denotation can be represented by different propositions, the formal theories would get all too clumsy without the derivations.

Reasoning methods have been discussed, both as being formal and informal, including a couple of formal reasoning methods that is usable for predicate logic. For reasoning about predicate logic the natural deduction reasoning method was chosen because it supports the Curry-Howard isomorphism discussed in chapter 3.

The predicate calculus of first order was presented to have some basis for deriving a formal theory implementing first order logic with equality with the natural deduction reasoning method. The axioms and inference rules of the predicate logic were stated and extended with some very useful theorems. These extending theorems, the axioms and the inference rules of the predicate calculus were then rewritten to fit the natural deduction reasoning style. Finally the formal theory $\mathcal{IFOL}$ was stated.

A discussion of proofs described what a proof in a logic is and how such a proof can be stated. The complexity of proof verification both with proofs as sequences and proofs as inference trees was discussed to discover that structure in a proof is a big advantage.

2.6.1 Literature

For introduction to logic [9] is an excellent book. Logic in general is discussed with a special focus on mathematical logic, the propositional calculus and the predicate calculus.

Different reasoning methods on predicate logic are discussed in [6], especially the sequent calculus and natural deduction. Advantages and disadvantages of natural deduction and sequent calculus are discussed. As well the difference of the sense and the denotation of a proposition is discussed.

A good text on reasoning about programming languages is [16], which also includes a good explaining section on how to deduce using natural deduction on first order logic.

2.6.2 Directions

On top of first order logic is higher order logic where not only terms can be parameters for predicates but also predicates and functions can be parameters for predicates.

First order logic can be extended with inductive datatypes for reasoning on different sorts of objects that can be interesting reasoning about. Inductive data types are discussed in chapter 5.

A proof of a first order logic theorem is discarded when the theorem is proven. The proof is very

interesting as documentation for the theorem. Without documentation or a good idea a theorem may be nearly impossible to prove, c.f. Fermats last theorem. A method for constructing proofs that can be stored and retrieved is linearization which is discussed in chapter 3. The chapter also reveals the very interesting result that such a linearized proof can be regarded as a program.

Chapter 3

Linearizing Proofs of First Order Logic

A theorem of first order logic states that a certain proposition of first order logic *is* valid. A proof of a theorem of first order logic states *how* it can be that the theorem is valid. The proof documents the theorem. This chapter is about the documentation.

Proofs of theorems of first order logic can be stated in many different ways. In section 2.2.1 inference trees were presented and in section 2.2.4 proof sequences were presented. This chapter deals with another method closely related to inference trees which is linearized proofs.

The relevance of linearized proofs are discussed using the Curry-Howard isomorphism. Connecting the Heyting interpretation and a linearized semantic for $\mathcal{IFOL}$ proofs defines a programming language. This language turns out to be semantically equal with the λ-calculus extended with constructs to be predicate-typed.

Eventually a theory for reasoning about linearized proofs is created.

3.1 Proof Representation

Some formal methods for proof representation are inference trees and linearization. These two methods have a tight connection. Inference trees are presented in section 2.2.1.

3.1.1 Linearization as Proof Representation

A proof of a proposition P can be linearized. A linearized proof of a proposition is a flat structure that represents the proof of P. The linearized proof is constructed using the hypotheses of P and inference rules and axioms of the logic in which P is proved.

Definition 3.1.1 *Let N be an n-ary inference rule, $P_1, \ldots, P_n$ be propositions proven with proofs $\Phi_{P_1}, \ldots, \Phi_{P_n}$ and P a proposition proven with inference rule N from premises $P_1, \ldots, P_n$. Then N defines a function f_{N} such that*

$$\Phi_P = f_{\mathsf{N}}(\Phi_{P_1}, \ldots, \Phi_{P_n})$$

and Φ_P is a linearized proof of P.

3.2 Curry-Howard Isomorphism

The Curry-Howard isomorphism states that if a theorem corresponds to some type τ (of some type system equivalent with the logic in which the theorem can be proved), by some definition of correspondence, then the proof of the theorem corresponds to a program having type τ. By the Curry-Howard Isomorphism programs can be created by proving proofs. To state the Curry-Howard isomorphism for a given programming language and a given formal theory for a logic, the type system of the programming language shall be isomorphic to the formal theory of the logic.

If proofs can be represented using the same terms as in which programs are written there is a syntactic equivalence between proofs and programs. Below the syntactic equivalence between proofs of $\mathcal{IFOL}$ and terms of a programming language is stated by stating

- a linearization of $\mathcal{IFOL}$ axioms and inference rules,
- an interpretation of propositions called Heyting interpretation constructs dynamic proofs which then are programs,
- a definition of an extension of the λ-calculus.

3.2.1 Denotational Semantics of First Order Logic

Each axiom and inference rule of $\mathcal{IFOL}$ can be given a functional interpretation of the denotation of the inference rule corresponding to definition 3.1.1.

Definition 3.2.1 *An inference rule*

$$\frac{P_1 \quad \ldots \quad P_n}{C}\ \mathsf{N}$$

denotes a function

$$f_{\mathsf{N}}(p_1, \ldots, p_n) \in C \quad \textit{where } p_i \in P_i$$

where $f_{\mathsf{N}} : P_1 \times \ldots \times P_n \rightarrow C$ *is the function describing the functional interpretation of inference rule* N*,*

$$f_{\mathsf{N}} \equiv \frac{P_1 \quad \ldots \quad P_n}{C}\ \mathsf{N}$$

Interpreting inference rules as functions assigns functions to each of the inference rules of $\mathcal{IFOL}$ as shown on figure 3.1. To ensure the ability to create a denotational interpretion of all proofs in $\mathcal{IFOL}$ hypotheses must have a denotational interpretion.

Definition 3.2.2 *The denotational interpretion of a hypothesis* A *is a variable,*

$$x \in A$$

The names of the functions will get a special meaning in section 3.2.4 where they are connected with programming language constructors.

Figure 3.1 Functions corresponding to the inference rules of $\mathcal{IFOL}$.

$$\texttt{ideq}(a) \equiv \frac{}{a = a}\ \mathsf{refl} \tag{3.1}$$

$$\texttt{contr}(p) \equiv \frac{\bot}{P}\ \bot\mathsf{E} \quad \text{where } p \in \bot \tag{3.2}$$

$$\texttt{idpeel}(p,f) \equiv \frac{\begin{array}{ccc} a = b & \Lambda x.P(x,x) & \begin{array}{c}[P(a,b)]\\ \vdots \\ R\end{array}\end{array}}{R}\ \mathsf{subst} \quad \text{where } \begin{array}{l} p \in (a = b) \\ f \in \mathrm{Dom}(x) \to P(x,x)\end{array} \tag{3.3}$$

$$\texttt{<}p,q\texttt{>} \equiv \frac{\begin{array}{cc} P & Q\end{array}}{P \wedge Q}\ \wedge\mathsf{I} \quad \text{where } p \in P, q \in Q \tag{3.4}$$

$$\texttt{fst}(p) \equiv \frac{P \wedge Q}{P}\ \wedge\mathsf{E}_1 \quad \text{where } p \in P \wedge Q \tag{3.5}$$

$$\texttt{snd}(p) \equiv \frac{P \wedge Q}{Q}\ \wedge\mathsf{E}_2 \quad \text{where } p \in P \wedge Q \tag{3.6}$$

$$\texttt{inl}(p) \equiv \frac{P}{P \vee Q}\ \vee\mathsf{I}_1 \quad \text{where } p \in P \tag{3.7}$$

$$\texttt{inr}(p) \equiv \frac{Q}{P \vee Q}\ \vee\mathsf{I}_2 \quad \text{where } p \in Q \tag{3.8}$$

$$\texttt{when}(p,q_l,q_r) \equiv \frac{\begin{array}{ccc} P \vee Q & \begin{array}{c}[P]\\ \vdots \\ R\end{array} & \begin{array}{c}[Q]\\ \vdots \\ R\end{array}\end{array}}{R}\ \vee\mathsf{E} \quad \text{where } \begin{array}{l} p \in P \vee Q \\ q_l \in R \quad \text{(with hyp. } [P]\text{)} \\ q_r \in R \quad \text{(with hyp. } [Q]\text{)}\end{array} \tag{3.9}$$

$$\texttt{lam } x.\ p(x) \equiv \frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \to Q}\ \to\mathsf{I} \quad \text{where } x \in P, p \in P \to Q \tag{3.10}$$

$$p\ \texttt{`}\ q \equiv \frac{\begin{array}{cc} P & P \to Q\end{array}}{Q}\ \to\mathsf{E} \quad \text{where } p \in P \to Q, q \in P \tag{3.11}$$

$$\texttt{all } x.\ p(x) \equiv \Lambda x \frac{P(x)}{\forall y.P(y)}\ \forall\mathsf{I} \quad \text{where } p \in \mathrm{Dom}(x) \to P(x) \tag{3.12}$$

$$f\texttt{`}a \equiv \frac{\forall x.P(x)}{P(a)}\ \forall\mathsf{E} \quad \text{where } f \in \mathrm{Dom}(a) \to P(a) \tag{3.13}$$

$$\texttt{[}a,p\texttt{]} \equiv \frac{P(a)}{\exists y.P(y)}\ \exists\mathsf{I} \quad \text{where } p \in P(a) \tag{3.14}$$

$$\texttt{xsplit}(p,f) \equiv \frac{\begin{array}{cc} \exists y.P(y) & \begin{array}{c}[P(x)]\\ \Lambda x\ \ \vdots \\ R\end{array}\end{array}}{R}\ \exists\mathsf{E} \quad \text{where } \begin{array}{l} p \in \exists x.P(x) \\ (x,u) \mapsto f(x,u) \in R \\ u \in P(y)\end{array} \tag{3.15}$$

3.2.2 Heyting Interpretation

Heyting has invented an interpretation Θ for describing a proof of a proposition in first order logic. The Θ interpretation [6] assigns a proof Θ_P to a proposition P with the below definition.

Definition 3.2.3 *The Heyting interpretation of a proposition P is a proof Θ_P constructed with rules*

$$\Theta_{P\wedge Q} = (\Theta_P, \Theta_Q) \tag{3.16}$$
$$\Theta_{P\vee Q} \in \{_{left}\Theta_P, _{right}\Theta_Q\} \tag{3.17}$$
$$\Theta_{P\to Q} = \Theta_Q(p) \text{ where } p \in \Theta_P \tag{3.18}$$
$$\Theta_{\forall x:\alpha.P} = \Theta_{P[a/x]} \text{ where } a \in \alpha \tag{3.19}$$
$$\Theta_{\exists x:\alpha.P} = (a \in \alpha, \Theta_{P[a/x]}) \tag{3.20}$$
$$\Theta_{atomic} = p_{atomic} \tag{3.21}$$

where, in (3.21), p_{atomic} is some proof of an atomic formula that is carried out somewhere else.

The Heyting interpretation of a proposition is a construction that builds up a linearized proof representation. The representation is not a static proof but a function with hypotheses as arguments. In fact, the proof Θ_P is a program conforming to P as a type.

With proofs as programs propositions is needed as types. A rewriting function from propositions to types can be constructed.

Definition 3.2.4 *A function ψ from $\mathcal{IFOL}$ propositions to a type system with type constructors $*$, $+$, $\to$, $\forall$ and $\exists$ is defined by*

$$\psi(P(x_1,\ldots,x_n)) = \alpha(x_1,\ldots,x_n) \quad \text{where } P \text{ is atomic} \tag{3.22}$$
$$\psi(P\wedge Q) = \psi(P) * \psi(Q) \tag{3.23}$$
$$\psi(P\vee Q) = \psi(P) + \psi(Q) \tag{3.24}$$
$$\psi(P\to Q) = \psi(P) \to \psi(Q) \tag{3.25}$$
$$\psi((\forall x.P(x)) = \forall x.\psi(P(x)) \tag{3.26}$$
$$\psi((\exists x.P(x)) = \exists x.\psi(P(x)) \tag{3.27}$$
$$\tag{3.28}$$

where x_i is an arbitrary term variable and α is an arbitrary parameterized type variable.

With these constructions Φ_P is a program of type $\psi(P)$ for all P.

3.2.3 The λ-Calculus Extended for Predicate Typing

The λ-calculus is the basis of all functional programming languages. This basis shall be used for a calculus that can represent terms typed with an extended type system based on the predicate calculus of first order.

The simply typed λ-calculus extended with pairing is defined on a type system consisting of atomic types $\tau_1,\ldots,\tau_n$, pair[1] types $\alpha * \beta$ and function types $\alpha \to \beta$ where α and β are types.

[1] Pair can be used to define general n-tuples $\alpha_1 * \cdots * \alpha_n$. A $n-tuple$ $\alpha_1 * \cdots * \alpha_n$ can be represented as $\alpha_1 * (\cdots * (\alpha_n * \blacksquare)\cdots)$ where $\blacksquare$ is some terminator.

Definition 3.2.5 *The terms and denotation of terms of the typed λ-calculus [6] are*

9_{48} *variables* x_i *of type* α *represents any term of type* α,

10_{49} `<a,b>` *of type* $\alpha * \beta$ *is the ordered pair of* a *and* b *if* a *is of type* α *and* b *is of type* β,

11_{49} `fst(t)` *of type* α *is the first projection of* t *if* t *has type* $\alpha * \beta$,

12_{49} `snd(t)` *of type* β *is the second projection of* t *if* t *has type* $\alpha * \beta$,

13_{49} `lam` x_i`.` t *of type* $\alpha \to \beta$ *is the abstraction of* x_i *over* t *which is a function that for any* a *of type* α *associates* $t[a/x_i]$ *if* t *is of type* β.

14_{49} t `‘` a *of type* β *is the result of applying* t *on the argument* a *if* t *is of type* $\alpha \to \beta$ *and* a *is of type* α.

Often `<a,b>` is written as $\langle a, b\rangle$, `fst(t)` as $\pi_1(t)$, `snd(t)` as $\pi_2(t)$, `lam` x_i`.` t as $\lambda x_i.t$, and t `‘` a as $t\ a$. Why a notation different from the usual is chosen will be clear later.

The above definition of the simply typed λ-calculus can be extended to a more complex type system corresponding to the first order predicate calculus with type connectives $+, \forall, \exists$ for sum, all quantification and existential quantification. Since types may be parameterised and must be of first order there must be a type sort corresponding to terms which will be written $\hat{\beta}$ for term type variables.

Definition 3.2.6 *The extended predicate typed λ-calculus is the simply typed λ-calculus extended with*

15_{49} `all` $\hat{\gamma}$`.` f *of type* $\forall\hat{\beta}.\alpha(\hat{\beta})$ *is the term type abstraction of* $\hat{\gamma}$ *over* f *which is a function that for any term type* $\hat{\delta}$ *associates* $f[\hat{\delta}/\hat{\gamma}]$ *if, for any* $\hat{\epsilon}$, $f(\hat{\epsilon})$ *is of type* $\alpha(\hat{\epsilon})$.

16_{49} f `^` $\hat{\beta}$ *of type* $\alpha(\hat{\beta})$ *is the result of applying* f *on the term type* $\hat{\beta}$ *if* f *is of type* $\forall\hat{\gamma}.\alpha(\hat{\gamma})$.

17_{49} `[`$\hat{\gamma}$`,p]` *of type* $\exists\hat{\beta}.\alpha(\hat{\beta})$ *associates to a program* p *a withness* $\hat{\gamma}$ *that* p *can be of type* $\alpha(\hat{\gamma})$.

18_{49} `xsplit(p,f)` *of type* γ *is for a term* p *of type* $\exists\hat{\beta}.\alpha(\hat{\beta})$ *constructing a function* f *that for any* $\hat{\delta}$ *and any* u *of type* $\alpha(\hat{\delta})$ *associates* $f(\hat{\delta}, u)$ *if* $f(\hat{\delta}, u)$ *is of type* γ.

19_{49} `inl(p)` *of type* $\alpha + \beta$ *is the type* α *choice of a two-choice datatype if* p *is of type* α.

20_{49} `inr(p)` *of type* $\alpha + \beta$ *is the type* β *choice of a two-choice datatype if* p *is of type* β.

21_{49} `when(a,f,g)` *of type* γ *is a conditional where* a *of type* $\alpha + \beta$ *is the choice and* $f(x)$ *of type* γ *for any* x *of type* α *is corresponding to an* α *choice and* $g(x)$ *of type* γ *for any* x *of type* β *is corresponding to a* β *choice.*

The above first order predicate typed program calculus seems a bit weird. Now program expressions can be applied to a certain kind of types.

3.2.4 Proofs have Same Syntax as Programs

The Heyting interpretation Φ of proofs of $\mathcal{IFOL}$ is a program and also a linearized proof by definition 3.1.1. A linearization as defined with the denotational semantic of $\mathcal{IFOL}$ in section

3.2.1 is isomorphic to the Heyting interpretation. Therefore the linearization functions of definition 3.2.1 defines a programming language.

A comparison of the predicate typed extended λ-calculus of definition 3.2.6 and the denotational semantic of theory $\mathcal{IFOL}$ of definition 3.2.1 reveals that these have the same terms except for inference rules subst, refl and $\bot$E which are not real inference rules since they are not dealing with connectives but predicate letters. By definition 3.2.4 they have the same type system.

An important quality of the Curry-Howard isomorphism is that proof transformation is isomorphic to program reduction in the λ-calculus, hence there really is an isomorphism between proofs and programs since structures are preserved under reduction. To state the Curry-Howard isomorphism it must be stated that proof reduction equals program reduction which eventually is stated in chapter 4.

3.3 Reasoning About Proofs

To be able to state that program evaluation is equal with proof reduction it is essential to study the functions constructing the denotational semantics of rules of the $\mathcal{IFOL}$ theory.

Since proofs must be constructed a formal theory for proof construction must be created and used for proof construction. The theory is created using the denotational semantic of $\mathcal{IFOL}$. Studies of proving of proofs of the formal theory for proof construction is also essential to be able to state the last part of the Curry-Howard isomorphism.

3.3.1 Constructors and Introduction, Destructors and Elimination

Introduction rules of the $\mathcal{IFOL}$ theory follow the pattern that the conclusion of a rule is constructed by connecting the premises of the rule using some connective. Since the conclusion of an introduction rule is constructed of the premises of the inference rule the corresponding denotation function is called a *constructor*.

Similarly elimination rules follow the pattern that the conclusion of a rule is created by destructing premises of the rule and choosing a destructed part to be the conclusion. Since the conclusion of a rule is a destructed premise the corresponding denotation function is called a *destructor*.

3.3.2 Formal Theory for Obtaining Proofs

Based on the denotational interpretation of the rules of the $\mathcal{IFOL}$ theory on figure 3.1, a formal theory $\mathcal{IFOLP}$ can be constructed by adding the denotational interpretation of each $\mathcal{IFOL}$ rule as a part of each $\mathcal{IFOLP}$ rule, such that an inference rule in $\mathcal{IFOLP}$ follows the scheme

$$\frac{t_1 : P_1 \quad \cdots \quad t_n : P_n}{f(t_1, \ldots, t_n) : C} \tag{3.29}$$

where f is the denotational function, or structor, of the same $\mathcal{IFOL}$ inference rule

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

Definition 3.3.1 *A formal theory $\mathcal{IFOLP}$ of first order logic with proofs consists of the following set of symbols:*

Symbols *of $\mathcal{IFOLP}$ are*

Delimiter	:
Connectives	$\rightarrow, \forall, \vee, \wedge, \exists$
Structors	`ideq`, `idpeel`, `contr`, `<·,·>`, `fst`, `snd`, `inl`, `inr`, `when`, `lam`, `‘`, `all`, `^`, `[·,·]`, `xsplit`, *each of fixed arity,*
Constants	$a_i, b_i, c_i, \ldots$, $i \in \mathbb{N}$, *arity 0*
Variables	$x_i, y_i, z_i, \ldots$, $i \in \mathbb{N}$, *arity 0*
Predicates	$P_i, Q_i, R_i, \ldots$, $i \in \mathbb{N}$, *arity* n, $n \in \mathbb{N}_0$, *and reserved predicates* $\bot$ *of arity 0 and* $=$ *of arity 2.*
Functions	$f_i, g_i, h_i, \ldots$, $i \in \mathbb{N}$, *arity* n, $n \in \mathbb{N}_0$

Propositions *are composed of a proof part, a delimiter and a logic part where p is a proof part and P is a logic part. The proof part has the following grammar:*

22_{51} x

23_{51} `ideq(`X`)`

24_{51} `idpeel(`p_1`,`p_2`)`

25_{51} `contr(`p_1`)`

26_{51} `fst(`p_1`)`

27_{51} `snd(`p_1`)`

28_{51} `<`p_1`,`p_2`>`

29_{51} `inl(`p_1`)`

30_{51} `inr(`p_1`)`

31_{51} `when(`p_1`,`p_2`,`p_3`)`

32_{51} `lam` x`.`p_1

33_{51} p_1 `‘`p_2

34_{51} `all` X`.`p_1

35_{51} p_1 `^` X

36_{51} `[`X`,`p_1`]`

37_{51} `xsplit(`p_1`,`p_2`)`

where p_n is a proof part, x is a proof variable and X represents a term variable. The definition of proof part defines all proof parts. The logic part is built of connectives and first order predicates parameterized on terms defined by

38_{51} *Constants and variables are terms,*

39_{51} *$f(t_1, \ldots, t_n)$ is a term iff f is a function symbol of arity n and $t_{i,i\in\{1,\ldots,n\}}$ is a term,*

which defines all terms. Logic parts are then defined by

40_{51} *$P(t_1, \ldots, t_n)$ is a well-formed logic part iff P is a predicate letter of arity n and $t_{i,i\in\{1,\ldots,n\}}$ is a term,*

41_{51} *$P \rightarrow Q$, $\forall x.P(x)$, $P \vee Q$, $P \wedge Q$, $\exists x.P(x)$ are well-formed logic parts iff P, Q are well-formed logic parts and x is a variable,*

which defines all well-formed logic parts. Eventually a well-formed formula in theory $\mathcal{IFOLP}$ is a proposition $p : P$ where p is a proof part and P is a logic part.

Axioms *see figure 3.2.*

Inference rules *see figure 3.2.*

The proof part p of a theorem $p : P$ proved using the $\mathcal{IFOLP}$ theory is a proof of $p : P$. Hence $\mathcal{IFOLP}$ theorems includes their own proof. Each structor of the $\mathcal{IFOLP}$ theory is used in exactly one of the inference rules and axioms of the theory. The rules are formed following scheme (3.29) whereby each structor of theory $\mathcal{IFOLP}$ corresponds to an inference rule or axiom of theory $\mathcal{IFOLP}$ such that p is a linearization of the proof of $p : P$, as presented in section 3.1.1. By section 3.2.1 p is the proof tree for $p : P$.

Figure 3.2 The inference rules and axioms of the $\mathcal{IFOLP}$ theory.

Axioms

$$\frac{}{\texttt{ideq}(a) : a = a}\ \textsf{refl}$$

Inference rules

$$\frac{p : \bot}{\texttt{contr}(p) : P}\ \bot\textsf{E}$$

$$\frac{p : P \qquad q : Q}{\texttt{<}p,q\texttt{>} : P \wedge Q}\ \wedge\textsf{I}$$

$$\frac{p : P \wedge Q}{\texttt{fst}(p) : P}\ \wedge\textsf{E}_1$$

$$\frac{p : P \wedge Q}{\texttt{snd}(p) : Q}\ \wedge\textsf{E}_2$$

$$\frac{p : P}{\texttt{inl}(p) : P \vee Q}\ \vee\textsf{I}_1$$

$$\frac{p : Q}{\texttt{inr}(p) : P \vee Q}\ \vee\textsf{I}_2$$

$$\frac{\begin{array}{ccc} & [x : P] & [y : Q] \\ & \Lambda x\ \vdots & \Lambda y\ \vdots \\ p : P \vee Q & q_l(x) : R & q_r(y) : R \end{array}}{\texttt{when}(p, q_l, q_r) : R}\ \vee\textsf{E}$$

$$\frac{[y : P]}{\begin{array}{c} \Lambda y\ \vdots \\ p(y) : Q \end{array}}{\texttt{lam}\ x\ .p(x) : P \to Q}\ \to\textsf{I}$$

$$\frac{p : P \to Q \qquad q : P}{p\ \texttt{`}\ q : Q}\ \to\textsf{E}$$

$$\frac{\Lambda y \dfrac{p(y) : P(y)}{\texttt{all}\ x.\ p(x) : \forall z.P(z)}}{}\ \forall\textsf{I}$$

$$\frac{\begin{array}{cc} & [q : P(a)] \\ & \Lambda q\ \vdots \\ p : \forall x.P(x) & q : R \end{array}}{p\texttt{\^{}}a : R}\ \forall\textsf{E}$$

$$\frac{p : P(a)}{\texttt{[}a,p\texttt{]} : \exists y.P(y)}\ \exists\textsf{I}$$

$$\frac{\begin{array}{cc} & [u : P(x)] \\ & \Lambda x\ u\ \vdots \\ p : \exists y.P(y) & f(x,u) : R \end{array}}{\texttt{xsplit}(p, f) : R}\ \exists\textsf{E}$$

$$\frac{\begin{array}{ccc} & & [q : P(a,b)] \\ & & \vdots \\ p : a = b & \Lambda x.\ f(x) : P(x,x) & q : R \end{array}}{\texttt{idpeel}(p, f) : R}\ \textsf{subst}$$

3.3.3 Obtaining Proofs in $\mathcal{IFOLP}$

With the theory $\mathcal{IFOLP}$ proofs of the $\mathcal{IFOL}$ theory can be obtained as functions with the functions defined on figure 3.1. A theorem may state that in $\mathcal{IFOL}$ a given proposition is valid. Letting the proposition be logic part of a $\mathcal{IFOLP}$ proposition helps making the documentation of the proof since the proof part of a $\mathcal{IFOLP}$ proposition work as an accumulator during proving.

A proof can, as in the $\mathcal{IFOL}$ theory, be represented as an inference tree. A $\mathcal{IFOLP}$ proof of a proposition including the logic part $P \to P$ (which also is a $\mathcal{IFOL}$ proof of theorem $P \to P$) can be represented as inference tree

$$\frac{\begin{array}{c} y \\ x : P \end{array}}{\texttt{lam}\ x.x : P \to P}\ \to \mathsf{I}^y \tag{3.30}$$

and also as inference tree

$$\frac{\dfrac{\dfrac{\begin{array}{cc} y & y \\ x : P & x : P \end{array}}{\texttt{<}x,x\texttt{>} : P \wedge P}\ \wedge\mathsf{I}}{\texttt{fst(<}x,x\texttt{>)} : P}\ \wedge\mathsf{E}_1}{\texttt{lam}\ x.\ \texttt{fst(<}x,x\texttt{>)} : P \to P}\ \to \mathsf{I}^y \tag{3.31}$$

which are obviously different. It is obvious that inference tree

$$\frac{\dfrac{\dfrac{\begin{array}{cc} \mathcal{D} & \mathcal{D} \\ x : P & x : P \end{array}}{\texttt{<}x,x\texttt{>} : P \wedge P}\ \wedge\mathsf{I}}{\texttt{fst(<}x,x\texttt{>)} : P}\ \wedge\mathsf{E}_1$$

which is a part of inference tree (3.31), can be reduced to inference tree

$$\begin{array}{c} \mathcal{D} \\ x : P \end{array}$$

which is a part of inference tree (3.31) but this is not done since then the proof part of the root of the inference tree is no longer the proof carried out but a reduced proof. Implementing the proof reduction in $\mathcal{IFOLP}$ simply ruin the idea of the theory.

Even though, the proof reduction is what is searched for. Chapter 4 is dedicated for this.

3.3.4 Validating a Theorem

A theorem of the $\mathcal{IFOLP}$ theory proving proposition $p : P$ is easily validated. It is simply to check that the inference tree leading to proposition $p : P$ can be recreated. Since p is the proof of $p : P$ it is a question of pattern matching on proof parts to find the one inference rule that can lead to proposition $p : P$ and repeat the pattern matching process on each of the proof trees coming out of this process.

3.4 Summary

This chapter discussed proofs in general and created a theory where proofs of theorems are included in the theorems. For creating such a theory some subjects were discussed.

Creating proofs can be a difficult and complicated process. To ease the proving process a little trick of re-using already proven theorems can be used. The re-using can result in proofs exploding in size whereby validating such a proof can be overwhelming. Hence proofs can be reduced to achieve advantages of both a smaller proof and easier validation. A representation method and proof language shall be found to deal with the problem to reduce proofs. The representation method ended up to be linearized proofs in a predicate typed extended λ-calculus.

For representing proofs there was a choice between some informal method, which rapidly was rejected, inference trees and linearized proofs. Inference trees and linearized proofs turns out to be isomorphic whereby linearized proofs can be used for proof representation whereas the reasoning method can be done on proofs as inference trees.

To find a language for representing proofs the Heyting proof interpretation was discussed since this interpretation represents proofs as dynamic structures that expresses how a proof is created. Hence proofs turns out to be program like structures. This technique was used to state a part of the Curry-Howard isomorphism that proofs are programs.

For discovering how inference rules of the $\mathcal{IFOL}$ theory can be used as proof constructs the denotational semantic of $\mathcal{IFOL}$ was examined which ended up in a set of functions with one function for each inference rule. Using the denotational functions a proof of a $\mathcal{IFOL}$ proposition can be written using function composition, i.e. as a linearized proof.

Even the the language of denotation functions can be used to express a proof of a $\mathcal{IFOL}$ proposition, another language was constructed that apparently did not have anything to do with the $\mathcal{IFOL}$ theory, except for syntactical identity. The other language is the λ-calculus extended with predicate typing, pairs, conditionals and quantification.

Hence the extended λ-calculus are syntactically equal with the denotational functions of $\mathcal{IFOL}$, proofs of $\mathcal{IFOL}$ propositions turns out to be written in the same language as programs. To state that the proofs really are programs proof transformation must be isomorphic to program reduction, i.e. not only have the same syntax but also the same semantics, which is stated in chapter 4.

Each of the denotational function names of $\mathcal{IFOL}$ is grouped as either a constructor corresponding to an introduction rule or a destructor corresponding to an elimination rule. This grouping turns out to be important in chapter 4.

Since the formal theory $\mathcal{IFOL}$ represents first order logic with equality this theory is used to create a formal theory $\mathcal{IFOLP}$ representing first order logic with equality and proofs. The $\mathcal{IFOLP}$ theory represents proofs as parts of the propositions such that a proposition $p : P$ consists of a proof part p and a logic part P where P represents what is a proposition of the $\mathcal{IFOL}$ theory and p is a linear denotation functional proof of P.

Proof inference trees of $\mathcal{IFOLP}$ proofs, demonstrating the linearized proofs in combination with inference trees, demonstrates that if the proof trees are not equal then the propositions are not equal (but they can have equal logic parts). Hereby there is a unique inference tree for each $\mathcal{IFOLP}$ theorem. This is a fact because the theorem holds its proof which is the flattened inference tree; if the proofs are different then the theorems are different.

Finally validating theorems were discussed. Theorem validating of $\mathcal{IFOLP}$ theorems are not a complex problem since a proof is part of a theorem and can thereby be solved using pattern matching on the inference rules of the $\mathcal{IFOLP}$ theory.

3.4.1 Literature

Books used in this chapter is [6] having a very good section on the Curry-Howard isomorphism but only for the $\rightarrow, \wedge$ part of propositional calculus, whereas the Heyting interpretation and denotation functional semantic of natural deductive first order logic are stated in full. Also [18] has a couple of sections on the Curry-Howard isomorphism, even though not stating the isomorphism precisely and exhaustive enough to state how the isomorphism is constructed and acts. As well [18] has a section that a predicate language is fine for safe programming.

A fine book for constructing a programming language based on logic is [16].

The theory files of the Isabelle proof system introduced in chapter 6 give very good information on how to construct a linearization proof and a theory for linearization proofs. Specially the files `FOL.thy`, `IFOL.thy`, `FOLP.thy` and `IFOLP.thy` gives a good understanding of the connections between first order logics and proofs hereof.

3.4.2 Directions

Since proving proofs by proof combining can result in an explosion of proof size and the size explosion can be reduced using proof reduction, proof reduction is worth studying. This subject is dealt with in chapter 4.

The predicate typed extended λ-calculus is, by my knowledge, a hypothetical language never implemented as a programming language. This can be caused by the fact that the languages based on the typed λ-calculus with extra constructions but only one quantifier λ, i.e. sml, are sufficient for expressing all sorts of programs. Then a program in the predicate typed extended λ-calculus may be converted to a program in the typed λ-calculus. This process is known as program extraction and is not handled in this thesis. A paper on program extraction from a related language ELF is [1].

Chapter 4

Proof Reduction

Proof reduction is the process of transforming a proof such that the resulting proof fulfils some requirements the original proof does not fulfil.

In chapter 3 the Curry-Howard isomorphism was introduced. The similarity of proofs of first order logic propositions and programs in an extended predicate typed λ-calculus was discussed. If the correspondence of program reduction and proof reduction can be stated, the Curry-Howard isomorphism is completed.

With the Curry-Howard isomorphism reduction methods from the theory of λ-calculus can be used for proof reduction. The reduction methods are used for creating a formal theory for proof reduction.

The aim of this chapter is to define the concepts of proof normal forms and what else is necessary to define a reduction strategy suitable for proof reduction. Amongst the necessary concepts are redexes and reduction strategies. Since prior chapters treat first order logic, the discussion of proof reduction will deal with reduction of proofs of first order logic propositions.

4.1 Proof Normalization

Natural deduction proofs may contain introduction rules and elimination rules right after each other. These applications form redexes. A redex does not add new information to a proof since the conclusion of a redex is reached just before the redex is created, and does thereby not prove anything that is not already proved.

4.1.1 Why introducing Redexes

There is a good motivation for creating proofs with redexes. Proofs can be used for creating proofs, i.e. *combined*. If a proof of a theorem C can be built using proofs of two other theorems A and B, and a few other proof steps, the proof of C built using proofs of A and B does not require as much work as proving C from scratch. The proof of C including proofs of A and B is bigger than A and B together, and probably also bigger than a proof of C from scratch.

The process of normalization is a method to clean up a proof by removing redexes that arise at the "interfaces" of subproofs.

4.1.2 Proof Explosion

The number of redexes in a proof can explode when a redex is removed. A deduction occuring once in a proof can after a reduction occur more times whereby the same deduction may be reduced more than once.

As an example the deduction

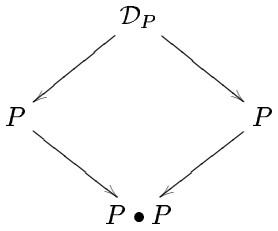
$$\mathcal{D} = \dfrac{\dfrac{\dfrac{\overset{\alpha}{P} \quad \overset{\alpha}{P}}{P \bullet P}\bullet I}{P \star (P \bullet P)}\star\mathsf{I}^{\alpha} \qquad \begin{matrix}\mathcal{D}_P \\ P\end{matrix}}{P \bullet P}\star\mathsf{E}$$

reduces to

$$\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_P \\ P\end{matrix} \quad \begin{matrix}\mathcal{D}_P \\ P\end{matrix}}{P \bullet P}\bullet I$$

If there are n proof steps in $\mathcal{D}_P$ the number of proof steps to prove $P \bullet P$ increases with $n - 2$ by reducing the redex.

Whether a proof explodes or not depends on the strategy for deciding the order in which redexes shall be reduced. Most important is that all redexes in a proof can be removed, the strategy only decides how many times the same redex has to be removed. Some proof explosion can be avoided by representing proofs as graphs instead of inference trees such that $\mathcal{D}$ is represented as



but graph reduction is not a subject of this thesis.

4.1.3 Proof Forms

Different schemes of proof forms can be used to describe proofs. A proof in normal form is free of redexes, a proof in head normal form is a sequence of introduction rules preceded by an elimination rule with hypotheses as arguments, and a proof is in weak head normal form if the last applied inference rule is an introduction rule.

The aim of describing form schemes of normality is to be able to determine how much a proof must be reduced to get the optimal benefit of proof reduction. The goal of proof reduction is to get a proof without any redexes.

Normal Form

The normal form is the wanted result of proof reduction. A proof reduces (in zero or more steps) to one and only one normalized proof by strong normalization. Proofs for a proposition created using different proving strategies can thereby be compared by comparing their normal forms.

Weak Head Normal Form

If a proof must be combined with another proof using an elimination rule in proving a theorem, the proof on which the elimination rule is applied must end in introduction rules to ensure that the resulting proof ends in a redex. The weak head normal form is a proof form for which it is possible to see if a proof can be combined with another proof or not. A proof in weak head normal form is either a hypothesis or ending in an introduction rule.

Proving Process and Normal Forms

Proof reduction is not necessary (or even helpful) in the process of deriving proofs. The proposition a proof proves does not change no matter how much the proof is reduced. Only the last done proof step is important in creating proofs. In a final proof of a theorem that is due for inspection (or whatever the proof shall be used for), there shall be no redexes. This cannot be reached by reducing provisional subproofs to weak head normal form or normal form during a proof. A discharged hypothesis inside a proof can be exchanged with a proof ending in an introduction rule by reducing an other redex. Such an inner redex will not be reduced by reducing to weak head normal form during the proving process.

4.2 Normalization in Natural Deductive First Order Logics

A redex involving a particular connective is discovered by constructing an imaginary proof ending in the elimination rule for the connective and letting the premise of the elimination rule including the connective be proven by an imaginary proof ending in introducing the connective. Then a premise of the introduction rule is equal to the conclusion of the elimination rule and the deduction of the premise forms a proof without the redex.

4.2.1 Redexes in Theory $\mathcal{IFOLP}$

Notation 4.2.1 *A deduction of the form*

$$\mathcal{D} = \begin{array}{c} \mathcal{D}_P \\ a : P \\ \mathcal{D}_Q \\ f(a) : Q \end{array}$$

shall be represented as

$$\mathcal{D} = \dfrac{\begin{array}{cccc} & & & [x:P] \\ \mathcal{D}_P & \Lambda x & \mathcal{D}_Q & \\ a : P & & f(x) : Q & \end{array}}{f(a) : Q}$$

It is clear why the two expressions of the deduction $\mathcal{D}$ above are equal. The right premise

$$\Lambda x \begin{array}{c} [x : P] \\ \mathcal{D}_Q \\ f(x) : Q \end{array}$$

states that the parametric proof $f(\cdot)$ is a proof of Q no matter what the parameter of f is (as long as the parameter is a proof of P). Therefore a random proof a of P (regarded as a constant) is a

valid parameter of f; $f(a)$ is a proof of Q. On the other hand, since $f(a)$ is a proof of Q if a is a random proof of P then $f(x)$, where proof x of P is a variable, is a proof of Q.

Definition 4.2.2 *The redexes of formal theory $\mathcal{IFOLP}$ is as presented on figure 4.1.*

Figure 4.1 The redexes of the $\mathcal{IFOLP}$ theory.

$$\frac{\dfrac{}{\texttt{ideq}(a) : a = a}\ \mathsf{refl} \qquad \Lambda x\ \dfrac{\mathcal{D}_{P(x,x)}}{f(x) : P(x,x)}}{\texttt{idpeel}(\texttt{ideq}(a), f) : P(a,a)}\ \mathsf{subst} = \dfrac{\mathcal{D}_{P(a,a)}}{f(a) : P(a,a)}$$

$$\frac{\dfrac{\begin{array}{cc}\mathcal{D}_P & \mathcal{D}_Q \\ p : P & q : Q\end{array}}{\texttt{<}p,q\texttt{>} : P \wedge Q}\ \wedge\mathsf{I}}{\texttt{fst(<}p,q\texttt{>)} : P}\ \wedge\mathsf{E}_1 = \dfrac{\mathcal{D}_P}{p : P}$$

$$\frac{\dfrac{\begin{array}{cc}\mathcal{D}_P & \mathcal{D}_Q \\ p : P & q : Q\end{array}}{\texttt{<}p,q\texttt{>} : P \wedge Q}\ \wedge\mathsf{I}}{\texttt{snd(<}p,q\texttt{>)} : Q}\ \wedge\mathsf{E}_2 = \dfrac{\mathcal{D}_Q}{q : Q}$$

$$\frac{\dfrac{\dfrac{\mathcal{D}_Q}{p : Q}}{\texttt{inr}(p) : P \vee Q}\ \vee\mathsf{I}_2 \qquad \Lambda x\ \dfrac{[x : P]}{\mathcal{D}_{R,P}} \qquad \Lambda y\ \dfrac{[y : Q]}{\mathcal{D}_{R,Q}}}{\texttt{when}(\texttt{inr}(p), q_l, q_r) : R}\ \vee\mathsf{E}_1 = \dfrac{\dfrac{\mathcal{D}_Q}{p : Q} \qquad \Lambda y\ \dfrac{[y : Q]}{\mathcal{D}_{R,Q}}}{q_r(p) : R}$$

$$\frac{\dfrac{\dfrac{\mathcal{D}_Q}{p : P}}{\texttt{inl}(p) : P \vee Q}\ \vee\mathsf{I}_1 \qquad \Lambda x\ \dfrac{[x : P]}{\mathcal{D}_{R,P}} \qquad \Lambda y\ \dfrac{[y : Q]}{\mathcal{D}_{R,Q}}}{\texttt{when}(\texttt{inl}(p), q_l, q_r) : R}\ \vee\mathsf{E}_2 = \dfrac{\dfrac{\mathcal{D}_P}{a : P} \qquad \Lambda x\ \dfrac{[x : P]}{\mathcal{D}_{R,Q}}}{q_l(p) : R}$$

$$\frac{\dfrac{\mathcal{D}_P}{q : P} \qquad \dfrac{\Lambda x\ \dfrac{[x : P]}{\mathcal{D}_Q}}{\texttt{lam } x.\ p(x) : P \to Q}\ \to\mathsf{I}}{(\texttt{lam } x.\ p(x))\texttt{`}q : Q}\ \to\mathsf{E} = \dfrac{\dfrac{\mathcal{D}_P}{q : P} \qquad \Lambda x\ \dfrac{[x : P]}{\mathcal{D}_Q}}{p(q) : Q}$$

$$\frac{\Lambda x\dfrac{\dfrac{\mathcal{D}_{P(x)}}{p(x) : P(x)}}{\texttt{all } x.\ p(x) : \forall y.P(y)}\ \forall\mathsf{I} \qquad \Lambda y, z\ \dfrac{[y(z) : P(z)]}{\mathcal{D}_{R(z)}}}{(\texttt{all } x.\ p(x))\texttt{\^{}}a : R(a)}\ \forall\mathsf{E} = \Lambda x\dfrac{\dfrac{\mathcal{D}_{P(x)}}{p(x) : P(x)}}{p(a) : P(a)}$$

$$\frac{\dfrac{\dfrac{\mathcal{D}_{P(a)}}{p : P(a)}}{[a,p] : \exists y.P(y)}\ \exists\mathsf{I} \qquad \Lambda x, u\ \dfrac{[u : P(x)]}{\mathcal{D}_R}}{\texttt{xsplit}([a,p], f) : R}\ \exists\mathsf{E} = \dfrac{\dfrac{\mathcal{D}_{P(a)}}{p : P(a)} \qquad \Lambda u, y\ \dfrac{[u : P(y)]}{\mathcal{D}_R}}{f(a,p) : R}$$

Most of the proofs are parametric. To represent parametric proofs, the parameters of proofs of propositions that are put together shall be unified. I.e., if a is a proof of P and $f(y)$ is a proof of Q, with a hypothesis that y is a proof of P, if proofs of P and Q forms a redex that is reduced then a and y shall be unified, hence $f(a)$ is a proof of Q.

The redex for the ∀ connective is

$$\mathcal{D} = \Lambda z. \begin{array}{c} \mathcal{D}_{P(x)} \\ f(x) : P(x) \\ \mathcal{D}_{R(z)} \\ f(z) : R(z) \end{array}$$

Obviously $f(x)$ proves $P(x)$ for all constants x and thereby $f(z)$ also proves $P(z)$ for all variables z. Since f then both proves $P(z)$ and $R(z)$ for all variables z then P and R have to be equal. Then deduction $\mathcal{D}_{R(z)}$ is simply just the empty deduction which gives the redex for the ∀ connective on figure 4.1.

4.3 Reduction in Predicate Typed Extended λ-Calculus

The λ-calculus is a calculus for reasoning about dynamic composition of objects with variables and two "structors", a constructor "λ" for implication and a destructor " " (space) for elimination.

Definition 4.3.1 *For a given term of the λ-calculus*[1] e' *with* x *as free variable,* $\lambda x.e'(x)$ *is the abstraction of* x *over* e. *For given terms* e'' *and* e''' *an application* e'' e''' *is* e'' *applied to* e'''. *The semantics of an application is defined by* β *reduction which is that a redex* $(\lambda x.e(x))$ e'' *is equivalent to a term* $e(x)[x/e''] = e(e'')$, *written as* $\lambda x.e(x)$ $e'' =_\beta e(e'')$.

All redexes in the λ-calculus are of the form $(\lambda x.e(x))$ e''.

Since the λ-calculus is the basis of the predicate typed extended λ-calculus the concept of β reduction can be transferred to the extended λ-calculus. The structors of the calculus can be denoted as constructors and destructors based on the action on the type of the term composed by the structor, such that constructors makes a term constructing a type and destructors makes a term deconstructing a type.

4.3.1 Redex

With basis in the description of the extended λ-calculus in section 3.2.3, reducts coming out of redexes of the calculus are constructed by unifying types of the elements of each redex construct. The reduct is the one of the redex components with the wanted type and parameters of the redex component substituted with other redex components having same type as the parameters.

Definition 4.3.2 *Redexes of the predicate typed extended λ-calculus are defined as on figure 4.2.*

As an example,

$$\texttt{xsplit} : \exists\hat{\beta}.\alpha(\hat{\beta}) \to (\hat{\delta} \to \alpha(\hat{\delta}) \to \gamma) \to \gamma \quad (4.1)$$

$$\texttt{[}\cdot\texttt{,}\cdot\texttt{]} : \hat{\delta} \to \alpha(\hat{\delta}) \to \exists\hat{\beta}.\alpha(\hat{\beta}) \quad (4.2)$$

$$p : \alpha(\hat{\delta}) \quad (4.3)$$

$$\texttt{[}\hat{\delta}\texttt{,}p\texttt{]} : \exists\hat{\beta}.\alpha(\hat{\beta}) \quad (4.4)$$

$$f : \hat{\delta} \to \alpha(\hat{\delta}) \to \gamma \quad (4.5)$$

$$f(\hat{\delta},p) : \gamma \quad (4.6)$$

$$\texttt{xsplit(}\texttt{[}\hat{\delta}\texttt{,}p\texttt{]}\texttt{,}f\texttt{)} : \gamma \quad (4.7)$$

[1] For theoretical reasoning about terms of the λ-calculus function symbols and abbreviations are introduced. These are not a part of the λ-calculus.

Figure 4.2 Redexes in the predicate typed extended λ calculus.

Constructor	*Destructor*	*Redex*	*Reduct*
lam	‘	(lam x. $f(x)$) ‘ e	$f(e)$
<·,·>	fst	fst(<a,b>)	a
<·,·>	snd	snd(<a,b>)	b
all	^	(all $\hat{\alpha}$. $f(\hat{\alpha})$)^$\hat{\beta}$	$f(\hat{\beta})$
[·,·]	xsplit	xsplit([$\hat{\alpha}$,p],f)	$f(\hat{\alpha},p)$
inl	when	when(inl(a),f,g)	$f(a)$
inr	when	when(inr(b),f,g)	$g(b)$

Types of the constructs are deduced such that

- applying (4.2) on $\hat{\delta}$ and (4.3) gives (4.4),
- applying (4.5) on $\hat{\delta}$ and (4.3) gives (4.6),
- applying (4.1) on (4.4) and (4.5) gives (4.7).

Eventually the terms (4.6) and (4.7) have the same types. The one term is constructed of a subset of the other term's components whereby the two terms are denotationally equal,

$$\texttt{xsplit([}\hat{\delta}\texttt{,}p\texttt{],}f\texttt{)} = f(\hat{\delta},p)$$

A redex with a destructor d contains one *applicant*, which is the term formed by the constructor c belonging to d, cf. figure 4.2, and the *arguments* of the destructor.

4.3.2 Proof Forms of Extended λ-Calculus Expressions

Proof forms of the extended λ-calculus follow the scheme given for proof normal forms. An expression in normal form is free of redexes. An expression in head normal form is constructors followed by a destructor with variables as arguments. An expression is in weak head normal form if the expression is a variable or the outermost structor is a constructor or a destructor applied on variables.

Normal Form

The normal form of an expression is not of much use in program evaluation. Program evaluation takes place when one program term is combined with another program term, which creates a redex. When such two combined terms are in normal form, the result is not on normal form. Therefore reducing a program to normal form may include lot of waste work.

Weak Head Normal Form

Weak head normal form is important in program evaluation. In a destruction the applicant must be in weak head normal form, else it is not possible to have a redex that can be reduced. A term in weak head normal form has therefore got all the reduction steps performed that has to be performed if the term should take part in a redex.

Weak head normal forms are used in most functional programming languages since it can be used to implement both lazy and eager evaluation. Lazy evaluation is in an application to reduce the applicant to weak head normal form and reducing the hereby constructed redex to a reduct that again is lazily evaluated. Eager evaluation is to reduce the applicant and then the arguments to weak head normal form and then reduce the hereby constructed redex.

Lazy evaluation is often denoted "call-by-name"[2] and eager evaluation is often denoted "call-by-value".

4.4 Proofs *are* Programs

To state the Curry-Howard isomorphism between first order logic proofs represented by the formal theory $\mathcal{IFOLP}$ and the predicate typed extended λ-calculus a number of connections must be constructed, namely:

42₆₃ terms in both λ-calculus and $\mathcal{IFOLP}$ must be programs, as stated in sections 3.2.4 and 3.2.3,

43₆₃ terms in both λ-calculus and $\mathcal{IFOLP}$ must be syntactically equal as stated in section 3.2.4,

44₆₃ The isomorphism shall preserve structure under reduction. By comparing the pairs of redexes and reducts in the extended λ-calculus and $\mathcal{IFOLP}$ as stated in figures 4.2 and 4.1, it is clear that the sets of pairs are exactly equal.

Hereby the Curry-Howard isomorphism is stated, whereby proofs in theory $\mathcal{IFOLP}$ are programs in the extended predicate typed λ-calculus.

By the Curry-Howard isomorphism it is possible to reason about proofs as if they were programs. This means that proofs can be evaluated as if they were programs, whereby there exist λ-calculus methods to reduce proofs to normal forms, or to reduce proofs to weak head normal form.

4.5 First Order Reduction Logic

To be able to use proof reduction for first order logic proofs, a logic representing redexes with their reducts shall be created. Such a logic must include rules for first order logic and rules describing redexes and their reducts. This sort of rules are called reduction rules.

Definition 4.5.1 *A connective* $\longmapsto$ *defining the connection between a redex and a reduct is defined as*

$$e \longmapsto e'$$

where e *is a redex that reduces to reduct* e'*.*

4.5.1 Reduction Rules in Formal Theories

Based on $\mathcal{IFOLP}$ and the connective $\longmapsto$ a theory for equality between proofs can be created. Logical rules for such a theory are on figure 4.1.

[2] Due to program explosion call-by-name evaluation is often replaced with call-by-reference evaluation which is equal to graph reduction.

A structural inference rule shall be added to state that a proof by connective $\longmapsto$ reduces in one or more steps to another proof.

Definition 4.5.2 *Transitivity for proof reduction is defined as inference rule*

$$\frac{\mathcal{S} \longmapsto \mathcal{T} \quad \mathcal{T} \longmapsto \mathcal{U}}{\mathcal{S} \longmapsto \mathcal{U}} \tag{4.8}$$

Well-formed formulas of a theory for reduction of $\mathcal{IFOLP}$ proofs are of the form $\mathcal{S} \overset{\vdash}{\sim} \mathcal{T}$ where $\mathcal{S}$ and $\mathcal{T}$ are proofs for propositions where the logic parts are equal, based on the same hypotheses, such that $\mathcal{S}$ and $\mathcal{T}$ prove the same $\mathcal{IFOL}$ proof.

Reduction of proofs in a natural deductive theory is more structured than reduction in a Hilbert-style theory since inference rules give structure. A formal theory taking basis in the reduction rules in figure 4.1 and rule (4.8) is mostly axiomatic, ony rule (4.8) is an inference rule. Therefore such a theory does not have as much structure as a formal theory based on inference rules.

4.5.2 Formal Theory for Proof Reduction

Structure for reasoning about redexes may be incorporated in a theory for proof reduction by incorporating structures in well-formed formulas.

A linearization of a proof can be reduced by removing redexes therein. This may be usable in a formal theory incorporating proof reduction in inference rules.

Structural Reasoning in Proof Reduction

Reduction rules as inference rules must follow a certain scheme, i.e. as defined below.

Definition 4.5.3 *A scheme for reduction rules can be defined as*

$$\frac{p_1 \longmapsto p_1' : P_1 \quad \cdots \quad p_n \longmapsto p_n' : P_n}{q \longmapsto q' : Q} \tag{4.9}$$

Proofs in premise parts of a reduction rule must be parts of the conclusion part proofs. The proofs must be grouped such that left parts of premise proofs must be parts of the left conclusion part proof and right parts of premise proofs must be parts of the right conclusion part. Hence q and q' are formed as

$$q = d(p_1, \ldots, p_n) \qquad q' = d'(p_1', \ldots, p_n')$$

where q is the original proof of Q and q' is the reduced proof of Q, as well $p_1, \ldots, p_n$ are original proofs of $P_1, \ldots, P_n$ and $p_1', \ldots, p_n'$ are reduced proofs of $P_1, \ldots, P_n$.

It is a question whether or not q' by an inference rule can be reduced to the same proof normal form as the premises are on. To reason about this, terms and examples from the λ-calculus are used.

Let the inference rule for reduction[3] be

$$\frac{\Lambda z.(f(z)\longmapsto g(z)) : \sigma \to \tau \qquad c\longmapsto d : \sigma}{(\lambda z.f(z))\ c\longmapsto g(d) : \tau}$$

such that the requirement is that if $\lambda z.g(z)$ and d is in a certain form then $g(d)$ is in the same form.

Let the desired proof form be normal form. Let $q = (\lambda x.(x(\lambda y.y)))(\lambda x.x)$. Apply the above inference rule on this term. Then $f(z) = z(\lambda y.y)$ and $\lambda z.f(z)$ are in normal form which $c = \lambda x.x$ also is. But $g(d) = (\lambda x.x)(\lambda x.x)$ is not in normal form. Also if the desired normal form is weak head normal form the result is not necessarily in weak head normal form. The above example shows that the scheme (4.9) for reduction rules does not ensure that the result of a reduction is of the desired form if the premises are on the desired form. Transitivity of proof reduction is then needed to produce the desired form. Transitivity can either be formulated with a specific inference rule as rule (4.8) or incorporated in each inference rule where it is necessary such that inference rules follow a scheme much like (4.9).

Definition 4.5.4 *A scheme for proof reduction rules that ensures that proofs on the right side of connective $\longmapsto$ are in the desired proof normal form, if all other reduction rules not following the scheme also ensure that proofs on the right side of connective $\longmapsto$ are in the desired proof normal form, can be defined as*

$$\frac{p_1\longmapsto p_1' : P_1 \quad \cdots \quad p_n\longmapsto p_n' : P_n \quad d'(p_1', \ldots, p_n')\longmapsto q' : Q}{d(p_1, \ldots, p_n)\longmapsto q' : Q} \tag{4.10}$$

where $d'(p_1', \ldots, p_n')$ is the reduct corresponding to redex $d(p_1', \ldots, p_n')$.

A problem with a theory for proof reduction may be that there is a redex in a proof but the root of the proof is not part of the redex. Then reduction rules for reducing redexes will not be able to reduce the redex. With only reduction rules is it not possible to reach the redex. Therefore a context rule may be needed.

Definition 4.5.5 *A context rule in a theory for proof reduction is defined as*

$$\frac{e\longmapsto e' : P}{C[e]\longmapsto C[e'] : R} \tag{4.11}$$

For weak head normal form and normal form it is possible to state that a variable is in the desired form.

Definition 4.5.6 *A variable is in normal form and weak head normal form. A variable reduces to itself if the variable proves a given proposition, and reduction is introduced,*

$$\frac{x : P}{x\longmapsto x : P}\ \longmapsto\! I \tag{4.12}$$

A theory for proof reduction of $\mathcal{IFOLP}$ theorems must only accept *reduced* $\mathcal{IFOLP}$ propositions as theorems. If a proof reduced to the desired form p for a proposition P must be created, an arbitrary proof a for proposition P must be created and then reduced to obtain p, which may be called *reduction elimination.*

[3]The inference rule could also be formed in another way such that f and c are not reduced to normal form before creating the reduct (as with normal β reduction of the redex), but then the parts of the reduct would not be reduced to the desired form before the inference rule is applied.

Definition 4.5.7 *Reduction elimination is defined as*

$$\frac{a : P \qquad a \longmapsto p : P}{p : P} \longmapsto E \tag{4.13}$$

If rule $\longmapsto$E is the last rule in a proof tree for $p : P$ it is ensured that p is in the desired proof form.

Definition 4.5.8 *Rules (4.11, 4.12, 4.13) are the structural rules for reduction.*

If the desired proof form is just that the proof is a proof, the structural rules are enough for a reduction theory.

Logical Reasoning in Proof Reduction

If a reduction theory must reduce proof terms the theory must include some *logical rules*. Due to different reduction strategies, the semantics of logical rules differs depending on what reduction strategy is used. For some reduction strategies it is enough that inference rules for reduction implements transitivity, other strategies require that all subproofs of a proof are brought to a certain form before the proof is reduced.

Before turning to construct a logic implementing a certain reduction strategy different reduction strategies may be analysed.

4.6 Reduction Strategies

Strategies for reduction decide how a given proof or program is reduced. Since reduction strategies of the λ-calculus have been studied deeply these strategies may be basis for a discussion on how to reduce proofs. These strategies are for program reduction and not constructed for proof reduction. But as section 4.4 shows proofs can be programs.

4.6.1 Leftmost-Outermost Reduction Strategy

A strategy for reduction of terms of the λ-calculus is the leftmost-outermost reduction strategy. The name of the strategy implies the actions of the reduction strategy which simply is repeatedly to reduce the redex nearest the root of the λ term regarded as a syntax tree; if more redexes are in the same distance of the root the leftmost of the redexes in the λ term (regarded as a program text) shall be reduced.

As an example the term

$$\lambda z.((\underwave{z((\lambda x.x)\underline{((\lambda x.x)(\lambda x.x))})})\underline{(z((\lambda x.x)\underline{((\lambda x.x)(\lambda x.x))}))})$$

has the underlined redexes and the wavy underlined redex as the leftmost-outermost redex.

For bringing a proof term on normal form the strategy may be useful. An important property of the leftmost-outermost reduction strategy is that if a λ term can be reduced to normal form then the leftmost-outermost reduction strategy will bring it to normal form [17]. An inconvenience of

the leftmost-outermost reduction strategy is that the number of redexes in a λ term can explode, i.e.

$$(\lambda x.xxxx)((\lambda x.x)(\lambda x.x)) \tag{4.14}$$

reduces to

$$((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))$$

since the redex in the argument of term (4.14) is not reduced before the application is reduced.

4.6.2 Leftmost-Innermost Reduction Strategy

Another strategy for reduction of terms of the λ-calculus is the leftmost-innermost reduction strategy. A leftmost-innermost redex is a redex in a λ term that is innermost, which is that the redex viewed as a syntax tree has no redexes in its subtrees, and also leftmost. As an example the term

$$\lambda z.((z((\lambda x.x)\underline{\underbrace{((\lambda x.x)(\lambda x.x))}}))\underline{(z\underline{((\lambda x.x)\underline{((\lambda x.x)(\lambda x.x))})})})$$

has the underlined redexes, and the wavy underlined redex is the leftmost-innermost redex.

Definition 4.6.1 *The leftmost-innermost reduction strategy is to repeatedly reduce leftmost-innermost redexes until all redexes are reduced.*

The leftmost-innermost strategy has the property that both applicants and arguments of a redex are reduced before the redex is reduced, whereby proof explosion can be partly prevented (but not totally removed). As an example (4.14) reduces to

$$(\lambda x.xxxx)(\lambda x.x)$$

which again reduces to

$$(\lambda x.x)(\lambda x.x)(\lambda x.x)(\lambda x.x)$$

A disadvantage of the leftmost-innermost reduction strategy is that not all terms with a normal form can be reduced to normal form. Reduction of a term

$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$$

has as leftmost-innermost redex $((\lambda x.xx)(\lambda x.xx))$ which reduces to itself even the term has normal form $\lambda y.y$. Notice that the term $((\lambda x.xx)(\lambda x.xx))$ is *not* typable in the simply typed λ-calculus.

4.6.3 Lazy Evaluation

A strategy for program reduction is lazy evaluation. The idea of lazy evaluation is to only evaluate terms that really is used.

Definition 4.6.2 *Reducing a λ term using lazy evaluation is to reduce the leftmost-outermost redex of the λ term until the term is in weak head normal form.*

Since reduction stops when the term is in weak head normal form only redexes outside abstractions are evaluated.

The lazy evaluation reduction strategy is important in implementations of functional languages. As an example the language Miranda is lazy evaluated.

4.6.4 Eager Evaluation

Another strategy for program reduction is eager evaluation. The idea of eager evaluation is to only evaluate terms that are not abstracted over. This also include arguments of applicants whereby arguments are evaluated only once (even if the arguments are not used).

Definition 4.6.3 *Reducing a λ term using eager evaluation is to reduce the leftmost-outermost redex of the λ term until the term is in weak head normal form such that arguments of a redex is eagerly evaluated before the redex is reduced.*

The eager evaluation reduction strategy is important in implementations of functional languages. As an example the language ML is eager evaluated.

An advantage of eagerly evaluated languages is that arguments of redexes always are evaluated exactly once. A downside of eagerly evaluated languages is that it is difficult to implement a data structure like infinite lists since it requires that the list is made up of abstractions.

4.6.5 Reduction Strategies and Inference Trees

The sections below discuss reduction strategies and inference trees. The discussion would take too much space if it regards the extended λ-calculus, therefore they discuss the simply typed λ-calculus. As well terms to be reduced are generally supposed to be typable.

Representing the Leftmost-Outermost Reduction Strategy

For a term $e_1\ e_2$ where the principal structor of e_1 is not a constructor, the leftmost-outermost redex of the term is placed in either e_1, if there are any redexes in e_1, or else in e_2 if there are any redexes at all in the term. Reducing such a term to normal form using the leftmost-outermost reduction strategy requires[4] a conditional in an inference rule, since e_2 must only be reduced if there are no redexes in e_1. Therefore the leftmost-outermost reduction strategy cannot be used with inference trees.

Representing the Leftmost-Innermost Reduction Strategy

In reducing a term $e_1\ e_2$ using the leftmost-innermost reduction strategy all redexes in both e_1 and e_2 will become leftmost-innermost redexes one by one. Redexes in e_1 and e_2 are reduced to e_1' respectively e_2' before a possible redex $e_1'\ e_2'$ is reduced. Following the structural guidelines in section 4.5.2 an inference rule for β reduction can be created as

$$\frac{\Lambda z.f(z) \longmapsto g(z) : P \qquad a \longmapsto a' : Q \qquad g(a') \longmapsto r : P}{(\lambda x.f(x))\ a \longmapsto r : P}$$

The structural guidelines also prescribes a context rule (4.11) for reducing inside a context. This context rule cannot be used for this reduction strategy since the context rule gives a reduction $((\lambda x.x)(\lambda x.x))\ t \longmapsto (\lambda x.x)\ t$ because there is no transitivity in the context rule. Adding transitivity to the context rule cannot reduce the term $\lambda x.x$ since it creates a looping proof of $\lambda x.x \longmapsto \ldots$. Defining a new proof connective for which it is ensured that the arguments on which the left hand principal structor is applied is in the desired form, can solve the problem.

[4] I cannot see how to avoid requiring a conditional. Probably it is possible.

Definition 4.6.4 *Let d be a destructor. A connective $\longmapsto_!$ denotes in a judgement*

$$d(p_1, \ldots, p_n) \longmapsto_! p : P$$

that $p_1, \ldots, p_n$ is on the desired proof form.

The connective $\longmapsto_!$ can then be used for adding transitivity to reduction rules. A context rule with restricted transitivity

$$\frac{e \longmapsto e' : Q \to P \qquad a \longmapsto a' : Q \qquad e'\ a' \longmapsto_! r : P}{e\ a \longmapsto r : P} \tag{4.15}$$

ensures that the contexts e and a of expression $e\ a$ are reduced to the desired form before the possible redex $e'\ a'$ is reduced. Since contexts e' and a' are in the desired form expression $e'\ a'$ is either a redex or in the desired form. Therefore with inference rules (4.11, 4.15) and

$$\frac{f(a) \longmapsto r : P}{(\lambda x.f(x))\ a \longmapsto_! r : P} \tag{4.16}$$

$$\frac{}{e\ a \longmapsto_! e\ a : P} \tag{4.17}$$

all proofs of the simply typed λ-calculus can be reduced. A problem is that it is also possible to prove that a proof is normalized even if it is not normalized. This can be avoided by requiring that

- inference rule (4.17) is only used to resolve a proposition if rule (4.16) cannot resolve the proposition, and
- the context rule may only be used to resolve a proposition if none other rules can resolve the proposition.

If the context rule is used, only one layer of structors may be regarded as the context. Then the context rule can just as well be rewritten to

$$\frac{\Lambda z.e(z) \longmapsto e'(z) : Q}{\lambda x.e(x) \longmapsto \lambda x.e'(x) : P \to Q} \tag{4.18}$$

since rule (4.15) serves all expressions having the destructor as principal connective.

Notice also that the above β reduction rule can be implemented using inference rules (4.15, 4.16, 4.17) as it follows from inference tree

$$\frac{\dfrac{\Lambda z.\ f(z) \longmapsto g(z) : P}{\lambda x.f(x) \longmapsto \lambda x.g(x) : Q \to P}\ (4.18) \qquad a \longmapsto a' : Q \qquad \dfrac{g(a') \longmapsto r : P}{(\lambda x.g(x))\ a' \longmapsto_! r : P}\ (4.16)}{(\lambda x.f(x))\ a \longmapsto a : P}\ (4.15)$$

Then the above set of inference rules (4.15, 4.16, 4.17, 4.18) and rule $\longmapsto$ I represents the leftmost-innermost reduction strategy.

Representing the Lazy Evaluation Strategy

A term $e_1\ e_2$ is in weak head normal form if the term is not a redex and e_1 is in weak head normal form. Therefore it is only necessary to reduce e_1 to a weak head normal form e_1' and reduce $e_1'\ e_2$ if it is a redex. This gives inference rule

$$\frac{e \longmapsto e' : Q \to P \qquad e'\ a \longmapsto_! r : P}{e\ a \longmapsto r : P} \tag{4.19}$$

and above rules (4.16, 4.17) for connective $|\longrightarrow_!$. Since lazy evaluation means that abstractions are don't-care-stuff unless they are applied on something there must not be a context rule for the λ constructor but a rule accepting an abstraction as being normalized,

$$\frac{}{\lambda x.e(x)|\longrightarrow \lambda x.e(x) : P \to Q} \tag{4.20}$$

The set of inference rules (4.16, 4.17, 4.19, 4.20) represents the leftmost-innermost reduction strategy.

Notice that even leftmost-outermost reduction cannot be implemented using inference trees it is possible to represent lazy evaluation. Because of rule (4.20) it is not possible to reduce other redexes than outermost redexes. By rule (4.19) redexes in arguments are not reduced.

Representing the Eager Evaluation Strategy

Expressed using inference rules eager evaluation is expressed by (4.15, 4.16, 4.17, 4.20), with the same prerequisites on order of application of the rules as for leftmost-innermost reduction.

Notice that even leftmost-outermost reduction cannot be implemented using inference trees it is possible to represent eager evaluation. Because of rule (4.20) it is not possible to reduce other redexes than outermost redexes in applicants and arguments.

Generalising Reduction Strategies

By looking at the discussions of representation of reduction strategies it is obvious that constructors (in the discussions the only constructor is "λ") as principal terms of proofs gives reason to the following sorts of inference rules:

45$_{70}$ Context rules for reducing to normal form, as in rule (4.18),

46$_{70}$ Acceptance rules for reducing to weak head normal form, as in rule (4.20).

This may then be the case for all other constructors in a reduction theory. As well it is obvious that destructors (in the discussions the only destructor is " ", application) as principal terms of proofs gives reason to the following sorts of inference rules:

47$_{70}$ A reduction preparation rule for possible redexes created with the destructor. A reduction preparation rule ensures that the parameters of the destructor have the desired form before reduction. The premises of the reduction preparation rule depend on the chosen reduction strategy. For inside-term reducing strategies the inside-term redexes are reduced before the redex is reduced as in rule (4.15). For strategies not reducing inside the term only the applicant is reduced before the redex is reduced as in rule (4.19). The reduction rules also include restricted proof transitivity on the reduct.

48$_{70}$ Reduct reduction rules where the destructor's arguments are on the desired form to construct a redex. There are two kinds of reduct reduction rules. One kind matches a redex and reduces the redex including transitivity such that the reduct can be further reduced as in rule (4.16). Another kind accepts the left side and right side of the reduct reduction connective as being equal, as in rule (4.17).

Looking at the above set of inference rules makes it clear that theories for leftmost-innermost reduction and eager evaluation are almost equal (but the set of inference rules does not state anything about whether the reduction outcomes are almost equal). Then reasoning about a linearized proof as a program or as a logic structure are almost equal. These two reasoning methods share reduction preparation and reduct reduction rules (which is points 47_{70}, 48_{70}) but differ in context rules for constructors (which is point 45_{70} for logic structure reasoning and point 46_{70} for program reasoning).

4.7 Proof Reduction Theories for First Order Logic with Proofs

With

- a general discussion on why to represent a theory for proof reduction as a natural deductive style theory,
- representing a reduction proposition as a term $a \longmapsto p : P$ stating that a proof a of P reduces to a proof p of P,
- a discussion on how to secure that a really is a proof of P

in section 4.5.2, the structure of a theory for reduction of natural deductive first order proofs is decided. Section 4.6.5 states how the logic of such a reduction strategy must be stated.

A logic for reasoning about reduction can be logically stated for either proof reduction or program evaluation.

The context rules presented in section 4.6.5 for constructor "λ" and destructor " " can be generalised to serve as schemes for all constructors and destructors of proofs for propositions in a theory for first order logic. The reduction rule can not serve as scheme for all other reduction rules since the outcomes of redexes depends of the semantic of the destructor.

The $\mathcal{IFOLP}$ theory can be used as theory for which proofs shall be reduced.

4.7.1 Reduction Rules

Reduction rules must be constructed for connectives $\rightarrow, \forall, \vee, \wedge, \exists$. The schemes presented as rules (4.16, 4.17) are used in constructing reduction rules. There is a difference between the $\mathcal{IFOLP}$ reduction rules and the reduction rules of the extended predicate typed λ calculus. Extra constructs of the $\mathcal{IFOLP}$ redexes can give delayed unifying of logic terms but this gives no denotational difference. Using the $\mathcal{IFOLP}$ reduction rules give reduction rules for a reduction theory as presented on figure 4.3. Notice that all inference rules of form (4.17) can be implemented using one single rule since all the rules follow the scheme that the left hand side equals the right hand side of the reduct connective.

4.7.2 Context Rules

A general context rule is formulated as rule (4.11). For a logic that requires that the rules are applied using some intelligence this context rule is well suited. It allows reduction wherever it can be applied in a proof, and with another rule for reduction transitivity all redexes could be found.

Figure 4.3 Reduction rules for reducing linearized first order logic proofs.

$$\frac{f(e)\longmapsto r : P}{\texttt{lam}\ x.\ f(x)\ \texttt{`}\ e\longmapsto_! r : P}\ \to\mathsf{R} \qquad \frac{f(\hat{\alpha},p)\longmapsto r : P}{\texttt{xsplit([}\hat{\alpha}\texttt{,}p\texttt{],}f\texttt{)}\longmapsto_! r : P}\ \exists\mathsf{R}$$

$$\frac{a\longmapsto r : P}{\texttt{fst(<}a\texttt{,}b\texttt{>)}\longmapsto_! r : P}\ \wedge\mathsf{R}_1 \qquad \frac{f(a)\longmapsto r : P}{\texttt{when(inl(}a\texttt{),}f\texttt{,}g\texttt{)}\longmapsto_! r : P}\ \vee\mathsf{R}_1$$

$$\frac{b\longmapsto r : P}{\texttt{snd(<}a\texttt{,}b\texttt{>)}\longmapsto_! r : P}\ \wedge\mathsf{R}_2 \qquad \frac{g(b)\longmapsto r : P}{\texttt{when(inr(}b\texttt{),}f\texttt{,}g\texttt{)}\longmapsto_! r : P}\ \vee\mathsf{R}_2$$

$$\frac{\begin{array}{cc} & [y : R(\hat{\alpha})] \\ \Lambda(\hat{\alpha}) & \vdots \\ f(\hat{\beta})\longmapsto r : R(\hat{\beta}) & y : P(\hat{\alpha}) \end{array}}{\texttt{all}\ \hat{\alpha}.\ f(\hat{\alpha})\texttt{\^{}}\hat{\beta}\longmapsto_! r : P(\hat{\beta})}\ \forall\mathsf{R} \qquad \frac{}{e\longmapsto_! e : P}\ \mathsf{IsNorm}$$

Moreover the rule fully survives a crash test: every reduction of a proof not in the desired form *can* crash using this context rule. Therefore the context rule must be specialised to fit each structor and reduction method, as well as transitivity may only be applied where redexes are created.

Since context rules for destructors are equal in the leftmost-innermost reduction strategy and the eager evaluation strategy these two reduction strategies may be chosen for proof reduction and program evaluation.

Definition 4.7.1 *Context rules for destructors of the extended λ-calculus are as given on figure 4.4.*

Context rules for an abstraction in a proof term shall make the term ready for reducing under the abstracting constructor. Therefore a context rule for a proof term constructor shall reduce the terms on which the constructor is applied.

Definition 4.7.2 *Context rules for constructors of the extended λ-calculus regarded as proof terms are as given on figure 4.5.*

As program terms abstractions are don't-care-stuff. Hereby abstractions must be accepted as being normalized. Abstractions must all be accepted as being normalized, therefore the context rules for constructors are all of the form

$$\frac{}{p\longmapsto p : P}$$

Given that context rules for constructors are not used to resolve a proposition unless no other reduction rule can be used, it is enough to state the context rule for program abstractions as above.

Also it is important to notice that the context rule above includes the $\longmapsto\mathsf{I}$ rule since the $\longmapsto\mathsf{I}$ rule is the above context rule restricted to proof variables. Therefore the context rule for constructors and term variables may be merged to one rule introducing reduction.

Definition 4.7.3 *The reduction introducing rule $\longmapsto\mathsf{I}$ for program evaluation is defined as*

$$\frac{}{p\longmapsto p : P}\ \longmapsto\mathsf{I} \tag{4.21}$$

and it must only be applied on abstractions or variables.

Figure 4.4 Context rules for destructors used for reduction strategies that requires that innermost redexes are reduced before the actual redex is reduced.

$$\frac{e_1 \longmapsto e_1' : Q \to P \qquad e_2 \longmapsto e_2' : Q \qquad e_1' \ \text{`} \ e_2' \longmapsto_! r : P}{e_1 \ \text{`} \ e_2 \longmapsto r : P} \to \mathsf{CE}$$

$$\frac{e \longmapsto e' : P \wedge Q \qquad \mathtt{fst}(e') \longmapsto_! r : P}{\mathtt{fst}(e) \longmapsto r : P} \wedge\mathsf{CE}_1$$

$$\frac{e \longmapsto e' : P \wedge Q \qquad \mathtt{snd}(e') \longmapsto_! r : Q}{\mathtt{snd}(e) \longmapsto r : Q} \wedge\mathsf{CE}_2$$

$$\frac{f \longmapsto g : \forall\hat{\beta}.R(\hat{\beta}) \qquad g\,\hat{}\,\hat{\alpha} \longmapsto_! r : R(\hat{\alpha}) \qquad \begin{array}{c}[y : R(\hat{\alpha})] \\ \Lambda(\hat{\alpha}) \ \vdots \\ y : P(\hat{\alpha})\end{array}}{f\,\hat{}\,\hat{\alpha} \longmapsto r : P(\hat{\alpha})} \forall\mathsf{CE}$$

$$\frac{p \longmapsto q : \exists\hat{\alpha}.P(\hat{\alpha}) \qquad \begin{array}{c}[u : P(x)] \\ \Lambda x\ u \ \vdots \\ f(x,u) \longmapsto g(x,u) : R\end{array} \qquad \mathtt{xsplit}(q,g) \longmapsto_! r : R}{\mathtt{xsplit}(p,f) \longmapsto r : R} \exists\mathsf{CE}$$

$$\frac{p \longmapsto q : P \vee Q \qquad \begin{array}{c}[x : P] \\ \Lambda x \ \vdots \\ f(x) \longmapsto f'(x) : R\end{array} \qquad \begin{array}{c}[x : Q] \\ \Lambda x \ \vdots \\ g(x) \longmapsto g'(x) : R\end{array} \qquad \mathtt{when}(q,f',g') \longmapsto_! r : R}{\mathtt{when}(p,f,g) \longmapsto r : R} \vee\mathsf{CE}$$

Figure 4.5 Constructor context rules for reducing proof terms.

$$\frac{a \longmapsto a' : P \qquad b \longmapsto b' : Q}{\mathtt{<}a,b\mathtt{>} \longmapsto \mathtt{<}a',b'\mathtt{>} : P \wedge Q} \wedge\mathsf{CI}$$

$$\frac{a \longmapsto a' : P}{\mathtt{inl}(a) \longmapsto \mathtt{inl}(a') : P \vee Q} \vee\mathsf{CI}_1$$

$$\frac{b \longmapsto b' : Q}{\mathtt{inr}(b) \longmapsto \mathtt{inr}(b') : P \vee Q} \vee\mathsf{CI}_2$$

$$\frac{\begin{array}{c}[y : P] \\ \Lambda y \ \vdots \\ f(y) \longmapsto g(y) : Q\end{array}}{\mathtt{lam}\ x\ .f(x) \longmapsto \mathtt{lam}\ x\ .g(x) : P \to Q} \to\mathsf{CI}$$

$$\Lambda y \frac{f(y) \longmapsto g(y) : P(y)}{\mathtt{all}\ x.\ f(x) \longmapsto \mathtt{all}\ x.\ g(x) : \forall z.P(z)} \forall\mathsf{CI}$$

$$\frac{p \longmapsto q : P(x)}{[x\,,p] \longmapsto [x\,,q] : \exists y.P(y)} \exists\mathsf{CI}$$

4.7.3 Formal Theory for Reducing Proofs

The $\mathcal{IFOLP}$ theory presented in section 3.3.2 must be used as basis for a formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$ for proof reduction. The $\mathcal{IFOLP}$ theory include proofs as parts of propositions whereby it is well suited for this purpose. As well proofs must be reduced using the leftmost-innermost reduction strategy. Since $\mathcal{IFOLP}$ also includes a reserved predicate letter for equality reduction of the equality letter must also be included in a theory for reducing proofs.

Definition 4.7.4 *The reduction rule for predicate letter "=" is*

$$\frac{f(\hat{\alpha})\longmapsto r : R(\hat{\alpha},\hat{\alpha}) \qquad \begin{array}{c}[y : R(\hat{\beta},\hat{\beta})]\\ \Lambda y \quad \vdots \\ y : P(\hat{\beta},\hat{\beta})\end{array}}{\texttt{idpeel(ideq(}\hat{\alpha}\texttt{),}f\texttt{)}\longmapsto_! r : P(\hat{\alpha},\hat{\alpha})} \tag{4.22}$$

Definition 4.7.5 *The context rule for destructor* `idpeel` *is*

$$\frac{p\longmapsto q : \hat{\alpha}=\hat{\beta} \qquad \begin{array}{c}[y : P(\hat{\alpha},\hat{\beta})]\\ \Lambda y \quad \vdots \\ y : R\end{array} \qquad \Lambda\hat{\gamma}.f(\hat{\gamma})\longmapsto g(\hat{\gamma}) : P(\hat{\gamma},\hat{\gamma}) \qquad \texttt{idpeel(}q\texttt{,}g\texttt{)}\longmapsto_! r : P(\hat{\alpha},\hat{\beta})}{\texttt{idpeel(}p\texttt{,}f\texttt{)}\longmapsto r : R} \tag{4.23}$$

To make rules (4.22, 4.23) usable for resolving unification of the logic parts of the premise and the conclusion can be delayed.

Definition 4.7.6 *The proof reduction context rule for constructor* `ideq` *is*

$$\frac{}{\texttt{ideq(}\hat{\alpha}\texttt{)}\longmapsto \texttt{ideq(}\hat{\alpha}\texttt{)} : \hat{\alpha}=\hat{\alpha}} \tag{4.24}$$

Definition 4.7.7 *Formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$ is defined as theory $\mathcal{IFOLP}$ extended with the following*

Delimiters $\longmapsto, \longmapsto_!$

Inner propositions *are $\mathcal{IFOLP}$ propositions or composed of a proof part, a delimiter $\longmapsto$ or $\longmapsto_!$, a proof part, a delimiter : and a logic part as $a\longmapsto p : P$ or $a\longmapsto_! p : P$ where a and p are proof parts and P is a logic part.*

Axioms *are $\mathcal{IFOLP}$ axioms, inference rules with no premises on figure 4.3 and rules (4.12), (4.24).*

Inference rules *are $\mathcal{IFOLP}$ inference rules, inference rules with premises of figures 4.3, 4.4, 4.5 and rules (4.13), (4.22), (4.23).*

Propositions *are inner propositions of form $p : P$ proved using only rules of theory $\mathcal{PR}_{\mathcal{IFOLP}}$ such that the following conditions are fulfilled:*

49_{74} the proof of $p : P$ ends in rule (4.13) and this is the only place in the proof where the rule is used,

50_{74} *Rule* IsNorm *of figure 4.3 may only be used if none of the other rules of figure 4.3 can be used,*

51_{74} *rule (4.12) may only be used on proof variables.*

Notice that in the above definition the only propositions are reduced proofs.

Reducing $\mathcal{IFOLP}$ proofs using the above theory $\mathcal{PR}_{\mathcal{IFOLP}}$ then consists of two parts:

- proving a $\mathcal{IFOLP}$ theorem using the $\mathcal{IFOLP}$ theory, and
- reducing the proof part of the proved $\mathcal{IFOLP}$ theorem to a proof part that is reduced using the context and reduction rules.

4.7.4 Formal Theory for Program Evaluation

Proofs can be viewed as programs. Therefore a theory for program evaluation has much in common with a theory for proof reduction. Therefore the ideas behind theory $\mathcal{PR}_{\mathcal{IFOLP}}$ can be used for constructing a theory $\mathcal{PE}_{\mathcal{IFOLP}}$ for program evaluation, with the little difference that instead of using leftmost-innermost reduction eager evaluation shall be used for reducing program terms.

Definition 4.7.8 *A formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$ is defined as theory $\mathcal{IFOLP}$ extended with the following*

Delimiters $\longmapsto, \longmapsto_!$

Inner propositions *are $\mathcal{IFOLP}$ propositions or composed of a proof part, a delimiter $\longmapsto$ or $\longmapsto_!$, a proof part, a delimiter : and a logic part as $a \longmapsto p : P$ or $a \longmapsto_! p : P$ where a and p are proof parts and P is a logic part.*

Axioms *are $\mathcal{IFOLP}$ axioms and inference rules with no premises on figure 4.3 and rule (4.21).*

Inference rules *are $\mathcal{IFOLP}$ inference rules and inference rules of figure 3.2, inference rules with premises of figures 4.3, 4.4 and rules (4.13), (4.22).*

Propositions *are well-formed formulas of form $p : P$ proved using only rules of theory $\mathcal{PR}_{\mathcal{IFOLP}}$ such that the following conditions are fulfilled:*

52_{75} *the proof of $p : P$ ends in rule (4.13) and this is the only place in the proof where the rule is used,*

53_{75} *Rule* IsNorm *of figure 4.3 may only be used if none of the other rules of figure 4.3 can be used,*

54_{75} *Rule (4.21) may only be used if none of the rules of figure 4.4 can be used.*

Notice that in the above definition the only propositions are reduced proofs.

Reducing $\mathcal{IFOLP}$ proofs using the above theory $\mathcal{PE}_{\mathcal{IFOLP}}$ then consists of two parts:

- proving a $\mathcal{IFOLP}$ theorem using the $\mathcal{IFOLP}$ theory, and
- reducing the proof part of the proved $\mathcal{IFOLP}$ theorem to a proof part that is reduced using the context and reduction rules.

4.7.5 Obtaining Proofs in $\mathcal{PR}_{\mathcal{IFOLP}}$ and $\mathcal{PE}_{\mathcal{IFOLP}}$

With the theories $\mathcal{PR}_{\mathcal{IFOLP}}$ and $\mathcal{PE}_{\mathcal{IFOLP}}$ linearized proofs of $\mathcal{IFOL}$ propositions, expressed as $\mathcal{IFOLP}$ propositions, can be reduced either to normal form or to a[5] weak head normal form. Examples of reducing different proofs of $\mathcal{IFOL}$ proposition $P \to P$ with the theories follows.

- Reducing $\texttt{lam}\ x.\ x$ in theory $\mathcal{PR}_{\mathcal{IFOLP}}$:

$$\dfrac{\dfrac{\overset{\alpha}{x : P}}{\texttt{lam}\ x.\ x : P \to P}\to\mathsf{I}^{\alpha \lhd x:P} \qquad \dfrac{\dfrac{\overset{\beta}{x}}{x \longmapsto x : P}\longmapsto\mathsf{I}}{\texttt{lam}\ x.\ x \longmapsto \texttt{lam}\ x.\ x : P \to P}\to\mathsf{CI}^{\beta \lhd x:P}}{\texttt{lam}\ x.\ x \longmapsto \texttt{lam}\ x.\ x : P \to P}\longmapsto\mathsf{E}$$

- Reducing $\texttt{lam}\ x.\ x$ in theory $\mathcal{PE}_{\mathcal{IFOLP}}$:

$$\dfrac{\dfrac{\overset{\alpha}{x : P}}{\texttt{lam}\ x.\ x : P \to P}\to\mathsf{I}^{\alpha \lhd x:P} \qquad \dfrac{}{\texttt{lam}\ x.\ x \longmapsto \texttt{lam}\ x.\ x : P \to P}\mathsf{CI}}{\texttt{lam}\ x.\ x \longmapsto \texttt{lam}\ x.\ x : P \to P}\longmapsto\mathsf{E}$$

- Reducing $\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y$ with hypothesis $\delta \lhd y : P$ in theory $\mathcal{PR}_{\mathcal{IFOLP}}$:

$$\dfrac{\overset{\mathcal{D}_1}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y : P} \qquad \overset{\mathcal{D}_2}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y \longmapsto y : P}}{y : P}\longmapsto\mathsf{E}$$

where

$$\mathcal{D}_1 = \dfrac{\dfrac{\dfrac{\dfrac{\overset{\alpha}{x : P}}{\texttt{lam}\ x.\ x : P \to P}\to\mathsf{I}^{\alpha \lhd x:P} \qquad \dfrac{\overset{\beta}{x : Q}}{\texttt{lam}\ x.\ x : Q \to Q}\to\mathsf{I}^{\beta \lhd x:P}}{\texttt{<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>} : (P \to P) \wedge (Q \to Q)}\wedge\mathsf{I}}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)} : P \to P}\wedge\mathsf{E}_1 \qquad \overset{\delta}{y : P}}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y : P}\to\mathsf{E}$$

$$\mathcal{D}_2 = \dfrac{\mathcal{D}_3 \qquad \dfrac{\overset{\delta}{y : P}}{y \longmapsto y : P}\longmapsto\mathsf{I} \qquad \dfrac{\overset{\delta}{y : P}}{y \longmapsto y : P}\longmapsto\mathsf{I}}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y \longmapsto y : P}\to\mathsf{CE}$$

$$\mathcal{D}_3 = \dfrac{\mathcal{D}_4 \qquad \dfrac{\dfrac{\dfrac{\overset{\gamma}{x : P}}{x \longmapsto x : P}\longmapsto\mathsf{I}}{\texttt{lam}\ x.\ x \longmapsto \texttt{lam}\ x.\ x : P \to P}\to\mathsf{CI}^{\gamma}}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)} \longmapsto_! \texttt{lam}\ x.\ x : P \to P}\wedge\mathsf{R}_1}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)} \longmapsto \texttt{lam}\ x.\ x : P \to P}\wedge\mathsf{CE}_1$$

[5] Since weak head normal forms do not handle any terms over which there are abstracted, for each normal form there are infinitely many weak head normal forms that all reduces to the same normal form.

$\mathcal{D}_4 =$

$$\dfrac{\dfrac{\dfrac{\overset{\epsilon}{x:P}}{x\mathord{\longmapsto} x:P}\,\mathord{\longmapsto}\mathsf{I}}{\texttt{lam}\ x.\ x\mathord{\longmapsto}\texttt{lam}\ x.\ x:P\to P}\,{\to}\mathsf{CI}^{\epsilon\ \triangleleft\ x:P} \qquad \dfrac{\dfrac{\overset{\zeta}{x:Q}}{x\mathord{\longmapsto} x:Q}\,\mathord{\longmapsto}\mathsf{I}}{\texttt{lam}\ x.\ x\mathord{\longmapsto}\texttt{lam}\ x.\ x:Q\to Q}\,{\to}\mathsf{CI}^{\zeta\ \triangleleft\ x:P}}{\texttt{<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>}\mathord{\longmapsto}\texttt{<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>}:(P\to P)\wedge(Q\to Q)}\,\wedge\mathsf{CI}$$

- Reducing $\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y$ with hypothesis $\delta\ \triangleleft\ y:P$ in theory $\mathcal{PE}_{\mathcal{IFOLP}}$:

$$\dfrac{\overset{\mathcal{D}_1}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y:P} \qquad \overset{\mathcal{D}_5}{\begin{array}{l}\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y\\ \mathord{\longmapsto} y:P\end{array}}}{y:P}\,\mathord{\longmapsto}\mathsf{E}$$

where $\mathcal{D}_1$ is as above,

$$\mathcal{D}_5 = \dfrac{\mathcal{D}_6 \qquad \dfrac{}{y\mathord{\longmapsto} y:P}\,\mathord{\longmapsto}\mathsf{I} \qquad \dfrac{\dfrac{}{y\mathord{\longmapsto} y:P}\,\mathord{\longmapsto}\mathsf{I}}{\texttt{lam}\ x.\ x\ \texttt{`}\ y\mathord{\longmapsto}_!\, y:P}\,{\to}\mathsf{R}}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\ \texttt{`}\ y\mathord{\longmapsto} y:P}\,{\to}\mathsf{CE}$$

$$\mathcal{D}_6 = \dfrac{\mathcal{D}_7 \qquad \dfrac{\dfrac{}{\texttt{lam}\ x.\ x\mathord{\longmapsto}\texttt{lam}\ x.\ x:P\to P}\,\mathord{\longmapsto}\mathsf{I}}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\mathord{\longmapsto}_!\,\texttt{lam}\ x.\ x:P\to P}\,\wedge\mathsf{R}_1}{\texttt{fst(<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>)}\mathord{\longmapsto}\texttt{lam}\ x.\ x:P\to P}\,\wedge\mathsf{CE}_1$$

$$\mathcal{D}_7 = \dfrac{}{\texttt{<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>}\mathord{\longmapsto}\texttt{<lam}\ x.\ x\texttt{,lam}\ x.\ x\texttt{>}:(P\to P)\wedge(Q\to Q)}\,\mathord{\longmapsto}\mathsf{I}$$

- Reducing $\texttt{lam}\ x.\ ((\texttt{lam}\ y.\ y)\ \texttt{`}\ x)$ in theory $\mathcal{PR}_{\mathcal{IFOLP}}$:

$$\dfrac{\mathcal{D}_8 \qquad \dfrac{\dfrac{\mathcal{D}_9 \qquad \dfrac{\overset{\eta}{x:P}}{x\mathord{\longmapsto} x:P}\,\mathord{\longmapsto}\mathsf{I} \qquad \mathcal{D}_{10}}{(\texttt{lam}\ y.\ y)\ \texttt{`}\ x\mathord{\longmapsto} x:P}\,{\to}\mathsf{CE}}{\texttt{lam}\ x.\ ((\texttt{lam}\ y.\ y)\ \texttt{`}\ x)\mathord{\longmapsto}\texttt{lam}\ x.\ x:P\to P}\,{\to}\mathsf{CI}^{\eta\ \triangleleft\ x:P}}{\texttt{lam}\ x.\ x:P\to P}\,\mathord{\longmapsto}\mathsf{E}$$

where

$$\mathcal{D}_8 = \dfrac{\dfrac{\dfrac{\overset{\kappa}{y:P}}{\texttt{lam}\ y.\ y:P\to P}\,{\to}\mathsf{I}^{\kappa\ \triangleleft\ y:P} \qquad \overset{\iota}{x:P}}{(\texttt{lam}\ y.\ y)\ \texttt{`}\ x:P}\,{\to}\mathsf{E}}{\texttt{lam}\ x.\ ((\texttt{lam}\ y.\ y)\ \texttt{`}\ x):P\to P}\,{\to}\mathsf{I}^{\iota\ \triangleleft\ x:P}$$

$$\mathcal{D}_9 = \dfrac{\dfrac{\overset{\theta}{y:P}}{y\mathord{\longmapsto} y:P}\,\mathord{\longmapsto}\mathsf{I}}{\texttt{lam}\ y.\ y\mathord{\longmapsto}\texttt{lam}\ y.\ y:P\to P}\,{\to}\mathsf{CI}^{\theta\ \triangleleft\ y:P}$$

$$\mathcal{D}_{10} = \dfrac{\dfrac{\dfrac{\eta}{x : P}}{x \longmapsto x : P}\ \longmapsto \mathsf{I}}{(\mathtt{lam}\ y.\ y)\ \text{`}\ x \longmapsto_! x : P}\ \to \mathsf{R}$$

- Reducing $\mathtt{lam}\ x.\ ((\mathtt{lam}\ y.\ y)\ \text{`}\ x)$ in theory $\mathcal{PE}_{\mathcal{IFOLP}}$:

$$\dfrac{\mathcal{D}_8 \quad \dfrac{}{\mathtt{lam}\ x.\ ((\mathtt{lam}\ y.\ y)\ \text{`}\ x) \longmapsto \mathtt{lam}\ x.\ ((\mathtt{lam}\ y.\ y)\ \text{`}\ x) : P \to P}\ \mathsf{CI}}{\mathtt{lam}\ x.\ ((\mathtt{lam}\ y.\ y)\ \text{`}\ x) : P \to P}\ \longmapsto \mathsf{E}$$

  where $\mathcal{D}_8$ is as above. Note that the the expression is not reduced since it is on weak head normal form.

The examples above show that there is a difference in how proofs are reduced. Most proofs are not reduced so thoroughly with the $\mathcal{PE}_{\mathcal{IFOLP}}$ formal theory as with the $\mathcal{PR}_{\mathcal{IFOLP}}$ formal theory. Whole abstractions can be declared as reduced with the $\mathcal{PE}_{\mathcal{IFOLP}}$ theory whereas only variables can be declared as reduced with the $\mathcal{PR}_{\mathcal{IFOLP}}$ theory. The last example exhausts another difference between the theories which is that a proof does not have the same reduct in each theory. For programs the weak head normal form is sufficient to see what an expression expresses. In a precompilation to avoid reducing redexes in a program at run-time reducing to normal form may be preferred.

4.8 Summary

This chapter has discussed the concepts of proof normalization such that it was possible to use these concepts to describe and reason about proofs and proof reduction methods. To define proof normalization redexes has been discussed. Different forms for proofs were discussed such that it came clear what a goal of a proof reduction may be. It turned out that for proofs the goal form was normal form and for programs the goal form was weak head normal form.

Normalization was discussed for first order logic presented as an axiomatic style logic. It turned out that an axiomatic style logic directly was not well suited for proof reduction because the proof reduction reasoning would be too informal. Therefore a logic for proof reduction was created based on the idea that reasoning about a natural deductive style proof can be done by natural deductive reasoning. The idea was presented as some structural inference rules.

Redexes in the predicate typed extended λ-calculus were presented. These redexes turned out to be exactly equal with redexes for proofs in the $\mathcal{IFOLP}$ theory. This equality were used to state the missing part of the Curry-Howard isomorphism whereby $\mathcal{IFOLP}$ proofs and predicate typed extended λ-calculus programs were the same thing.

For reducing redexes in proofs some strategies were presented. The strategies were discussed to evaluate how well the strategies could be represented as inference rules. It turned out that one reduction strategy could not be implemented and two other strategies had most parts in common. With the theoretical knowledge defined theories for reduction could be formalised. These theories were formalised as theories $\mathcal{PR}_{\mathcal{IFOLP}}$ and $\mathcal{PE}_{\mathcal{IFOLP}}$ using two different (but much alike) reduction strategies.

4.8.1 Directions

This chapter presents proofs as programs and methods to reason about proofs and programs. Since programs are presented as expressions of an extended λ-calculus that is probably not implemented anywhere and as well could be represented using the simply typed λ-calculus, simply typed λ-calculus expressions could be extracted from the $\mathcal{IFOLP}$ proofs. The subject of program extraction is handled well in [1] where programs are extracted from Elf proofs, whence the subject is not discussed in this thesis.

Since proof reduction leads to reduction of don't-care-stuff of programs that is in weak head normal form, a study of implementing partial evaluation of programs using the proof reduction technique may be interesting. This thesis only studies how to represent logics therefore such a specialised logic is too far beyond the scope of this thesis. Another interesting subject that is far beyond the scope of this thesis is graph representations of proofs presented in [17] that could reduce the complexity of proofs.

The proof reduction methods studies only how to create the smallest possible representation of a certain proof. Since reduction only operates inside a given equivalence class of proofs that reduce to the same normal form, reduction does not find a better algorithm for a given program. Therefore it could be profitable to study functions that maps one normal form equivalence class into another.

With formal theories specified for both $\mathcal{IFOLP}$ proofs and proof reduction for $\mathcal{IFOLP}$ proofs applications could be interesting. One theoretical application is to create object logics for $\mathcal{IFOLP}$ representing recursive datatypes as natural numbers and lists, which is done in chapter 5. One practical implementation is to implement the formal theories and do some examples to check whether the theory can be used in practice which is done in chapters 6, 7 and 8.

4.8.2 Literature

Reduction methods are discussed briefly in [17] for the untyped λ-calculus. The correspondence between proofs and λ terms is discussed in [4, 5] which is not used in this paper, and [6, 18].

Chapter 5

Inductive Datatypes and Primitive Recursion

Inductive datatypes play a very important role in applied computer science, especially in programs of a functional programming language. Proofs of propositions including inductive datatypes may have an important role if the resulting proof reducers and program evaluators of this project ever shall be used in practice. Most often the intention of creating a program is to apply the program on some data and it is difficult to represent any usable data without using smart tricks or inductive datatypes.

This chapter discusses what induction is and what inductive proofs can be used for, defines the concept of inductive datatypes and connects proofs including inductive datatypes with primitive recursive programs. An axiomatic system for inductive datatypes is defined and this system is used to define formal theories for natural numbers and lists. These formal theories are extended with linearized proofs and reduction of proofs.

5.1 Induction and Primitive Recursion

Induction is a logical reasoning method for deducing that a proposition is valid if it is valid for some specific cases. Recursion can be viewed as the opposite of induction, which is from a generally valid recursive specification to deduce that the specification is valid for a given specific case by deducing that the specification is valid for another specific case.

A proof by induction may be linearized. Such a linearized proof is a recursive program. Some sorts of induction and recursion are used in this thesis.

5.1.1 Induction

Definition 5.1.1 *Let a set S and an ordering relation $\prec$ on S be given. Furthermore, let $v, w \in S$ be given. if $w \prec v$ then w is an immediate predecessor of v, denoted $w \overset{!}{\prec} v$, iff*

$$\forall x \in S : (w \prec x) \Rightarrow (x \not\prec v)$$

In the definition, "immediate" refers to the fact that no element bigger than w is predecessor of v.

Induction is often formed as an axiom for some set of elements. To define an induction axiom on a set S, the set must fulfil some requirements.

Definition 5.1.2 *To make induction over a set S the set has to be*

55s2 *enumerable, and*

56s2 *equipped with an ordering relation $\prec$, such that for any element $v \in S$, either v is a minimal element, or $\exists w \in S : w \overset{!}{\prec} v$.*

In mathematics and applied computer science, different sorts of induction are often in use. The one used in this thesis is structural induction, a corollary on the induction axiom of definition 5.3.2.

Consequence 5.1.3 *Let S be a set fulfilling 5.1.2. A given predicate P is proven to be valid for all $s \in S$ by structural induction on S if*

- *P is valid for all minimal elements of S, and*
- *P is valid for v if P is valid for any immediate predecessor of v, for any $v \in S$.*

Among other induction methods are complete induction.

5.1.2 About Recursion

Different methods exists for expressing function repeating, i.e. as expressed in [10]. A sort of recursion is primitive recursion.

Definition 5.1.4 *Let A be a set and let S be a set that is suitable for induction. A primitive recursive function $f : S \to A$ is a function that*

57s2 *if $v \in S$ is a minimal element then $f(v)$ is defined directly, else*

58s2 *for any v having an immediate predecessor w is defined such that $f(v) = g(f(w))$ for some function g (that may depend on v).*

The connection between primitive recursion and structural induction is that *55s2* has a proof of form *57s2* and *56s2* has a proof of form *58s2*.

5.2 Datatype Definitions

Datatypes are used to construct sorts of data and to group informations and as well also define which interpretation that must be used on the informations. If nothing is known about the type of some data it may be impossible to interpret what the data denotes. I.e. the data 666 may either be a house number, a room number, a trace path 1010011010 to a leaf in a binary tree, a reference to Revelation 13, the numbers of the delimiters in an EAN bar code, a reminder to the lyrics of "The Number of The Beast" by Iron Maiden, a page number in a book or a testing argument to

a function. Therefore a datatype may be constructed where the datatype has some construction operators that tag a datum to be of a certain type.

The interesting thing about a datatype is that it can be made inductive such that a datatype uses a value of its own type for creating a new value. An inductive datatype can be used for defining infinite sets. As a simple example, exhausted below, natural numbers as an inductive datatype consists of either a constructor denoting zero or a constructor applied on a natural number (which is the reason that the definition is inductive) that denotes the successor of the natural number.

5.3 Using Induction and Datatypes for Creating Formal Theories

Inductive datatypes can be used for implementing formal theories for reasoning about infinite sets. An axiomatic system that can be used to define reasoning about such sets is the Peano axioms.

5.3.1 Peano Axioms

The Peano axioms [3, page 667] are originally stated for natural numbers but can be used for other inductive datatypes as well. The below set of axioms is therefore stated for values of some arbitrary inductive datatype.

Definition 5.3.1 *Let base constructors of an inductive datatype be the constructors that cannot be applied on values of the same datatype and let step constructors be the constructors that must be applied on values of the same datatype.*

All constructors of an inductive datatype are grouped as either base or step constructors.

Definition 5.3.2 *An axiomatic system for values of a given inductive datatype consists of the following set of axioms.*

59ss *Base constructors are values.*

60ss *Let S be a step constructor taking n arguments of the datatype. If $a_1, \ldots, a_n$ are values, then the result of applying S on $a_1, \ldots, a_n$ is a value.*

61ss *No base constructor is the result of applying a step constructor on some values.*

62ss *Let S be a step constructor taking n arguments of its own datatype. If the result of applying S on the ordered set $a_1, \ldots, a_n$ is equal to the result of applying S on the ordered set $b_1, \ldots, b_n$ then a_i is equal with b_i for all $i \in \{1, \ldots, n\}$.*

63ss *If a property P is valid for all base constructors, and also is valid for any values that can be constructed with any step constructor of the datatype applied on values for which P is valid, then P is valid for any value of the datatype.*

An examination of the Peano axioms gives that axioms *59ss* and *60ss* are properties of the datatype and the axioms *61ss*, *62ss* and *63ss* are definitions of the behaviour of values of the datatype.

The Peano axioms define how to state and reason about an inductive datatype. Though some values of an inductive datatype may be infinitely big, which is impossible to represent within finite

space, recursion may make it possible to represent some infinite structures if they follow a decent pattern. The pattern is that the same action happens with the recursive term for each datatype constructor.

Definition 5.3.3 *An axiom for primitive recursion is defined as*

64s4 An operator $R_{c_1,\dots,c_n,f_1,\dots,f_m}(v)$ *taking as arguments*

- *a value* v *of the datatype,*
- *for each datatype base constructor a corresponding constant* c_i*, and*
- *for each datatype step constructor a corresponding function* f_i *with arguments being*
  - *the* l *arguments of the step constructor, and*
  - *for each of the* l *arguments of the step constructor that is of the datatype an application of* R *on the argument as argument to* f_i*.*

R is a recursion operator if it fulfils the requirements

(a) if the value v *on which* R *is applied is a base constructor then the application of* R *on* v *equals the corresponding constant* c_i*,*

(b) if the value v *on which* R *is applied is a step constructor with arity* l *that is applied on values* $v_1,\dots,v_l$ *and extra arguments* $a_1,\dots,a_k$ *not of the datatype, then the application of* R *on* v *equals the corresponding function* f_i *applied on the values* $v_1,\dots,v_l$ *and* R *applied on* $v_1,\dots,v_l$*,*

$$f_i(a_1,\dots,a_k,v_1,\dots,v_l,R_{c_1,\dots,c_n,f_1,\dots,f_m}(v_1),\dots,R_{c_1,\dots,c_n,f_1,\dots,f_m}(v_l))$$

In a formal theory based on the above axioms, the axioms *59ss* and *60ss*, defining a complete specification of the datatype, specify the term letters that construct the datatype. The axioms *61ss*, *62ss* and *63ss* and also definition *64s4* specify the axioms and inference rules define how to reason about values of the datatype.

5.3.2 Natural Numbers

A formal theory for natural numbers must be created. Such a formal theory may be an extension of an existing formal theory such that the formal theory for natural numbers makes it possible to reason about reasoning about natural numbers. Such a formal theory could be $\mathcal{IFOL}$ of section 2.4.4.

The formal theory for natural numbers is extended with linearized proofs, and logics for reasoning about linearized proofs with natural numbers are created.

Natural Number Datatype

A datatype for natural numbers must resemble the nature of natural numbers using the Peano axioms. Therefore the datatype for natural numbers has two datatype constructors which are a constructor for the base of natural numbers, zero, and a constructor for creating the step on a natural number, the successor. Let these constructors be denoted $\mathsf{0}$ and S fulfilling axiom *59ss*, respectively axiom *60ss*, and let the name of the datatype be Nat.

Natural Number Reasoning

Reasoning about natural numbers may be done using inference rules corresponding to the Peano axioms and the recursion axiom. These rules may either be specified as axioms or as inference rules. Specifying the rules as axioms will require that some constructions for implication and quantification are used for building the axioms. These constructions must be defined somewhere, necessarily in the base formal theory. Therefore inference rules may be preferred to axioms to get the full exchangeability of base formal theories. One predicate letter is needed for equality among numbers.

A problem with the axioms stated as inference rules is that i.e. induction stated as an inference rule may not assure that only one set of premises can lead to the conclusion since. Property P of axiom *63ss* is parameterized and without a quantifier the arguments of P may probably not be bound correct. Since the inference rules can be used to prove theorems that resemble the axioms, the inference rules may be preferred in a formal theory.

Inference rules based on the Peano axioms are stated as follows. Axiom *61ss* leads to the inference rule

$$\frac{\mathsf{S}(m) = \mathsf{0}}{R}\ \mathsf{S} \neq \mathsf{0}$$

axiom *62ss* leads to the inference rule

$$\frac{\mathsf{S}(m) = \mathsf{S}(n)}{m = n}\ \text{S-inject}$$

axiom *63ss* leads to the inference rule

$$\frac{P(\mathsf{0}) \qquad \begin{array}{c}[P(x)]\\ \Lambda x \quad \vdots \\ P(\mathsf{S}(x))\end{array}}{P(n)}$$

without stating directly that the conclusion is valid for any natural number. Therefore it may be convenient to bind variable n of the conclusion giving rule

$$\frac{P(\mathsf{0}) \qquad \begin{array}{c}[P(x)]\\ \Lambda x \quad \vdots \\ P(\mathsf{S}(x))\end{array}}{\forall n.P(n)}\ \text{induct}$$

Let the recursion operator for natural numbers be rec, let the first subscript of rec be the constant corresponding to datatype constructor 0 and let the second subscript of rec be the function corresponding to datatype constructor S. Then definition *64s4* leads to the logical axioms

$$\frac{}{\mathsf{rec}_{a,f}(\mathsf{0}) = a}\ \mathsf{rec}_0 \quad \text{and} \quad \frac{}{\mathsf{rec}_{a,f}(\mathsf{S}(m)) = f(m, \mathsf{rec}_{a,f}(m))}\ \mathsf{rec}_\mathsf{S}$$

Definition 5.3.4 *A formal theory $\mathcal{NAT}$ for natural numbers based upon formal theory $\mathcal{IFOL}$ is defined as the following extensions to $\mathcal{IFOL}$*

Symbols *are extended with constant* 0*, with function* S *of arity 1 and with reserved predicate* rec *of arity 3.*

Axioms *are extended with* rec_0 *and* rec_S.

Inference Rules *are extended with* $\mathsf{S} \neq \mathsf{0}$*, S-inject and induct.*

Proofs Including Natural Numbers

Formal theory $\mathcal{NAT}$ can be extended with linearized proof constructions just like $\mathcal{IFOL}$ was extended to $\mathcal{IFOLP}$. To do this, some proof structors must be created. These structors cannot be grouped as either constructors or destructors since the rules for reasoning about natural numbers cannot be grouped as either introduction or elimination rules. Since rules rec_0, rec_S, $\mathsf{S} \neq 0$, S-inject and induct are used to build up knowledge about natural numbers, these rules may be viewed as introduction rules corresponding to constructors; since all rules can be viewed as introduction rules, there are no elimination rules for natural numbers.

Definition 5.3.5 *A formal theory $\mathcal{NATP}$ for natural numbers with proofs based upon formal theory $\mathcal{IFOLP}$ can be defined as the following extensions to formal theory $\mathcal{IFOLP}$*

Symbols *The symbol set of $\mathcal{IFOLP}$ is extended with the following predicates, structors, constants and functions:*

Structors *nrec, ninj, nneq, recS, rec0, each of fixed arity,*

Constants *0,*

Functions *S of arity 1,*

Predicates *reserved predicate rec of arity 3.*

Propositions *consists of a proof part and a proposition part; the definition of proof parts is extended with*

65_{s6} `nrec`(p_1, p_2) 68_{s6} `recS`

66_{s6} `ninj`(p_1)

67_{s6} `nneq`(p_1) 69_{s6} `rec0`

where p_n is a proof part.

Axioms *The axiom set of $\mathcal{IFOLP}$ is extended with the axioms of figure 5.1.*

Inference Rules *The set of inference rules of $\mathcal{IFOLP}$ is extended with the inference rules of figure 5.1.*

Notice that if the quantification of the conclusion of rule induct was not done in the discussion of theory $\mathcal{NAT}$, the conclusion of rule induct should be something like `nrec(n, b, c)` : P(`n`). This gives a double representation of the induction variable that may be unfortunate. Also notice that the variables of c in the premise are quantified in the proof part of the conclusion.

Reduction of Proofs including Natural Numbers

In sections 4.5–4.7.4 reduction of proofs of first order logic is discussed. This discussion may be used to create a formal theory for reduction of proofs including natural numbers.

Since the rules rec_0, rec_S, $\mathsf{S} \neq 0$, S-inject and induct leads to constructors, these rules lead to the set of context introduction rules on figure 5.2. The only one of the constructors that can take part in a redex is `nrec` in combination with destructor `^`. This constructor leads to the reduction rules

$$\frac{b \longmapsto b' : A}{\texttt{nrec}(b,\ i)\ \texttt{\^{}}\ 0 \longmapsto_! b' : A}\ \mathsf{inductR}_0$$

Figure 5.1 Rules for natural numbers with proofs based on the Peano axioms and the recursion definition.

$$\frac{b : P(\mathsf{0}) \qquad \begin{array}{c}[u : P(x)] \\ \Lambda x, u \quad \vdots \\ c(x,u) : P(\mathsf{S}(x))\end{array}}{\texttt{nrec(}b\texttt{, all } x\texttt{. lam } u\texttt{. } c(x,u)\texttt{)} : \forall n.P(n)}\ \textsf{induct}$$

$$\frac{p : \mathsf{S}(m) = \mathsf{S}(n)}{\texttt{ninj(}p\texttt{)} : m = n}\ \textsf{S-inject}$$

$$\frac{p : \mathsf{S}(m) = \mathsf{0}}{\texttt{nneq(}p\texttt{)} : R}\ \mathsf{S} \neq \mathsf{0}$$

$$\frac{}{\begin{array}{l}\texttt{recS} : \mathsf{rec}_{a,f}(\mathsf{S}(m)) \\ = f(m, \mathsf{rec}_{a,f}(m))\end{array}}\ \mathsf{rec_S}$$

$$\frac{}{\texttt{rec0} : \mathsf{rec}_{a,f}(\mathsf{0}) = a}\ \mathsf{rec_0}$$

and

$$\frac{(i\ \hat{}\ n)\ \texttt{`}\ \texttt{(nrec(}b\texttt{, } i\texttt{)}\ \hat{}\ n) \longmapsto c : A}{\texttt{nrec(}b\texttt{, } i\texttt{)}\ \hat{}\ \mathsf{S(n)} \longmapsto_! c : A}\ \mathsf{inductR_S}$$

where

`b` is the proof of the case for $\mathsf{0}$ that must be the result when the recursion terminates,

`i` is the proof of the case for S that must build the result of the recursive case of the recursion. It is a function that must be applied on the argument n of S and the recursion term applied on the argument n of S.

Notice that the second argument to `nrec` is specified a bit awkward in rule induct, the reason for the specification is specified in rule $\mathsf{inductR_S}$ where i is applied on its arguments using destructors instead of using the arguments to i as parameters of i.

With the above defined context rules and reduction rules a formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ for proof reduction of proofs including natural numbers can be created.

Definition 5.3.6 *Formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ is based on formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$, and is therefore defined as $\mathcal{PR}_{\mathcal{IFOLP}}$ extended with the following*

Symbols *are the symbols used to extend $\mathcal{IFOLP}$ to $\mathcal{NATP}$.*

Propositions *are extended like $\mathcal{IFOLP}$ is extended to $\mathcal{NATP}$.*

Axioms *are the axioms used to extend $\mathcal{IFOLP}$ to $\mathcal{NATP}$ and axioms of figure 5.2.*

Inference rules *are the inference rules used to extend $\mathcal{IFOLP}$ to $\mathcal{NATP}$, inference rules of figure 5.2 and rules $\mathsf{inductR_0}$ and $\mathsf{inductR_S}$.*

Also, a formal theory for program evaluation of proofs as programs including natural numbers can be created. Notice that no context rules must be included, program evaluation uses only context elimination rules and there are no context elimination rules for natural numbers.

Figure 5.2 Constructor context rules for reducing of proof terms for natural numbers.

$$\frac{b \longmapsto b' : P(\mathsf{0}) \qquad \begin{array}{c}[u : P(x)] \\ \Lambda x, u \quad \vdots \\ c(x,u) \longmapsto c'(x,u) : P(\mathsf{S}(x))\end{array}}{\texttt{nrec(}b\texttt{, all } x\texttt{. lam } u\texttt{. } c(x,u)\texttt{)} \longmapsto \texttt{nrec(}b'\texttt{, all } x\texttt{. lam } u\texttt{. } c'(x,u)\texttt{)} : \forall n.P(n)}\ \mathsf{inductCI}$$

$$\frac{p \longmapsto p' : \mathsf{S}(m) = \mathsf{S}(n)}{\texttt{ninj(}p\texttt{)} \longmapsto \texttt{ninj(}p'\texttt{)} : m = n}\ \mathsf{S\text{-}injectCI}$$

$$\frac{p \longmapsto p' : \mathsf{S}(m) = \mathsf{0}}{\texttt{nneq(}p\texttt{)} \longmapsto \texttt{nneq(}p'\texttt{)} : R}\ \mathsf{S \neq 0CI}$$

$$\frac{}{\texttt{recS} \longmapsto \texttt{recS} : \mathsf{rec}_{a,f}(\mathsf{S}(m)) = f(m, \mathsf{rec}_{a,f}(m))}\ \mathsf{rec_S CI}$$

$$\frac{}{\texttt{rec0} \longmapsto \texttt{rec0} : \mathsf{rec}_{a,f}(\mathsf{0}) = a}\ \mathsf{rec_0 CI}$$

Definition 5.3.7 *Formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ is based on formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$, and is therefore defined as $\mathcal{PE}_{\mathcal{IFOLP}}$ extended with the following*

Symbols *are the symbols used to extend $\mathcal{IFOLP}$ to $\mathcal{NATP}$.*

Propositions *are extended like $\mathcal{IFOLP}$ is extended to $\mathcal{NATP}$.*

Axioms *are the axioms used to extend $\mathcal{IFOLP}$ to $\mathcal{NATP}$.*

Inference rules *are the inference rules used to extend $\mathcal{IFOLP}$ to $\mathcal{NATP}$ and rules $\mathsf{inductR_0}$ and $\mathsf{inductR_S}$.*

5.3.3 Lists

A formal theory for lists of elements of some type must be created. Such a formal theory may be an extension of an existing formal theory such that the formal theory for lists of elements of some type makes it possible to reason about reasoning about lists of elements of some type. As in constructing the formal theory for natural numbers, $\mathcal{IFOL}$ defined in section 2.4.4 is used as basis for the new formal theory.

The formal theory for lists of elements of some type is extended with linearized proofs, and logics for reasoning about linearized proofs with lists of elements of some type are created.

List Datatype

A datatype for lists of elements of some type must resemble the nature of lists of elements of some type using the Peano axioms. Therefore the datatype for lists of elements of some type has two datatype constructors which are a constructor for the base of lists of elements of some type, the empty list, and a constructor for creating a step on a list of elements of some type, the cons-operator extending a list with another element. Let the name of the datatype for lists of elements of some type α be α list and let the constructors of the datatype be Nil and C taking an extension-element of type α and an α list as arguments.

List Reasoning

Reasoning about lists of elements of some type may be done using inference rules corresponding to the Peano axioms and the recursion definition. As for natural numbers these rules may be specified either as axioms or as inference rules and, as for natural numbers, inference rules may be preferred following the same argumentation as for natural numbers.

Inference rules based on the Peano axioms are stated as follows. Axiom *61ss* leads to inference rule

$$\frac{\mathsf{C}(x, xs) = \mathsf{Nil}}{R}\ \mathsf{C} \neq \mathsf{Nil}$$

axiom *62ss* leads to inference rules

$$\frac{\mathsf{C}(x_1, xs_1) = \mathsf{C}(x_2, xs_2)}{xs_1 = xs_2}\ \mathsf{C}_{\mathsf{inj1}} \quad \text{and} \quad \frac{\mathsf{C}(x_1, xs_1) = \mathsf{C}(x_2, xs_2)}{x_1 = x_2}\ \mathsf{C}_{\mathsf{inj2}}$$

axiom *63ss* leads to inference rule

$$\frac{P(\mathsf{Nil}) \qquad \begin{array}{c}[P(xs)] \\ \Lambda x, xs \ \vdots \\ P(\mathsf{C}(x, xs))\end{array}}{P(l)}$$

without stating directly that the conclusion is valid for any list of elements of some type. Therefore it may be convenient to bind variable l of the conclusion giving rule

$$\frac{P(\mathsf{Nil}) \qquad \begin{array}{c}[P(xs)] \\ \Lambda x, xs \ \vdots \\ P(\mathsf{C}(x, xs))\end{array}}{\forall l.P(l)}\ \mathsf{inductl}$$

Let the recursion operator for lists of elements of some type be recl, the first subscript of recl be the constant corresponding to datatype constructor Nil and let the second subscript of recl be the function corresponding to datatype constructor C. Then definition *64s4* leads to the logical axioms

$$\frac{}{\mathsf{recl}_{a,f}(\mathsf{Nil}) = a}\ \mathsf{recl}_{\mathsf{Nil}} \quad \text{and} \quad \frac{}{\mathsf{recl}_{a,f}(\mathsf{C}(x, xs)) = f(x, xs, \mathsf{recl}_{a,f}(xs))}\ \mathsf{recl}_{\mathsf{C}}$$

Definition 5.3.8 *A formal theory $\mathcal{LIST}$ for lists of elements of some type based upon formal theory $\mathcal{IFOL}$ is defined as the following extensions to $\mathcal{IFOL}$*

Symbols *are extended with constant Nil, with function C of arity 2 and with reserved predicate recl of arity 3.*

Axioms *are extended with $\mathsf{recl}_{\mathsf{Nil}}$ and $\mathsf{recl}_{\mathsf{C}}$*

Inference Rules *are extended with $\mathsf{C} \neq$ Nil, $\mathsf{C}_{\mathsf{inj1}}$, $\mathsf{C}_{\mathsf{inj2}}$ and inductl.*

Proofs Including Lists

Formal theory $\mathcal{LIST}$ can as well be extended with linearized proof constructions. To do this, some proof structors must be created. As with natural numbers these structors cannot be grouped as either constructors or destructors. Since rules $\mathsf{recl}_{\mathsf{Nil}}$, $\mathsf{recl}_{\mathsf{C}}$, $\mathsf{C} \neq \mathsf{Nil}$, $\mathsf{C}_{\mathsf{inj1}}$, $\mathsf{C}_{\mathsf{inj2}}$ and inductl are used to build up knowledge about lists of elements of some type these rules may be viewed as introduction rules corresponding to constructors. So there are no elimination rules for lists of elements of some type.

Definition 5.3.9 *A formal theory $\mathcal{LIST}\mathcal{P}$ for lists of elements of some type with proofs based upon formal theory $\mathcal{IFOLP}$ is defined as the following extensions to formal theory $\mathcal{IFOLP}$*

Symbols *The symbol set of $\mathcal{IFOLP}$ is extended with the following predicates, structors, constants and functions:*

Structors `lrec`, `linj`, `ainj`, `lneq`, `recC`, `recNil`, *each of fixed arity,*

Constants *Nil,*

Functions *C of arity 2,*

Predicates *reserved predicate lrec of arity 3.*

Propositions *consists of a proof part and a proposition part; the definition of proof parts is extended with*

70_{po} `lrec(`p_1`,`p_2`)`	73_{po} `lneq(`p_1`)`
71_{po} `linj(`p_1`)`	74_{po} `recC`
72_{po} `ainj(`p_1`)`	75_{po} `recNil`

where p_n is a proof part.

Axioms *The axiom set of $\mathcal{IFOLP}$ is extended with the axioms of figure 5.3.*

Inference Rules *The set of inference rules of $\mathcal{IFOLP}$ is extended with the inference rules of figure 5.3.*

Figure 5.3 Rules for lists of elements of some type with proofs based on the Peano axioms and the recursion definition.

$$\frac{b : P(\mathsf{Nil}) \qquad \begin{array}{c}[u : P(xs)] \\ \Lambda x,\ xs,\ u \quad \vdots \\ c(x, xs, u) : P(\mathsf{C}(x, xs))\end{array}}{\mathtt{lrec}(b, \mathtt{all}\ x\ xa.\ \mathtt{lam}\ xb.\ c(x, xa, xb) : \forall l.P(l)}\ \mathsf{inductl}$$

$$\frac{p : \mathsf{C}(x_1, xs_1) = \mathsf{C}(x_2, xs_2)}{\mathtt{linj}(p) : xs_1 = xs_2}\ \mathsf{C}_{\mathsf{inj}_1}$$

$$\frac{p : \mathsf{C}(x_1, xs_1) = \mathsf{C}(x_2, xs_2)}{\mathtt{ainj}(p) : x_1 = x_2}\ \mathsf{C}_{\mathsf{inj}_2}$$

$$\frac{p : \mathsf{C}(x, xs) = \mathsf{Nil}}{\mathtt{lneq}(p) : R}\ \mathsf{C} \neq \mathsf{Nil}$$

$$\frac{}{\mathtt{recNil} : \mathsf{recl}(\mathsf{Nil}, a, f) = a}\ \mathsf{recl}_{\mathsf{Nil}}$$

$$\frac{}{\mathtt{recC} : \mathsf{recl}_{a,f}(\mathsf{C}(x, xs)) = f(x, xs, \mathsf{recl}_{a,f}(xs))}\ \mathsf{recl}_{\mathsf{C}}$$

As for natural numbers the quantification of the conclusion of rule inductl is done to avoid double representation of the induction variable that may be unfortunate. Notice that the variables of c in the premise are quantified in the proof part of the conclusion.

Reduction of Proofs including Lists

A formal theory for reduction of proofs including lists is created like the formal theory for reducing proofs with natural numbers. Since the rules $\mathsf{rec}_{\mathsf{Nil}}$, $\mathsf{rec}_{\mathsf{C}}$, $\mathsf{C} \neq \mathsf{Nil}$, $\mathsf{C}_{\mathsf{inj}1}$, $\mathsf{C}_{\mathsf{inj}2}$ and inductl lead to constructors these rules leads to the set of context introduction rules on figure 5.4. The only one

Figure 5.4 Constructor context rules for reducing of proof terms for lists of elements of some type.

$$\frac{\begin{array}{c}[\mathtt{u} : P(\mathtt{xs})] \\ \Lambda \mathtt{x}, \mathtt{xs}, \mathtt{u}. \quad \vdots \\ \mathtt{b} \longmapsto \mathtt{b'} : P(\mathsf{Nil}) \qquad \mathtt{c(x,\ xs,\ u)} \longmapsto \mathtt{c'(x,\ xs,\ u)} : P(\mathsf{C}(\mathtt{x}, \mathtt{xs}))\end{array}}{\mathtt{lrec(b,\ c)} \longmapsto \mathtt{lrec(b',\ c')} : \forall l.P(l)}\ \mathsf{inductlCI}$$

$$\frac{\mathtt{p} \longmapsto \mathtt{p'} : \mathsf{C}(x_1, xs_1) = \mathsf{C}(x_2, xs_2)}{\mathtt{linj(p)} \longmapsto \mathtt{linj(p')} : xs_1 = xs_2}\ \mathsf{C}_{\mathsf{inj}_1}\mathsf{CI}$$

$$\frac{\mathtt{p} \longmapsto \mathtt{p'} : \mathsf{C}(x_1, xs_1) = \mathsf{C}(x_2, xs_2)}{\mathtt{ainj(p)} \longmapsto \mathtt{ainj(p')} : x_1 = x_2}\ \mathsf{C}_{\mathsf{inj}_2}\mathsf{CI}$$

$$\frac{\mathtt{p} \longmapsto \mathtt{p'} : \mathsf{C}(x, xs) = \mathsf{Nil}}{\mathtt{lneq(p)} \longmapsto \mathtt{lneq(p')} : R}\ \mathsf{C} \neq \mathsf{Nil}\mathsf{CI}$$

$$\frac{}{\mathtt{recNil} \longmapsto \mathtt{recNil} : \mathsf{recl}(\mathsf{Nil}, a, f) = a}\ \mathsf{recl}_{\mathsf{Nil}}\mathsf{CI}$$

$$\frac{}{\mathtt{recC} \longmapsto \mathtt{recC} : \mathsf{recl}(\mathsf{C}(x, xs), a, f) = f(x, xs, \mathsf{recl}(xs, a, f))}\ \mathsf{recl}_{\mathsf{C}}$$

of the constructors that can take part in a redex is `lrec` in combination with destructor `^`. This constructor leads to the reduction rules

$$\frac{b \longmapsto b' : A}{\mathtt{lrec(}b,\ i\mathtt{)\ ^\wedge\ Nil} \longmapsto_! b' : A}\ \mathsf{inductlR}_{\mathsf{Nil}}$$

and

$$\frac{\mathtt{((}i\mathtt{\hat{}}x\mathtt{)\ ^\wedge\ }xs\mathtt{)\ `\ (lrec(}b,\ \mathtt{i)\ ^\wedge\ }xs\mathtt{)} \longmapsto c : A}{\mathtt{lrec(}b,\ i\mathtt{)\ ^\wedge\ C(}x, xs\mathtt{)} \longmapsto_! c : A}\ \mathsf{inductlR}_{\mathsf{C}}$$

where

`b` is the proof of the case for Nil that must be the result when the recursion terminates,

`i` is the proof of the case for C that must build the result of the recursive case of the recursion. It is a function that must be applied on the arguments x, xs of C, and recursion on the inductive argument xs of C.

Notice that the second argument to `lrec` is specified a bit awkwardly in rule inductl, the reason for the specification is given in rule $\mathsf{inductlR}_{\mathsf{C}}$ where i is applied on its arguments using destructors instead of using the arguments to i as parameters of i.

With the above defined context rules and reduction rules a formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ for proof reduction of proofs including lists of elements of some type can be created.

Definition 5.3.10 *Formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ is based on formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$, and is therefore defined as $\mathcal{PR}_{\mathcal{IFOLP}}$ extended with the following*

Symbols *are the symbols used to extend $\mathcal{IFOLP}$ to $\mathcal{LISTP}$.*

Propositions *are extended like $\mathcal{IFOLP}$ is extended to $\mathcal{LISTP}$.*

Axioms *are the axioms used to extend $\mathcal{IFOLP}$ to $\mathcal{LISTP}$ and axioms of figure 5.4.*

Inference rules *are the inference rules used to extend $\mathcal{IFOLP}$ to $\mathcal{LISTP}$, inference rules of figure 5.4 and rules $\mathsf{inductIR}_{\mathsf{Nil}}$ and $\mathsf{inductIR}_{\mathsf{C}}$.*

Also a formal theory for program evaluation of proofs as programs including lists of elements of some type $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ can be created. Notice that no context rules must be included, program evaluation uses only context elimination rules and there are no context elimination rules present for lists of elements of some type.

Definition 5.3.11 *Formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ is based on formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$, and is therefore defined as $\mathcal{PE}_{\mathcal{IFOLP}}$ extended with the following*

Symbols *are the symbols used to extend $\mathcal{IFOLP}$ to $\mathcal{LISTP}$.*

Propositions *are extended like $\mathcal{IFOLP}$ is extended to $\mathcal{LISTP}$.*

Axioms *are the axioms used to extend $\mathcal{IFOLP}$ to $\mathcal{LISTP}$.*

Inference rules *are the inference rules used to extend $\mathcal{IFOLP}$ to $\mathcal{LISTP}$ and rules $\mathsf{inductIR}_{\mathsf{Nil}}$ and $\mathsf{inductIR}_{\mathsf{C}}$.*

5.4 Summary

This chapter connected structural induction with primitive recursion and used this connection for creating formal theories for reasoning about recursive programs on inductive data.

The Peano axioms were extended to cover structural inductive sets and primitive recursion thereon. The extended Peano axioms were used to define natural numbers and lists of elements of some type.

With the definition of formal theories $\mathcal{IFOL}$, $\mathcal{IFOLP}$, $\mathcal{PR}_{\mathcal{IFOLP}}$ and $\mathcal{PE}_{\mathcal{IFOLP}}$ formal theories for reasoning about

- natural numbers,
- natural numbers with proofs,
- reasoning about proofs of natural numbers with proofs,
- reasoning about programs as if they were of natural numbers with proofs,
- lists of elements of some type,
- lists of elements of some type with proofs,
- reasoning about proofs of lists of elements of some type with proofs,
- reasoning about programs as they were of lists of elements of some type with proofs,

were created.

5.4.1 Directions

This chapter presents primitive recursion. It could be interesting to study how to represent other sorts of recursion.

The formal theories can be implemented and experimented with. The formal theories are implemented in appendix A and experimented with in appendix B. The formal theories also take part in the case study in chapter 8.

5.4.2 Literature

The history of the Peano axioms can be found in [3] together with other information about how mathematics evolved from unary sheep counting to what is known today. How to construct inductive proofs in general is handled in [7] that presents inductive proofs for first year students. This is further extended in [10] that may be good background material for creating formal theories for other sorts of recursion and induction.

Part II

Implementation

Chapter 6

Implementing Logic Theories in Isabelle

The first part of this thesis presented some formal theories using different ways to represent the first order logic and other logics either for reasoning about first order logic or using first order logic as meta logic.

The formal theories are implemented in a proof search system called Isabelle.

This chapter discusses *why* to choose Isabelle as proof search system and presents *how* to use Isabelle. Eventually some examples are presented.

6.1 Proof Search System Isabelle

Isabelle is a proof search system. Proof search systems are tools that can assist a user in searching for and creating proofs for propositions. Such a proof search system is a system[1] that implements a logic and only can do proofs that are valid in the implemented logic. Therefore proofs created with a proof search system are valid proofs in the implemented logic. Another fortune of proof search systems is that human errors (like reading errors and typig errors) are restricted to theories and propositions stated by some user. Then human errors cannot occur in the proofs.

There are several proof search systems to choose among, e.g. COQ, ALF, LEGO and Isabelle. Isabelle was recommended by my first thesis advisor. Web pages have been studied briefly to discover if any of the other proof search systems have advantages over Isabelle. Some were fastly rejected for requiring use of a graphical user interface, others did not allow freedom in object logic creating.

Isabelle implements several logics, among which is a version of natural deductive first order logic and a version of natural deductive first order logic with proofs. The user can add new logics to the logics implemented and formal theories can be represented as inference rules of a meta logic instead of as implementation language functions. Isabelle does not have to be used interactively as other proof search systems and can thereby be used for doing processor time consuming proof crunching without human supervision in front of a terminal as with arm-destroying throw-a-mouse-through-a-window systems. Eventually the implementation language of Isabelle is ML in which extensions

[1] Often such a system is a piece of computer software but can of course also be another sort of strategy. This work focuses only on computer software since the goal is automatically derived minimal proofs.

can be made, and the source code seems to be easy to correct or update if necessary. Among several proof search systems Isabelle is a reasonable choice; probably other proof search systems can do the same as Isabelle and may thereby be just as good a choice, but Isabelle is sufficient for the use of a proof search system in this thesis.

Isabelle can be retrieved through world wide web at the Isabelle FTP-site located at

`ftp://ftp.cl.cam.ac.uk/ml/index.html`

The version of Isabelle used in this project is Isabelle94-4. This old version is still used since it may be a bad idea to upgrade software in the middle of a project if the old one seems to be sufficient as long as the project runs[2].

The following sections give an overview of Isabelle and describe how it is structured. The start of the development (or more precisely: evolution) of Isabelle is described in [11].

6.2 Meta Logic

The meta logic used in Isabelle is the logic that is used as description language for all the logics implemented in Isabelle. The meta logic is documented in [11, 14]. It is programmed as real ML functions and not as inference rules. The meta logic is implemented as a fragment of higher order logic, which consists of rules for implication ("$\Rightarrow$" operator), equality ("$\equiv$" operator) and universal quantification ("Λ" operator) together with conversion for the operators, implemented with the inference rules

$$\frac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \Rightarrow \psi}\ \Rightarrow \mathsf{I} \qquad \frac{\phi[x]}{\Lambda x.\ \phi[x]}\ \Lambda \mathsf{I} \qquad \frac{\begin{array}{cc}[\phi] & [\psi]\\ \vdots & \vdots\\ \psi & \phi\end{array}}{\phi \equiv \psi}\ \equiv \mathsf{I}$$

$$\frac{\phi \Rightarrow \psi \quad \phi}{\psi}\ \Rightarrow \mathsf{E} \qquad \frac{\Lambda x.\phi[x]}{\phi[t/x]}\ \Lambda \mathsf{E} \qquad \frac{\phi \equiv \psi \quad \phi}{\psi}\ \equiv \mathsf{E}$$

as described and explained in [11, 14]. The variable x may not be free in the assumptions of rule ΛI. Variables are instantiated with the rule

$$\frac{\phi}{\phi[t_1/x_1, \ldots, t_n/x_n]}$$

The equality operator "$\equiv$" is reflexive, symmetrical and transitive. Moreover the equality operator defines the conversion rules for abstractions, where abstractions are represented with the Π operator, i.e. $\Pi x.\ P(x)$, using the following rules:

$$\Pi x.\phi \equiv \Pi y.\phi[y/x] \qquad \frac{\phi[x] \equiv \psi[x]}{\phi \equiv \psi}$$

[2] A software upgrade from Isabelle94-4 to Isabelle98-8 is not only an ML source code upgrade since Isabelle94-4 is written in the old ML version and Isabelle98-8 is written in the new ML version, also the ML system has to be replaced by another.

The equality operator also implements meta level β conversion by mapping $(\Pi x.\ \phi)(u)$ to the equality $(\Pi x.\ \phi)(u) \equiv \phi[u/x]$.

Abstraction and combination are implemented with the rules

$$\frac{\phi \equiv \psi}{\Pi x.\phi \equiv \Pi x.\psi} \qquad\qquad \frac{\phi \equiv \psi \quad t \equiv u}{\phi(t) \equiv \psi(u)}$$

where x must not be free in ϕ or ψ in the abstraction rule.

A comparison of Isabelle and the Edinburgh Logical Framework LF can be found in [11].

All the above inference rules for the meta logic, except the α conversion rule, can be used manually by the user of Isabelle. Notice that the meta level connectives do not look like ordinary connectives – the usual connective symbols are left for the object logics. None of the meta logic inference rules will in this thesis be used directly for proving proofs. Each proof given in this thesis must be proved using only the inference rules and axioms specified in the logic with which a proof is proven.

6.3 Implementing a Theory

Theories are implemented in what can be named as an object logic. An object logic consists of an implementation of a formal theory and can also consist of a collection of theorems that follows immediately from the formal theory. Isabelle will treat object logics as plug-in modules such that Isabelle without a plug-in module only is able to prove propositions that follows immediately from the meta logic. In this paper Isabelle is not used without plug-in modules. Object logics can be stacked such that one logic is on top of other logics. Therefore theories can be used as basic libraries. e.g. an implementation of reasoning about natural numbers using natural deductive first order logic does not have to implement first order logic, it can be built on top of it.

Isabelle only helps proving proofs using an object logic. It does not in any way examine the object logic. Therefore, the reliability of an Isabelle proved theorem depends on the reliability of the object logic (and of course also the reliability of Isabelle, but it can be assumed that Isabelle is reliable).

Each of the theories in Isabelle is represented by

- a theory file `T.thy` defining the axioms and rules of theory $\mathcal{T}$ and
- an ML file `T.ML` claiming and solving the collection of theorems following from the theory.

The meta logic is represented as object logic `PURE`.

6.3.1 Object Logic

An object logic implementing a formal theory consists of connectives, well-formed formulas, axioms and inference rules. Predicate letters, function letters, constants and variables are pre-defined in Isabelle for use with Isabelle's meta logic.

The elements of the object logic are presented using Isabelle's external representation discussed in section 6.4.1.

6.3.2 Theorem Collection

The collection of theorems is intended to be a collection of often used results that can be derived from the object logic. The collection of theorems should be restricted in size since each theorem in the collection is proven when the theory is invoked. Each theorem of the collection should only be in the collection if it should be a part of the formal theory if not the theorem violates the theory's independency condition[3].

The theorem file is an ML file and can therefore define functions that may be convenient for proving proofs in the object logic. Such functions may be tactics and tacticals that are described in sections 6.5.3 and 6.5.4.

6.4 Theory and Proposition Representation in Isabelle

Isabelle uses two representation languages for theories and proofs. These representations are an external representation for representing and stating theories and propositions in the user interface and an internal representation for propositions.

6.4.1 External Representation

Theories and propositions to be proven in a theory are stated using Isabelle's external representation.

Letters and connectives of Isabelle's external representation are function symbols. These function symbols can either be static or dynamic, either being constants or variables. Connectives and pre-defined letters are always constants, and the rest are variables. Each function symbol has an arity, and constant function symbols (like "`0`" and "$\bot$") have arity zero.

Constants can only be defined in theories, not in propositions to be proven.

Isabelle variable letters are categorized as below.

Free variables are variables that, in propositions due for proving, in no way can be touched by the Isabelle proving system. A free variable V is denoted as `V`.

Bound variables are variables in an expression that are bound by some binder in that same expression such that Isabelle can do nothing about the variable except removing the binder if some rule of a theory allows it. A bound variable V in an expression with its binder (here: `#`) is denoted as `# V.` ...`V`....

Schematic variables are variables in a proposition that can be instantiated by Isabelle. A schematic variable V is denoted as `?V`.

Schematic variables must only be used in (meta logic) conclusions of theorems due for proving. Free variables in a proposition that is proven are converted to schematic variables when obtaining a theorem from the proposition. Therefore, free variables may be used for variables in theories and for variables that may not or cannot be instantiated in propositions due for proving. Schematic variables may be used for the rest of the unbound variables.

Inference rules, axioms and propositions can be in expressed in Isabelle's meta logic using the following rewriting rules from textual layout to meta logic constructions:

[3] A theory $\mathcal{T}$ is independent if no rule of $\mathcal{T}$ can be proved using the other rules in $\mathcal{T}$.

$\Longrightarrow$	$\mapsto$	==>
Λ	$\mapsto$	!!
$\equiv$	$\mapsto$	==
Π	$\mapsto$	%

A hypothesis H of an inference rule premise P in an inference rule

$$\dfrac{\begin{array}{c}[H]\\ \vdots \\ P\end{array}}{C}$$

is rewritten as

$$(H \texttt{ ==> } P) \texttt{ ==> } C$$

More than one premise in an inference rule

$$\dfrac{P_1 \quad \cdots \quad P_n}{C}$$

is represented as

$$P_1 \texttt{ ==> } (\ldots \texttt{ ==> } (P_n \texttt{ ==> } C)\ldots) \quad \text{or as short-hand} \quad \texttt{[|}P_1\texttt{,} \ldots \texttt{,}P_n\texttt{|] ==> } C$$

Types of Isabelle letters can be grouped in sorts, which are type classes. An Isabelle sort for types of letters that represent object logic variables (representing truth values that can be reasoned about) is `logic`. New sorts can be defined as part of existing sorts. Defining a sort for term letters (that cannot be reasoned about) and making this sort the default sort for type variables can be done as

```
classes term < logic
default term
```

Types can be defined with a specification of the types' sorts, i.e. defining a type `o` for object logic entities that can be reasoned about is done as

```
types o
arities o ::  {logic}
```

In a theory definition file the connectives of a logic are stated as object logic constants with their corresponding type, i.e. making a right associated infix connective ** with priority 10, corresponding to a connective ∘, taking two arguments of type `o` returning an argument of type `o` is done with

```
consts "**" ::  "[o, o] => o" (infixr 10)
```

Defining a quantifier quantifying a variable requires a connective stating the proposition over which the variable is quantified. A binder helping readability can then be defined, i.e. defining a quantifier `Some` with a binder `SOME` representing a quantifier $\triangle$ is done with

```
consts Some ::  "('a => o) => o" (binder "SOME " 10)
```

making expressions `Some(P)` and `SOME x. P(x)` equivalent.

Inference rules and axioms of a theory definition file are stated as object logic rules, i.e. stating an inference rule `circI` introducing connective `**` corresponding to an inference rule

$$\frac{P \quad Q}{P \circ Q}$$

can be done with

```
rules circI " [| P ; Q |] ==> P ** Q "
```

Isabelle can only handle meta logic propositions which are propositions of type `prop`. Therefore a coercion coercing object logic propositions of type `o` to meta logic propositions of type `prop` must be introduced,

```
consts Trueprop ::  o => prop ("_" 5)
```

Whether a formula is a well-formed formula or not is judged by Isabelle using the theory definition where the type family system plays a big role. With the above definitions i.e. a formula as

```
SOME x. (?P(x) ** x)
```

is not well-formed because `x` has type `o` by definition of connective `**` and also has type `'a` by definition of connective `SOME`. But `o` cannot be unified with type variable `'a` because `o` is of sort `logic` and `'a` is of default sort `term`, and these sorts are not unifiable.

An Isabelle theory definition file consists of the name of the theory, the names of the theories the new theory are based upon and new definitions followed by `end`, i.e.

```
T = Pure +

classes term < logic
default term

types o
arities o :: {logic}

consts
  Trueprop :: o => prop ("\_" 5)
  "**"     :: "[o, o] => o" (infixr 10)
  Some     :: "('a => o) => o" (binder "SOME " 10)

rules
  circI       " [| P ; Q |] ==> P ** Q "

end
```

The full syntax of Isabelle theory definitions, including the mixfix templates `(...)` used heavily above, can be found in [14, chapter 7] and a lot of examples can be found in appendix A.

6.4.2 Internal Representation

Internally Isabelle represents propositions using a datatype `term` (not to be mistaken with the above sort with the same name) representing the following symbols:

Constants are names of connectives and other predefined symbols, represented by a construction function `Const` with the constant's name and the Isabelle type of the constant as arguments.

Free variables are free variables that cannot be instantiated in the proving process, represented by a construction function `Free` with the variable's name and the Isabelle type of the variable as arguments.

Schematic variables are free variables that can be instantiated in the proving process, represented by a construction function `Var` with the variable's name and the Isabelle type of the variable name as arguments. Variable names for schematic variables consists of a name and an index.

Abstractions introduces variables over which a term is abstracted or quantified, represented by a construction function `Abs` with the variable's name, the Isabelle type of the variable and the term over which is abstracted.

Bound variables are variables introduced by abstractions and quantifications, represented by a construction function `Bound` with the variable's DeBruijn index as argument.

Eventually a term can be an application, represented by a construction function `$` with the applicant and the applicant's argument as arguments. Applications are combinations of the above symbols such that i.e. a connective (a constant symbol) can be applied on some variables.

Since bound variables in the above datatype are DeBruijn indexed no α conversion is needed. For a demonstration of the internal representation the proposition with external representation `SOME x. ?P(x) ** Q` is represented as

```
$
  (Const ("Trueprop","o => prop"),
   $
     (Const ("Some ","('a::term => o) => o"),
      Abs
        ("x","'a::term",
         $
           ($
              (Const ("op **","[o, o] => o"),
               $ (Var (("P",0),"'a::term => o"),Bound 0)),Free ("Q","o")))))
```

In the above example, the construction function `$` is nonfix to ease readability, normally it is infix.

A Isabelle user, using Isabelle only for proving proofs, will never get in contact with the internal proposition representation, but it can be necessary to study the internal structure of a proposition (either due for proving or a theorem that shall be used) to find reasons for implementational problems.

The internal structure of Isabelle propositions is presented in [14, chapter 6].

6.5 Proving Theorems

The theory part of this paper states that a theorem is a proven proposition. Propositions can be stated, and theorems extracted from proven propositions in Isabelle. Propositions can be solved and resolved using tactics.

6.5.1 Stating Propositions and Extracting Theorems

An object logic must be invoked before it can be used for proving propositions. Invocation of the object logic files for the formal theory $\mathcal{T}$ is done with the command

```
use_thy "T"
```

A proposition P due for proving in the formal theory $\mathcal{T}$ is in Isabelle stated using the `goal` command with the syntax

```
goal T.thy "P"
```

where `T` is the name of the formal theory $\mathcal{T}$ and `P` is the meta logic representation of proposition P. A goal stating does the following:

- creates an internal Isabelle state that can be reasoned about, and
- returns a list holding the hypotheses of the proposition, and
- pretty-prints the Isabelle state.

The state can be reasoned about using tactics.

As an example stating the goal

```
[| SOME x.  R(x); Q |] ==> (SOME x.  R(x)) ** Q
```

with the command

```
val [h1,h2] = goal T.thy "[| SOME x.  R(x); Q |] ==> (SOME x.  R(x)) ** Q"
```

returns the hypotheses

```
val h1 = "SOME x. R(x)  [SOME x. R(x)]" : thm
val h2 = "Q  [Q]" : thm
```

and pretty-prints the state

```
Level 0
(SOME x. R(x)) ** Q
 1. (SOME x. R(x)) ** Q
```

Extracting the theorem that will come out of the stated proposition with the unproven subgoals as (additional) hypotheses is done with the command `uresult`. Extracting the theorem from the above state with

```
val notProven = uresult();
```

gives the theorem

```
val notProven =
  "[| SOME x. ?R(x); ?Q; (SOME x. ?R(x)) ** ?Q |]
   ==> (SOME x. ?R(x)) ** ?Q" : thm
```

If a state has none unproven subgoals it represents a theorem. The theorem represented by the state is extracted with the command `result`. A proven state is pretty-printed as

```
Level ...
...
No subgoals!
```

The theorem in the above example is extracted with

```
val proven = result()
```

yielding

```
val proven = "[| SOME x. ?R(x); ?Q |] ==> (SOME x. ?R(x)) ** ?Q" : thm
```

6.5.2 Proving Subgoals

In Isabelle proofs are made by proving subgoals. A subgoal can be viewed as an unproven leaf of an inference tree. Stating a proposition due for proving creates a subgoal, the root leaf of an inference tree. Application of a tactic on subgoals refines the subgoals with the main intention to end up with subgoals that can be proved using axioms, hypotheses or theorems.

Some tactics are step-by-step tactics used for applying exactly one rule on a certain subgoal, either on the hypotheses of the subgoal or on the conclusion of the subgoal. Other tactics are more automatic and implement some proving strategies proving the subgoal on which the tactic is applied, optimally such that the subgoal is proven.

Step-by-step tactics can be used for human directed proving. Automata tactics are used for proving proofs without human participation in the proving process.

When an inference rule is applied on a leaf of a proof tree, the leaf becomes root in an inference tree having branches ending in leaves corresponding to the premises of the inference rule. Applying i.e. rule $\circ$I on leaf $P \circ Q$ in an inference tree

$$\dfrac{\begin{array}{ccc}[R] & & [S] \\ \vdots & & \vdots \\ R & P \circ Q & S\end{array}}{\ldots} \tag{6.1}$$

gives inference tree

$$\dfrac{\begin{array}{ccc}[R] & & [S] \\ \vdots & \dfrac{P \quad Q}{P \circ Q} & \vdots \\ R & & S\end{array}}{\ldots} \tag{6.2}$$

The parts of an inference tree that is interesting, when proving the root of the tree, are the unproved leaves. Read from left to right the sequence of unproven leaves of tree (6.1) is

$$\langle [R] \cdots R, P \circ Q, [S] \cdots S \rangle$$

and the sequence of unproven leaves of proof tree (6.2) is

$$\langle [R] \cdots R, P, Q, [S] \cdots S \rangle$$

Generally, applying an inference rule

$$\frac{P_1, \ldots, P_n}{C}$$

on leaf C in a sequence of m unproven leaves

$$\langle L_1, \ldots, L_{i-1}, C, L_{i+1}, \ldots, L_m \rangle$$

yields a new sequence of $m + n - 1$ unproven leaves where C is replaced with $P_1, \ldots, P_n$,

$$\langle L_1, \ldots, L_{i-1}, P_1, \ldots, P_n, L_{i+1}, \ldots, L_m \rangle$$

Notice that the conclusion of the inference rule and the leaf on which the inference rule is applied are equal. If these parts are not equal, they must be unified before the application.

Inference rules can also be applied on hypotheses of leaves of a proof tree following the above scheme. Hypotheses of a leaf occuring as premises of the inference rule are replaced with the conclusion of the inference rule.

Isabelle implements applying tactics on leaves of sequences of unproven proof tree leaves. Proving the Isabelle state representing proof tree (6.1)

```
Level 4
...
 1. R ==> R
 2. P ** Q
 3. S ==> S
val it = () : unit
```

with a tactic that applies rule `starI` on subgoal 2 results in a state

```
Level 5
...
 1. R ==> R
 2. P
 3. Q
 4. S ==> S
val it = () : unit
```

When a tactic proves a leaf of an inference tree the leaf is no longer unproven and thus dissapears from the sequence of unproven leaves. The proposition R in inference tree (6.2) can by proved by the discharged hypothesis $[R]$ of the proposition which results in the Isabelle state

```
Level 6
...
 1. P
 2. Q
 3. S ==> S
val it = () : unit
```

A tactic is applied on a state using the `by` command. Either it succeeds in applying the tactic whereby a new state is created or it fails to apply the tactic.

Applying a tactic on a state may result in more than one possible state. Applying a tactic corresponding to unifying the conclusion of an inference rule

$$\frac{\triangle x.P(x)}{P(y)} \tag{6.3}$$

with a certain subgoal corresponding to proposition `Q(z)` of a state may result in a state having subgoal `Q(z)` replaced by either subgoal `SOME x. Q(x)` or subgoal `SOME x. Q(z)` in the list of unproven subgoals. Only one of the possible states is returned which is the state where as much as possible is unified. Backtracking the unification resulting in other states can be done with the `back` command.

6.5.3 Isabelle Tactics

Most tactics are generated by tactic generators that generates a tactic given some rules as arguments. Many tactic generators also require the number of the subgoal on which the generated tactic must be applied. Among the step-by-step tactic generators implemented are the following generators

`resolve_tac` $[r_1, \ldots, r_m]$ n generates a tactic for resolving subgoal n with the first rule of the rules $r_1, \ldots, r_m$ which conclusion can be unified with subgoal n. If a unification succeeds, some variables in the conclusion may have been unified with other variables or instantiated to some value. Changed variables are updated and the premises of the inference rule (with the variables updated) will be new subgoals. A resolving gives zero or more subgoals instead of the old subgoal. The tactic generator is used for a short-hand tactic application defined by

$$\texttt{br}\ r\ n\ =\ \texttt{by (resolve\_tac [}r\texttt{]}\ n\texttt{)}$$

and also

$$\texttt{rtac}\ r\ n\ =\ \texttt{resolve\_tac [}r\texttt{]}\ n$$

`dresolve_tac` $[r_1, \ldots, r_m]$ n generates a tactic for destructing subgoal n with the first rule of the rules $r_1, \ldots, r_m$, the premises of which all can be unified with hypotheses of subgoal n. If a unification succeeds, some variables in the hypotheses may have been unified with other variables or instantiated to some value. Changed variables are updated and the conclusion of the inference rule (with the variables updated) will be new hypothesis of the subgoal instead of the hypotheses unified with the premises of the inference rule. The tactic generator is used for a short-hand tactic application defined by

$$\texttt{bd}\ r\ n\ =\ \texttt{by (dresolve\_tac [}r\texttt{]}\ n\texttt{)}$$

and also

$$\texttt{dtac}\ r\ n\ =\ \texttt{dresolve\_tac [}r\texttt{]}\ n$$

`assume_tac` n generates a tactic for proving subgoal n that succeeds if the conclusion of subgoal n can be unified with one of the hypotheses of the subgoal. The tactic generator is used for a short-hand tactic application defined by

$$\texttt{ba}\ n\ =\ \texttt{by (assume\_tac}\ n\texttt{)}$$

and also

$$\texttt{atac}\ n\ =\ \texttt{assume\_tac}\ n$$

`rewrite_goals_tac` $[r_1, \ldots, r_n]$ generates a tactic that rewrites all subgoals as specified in rules $r_1, \ldots, r_n$. The rewrite rules must have conclusions matching $L \equiv R$. Rewriting with a rule $L \equiv R$ replaces all occurrences of L with R in all subgoals. The tactic generator is used for a short-hand defined by

$$\texttt{rewtac}\ r\ =\ \texttt{rewrite\_goals\_tactic [}r\texttt{]}$$

`fold_goals_tac` $[r_1, \ldots, r_n]$ generates a tactic that rewrites all subgoals as specified in rules $r_1, \ldots, r_n$. The rewrite rules must have conclusions matching $L \equiv R$. Rewriting with a rule $L \equiv R$ replaces all occurrences of R with L in all subgoals. This is the reverse tactic to `rewrite_goals_tac`.

`rewrite_tactic` $[r_1, \ldots, r_n]$ acts like tactic generator `rewrite_goals_tac` but also rewrites the main goal.

`fold_tactic` $[r_1, \ldots, r_n]$ acts like tactic generator `fold_goals_tac` but also rewrites the main goal.

Two other tactics that at a glance may seem useless are tactics `all_tac` and `no_tac`, respectively always succeeding and always failing at application. These tactics are not useless at all since in combination with other tactics they serve as neutral elements.

A full description of all tactics and tactic generators implemented in Isabelle can be found in [14].

6.5.4 Isabelle Tacticals

Tactics and tactic generators can be combined giving added functionality using tacticals. Among the tacticals are

t_1 `THEN` t_2 combines tactics t_1 and t_2 such that application of tactic t_1 `THEN` t_2 on a state S is application of tactic t_2 on the state S_1 resulting of applying tactic t_1 on the state S. If the application of tactic t_1 on state S can return more than one state, the state S_1 is possibly backtracked until it succeeds applying t_2 on S_1. Tactic `all_tac` is neutral element for this tactical.

t_1 `ORELSE` t_2 creates a tactic that when applied on a state acts like tactic t_1 if it succeeds to apply tactic t_1 on the state, or else acts like tactic t_2. Tactic `no_tac` is neutral element for this tactical.

`REPEAT` t repeats application of tactic t on states. `REPEAT` t applied on a state S is, if S_1 is the result of a succeeding application of tactic t on state S, the result of tactic `REPEAT` t applied on the state S_1. If application of tactic t on state S fails, the result of applying `REPEAT` t on state S is state S.

tg_1 `THEN'` tg_2 combines tactic generators tg_1 and tg_2, generating tactics when applied on a subgoal number, such that for a subgoal number n, tactic

$$(tg_1 \texttt{ THEN' } tg_2)\ n$$

is equal to tactic

$$(tg_1\ n) \texttt{ THEN } (tg_2\ n)$$

tg_1 `ORELSE'` tg_2 combines tactic generators tg_1 and tg_2, generating tactics when applied on a subgoal number, such that for a subgoal number n, tactic

$$(tg_1 \texttt{ ORELSE' } tg_2)\ n$$

is equal to tactic

$$(tg_1\ n) \texttt{ ORELSE } (tg_2\ n)$$

`FIRSTGOAL` tg, where tg is a tactic generator that generates a tactic when applied on a subgoal number, is the tactic corresponding to

$$(tg\ 1)\ \texttt{ORELSE}\ \ldots\ \texttt{ORELSE}\ (tg\ n)$$

where n is the number of subgoals in the state on which tactic `FIRSTGOAL` tg is applied.

`SOMEGOAL` tg, where tg is a tactic generator that generates a tactic when applied on a subgoal number, is the tactic corresponding to

$$(tg\ n)\ \texttt{ORELSE}\ \ldots\ \texttt{ORELSE}\ (tg\ 1)$$

where n is the number of subgoals in the state on which tactic `SOMEGOAL` tg is applied.

All the Isabelle tacticals are described in [14, chapter 4].

6.5.5 Rule Combinators

Rules that can be applied on a subgoal can be combined using rule combinators just like tactics can be combined using tacticals. Some rules may return many possible states when applied on a state which may end up in a lot of state backtracking. This can be avoided by resolving the premises of one rule with other rules' conclusions.

As an example, let proposition $z = z$ first be resolved with inference rule (6.3) and the result hereof be resolved with axiom

$$\frac{}{\triangle v.v = v} \tag{6.4}$$

Resolving proposition $z = z$ with rule (6.3) may yield four different propositions as possible premises for the inference rule, namely $\triangle x.x = x$, $\triangle x.x = z$, $\triangle x.z = x$ and $\triangle x.z = z$, but only one of these propositions can be resolved with rule (6.4). By combining rules (6.3, 6.4) before application on proposition $z = z$ to a rule

$$\frac{}{y = y}$$

corresponding to the proof

$$\frac{\dfrac{}{\triangle x.x = x}}{y = y}$$

a possible backtracking can be avoided.

Isabelle has among others the following rule combinators

r_1 `RSN` (i, r_2) where r_1 corresponds to a rule

$$\frac{Q_1 \quad \cdots \quad Q_m}{D}$$

and r_2 corresponds to a rule

$$\frac{P_1 \quad \cdots \quad P_i \quad D \quad P_{i+1} \quad \cdots \quad P_n}{C}$$

resolves premise D of rule r_2 with the conclusion D of rule r_1 and returns the resolving result as the rule

$$\frac{P_1 \quad \cdots \quad P_i \quad Q_1 \quad \cdots \quad Q_m \quad P_{i+1} \quad \cdots \quad P_n}{C}$$

r_1 `RS` r_2 abbreviates r_1 `RSN` $(1, r_2)$.

Several combinators combining more than two rules can be created using combinator `RSN`. Some of these are described in [14, chapter 5.1].

6.6 Examples

Object logics representing all formal theories presented in the theory part of this thesis have been implemented. The implementations are presented in appendix A. They are exhausted and demonstrated in appendix B. The implemented object logics are connected as shown on figures 6.2 and 6.1.

Figure 6.1 Inheritance between object logics reasoning about first order logic proofs. In the tree an arrow from `A` to `B` means that object logic `A` is a part of object logic `B`.

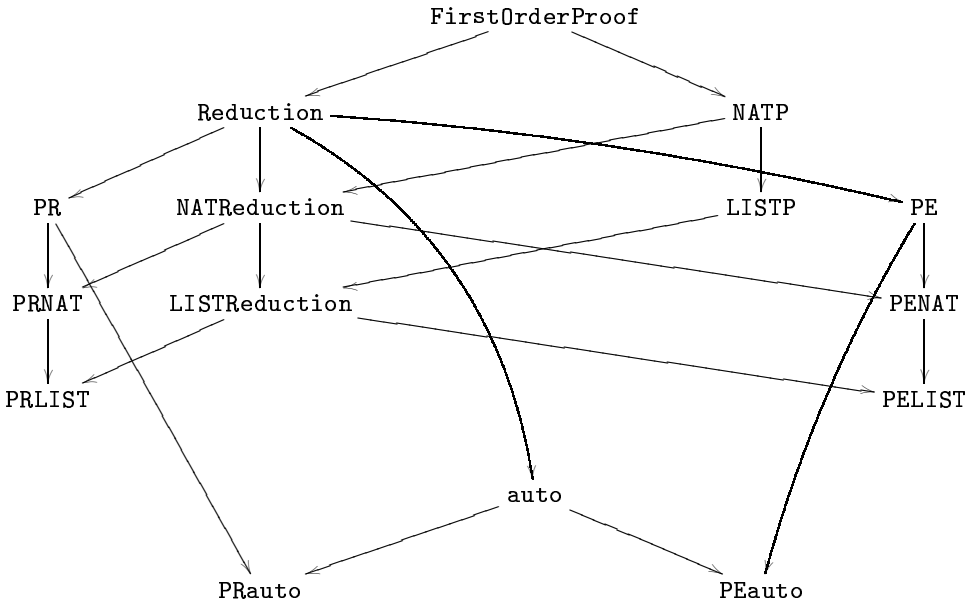
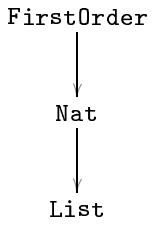


Figure 6.2 Inheritance between object logics reasoning about first order logic. In the tree an arrow from `A` to `B` means that object logic `A` is a part of object logic `B`.



6.7 Summary

This chapter discussed why to choose Isabelle for representing formal theories. Isabelle syntax, including the meta logic, and how to use Isabelle was presented.

6.7.1 Directions

Other proof search systems not studied yet could be examined to find out if some of them are better than Isabelle. Isabelle was chosen since it fulfils my requirements for a proof search system, not because it was the best of all possible (existing) proof search systems. This software studying is not a part of this thesis.

Experiments with Isabelle can be done. This is done in appendices A and B. Isabelle is also used for an example on proof reduction and program evaluation in chapter 8.

6.7.2 Literature

An argumentation of why to create a proof search system like Isabelle is presented in [11]. The language ML is described in [12]. The manuals for Isabelle are [13, 14, 15].

Chapter 7

Implementing the Normalizer Generator

Reduction of proofs were discussed in chapter 4 where formal theories for normalization of proofs were created. This chapter discusses how to automate the reduction process.

The normalization of proof terms as specified in object logics `PR` and `PE` may be automated such that the application of a tactic on a reducible subgoal results in a normalization of the subgoal. A tactic solving a state completely is what is sought.

This chapter first presents the actions of a normalizer tactic, an algorithm based here on, eventually the interface for a normalizer tactic generator is presented.

7.1 Actions of a Normalizer

The action of a normalizer is to apply some tactic on a subgoal of an Isabelle state such that the subgoal is reduced. The normalizer must itself decide which tactic to use. Tactics used by a normalizer depends on the inference rules and axioms used for reduction.

Definition 7.1.1 *A normalizer tactic and the tactics used by the normalizer must fulfil some requirements which are:*

- *The actions performed by a normalizer must be traceable.*
- *The tactics used by the normalizer must be specified such that it always is the correct tactic that is applied on a subgoal, based on some definition of correctness.*
- *The normalizer must always choose the correct subgoal to normalize, based on some definition of correctness.*
- *When specifying a new object logic for reduction on top of another object logic for which a normalizer is created, it must be possible to obtain a normalizer for the new object logic by adding extra tactics to the existing normalizer.*

The requirements of the definition are discussed below.

7.1.1 Tracing

The reason for implementing tracing is that it helps inspecting whether or not a normalizer implements the intended normalization. As well, it can be interesting to supervise how the normalization takes place[1]. Moreover it can be convenient to refine a subgoal or backtrack some normalization step in the implementation phase of defining an object logic for normalization.

Tracing can be implemented by creating a normalizer that applies exactly one tactic doing one proof step on exactly one subgoal exactly once.

7.1.2 Solving a Subgoal

If a normalizer must choose the correct tactic to apply on a given subgoal, the normalizer must either implement a clever algorithm for choosing the correct tactic or else have nothing but one rule to choose from. A good solution may be the golden mean, a dumb algorithm choosing from few rules.

The golden mean algorithm is to try to apply each tactic in a prearranged order and if the application succeeds, the tactic is the correct tactic. The golden mean algorithm is dumb since it does not inspect the tactics but just applies them. There may be a few tactics to choose among since the chosen tactic is the first usable tactic, not the only one usable tactic.

If more than one tactic can be used to solve a certain subgoal, then these tactics must be ordered such that the most general of the tactics is specified to be tried after the less general tactics, and the less general tactics must be ordered in the same way.

7.1.3 Choosing Subgoal to Normalize

Since tactics are created upon inference rules and axioms, the tactics can be grouped by these rules. The rules give two classes of tactics:

76_{114} tactics that *do not* instantiate predicate letters, and

77_{114} tactics that *do* instantiate predicate letters.

By constructing some inference rules such that unification of premises and conclusions is delayed, most rules belong to class 76_{114}. Only axioms cannot be constructed such that they do not instantiate variables. But this can be handled by turning them into inference rules.

Consequence 7.1.2 *An axiom is turned into an inference rule by resolving the axiom with inference rule*

$$\frac{x \longmapsto x : P \qquad \begin{array}{c}[y:P]\\ \Lambda y \quad \vdots \\ y : R\end{array}}{x \longmapsto x : R}\ \textit{Delay} \tag{7.1}$$

[1]The supervising possibility opens up for using the normalizer as an educational tool showing students how a reduction following a decent strategy takes place without making errors at the blackboard. But that is not important in this thesis.

As an example, axiom (4.24) gives inference rule

$$\frac{\begin{array}{c}[y : a = a]\\ \Lambda y \quad \vdots \\ y : R\end{array}}{\texttt{ideq}(a) \longmapsto \texttt{ideq}(a) : R}$$

With the conversion of axioms into inference rules, only applying hypotheses belong to class 77_{114}.

The above discussion gives the result that if a subgoal cannot be resolved by using a tactic applying an inference rule, the subgoal can be assumed correct by hypothesis. Since solving by assumption does not create new subgoals, solving by assumption is the last thing to do in a proving session, and thereby the last thing to do for a normalizer.

7.1.4 Adding Rules

Object logics can include other object logics. Therefore, a normalizer may be able to include other object logics. Object logic `Pure` includes only the meta logic, a meta normalizer may only include tactics that can be constructed using the meta logic.

When a normalizer includes another normalizer, tactics for inference rules and axioms are possibly added. The newly added tactics must fit in the order of tactics in the included normalizer. Since an object logic including other object logics seldom define new rules for old constructors, it seldom will be a problem to just bring the added tactics in order and let them be tried before the tactics of the included normalizer.

7.2 Algorithm Implementing Strategy

The above discussion of the actions of a normalizer gives rise to an algorithm.

Definition 7.2.1 *An algorithm for a normalizer is:*

78_{115} *repeatedly resolve the first possible subgoal with the first possible of the tactics, and when no subgoals can be solved with the tactics*

79_{115} *solve the rest of the subgoals by assumption.*

The inference rules used for tactics in 78_{115} must be specified in an ordered manner reflecting the tactic ordering.

7.3 Order Of Tactics Using Theories for First Order Logic Reduction

With the context rules in section 4.7, if in a subgoal $p \longmapsto q : P$ proof p does not contain schematic variables, then only one context rule can be used for resolving the subgoal. As well, for any reduction rules of section 4.7, if in a subgoal $p \longmapsto_! q : P$ proof p does not contain schematic

variables, then only one reduction rule can be used for resolving the subgoal, if it does not contain a redex, else also one redex reduction rule can resolve the subgoal.

Since the normalizer is always applied on the first possible subgoal, it must be assured that this subgoal has no schematic variables in the left proof part. This can be done by ordering tactics such that

- Context rules are tried first, since all context rules create only subgoals, on which context rules can be applied, without schematic variables in the left proof. A tactic for handling context subgoals on which no context rule can be used is regarded as a context rule. (Examples here on are `vartac` and `try_allRules`).
- Reduction rules are tried last. Subgoals on which a reduction rule can be applied are constructed with schematic variables that are instantiated by solving subgoals with context rules. With no subgoals on which context rules can be applied, the first reduction subgoal has no schematic variables in the left proof.

7.4 Interface for Normalizer

The user must have the following possibilities with the normalizer:

- Adding context inference rules in front of the ordered set of context inference rules for which the normalizer create tactics.
- Adding context axioms in front of the ordered set of context axioms for which the normalizer create inference rules and hereof tactics.
- Specifying which tactic to use for solving context subgoals for which no context rule is defined.
- Adding reduction rules in front of the ordered set of reduction rules for which the normalizer create tactics.
- Resetting the set of tactics used by the normalizer.
- Retrieving a normalizer that uses the stated set of rules.

7.5 Implemented Normalizers

A meta normalizer can be found in appendix A.7. A normalizer for object logic `PR` can be found in appendix A.8 and a normalizer for object logic `PE` can be found in appendix A.9. These normalizers are tested in appendices B.13 and B.14. In section 8.7, two normalizers are created for reducing proofs of all proof reduction object logics presented in this thesis, respectively evaluating programs of all program evaluation object logics presented in this thesis.

It seems, based on section 8.7 and the exhaustive tests of basic normalizers in appendices B.13 and B.14, that the implemented normalizers probably not contain harmful errors.

Due to lack of time, normalizers for natural numbers and lists have not been created. Also, in chapter 8 the structure that one object logic introduces one layer of constructors is ruined by creating a normalizer for normalizing in all object logics at the same time.

7.6 Summary

This chapter defined how a normalizer works. It must be traceable. It solves a subgoal by applying tactics to the subgoal in a decent order until one of the applications succeeds. It decides which subgoal to normalize by first applying tactics that do not instantiate variables to all subgoals, and if no subgoals can be solved hereby, tactics instantiating variables are tried.

Also, this chapter described the interface for a normalizer. The interface was specified such that modularity in object logic construction can be preserved.

7.6.1 Directions

A meta normalizer may be implemented. This is done in appendix A.7. Instances hereof may be created for object logics representing the formal theories presented in this thesis, which can be found in appendices A.8 and A.9. Experimenting with the normalizer is done in the case study and in appendices B.13 and B.14.

7.6.2 Literature

No literature has been used for this chapter.

Chapter 8

Case Study: Insertion Sort

Insertion sort is an algorithm for sorting lists of values on which an ordering predicate exists. The insertion sort algorithm is used as an example of a sorting algorithm at mostly all basic computer science courses teaching functional programming.

This chapter documents a study on how to create the insertion sort algorithm as a proof of a proposition and use the proof as a program to sort lists of natural numbers. Hereby this chapter contains the following:

- A presentation of a formal theory describing properties of list permutation and sorted lists. By using this formal theory a proposition describing insertion sort is created.
- Implementation of an object logic corresponding to the formal theory describing list permutation and sorted lists.
- Documentation of a proof of the insertion sort proposition using the object logic describing list permutation and sorted lists.
- Extending the formal theory for list permutation and sorted lists with proof terms for creating a linearized proof of the insertion sort proposition.
- Implementation of an object logic corresponding to the formal theory describing linearized proofs with list permutation and sorted lists.
- Documentation of a proof of the insertion sort proposition using the object logic describing linearized proofs with list permutation and sorted lists.
- Reduction rules for list permutations and sorted lists are created. Formal theories expressing proof reduction and program evaluation including list permutations and sorted lists are created and object logics representing these formal theories are implemented.
- The proof for insertion sort is normalized by reducing it to head normal form using proof reduction, and by reducing it to weak head normal form using program evaluation.
- The normalized proofs for insertion sort is applied on an unsorted list of some values. The resulting subgoal is then normalized. For values and ordering relation a formal theory implementing $(\mathbb{N}_0, <_{\mathbb{N}_0})$ is used.

Komagata and Schmidt have made a report [8] on how to extract programs from proofs. They use insertion sort as a case for studying the extraction. Their report contains among other things a

proof of a proposition declaring insertion sort. Their basic ideas are used to create the rules of a formal theory describing sorting and the proposition on insertion sort. Komagata and Schmidt's proof sketch for their insertion sort proposition is used as an idea for how a proof of the insertion sort proposition can be done.

There will be two interesting outcomes of this case study:

80$_{120}$ Will the proof reduced insertion sort proof applied on a list of natural numbers by proof reducing result in a sorted list?

81$_{120}$ Will the program evaluated insertion sort proof applied on a list of natural numbers by program evaluation result in a sorted list?

If the questions both are answered positively it would be interesting to study which of the reductions that includes the smallest number of tactic applications to get an idea on whether proof reduction could be used instead of program evaluation on run-time. Proof reduced proofs have the same number or fewer redexes than program evaluated proofs so it may be a fortune to proof reduce a proof before application.

The questions are answered in sections 8.6.2 and 8.6.3.

8.1 Insertion Sort

Insertion sort is a method for sorting a list of values with some ordering relation. The ordering relation has to be total and applicable to the values of the list, else the result of applying insertion sort on a list will probably not be a sorted list. The average time complexity of insertion sort is $\Theta(n^2)$ for a list with n elements, whereby it is not often used for other purposes than examples.

8.1.1 Insertion Sort Algorithm

Definition 8.1.1 *The insertion sort algorithm is recursive and defined by two cases which are*

Stop case: *insertion sort on an empty list is the empty list.*

Recursion case: *insertion sort on a non-empty list is to insert the first element of the list into the result of applying insertion sort on the tail of the list, such that the inserting results in a list that is sorted.*

The cases are derived from the version of insertion sort presented in [12, p. 97].

If some proposition for the above algorithm may be created, the algorithm has to be expressed in a way that can be implemented using intuitionistic first order logic with datatype extensions.

Consequence 8.1.2 *The insertion sort algorithm can be formulated as:*

> *There is a permutation of every list which is sorted with respect to some ordering relation; for every sorted list and every element there is a sorted list which is a permutation of the prior sorted list extended with the element.*

The definition presents a statement of *what* can be sorted and a *constructive description* on how to create a sorted list. The mentioned ordering relation has to support the list's elements.

8.1.2 Formal Description of Insertion Sort

Definition 8.1.2 uses some terms not supported by any of the until now presented formal theories. These terms, "permutation" and "sorted", have to be implemented in a theory before any proposition using the terms can be proved. Let $\mathcal{P}$ and $\mathcal{S}$ be reserved predicate letters such that $\mathcal{P}$ denotes permutation and $\mathcal{S}_\prec$ denotes sorting using some ordering relation $\prec$.

Definition 8.1.3 *Definition 8.1.2 of sorting and how to construct a sorted list defines propositions*

$$\forall l \exists m.\ \mathcal{S}_\prec(m) \wedge \mathcal{P}(m, l) \tag{8.1}$$

$$\forall l.\ \mathcal{S}_\prec(l) \rightarrow (\forall a \exists m.\ \mathcal{S}_\prec(m) \wedge \mathcal{P}(\mathsf{C}(a, l), m) \tag{8.2}$$

Proposition (8.1) denotes *what* must be proved and proposition (8.2) denotes *how* it must be done for one non-trivial case. The proposition for the non-trivial case covers all constructed lists. Sorting the empty list is a question of definition of sorting and permutation.

8.2 Formal Theory on Sorted Lists

The above used predicate letters $\mathcal{P}$, $\mathcal{S}$ and $\prec$ are defined below. These are used for defining a formal theory.

8.2.1 Permutation Predicate $\mathcal{P}$

The permutation predicate $\mathcal{P}$ is a binary equivalence relation.

Definition 8.2.1 *Two lists are permutations of each other if the lists contain the same elements. Let l_1 and l_2 be lists. If*

$$\forall x \in l_1 : x \in l_2 \wedge \forall x \in l_2 : x \in l_1$$

then $\mathcal{P}(l_1, l_2)$

To fulfil the intentions the predicate $\mathcal{P}$ must be reflexive, symmetric and transitive. Moreover $\mathcal{P}$ must fulfil that

- $\mathcal{P}$ **is injective:** Given lists l, m and some element a, if l and m are permutations of each other, then l extended with a as first element is a permutation of m extended with a as first element.
- $\mathcal{P}$ **is associative:** If l is a list with at least two elements, then l with the two first elements swapped around is a permutation of l.

8.2.2 Sorting Predicate $\mathcal{S}$

The sorting predicate $\mathcal{S}_\prec$ is an unary predicate on a list.

Definition 8.2.2 $\mathcal{S}_\prec(m)$ *defines that the list m is ordered by the ordering relation $\prec$,*

$$\mathcal{S}_\prec(\mathsf{C}(x_1, \mathsf{C}(x_2, \mathsf{C}(x_3, \ldots \mathsf{C}(x_n, \mathsf{Nil}) \ldots)))) \quad \Longleftrightarrow \quad x_1 \prec x_2 \prec x_3 \prec \ldots \prec x_n$$

The sorting predicate $\mathcal{S}_\prec$ is defined such that the empty list and all singleton lists are sorted. Moreover, if a list with at least one element is sorted, then the list is still sorted if the head element is removed, i.e. $\mathcal{S}_\prec$ is surjective.

The description of a sorting construction depends on the chosen construction algorithm. The best way to express constructing a sorted list based on a list of unsorted elements is, for insertion sort, to insert the elements one by one into a sorted list.

Definition 8.2.3 *Based on a definition of "smaller" a sorted list is defined as*

> *To obtain a sorted list m, given an element a and a sorted list l, if a is smaller than the head element of l then m is l with a added as head element, else m is the head element of l added on a sorted permutation of the list consisting the tail of l with a added as head element.*
>
> (8.3)

The above definition of inserting an element into a sorted list is in fact the inserting function of insertion sort which insertion sort folds over the list to be sorted.

8.2.3 Ordering Relation $\prec$

An ordering relation is often written as an infix $\prec$, which is a binary relation

$$\prec : (\alpha, \alpha) \to bool$$

where α is some type the ordering relation supports, such that $\prec$ supports the laws of orders, i.e. $\prec$ is:

reflexive: $\forall x \in \alpha : x \prec x$

antisymmetrical: $\forall x, y \in \alpha : x \prec y \wedge y \prec x \Rightarrow x = y$

transitive: $\forall x, y, z \in \alpha : x \prec y \wedge y \prec z \Rightarrow x \prec z$

To be able to sort any list with elements of type α with the ordering relation $\prec$, the ordering relation $\prec$ has to be

total: $\forall x, y \in \alpha : x \prec y \vee y \prec x$

If $\prec$ is not total it is not possible to decide for any pair of two different elements which one of the elements is "the first one". I.e., $\leq_{\mathbb{N}_0}$ is a total order, while $|_{\mathbb{N}_0}$ and $\subseteq_{\mathcal{P}(M)}$ (where $\mathcal{P}(M)$ is the powerset of a set M) are not total orders [2, p. 1.4].

The ordering relation is used to decide whether or not one element precedes another element, the ordering relation must therefore be decidable.

decidability: $\forall x, y \in \alpha : x \prec y \vee x \not\prec y$

Notice that the above decidability is a special instance of the classical rule. The formal theory $\mathcal{IFOL}$ for first order logic is intuitioninstic and does therefore not support decideability.

Consequence 8.2.4 *For an ordering relation $\prec$ if $a \prec b$ then a is said to be smaller than b.*

8.2.4 Formal Theory $\mathcal{SL}$ on Sorted Lists

A formal theory $\mathcal{SL}$ for permutation and sorting on lists may be based upon formal theory $\mathcal{LIST}$ defining lists of elements of some type upon natural deductive intuitionistic first order logic with equality.

Definition 8.2.5 *Formal theory $\mathcal{SL}$ is defined as formal theory $\mathcal{LIST}$ extended with the following*

Symbols *are extended with reserved predicate letters $\mathcal{P}$ of arity 2, $\mathcal{S}_\prec$ of arity 1 and $\prec$ of arity 2.*

Axioms *are extended with the axioms of figure 8.1.*

Inference Rules *are extended with the inference rules of figure 8.1.*

Figure 8.1 Rules for the predicates $\mathcal{P}$, $\mathcal{S}$ and $\prec$ extending the formal theory $\mathcal{LIST}$ to the formal theory $\mathcal{SL}$.

$$\frac{}{\mathcal{P}(l,l)}\ \mathcal{P}_{\mathsf{refl}} \qquad \frac{\mathcal{S}_\prec(\mathsf{C}(a,l))}{\mathcal{S}_\prec(l)}\ \mathcal{S}_{\mathsf{inj}}$$

$$\frac{\mathcal{P}(l,m)}{\mathcal{P}(m,l)}\ \mathcal{P}_{\mathsf{sym}} \qquad \frac{\mathcal{S}_\prec(\mathsf{C}(a,l)) \quad b \prec a}{\mathcal{S}_\prec(\mathsf{C}(b,\mathsf{C}(a,l)))}\ \mathcal{S}_{\mathsf{add}_\prec}$$

$$\frac{\mathcal{P}(l,m) \quad \mathcal{P}(m,n)}{\mathcal{P}(l,n)}\ \mathcal{P}_{\mathsf{trans}} \qquad \frac{}{x \prec x}\ \prec_{\mathsf{refl}}$$

$$\frac{\mathcal{P}(l,m)}{\mathcal{P}(\mathsf{C}(a,l),\mathsf{C}(a,m))}\ \mathcal{P}_{\mathsf{inj}} \qquad \frac{x \prec y \quad y \prec x}{x = y}\ \prec_{\mathsf{antisym}}$$

$$\frac{}{\mathcal{P}(\mathsf{C}(a,\mathsf{C}(b,l)),\mathsf{C}(b,\mathsf{C}(a,l)))}\ \mathcal{P}_{\mathsf{assoc}} \qquad \frac{x \prec y \quad y \prec z}{x \prec z}\ \prec_{\mathsf{trans}}$$

$$\frac{}{\mathcal{S}_\prec(\mathsf{Nil})}\ \mathcal{S}_{\mathsf{nil}} \qquad \frac{}{x \prec y \vee y \prec x}\ \prec_{\mathsf{total}}$$

$$\frac{}{\mathcal{S}_\prec(\mathsf{C}(a,\mathsf{Nil}))}\ \mathcal{S}_{\mathsf{single}} \qquad \frac{}{x \prec y \vee (x \prec y \to \bot)}\ \prec_{\mathsf{decideability}}$$

$$\frac{\mathcal{S}_\prec(\mathsf{C}(a,l)) \quad b \prec a \to \bot \quad \mathcal{P}(\mathsf{C}(b,l),m) \quad \mathcal{S}_\prec(m)}{\mathcal{S}_\prec(\mathsf{C}(a,m))}\ \mathcal{S}_{\mathsf{add}_\not\prec}$$

The rules for predicates $\mathcal{S}$ and $\mathcal{P}$ in the above definition are more or less adobted from [8, page 65].

Notice that the rules for predicate letter "$\prec$" does *not* make it possible to derive wether or not a relation $x \prec y$ is true for other cases than $x \prec x$. Rules specifying the relationship between specific elements of a certain type may be specified in a formal theory defining the relationship for the certain type.

8.2.5 Object logic representing formal theory $\mathcal{SL}$

The object logic for the formal theory $\mathcal{SL}$ is called `Insort` and is placed in the files `Insort.thy` and `Insort.ML`. The theory in the files does not implement exactly the same rules as specified in

$\mathcal{SL}$ since redundancy is eliminated. The differences are that

- rule $\mathcal{P}_{\mathsf{refl}}$ is proved by the $\mathcal{P}_{\mathsf{inj}}$ rule and axiom $\mathcal{P}_{\mathsf{Nil}}$ proposing $\mathcal{P}(\mathsf{Nil}, \mathsf{Nil})$,
- rule $\mathcal{S}_{\mathsf{Nil}}$ is proved by rule $\mathcal{S}_{\mathsf{inj}}$ and axiom $\mathcal{S}_{\mathsf{single}}$.

Theory File `Insort.thy`

```
(* Time-stamp: <1999-01-27 16:23:24 kokdg> *)

Insort = List +

consts Pe    ::  "['a list, 'a list] => o"
       So    ::  "'a list => o"
       "-<"  ::  "['a, 'a] => o" (infixl 50)

rules  PeNil     "Pe(l,l)"
       PeSym     "Pe(l,m) ==> Pe(m,l)"
       PeTrans   "[| Pe(l,m); Pe(m,n) |] ==> Pe(l,n)"
       PeInj     "Pe(l,m) ==> Pe(C(a,l), C(a,m))"
       PeAssoc   "Pe(C(a,C(b,l)), C(b,C(a,l)))"

       SoSingle  "So(C(a,Nil))"
       SoInj     "So(C(a,l)) ==> So(l)"
       SoAddLe   "[| So(C(a,l)); b -< a |] ==> So(C(b,C(a,l)))"
       SoAddNLe  "[| So(C(a,l)); (b -< a) --> False; Pe(C(b,l),m);
                     So(m) |] ==> So(C(a,m))"

       OrdRefl   "x -< x"
       OrdAnSym  "[| x -< y; y -< x |] ==> x = y"
       OrdTrans  "[| x -< y; y -< z |] ==> x -< z"
       OrdTotal  "x -< y | y -< x"
       OrdDecide "x -< y | (x -< y --> False)"

end
```

Theory File `Insort.ML`

```
(* Time-stamp: <1999-01-28 13:33:51 kokdg> *)

open Insort;

  goal Insort.thy "Pe(?l, ?l)";
  br allE 1;
  ba 2;
  br induction 1;
  br PeNil 1;
  br allI 1;
  br allI 1;
  br impI 1;
  br PeInj 1;
  ba 1;
```

```
val PeRefl = result();

  goal Insort.thy "So(Nil)";
  br SoInj 1;
  br SoSingle 1;
val SoNil = result();
```

8.3 Proving the Sorting Propositions

The propositions (8.1) and (8.2) describing sorting and insertion sort are proved below. Ideas for the proofs are in [8, page 65 and 73], and these ideas will be followed.

The inserting proposition (8.2) is proved first since it does only use the rules for $\mathcal{P}$ and $\mathcal{S}$ where the sorting proposition (8.1) uses the proof for the inserting proposition.

The proof of proposition (8.2) is by induction on the structure of the list l and is as follows

```
- goal Insort.thy "ALL l. (So(l) --> \
 \ (ALL a . EX m . So(m) & Pe(C(a,l), m)))";
= Level 0
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. ALL l.
       So(l) -->
       (ALL a. EX m. So(m) & Pe(C(a, l), m))
val it = [] : thm list
- br induction 1;
Level 1
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. So(Nil) -->
    (ALL a. EX m. So(m) & Pe(C(a, Nil), m))
 2. ALL x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br impI 1;
Level 2
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. So(Nil) ==>
    ALL a. EX m. So(m) & Pe(C(a, Nil), m)
 2. ALL x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br allI 1;
Level 3
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!a. So(Nil) ==>
         EX m. So(m) & Pe(C(a, Nil), m)
 2. ALL x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br exI 1;
Level 4
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!a. So(Nil) ==>
         So(?m3(a)) & Pe(C(a, Nil), ?m3(a))
 2. ALL x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br conjI 1;
Level 5
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!a. So(Nil) ==> So(?m3(a))
 2. !!a. So(Nil) ==> Pe(C(a, Nil), ?m3(a))
 3. ALL x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br PeRefl 2;
Level 6
```

```
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!a. So(Nil) ==> So(C(a, Nil))
 2. ALL x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br SoSingle 1;
Level 7
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. ALL x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br allI 1;
Level 8
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x. ALL xs.
            (So(xs) -->
             (ALL a.
                 EX m.
                    So(m) &
                    Pe(C(a, xs), m))) -->
            So(C(x, xs)) -->
            (ALL a.
                EX m.
                   So(m) &
                   Pe(C(a, C(x, xs)), m))
val it = () : unit
- br allI 1;
Level 9
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs.
       (So(xs) -->
        (ALL a.
            EX m. So(m) & Pe(C(a, xs), m))) -->
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br impI 1;
Level 10
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs.
       So(xs) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, xs), m)) ==>
       So(C(x, xs)) -->
       (ALL a.
           EX m. So(m) & Pe(C(a, C(x, xs)), m))
val it = () : unit
- br impI 1;
Level 11
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs)) |] ==>
       ALL a.
          EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br allI 1;
Level 12
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs)) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br disjE 1;
Level 13
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs)) |] ==>
       ?P12(x, xs, a) | ?Q12(x, xs, a)
 2. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs)); ?P12(x, xs, a) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
 3. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs)); ?Q12(x, xs, a) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br OrdDecide 1;
Level 14
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
 2. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
```

```
          So(C(x, xs));
          ?x13(x, xs, a) -< ?y13(x, xs, a) -->
          False |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br exI 1;
Level 15
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       So(?m14(x, xs, a)) &
       Pe(C(a, C(x, xs)), ?m14(x, xs, a))
 2. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -< ?y13(x, xs, a) -->
          False |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br conjI 1;
Level 16
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       So(?m14(x, xs, a))
 2. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       Pe(C(a, C(x, xs)), ?m14(x, xs, a))
 3. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -< ?y13(x, xs, a) -->
          False |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br SoAddLe 1;
Level 17
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       So(C(?a16(x, xs, a), ?l16(x, xs, a)))
 2. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       ?b16(x, xs, a) -< ?a16(x, xs, a)
 3. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       Pe(C(a, C(x, xs)),
          C(?b16(x, xs, a),
            C(?a16(x, xs, a), ?l16(x, xs, a))))
 4. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -< ?y13(x, xs, a) -->
          False |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- ba 2;
Level 18
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       So(C(?y13(x, xs, a), ?l16(x, xs, a)))
 2. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -<
          ?y13(x, xs, a) |] ==>
       Pe(C(a, C(x, xs)),
          C(?x13(x, xs, a),
            C(?y13(x, xs, a), ?l16(x, xs, a))))
 3. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -< ?y13(x, xs, a) -->
          False |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- ba 1;
Level 19
ALL l.
   So(l) -->
```

```
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -< x |] ==>
       Pe(C(a, C(x, xs)),
          C(?x13(x, xs, a), C(x, xs)))
 2. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs));
          ?x13(x, xs, a) -< x --> False |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br PeRefl 1;
Level 20
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(xs) -->
          (ALL a.
              EX m. So(m) & Pe(C(a, xs), m));
          So(C(x, xs)); a -< x --> False |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- bd mp 1;
Level 21
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(C(x, xs)); a -< x --> False |] ==>
       So(xs)
 2. !!x xs a.
       [| So(C(x, xs)); a -< x --> False;
          ALL a.
             EX m.
                So(m) & Pe(C(a, xs), m) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br SoInj 1;
Level 22
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(C(x, xs)); a -< x --> False |] ==>
       So(C(?a19(x, xs, a), xs))
 2. !!x xs a.
       [| So(C(x, xs)); a -< x --> False;
          ALL a.
             EX m.
                So(m) & Pe(C(a, xs), m) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- ba 1;
Level 23
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(C(x, xs)); a -< x --> False;
          ALL a.
             EX m.
                So(m) & Pe(C(a, xs), m) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br allE 1;
Level 24
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(C(x, xs)); a -< x --> False;
          ALL a.
             EX m.
                So(m) & Pe(C(a, xs), m) |] ==>
       ALL xa. ?P20(x, xs, a, xa)
 2. !!x xs a.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          ?P20(x, xs, a, ?x20(x, xs, a)) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- ba 1;
Level 25
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          EX m.
             So(m) &
             Pe(C(?x20(x, xs, a), xs), m) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- bd exE 1;
Level 26
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       ?R21(x, xs, a)
 2. !!x xs a.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          ?R21(x, xs, a) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- ba 2;
Level 27
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       EX m. So(m) & Pe(C(a, C(x, xs)), m)
val it = () : unit
- br exI 1;
Level 28
ALL l.
   So(l) -->
```

```
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(?m22(x, xs, a, m)) &
       Pe(C(a, C(x, xs)), ?m22(x, xs, a, m))
val it = () : unit
- br conjI 1;
Level 29
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(?m22(x, xs, a, m))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(a, C(x, xs)), ?m22(x, xs, a, m))
val it = () : unit
- br SoAddNLe 1;
Level 30
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(C(?a24(x, xs, a, m),
            ?l24(x, xs, a, m)))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       ?b24(x, xs, a, m) -<
       ?a24(x, xs, a, m) -->
       False
 3. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(?b24(x, xs, a, m),
            ?l24(x, xs, a, m)),
          ?m24(x, xs, a, m))
 4. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(?m24(x, xs, a, m))
 5. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(a, C(x, xs)),
          C(?a24(x, xs, a, m),
            ?m24(x, xs, a, m)))
val it = () : unit
- ba 2;
Level 31
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(C(x, ?l24(x, xs, a, m)))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(a, ?l24(x, xs, a, m)),
          ?m24(x, xs, a, m))
 3. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(?m24(x, xs, a, m))
 4. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(a, C(x, xs)),
          C(x, ?m24(x, xs, a, m)))
val it = () : unit
- ba 1;
Level 32
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(a, xs), ?m24(x, xs, a, m))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(?m24(x, xs, a, m))
 3. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(a, C(x, xs)),
          C(x, ?m24(x, xs, a, m)))
val it = () : unit
- br conjunct2 1;
Level 33
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
```

```
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       ?P25(x, xs, a, m) &
       Pe(C(a, xs), ?m24(x, xs, a, m))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       So(?m24(x, xs, a, m))
 3. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) &
          Pe(C(?x20(x, xs, a), xs), m) |] ==>
       Pe(C(a, C(x, xs)),
          C(x, ?m24(x, xs, a, m)))
val it = () : unit
- ba 1;
Level 34
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       So(m)
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(C(a, C(x, xs)), C(x, m))
val it = () : unit
- br conjunct1 1;
Level 35
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       So(m) & ?Q26(x, xs, a, m)
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(C(a, C(x, xs)), C(x, m))
val it = () : unit
- ba 1;
Level 36
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(C(a, C(x, xs)), C(x, m))
val it = () : unit
- br PeTrans 1;
Level 37
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(C(a, C(x, xs)), ?m27(x, xs, a, m))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(?m27(x, xs, a, m), C(x, m))
val it = () : unit
- br PeInj 2;
Level 38
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(C(a, C(x, xs)),
          C(x, ?l28(x, xs, a, m)))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(?l28(x, xs, a, m), m)
val it = () : unit
- br conjunct2 2;
Level 39
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(C(a, C(x, xs)),
          C(x, ?l28(x, xs, a, m)))
 2. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       ?P29(x, xs, a, m) &
       Pe(?l28(x, xs, a, m), m)
val it = () : unit
- ba 2;
Level 40
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
 1. !!x xs a m.
       [| So(C(x, xs)); a -< x --> False;
          ALL a. EX m. So(m) & Pe(C(a, xs), m);
          So(m) & Pe(C(a, xs), m) |] ==>
       Pe(C(a, C(x, xs)), C(x, C(a, xs)))
val it = () : unit
- br PeAssoc 1;
Level 41
ALL l.
   So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))
No subgoals!
val it = () : unit
- val insert = result();
val insert =
  "ALL l. So(l) -->
   (ALL a. EX m. So(m) & Pe(C(a, l), m))" : thm
```

With the insert proposition (8.2) proved, the insort proposition (8.1) can be proved. The insort proposition is proved by induction on the structure of the list l and the insert proposition is used in the proof of the inductive case. The proof is as follows.

```
- goal Insort.thy
= "ALL l. EX m. So(m) & Pe(m,l)";
Level 0
ALL l. EX m. So(m) & Pe(m, l)
 1. ALL l. EX m. So(m) & Pe(m, l)
val it = [] : thm list
-   br induction 1;
Level 1
ALL l. EX m. So(m) & Pe(m, l)
 1. EX m. So(m) & Pe(m, Nil)
 2. ALL x xs.
       (EX m. So(m) & Pe(m, xs)) -->
       (EX m. So(m) & Pe(m, C(x, xs)))
val it = () : unit
-   br exI 1;
Level 2
ALL l. EX m. So(m) & Pe(m, l)
 1. So(?m1) & Pe(?m1, Nil)
 2. ALL x xs.
       (EX m. So(m) & Pe(m, xs)) -->
       (EX m. So(m) & Pe(m, C(x, xs)))
val it = () : unit
-   br conjI 1;
Level 3
ALL l. EX m. So(m) & Pe(m, l)
 1. So(?m1)
 2. Pe(?m1, Nil)
 3. ALL x xs.
       (EX m. So(m) & Pe(m, xs)) -->
       (EX m. So(m) & Pe(m, C(x, xs)))
val it = () : unit
-   br SoNil 1;
Level 4
ALL l. EX m. So(m) & Pe(m, l)
 1. Pe(Nil, Nil)
 2. ALL x xs.
       (EX m. So(m) & Pe(m, xs)) -->
       (EX m. So(m) & Pe(m, C(x, xs)))
val it = () : unit
-   br PeNil 1;
Level 5
ALL l. EX m. So(m) & Pe(m, l)
 1. ALL x xs.
       (EX m. So(m) & Pe(m, xs)) -->
       (EX m. So(m) & Pe(m, C(x, xs)))
val it = () : unit
-   br allI 1;
Level 6
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x. ALL xs.
           (EX m. So(m) & Pe(m, xs)) -->
           (EX m. So(m) & Pe(m, C(x, xs)))
val it = () : unit
-   br allI 1;
Level 7
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs.
       (EX m. So(m) & Pe(m, xs)) -->
       (EX m. So(m) & Pe(m, C(x, xs)))
val it = () : unit
-   br impI 1;
Level 8
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs.
       EX m. So(m) & Pe(m, xs) ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   bd exE 1;
Level 9
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m. So(m) & Pe(m, xs) ==> ?R8(x, xs)
 2. !!x xs.
       ?R8(x, xs) ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   ba 2;
Level 10
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   br exE 1;
Level 11
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==>
       EX xa. ?P9(x, xs, m, xa)
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          ?P9(x, xs, m, xa) |] ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   br allE 1;
Level 12
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==>
       ALL xa. ?P10(x, xs, m, xa)
 2. !!x xs m.
       [| So(m) & Pe(m, xs);
          ?P10(x, xs, m, ?x10(x, xs, m)) |] ==>
       EX xa. ?P9(x, xs, m, xa)
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          ?P9(x, xs, m, xa) |] ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   br mp 1;
Level 13
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==>
       ?P11(x, xs, m) -->
       (ALL xa. ?P10(x, xs, m, xa))
 2. !!x xs m.
       So(m) & Pe(m, xs) ==> ?P11(x, xs, m)
 3. !!x xs m.
       [| So(m) & Pe(m, xs);
          ?P10(x, xs, m, ?x10(x, xs, m)) |] ==>
       EX xa. ?P9(x, xs, m, xa)
 4. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          ?P9(x, xs, m, xa) |] ==>
```

```
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   br allE 1;
Level 14
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==>
       ALL xa. ?P12(x, xs, m, xa)
 2. !!x xs m.
       [| So(m) & Pe(m, xs);
          ?P12(x, xs, m, ?x12(x, xs, m)) |] ==>
       ?P11(x, xs, m) -->
       (ALL xa. ?P10(x, xs, m, xa))
 3. !!x xs m.
       So(m) & Pe(m, xs) ==> ?P11(x, xs, m)
 4. !!x xs m.
       [| So(m) & Pe(m, xs);
          ?P10(x, xs, m, ?x10(x, xs, m)) |] ==>
       EX xa. ?P9(x, xs, m, xa)
 5. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          ?P9(x, xs, m, xa) |] ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   br insert 1;
Level 15
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       [| So(m) & Pe(m, xs);
          So(?x12(x, xs, m)) -->
          (ALL a.
              EX ma.
                 So(ma) &
                 Pe(C(a, ?x12(x, xs, m)),
                    ma)) |] ==>
       ?P11(x, xs, m) -->
       (ALL xa. ?P10(x, xs, m, xa))
 2. !!x xs m.
       So(m) & Pe(m, xs) ==> ?P11(x, xs, m)
 3. !!x xs m.
       [| So(m) & Pe(m, xs);
          ?P10(x, xs, m, ?x10(x, xs, m)) |] ==>
       EX xa. ?P9(x, xs, m, xa)
 4. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          ?P9(x, xs, m, xa) |] ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   ba 1;
Level 16
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==> So(?x12(x, xs, m))
 2. !!x xs m.
       [| So(m) & Pe(m, xs);
          EX ma.
             So(ma) &
             Pe(C(?x10(x, xs, m),
                  ?x12(x, xs, m)),
                ma) |] ==>
       EX xa. ?P9(x, xs, m, xa)
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          ?P9(x, xs, m, xa) |] ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   ba 2;
Level 17
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==> So(?x12(x, xs, m))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), ?x12(x, xs, m)),
             xa) |] ==>
       EX m. So(m) & Pe(m, C(x, xs))
val it = () : unit
-   br exI 2;
Level 18
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==> So(?x12(x, xs, m))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), ?x12(x, xs, m)),
             xa) |] ==>
       So(?m14(x, xs, m, xa)) &
       Pe(?m14(x, xs, m, xa), C(x, xs))
val it = () : unit
-   br conjI 2;
Level 19
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==> So(?x12(x, xs, m))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), ?x12(x, xs, m)),
             xa) |] ==>
       So(?m14(x, xs, m, xa))
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), ?x12(x, xs, m)),
             xa) |] ==>
       Pe(?m14(x, xs, m, xa), C(x, xs))
val it = () : unit
-   br conjunct1 1;
Level 20
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m.
       So(m) & Pe(m, xs) ==>
       So(?x12(x, xs, m)) & ?Q16(x, xs, m)
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), ?x12(x, xs, m)),
             xa) |] ==>
       So(?m14(x, xs, m, xa))
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), ?x12(x, xs, m)),
             xa) |] ==>
       Pe(?m14(x, xs, m, xa), C(x, xs))
val it = () : unit
-   ba 1;
Level 21
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
```

```
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       So(?m14(x, xs, m, xa))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(?m14(x, xs, m, xa), C(x, xs))
val it = () : unit
-   br PeTrans 2;
Level 22
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       So(?m14(x, xs, m, xa))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(?m14(x, xs, m, xa),
          ?m17(x, xs, m, xa))
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(?m17(x, xs, m, xa), C(x, xs))
val it = () : unit
-   br PeInj 3;
Level 23
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       So(?m14(x, xs, m, xa))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(?m14(x, xs, m, xa),
          C(x, ?l18(x, xs, m, xa)))
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(?l18(x, xs, m, xa), xs)
val it = () : unit
-   br PeSym 2;
Level 24
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       So(?m14(x, xs, m, xa))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(C(x, ?l18(x, xs, m, xa)),
          ?m14(x, xs, m, xa))
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(?l18(x, xs, m, xa), xs)
val it = () : unit
-   br conjunct2 2;
Level 25
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       So(?m14(x, xs, m, xa))
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       ?P20(x, xs, m, xa) &
       Pe(C(x, ?l18(x, xs, m, xa)),
          ?m14(x, xs, m, xa))
 3. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) &
          Pe(C(?x10(x, xs, m), m), xa) |] ==>
       Pe(?l18(x, xs, m, xa), xs)
val it = () : unit
-   ba 2;
Level 26
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) & Pe(C(x, m), xa) |] ==>
       So(xa)
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) & Pe(C(x, m), xa) |] ==>
       Pe(m, xs)
val it = () : unit
-   br conjunct1 1;
Level 27
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) & Pe(C(x, m), xa) |] ==>
       So(xa) & ?Q21(x, xs, m, xa)
 2. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) & Pe(C(x, m), xa) |] ==>
       Pe(m, xs)
val it = () : unit
-   ba 1;
Level 28
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) & Pe(C(x, m), xa) |] ==>
       Pe(m, xs)
val it = () : unit
-   br conjunct2 1;
Level 29
ALL l. EX m. So(m) & Pe(m, l)
 1. !!x xs m xa.
       [| So(m) & Pe(m, xs);
          So(xa) & Pe(C(x, m), xa) |] ==>
       ?P22(x, xs, m, xa) & Pe(m, xs)
val it = () : unit
-   ba 1;
Level 30
ALL l. EX m. So(m) & Pe(m, l)
No subgoals!
val it = () : unit
```

```
- val insort = result();                          "ALL l. EX m. So(m) & Pe(m, l)" : thm
val insort =
```

Hereby the propositions have been proved, and insertion sort is proved.

8.4 Formal Theory on Sorted Lists with Proofs

Formal theory $\mathcal{SL}$ can be extended with linearized proof constructions.

For the predicate $\mathcal{P}$ rules $\mathcal{P}_{\mathsf{refl}}$, $\mathcal{P}_{\mathsf{inj}}$ and $\mathcal{P}_{\mathsf{assoc}}$ of formal theory $\mathcal{SL}$ may be regarded as introduction rules since they introduce new information to the arguments of $\mathcal{P}$. Rules $\mathcal{P}_{\mathsf{sym}}$ and $\mathcal{P}_{\mathsf{trans}}$ do not introduce new information but helps introducing new information in the following applications of rules for predicate $\mathcal{P}$ and may be regarded as elimination rules. If an elimination rule is a rule that always takes part in a redex when used with a certain other rule, then only rule $\mathcal{P}_{\mathsf{sym}}$ is an elimination rule.

For the predicate $\mathcal{S}$ rules $\mathcal{S}_{\mathsf{nil}}$, $\mathcal{S}_{\mathsf{single}}$, $\mathcal{S}_{\mathsf{add}_\prec}$ and $\mathcal{S}_{\mathsf{add}_\nprec}$ add new information about what is sorted and may therefore be introduction rules. Rule $\mathcal{S}_{\mathsf{inj}}$ removes information about what is sorted and may therefore be an elimination rule.

For the predicate $\prec$ all rules adds information about what is ordered and the rules may therefore all be introduction rules.

An axiom is normally proven by a proof without arguments. Since a proof of an axiom for the predicate $\prec$ acts as a stand-in for another proof, the arguments between which the relation must be stated must also be arguments of the stand-in proof. If not the proof is specified with arguments situations could arise where it is not possible to discover which proof to use instead of the stand-in proof. A reduction proposition like

$$\mathtt{when}(s, p_l, p_r) \longmapsto p : R$$

having a stand-in proof s that by reduction may be replaced with a "real" proof r is reduced to reduction propositions

$$s \longmapsto s' \quad : \quad P \vee Q \tag{8.4}$$

$$[x : P] \quad \cdots \quad p_l \longmapsto p_l' \quad : \quad R \tag{8.5}$$

$$[y : Q] \quad \cdots \quad p_r \longmapsto p_r' \quad : \quad R \tag{8.6}$$

$$\mathtt{when}(s', p_l', p_r') \longmapsto_! p \quad : \quad R \tag{8.7}$$

where propositions (8.5, 8.6) do not necessarily have to use their hypotheses. Then the logic part $P \vee Q$ of proposition (8.4) is not necessarily instantiated and do therefore necessarily not contain the arguments which s is a proof of. The arguments that fulfil some $\prec$-axiom may therefore not be used with the proof r if not they are given as arguments to the stand-in proof and thereby can be transferred to r.

8.4.1 Formal Theory $\mathcal{SLP}$

A formal theory $\mathcal{SLP}$ for permutation and sorting on lists with proofs may be based upon formal theory $\mathcal{LISTP}$ like formal theory $\mathcal{SL}$ is based upon formal theory $\mathcal{LIST}$.

Definition 8.4.1 *Formal theory $\mathcal{SLP}$ is defined as formal theory $\mathcal{LISTP}$ extended with the following*

Symbols *are extended with reserved predicate letters $\mathcal{P}$ of arity 2, $\mathcal{S}_\prec$ of arity 1 and $\prec$ of arity 2.*

Propositions *consists of a proof part and a proposition part; the definition of proof parts is extended with*

82_{135} `PeRefl`
83_{135} `PeSym`(p_1)
84_{135} `PeTrans`(p_1,p_2)
85_{135} `PeInj`(p_1)
86_{135} `PeAssoc`
87_{135} `SoNil`
88_{135} `SoSingle`
89_{135} `SoInj`(p_1)
90_{135} `SoLeq`(p_1,p_2)
91_{135} `SoNLe`(p_1,p_2,p_3,p_4)
92_{135} `OrdRefl`(X_1)
93_{135} `OrdAnsym`(p_1,p_2)
94_{135} `OrdTrans`(p_1,p_2)
95_{135} `OrdTotal`(X_1,X_2)
96_{135} `OrdDecide`(X_1,X_2)

where p_n is a proof part and X_n is a term variable.

Axioms *The axiom set of $\mathcal{LISTP}$ is extended with the axioms of figure 8.2.*

Inference Rules *The set of inference rules of $\mathcal{LISTP}$ is extended with the inference rules of figure 8.2.*

Figure 8.2 Rules for the predicates $\mathcal{P}$, $\mathcal{S}$ and $\prec$ extending the formal theory $\mathcal{LISTP}$ to the formal theory $\mathcal{SLP}$.

$$\frac{}{\mathtt{PeRefl} : \mathcal{P}(l,l)}\ \mathcal{P}_{\mathsf{refl}} \qquad \frac{}{\mathtt{SoSingle} : \mathcal{S}_\prec(\mathsf{C}(a,\mathsf{Nil}))}\ \mathcal{S}_{\mathsf{single}}$$

$$\frac{x : \mathcal{P}(l,m)}{\mathtt{PeSym}(x) : \mathcal{P}(m,l)}\ \mathcal{P}_{\mathsf{sym}} \qquad \frac{x : \mathcal{S}_\prec(\mathsf{C}(a,l))}{\mathtt{SoInj}(x) : \mathcal{S}_\prec(l)}\ \mathcal{S}_{\mathsf{inj}}$$

$$\frac{x : \mathcal{P}(l,m) \qquad y : \mathcal{P}(m,n)}{\mathtt{PeTrans}(x,y) : \mathcal{P}(l,n)}\ \mathcal{P}_{\mathsf{trans}} \qquad \frac{x : \mathcal{S}_\prec(\mathsf{C}(a,l)) \qquad y : b \prec a}{\mathtt{SoLeq}(x,y) : \mathcal{S}_\prec(\mathsf{C}(b,\mathsf{C}(a,l)))}\ \mathcal{S}_{\mathsf{add}_\prec}$$

$$\frac{x : \mathcal{P}(l,m)}{\mathtt{PeInj}(x) : \mathcal{P}(\mathsf{C}(a,l),\mathsf{C}(a,m))}\ \mathcal{P}_{\mathsf{inj}} \qquad \frac{}{\mathtt{OrdRefl}(x) : x \prec x}\ \prec_{\mathsf{refl}}$$

$$\frac{w : x \prec y \qquad z : y \prec x}{\mathtt{OrdAnsym}(w,z) : x = y}\ \prec_{\mathsf{antisym}}$$

$$\frac{}{\mathtt{PeAssoc} : \mathcal{P}(\mathsf{C}(a,\mathsf{C}(b,l)),\mathsf{C}(b,\mathsf{C}(a,l)))}\ \mathcal{P}_{\mathsf{assoc}} \qquad \frac{v : x \prec y \qquad w : y \prec z}{\mathtt{OrdTrans}(v,w) : x \prec z}\ \prec_{\mathsf{trans}}$$

$$\frac{}{\mathtt{SoNil} : \mathcal{S}_\prec(\mathsf{Nil})}\ \mathcal{S}_{\mathsf{nil}} \qquad \frac{}{\mathtt{OrdTotal}(x,y) : x \prec y \vee y \prec x}\ \prec_{\mathsf{total}}$$

$$\frac{w : \mathcal{S}_\prec(\mathsf{C}(a,l)) \qquad x : b \prec a \to \bot \qquad y : \mathcal{P}(\mathsf{C}(b,l),m) \qquad z : \mathcal{S}_\prec(m)}{\mathtt{SoNLe}(w,x,y,z) : \mathcal{S}_\prec(\mathsf{C}(a,m))}\ \mathcal{S}_{\mathsf{add}_\not\prec}$$

$$\frac{}{\mathtt{OrdDecide}(x,y) : x \prec y \vee (x \prec y \to \bot)}\ \prec_{\mathsf{decideability}}$$

8.4.2 Object Logic representing Formal Theory $\mathcal{SLP}$

The object logic for the formal theory $\mathcal{SLP}$ is called `InsortProof` and is placed in the files `InsortProof.thy` and `InsertProof.ML`. The theory in the files does not implement exactly the same rules as specified in formal theory $\mathcal{SLP}$ since redundancy is eliminated, which gives that the object logic implements a constructor `PeNil` and does *not* implement constructors `PeRefl` and `SoNil`.

Theory File `InsortProof.thy`

```
(* Time-stamp: <1999-02-03 12:40:11 kokdg> *)

InsortProof = LISTP +

consts Pe         ::  "['a list, 'a list] => o"
       So         ::  "'a list => o"
       "-<"       ::  "['a, 'a] => o" (infixl 50)
       PeNil      :: "p"
       PeAssoc    :: "p"
       SoSingle   :: "p"
       OrdRefl    :: "'a => p"
       OrdTotal   :: "['a, 'a] => p"
       OrdDecide  :: "['a, 'a] => p"

       PeSym      :: "p => p"
       PeInj      :: "p => p"
       SoInj      :: "p => p"

       PeTrans    :: "[p, p] => p"
       SoLeq      :: "[p, p] => p"
       OrdAnsym   :: "[p, p] => p"
       OrdTrans   :: "[p, p] => p"

       SoNLe      :: "[p, p, p, p] => p"

rules  PeNil     "PeNil : Pe(l,l)"
       PeSym     "x : Pe(l,m) ==> PeSym(x) : Pe(m,l)"
       PeTrans   "[| x : Pe(l,m); y : Pe(m,n) |] ==> PeTrans(x,y) : Pe(l,n)"
       PeInj     "x : Pe(l,m) ==> PeInj(x) : Pe(C(a,l), C(a,m))"
       PeAssoc   "PeAssoc : Pe(C(a,C(b,l)), C(b,C(a,l)))"

       SoSingle  "SoSingle : So(C(a,Nil))"
       SoInj     "x : So(C(a,l)) ==> SoInj(x) : So(l)"
       SoAddLe   "[| x : So(C(a,l)); y : b -< a |] ==>
                  SoLeq(x,y) : So(C(b,C(a,l)))"
       SoAddNLe  "[| w : So(C(a,l)); x : (b -< a) --> False; y : Pe(C(b,l),m);
                     z : So(m) |] ==> SoNLe(w,x,y,z) : So(C(a,m))"

       OrdRefl   "OrdRefl(x) : x -< x"
       OrdAnSym  "[| w : x -< y; z : y -< x |] ==> OrdAnsym(w,z) : x = y"
       OrdTrans  "[| v : x -< y; w : y -< z |] ==> OrdTrans(v,w) : x -< z"
       OrdTotal  "OrdTotal(x,y) : x -< y | y -< x"
```

```
       OrdDecide "OrdDecide(x,y) : x -< y | (x -< y --> False)"

end
```

Theory File `InsortProof.ML`

```
(* Time-stamp: <1999-01-28 17:30:14 kokdg> *)

open InsortProof;

  goal InsortProof.thy "?p : Pe(?l, ?l)";
  br allE 1;
  ba 2;
  br induction 1;
  br PeNil 1;
  br allI 1;
  br allI 1;
  br impI 1;
  br PeInj 1;
  ba 1;
val PeRefl = result();

  goal InsortProof.thy "?p : So(Nil)";
  br SoInj 1;
  br SoSingle 1;
val SoNil = result();
```

8.4.3 Proofs for the Sorting Propositions

In section 8.3 the propositions (8.1) and (8.2) were proven for formal theory $\mathcal{SL}$. Proofs for the propositions in formal theory $\mathcal{SLP}$ are created using the exact same applications of tactics and order of application as in formal theory $\mathcal{SL}$. This ends up in a proof

```
val insort =
  "lrec
    ([Nil,<SoInj(SoSingle),PeNil>],
     all x xa.
        lam xa.
           xsplit
            (xa,
             %m u.
                xsplit
                 ((lrec
                    (lam x.
                        all x.
                           [C(x, Nil),
                            <SoSingle,
                             lrec(PeNil, all x xa. lam x. PeInj(x)) ^
                             C(x, Nil)>],
                     all x xa.
                        lam xb xc.
```

```
                        all xd.
                           when
                            (OrdDecide(xd, x),
                             %xca.
                                [C(xd, C(x, xa)),
                                 <SoLeq(xc, xca),
                                  lrec(PeNil, all x xa. lam x. PeInj(x)) ^
                                  C(xd, C(x, xa))>],
                             %xca.
                                xsplit
                                 (xb ‘ SoInj(xc) ^ xd,
                                  %m u.
                                     [C(x, m),
                                      <SoNLe(xc, xca, snd(u), fst(u)),
                                       PeTrans
                                        (PeAssoc, PeInj(snd(u)))>])) ^ m) ‘
                fst(u) ^ x,
                %xb ua.
                   [xb,<fst(ua),PeTrans(PeSym(snd(ua)), PeInj(snd(u)))>]))
  : ALL l. EX m. So(m) & Pe(m, l)" : thm
```

8.5 Reduction of Proofs on Sorted Lists

Rule $\mathcal{S}_{\mathsf{inj}}$ creates redexes with either rule $\mathcal{S}_{\mathsf{add}_{\not\prec}}$ or rule $\mathcal{S}_{\mathsf{add}_{\prec}}$ giving the reduction rules

$$\frac{p \longmapsto q : \mathcal{S}_{\prec}(\mathsf{C}(a,l))}{\mathtt{SoInj(SoLeq}(p,r)) \longmapsto_! q : \mathcal{S}_{\prec}(\mathsf{C}(a,l))}\ \mathcal{S}_{\mathsf{inj}}\mathcal{S}_{\mathsf{add}_{\prec}}\mathsf{R}$$

and

$$\frac{p \longmapsto q : \mathcal{S}_{\prec}(l)}{\mathtt{SoInj(SoNLe}(w,x,y,p)) \longmapsto_! q : \mathcal{S}_{\prec}(l)}\ \mathcal{S}_{\mathsf{inj}}\mathcal{S}_{\mathsf{add}_{\not\prec}}\mathsf{R}$$

Rule $\mathcal{P}_{\mathsf{sym}}$ creates redexes with itself,

$$\frac{p \longmapsto q : \mathcal{P}(l,m)}{\mathtt{PeSym(PeSym}(p)) \longmapsto_! q : \mathcal{P}(l,m)}\ \mathcal{P}_{\mathsf{sym}}\mathsf{R}$$

which means that the same rule acts both as introduction rule and elimination rule since the rule is its own inverse.

An elimination rule in connection with reduction gives reason to more information than an introduction rule. An elimination rule gives reason to a context rule with a reduction premise, a reduction rule, and the context rule are used with program evaluation, which all is information that introduction rules do not give. Therefore a rule that acts both like introduction rule and elimination rule must be regarded as an elimination rule since else the outcoming proofs of a reduction will not be on the desired form if the proof due for reduction includes an application of the rule.

With the above splitting in introduction and elimination rules the context introduction rules on figure 8.3 and the context elimination rules on figure 8.4 can be created.

Figure 8.3 Constructor context rules for the predicates $\mathcal{P}$, $\mathcal{S}$ and $\prec$.

$$\frac{}{\texttt{PeRefl} \longmapsto \texttt{PeRefl} : \mathcal{P}(l,l)}\ \mathcal{P}_{\mathsf{refl}}\mathsf{CI}$$

$$\frac{x \longmapsto x' : \mathcal{P}(l,m) \qquad y \longmapsto y' : \mathcal{P}(m,n)}{\texttt{PeTrans}(x,y) \longmapsto \texttt{PeTrans}(x',y') : \mathcal{P}(l,n)}\ \mathcal{P}_{\mathsf{trans}}\mathsf{CI}$$

$$\frac{x \longmapsto x' : \mathcal{P}(l,m)}{\texttt{PeInj}(x) \longmapsto \texttt{PeInj}(x') : \mathcal{P}(\mathsf{C}(a,l),\mathsf{C}(a,m))}\ \mathcal{P}_{\mathsf{inj}}\mathsf{CI}$$

$$\frac{}{\texttt{PeAssoc} \longmapsto \texttt{PeAssoc} : \mathcal{P}(\mathsf{C}(a,\mathsf{C}(b,l)),\mathsf{C}(b,\mathsf{C}(a,l)))}\ \mathcal{P}_{\mathsf{assoc}}\mathsf{CI}$$

$$\frac{}{\texttt{SoNil} \longmapsto \texttt{SoNil} : \mathcal{S}_{\prec}(\mathsf{Nil})}\ \mathcal{S}_{\mathsf{nil}}\mathsf{CI}$$

$$\frac{}{\texttt{SoSingle} \longmapsto \texttt{SoSingle} : \mathcal{S}_{\prec}(\mathsf{C}(a,\mathsf{Nil}))}\ \mathcal{S}_{\mathsf{single}}\mathsf{CI}$$

$$\frac{x \longmapsto x' : \mathcal{S}_{\prec}(\mathsf{C}(a,l)) \qquad y \longmapsto y' : b \prec a}{\texttt{SoLeq}(x,y) \longmapsto \texttt{SoLeq}(x',y') : \mathcal{S}_{\prec}(\mathsf{C}(b,\mathsf{C}(a,l)))}\ \mathcal{S}_{\mathsf{add}_{\prec}}\mathsf{CI}$$

$$\frac{w \longmapsto w' : \mathcal{S}_{\prec}(\mathsf{C}(a,l)) \quad x \longmapsto x' : b \prec a \to \bot \quad y \longmapsto y' : \mathcal{P}(\mathsf{C}(b,l),m) \quad z \longmapsto z' : \mathcal{S}_{\prec}(m)}{\texttt{SoNLe}(w,x,y,z) \longmapsto \texttt{SoNLe}(w',x',y',z') : \mathcal{S}_{\prec}(\mathsf{C}(a,m))}\ \mathcal{S}_{\mathsf{add}_{\not\prec}}\mathsf{CI}$$

$$\frac{}{\texttt{OrdRefl}(x) \longmapsto \texttt{OrdRefl}(x) : x \prec x}\ \prec_{\mathsf{refl}}\mathsf{CI}$$

$$\frac{w \longmapsto w' : x \prec y \qquad z \longmapsto z' : y \prec x}{\texttt{OrdAnsym}(w,z) \longmapsto \texttt{OrdAnsym}(w',z') : x = y}\ \prec_{\mathsf{antisym}}\mathsf{CI}$$

$$\frac{v \longmapsto v' : x \prec y \qquad w \longmapsto w' : y \prec z}{\texttt{OrdTrans}(v,w) \longmapsto \texttt{OrdTrans}(v',w') : x \prec z}\ \prec_{\mathsf{trans}}\mathsf{CI}$$

$$\frac{}{\texttt{OrdTotal}(x,y) \longmapsto \texttt{OrdTotal}(x,y) : x \prec y \vee y \prec x}\ \prec_{\mathsf{total}}\mathsf{CI}$$

$$\frac{}{\texttt{OrdDecide}(x,y) \longmapsto \texttt{OrdDecide}(x,y) : x \prec y \vee (x \prec y \to \bot)}\ \prec_{\mathsf{decideability}}CI$$

Figure 8.4 Destructor context rules for the predicates $\mathcal{P}$, $\mathcal{S}$ and $\prec$.

$$\frac{x \longmapsto x' : \mathcal{S}_{\prec}(\mathsf{C}(a,l)) \qquad \texttt{SoInj}(x') \longmapsto_! r : \mathcal{S}_{\prec}(l)}{\texttt{SoInj}(x) \longmapsto r : \mathcal{S}_{\prec}(l)}\ \mathcal{S}_{\mathsf{inj}}\mathsf{CE}$$

$$\frac{x \longmapsto x' : \mathcal{P}(l,m) \qquad \texttt{PeSym}(x') \longmapsto_! r : \mathcal{P}(m,l)}{\texttt{PeSym}(x) \longmapsto r : \mathcal{P}(m,l)}\ \mathcal{P}_{\mathsf{sym}}\mathsf{CE}$$

8.5.1 Formal Theory $\mathcal{PR}_{\ldots\ \mathcal{SL}}$

With the above defined context rules and reduction rules a formal theory for proof reduction of proofs regarding list sorting $\mathcal{PR}_{\ldots\ \mathcal{SL}}$ can be created. This formal theory is based on formal theory $\mathcal{PR}_{\mathcal{IFOLP},\ \mathcal{LIST}}$.

Definition 8.5.1 *Formal theory $\mathcal{PR}_{\ldots\ \mathcal{SL}}$ is formal theory $\mathcal{PR}_{\mathcal{IFOLP},\ \mathcal{LIST}}$ extended with the following*

Symbols *are the symbols used to extend $\mathcal{LISTP}$ to $\mathcal{SLP}$.*

Propositions *are extended like $\mathcal{LISTP}$ is extended to $\mathcal{SLP}$.*

Axioms *are the axioms used to extend $\mathcal{LISTP}$ to $\mathcal{SLP}$ and axioms of figure 8.3.*

Inference rules *are the inference rules used to extend $\mathcal{LISTP}$ to $\mathcal{SLP}$, inference rules of figures 8.3 and 8.4 and reduction rules $\mathcal{S}_{\mathsf{inj}}\mathcal{S}_{\mathsf{add}_{\prec}}\mathsf{R}$, $\mathcal{S}_{\mathsf{inj}}\mathcal{S}_{\mathsf{add}_{\not\prec}}\mathsf{R}$ and $\mathcal{P}_{\mathsf{sym}}\mathsf{R}$*

8.5.2 Formal Theory $\mathcal{PE}_{\ldots\ \mathcal{SL}}$

With the above defined context rules and reduction rules a formal theory for program evaluation of proofs regarding list sorting regarded as programs $\mathcal{PE}_{\ldots\ \mathcal{SL}}$ can be created. This formal theory is based on formal theory $\mathcal{PE}_{\mathcal{IFOLP},\ \mathcal{LIST}}$.

Definition 8.5.2 *Formal theory $\mathcal{PE}_{\ldots\ \mathcal{SL}}$ is formal theory $\mathcal{PE}_{\mathcal{IFOLP},\ \mathcal{LIST}}$ extended with the following*

Symbols *are the symbols used to extend $\mathcal{LISTP}$ to $\mathcal{SLP}$.*

Propositions *are extended like $\mathcal{LISTP}$ is extended to $\mathcal{SLP}$.*

Axioms *are the axioms used to extend $\mathcal{LISTP}$ to $\mathcal{SLP}$.*

Inference rules *are the inference rules used to extend $\mathcal{LISTP}$ to $\mathcal{SLP}$, inference rules of figure 8.4 and reduction rules $\mathcal{S}_{\mathsf{inj}}\mathcal{S}_{\mathsf{add}_{\prec}}\mathsf{R}$, $\mathcal{S}_{\mathsf{inj}}\mathcal{S}_{\mathsf{add}_{\not\prec}}\mathsf{R}$ and $\mathcal{P}_{\mathsf{sym}}\mathsf{R}$*

8.5.3 Object Logics representing Formal Theories $\mathcal{PR}_{\ldots\ \mathcal{SL}}$ and $\mathcal{PE}_{\ldots\ \mathcal{SL}}$

The formal theories $\mathcal{PR}_{\ldots\ \mathcal{SL}}$ and $\mathcal{PE}_{\ldots\ \mathcal{SL}}$ must be implemented in two object logics. These object logics are named `PRInsort` respectively `PEInsort`, represented in files `PRInsort.thy` and `PRInsort.ML` respectively `PEInsort.thy` and `PEInsort.ML`.

The object logics `PRInsort` and `PEInsort` share some information which are placed in an object logic `InsortReduction`. The shared information are the context elimination rules, reduction rules, the already constructed object logics `InsortProof` and `LISTReduction` and an update of the tactic `try_all_rules`.

Object logic `PRInsort` contains the parts shared with object logic `PEInsort`, and context introduction rules since these rules are not shared with object logic `PEInsort`. The object logic differs a bit from the formal theory $\mathcal{PR}_{\ldots\ \mathcal{SL}}$ since rules $\mathcal{P}_{\mathsf{refl}}\mathsf{CI}$ and $\mathcal{S}_{\mathsf{nil}}\mathsf{CI}$ are not implemented because of redundancy. Instead a rule with name `PeNilCI` is implemented.

Object logic `PEInsort` contains the parts shared with object logic `PRInsort` and an update of the tactic `try_allRules`.

Theory File `InsortReduction.thy`

```
(* Time-stamp: <1999-02-02 16:54:22 kokdg> *)

InsortReduction = InsortProof + LISTReduction +

rules

(* Context elimination rules *)

       PeSymCE   "[| x >> x' : Pe(l,m); PeSym(x') !> r : Pe(m,l) |]
                  ==> PeSym(x) >> r : Pe(m,l)"

       SoInjCE   "[| x >> x' : So(C(a,l)); SoInj(x') !> r : So(l) |]
                  ==> SoInj(x) >> r : So(l)"




(* reduction Rules *)

       SoInjLeR  "p >> q : So(C(a,l)) ==>
                  SoInj(SoLeq(p,r)) !> q : So(C(a,l))"
       SoInjNLeR "P >> q : So(l) ==> SoInj(SoNLe(w,x,y,p)) !> q : So(l)"

       PeSymR    "p >> q : Pe(l,m) ==> PeSym(PeSym(p)) !> q : P(l,m)"

end
```

Theory File `InsortReduction.ML`

```
(* Time-stamp: <1999-01-31 14:37:29 kokdg> *)

open InsortReduction;

val t' = try_all_rules;
val try_all_rules =
  (resolve_tac [SoInjLeR, SoInjNLeR, PeSymR]) ORELSE' t';
fun bt n = by (try_all_rules n);
```

Theory File `PRInsort.thy`

```
(* Time-stamp: <1999-02-21 13:39:34 kokdg> *)

PRInsort = InsortReduction + PRLIST +

rules  PeNilCI     "PeNil >> PeNil : Pe(l,l)"

       PeTransCI   "[| x >> x' : Pe(l,m); y >> y' : Pe(m,n) |] ==>
                    PeTrans(x,y) >> PeTrans(x',y') : Pe(l,n)"
       PeInjCI     "x >> x' : Pe(l,m) ==>
```

```
                PeInj(x) >> PeInj(x') : Pe(C(a,l), C(a,m))"
      PeAssocCI   "PeAssoc >> PeAssoc : Pe(C(a,C(b,l)), C(b,C(a,l)))"

      SoSingleCI  "SoSingle >> SoSingle : So(C(a,Nil))"

      SoAddLeCI   "[| x >> x' : So(C(a,l)); y >> y' : b -< a |] ==>
                   SoLeq(x,y) >> SoLeq(x',y') : So(C(b,C(a,l)))"
      SoAddNLeCI  "[| w >> w' : So(C(a,l)); x >> x' : (b -< a) --> False;
                      y >> y' : Pe(C(b,l),m); z >> z' : So(m) |] ==>
                   SoNLe(w,x,y,z) >> SoNLe(w',x',y',z') : So(C(a,m))"

      OrdReflCI   "OrdRefl(x) >> OrdRefl(x) : x -< x"
      OrdAnSymCI  "[| w >> w' : x -< y; z >> z' : y -< x |] ==>
                   OrdAnsym(w,z) >> OrdAnsym(w',z') : x = y"
      OrdTransCI  "[| v >> v' : x -< y; w >> w' : y -< z |] ==>
                   OrdTrans(v,w) >> OrdTrans(v',w') : x -< z"
      OrdTotalCI  "OrdTotal(x,y) >> OrdTotal(x,y) : x -< y | y -< x"

      OrdDecideCI "OrdDecide(x,y) >> OrdDecide(x,y) :
                   x -< y | (x -< y --> False)"


(* REMEMBER TO DOCUMENT THE HACK BELOW !!! *)

allCE      "[| f >> g : ALL b. R(b) ; g ^ a !> r : R(a)
              |] ==> f ^ a >> r : P(a)"

end
```

Theory File `PRInsort.ML`

```
(* Time-stamp: <1999-01-31 14:38:12 kokdg> *)

open PRInsort;
```

Theory File `PEInsort.thy`

```
(* Time-stamp: <1999-02-02 17:01:23 kokdg> *)

PEInsort = InsortReduction + PELIST
```

Theory File `PEInsort.ML`

```
(* Time-stamp: <1999-02-04 14:51:36 kokdg> *)

open PEInsort;

val t' = try_allRules;
val try_allRules =
  (resolve_tac [PeSymCE, SoInjCE]) ORELSE' t';
```

```
fun bta n = by (try_allRules n);
```

8.5.4 Reducing the Sorting Propositions

With the above defined formal theories the proofs for the sorting propositions can be reduced. Since the proof for the insertion proposition (8.2) is a part of the proof for the sorting proposition (8.1) it is not reduced for itself but as being part of the proof of the sorting proposition. Using the formal theory $\mathcal{PR}_{\ldots\, \mathcal{SL}}$ for reducing the proof of the sorting proposition presented in section 8.4.3, a reduced proof in normal form is obtained as

```
val PRinsort =
  "lrec
    ([Nil,<SoInj(SoSingle),PeNil>],
     all x xa.
        lam xa.
           xsplit
            (xa,
             %xa u.
                xsplit
                 ((lrec
                    (lam x. all x. [C(x, Nil),<SoSingle,PeInj(PeNil)>],
                     all x xa.
                        lam xb xc.
                           all xd.
                              when
                               (OrdDecide(xd, x),
                                %xaa.
                                   [C(xd, C(x, xa)),
                                    <SoLeq(xc, xaa),
                                     PeInj
                                      (PeInj
                                        (lrec
                                          (PeNil, all x xa. lam x. PeInj(x)) ^
                                         xa))>],
                                %xa.
                                   xsplit
                                    (xb ‘ SoInj(xc) ^ xd,
                                     %xb ua.
                                        [C(x, xb),
                                         <SoNLe(xc, xa, snd(ua), fst(ua)),
                                          PeTrans
                                           (PeAssoc, PeInj(snd(ua)))>])))) ^
                  xa) ‘ fst(u) ^ x,
                 %xa ua.
                    [xa,<fst(ua),PeTrans(PeSym(snd(ua)), PeInj(snd(u)))>])))
   : ALL l. EX m. So(m) & Pe(m, l)" : thm
```

The proving session for the above proof consists of the tactic applications

```
goal PRInsort.thy
  "?p : ?P";
br redE 1;
br insort 1;
br inductionCI 1;
br exCI 1;
br conjCI 1;
br SoInjCE 1;
br SoSingleCI 1;
bt 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br exCE 1;
bv 1;
br exCE 1;
br allCE 1;
br impCE 1;
br allCE 1;
br conjCE1 3;
bv 3;
bt 3;
br inductionCI 1;
br impCI 1;
br allCI 1;
br exCI 1;
br conjCI 1;
br SoSingleCI 1;
br allCE 1;
br inductionCI 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
br inductionRC 1;
br impCE 1;
br allCE 1;
br allCE 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
br allR 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
ba 1;
br allCE 1;
br inductionCI 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
br inductionRNil
  1;
br PeNilCI 1;
br impR 1;
br PeInjCI 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br impCI 1;
br allCI 1;
br disjCE 1;
br OrdDecideCI 1;
br exCI 1;
br conjCI 1;
br SoAddLeCI 1;
bv 1;
bv 1;
br allCE 1;
br inductionCI 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
br inductionRC 1;
br impCE 1;
br allCE 1;
br allCE 1;
(* Level 100 *)
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
br allR 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
ba 1;
br allR 1;
br impCI 1;
br PeInjCI 1;
bv 1;
ba 1;
br allCE 1;
br inductionCI 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
br inductionRC 1;
br impCE 1;
br allCE 1;
br allCE 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
br allR 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
ba 1;
br allR 1;
br impCI 1;
br PeInjCI 1;
bv 1;
bt 1;
br impR 1;
br PeInjCI 1;
br allCE 1;
br inductionCI 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
bt 1;
br impR 1;
br PeInjCI 1;
br PeInjCI 1;
br allCE 1;
br inductionCI 1;
br PeNilCI 1;
br allCI 1;
br allCI 1;
br impCI 1;
br PeInjCI 1;
bv 1;
bt 1;
br exCE 1;
br allCE 1;
br impCE 1;
bv 1;
br SoInjCE 1;
bv 1;
bt 1;
bt 1;
bt 1;
br exCI 1;
br conjCI 1;
(* Level 179 *)
br SoAddNLeCI 1;
bv 1;
bv 1;
br conjCE2 1;
bv 1;
bt 1;
br conjCE1 1;
bv 1;
bt 1;
br PeTransCI 1;
br PeInjCI 2;
br conjCE2 2;
bv 2;
bt 2;
br PeAssocCI 1;
bt 1;
bt 1;
bt 1;
bt 1;
bt 1;
br exCI 1;
br conjCI 1;
br conjCE1 1;
bv 1;
bt 1;
br PeTransCI 1;
br PeSymCE 1;
br conjCE2 1;
bv 1;
bt 1;
bt 1;
br PeInjCI 1;
br conjCE2 1;
bv 1;
bt 1;
bt 1;
bt 1;
```

Rule `insort` used above is the proof of the sorting proposition derived in section 8.4.3.

Since the proof in section 8.4.3 already is in weak head normal form a reduction using formal theory $\mathcal{PE}_{\ldots\ \mathcal{SL}}$ will not reduce the proof to a smaller proof.

8.6 Application of Reduced Proofs on Lists of Natural Numbers

A list of natural numbers can be sorted by applying a sorting proof on the list of natural numbers.

Definition 8.6.1 *The outcome of a reduction of an insort application on a list m is*

- *a theorem*

$$[l', \langle sort, perm \rangle] : \exists l.\mathcal{S}_{\prec}(l) \wedge \mathcal{P}(l, m) \tag{8.8}$$

where

  - *l' is the sorted list,*
  - *sort is a proof that l' is sorted,*
  - *perm is a proof that l' is a permutation of m.*

if it is possible to decide the order of the elements of the list,

- *a theorem*

$$\mathtt{xsplit(when(OrdDecide}(v_1, v_2)\mathtt{,}p_1\mathtt{,}p_2\mathtt{),}f\mathtt{)} : \exists l.\mathcal{S}_{\prec}(l) \wedge \mathcal{P}(l, m)$$

where

  - *p_1 and p_2 are proofs like (8.8) depending on how elements v_1 and v_2 are ordered,*
  - *f is the output proof term, like (8.8), in which either p_1 or else p_2 should be inserted.*

if not it is possible to decide the order of the elements in the list.

For a list with n elements there are 2^n different permutations of the list. A proof where the decidability cannot be proven may be huge, and therefore the decidability must be stated for natural numbers.

To decide the order of two natural numbers an object logic describing an order on natural numbers must be implemented. Such an order may be $\leq$ stated in the below presented object logic `NatOrderProof`.

With a decidability proof for natural numbers, reduction of applications of the `insort` proof on lists of natural numbers can be reduced to proofs matching theorem schema (8.8).

8.6.1 Object logic Representing Ordering Relation

The ordering relation $\leq$ for natural numbers can be specified by stating some rules representing the properties that characterises the ordering relation. With these rules proofs replacing the stand-in proofs for axioms specified for predicate $\prec$ in formal theories $\mathcal{SL}$ and $\mathcal{SLP}$ can be proven.

To sort lists with natural numbers the ordering relation $\leq$ must be used together with the object logics `PRInsort` and `PEInsort`.

A reduction rule for replacing stand-in proof term `OrdDecide` with a decideability proof must be included. The replacing rule may be regarded as an elimination rule since the proof replacing the stand-in proof term may contain redexes and therefore be reducible with both proof reduction and program evaluation. To use the ordering relation with the object logics `PRInsort` and `PEInsort` these must be merged with the `NatOrderProof` object logic and proofs for replacing stand-in proof term `OrdDecide` to object logics `PRInsortNat` and `PEInsortNat`.

Theory File `NatOrderProof.thy`

```
(* Time-stamp: <1999-02-21 12:26:40 kokdg> *)
```

```
NatOrderProof = InsortReduction +


consts leq, eq :: "p"
       less, nleq, Oinj, Ocon, less2 :: "p => p"

rules

  Zle    " leq : 0 -< n "
  Znle   " x : S(n) -< 0 ==> nleq(x) : R "
  Ordinj " x : n -< m ==> Oinj(x) : S(n) -< S(m) "
  OrderC " x : S(n) -< S(m) ==> Ocon(x) : n -< m "   (* To be proven!!! *)
  Less   " x : n -< m ==> less(x) : n -< S(m) "
  Less2  " x : S(n) -< m ==> less2(x) : n -< m "

  ZleCI    " leq >> leq : 0 -< n "
  ZnleCI   " x >> x' : S(n) -< 0 ==> nleq(x) >> nleq(x') : R "
  OrdinjCI " x >> x' : n -< m ==> Oinj(x) >> Oinj(x): S(n) -< S(m) "
  LessCI   " x >> x' : n -< m ==> less(x) >> less(x') : n -< S(m) "
  Less2CI  " x >> x' : S(n) -< m ==> less2(x) >> less2(x') : n -< m "
  OrderCCI " x >> x' : S(n) -< S(m) ==> Ocon(x) >> Ocon(x') : n -< m "
(* To be proven!!! *)


end
```

Theory File `NatOrderProof.ML`

```
(* Time-stamp: <1999-02-04 15:00:36 kokdg> *)

open NatOrderProof;

goal NatOrderProof.thy "?p : ALL n m. (n::nat) -< m | (n -< m --> False)";
br NATP.induction 1;
br allI 1;
br disjI1 1;
br Zle 1;
br allI 1;
br impI 1;
br NATP.induction 1;
br disjI2 1;
br impI 1;
br Znle 1;
ba 1;
br allI 1;
br impI 1;
br disjE 1;
br disjI1 2;
br disjI2 3;
br Ordinj 2;
ba 2;
br allE 1;
```

```
ba 2;
ba 1;
br impI 1;
br mp 1;
ba 1;
br OrderC 1;
ba 1;
val decideNat = result();
```

Theory File PRInsortNat.thy

```
(* Time-stamp: <1999-02-04 15:00:56 kokdg> *)

PRInsortNat = PRInsort + NatOrderProof +

rules

  OrdDecideCE' "[| p : x -< y | (x -< y --> False) ;
                   p >> p' : x -< y | (x -< y --> False) |]
                ==> OrdDecide(x,y) >> p' : x -< y | (x -< y --> False)"

end
```

Theory File PRInsortNat.ML

```
(* Time-stamp: <1999-02-19 14:44:04 kokdg> *)

open PRInsortNat;

  goal PRInsortNat.thy "?p >> ?q : ?P";
  br OrdDecideCE' 1;
  br allE 1;
  ba 2;
  br allE 1 ;
  ba 2;
  br decideNat 1;
val OrdDecideCE = uresult();
```

Theory File PEInsortNat.thy

```
(* Time-stamp: <1999-02-04 14:31:18 kokdg> *)

PEInsortNat = PEInsort + NatOrderProof +

rules

  OrdDecideCE' "[| p : x -< y | (x -< y --> False) ;
                   p >> p' : x -< y | (x -< y --> False) |]
                ==> OrdDecide(x,y) >> p' : x -< y | (x -< y --> False)"
```

```
end
```

Theory File `PEInsortNat.ML`

```
(* Time-stamp: <1999-02-04 15:01:50 kokdg> *)

open PEInsortNat;

  goal PEInsortNat.thy "?p >> ?q : ?P";
  br OrdDecideCE' 1;
  br allE 1;
  ba 2;
  br allE 1 ;
  ba 2;
  br decideNat 1;
val OrdDecideCE = uresult();

val t' = try_allRules;
val try_allRules =
  (resolve_tac [OrdDecideCE]) ORELSE' t';
fun bta n = by (try_allRules n);
```

8.6.2 Reducing Applications using $\mathcal{PR}_{\ldots\,\mathcal{SL}}$

The $\mathcal{PR}_{\ldots\,\mathcal{SL}}$-reduced proof `PRinsort` can be applied on different lists to obtain sorted lists by reducing the applications. Below this is illustrated by some examples. To reduce the examples to fit into theorem schema (8.8) the ordering relation for natural numbers must be included, therefore the below reductions are done using object logic `PRInsortNat`.

Application of `PRinsort` on the list Nil reduces to the theorem

$$\texttt{[Nil,<SoInj(SoSingle),PeNil>]} : \exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{Nil})$$

using 110 tactic applications on the current state. Application of `PRinsort` on the list $\mathsf{C}(0, \mathsf{Nil})$ reduces to the theorem

$$\texttt{[}\mathsf{C}(0,\mathsf{Nil})\texttt{,<SoSingle,PeTrans(PeSym(PeInj(PeNil)), PeInj(PeNil))>]} : \exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(0, \mathsf{Nil})) \quad (8.9)$$

using 720 tactic applications on the current state. The number of tactic applications is very high since the proof due for reducing contains less than 90 proof structors, and less than 15 redices are reduced during the proof reduction. With that explosion in amount of tactic applications it may be overwhelming to do a proof that includes more than one element, therefore this is not illustrated with this theory.

8.6.3 Reducing Applications using $\mathcal{PE}_{\ldots\,\mathcal{SL}}$

The proof `insort` is in weak head normal form and therefore already reduced using formal theory $\mathcal{PE}_{\ldots\,\mathcal{SL}}$. The proof can be applied on different lists to obtain sorted lists by reducing the applications. Below this is illustrated by some examples. To reduce the examples to fit into

theorem schema (8.8) the ordering relation for natural numbers must be included, therefore the below reductions are done using object logic `PEInsortNat`.

Application of `insort` on the list Nil reduces to the theorem

$$\texttt{[Nil,<SoInj(SoSingle),PeNil>]} : \exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{Nil})$$

using 9 tactic applications on the current state. Application of `insort` on the list $\mathsf{C}(0, \mathsf{Nil})$ reduces to the theorem

```
[C(0, Nil),
 <fst(<SoSingle,lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(0, Nil)>),
  PeTrans(
   PeSym(snd(<SoSingle,lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(0, Nil)>)),
   PeInj(snd(<SoInj(SoSingle),PeNil>)))>]
```

$$: \exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(0, \mathsf{Nil})) \quad (8.10)$$

using 67 tactic applications. The proof is in weak head normal form and would easily be reduced to normal form using formal theory $\mathcal{PR}_{\ldots\ \mathcal{SL}}$ for this reduction, which gives the proposition (8.9).

Demonstrating the sorting abilities can be done by demonstrating sorting of lists $\mathsf{C}(\mathsf{S}(0), \mathsf{C}(0, \mathsf{Nil}))$ and $\mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil}))$ that both must reduce to the list $\mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil}))$. Sorting application of `insort` on list $\mathsf{C}(\mathsf{S}(0), \mathsf{C}(0, \mathsf{Nil}))$ reduces to a proposition with logic part

$$\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(\mathsf{S}(0), \mathsf{C}(0, \mathsf{Nil})))$$

and proof

```
val PEs1 =
  "[C(0, C(S(0), Nil)),
    <fst
      (<SoNLe
         (SoSingle, lam x. nleq(x),
          snd
           (<SoSingle,lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(S(0), Nil)>),
          fst
           (<SoSingle,lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(S(0), Nil)>)),
        PeTrans
         (PeAssoc,
          PeInj
           (snd
             (<SoSingle,
               lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(S(0), Nil)>)))>),
     PeTrans
      (PeSym
        (snd
          (<SoNLe
             (SoSingle, lam x. nleq(x),
              snd
               (<SoSingle,
                 lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(S(0), Nil)>),
              fst
               (<SoSingle,
                 lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(S(0), Nil)>)),
            PeTrans
```

```
            (PeAssoc,
             PeInj
              (snd
                (<SoSingle,
                  lrec(PeNil, all x xa. lam x. PeInj(x)) ^
                  C(S(0), Nil)>))))>)),
      PeInj
       (snd
         (<fst(<SoSingle,lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(0, Nil)>),
           PeTrans
            (PeSym
              (snd
                (<SoSingle,
                  lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(0, Nil)>)),
             PeInj(snd(<SoInj(SoSingle),PeNil>))))>))))>]
```

that can be reduced to normal form

```
[C(0, C(S(0), Nil)),
 <SoNLe(SoSingle, lam x.  nleq(x), PeInj(PeNil), SoSingle),
  PeTrans                                                          (8.11)
   (PeSym(PeTrans(PeAssoc, PeInj(PeInj(PeNil)))),
    PeInj(PeTrans(PeSym(PeInj(PeNil)), PeInj(PeNil))))>]
```

Sorting application of `insort` on list $\mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil}))$. reduces to a proof with logic part

$$\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil})))$$

and proof

```
[C(0, C(S(0), Nil)),
    <fst
      (<SoLeq(SoSingle, leq),
        lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(0, C(S(0), Nil))>),
     PeTrans
      (PeSym
        (snd
          (<SoLeq(SoSingle, leq),
            lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(0, C(S(0), Nil))>)),
       PeInj
        (snd
          (<fst
             (<SoSingle,
               lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(S(0), Nil)>),
            PeTrans
             (PeSym
               (snd
                 (<SoSingle,
                   lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(S(0), Nil)>)),
              PeInj(snd(<SoInj(SoSingle),PeNil>))))>))))>]
```

that can be reduced to normal form

```
[C(0, C(S(0), Nil)),
 <SoLeq(SoSingle, leq),
  PeTrans                                                          (8.12)
   (PeSym(PeInj(PeInj(PeNil))),
    PeInj(PeTrans(PeSym(PeInj(PeNil)), PeInj(PeNil))))>]
```

The above examples of reduction results are both shown in weak head normal form and normal form. Theory $\mathcal{PE}_{\ldots\ \mathcal{SL}}$ cannot reduce the proofs to normal form, therefore the weak head normal form proofs are the outcome of reduction with object logic `PEInsortNat`. The normal forms proofs shows how the reduction results looks when they are reduced to normal form since then it is easier to see whether the reduction is correct. The normal form proofs are created by reducing the weak head normal form proofs using the object logic `PRInsortNat`.

With the above examples, sorting an empty list, a singleton list, a list containing elements that are not ordered by $\leq$ and a list containing elements that are ordered by $\leq$, it is demonstrated that the insertion sort proof really is a program that do sort lists.

8.7 Automatic Normalizing of Insertion Sort Proofs

Normalizers can be created for proof reduction and program evaluation using the context and reduction rules of the object logics presented in sections 8.5.3 and 8.6.1.

8.7.1 Proof Reduction Normalizer

The proof reduction normalizer is constructed with the object logic below. The object logic can be used to reduce proofs like it is done by hand in section 8.6.2. Notice that the normalizer below is not built on top of a normalizer for lists, but on top of the first order logic normalizer.

File `PRInsortNatauto.thy`

```
(* Time-stamp: <1999-02-21 16:38:08 kokdg> *)

PRInsortNatauto = PRInsortNat + PRauto
```

File `PRInsortNatauto.ML`

```
(* Time-stamp: <1999-02-21 16:44:55 kokdg> *)

open PRInsortNatauto;

  (* Nat-rules *)
  Norm.addinfs [PRNAT.inductionCI, S_injectCI, S_neq_0CI];
  Norm.addaxs [rec_0CI, rec_SCI];
  Norm.addreds [inductionR0, inductionRS];
```

```
  (* List-rules *)
  Norm.addinfs [inductionCI,C_inject1CI, C_inject2CI, C_neq_NilCI];
  Norm.addaxs [recl_NilCI, recl_CCI];
  Norm.addreds [inductionRNil, inductionRC];

  (* NatOrder-rules *)
  Norm.addinfs [ZnleCI,OrdinjCI,LessCI,Less2CI, OrderCCI];
  Norm.addaxs [ZleCI];
  Norm.addreds [];

  (* Insort-rules *)
  Norm.addinfs [PeSymCE, SoInjCE, OrdDecideCE, PeTransCI, PeInjCI,
                SoAddLeCI, SoAddNLeCI];
  Norm.addaxs [PeNilCI, PeAssocCI, SoSingleCI, OrdReflCI, OrdAnSymCI,
               OrdTransCI, OrdTotalCI];
  Norm.addreds [SoInjLeR, SoInjNLeR, PeSymR];

val norma = Norm.get_tactic();

val complete_norma = (REPEAT norma) THEN (ALLGOALS assume_tac);
```

Reduced Proofs

The normalizer is applied to proofs that are reduced in section 8.6.2. Before solving subgoals that only consist of hypotheses, the below proofs are created. Solving subgoals consisting of only hypotheses are not done because Isabelle have some problems unifying logic parts of the proofs.

The normalizer give the below results.

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{Nil})$ reduces to proof

  ```
  [Nil,<SoInj(SoSingle),PeNil>]
  ```

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(0, \mathsf{Nil}))$ reduces to

  ```
  [C(0,Nil),<SoSingle,PeTrans(PeSym(PeInj(PeNil)), PeInj(PeNil))>]
  ```

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(\mathsf{S}(0), \mathsf{C}(0, \mathsf{Nil})))$ reduces to

  ```
  [C(0,C(S(0),Nil)),
   <SoNLe(SoSingle, lam x.  nleq(x), PeInj(PeNil), SoSingle),
    PeTrans(PeSym(PeTrans(PeAssoc, PeInj(PeInj(PeNil)))),
            PeInj(PeTrans(PeSym(PeInj(PeNil)), PeInj(PeNil))))>]
  ```

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil})))$ reduces to

  ```
  [C(0,C(S(0),Nil)),
   <SoLeq(SoSingle, leq),
    PeTrans(PeSym(PeInj(PeInj(PeNil))),
            PeInj(PeTrans(PeSym(PeInj(PeNil)), PeInj(PeNil))))>]
  ```

The first two proofs are equal with the result proofs of section 8.6.2 and may thereby be correct. The last two proofs cannot be compared with hand made proofs since such proofs have not been done; by inspecting the proofs seem to be correct.

8.7.2 Program Evaluation Normalizer

The program evaluation normalizer is constructed with the object logic below. The object logic can be used to evaluate programs like it is done by hand in section 8.6.3, returning the same results. Notice that the normalizer below is not built on top of a normalizer for lists, but on top of the first order logic normalizer.

File PEInsortNatauto.thy

```
(* Time-stamp: <1999-02-21 17:28:09 kokdg> *)

PEInsortNatauto = PEInsortNat + PEauto
```

File PEInsortNatauto.ML

```
(* Time-stamp: <1999-02-21 17:27:29 kokdg> *)

open PEInsortNatauto;

  (* Nat-rules *)
  Norm.addinfs [];
  Norm.addaxs [];
  Norm.addreds [inductionR0, inductionRS];

  (* List-rules *)
  Norm.addinfs [];
  Norm.addaxs [];
  Norm.addreds [inductionRNil, inductionRC];

  (* NatOrder-rules *)
  Norm.addinfs [];
  Norm.addaxs [];
  Norm.addreds [];

  (* Insort-rules *)
  Norm.addinfs [PeSymCE, SoInjCE, OrdDecideCE];
  Norm.addaxs [];
  Norm.addreds [SoInjLeR, SoInjNLeR, PeSymR];

val norma = Norm.get_tactic();

val complete_norma = (REPEAT norma) THEN (ALLGOALS assume_tac);
```

Evaluated Programs

The normalizer is applied to programs that are evaluated in section 8.6.3. Before solving subgoals that only consist of hypotheses, the below proofs are created. Solving subgoals consisting of only hypotheses are not done because Isabelle have some problems unifying logic parts of the proofs.

The normalizer give the below results.

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{Nil})$ reduces to proof

  $$[\mathsf{Nil}, \texttt{<SoInj(SoSingle),PeNil>}]$$

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(0, \mathsf{Nil}))$ reduces to

```
[C(0,Nil),
 <fst(<SoSingle,lrec(PeNil, all x xa. lam x. PeInj(x)) ^ C(0, Nil)>),
  PeTrans(PeSym(snd(<SoSingle,
                     lrec(PeNil,all x xa. lam x. PeInj(x)) ^ C(0, Nil)>)),
          PeInj(snd(<SoInj(SoSingle),PeNil>)))>]
```

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(\mathsf{S}(0), \mathsf{C}(0, \mathsf{Nil})))$ reduces to a huge proof term

  $$[\mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil})), \texttt{< ... , ... >}]$$

  where ... represent proofs of sorting and permutation that are not reduced since the term is in weak head normal form.

- The proof of proposition $\exists m.\mathcal{S}_{\prec}(m) \wedge \mathcal{P}(m, \mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil})))$ reduces to a huge proof term

  $$[\mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil})), \texttt{< ... , ... >}]$$

  where ... represent proofs of sorting and permutation that are not reduced since the term is in weak head normal form.

The first two proofs are equal with the result proofs of section 8.6.3 and may thereby be correct. The last two proofs cut out above are equal to the proofs in section 8.6.3.

8.7.3 Time Measurements

The time used by Isabelle for reducing proofs and evaluating programs can be compared. The comparing gives the scheme on figure 8.5. It shows that time used on proof reduction are much bigger than time spent on program evaluation.

Figure 8.5 Measuring the difference on proof reduction and program evaluation.

List	Reduction/sec	Evaluation/sec	Factor
Nil	28.6	0.4	72
$\mathsf{C}(0, \mathsf{Nil})$	359.6	0.9	400
$\mathsf{C}(\mathsf{S}(0), \mathsf{C}(0, \mathsf{Nil}))$	2059.6	4.7	438
$\mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil}))$	1877.5	3.1	606
$\mathsf{C}(\mathsf{S}(\mathsf{S}(0)), \mathsf{C}(0, \mathsf{C}(\mathsf{S}(0), \mathsf{Nil})))$	?	18.3	?

That the difference is so big may be because the algorithm used for proof normalization is not good enough.

8.8 Conclusion On case Study

The case study shows that for a given case from real life a proof of a first order logic proposition can be used as a program for deriving results. The examples show that normalization results contain

more information than results of evaluating programs in a normal programming language. The normalization results above contain, out over the sorted lists, proofs as documentation why the lists are sorted and documentation how to obtain the sorted lists from the original lists. The part of a proposition that seems to be crucial for obtaining a result, as it would come from a programming language, is the $\exists$ connective that ensures that a withness for the program is returned.

The case study also shows that even though proof reduction give proofs without redexes, proof reduction should not be used for interpreting programs, simply because it takes all too many evaluation steps to reduce proofs to normal form.

The case study gives an idea that if a proof term must be reduced to normal form with the formal theories constructed in this thesis, it may be an advantage to reduce it to weak head normal form before reducing the outcome hereof to normal form.

Chapter 9

Problems, Obtained Results and Further Work

This chapter describes which problems in the thesis that have not been solved yet, which results have been obtained, what can be done with these results and what related work that could be done in the future.

9.1 Inconveniences in Isabelle, Object Logics and Formal Theories

There have been some problems using Isabelle, but not bigger than it has been possible to get around them or solve them. As well, some formal theories could be constructed in a better way. Object logics may, as all other programs, contain undiscovered errors.

Problems that are not solved or have influence on parts of this thesis are described below.

9.1.1 Isabelle and Unifying Variables

Isabelle has a problem unifying variables. Some unifications create flex-flex pairs that are pairs of variables that cannot be unified without information loss. The problem is illustrated by inference rule $\forall$CE, i.e. as demonstrated in the example in chapter B.3.2 on page 216. It is not always possible to get these variables unified without reaching Isabelle's unification bound.

The problem is that some object logic rules creating flex-flex pairs by application is wrongly constructed, as written below. Sadly, I was not aware of that. But the flex-flex pairs do not only arise using these rules, they may also arise from solving by hypothesis.

The problem with unification made it impossible to obtain reduced proofs using the $\mathcal{PR}$ ... $\mathcal{SL}$ theory. The premise making proposition unification delaying possible in the $\forall$CE rule was removed in object logic `PRInsort` such that the unification did not happen.

It can be discussed whether or not it is a good idea to let types for program terms be parts of redexes to reduce, if the program to be reduced is type correct. Without types unification would not be a problem. On the other hand an object logic may contain rules that reduce a program such that it is no longer type correct. Therefore types must be included.

9.1.2 Errors and Inconveniences in Formal Theories

In formal theory $\mathcal{SL}$ and theories and object logics following here from the fact that sorting is *surjective* is represented by rules indicating that sorting is injective. It would have too much influence on the documentation to correct this name error.

Object logic `PRInsort` specialised to natural numbers with object logic `NatOrder` specifies that decidability is specialised during reduction. The proof and reduction would be much more beautiful using currying and thus letting the decidability proof be an argument of the sorting proof.

9.1.3 Errors in Object Logics

The following errors have been discovered in object logics:

`PR` had had an error in the definition of tactic `vartac`, which has been corrected. Subgoals of type `x >> ?y : ?P` were solved completely such that the subgoal solving did not result in a new subgoal `x : ?P` as required by rule (4.12). Most of the documentation in appendix B is therefore somewhat erroneous: each subgoal solving by tactic application

`by (vartac` n`)`

must be replaced with tactic application

`by ((vartac` n`) THEN ((assume_tac ORELSE' resolve_tac premises()` n`))`

where `premises()` is a function returning hypotheses stated with the `goal` command. It would take too much time to correct re-create the documentation[1].

`InsortProof` **and** `PRInsort` have the rules `PeNil : Pe(l,l)` and `PeNil >> PeNil : Pe(l,l)` where `l` should be `Nil`. The problem is that `PeNil` proves that any list is a permutation of itself, which is correct, but it should not be shown by `PeNil`.

`Reduction` contains rules `allR` and `allCE` where there is problems with a predicate variable. All occurences if predicate variable `P(a)` and `P(b)` must be replaced with `P` in these two rules, and ten they will implement the rules in $\mathcal{PR}_{\mathcal{IFOLP}}$ and $\mathcal{PE}_{\mathcal{IFOLP}}$. This correction will remove some flex-flex pairs, but not all of them.

9.2 Results Obtained in This Thesis

Proofs and theorems were in chapter 3 constructed to be syntactically equal to program terms and types. In chapter 4, equality were shown for the semantics of proofs and program terms. This connection was used to obtain the below results.

A formal theory for proof reduction was developed such that proofs reduce by leftmost-innermost reduction; strategies for λ-calculus were used on proofs. The proof reduction can be used to obtain a proof without redundant proof steps. Proof reduction was created such that it became possible to automate the reduction process by repeating an unintelligent application of the same tactic until all redexes were solved.

As well, a formal theory for program evaluation was developed such that programs evaluate by resolving subgoals using a given set of rules corresponding to eager evaluation; strategies for proof

[1] It could of course be done with search-and-replace, but that's not my style.

resolving were used on programs. The program evaluation can be used to obtain a program term in weak head normal form. Program evaluation was also created such that it became possible to automate the evaluation process like with proof reduction.

Because of the similarity of proof reduction and program evaluation, these two techniques were implemented in object logics with the same reduction basis. Also, a meta normalizer was implemented as an object logic that could be specified to do either program evaluation or proof reduction.

Proof reduction and program evaluation were evaluated in chapter 8 where a proof of a proposition should turn out to be a program implementing insertion sort. That case study shows that it is correct. It also seems to be correct that program evaluation and proof reduction give the wanted results.

The case study in chapter 8 gave up on manually reducing proof of sorting a list with two or more elements since it simply took too much time to do the reductions. Program evaluation of the same proofs took not nearly as much time. The normal forms of these proofs were obtained by first reducing to weak head normal form which then were reduced to normal form.

These two-step reductions took not nearly as much time as reduction directly to normal form since many context rule applications were not done with weak head reduction. Therefore a reduction strategy reducing firstly to weak head normal form and then to normal form may be the next strategy to implement, since it at a glance seems somewhat more effective and thereby usable.

9.3 Further Work

With proof reduction implemented, program extraction would be an interesting subject. Program extraction could possibly be implemented by taking basis in [1].

Another interesting subject would be to implement a reduction strategy for proof reduction that does not consume so much time. One reduction strategy is proposed above, and many others can possibly be found in the literature.

As well, it would be interesting to develop the same proof/program connection for other logics than first order logic. A first attempt may be to study why Girard states [6, p. 28] that the connection cannot be stated for the sequent calculus.

Part III

Appendices

Appendix A

Isabelle Theory Files

This appendix contains the files used to implement formal theories in object logics.

A.1 Starting Isabelle

Isabelle is normally started by executing an executable produced by `make` for a decent object logic. If Isabelle shall be started up without any object logic just implementing the meta logic the below sml program, where `../src` is the path for the Isabelle distribution files, can be executed in an sml of New Jersey interpreting using the old definition of standard ml, i.e. version 0.93.

```
val cd = System.Directory.cd;
fun pwd () = System.Directory.getWD ();

val curr_dir = pwd();
  cd "../src/Pure";
  use "NJ.ML";        (* Use sml/NJ specific parts *)
  use "ROOT.ML";      (* The Isabelle system witl logic Pure *)
  cd curr_dir;
  init_thy_reader();
  print_depth 100;
```

The meta logic can be included in user defined object logics as object logic `Pure`. (It is not a good idea not to include it – it defines the connectives used for stating rules, whereby it is hard to define anything without it).

A.2 First Order Logic

First order logic represented by formal theory $\mathcal{IFOL}$ is implemented by using a part of the original work by Larry Paulson, represented in files `IFOL.thy` and `IFOL.ML` of the Isabelle distribution package. These parts used for implementing the object logic representing formal theory $\mathcal{IFOL}$ is object logic `FirstOrder` in files `FirstOrder.thy` and `FirstOrder.ML`.

A.2.1 Theory File `FirstOrder.thy`

```
(* Time-stamp: <1999-02-07 17:22:23 kokdg> *)

(* Theory for intuitionistic first order logic using natural *)
(* deduction. *)

(* THIS FILE IS AN EXTRACT OF THE FILE FOL/IFOL.thy IN THE ISABELLE *)
(* DISTRIBUTION PACKAGE! *)

(* Original file is copyrighted by Larry Paulson, 1993, modified and *)
(* extracted by Bo Kjellerup, 1998 *)

FirstOrder = Pure +

classes
  term < logic

default
  term

types
  o

arities
  o :: logic


consts

  Trueprop      :: "o => prop"                  ("(_)" 5)

  (* Reserved predicate letters *)

  False         :: "o"
  "="           :: "['a, 'a] => o"              (infixl 50)

  (* Connectives *)

  "&"           :: "[o, o] => o"                (infixr 35)
  "|"           :: "[o, o] => o"                (infixr 30)
  "--->"        :: "[o, o] => o"                (infixr 25)

  (* Quantifiers *)

  All           :: "('a => o) => o"             (binder "ALL " 10)
  Ex            :: "('a => o) => o"             (binder "EX " 10)


rules

  (* Equality *)
```

```
  refl          "a=a"
  subst'        "[| a=b;  P(a) |] ==> P(b)"

  (* Propositional logic *)

  conjI         "[| P;  Q |] ==> P&Q"
  conjunct1     "P&Q ==> P"
  conjunct2     "P&Q ==> Q"

  disjI1        "P ==> P|Q"
  disjI2        "Q ==> P|Q"
  disjE         "[| P|Q;  P ==> R;  Q ==> R |] ==> R"

  impI          "(P ==> Q) ==> P-->Q"
  mp            "[| P-->Q;  P |] ==> Q"

  FalseE        "False ==> P"

  (* Quantifiers *)

  allI          "(!!x. P(x)) ==> (ALL x.P(x))"
  spec          "(ALL x.P(x)) ==> P(x)"

  exI           "P(x) ==> (EX x.P(x))"
  exE           "[| EX x.P(x);  !!x. P(x) ==> R |] ==> R"

end
```

A.2.2 Theory File `FirstOrder.ML`

```
(* Time-stamp: <1999-02-07 17:22:45 kokdg> *)

(* Theory for intuitionistic first order logic using natural *)
(* deduction. *)

(* THIS FILE IS AN EXTRACT OF THE FILE FOL/IFOL.ML IN THE ISABELLE *)
(* DISTRIBUTION PACKAGE! *)

(* Original file is copyrighted by Larry Paulson, 1993, modified and *)
(* extracted by Bo Kjellerup, 1998 *)

open FirstOrder;

(*** Sequent-style elimination rule for ALL ***)

qed_goal "allE" FirstOrder.thy
    "[| ALL x.P(x); P(x) ==> R |] ==> R"
 (fn prems=> [ (REPEAT (resolve_tac (prems@[spec]) 1)) ]);

(*** Equality rules ***)

qed_goal "subst" FirstOrder.thy
```

```
"[| x = y ; !!x. P(x,x) ; P(x,y) ==> R |] ==> R"
  (fn prems => [REPEAT (resolve_tac (prems@[subst']) 1)]);

qed_goal "sym" FirstOrder.thy "a=b ==> b=a"
  (fn [major] => [rtac subst 1, rtac major 1, atac 2, rtac refl 1 ]);

qed_goal "trans" FirstOrder.thy "[| a=b;  b=c |] ==> a=c"
  (fn [prem1,prem2] => [rtac mp 1, rtac prem1 2, rtac subst 1, rtac
prem2 1, atac 2, rtac impI 1, atac 1]);

(** Polymorphic congruence rules **)

qed_goal "subst_context" FirstOrder.thy
   "[| a=b |]  ==>  t(a)=t(b)"
 (fn prems=>
  [ (resolve_tac (prems RL [sym RS subst]) 1),
    (resolve_tac [refl] 1), (assume_tac 1) ]);
```

A.3 First Order Logic with Proofs

First order logic represented by formal theory $\mathcal{IFOLP}$ is implemented using a part of the original work by Larry Paulson, represented in files `IFOLP.thy` and `IFOLP.ML` in the Isabelle distribution package. These parts used for implementing the object logic representing formal theory $\mathcal{IFOLP}$ is in object logic `FirstOrderProof` in files `FirstOrderProof.thy` and `FirstOrderProof.ML`.

A.3.1 Theory File `FirstOrderProof.thy`

```
(* Time-stamp: <1999-02-07 17:23:28 kokdg> *)

(* Theory for intuitionistic first order logic using natural deduction *)
(* with proof terms *)

(* THIS FILE IS AN EXTRACT OF THE FILE FOLP/IFOLP.thy IN THE ISABELLE *)
(* DISTRIBUTION PACKAGE! *)

(* Original file is copyrighted by Martin D Coen, Cambridge 1992, *)
(* extracted and modified by Bo Kjellerup, 1998 *)

FirstOrderProof = Pure +

classes term < logic

default term

types
  p
  o

arities
  p,o :: logic
```

```
consts
      (*** Judgements ***)
 "@Proof"       ::   "[p,o]=>prop"      ("(_ /: _)" [51,10] 5)
 Proof          ::   "[o,p]=>prop"

      (*** Reserved Predicate letters ***)

 False          ::      "o"
 "="            ::      "['a,'a] => o"  (infixl 50)

      (*** Logical Connectives ***)

 "&"            ::      "[o,o] => o"    (infixr 35)
 "|"            ::      "[o,o] => o"    (infixr 30)
 "-->"          ::      "[o,o] => o"    (infixr 25)

      (*Quantifiers*)
 All            ::      "('a => o) => o"        (binder "ALL " 10)
 Ex             ::      "('a => o) => o"        (binder "EX " 10)

      (*** Proof Term Formers: precedence must exceed 50 ***)

 contr          :: "p=>p"
 fst,snd        :: "p=>p"
 pair           :: "[p,p]=>p"           ("(1<_,/_>)")
 split          :: "[p, [p,p]=>p] =>p"
 inl,inr        :: "p=>p"
 when           :: "[p, p=>p, p=>p]=>p"
 lambda         :: "(p => p) => p"      (binder "lam " 55)
 "`"            :: "[p,p]=>p"           (infixl 60)
 alll           :: "['a=>p]=>p"         (binder "all " 55)
 "^"            :: "[p,'a]=>p"          (infixl 55)
 exists         :: "['a,p]=>p"          ("(1[_,/_])")
 xsplit         :: "[p,['a,p]=>p]=>p"
 ideq           :: "'a=>p"
 idpeel         :: "[p,'a=>p]=>p"


rules

(**** Propositional logic ****)

(*Equality*)

ieqI      "ideq(a) : a=a"
ieqE      "[| p : a=b;  !!x.f(x) : P(x,x) |] ==> idpeel(p,f) : P(a,b)"

(* Falsity *)

FalseE    "a:False ==> contr(a):P"

(* Conjunction *)
```

```
conjI     "[| a:P;  b:Q |] ==> <a,b> : P&Q"
conjunct1 "p:P&Q ==> fst(p):P"
conjunct2 "p:P&Q ==> snd(p):Q"

(* Disjunction *)

disjI1    "a:P ==> inl(a):P|Q"
disjI2    "b:Q ==> inr(b):P|Q"
disjE     "[| a:P|Q;  !!x.x:P ==> f(x):R;  !!x.x:Q ==> g(x):R |]
           ==> when(a,f,g):R"

(* Implication *)

impI      "(!!x.x:P ==> f(x):Q) ==> lam x.f(x):P-->Q"
mp        "[| f:P-->Q;  a:P |] ==> f`a:Q"

(*Quantifiers*)

allI      "(!!x. f(x) : P(x)) ==> all x.f(x) : ALL x.P(x)"
spec      "(f:ALL x.P(x)) ==> f^x : P(x)"

exI       "p : P(x) ==> [x,p] : EX x.P(x)"
exE       "[| p: EX x.P(x);  !!x u. u:P(x) ==> f(x,u) : R |] ==>
           xsplit(p,f):R"

(* HELP FOR IMPLEMENTATION AND VERIFICATION -- NOT A PART OF THE REAL *)
(* THEORY !!! *)

double    "[| p : P ; p : P |] ==> p : P"


end

ML

(*show_proofs:=true displays the proof terms*)
val show_proofs = ref true;

fun proof_tr [p,P] = Const("Proof",dummyT) $ P $ p;

fun proof_tr' [P,p] =
    if !show_proofs then Const("@Proof",dummyT) $ p $ P
    else P  (*this case discards the proof term*);

val parse_translation = [("@Proof", proof_tr)];
val print_translation  = [("Proof", proof_tr')];
```

A.3.2 Theory File `FirstOrderProof.ML`

```
(* Time-stamp: <1999-02-07 17:23:47 kokdg> *)

(* Theory for intuitionistic first order logic using natural deduction *)
```

```
(* with proof terms *)

(* THIS FILE IS AN EXTRACT OF THE FILE FOLP/IFOLP.thy IN THE ISABELLE *)
(* DISTRIBUTION PACKAGE! *)

(* Original file is copyrighted by Martin D Coen, Cambridge 1992, *)
(* extracted and modified by Bo Kjellerup, 1998 *)

open FirstOrderProof;

(*** Sequent-style elimination rules for ALL ***)

val allE = prove_goal FirstOrderProof.thy
"[| p:ALL x.P(x); !!y. y:P(z) ==> y:R |] ==> ?r:R"
  (fn prems=> [ (REPEAT (resolve_tac (prems@[spec]) 1)) ]);

(*** Equality rules ***)

val refl = ieqI;

(* The subst rule below may not be correct yet! *)

val subst = prove_goal FirstOrderProof.thy
  "[| p : x = y; !!x. f(x) : P(x,x); !!q. q:P(x,y) ==> q: R|] ==> ?r: R"
    (fn [p1,p2,p3] => [rtac p3 1, rtac ieqE 1, rtac p1 1, rtac p2
    1]);

val sym = prove_goal FirstOrderProof.thy "q:a=b ==> ?c:b=a"
 (fn [major] => [rtac subst 1, rtac major 1, atac 2, rtac refl 1 ]);

val trans = prove_goal FirstOrderProof.thy "[| p:a=b;  q:b=c |] ==> ?d:a=c"
 (fn [prem1,prem2] => [rtac mp 1, rtac prem1 2, rtac subst 1, rtac
       prem2 1, atac 2, rtac impI 1, atac 1]);


(** Polymorphic congruence rules **)

qed_goal "subst_context" FirstOrderProof.thy
   "[| p : a=b |]  ==>  ?q : t(a)=t(b)"
 (fn prems=>
  [ (resolve_tac (prems RL [sym RS subst]) 1),
    (resolve_tac [refl] 1), (assume_tac 1) ]);
```

A.4 Basic Reduction

Proof reduction and proof evaluation as theories $\mathcal{PR}_{\mathcal{IFOLP}}$ and $\mathcal{PE}_{\mathcal{IFOLP}}$ as presented in sections 4.7.3 and 4.7.4 share some set of rules. These shared rules are implemented in object logic `Reduction`. The object logic `FirstOrderProof` in which proofs are created is included in the object logic for basic reduction.

Some shared rules have some properties and conditions as specified in section 4.7.3 and 4.7.4 which shall be implemented:

- Rule IsNorm must not bebe applied on a state before all reduction rules have been tried and failed. Therefore the rule may not occur in the signature for the object logic and may only be included as the last reduction rule tried in a tactic trying resolving with all reduction rules.

Notice that properties and conditions are implemented using ML structures and signatures eliminating what may not be user visible from the Isabelle generated structure `Reduction`. Rules are restricted using tactic generators and tactics.

A.4.1 Theory File `Reduction.thy`

```
(* Time-stamp: <1999-01-14 10:32:01 kokdg> *)

(* Theory for reduction on proof terms of the FirstOrderProof object *)
(* logic. *)

(* This file contains
   -  reduction rules,
   -  context rules for destructors,
   -  reduction elimination rule,
   context rules for constructors depends on the chosen reduction
   method. *)

(* This file may be parted in minor parts if either reduction rules or
   destructor context rules shall be altered for other reduction
   methods. *)


Reduction = FirstOrderProof +

consts

Redex   :: "[p,p,o] => prop" ("(3_ /!> _ :/ _)" [51,51,10] 5)
Term    :: "[p,p,o] => prop" ("(3_ />> _ :/ _)" [51,51,10] 5)

rules

(* Reduction rules for redexes *)

impR       "f(e) >> r : P ==> (lam x. f(x)) ` e !> r : P"

conjR1     "a >> r : P ==> fst(<a,b>) !> r : P"
conjR2     "b >> r : P ==> snd(<a,b>) !> r : P"

allR       "[| f(b) >> r : R(b) ; !! y. y : R(a) ==> y : P(a) |] ==>
            (all a. f(a)) ^ b !> r : P(b)"

exR        "f(a,p) >> r : P ==> xsplit([a,p],f) !> r : P"

disjR1     "f(a) >> r : P ==> when(inl(a),f,g) !> r : P"
disjR2     "g(b) >> r : P ==> when(inr(b),f,g) !> r : P"
```

```
substR     "[| f(a) >> r : R(a,a) ; !!y. y : R(b,b) ==> y : P(b,b)|]
            ==> idpeel(ideq(a), f) !> r : P(a,a)"

(* Reduction rule for non-redexes. May NOT be used before any of the rules *)
(* for redexes *)

IsNorm     "e !> e : P"

(* Context rules for destructors *)

impCE      "[| e1 >> e1' : Q --> P ; e2 >> e2' : Q ; e1' ` e2' !> r : P |]
            ==> e1 ` e2 >> r : P"

conjCE1    "[| e >> e' : P & Q ; fst(e') !> r : P |] ==> fst(e) >> r : P"
conjCE2    "[| e >> e' : P & Q ; snd(e') !> r : Q |] ==> snd(e) >> r : Q"

allCE      "[| f >> g : ALL b. R(b) ; g ^ a !> r : R(a) ;
               !!y. y : R(a) ==> y : P(a) |] ==> f ^ a >> r : P(a)"

exCE       "[| p >> q : EX a. P(a) ;
               !!x u. u : P(x) ==> f(x,u) >> g(x,u) : R ;
               xsplit(q,g) !> r : R |] ==> xsplit(p,f) >> r : R"

disjCE     "[| p >> q : P | Q ; !!x. x : P ==> f(x) >> f'(x) : R ;
               !!x. x : Q ==> g(x) >> g'(x) : R ; when(q, f', g') !> r : R |]
            ==> when(p, f, g) >> r : R"

substCE    "[| p >> q : a = b; !!y. y : P(a,b) ==> y : R;
               !!c. f(c) >> g(c) : P(c,c);
               idpeel(q,g) !> r : P(a,b) |] ==> idpeel(p,f) >> r : R"

(* Reduction Elimination *)

redE       "[| a : P ; a >> p : P |] ==> p : P"

end
```

A.4.2 Theory File `Reduction.ML`

```
(* Time-stamp: <1999-01-06 15:59:57 kokdg> *)

signature Reduction =
  sig
    val allR : thm
    val conjR1 : thm
    val conjR2 : thm
    val disjR1 : thm
    val disjR2 : thm
    val exR : thm
    val impR : thm
    val substR : thm
    val allCE : thm
```

```
    val conjCE1 : thm
    val conjCE2 : thm
    val disjCE : thm
    val exCE : thm
    val impCE : thm
    val redE : thm
    val substCE : thm
    val try_all_rules : int -> tactic
    val bt : int -> unit
    val thy : theory
  end

local
  structure Red = Reduction
in
  structure Reduction : Reduction =
    struct
      open Red
      (* tactic using all reduction rules before IsNorm *)
      val try_all_rules = resolve_tac [impR, conjR1, conjR2, allR,
                                       exR, disjR1, disjR2, substR,
                                       IsNorm]
      fun bt n = by (try_all_rules n) (* short-hand like br, etc... *)
    end
end (*local *)

open Reduction;
```

A.5 Reducing Proofs in First Order Logic with Equality and Proofs

The object logic implementing formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$ includes the basic reduction object logic `Reduction`. Specific rules for proof reduction are in object logic `PR` which includes

- context rules for constructors,
- a rule for accepting proofs as reduced with the condition that the proof is a variable. The proof accepting rule may not applicable on other proofs than bound variables, therefore the rule must be encapsulated in a pattern matching that ensures that the goal on which the rule is applied has the correct form.

  A goal that can be accepted as being reduced consists of a series of quantified variables (which is at least the variable representing the reduced proof), and either a meta implication applied on some premises and the conclusion or else the conclusion. The conclusion consists of a proposition part and a proof part, and the proof part consists of the $\mid\!\longrightarrow$ connective representation `>>` having a bound variable as first argument; the proof is a free or bound variable and is thereby reduced.

Notice that properties and conditions are implemented using ML structures and signatures, eliminating what may not be user visible from the Isabelle generated structure `PR`, and restricting rules using tactic generators, tactics and formula destruction functions.

A.5.1 Theory File PR.thy

```
(* Time-stamp: <1999-01-03 14:16:24 kokdg> *)

PR = Reduction +

rules

(* Context rules for constructors *)

impCI      "(!!y. y : P ==> f(y) >> g(y) : Q) ==>
            lam x. f(x) >> lam x. g(x) : P --> Q"

conjCI     "[| a >> a' : P ; b >> b' : Q |] ==> <a,b> >> <a',b'> : P & Q"

allCI      "(!!y. f(y) >> g(y) : P(y)) ==>
            all x. f(x) >> all x. g(x) : ALL z. P(z)"

exCI       "p >> q : P(x) ==> [x,p] >> [x,q] : EX y. P(y)"

disjCI1    "a >> a' : P ==> inl(a) >> inl(a') : P | Q"
disjCI2    "b >> b' : Q ==> inr(b) >> inr(b') : P | Q"

substCI    "ideq(a) >> ideq(a) : a = a"

(* Variable acceptance, part of rule redI implemented as vartac, may
   NOT occur as usable for the user *)

redIFV      "x >> x : P"
redIBV      "x : P ==> x >> x : P"

end
```

A.5.2 Theory File PR.ML

```
(* Time-stamp: <1999-02-22 14:29:01 kokdg> *)

signature PR =
  sig
    val allCI : thm
    val conjCI : thm
    val disjCI1 : thm
    val disjCI2 : thm
    val exCI : thm
    val impCI : thm
    val substCI : thm
    val thy : theory
    val vartac : int -> tactic
    val bv : int -> unit
  end

local
```

```
  structure PR' = PR
in
  structure PR : PR =
    struct
      open PR'
      (* strip_imps_body returns the meta conclusion of a term *)

      fun strip_imps_body (op$ ( op$ (Const("==>",_), prems), concl))
        = strip_imps_body concl
        | strip_imps_body t = t
      (* split returns the left and right proof parts of the
       conclusion of the term representing the subgoal on which vartac
       is applied. *)
      fun split t =
        let
          val (op$ (op$ ( op$ (termC, left), right), prop)) =
            strip_imps_body (strip_all_body t)
        in
          (left,right)
        end
      fun getLI (left $ right) = getLI left
        | getLI x = x
      (* vartac is rule x >> x : P  where x is either a bound variable
       appearing in the assumptions or a free variable, and therefore
       corrresponds to rule redI for variables *)
      val vartac =
        SUBGOAL
(fn (t,i) =>
 let
   val (proof,_) = split t
   val proofMain = getLI proof
 in
   (case proofMain
      of (Bound _) => (rtac redIBV i) (* THEN (assume_tac i) *)
       | (Free _) =>  (rtac redIBV i) (* was: ... redIFV ... *)
       | _ => no_tac) handle _ => no_tac
 end)
      fun bv n = by (vartac n);   (* short-hand like br, ba, etc... *)
    end (* struct *)
end (* local *)

open PR;
```

A.6 Evaluating Proofs as Programs in First Order Logic with Equality and Proofs

The object logic implementing formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$ includes the basic reduction object logic `Reduction`. Specific rules for proof reduction are in object logic `PE` which includes

- A reduction introduction rule that accepts proofs as being true if and only if they are in weak head normal form.

A proof is in weak head normal form if the proof either has a constructor as root or is a variable. This is the fact if none destructor context rule can be unified with the proof. Therefore the above condition on the reduction introduction rule can be fulfilled if the rule may only be applied after all applications of destructor context rules has failed.

Notice that properties and conditions are implemented using ML structures and signatures, eliminating what may not be user visible from the Isabelle generated structure `PE`, and restricting rules using tactic generators, tactics and formula destruction functions.

A.6.1 Theory File `PE.thy`

```
(* Time-stamp: <1998-12-17 15:59:15 kokdg> *)

PE = Reduction +

rules

redI    "p >> p : P"

end
```

A.6.2 Theory File `PE.ML`

```
(* Time-stamp: <1999-01-05 13:53:10 kokdg> *)

signature PE =
  sig
    val thy : theory
    val try_allRules : int -> tactic
    val bta : int -> unit
  end

local
  structure PE' = PE
in
  structure PE : PE =
    struct
      open PE'
      val try_allRules = resolve_tac [impCE, conjCE1, conjCE2, allCE,
                                      exCE, disjCE, substCE, redI]
      fun bta n = by (try_allRules n)
    end
end

open PE;
```

A.7 Meta Normalizer

A meta normalizer corresponding to the algorithm stated in section 7.2 but specialized to object logics for reduction based on object logic `Reduction` can be constructed like the object logic below.

A.7.1 Implementation of the Algorithm

All reduction rules is specified to be tried in correct order in tactic `try_all_rules` of object logic `Reduction`. If the left proof term of the subgoal on which a reduction rule is applied does not include a schematic variable then only one of the reduction rules except rule IsNorm can be applied on the subgoal. Therefore the reduction rules can be ordered mutually arbitrarily, except that rule `IsNorm` must be applied after all other reduction rules since this rule unifies with any term the other reduction rules unifies with.

Context rules can be ordered arbitrarily. None of the context rules intersects if the subgoal on which the normalizer is applied do not have a schematic variable in the left proof term.

The above discussion leads to a prerequisite on proofs to be normalized. A proof to be normalized (i.e. the left proof term of a stated goal) may not contain any schematic variables.

A tactic for variables must be stated specially since it cannot be created from a rule. As well axioms must be converted to inference rules.

A.7.2 File `auto.thy`

```
(* Time-stamp: <1999-02-21 13:03:57 kokdg> *)

auto = Reduction +

rules

  delay  "[| x >> x : P ; !!y. y : P ==> y : R |] ==> x >> x : R"

end
```

A.7.3 File `auto.ML`

```
(* Time-stamp: <1999-02-22 11:39:51 kokdg> *)

open auto;

  goal Reduction.thy "?a >> ?p : ?P";
val varHelp = uresult();


signature NORM =
  sig
    val addinfs : thm list -> unit
    val addreds : thm list -> unit
    val addaxs : thm list -> unit
```

```
    val setvartac : (int -> tactic) -> unit
    val get_tactic : unit -> tactic
    val reset : unit -> unit
  end;

structure Norm : NORM =
  struct

    fun reduce' infs axs reds var_tac=
      let val rules = (map (fn x => x RS delay) axs) @ infs
      in
        (FIRSTGOAL
 (resolve_tac rules))
        ORELSE
        (FIRSTGOAL
 ((rtac varHelp) THEN' var_tac))
        ORELSE (FIRSTGOAL ((resolve_tac reds) ORELSE' try_all_rules))
      end

    (* tactic try_all_rules above includes rule IsNorm for *)
    (* accepting a proof term as normalized. The rule can not be *)
    (* accessed using other methods. *)

    (* initially no inference rules is included *)

    val inference_rules = ref ([] : thm list)
    val axiom_rules = ref ([] : thm list)
    val reduction_rules = ref ([] : thm list)
    val variable_tactic = ref (fn n:int => no_tac);

    (* Adding inference rules *)

    fun addinfs reds = (inference_rules := reds @ (!inference_rules))
    fun addreds reds = (reduction_rules := reds @ (!reduction_rules))
    fun addaxs reds = (axiom_rules := reds @ (!axiom_rules))
    fun setvartac tac = (variable_tactic := tac)

    (* Resetting all sets of inference rules *)

    fun reset () = (inference_rules := []; axiom_rules := [];
                    reduction_rules := [];
                    variable_tactic := (fn n => no_tac))

    (* Retrieving normalizer *)

    fun get_tactic () = reduce' (!inference_rules) (!axiom_rules)
      (!reduction_rules) (!variable_tactic)

  end; (*structure Norm*)
```

A.8 Normalizer for theory $\mathcal{PR}_{\mathcal{IFOLP}}$

A normalizer automating the proof reduction implemented in object logic `PR` must be created using the meta normalizer. Context inference rules, context axioms, reduction rules and the variable tactic used by the normalizer must be specified.

Since the normalizer is defined in an object logic there must be a theory definition file for object logic `PRauto` defining the normalizer. No new rules are added, therefore the object logic does only include object logics `PR` and `auto`.

Rules that must be added are the context elimination rules as they are stated in object logic `Reduction` and the context introduction rules as they are stated in object logic `PR`. None of these inference rules collides with each other or with any of the reduction rules in the meta normalizer.

The variable tactic must be set as tactic `vartac`.

A tactic `retac` is created that is the normalizer. A state can be solved completely using tactic `(REPEAT retac) THEN (ALLGOALS assume_tac)`.

A.8.1 Theory File `PRauto.thy`

```
(* Time-stamp: <1999-02-21 13:39:55 kokdg> *)

PRauto = PR + auto
```

A.8.2 Theory File `PRauto.ML`

```
(* Time-stamp: <1999-02-21 16:09:06 kokdg> *)

open PRauto;

Norm.addinfs [impCE, conjCE1, conjCE2, allCE, exCE, disjCE,
              substCE];

Norm.addinfs [allCI, conjCI, disjCI1, disjCI2, exCI, impCI];

Norm.addaxs [substCI];

Norm.setvartac vartac;

val retac = Norm.get_tactic();
```

A.9 Normalizer for theory $\mathcal{PE}_{\mathcal{IFOLP}}$

A normalizer automating the evaluation of proofs as programs implemented in object logic `PE` must be created using the meta normalizer. Rules and tactics are added like for the $\mathcal{PR}_{\mathcal{IFOLP}}$ normalizer. Used rules are context elimination rules and reduction rules, context tactic is `try_allRules`.

A tactic `retac` is created that is the normalizer. A state can be solved completely using tactic `(REPEAT retac) THEN (ALLGOALS assume_tac)`.

A.9.1 Theory File PEauto.thy

```
(* Time-stamp: <1999-02-21 13:02:26 kokdg> *)

PEauto = PE + auto
```

A.9.2 Theory File PEauto.ML

```
(* Time-stamp: <1999-02-21 17:25:19 kokdg> *)

open PEauto;

Norm.addinfs [impCE, conjCE1, conjCE2, allCE, exCE, disjCE,
              substCE];

Norm.setvartac try_allRules;


val retac = Norm.get_tactic();
```

A.10 Natural Numbers

Reasoning about natural numbers represented by formal theory $\mathcal{NAT}$ is implemented by using a part of the original work by Larry Paulson, represented in file ex/Nat.thy of the Isabelle distribution package. The ideas for the lemmas are found in ex/Nat.ML but proven in another way.

The implementation of the object logic Nat for formal theory $\mathcal{NAT}$ is in files Nat.thy and Nat.ML. Since reasoning about natural numbers is based on formal theory $\mathcal{IFOL}$ object logic FirstOrder is included as a part of object logic Nat.

A.10.1 Theory File Nat.thy

```
(* Time-stamp: <1999-01-14 13:33:52 kokdg> *)

(* Theory for natural numbers based on the Peano axioms and the *)
(* definition of recursion. *)

(* THIS FILE IS MOSTLY A COPY OF THE FILE FOL/ex/Nat.thy IN THE *)
(* ISABELLE DISTRIBUTION PACKAGE! *)

(* Original file is copyrighted by Larry Paulson, 1992, modified and *)
(* extracted by Bo Kjellerup, 1999 *)

Nat = FirstOrder +

types   nat
```

```
arities nat :: term

consts  "0" :: "nat"    ("0")
        S   :: "nat=>nat"
        rec :: "[nat, 'a, [nat,'a]=>'a] => 'a"
        "+" :: "[nat, nat] => nat"              (infixl 60)

rules   induct    "[| P(0);  !!x. P(x) ==> P(S(x)) |]  ==> ALL n. P(n)"
        S_inject  "S(m)=S(n) ==> m=n"
        S_neq_0   "S(m)=0 ==> R"
        rec_0     "rec(0,a,f) = a"
        rec_S     "rec(S(m), a, f) = f(m, rec(m,a,f))"
        add_def   "m+n == rec(m, n, %x y. S(y))"
end
```

A.10.2 Theory File Nat.ML

```
(* Time-stamp: <1999-01-19 13:19:21 kokdg> *)

(* Theory for natural numbers based on the Peano axioms and the *)
(* definition of recursion. *)

(* The file FOL/ex/Nat.ML in the Isabelle distribution has been used *)
(* for inspiration -- some of the proofs in the mentioned file was *)
(* proven using some constructions that is not available in this *)
(* project. *)

open Nat;

(* An induction theorem where each predicate variable are quantified *)
(* which makes it more safe for proving proofs. *)

val [hb,hi] = goal Nat.thy
  "[| P(0); ALL x. P(x) --> P(S(x)) |] ==> ALL n. P(n)";
    br induct 1;
    br hb 1;
    br mp 1;
    br allE 1;
    br hi 1;
    ba 1;
    ba 1;
val induction = result();

(* The "inverse" of rule S_inject. *)

val [h] = goal Nat.thy "m = n ==> S(m) = S(n)";
  by ((rtac subst_context 1) THEN (rtac h 1));
val S_surj = result();
```

A.11 Natural Numbers with Proofs

Reasoning about natural numbers with proofs represented by formal theory $\mathcal{NATP}$ is implemented using a part of the original work by Larry Paulson, represented in file `ex/Nat.thy` of the Isabelle distribution package. The ideas for the lemmas are found in `ex/Nat.ML` but proven in another way.

The implementation of the object logic `NATP` for formal theory $\mathcal{NATP}$ is in files `NATP.thy` and `NATP.ML`. Since reasoning about proofs including natural numbers is based on formal theory $\mathcal{IFOLP}$ object logic `FirstOrderProof` is included as a part of object logic `NATP`.

Notice the difference between the implemented rules in the object logics `Nat` and `NATP`. In `Nat` rule `induct` is in the theory file and rule `induction` is proven as a theorem, where in `NATP` rule `induction` is in the theory file and rule `induct` is proven as a theorem.

A.11.1 Theory File `NATP.thy`

```
(* Time-stamp: <1999-01-14 13:33:51 kokdg> *)

(* Theory for natural numbers with linearized proofs based on the *)
(* Peano axioms and the definition of recursion. *)

(* THIS FILE IS MOSTLY A COPY OF THE FILE FOL/ex/Nat.thy IN THE *)
(* ISABELLE DISTRIBUTION PACKAGE! *)

(* Original file is copyrighted by Larry Paulson, 1992, modified and *)
(* extracted by Bo Kjellerup, 1999 *)

NATP = FirstOrderProof +

types   nat

arities nat :: term

consts  "0" :: "nat"    ("0")
        S   :: "nat=>nat"
        rec :: "[nat, 'a, [nat,'a]=>'a] => 'a"
        "+" :: "[nat, nat] => nat"              (infixl 60)

  (*Proof terms*)
       nrec         :: "[p,p]=>p"
       ninj,nneq    :: "p=>p"
       rec0, recS   :: "p"

rules

  induction  "[| b: P(0);
                 f: ALL x. P(x) --> P(S(x)) |]
              ==> nrec(b,f) : ALL n. P(n)"

  S_inject   "p : S(m)=S(n) ==> ninj(p) : m=n"
  S_neq_0    "p : S(m)=0 ==> nneq(p) : R"
```

```
  rec_0      "rec0 : rec(0,a,f) = a"
  rec_S      "recS : rec(S(m), a, f) = f(m, rec(m,a,f))"
  add_def    "m+n == rec(m, n, %x y. S(y))"
end
```

A.11.2 Theory File NATP.ML

```
(* Time-stamp: <1999-01-19 14:59:06 kokdg> *)

(* Theory for natural numbers with linearized proofs based on the *)
(* Peano axioms and the definition of recursion. *)

(* This files is mostly equal with the Nat.ML file. *)

open NATP;

(* The induction theorem as specified in the formal theory *)

val [hb,hi] =  goal NATP.thy
  "[| b : P(0); !!x xa. xa : P(x) ==> f(x, xa) : P(S(x)) |] ==> \
  \ nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)";
    br induction 1;
    br hb 1;
    br allI 1;
    br impI 1;
    br hi 1;
    ba 1;
val induct = result();

(* The "inverse" of rule S_inject. *)

val [h] = goal NATP.thy "r : m = n ==> ?p : S(m) = S(n)";
  by ((rtac subst_context 1) THEN (rtac h 1));
val S_surj = result();
```

A.12 Natural Number Reduction

Reduction of proofs including natural numbers and evaluation of programs including natural numbers, described as formal theories $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ and $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$, share some rules. The shared set of rules is implemented in the object logic `NATReduction` that is implemented in files `NATReduction.thy` and `NATReduction.ML`.

The set of shared rules consists of reduction rules for natural numbers, the rules for proofs of natural numbers as object logic `NATP`, and the reduction rules for first order logic with proofs as object logic `Reduction`.

Notice that the tactic `try_all_rules` of object logic `Reduction` does not longer try all rules before applying rule IsNorm since extra reduction rules have been added. Therefore tactic `try_all_rules` must be modified to try to apply the added reduction rules before continuing with the original

tactic.

A.12.1 Theory File NATReduction.thy

```
(* Time-stamp: <1999-01-14 13:43:01 kokdg> *)

(* Theory for reduction of linearized proofs of natural numbers based *)
(* on the Peano axioms and the definition of recursion. *)

NATReduction = NATP + Reduction +

rules

  inductionR0  "b >> b' : A ==> nrec(b,i) ^ 0 !> b' : A"
  inductionRS  "((i ^ n) ‘ (nrec(b,i) ^ n)) >> c : A
                ==> nrec(b,i) ^ S(n) !> c : A"

end
```

A.12.2 Theory File NATReduction.ML

```
(* Time-stamp: <1999-01-14 13:40:43 kokdg> *)

open NATReduction;

val t' =try_all_rules;
val try_all_rules =
  (resolve_tac [inductionR0, inductionRS]) ORELSE' t';
fun bt n = by (try_all_rules n);
```

A.13 Reducing Proofs with Natural Numbers

The object logic implementing formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ includes the reduction object logic NATReduction and the object logic PR implementing formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$. Specific rules for proof reduction for natural numbers are in object logic PRNAT completing the implementation of object logic $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$.

A.13.1 Theory File PRNAT.thy

```
(* Time-stamp: <1999-02-22 14:40:06 kokdg> *)

PRNAT = NATReduction + PR +

rules

  inductionCI  "[| b >> b': P(0);
                  f >> f' : ALL x. P(x) --> P(S(x)) |]
```

```
                ==> nrec(b, f) >> nrec(b', f') : ALL n. P(n)"

  S_injectCI   "p >> p':S(m)=S(n) ==> ninj(p) >> ninj(p') : m=n"
  S_neq_0CI    "p >> p':S(m)=0      ==> nneq(p) >> nneq(p') : R"
  rec_0CI      "rec0 >> rec0 : rec(0,a,f) = a"
  rec_SCI      "recS >> recS : rec(S(m), a, f) = f(m, rec(m,a,f))"

end
```

A.13.2 Theory File PRNAT.ML

```
(* Time-stamp: <1999-01-14 13:52:35 kokdg> *)

open PRNAT;

val [hb,hi] = goal PRNAT.thy
  "[| b >> b' : P(0); \
   \  !!y ya. ya : P(y) ==> f(y, ya) >> g(y, ya) : P(S(y)) |] ==> \
   \ nrec(b, all x. lam xa. f(x, xa)) >> nrec(b', all x. lam xa. g(x, xa)) : \
   \   ALL n. P(n)";
  br inductionCI 1;
  br allCI 2;
  br impCI 2;
  br hb 1;
  br hi 1;
  ba 1;
val inductCI = result();
```

A.14 Evaluating Proofs with Natural Numbers as Programs

The object logic implementing formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ includes the reduction object logic NATReduction and the object logic PE implementing formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$. Specific rules for program evaluation for natural numbers are in object logic PENAT completing the implementation of object logic $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$.

A.14.1 Theory File PENAT.thy

```
(* Time-stamp: <1999-01-14 14:18:01 kokdg> *)

PENAT = NATReduction + PE
```

A.14.2 Theory File PENAT.ML

```
open PENAT;
```

A.15 Lists

Reasoning about lists of elements of some type represented by formal theory $\mathcal{LIST}$ is implemented in object logic `List`. The lemmas of the `List.ML` file are based on the lemmas for natural numbers.

The implementation of the object logic `List` for formal theory $\mathcal{LIST}$ is in files `List.thy` and `List.ML`. Since reasoning about list of elements of some type is based on formal theory $\mathcal{IFOL}$ object logic `FirstOrder` is included as a part of object logic `List`. The implementation also includes a definition for calculating the length of a list which is a natural number, therefore object logic `Nat` must also be included.

A.15.1 Theory File `List.thy`

```
(* Time-stamp: <1999-01-21 15:04:04 kokdg> *)

(* Theory for lists of elements of some type based on the Peano axioms *)
(* and the definition of recursion. *)


List = FirstOrder + Nat +

types   'a list

arities list    :: (term) term

consts  Nil     :: "'a list"
        C       :: "['a, 'a list] => 'a list"
        recl    :: "['a list, 'b, ['a , 'a list, 'b]=>'b] => 'b"
        Append  :: "['a list,'a list] => 'a list"
        Length  :: "'a list => nat"

rules   inductl   "[| P(Nil); !!x xs. P(xs) ==> P(C(x,xs)) |] ==>
                   ALL l. P(l)"

        C_inject1 "C(x,xs) = C(y,ys) ==> xs = ys"
        C_inject2 "C(x,xs) = C(y,ys) ==> x = y"

        C_neq_Nil "C(x,xs) = Nil ==> R"

        recl_Nil  "recl(Nil,a,f) = a"
        recl_C    "recl(C(x,xs),a,f) = f(x,xs,recl(xs,a,f))"

        app_def         "Append(xs,ys) ==
                         recl(xs, ys, %x xs r . C(x,r))"
        len_def         "Length(xs) == recl(xs, 0, %x xs r. S(r))"


end
```

A.15.2 Theory File List.ML

```
(* Time-stamp: <1999-01-22 11:45:19 kokdg> *)

(* Theory for lists of elements of some type based on the Peano axioms *)
(* and the definition of recursion. *)

open List;

(* An induction theorem where each predicate variable are quantified *)
(* which makes it more safe for proving proofs. *)

val [hb,hi] = goal List.thy
  "[| P(Nil); ALL x xs. P(xs) --> P(C(x,xs)) |] ==> ALL l. P(l)";
    br inductl 1;
    br hb 1;
    br mp 1;
    br allE 1;
    br allE 1;
    br hi 1;
    ba 1;
    ba 1;
    ba 1;
val induction = result();

(* The "inverse" of rule S_inject. *)

val [h] = goal List.thy "x = y ==> C(x,l) = C(y,l)";
  by ((rtac subst_context 1) THEN (rtac h 1));
val C_surj1 = result();

val [h] = goal List.thy "xs = ys ==> C(a,xs) = C(a,ys)";
  by ((rtac subst_context 1) THEN (rtac h 1));
val C_surj2 = result();

val [h1,h2] = goal List.thy
  "[| x = (y::'a) ; xs = (ys::'a list) |] ==> C(x,xs) = C(y,ys)";
  br subst 1;
  br h1 1;
  ba 2;
  br C_surj2 1;
  br h2 1;
val C_surj = result();
```

A.16 Lists with Proofs

Reasoning about proofs including lists of elements of some type represented by formal theory $\mathcal{LISTP}$ is implemented in object logic LISTP. The lemmas of the LISTP.ML file is based on the lemmas for natural numbers.

The implementation of the object logic LISTP for formal theory $\mathcal{LISTP}$ is in files LISTP.thy and

LISTP.ML. Since reasoning about proofs including lists of elements of some type is based on formal theory $\mathcal{IFOLP}$ object logic FirstOrderProof is included as a part of object logic LISTP. The implementation also includes a definition for calculating the length of a list which is a natural number, therefore object logic NATP for reasoning about proofs including natural numbers must also be included.

Notice the difference between the implemented rules in the object logics List and LISTP. In the List object logic inference rule inductl is in the theory file and rule induction is proven as a theorem where in object logic LISTP rule induction os in the theory file and rule inductl is proven as a theorem.

A.16.1 Theory File LISTP.thy

```
(* Time-stamp: <1999-01-22 15:09:31 kokdg> *)

(* Theory for proofs including lists of elements of some type based *)
(* on the Peano axioms and the definition of recursion. *)


LISTP = FirstOrderProof + NATP +

types   'a list

arities list      :: (term) term

consts  Nil       :: "'a list"
        C         :: "['a, 'a list] => 'a list"
        recl      :: "['a list, 'b, ['a , 'a list, 'b]=>'b] => 'b"
        Append    :: "['a list,'a list] => 'a list"
        Length    :: "'a list => nat"

(* proof terms *)

        lrec      :: "[p,p] => p"
        linj,
        ainj,
        lneq      :: "p => p"
        recNil,
        recC      :: "p"

rules   induction "[| b : P(Nil);
                      f : ALL x xs. P(xs) --> P(C(x,xs)) |] ==>
                  lrec(b,f) : ALL l. P(l)"

        C_inject1   "p : C(x,xs) = C(y,ys) ==> linj(p) : xs = ys"
        C_inject2   "p : C(x,xs) = C(y,ys) ==> ainj(p) : x = y"

        C_neq_Nil   "p : C(x,xs) = Nil ==> lneq(p) : R"

        recl_Nil    "recNil : recl(Nil,a,f) = a"
        recl_C      "recC : recl(C(x,xs),a,f) = f(x,xs,recl(xs,a,f))"

        app_def     "Append(xs,ys) == recl(xs, ys, %x xs r . C(x,r))"
```

```
        len_def     "Length(xs) == recl(xs, 0, %x xs r. S(r))"

end
```

A.16.2 Theory File LISTP.ML

```
(* Time-stamp: <1999-01-22 13:44:21 kokdg> *)

(* Theory for proofs including lists of elements of some type based *)
(* on the Peano axioms and the definition of recursion. *)

open LISTP;

(* An induction theorem where each predicate variable are quantified *)
(* which makes it more safe for proving proofs. *)

val [hb,hi] = goal LISTP.thy
  "[| b : P(Nil); \
  \   !! x xs u. u : P(xs) ==> \
  \      f(x,xs,u) : P(C(x,xs)) |] ==> \
  \?p : ALL l. P(l)";
    br induction 1;
    br hb 1;
    br allI 1;
    br allI 1;
    br impI 1;
    br hi 1;
    ba 1;
val inductl = result();

(* The "inverses" of rules C_inject1, C_inject2. *)

val [h] = goal LISTP.thy "p : x = y ==> ?q : C(x,l) = C(y,l)";
  by ((rtac subst_context 1) THEN (rtac h 1));
val C_surj1 = result();

val [h] = goal LISTP.thy "p : xs = ys ==> ?q : C(a,xs) = C(a,ys)";
  by ((rtac subst_context 1) THEN (rtac h 1));
val C_surj2 = result();

val [h1,h2] = goal LISTP.thy
  "[| p : x = (y::'a) ; q : xs = (ys::'a list) |] ==> ?r : C(x,xs) = C(y,ys)";
  br subst 1;
  br h1 1;
  ba 2;
  br C_surj2 1;
  br h2 1;
val C_surj = result();
```

A.17 List Reduction

Reduction of proofs including lists of elements of some type and evaluation of programs including lists of elements of some type, described as formal theories $\mathcal{PR}_{\mathcal{IFOLP},\mathcal{LIST}}$ and $\mathcal{PE}_{\mathcal{IFOLP},\mathcal{LIST}}$, share some rules. The shared set of rules is implemented in object logic `LISTReduction` implemented in files `LISTReduction.thy` and `LISTReduction.ML`.

The set of shared rules consists of reduction rules for lists of elements of some type, the rules for proofs including lists of elements of some type as object logic `LISTP` and the reduction rules for first order logic with proofs as object logic `Reduction`. Since natural numbers are used for a definition in the object logic for lists, reduction of natural numbers represented as object logic `NATReduction` must be included.

Notice that the tactic `try_all_rules` defined in object logic `Reduction` and modified in object logic `NATReduction` does not longer try all rules before applying rule IsNorm since extra reduction rules have been added. Therefore tactic `try_all_rules` must be modified to try to apply the added reduction rules before continuing with the original tactic.

A.17.1 Theory File `LISTReduction.thy`

```
(* Time-stamp: <1999-01-22 18:27:22 kokdg> *)

(* Theory for reduction of linearized proofs including lists of *)
(* elements of some type based on the Peano axioms and the definition *)
(* of recursion. *)


LISTReduction = LISTP + NATReduction +

rules

  inductionRNil  "b >> b' : A ==> lrec(b, i) ^ Nil !> b' : A"

  inductionRC    "((i ^ x) ^ xs) ‘ (lrec(b,i) ^ xs) >> c : A
                  ==> lrec(b, i) ^ C(x,xs) !> c : A"

end
```

A.17.2 Theory File `LISTReduction.ML`

```
(* Time-stamp: <1999-01-22 17:10:50 kokdg> *)

open LISTReduction;

val t' = try_all_rules;
val try_all_rules =
  (resolve_tac [inductionRNil, inductionRC]) ORELSE' t';
fun bt n = by (try_all_rules n);
```

A.18 Reducing Proofs with Lists

The object logic implementing formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ include the reduction object logic `LISTReduction` and the object logic `PRNAT` implementing formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ extending formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$ with natural number reduction. Specific rules for proof reduction for lists of elements of some type are in object logic `PRLIST` completing the implementation of object logic $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{LIST}}$.

A.18.1 Theory File `PRLIST.thy`

```
(* Time-stamp: <1999-01-22 17:23:56 kokdg> *)

(* Theory for proofs including lists of elements of some type based *)
(* on the Peano axioms and the definition of recursion. *)


PRLIST = LISTReduction + PRNAT +

rules

  inductionCI   "[| b >> b' : P(Nil);
                    f >> f' : ALL x xs. P(xs) --> P(C(x,xs)) |] ==>
                 lrec(b, f) >> lrec(b', f'): ALL l. P(l)"

  C_inject1CI   "p >> p' : C(x,xs) = C(y,ys) ==> linj(p) >> linj(p') : xs = ys"
  C_inject2CI   "p >> p' : C(x,xs) = C(y,ys) ==> ainj(p) >> ainj(p') : x = y"

  C_neq_NilCI   "p >> p' : C(x,xs) = Nil ==> lneq(p) >> lneq(p') : R"

  recl_NilCI    "recNil >> recNil : recl(Nil,a,f) = a"
  recl_CCI      "recC >> recC : recl(C(x,xs),a,f) = f(x,xs,recl(xs,a,f))"

end
```

A.18.2 Theory File `PRLIST.ML`

```
(* Time-stamp: <1999-01-22 17:48:56 kokdg> *)

open PRLIST;

val [hb,hi] = goal PRLIST.thy
  "[| b >> b' : P(Nil); \
   \  !!x xs u. u : P(xs) ==> \
   \            f(x, xs, u) >> g(x, xs, u) : P(C(x,xs)) |] ==> \
   \ lrec(b, all x xa. lam xb. f(x, xa, xb)) >> \
   \ lrec(b', all x xa. lam xb. g(x, xa, xb)) : \
   \   ALL n. P(n)";
  br inductionCI 1;
  br allCI 2;
  br allCI 2;
```

```
  br impCI 2;
  br hb 1;
  br hi 1;
  ba 1;
val inductCI = result();
```

A.19 Evaluating Proofs with Lists as Programs

The object logic implementing formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ include the reduction object logic `LISTReduction` and the object logic `PENAT` implementing formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ extending formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$ with natural number program evaluation. Specific rules for program evaluation for lists of elements of some type are in object logic `PELIST` completing the implementation of object logic $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{LIST}}$.

A.19.1 Theory File `PELIST.thy`

```
(* Time-stamp: <1999-01-26 10:26:37 kokdg> *)

PELIST = LISTReduction + PENAT
```

A.19.2 Theory File `PELIST.ML`

```
(* Time-stamp: <1999-01-26 10:34:50 kokdg> *)

open PELIST;
```

Appendix B

Examples Using Object Logics

Each formal theory is implemented as an Isabelle theory and some examples are done to show how each of these theories works.

B.1 First Order Logic

An object logic for natural deductive intuitionistic first order logic with equality is implemented in Isabelle in the theory files `IFOL.thy` and `IFOL.ML`. The implementation does correspond well with the formal theory $\mathcal{IFOL}$ implemented in chapter 2.4.4 but not well enough. Some rules in the implementation are not present in $\mathcal{IFOL}$ and other rules in the implementation are not the same as the implemented rules.

B.1.1 Implementing Theory

Formal theory $\mathcal{IFOL}$ defines well-formed formula predicates to have arguments of first order; the arguments may only be variables, constants and functions but not predicates. Therefore a sort for variables shall be created and used for arguments to predicates; the sort may be `term` as defined in section 6.4.1. All predicates may be of type `o` with arity `logic`.

The object logic representing formal theory $\mathcal{IFOL}$ can be found in appendix A.2.

B.1.2 Examples

Some examples may show how the object logic can be used. Since the implementation combines propositional logic with quantifiers and equality, an example from each of these parts may show how the object logic can be used.

The proposition $(P \vee Q) \wedge R \rightarrow P \wedge R \vee Q \wedge R$ is proven by the proof tree

$$\dfrac{\dfrac{\dfrac{\overset{\alpha}{(P \vee Q) \wedge R}}{P \vee Q}\,\wedge E_1 \qquad \mathcal{D}_1 \qquad \mathcal{D}_2}{P \wedge R \vee Q \wedge R}\,\vee E^{\beta \lhd P,\ \gamma \lhd Q}}{(P \vee Q) \wedge R \to P \wedge R \vee Q \wedge R}\,\to I^{\alpha \lhd (P \vee Q) \wedge R}$$

where

$$\mathcal{D}_1 \;=\; \dfrac{\dfrac{\overset{\beta}{P} \qquad \dfrac{\overset{\alpha}{(P \vee Q) \wedge R}}{R}\,\wedge E_2}{P \wedge R}\,\wedge I}{P \wedge R \vee Q \wedge R}\,\vee I_1$$

$$\mathcal{D}_2 \;=\; \dfrac{\dfrac{\overset{\gamma}{Q} \qquad \dfrac{\overset{\alpha}{(P \vee Q) \wedge R}}{R}\,\wedge E_2}{Q \wedge R}\,\wedge I}{P \wedge R \vee Q \wedge R}\,\vee I_1$$

An interactive proving session in Isabelle proving the above proof can be done like the following session

```
- goal FirstOrder.thy
  "(P | Q) & R --> (P & R) | (Q & R)";
Level 0
(P | Q) & R --> P & R | Q & R
 1. (P | Q) & R --> P & R | Q & R
val it = [] : thm list
- br impI 1;
Level 1
(P | Q) & R --> P & R | Q & R
 1. (P | Q) & R ==> P & R | Q & R
val it = () : unit
- br disjE 1;
Level 2
(P | Q) & R --> P & R | Q & R
 1. (P | Q) & R ==> ?P1 | ?Q1
 2. [| (P | Q) & R; ?P1 |] ==> P & R | Q & R
 3. [| (P | Q) & R; ?Q1 |] ==> P & R | Q & R
val it = () : unit
- br conjunct1 1;
Level 3
(P | Q) & R --> P & R | Q & R
 1. (P | Q) & R ==> (?P1 | ?Q1) & ?Q2
 2. [| (P | Q) & R; ?P1 |] ==> P & R | Q & R
 3. [| (P | Q) & R; ?Q1 |] ==> P & R | Q & R
val it = () : unit
- ba 1;
Level 4
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; P |] ==> P & R | Q & R
 2. [| (P | Q) & R; Q |] ==> P & R | Q & R
val it = () : unit
- br disjI1 1;
Level 5
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; P |] ==> P & R
 2. [| (P | Q) & R; Q |] ==> P & R | Q & R
val it = () : unit
- br conjI 1;
Level 6
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; P |] ==> P
 2. [| (P | Q) & R; P |] ==> R
 3. [| (P | Q) & R; Q |] ==> P & R | Q & R
val it = () : unit
- ba 1;
Level 7
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; P |] ==> R
 2. [| (P | Q) & R; Q |] ==> P & R | Q & R
val it = () : unit
- br conjunct2 1;
Level 8
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; P |] ==> ?P5 & R
 2. [| (P | Q) & R; Q |] ==> P & R | Q & R
val it = () : unit
- ba 1;
Level 9
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; Q |] ==> P & R | Q & R
val it = () : unit
- br disjI2 1;
Level 10
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; Q |] ==> Q & R
val it = () : unit
- br conjI 1;
Level 11
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; Q |] ==> Q
 2. [| (P | Q) & R; Q |] ==> R
val it = () : unit
- ba 1;
Level 12
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; Q |] ==> R
```

```
val it = () : unit
- br conjunct2 1;
Level 13
(P | Q) & R --> P & R | Q & R
 1. [| (P | Q) & R; Q |] ==> ?P8 & R
val it = () : unit
- ba 1;
Level 14
(P | Q) & R --> P & R | Q & R
No subgoals!
val it = () : unit
- val propEx = result();
val propEx =
"(?P | ?Q) & ?R --> ?P & ?R | ?Q & ?R" : thm
```

The proposition $P \to (P \to Q) \to Q$ is proven by the proof tree

$$\dfrac{\dfrac{\dfrac{\overset{\beta}{P \to Q} \quad \overset{\alpha}{P}}{Q}\,{\to}\mathsf{E}}{(P \to Q) \to Q}\,{\to}\mathsf{I}^{\beta \,\triangleleft\, P \to Q}}{P \to (P \to Q) \to Q}\,{\to}\mathsf{I}^{\alpha \,\triangleleft\, P}$$

An interactive proving session in Isabelle proving the above proof can be done like the following session

```
- goal FirstOrder.thy "P --> (P --> Q) --> Q";
Level 0
P --> (P --> Q) --> Q
 1. P --> (P --> Q) --> Q
val it = [] : thm list
- br impI 1;
Level 1
P --> (P --> Q) --> Q
 1. P ==> (P --> Q) --> Q
val it = () : unit
- br impI 1;
Level 2
P --> (P --> Q) --> Q
 1. [| P; P --> Q |] ==> Q
val it = () : unit
- br mp 1;
Level 3
P --> (P --> Q) --> Q
 1. [| P; P --> Q |] ==> ?P2 --> Q
 2. [| P; P --> Q |] ==> ?P2
val it = () : unit
- ba 1;
Level 4
P --> (P --> Q) --> Q
 1. [| P; P --> Q |] ==> P
val it = () : unit
- ba 1;
Level 5
P --> (P --> Q) --> Q
No subgoals!
val it = () : unit
- val impEx = result{};
val impEx = "?P --> (?P --> ?Q) --> ?Q" : thm
```

The proposition $(\exists x \forall y\colon P(x,y)) \to (\forall y \exists x\colon P(x,y))$ is proven by the proof tree

$$\dfrac{\dfrac{\dfrac{\Lambda y.\mathcal{D}}{\forall y \exists x\colon P(x,y)}\,\forall\mathsf{I}}{(\exists x \forall y\colon P(x,y)) \to (\forall y \exists x\colon P(x,y))}\,{\to}\mathsf{I}^{\beta \,\triangleleft\, \exists x \forall y\colon P(x,y)}$$

where

$$\mathcal{D} = \dfrac{\overset{\beta}{\exists x.\forall y.P(x,y)} \quad \dfrac{\dfrac{\overset{\alpha}{\forall y.P(xa,y)} \quad \overset{\gamma}{P(xa,y)}}{P(xa,y)}\,\forall\mathsf{E}^{\gamma \,\triangleleft\, P(xa,y)}}{\exists x.P(x,y)}\,\exists\mathsf{I}}{\exists x.P(x,y)}\,\exists\mathsf{E}^{\alpha \,\triangleleft\, \forall y.P(xa,y)}$$

An interactive proving session in Isabelle proving the above proof can be done like the session below. Note the use of destruction solving.

```
- goal FirstOrder.thy
= "(EX (x::'a). (ALL (y::'b). P(x,y))) \
= \ --> (ALL (y::'b). (EX (x::'a). P(x,y)))";
Level 0
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
```

```
 1. (EX x. ALL y. P(x, y)) -->
    (ALL y. EX x. P(x, y))
val it = [] : thm list
- br impI 1;
Level 1
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
 1. EX x. ALL y. P(x, y) ==>
    ALL y. EX x. P(x, y)
val it = () : unit
- br allI 1;
Level 2
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
 1. !!y. EX x. ALL y. P(x, y) ==> EX x. P(x, y)
val it = () : unit
- bd exE 1;
Level 3
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
 1. !!y x. ALL y. P(x, y) ==> ?R2(y)
 2. !!y. ?R2(y) ==> EX x. P(x, y)
val it = () : unit
- ba 2;
Level 4
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
 1. !!y x. ALL y. P(x, y) ==> EX x. P(x, y)
val it = () : unit
- bd allE 1;
Level 5
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
 1. !!y x. P(x, ?y3(y, x)) ==> ?R3(y, x)
 2. !!y x. ?R3(y, x) ==> EX x. P(x, y)
val it = () : unit
- ba 2;
Level 6
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
 1. !!y x. P(x, ?y3(y, x)) ==> EX x. P(x, y)
val it = () : unit
- br exI 1;
Level 7
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
 1. !!y x. P(x, ?y3(y, x)) ==> P(?x4(y, x), y)
val it = () : unit
- ba 1;
Level 8
(EX x. ALL y. P(x, y)) -->
(ALL y. EX x. P(x, y))
No subgoals!
val it = () : unit
- val QuantEx = result();
val QuantEx = "(EX x. ALL y. ?P(x, y)) --> \
\ (ALL y. EX x. ?P(x, y))" : thm
```

Lemmas for proposition symmetry and transitivity can be proven using the inference rule for substitution. This gives the below proof trees and proof sessions.

An inference rule $\dfrac{a = b}{b = a}$ is proven by the proof tree

$$\frac{a = b \qquad \Lambda x\, \overline{x = x}^{\ \text{refl}} \qquad \begin{matrix}\alpha(a,b)\\ b = a\end{matrix}}{b = a}\ \text{subst}^{\alpha \,\triangleleft\, (x,y)\mapsto y=x}$$

which is proven by the proving session

```
- val [prems] = goal FirstOrder.thy
= "a=b ==> b =a";
std_in:0.0-40.14 Warning: binding not exhaustive
          prems :: nil = ...
Level 0
b = a
 1. b = a
val prems = "a = b  [a = b]" : thm
- br subst 1;
Level 1
b = a
 1. ?x = ?y
 2. !!x. ?P(x, x)
 3. ?P(?x, ?y) ==> b = a
val it = () : unit
- br prems 1;
Level 2
b = a
 1. !!x. ?P(x, x)
 2. ?P(a, b) ==> b = a
val it = () : unit
- br refl 1;
Level 3
b = a
 1. ?a3(a, b) = ?a2(a, b) ==> b = a

Flex-flex pairs:
  (%x. ?a3(x, x)) =?= (%x. ?a2(x, x))
val it = () : unit
- ba 1;
Level 4
b = a
No subgoals!
val it = () : unit
- val sym = result();
val sym = "?a = ?b ==> ?b = ?a" : thm
```

An inference rule $\dfrac{a=b \quad b=c}{a=c}$ is proven by the proof tree

$$\dfrac{\dfrac{b=c \qquad \Lambda x\ \dfrac{\dfrac{\beta}{a=x}}{a=x \to a=x} \to \mathrm{I}^{\beta \,\triangleleft\, a=x} \qquad \alpha(b,c)}{a=b \to a=c}\ \mathsf{subst}^{\alpha \,\triangleleft\, (x,y) \mapsto a=x \to a=y} \qquad a=b}{a=c} \to \mathsf{E}$$

which is proven by the proving session

```
- val [p1,p2] = goal FirstOrder.thy
=  "[| a=b; b=c |] ==> a=c";
std_in:47.1-48.25 Warning:
  binding not exhaustive
          p1 :: p2 :: nil = ...
Level 0
a = c
 1. a = c
val p1 = "a = b  [a = b]" : thm
val p2 = "b = c  [b = c]" : thm
- br mp 1;
Level 1
a = c
 1. ?P --> a = c
 2. ?P
val it = () : unit
- br p1 2;
Level 2
a = c
 1. a = b --> a = c
val it = () : unit
- br subst 1;
Level 3
a = c
 1. ?x1 = ?y1
 2. !!x. ?P1(x, x)
 3. ?P1(?x1, ?y1) ==> a = b --> a = c
val it = () : unit
- br p2 1;
Level 4
a = c
 1. !!x. ?P1(x, x)
 2. ?P1(b, c) ==> a = b --> a = c
val it = () : unit
- ba 2;
Level 5
a = c
 1. !!x. a = x --> a = x
val it = () : unit
- br impI 1;
Level 6
a = c
 1. !!x. a = x ==> a = x
val it = () : unit
- ba 1;
Level 7
a = c
No subgoals!
val it = () : unit
- val trans = result();
val trans =
"[| ?a = ?b; ?b = ?c |] ==> ?a = ?c" : thm
```

It may be a good idea to take care which lemmas to create. As inference rules can be created to be ill suited for backward reasoning as stated in section 2.2.1, which also is taken care of in the creation of formal theory $\mathcal{IFOL}$ in section 2.4.4, lemmas created using rules well suited for backward reasoning can be ill suited for backward reasoning which the following example shows. An inference rule $\dfrac{a=b}{t(a)=t(b)}$ for substituting in a context can be proven by the proof tree

$$\dfrac{a=b \qquad \Lambda x\ \overline{t(x)=t(x)}\ \mathsf{refl} \qquad \begin{matrix}\alpha(a,b)\\ t(a)=t(b)\end{matrix}}{t(a)=t(b)}\ \mathsf{subst}^{\alpha \,\triangleleft\, (x,y) \mapsto t(x)=t(y)}$$

which is proven by the proving session

```
- val [p] = goal FirstOrder.thy
= "[| a=b |] ==> t(a)=t(b)";
std_in:58.1-59.25 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
t(a) = t(b)
 1. t(a) = t(b)
val p = "a = b  [a = b]" : thm
- br subst 1;
Level 1
t(a) = t(b)
 1. ?x = ?y
 2. !!x. ?P(x, x)
 3. ?P(?x, ?y) ==> t(a) = t(b)
val it = () : unit
- br p 1;
Level 2
t(a) = t(b)
 1. !!x. ?P(x, x)
```

```
 2. ?P(a, b) ==> t(a) = t(b)
val it = () : unit
- br refl 1;
Level 3
t(a) = t(b)
 1. ?a3(a, b) = ?a2(a, b) ==> t(a) = t(b)

Flex-flex pairs:
  (%x. ?a3(x, x)) =?= (%x. ?a2(x, x))
```

```
val it = () : unit
- ba 1;
Level 4
t(a) = t(b)
No subgoals!
val it = () : unit
- val subst_context = result();
val subst_context =
"?a = ?b ==> ?t(?a) = ?t(?b)" : thm
```

The rule is ill suited for backward reasoning since for many propositions it is impossible to decide what a function t is. (In fact is it only possible to decide what t may be if the rule is applied on a proposition $x = y$ where x and y either are constants or have no terms in common; then t is the identity function. In all other applications, i.e. the applications where the rule may be usable, t can be either an intended function or the identity function.)

The above example completes a presentation of how all inference rules of formal theory $\mathcal{IFOL}$ and the corresponding object logic `FirstOrder` can be used in proving theorems.

B.2 First Order Logic with Proofs

An object logic `IFOLP` for natural deductive intuitionistic first order logic with equality and proofs is implemented in Isabelle in the theory files `IFOLP.thy` and `IFOLP.ML`. The implementation does correspond well with the formal theory $\mathcal{IFOLP}$ implemented in section 3.3.2 but not well enough. Some rules in the implementation are not present in $\mathcal{IFOLP}$ and other rules in the implementation are not the same as the implemented rules. Among rules in the implementation that is not present in the formal theory are some β conversion rules and other rules for reasoning about proofs. In this project reasoning about proofs is seperated from the proof generation, therefore `IFOLP` is not used as a $\mathcal{IFOLP}$ implementation, but can and will be used heavily as inspiration for a $\mathcal{IFOLP}$ implementation.

B.2.1 Implementing Theory

Formal theory $\mathcal{IFOLP}$ defines well-formed formulas to consist of a proof part and a proposition part. Arguments of proposition parts may only be variables, constants and functions but not proposition parts. Therefore a sort for variables shall be created and used for arguments to predicates; the sort may be `term` as defined in section 6.4.1. Since propositions and proofs may not be mixed proofs may have their own type `p` of arity `logic` and propositions may have their own type `o` of arity `logic`. Since a well-formed formula consists of a proof and a proposition the coercion from well-formed formulas to objects that can be reasoned about combines a proof with a proposition.

The object logic representing formal theory $\mathcal{IFOLP}$ can be found in appendix A.3.

B.2.2 Examples

Some examples may show how the object logic can be used. Since the implementation combines propositional logic with quantifiers and equality an example from each of these parts may show how the object logic can be used. Note that the example proof sessions are the same as the example proof sessions for the object logic for $\mathcal{IFOL}$, proofs are just added as an appendix to the propositions.

In the example proof trees proof parts are written using `typewriter` letters and proposition parts are written using the same sort of letters as in section B.1.2. Arguments of propositions can though be parts of proofs, therefore arguments that both are arguments of proofs and propositions are typeset like proof parts. In section 4.3 these arguments are greek letters with hats but these are not typable as parts of a programming language whereby they look awkward in the proofs.

The proposition $(P \vee Q) \wedge R \rightarrow P \wedge R \vee Q \wedge R$ with proof

```
lam x. when(fst(x), %xa. inl(<xa,snd(x)>), %xb. inr(<xb,snd(x)>))
```

is proven by the proof tree

$$\dfrac{\dfrac{\dfrac{\overset{\alpha}{\mathtt{x} : (P \vee Q) \wedge R}}{\mathtt{fst(x)} : P \vee Q}\,\wedge E_1 \qquad \mathcal{D}_1 \qquad \mathcal{D}_2}{\begin{array}{l}\mathtt{when(fst(x),}\\ \quad \mathtt{\%xa.\ inl(<xa,snd(x)>),}\\ \quad \mathtt{\%xb.\ inr(<xb,snd(x)>))}\\ : P \wedge R \vee Q \wedge R\end{array}}\,\vee E^{\beta \,\triangleleft\, \mathtt{xa}:P,\ \gamma \,\triangleleft\, \mathtt{xb}:Q}}{\begin{array}{l}\mathtt{lam\ x.\ when(fst(x),}\\ \quad \mathtt{\%xa.\ inl(<xa,snd(x)>),}\\ \quad \mathtt{\%xb.\ inr(<xb,snd(x)>))}\\ : (P \vee Q) \wedge R \rightarrow P \wedge R \vee Q \wedge R\end{array}}\,\rightarrow I^{\alpha \,\triangleleft\, \mathtt{x}:(P \vee Q) \wedge R}$$

where

$$\mathcal{D}_1 \;=\; \Lambda \mathtt{xa}\dfrac{\dfrac{\overset{\beta}{\mathtt{xa} : P} \qquad \dfrac{\overset{\alpha}{\mathtt{x} : (P \vee Q) \wedge R}}{\mathtt{snd(x)} : R}\,\wedge E_2}{\mathtt{<xa,snd(x)>} : P \wedge R}\,\wedge I}{\mathtt{inl(<xa,snd(x)>)} : P \wedge R \vee Q \wedge R}\,\vee I_1$$

$$\mathcal{D}_2 \;=\; \Lambda \mathtt{xb}\dfrac{\dfrac{\overset{\gamma}{\mathtt{xb} : Q} \qquad \dfrac{\overset{\alpha}{\mathtt{x} : (P \vee Q) \wedge R}}{\mathtt{snd(x)} : R}\,\wedge E_2}{\mathtt{<xb,snd(x)>} : Q \wedge R}\,\wedge I}{\mathtt{inr(<xb,snd(x)>)} : P \wedge R \vee Q \wedge R}\,\vee I_1$$

An interactive proving session in Isabelle proving the above proof can be done like the session below. (The variable `xb` in $\mathcal{D}_2$ is called `xa` in the session, since there is no collision between `xa` : P and `xa` : Q in the session, which is also the case in the proof tree but it is not obvious).

```
- goal FirstOrderProof.thy
= "?p : (P | Q) & R --> (P & R) | (Q & R)";
Level 0
?p : (P | Q) & R --> P & R | Q & R
 1. ?p : (P | Q) & R --> P & R | Q & R
val it = [] : thm list
- br impI 1;
Level 1
lam x. ?f1(x) : (P | Q) & R --> P & R | Q & R
 1. !!x. x : (P | Q) & R ==>
         ?f1(x) : P & R | Q & R
val it = () : unit
- br disjE 1;
Level 2
lam x. when(?a2(x), ?f2(x), ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x. x : (P | Q) & R ==>
         ?a2(x) : ?P2(x) | ?Q2(x)
 2. !!x xa.
       [| x : (P | Q) & R; xa : ?P2(x) |] ==>
       ?f2(x, xa) : P & R | Q & R
 3. !!x xa.
       [| x : (P | Q) & R; xa : ?Q2(x) |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- br conjunct1 1;
Level 3
lam x. when(fst(?p3(x)), ?f2(x), ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x. x : (P | Q) & R ==>
         ?p3(x) : (?P2(x) | ?Q2(x)) & ?Q3(x)
 2. !!x xa.
```

```
       [| x : (P | Q) & R; xa : ?P2(x) |] ==>
       ?f2(x, xa) : P & R | Q & R
 3. !!x xa.
       [| x : (P | Q) & R; xa : ?Q2(x) |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- ba 1;
Level 4
lam x. when(fst(x), ?f2(x), ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : P |] ==>
       ?f2(x, xa) : P & R | Q & R
 2. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- br disjI1 1;
Level 5
lam x.
   when(fst(x), %xa. inl(?a4(x, xa)), ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : P |] ==>
       ?a4(x, xa) : P & R
 2. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- br conjI 1;
Level 6
lam x.
   when
    (fst(x), %xa. inl(<?a5(x, xa),?b5(x, xa)>),
     ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : P |] ==>
       ?a5(x, xa) : P
 2. !!x xa.
       [| x : (P | Q) & R; xa : P |] ==>
       ?b5(x, xa) : R
 3. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- ba 1;
Level 7
lam x.
   when
    (fst(x), %xa. inl(<xa,?b5(x, xa)>), ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : P |] ==>
       ?b5(x, xa) : R
 2. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- br conjunct2 1;
Level 8
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(?p6(x, xa))>),
     ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : P |] ==>
       ?p6(x, xa) : ?P6(x, xa) & R
 2. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- ba 1;
Level 9
lam x.
   when(fst(x), %xa. inl(<xa,snd(x)>), ?g2(x))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?g2(x, xa) : P & R | Q & R
val it = () : unit
- br disjI2 1;
Level 10
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(?b7(x, xa)))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?b7(x, xa) : Q & R
val it = () : unit
- br conjI 1;
Level 11
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(<?a8(x, xa),?b8(x, xa)>))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?a8(x, xa) : Q
 2. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?b8(x, xa) : R
val it = () : unit
- ba 1;
Level 12
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(<xa,?b8(x, xa)>))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?b8(x, xa) : R
val it = () : unit
- br conjunct2 1;
Level 13
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(<xa,snd(?p9(x, xa))>))
: (P | Q) & R --> P & R | Q & R
 1. !!x xa.
       [| x : (P | Q) & R; xa : Q |] ==>
       ?p9(x, xa) : ?P9(x, xa) & R
val it = () : unit
- ba 1;
Level 14
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
```

```
    %xa. inr(<xa,snd(x)>))
: (P | Q) & R --> P & R | Q & R
No subgoals!
val it = () : unit
- val propEx = result();
```

```
val propEx =
  "lam x. when(fst(x), %xa. inl(<xa,snd(x)>),
   %xa. inr(<xa,snd(x)>))
   : (?P | ?Q) & ?R --> ?P & ?R | ?Q & ?R" : thm
```

The proposition $P \to (P \to Q) \to Q$ with proof `lam x xa. xa ‘ x` is proven by the proof tree

$$\dfrac{\dfrac{\dfrac{\overset{\beta}{\texttt{xa} : P \to Q} \qquad \overset{\alpha}{\texttt{x} : P}}{\texttt{xa ‘ x} : Q}\,{\to}\mathsf{E}}{\texttt{lam xa. xa ‘ x} : (P \to Q) \to Q}\,{\to}\mathsf{I}^{\beta \,\triangleleft\, \texttt{xa}:P\to Q}}{\texttt{lam x xa. xa ‘ x} : P \to (P \to Q) \to Q}\,{\to}\mathsf{I}^{\alpha \,\triangleleft\, \texttt{x}:P}$$

An interactive proving session in Isabelle proving the above proof can be done like the following session

```
- goal FirstOrderProof.thy
= "?p : P --> (P --> Q) --> Q";
Level 0
?p : P --> (P --> Q) --> Q
 1. ?p : P --> (P --> Q) --> Q
val it = [] : thm list
- br impI 1;
Level 1
lam x. ?f1(x) : P --> (P --> Q) --> Q
 1. !!x. x : P ==> ?f1(x) : (P --> Q) --> Q
val it = () : unit
- br impI 1;
Level 2
lam x xa. ?f2(x, xa) : P --> (P --> Q) --> Q
 1. !!x xa.
       [| x : P; xa : P --> Q |] ==>
       ?f2(x, xa) : Q
val it = () : unit
- br mp 1;
Level 3
lam x xa. ?f3(x, xa) ‘ ?a3(x, xa)
: P --> (P --> Q) --> Q
 1. !!x xa.
       [| x : P; xa : P --> Q |] ==>
       ?f3(x, xa) : ?P3(x, xa) --> Q
 2. !!x xa.
       [| x : P; xa : P --> Q |] ==>
       ?a3(x, xa) : ?P3(x, xa)
val it = () : unit
- ba 1;
Level 4
lam x xa. xa ‘ ?a3(x, xa) :
  P --> (P --> Q) --> Q
 1. !!x xa.
       [| x : P; xa : P --> Q |] ==>
       ?a3(x, xa) : P
val it = () : unit
- ba 1;
Level 5
lam x xa. xa ‘ x : P --> (P --> Q) --> Q
No subgoals!
val it = () : unit
- val impEx = result{};
val impEx =
  "lam x xa. xa ‘ x :
   ?P --> (?P --> ?Q) --> ?Q" : thm
```

The proposition $(\exists x \forall y\colon P(x,y)) \to (\forall y \exists x\colon P(x,y))$ with proof

```
lam x. all xa. xsplit(x, %xaa u. [xaa,u^xa])
```

is proven by the proof tree

$$\dfrac{\dfrac{\mathcal{D}}{\begin{array}{l}\texttt{all xa. xsplit(x,}\\ \quad\texttt{\%xaa u. [xaa,u\^{}xa])}\\ : \forall y \exists x\colon P(x,y)\end{array}}\,\forall\mathsf{I}}{\begin{array}{l}\texttt{lam x. all xa. xsplit(x,}\\ \quad\texttt{\%xaa u. [xaa,u\^{}xa])}\\ : (\exists x \forall y\colon P(x,y)) \to (\forall y \exists x\colon P(x,y))\end{array}}\,{\to}\mathsf{I}^{\beta \,\triangleleft\, \texttt{x}:\exists x \forall y\colon P(x,y)}$$

where

$\mathcal{D} =$

$$\dfrac{\mathtt{x} : \exists x.\forall y.P(x,y) \qquad \Lambda \mathtt{xaa} \quad \dfrac{\dfrac{\mathtt{u} : \forall y.P(\mathtt{xaa},y) \qquad \mathtt{ya} : P(\mathtt{xaa},\mathtt{y})}{\mathtt{u\ \hat{}\ y} : P(\mathtt{xaa},\mathtt{y})}\ \forall \mathsf{E}^{\gamma \,\triangleleft\, \mathtt{ya}:P(\mathtt{xaa},\mathtt{y})}}{[\mathtt{xaa},\mathtt{u\ \hat{}\ y}] : \exists x.P(x,\mathtt{y})}\ \exists \mathsf{I}}{\mathtt{xsplit(x,\ \%xaa\ u.\ [xaa,u\ \hat{}\ y])} : \exists x.P(x,\mathtt{y})}\ \exists \mathsf{E}^{\alpha \,\triangleleft\, \mathtt{u}:\forall y.P(\mathtt{xaa},y)}$$

An interactive proving session in Isabelle proving the above proof can be done like the session below. Notice the use of destruction solving.

```
- goal FirstOrderProof.thy
= " ?p : (EX (x::'a). (ALL (y::'b). P(x,y))) \
= \ --> (ALL (y::'b). (EX (x::'a). P(x,y)))";
Level 0
?p
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. ?p
    : (EX x. ALL y. P(x, y)) -->
      (ALL y. EX x. P(x, y))
val it = [] : thm list
- br impI 1;
Level 1
lam x. ?f1(x)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!x. x : EX x. ALL y. P(x, y) ==>
         ?f1(x) : ALL y. EX x. P(x, y)
val it = () : unit
- br allI 1;
Level 2
lam x. all xa. ?f2(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!x y.
       x : EX x. ALL y. P(x, y) ==>
       ?f2(x, y) : EX x. P(x, y)
val it = () : unit
- bd exE 1;
Level 3
lam x. all xa. ?f2(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!x y xa u.
       u : ALL y. P(xa, y) ==>
       ?f3(x, y, xa, u) : ?R3(x, y)
 2. !!x y.
       xsplit(x, ?f3(x, y)) : ?R3(x, y) ==>
       ?f2(x, y) : EX x. P(x, y)
val it = () : unit
- ba 2;
Level 4
lam x. all xa. xsplit(x, ?f3(x, xa))
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!x y xa u.
       u : ALL y. P(xa, y) ==>
       ?f3(x, y, xa, u) : EX x. P(x, y)
val it = () : unit
- bd allE 1;
Level 5
lam x. all xa. xsplit(x, ?f3(x, xa))
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!x y xa u ya.
       ya : P(xa, ?z4(x, y, xa, u)) ==>
       ya : ?R4(x, y, xa, u)
 2. !!x y xa u.
       u ^ ?z4(x, y, xa, u) :
       ?R4(x, y, xa, u) ==>
       ?f3(x, y, xa, u) : EX x. P(x, y)
val it = () : unit
- ba 1;
Level 6
lam x. all xa. xsplit(x, ?f3(x, xa))
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!x y xa u.
       u ^ ?z5(x, y, xa, u)
       : P(xa, ?z5(x, y, xa, u)) ==>
       ?f3(x, y, xa, u) : EX x. P(x, y)
val it = () : unit
- br exI 1;
Level 7
lam x.
   all xa.
      xsplit
       (x,
        %xaa u.
           [?x6(x, xa, xaa, u),
            ?p6(x, xa, xaa, u)])
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!x y xa u.
       u ^ ?z5(x, y, xa, u)
       : P(xa, ?z5(x, y, xa, u)) ==>
       ?p6(x, y, xa, u) : P(?x6(x, y, xa, u), y)
val it = () : unit
- ba 1;
Level 8
lam x. all xa. xsplit(x, %xaa u. [xaa,u ^ xa])
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
No subgoals!
val it = () : unit
- val QuantEx = result();
val QuantEx =
  "lam x.
      all xa. xsplit(x, %xaa u. [xaa,u ^ xa])
   : (EX x. ALL y. ?P(x, y)) -->
     (ALL y. EX x. ?P(x, y))" : thm
```

Lemmas for proposition symmetry and transitivity can be proven using the inference rule for substitution. This gives the below proof trees and proof sessions.

An inference rule $\dfrac{\mathtt{q} : a = b}{\mathtt{idpeel(q,\ \%x.\ ideq(x))} : b = a}$ is proven by the proof tree

$$\dfrac{\mathtt{q} : a = b \qquad \Lambda x\ \dfrac{}{\mathtt{ideq(x)} : \mathtt{x} = \mathtt{x}}\ \mathsf{refl} \qquad \begin{matrix}\alpha(a,b)\\ \mathtt{z} : b = a\end{matrix}}{\mathtt{idpeel(q,\ \%x.\ ideq(x))} : b = a}\ \mathsf{subst}^{\alpha\ \triangleleft\ \mathtt{z}:(x,y)\mapsto y=x}$$

which is proven by the proving session

```
- val [prems] = goal FirstOrderProof.thy
= "q : a=b ==> ?p : b =a";
std_in:0.0-18.23 Warning: binding not exhaustive
          prems :: nil = ...
Level 0
?p : b = a
 1. ?p : b = a
val prems = "q : a = b  [q : a = b]" : thm
- br subst 1;
Level 1
idpeel(?p1, %x. ?f1(x)) : b = a
 1. ?p1 : ?x1 = ?y1
 2. !!x. ?f1(x) : ?P1(x, x)
 3. !!q. q : ?P1(?x1, ?y1) ==> q : b = a
val it = () : unit
- br prems 1;
Level 2
idpeel(q, %x. ?f1(x)) : b = a
 1. !!x. ?f1(x) : ?P1(x, x)
 2. !!q. q : ?P1(a, b) ==> q : b = a
val it = () : unit
- br refl 1;
Level 3
idpeel(q, %x. ideq(?a3(x, x))) : b = a
 1. !!q. q : ?a4(a, b) = ?a3(a, b) ==> q : b = a

Flex-flex pairs:
  (%x. ?a4(x, x)) =?= (%x. ?a3(x, x))
val it = () : unit
- ba 1;
Level 4
idpeel(q, %x. ideq(x)) : b = a
No subgoals!
val it = () : unit
- val sym = result();
val sym = "?q : ?a = ?b ==>
  idpeel(?q, %x. ideq(x)) : ?b = ?a" : thm
```

An inference rule $\dfrac{\mathtt{p} : a = b \qquad \mathtt{q} : b = c}{\mathtt{idpeel(q,\ \%x.\ lam\ x.\ x)\ `\ p} : a = c}$ is proven by the proof tree

$$\dfrac{\mathcal{D} \qquad \mathtt{p} : a = b}{\mathtt{idpeel(q,\ \%x.\ lam\ x.\ x)\ `\ p} : a = c}\ \to \mathsf{E}$$

where

$$\mathcal{D} = \dfrac{\dfrac{\begin{matrix}\beta\\ \mathtt{xa} : a = x\end{matrix}}{\mathtt{lam\ xa.\ xa}}\ \to \mathrm{I}^{\beta\ \triangleleft\ \mathtt{x}:a=x} \qquad \begin{matrix}\alpha(b,c)\\ \mathtt{ya} : a = b \to a = c\end{matrix}}{\mathtt{q} : b = c \quad \Lambda x\ : a = x \to a = x \qquad\qquad \mathtt{idpeel(q,\ \%x.\ lam\ xa.\ xa)} : a = b \to a = c}\ \mathsf{subst}^{\alpha\ \triangleleft\ \mathtt{ya}:(x,y)\mapsto a=x\to a=y}$$

which is proven by the proving session

```
- val [p1,p2] = goal FirstOrderProof.thy
=  "[| p:a=b;  q:b=c |] ==> ?d:a=c";
std_in:155.1-156.33 Warning:
  binding not exhaustive
          p1 :: p2 :: nil = ...
Level 0
?d : a = c
 1. ?d : a = c
val p1 = "p : a = b  [p : a = b]" : thm
val p2 = "q : b = c  [q : b = c]" : thm
- br mp 1;
Level 1
?f1 ` ?a1 : a = c
 1. ?f1 : ?P1 --> a = c
 2. ?a1 : ?P1
val it = () : unit
- br p1 2;
Level 2
?f1 ` p : a = c
 1. ?f1 : a = b --> a = c
val it = () : unit
- br subst 1;
```

```
Level 3
idpeel(?p2, %x. ?f2(x)) ‘ p : a = c
 1. ?p2 : ?x2 = ?y2
 2. !!x. ?f2(x) : ?P2(x, x)
 3. !!q. q : ?P2(?x2, ?y2) ==>
        q : a = b --> a = c
val it = () : unit
- br p2 1;
Level 4
idpeel(q, %x. ?f2(x)) ‘ p : a = c
 1. !!x. ?f2(x) : ?P2(x, x)
 2. !!q. q : ?P2(b, c) ==> q : a = b --> a = c
val it = () : unit
- ba 2;
Level 5
idpeel(q, %x. ?f2(x)) ‘ p : a = c
 1. !!x. ?f2(x) : a = x --> a = x
val it = () : unit
- br impI 1;
Level 6
idpeel(q, %x. lam xa. ?f3(x, xa)) ‘ p : a = c
 1. !!x xa. xa : a = x ==> ?f3(x, xa) : a = x
val it = () : unit
- ba 1;
Level 7
idpeel(q, %x. lam x. x) ‘ p : a = c
No subgoals!
val it = () : unit
- val trans = result();
val trans =
  "[| ?p : ?a = ?b; ?q : ?b = ?c |] ==>
   idpeel(?q, %x. lam x. x) ‘ ?p :
   ?a = ?c" : thm
```

It may be a good idea to take care which lemmas to create. As inference rules can be created to be ill suited for backward reasoning as stated in section 2.2.1, which also is taken care of in the creation of formal theory $\mathcal{IFOLP}$ in section 3.3.2, lemmas created using rules well suited for backward reasoning can be ill suited for backward reasoning which the following example shows. An inference rule $\dfrac{\mathtt{p} : a = b}{\mathtt{idpeel(p,\ \%x.\ ideq(t(x))) : t}(a) = \mathtt{t}(b)}$ for substituting in a context can be proven by the proof tree

$$\dfrac{\mathtt{p} : a = b \qquad \Lambda \mathtt{x}\ \dfrac{}{\mathtt{ideq(t(x)) : t(x) = t(x)}}\ \mathsf{refl} \qquad \begin{matrix}\alpha(a,b)\\ \mathtt{ya} : \mathtt{t}(a) = \mathtt{t}(b)\end{matrix}}{\mathtt{idpeel(p,\ \%x.\ ideq(t(x))) : t}(a) = \mathtt{t}(b)}\ \mathsf{subst}^{\alpha\ \lhd\ \mathtt{ya}:(x,y)\mapsto \mathtt{t}(x)=\mathtt{t}(y)}$$

which is proven by the proving session

```
- val [p] = goal FirstOrderProof.thy
= "[| p : a=b |]  ==>  ?q : t(a)=t(b)";
std_in:177.1-178.38 Warning: binding not
exhaustive
         p :: nil = ...
Level 0
?q : t(a) = t(b)
 1. ?q : t(a) = t(b)
val p = "p : a = b  [p : a = b]" : thm
- br subst 1;
Level 1
idpeel(?p1, %x. ?f1(x)) : t(a) = t(b)
 1. ?p1 : ?x1 = ?y1
 2. !!x. ?f1(x) : ?P1(x, x)
 3. !!q. q : ?P1(?x1, ?y1) ==> q : t(a) = t(b)
val it = () : unit
- br p 1;
Level 2
idpeel(p, %x. ?f1(x)) : t(a) = t(b)
 1. !!x. ?f1(x) : ?P1(x, x)
 2. !!q. q : ?P1(a, b) ==> q : t(a) = t(b)
val it = () : unit
- br refl 1;
Level 3
idpeel(p, %x. ideq(?a3(x, x))) : t(a) = t(b)
 1. !!q. q : ?a4(a, b) = ?a3(a, b)
         ==> q : t(a) = t(b)

Flex-flex pairs:
  (%x. ?a4(x, x)) =?= (%x. ?a3(x, x))
val it = () : unit
- ba 1;
Level 4
idpeel(p, %x. ideq(t(x))) : t(a) = t(b)
No subgoals!
val it = () : unit
- val subst_context = result();
val subst_context =
  "?p : ?a = ?b
   ==> idpeel(?p, %x. ideq(?t(x)))
       : ?t(?a) = ?t(?b)" : thm
```

The above example completes a presentation of how all inference rules of formal theory $\mathcal{IFOLP}$ and the corresponding object logic `FirstOrderProof` can be used in proving theorems.

B.3 First Order Logic with Proofs and Proof Reduction

An object logic for reducing proofs of natural deductive intuitionistic first order logic propositions is not implemented in Isabelle and is therefore not a part of the Isabelle distribution. Therefore an object logic `PR` corresponding to formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$ in section 4.7.3 shall be created.

B.3.1 Implementing Theory

Formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$ specifies that proof reduction shall be reduction of proofs that are proven using formal theory $\mathcal{IFOLP}$. Therefore object logic `PR` shall be built on top of object logic `FirstOrderProof`.

There shall be two object logics implementing reducing of proofs, both as reduction of proofs theoretically regarded as λ calculus terms with reduction to normal form, and as reduction of proofs regarded as programs that shall be evaluated without doing unneccessary reduction steps. The formal theories for these two parts share some parts of the theories. These parts are implemented in object logic `Reduction` presented in appendix A.4.

Object logic `PR` for proof reduction implements the parts of formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$ that is not shared with formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$. Object logic `PR` is in appendix A.5.

B.3.2 Examples

To demonstrate how the theory for proof reduction can be used examples may be given that demonstrates each inference rule. Furthermore, to ensure that the theory will reduce correctly examples may be given that demonstrates the reduction of the following cases:

97$_{205}$ A redex outermost of a proof; the redex shall be reduced.

98$_{205}$ A redex that is argument of a destructor; the redex shall be reduced.

99$_{205}$ A redex that is argument of a constructor; the redex shall be reduced.

100$_{205}$ A destructor that does not take part in a redex before nor after reduction of the arguments of the destructor; the destructor shall not be part in a reduction.

101$_{205}$ A destructor that does not take part in a redex before reduction of the arguments of the destructor but do take part in a redex after reduction of the arguments of the destructor; the redex shall be reduced when it is created.

102$_{205}$ A constructor that does not take part in a redex before and after reduction of the arguments of the constructor; the constructor shall not be part in a reduction.

103$_{205}$ A proof that is a bound variable; the variable itself is a reduced proof.

104$_{205}$ A proof that is a free variable; the variable itself is a reduced proof.

Context Rules

Each context rule can be demonstrated by reducing the examples of section B.2.2 since none of these proofs contains redices. Each context rule example is created by stating the goal `?p : P`, where `P` is the proposition to be proven, and resolving this goal with rule $\longmapsto$ `E` which for goal `?p : ?P` gives the subgoals

```
- goal PR.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
- br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
```

whereafter the first subgoal is proved like the each of the propositions in section B.2.2. Then the second of the above subgoals is instantiated with the proof created by proving the first subgoal and the reduction can take place. In each of the proving sessions below the first subgoal is proved using the theorems created in section B.2.2. Each of the context rules are demonstrated by the following proof trees and proving sessions.

Below is the reduction of the proof

```
lam x. when(fst(x), %xa. inl(<xa,snd(x)>), %xb. inr(<xb,snd(x)>))
```

of theorem $(P \vee Q) \wedge R \rightarrow P \wedge R \vee Q \wedge R$. The reduction demonstrates the use of

- context rules $\vee\mathsf{CE}$, $\vee\mathsf{CI}_1$, $\vee\mathsf{CI}_2$, $\wedge\mathsf{CE}_1$, $\wedge\mathsf{CE}_2$, $\wedge\mathsf{CI}$,
- tactic `vartac` applied on bound variables corresponding to 103_{205},
- tactic `try_all_rules` on different constructors on which a reduction rule can not be applied,
- destructors that do not take part in reduction before nor after reduction of the destructor's arguments, shown by first using an appropriate context rule on a destructor, then reducing the arguments followed by using tactic `try_all_rules` on the destructor corresponding to 100_{205}
- constructors that do not take part in reduction before nor after reduction of the constructor's arguments, shown by using an appropriate context rule on the constructor corresponding to 102_{205}

```
- goal PR.thy
= "?p : (P | Q) & R --> (P & R) | (Q & R)";
Level 0
?p : (P | Q) & R --> P & R | Q & R
 1. ?p : (P | Q) & R --> P & R | Q & R
val it = [] : thm list
- br redE 1;
Level 1
?p : (P | Q) & R --> P & R | Q & R
 1. ?a1 : (P | Q) & R --> P & R | Q & R
 2. ?a1 >> ?p : (P | Q) & R --> P & R | Q & R
val it = () : unit
- br propEx 1;
Level 2
?p : (P | Q) & R --> P & R | Q & R
 1. lam x.
       when
        (fst(x), %xa. inl(<xa,snd(x)>),
         %xa. inr(<xa,snd(x)>))
       >> ?p : (P | Q) & R --> P & R | Q & R
val it = () : unit
- br impCI 1;
Level 3
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y. y : (P | Q) & R ==>
         when
          (fst(y), %xa. inl(<xa,snd(y)>),
           %xa. inr(<xa,snd(y)>))
           >> ?g11(y) : P & R | Q & R
val it = () : unit
- br disjCE 1;
Level 4
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y. y : (P | Q) & R ==>
         fst(y) >> ?q12(y) : ?P12(y) | ?Q12(y)
 2. !!y x.
       [| y : (P | Q) & R; x : ?P12(y) |] ==>
       inl(<x,snd(y)>) >> ?f'12(y, x) :
          P & R | Q & R
 3. !!y x.
```

```
       [| y : (P | Q) & R; x : ?Q12(y) |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 4. !!y. y : (P | Q) & R ==>
         when(?q12(y), ?f'12(y), ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- br conjCE1 1;
Level 5
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y. y : (P | Q) & R ==>
         y >> ?e'13(y) :
            (?P12(y) | ?Q12(y)) & ?Q13(y)
 2. !!y. y : (P | Q) & R ==>
         fst(?e'13(y)) !> ?q12(y) :
            ?P12(y) | ?Q12(y)
 3. !!y x.
       [| y : (P | Q) & R; x : ?P12(y) |] ==>
       inl(<x,snd(y)>) >> ?f'12(y, x) :
          P & R | Q & R
 4. !!y x.
       [| y : (P | Q) & R; x : ?Q12(y) |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 5. !!y. y : (P | Q) & R ==>
         when(?q12(y), ?f'12(y), ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- bv 1;
Level 6
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y. y : (P | Q) & R ==>
         fst(y) !> ?q12(y) : P | Q
 2. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       inl(<x,snd(y)>) >> ?f'12(y, x) :
          P & R | Q & R
 3. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 4. !!y. y : (P | Q) & R ==>
         when(?q12(y), ?f'12(y), ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- bt 1;
Level 7
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       inl(<x,snd(y)>) >> ?f'12(y, x) :
          P & R | Q & R
 2. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 3. !!y. y : (P | Q) & R ==>
         when(fst(y), ?f'12(y), ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- br disjCI1 1;
Level 8
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       <x,snd(y)> >> ?a'16(y, x) : P & R
 2. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 3. !!y. y : (P | Q) & R ==>
         when
          (fst(y), %x. inl(?a'16(y, x)),
           ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- br conjCI 1;
Level 9
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       x >> ?a'17(y, x) : P
 2. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       snd(y) >> ?b'17(y, x) : R
 3. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 4. !!y. y : (P | Q) & R ==>
         when
          (fst(y),
           %x. inl(<?a'17(y, x),?b'17(y, x)>),
           ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- bv 1;
Level 10
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       snd(y) >> ?b'17(y, x) : R
 2. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 3. !!y. y : (P | Q) & R ==>
         when
          (fst(y), %x. inl(<x,?b'17(y, x)>),
           ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- br conjCE2 1;
Level 11
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       y >> ?e'19(y, x) : ?P19(y, x) & R
 2. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       snd(?e'19(y, x)) !> ?b'17(y, x) : R
 3. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 4. !!y. y : (P | Q) & R ==>
         when
          (fst(y), %x. inl(<x,?b'17(y, x)>),
           ?g'12(y))
            !> ?g11(y) : P & R | Q & R
val it = () : unit
- bv 1;
Level 12
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
```

```
 1. !!y x.
       [| y : (P | Q) & R; x : P |] ==>
       snd(y) !> ?b'17(y, x) : R
 2. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 3. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,?b'17(y, x)>),
            ?g'12(y))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- bt 1;
Level 13
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       inr(<x,snd(y)>) >> ?g'12(y, x) :
          P & R | Q & R
 2. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,snd(y)>),
            ?g'12(y))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- br disjCI2 1;
Level 14
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       <x,snd(y)> >> ?b'22(y, x) : Q & R
 2. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,snd(y)>),
            %x. inr(?b'22(y, x)))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- br conjCI 1;
Level 15
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       x >> ?a'23(y, x) : Q
 2. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       snd(y) >> ?b'23(y, x) : R
 3. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,snd(y)>),
            %x. inr(<?a'23(y, x),?b'23(y, x)>))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- bv 1;
Level 16
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       snd(y) >> ?b'23(y, x) : R
 2. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,snd(y)>),
            %x. inr(<x,?b'23(y, x)>))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- br conjCE2 1;
Level 17
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       y >> ?e'25(y, x) : ?P25(y, x) & R
 2. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       snd(?e'25(y, x)) !> ?b'23(y, x) : R
 3. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,snd(y)>),
            %x. inr(<x,?b'23(y, x)>))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- bv 1;
Level 18
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y x.
       [| y : (P | Q) & R; x : Q |] ==>
       snd(y) !> ?b'23(y, x) : R
 2. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,snd(y)>),
            %x. inr(<x,?b'23(y, x)>))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- bt 1;
Level 19
lam x. ?g11(x) : (P | Q) & R --> P & R | Q & R
 1. !!y. y : (P | Q) & R ==>
          when
           (fst(y), %x. inl(<x,snd(y)>),
            %x. inr(<x,snd(y)>))
             !> ?g11(y) : P & R | Q & R
val it = () : unit
- bt 1;
Level 20
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(<xa,snd(x)>))
: (P | Q) & R --> P & R | Q & R
No subgoals!
val it = () : unit
- val propEx' = result();
val propEx' =
  "lam x.
     when(fst(x),
          %xa. inl(<xa,snd(x)>),
          %xa. inr(<xa,snd(x)>))
   : (?P | ?Q) & ?R
     --> ?P & ?R | ?Q & ?R" : thm
```

Below is the reduction of the proof `lam x xa. xa ‘ x` of theorem $P \to (P \to Q) \to Q$. The reduction demonstrates the use of context rules $\to$ CI, $\to$ CE.

```
- goal PR.thy
= "?p : P --> (P --> Q) --> Q";
Level 0
?p : P --> (P --> Q) --> Q
 1. ?p : P --> (P --> Q) --> Q
val it = [] : thm list
- br redE 1;
Level 1
?p : P --> (P --> Q) --> Q
 1. ?a1 : P --> (P --> Q) --> Q
 2. ?a1 >> ?p : P --> (P --> Q) --> Q
val it = () : unit
- br impEx 1;
Level 2
?p : P --> (P --> Q) --> Q
 1. lam x xa. xa ` x >> ?p :
       P --> (P --> Q) --> Q
val it = () : unit
- br impCI 1;
Level 3
lam x. ?g5(x) : P --> (P --> Q) --> Q
 1. !!y. y : P ==>
          lam x. x ` y >> ?g5(y) :
             (P --> Q) --> Q
val it = () : unit
- br impCI 1;
Level 4
lam x xa. ?g6(x, xa) : P --> (P --> Q) --> Q
 1. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       ya ` y >> ?g6(y, ya) : Q
val it = () : unit
- br impCE 1;
Level 5
lam x xa. ?g6(x, xa) : P --> (P --> Q) --> Q
 1. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       ya >> ?e1'7(y, ya) : ?Q7(y, ya) --> Q
 2. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       y >> ?e2'7(y, ya) : ?Q7(y, ya)
 3. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       ?e1'7(y, ya) ` ?e2'7(y, ya)
          !> ?g6(y, ya) : Q
val it = () : unit
- bv 1;
Level 6
lam x xa. ?g6(x, xa) : P --> (P --> Q) --> Q
 1. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       y >> ?e2'7(y, ya) : P
 2. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       ya ` ?e2'7(y, ya) !> ?g6(y, ya) : Q
val it = () : unit
- bv 1;
Level 7
lam x xa. ?g6(x, xa) : P --> (P --> Q) --> Q
 1. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       ya ` y !> ?g6(y, ya) : Q
val it = () : unit
- bt 1;
Level 8
lam x xa. xa ` x : P --> (P --> Q) --> Q
No subgoals!
val it = () : unit
- val impEx' = result{};
val impEx' =
  "lam x xa. xa ` x
   : ?P --> (?P --> ?Q) --> ?Q" : thm
```

Below is the reduction of the proof

```
lam x. all xa. xsplit(x, %xaa u. [xaa,u^ xa])
```

of theorem $(\exists x \forall y\colon P(x,y)) \to (\forall y \exists x\colon P(x,y))$. The reduction demostrates the use of context rules $\forall$CI, $\forall$CE, $\exists$CE, $\exists$CI.

```
- goal PR.thy
= "?p : (EX (x::'a). (ALL (y::'b). P(x,y))) \
= \ --> (ALL (y::'b). (EX (x::'a). P(x,y)))";
Level 0
?p
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. ?p
    : (EX x. ALL y. P(x, y)) -->
      (ALL y. EX x. P(x, y))
val it = [] : thm list
- br redE 1;
Level 1
?p
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. ?a1
    : (EX x. ALL y. P(x, y)) -->
      (ALL y. EX x. P(x, y))
 2. ?a1 >> ?p :
       (EX x. ALL y. P(x, y)) -->
       (ALL y. EX x. P(x, y))
val it = () : unit
- br QuantEx 1;
Level 2
?p
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. lam x.
       all xa. xsplit(x, %xaa u. [xaa,u ^ xa])
       >> ?p :
       (EX x. ALL y. P(x, y)) -->
       (ALL y. EX x. P(x, y))
val it = () : unit
- br impCI 1;
Level 3
lam x. ?g8(x)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y. y : EX x. ALL y. P(x, y) ==>
          all x. xsplit(y, %xa u. [xa,u ^ x])
             >> ?g8(y) : ALL y. EX x. P(x, y)
```

```
val it = () : unit
- br allCI 1;
Level 4
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(y, %xa u. [xa,u ^ ya])
          >> ?g9(y, ya) : EX x. P(x, ya)
val it = () : unit
- br exCE 1;
Level 5
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       y >> ?q10(y, ya) : EX a. ?P10(y, ya, a)
 2. !!y ya x u.
       [| y : EX x. ALL y. P(x, y);
          u : ?P10(y, ya, x) |] ==>
       [x,u ^ ya] >> ?g10(y, ya, x, u) :
          EX x. P(x, ya)
 3. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(?q10(y, ya), ?g10(y, ya))
          !> ?g9(y, ya) : EX x. P(x, ya)
val it = () : unit
- bv 1;
Level 6
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya x u.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y) |] ==>
       [x,u ^ ya] >> ?g10(y, ya, x, u) :
          EX x. P(x, ya)
 2. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(y, ?g10(y, ya)) !> ?g9(y, ya) :
          EX x. P(x, ya)
val it = () : unit
- br exCI 1;
Level 7
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya x u.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y) |] ==>
       u ^ ya >> ?q12(y, ya, x, u) : P(x, ya)
 2. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(y, %x u. [x,?q12(y, ya, x, u)])
          !> ?g9(y, ya) : EX x. P(x, ya)
val it = () : unit
- br allCE 1;
Level 8
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya x u.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y) |] ==>
       u >> ?g13(y, ya, x, u) :
          ALL b. ?R13(y, ya, x, u, b)
 2. !!y ya x u.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y) |] ==>
       ?g13(y, ya, x, u) ^ ya
          !> ?q12(y, ya, x, u) :
          ?R13(y, ya, x, u, ya)
 3. !!y ya x u yb.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y);
          yb : ?R13(y, ya, x, u, ya) |] ==>
       yb : P(x, ya)
 4. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(y, %x u. [x,?q12(y, ya, x, u)])
          !> ?g9(y, ya) : EX x. P(x, ya)
val it = () : unit
- bv 1;
Level 9
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya x u.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y) |] ==>
       u ^ ya !> ?q12(y, ya, x, u) : P(x, ya)
 2. !!y ya x u yb.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y);
          yb : P(x, ya) |] ==>
       yb : P(x, ya)
 3. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(y, %x u. [x,?q12(y, ya, x, u)])
          !> ?g9(y, ya) : EX x. P(x, ya)
val it = () : unit
- bt 1;
Level 10
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya x u yb.
       [| y : EX x. ALL y. P(x, y);
          u : ALL y. P(x, y);
          yb : P(x, ya) |] ==>
       yb : P(x, ya)
 2. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(y, %x u. [x,u ^ ya])
          !> ?g9(y, ya) : EX x. P(x, ya)
val it = () : unit
- ba 1;
Level 11
lam x. all xa. ?g9(x, xa)
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       xsplit(y, %x u. [x,u ^ ya])
          !> ?g9(y, ya) : EX x. P(x, ya)
val it = () : unit
- bt 1;
Level 12
lam x. all xa. xsplit(x, %x u. [x,u ^ xa])
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
No subgoals!
```

```
val it = () : unit
- val QuantEx' = result();
val QuantEx' =
  "lam x. all xa. xsplit(x, %x u. [x,u ^ xa])
   : (EX x. ALL y. ?P(x, y)) -->
     (ALL y. EX x. ?P(x, y))" : thm
```

Below is the reduction of the proof/theorem combination

$$\frac{\texttt{q} : a = b}{\texttt{idpeel(q, \%x. ideq(x))} : b = a}$$

stating that if `q` is a proof of $a = b$ then `idpeel(q, %x. ideq(x))` is a proof of $b = a$. The reduction demonstrates the use of tactic `vartac` applied on free variables corresponding to 104_{205} and the use of context rules substCE, substCI.

```
- val [prems] = goal PR.thy
=   "q : a=b ==> ?p : b =a";
std_in:4.1-5.25 Warning: binding not exhaustive
          prems :: nil = ...
Level 0
?p : b = a
 1. ?p : b = a
val prems = "q : a = b  [q : a = b]" : thm
- br redE 1;
Level 1
?p : b = a
 1. ?a1 : b = a
 2. ?a1 >> ?p : b = a
val it = () : unit
- br sym 1;
Level 2
?p : b = a
 1. ?q2 : a = b
 2. idpeel(?q2, %x. ideq(x)) >> ?p : b = a
val it = () : unit
- br prems 1;
Level 3
?p : b = a
 1. idpeel(q, %x. ideq(x)) >> ?p : b = a
val it = () : unit
- br substCE 1;
Level 4
?p : b = a
 1. q >> ?q3 : ?a3 = ?b3
 2. !!y. y : ?P3(?a3, ?b3) ==> y : b = a
 3. !!c. ideq(c) >> ?g3(c) : ?P3(c, c)
 4. idpeel(?q3, ?g3) !> ?p : ?P3(?a3, ?b3)
val it = () : unit
- bv 1;
Level 5
?p : b = a
 1. !!y. y : ?P3(?a3, ?b3) ==> y : b = a
 2. !!c. ideq(c) >> ?g3(c) : ?P3(c, c)
 3. idpeel(q, ?g3) !> ?p : ?P3(?a3, ?b3)
val it = () : unit
- ba 1;
Level 6
?p : b = a
 1. !!c. ideq(c) >> ?g3(c) : c = c
 2. idpeel(q, ?g3) !> ?p : b = a
val it = () : unit
- br substCI 1;
Level 7
?p : b = a
 1. idpeel(q, %c. ideq(c)) !> ?p : b = a
val it = () : unit
- bt 1;
Level 8
idpeel(q, %c. ideq(c)) : b = a
No subgoals!
val it = () : unit
- val sym' = result();
val sym' =
  "?q : ?a = ?b ==>
   idpeel(?q, %c. ideq(c)) : ?b = ?a" : thm
```

Below is the reduction of the proof/theorem combination

$$\frac{\texttt{p} : a = b \qquad \texttt{q} : b = c}{\texttt{idpeel(q, \%x. lam x. x) `} \ \texttt{p} : a = c}$$

stating that if `p` is a proof of $a = b$ and `q` is a proof of $b = c$ then `idpeel(q, %x. lam x. x) ‘` `p` is a proof of $a = c$.

```
- val [p1,p2] = goal PR.thy
=  "[| p:a=b;  q:b=c |] ==> ?d:a=c";
std_in:15.1-16.33 Warning:
  binding not exhaustive
          p1 :: p2 :: nil = ...
Level 0
?d : a = c
 1. ?d : a = c
val p1 = "p : a = b  [p : a = b]" : thm
val p2 = "q : b = c  [q : b = c]" : thm
- br redE 1;
Level 1
?d : a = c
 1. ?a1 : a = c
 2. ?a1 >> ?d : a = c
val it = () : unit
-  br trans 1;
Level 2
```

```
?d : a = c
 1. ?p2 : a = ?b2
 2. ?q2 : ?b2 = c
 3. idpeel(?q2, %x. lam x. x) ‘ ?p2 >> ?d
    : a = c
val it = () : unit
-  br p1 1;
Level 3
?d : a = c
 1. ?q2 : b = c
 2. idpeel(?q2, %x. lam x. x) ‘ p >> ?d
    : a = c
val it = () : unit
-  br p2 1;
Level 4
?d : a = c
 1. idpeel(q, %x. lam x. x) ‘ p >> ?d : a = c
val it = () : unit
-  br impCE 1;
Level 5
?d : a = c
 1. idpeel(q, %x. lam x. x) >> ?e1'3
    : ?Q3 --> a = c
 2. p >> ?e2'3 : ?Q3
 3. ?e1'3 ‘ ?e2'3 !> ?d : a = c
val it = () : unit
-  br substCE 1;
Level 6
?d : a = c
 1. q >> ?q4 : ?a4 = ?b4
 2. !!y. y : ?P4(?a4, ?b4) ==> y : ?Q3 --> a = c
 3. !!c. lam x. x >> ?g4(c) : ?P4(c, c)
 4. idpeel(?q4, ?g4) !> ?e1'3 : ?P4(?a4, ?b4)
 5. p >> ?e2'3 : ?Q3
 6. ?e1'3 ‘ ?e2'3 !> ?d : a = c
val it = () : unit
-  bv 1;
Level 7
?d : a = c
 1. !!y. y : ?P4(?a4, ?b4) ==> y : ?Q3 --> a = c
 2. !!c. lam x. x >> ?g4(c) : ?P4(c, c)
 3. idpeel(q, ?g4) !> ?e1'3 : ?P4(?a4, ?b4)
 4. p >> ?e2'3 : ?Q3
 5. ?e1'3 ‘ ?e2'3 !> ?d : a = c
val it = () : unit
-  ba 1;
Level 8
?d : a = c
 1. !!c. lam x. x >> ?g4(c)
         : ?Q6(c, c) --> c = c
 2. idpeel(q, ?g4) !> ?e1'3
    : ?Q6(a, c) --> a = c
 3. p >> ?e2'3 : ?Q6(a, c)
 4. ?e1'3 ‘ ?e2'3 !> ?d : a = c
val it = () : unit
-  br impCI 1;
Level 9
?d : a = c
 1. !!c y. y : ?Q6(c, c) ==>
           y >> ?g7(c, y) : c = c
 2. idpeel(q, %c. lam x. ?g7(c, x))
    !> ?e1'3 : ?Q6(a, c) --> a = c
 3. p >> ?e2'3 : ?Q6(a, c)
 4. ?e1'3 ‘ ?e2'3 !> ?d : a = c
val it = () : unit
-  bv 1;
Level 10
?d : a = c
 1. idpeel(q, %c. lam x. x) !> ?e1'3
    : c = c --> a = c
 2. p >> ?e2'3 : c = c
 3. ?e1'3 ‘ ?e2'3 !> ?d : a = c
val it = () : unit
-  bt 1;
Level 11
?d : a = c
 1. p >> ?e2'3 : c = c
 2. idpeel(q, %c. lam x. x) ‘ ?e2'3 !> ?d
    : a = c
val it = () : unit
-  bv 1;
Level 12
?d : a = c
 1. idpeel(q, %c. lam x. x) ‘ p !> ?d : a = c
val it = () : unit
-  bt 1;
Level 13
idpeel(q, %c. lam x. x) ‘ p : a = c
No subgoals!
val it = () : unit
- val trans' = result();
val trans' =
  "[| ?p : ?a = ?b; ?q : ?b = ?c |] ==>
   idpeel(?q, %c. lam x. x) ‘ ?p
   : ?a = ?c" : thm
```

Below is the reduction of the proof/theorem combination

$$\frac{\texttt{p} : a = b}{\texttt{idpeel(p, \%x. ideq(t(x))} : \texttt{t}(a) = \texttt{t}(b)}$$

stating that if `p` is a proof of $a = b$ then `idpeel(p, %x. ideq(t(x)))` is a proof of $\texttt{t}(a) = \texttt{t}(b)$.

```
- val [p] = goal PR.thy
=   "[| p : a=b |]  ==>  ?q : t(a)=t(b)";
std_in:7.1-8.38 Warning: binding not exhaustive
          p :: nil = ...
Level 0
?q : t(a) = t(b)
 1. ?q : t(a) = t(b)
val p = "p : a = b  [p : a = b]" : thm
- br redE 1;
Level 1
?q : t(a) = t(b)
 1. ?a1 : t(a) = t(b)
 2. ?a1 >> ?q : t(a) = t(b)
val it = () : unit
-  by ((rtac subst_context 1) THEN (rtac p 1));
Level 2
?q : t(a) = t(b)
 1. idpeel(p, %x. ideq(t(x))) >> ?q :
```

```
      t(a) = t(b)
val it = () : unit
-  br substCE 1;
Level 3
?q : t(a) = t(b)
 1. p >> ?q3 : ?a3 = ?b3
 2. !!y. y : ?P3(?a3, ?b3) ==> y : t(a) = t(b)
 3. !!c. ideq(t(c)) >> ?g3(c) : ?P3(c, c)
 4. idpeel(?q3, ?g3) !> ?q : ?P3(?a3, ?b3)
val it = () : unit
-  bv 1;
Level 4
?q : t(a) = t(b)
 1. !!y. y : ?P3(?a3, ?b3) ==> y : t(a) = t(b)
 2. !!c. ideq(t(c)) >> ?g3(c) : ?P3(c, c)
 3. idpeel(p, ?g3) !> ?q : ?P3(?a3, ?b3)
val it = () : unit
-  ba 1;
Level 5
?q : t(a) = t(b)
 1. !!c. ideq(t(c)) >> ?g3(c) : t(c) = t(c)
 2. idpeel(p, ?g3) !> ?q : t(a) = t(b)
val it = () : unit
-  br substCI 1;
Level 6
?q : t(a) = t(b)
 1. idpeel(p, %c. ideq(t(c))) !> ?q :
       t(a) = t(b)
val it = () : unit
-  bt 1;
Level 7
idpeel(p, %c. ideq(t(c))) : t(a) = t(b)
No subgoals!
val it = () : unit
- val subst_context' = result();
val subst_context' =
  "?p : ?a = ?b ==>
   idpeel(?p, %c. ideq(?t(c)))
   : ?t(?a) = ?t(?b)" : thm
```

The proof session shows why theorems must be safe such that only one set of premises leads to a conclusion. Level 1 is proven using two rules right after each other since using only the theorem `subst_context` yields the new state

```
- br subst_context 1;
Level 2
?q : t(a) = t(b)
 1. ?p2 : t(a) = t(b)
 2. idpeel(?p2, %x. ideq(x)) >> ?q : t(a) = t(b)
val it = () : unit
```

by unifying `t` of theorem `subst_context` with the identity-function.

Reduction Rules

Each reduction rule can be demonstrated by creating a redex and then reducing the redex. For a given connective $\star$ the reduction rule for $\star$ is demonstrated by creating a goal `?p : ?P` and resolving the first subgoal in the current state in turn with

- rules $\longmapsto$ E, $\star$E, $\star$I
- then solving what can be proved by assumption and hypotheses,
- then rules $\star$CE, $\star$CI
- then solving what can be proved by tactic `vartac` and by assumption,
- then rule $\star$R,
- finally the rest of the subgoals can be proved by tactic `vartac` and by assumption.

Reduction rule $\rightarrow$ R is demonstrated with the proving session

```
- val [p] = goal PR.thy "a : P ==> ?p : ?P";
std_in:100.1-100.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P  [a : P]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 2
?p : ?P
 1. ?f2 : ?P2 --> ?P
 2. ?a2 : ?P2
 3. ?f2 ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 3
?p : ?P
 1. !!x. x : ?P2 ==> ?f3(x) : ?P
 2. ?a2 : ?P2
 3. (lam x. ?f3(x)) ‘ ?a2 >> ?p : ?P
val it = () : unit
-   ba 1;
Level 4
?p : ?P
 1. ?a2 : ?P
 2. (lam x. x) ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br p 1;
Level 5
?p : P
 1. (lam x. x) ‘ a >> ?p : P
val it = () : unit
-   br impCE 1;
Level 6
?p : P
 1. lam x. x >> ?e1'4 : ?Q4 --> P
 2. a >> ?e2'4 : ?Q4
 3. ?e1'4 ‘ ?e2'4 !> ?p : P
val it = () : unit
-   br impCI 1;
Level 7
?p : P
 1. !!y. y : ?Q4 ==> y >> ?g5(y) : P
 2. a >> ?e2'4 : ?Q4
 3. (lam x. ?g5(x)) ‘ ?e2'4 !> ?p : P
val it = () : unit
-   bv 1;
Level 8
?p : P
 1. a >> ?e2'4 : P
 2. (lam x. x) ‘ ?e2'4 !> ?p : P
val it = () : unit
-   bv 1;
Level 9
?p : P
 1. (lam x. x) ‘ a !> ?p : P
val it = () : unit
-   br impR 1;
Level 10
?p : P
 1. a >> ?p : P
val it = () : unit
-   bv 1;
Level 11
a : P
No subgoals!
val it = () : unit
- val impRex = result();
val impRex = "?a : ?P ==> ?a : ?P" : thm
```

The above proving session corresponds to 97_{205}.

Reduction rule $\wedge R_1$ is demonstrated with the proving session

```
- val [p] = goal PR.thy " a : P ==> ?p : ?P";
std_in:115.1-115.42 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P  [a : P]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br conjunct1 1;
Level 2
?p : ?P
 1. ?p2 : ?P & ?Q2
 2. fst(?p2) >> ?p : ?P
val it = () : unit
-   br conjI 1;
Level 3
?p : ?P
 1. ?a3 : ?P
 2. ?b3 : ?Q2
 3. fst(<?a3,?b3>) >> ?p : ?P
val it = () : unit
-   br p 1;
Level 4
?p : P
 1. ?b3 : ?Q2
 2. fst(<a,?b3>) >> ?p : P
val it = () : unit
-   br p 1;
Level 5
?p : P
 1. fst(<a,a>) >> ?p : P
val it = () : unit
-   br conjCE1 1;
Level 6
?p : P
 1. <a,a> >> ?e'4 : P & ?Q4
 2. fst(?e'4) !> ?p : P
val it = () : unit
```

```
-   br conjCI 1;
Level 7
?p : P
 1. a >> ?a'5 : P
 2. a >> ?b'5 : ?Q4
 3. fst(<?a'5,?b'5>) !> ?p : P
val it = () : unit
-   bv 1;
Level 8
?p : P
 1. a >> ?b'5 : ?Q4
 2. fst(<a,?b'5>) !> ?p : P
val it = () : unit
-   bv 1;
Level 9
?p : P
 1. fst(<a,a>) !> ?p : P
val it = () : unit
-   br conjR1 1;
Level 10
?p : P
 1. a >> ?p : P
val it = () : unit
-   bv 1;
Level 11
a : P
No subgoals!
val it = () : unit
- val conjR1ex = result();
val conjR1ex = "?a : ?P ==> ?a : ?P" : thm
```

Reduction rule $\wedge R_2$ is demostrated with the proving session

```
- val [p] = goal PR.thy " a : P ==> ?p : ?P";
std_in:128.1-128.42 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P  [a : P]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br conjunct2 1;
Level 2
?p : ?P
 1. ?p2 : ?P2 & ?P
 2. snd(?p2) >> ?p : ?P
val it = () : unit
-   br conjI 1;
Level 3
?p : ?P
 1. ?a3 : ?P2
 2. ?b3 : ?P
 3. snd(<?a3,?b3>) >> ?p : ?P
val it = () : unit
-   br p 1;
Level 4
?p : ?P
 1. ?b3 : ?P
 2. snd(<a,?b3>) >> ?p : ?P
val it = () : unit
-   br p 1;
Level 5
?p : P
 1. snd(<a,a>) >> ?p : P
val it = () : unit
-   br conjCE2 1;
Level 6
?p : P
 1. <a,a> >> ?e'4 : ?P4 & P
 2. snd(?e'4) !> ?p : P
val it = () : unit
-   br conjCI 1;
Level 7
?p : P
 1. a >> ?a'5 : ?P4
 2. a >> ?b'5 : P
 3. snd(<?a'5,?b'5>) !> ?p : P
val it = () : unit
-   bv 1;
Level 8
?p : P
 1. a >> ?b'5 : P
 2. snd(<a,?b'5>) !> ?p : P
val it = () : unit
-   bv 1;
Level 9
?p : P
 1. snd(<a,a>) !> ?p : P
val it = () : unit
-   br conjR2 1;
Level 10
?p : P
 1. a >> ?p : P
val it = () : unit
-   bv 1;
Level 11
a : P
No subgoals!
val it = () : unit
- val conjR2ex = result();
val conjR2ex = "?a : ?P ==> ?a : ?P" : thm
```

Reduction rule $\forall R$ is demonstrated with the following proving session

```
- val [p] = goal PR.thy
= "(!!x. f(x) : Q(x)) ==> ?p : ?P";
std_in:142.1-142.54 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p =
  "f(?x) : Q(?x)  [!!x. f(x) : Q(x)]" : thm
-   br redE 1;
Level 1
```

```
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br allE 1;
Level 2
?p : ?P
 1. ?p2 : ALL x. ?P2(x)
 2. !!y. y : ?P2(?z2) ==> y : ?P
 3. ?p2 ^ ?z2 >> ?p : ?P
val it = () : unit
-   br allI 1;
Level 3
?p : ?P
 1. !!x. ?f3(x) : ?P2(x)
 2. !!y. y : ?P2(?z2) ==> y : ?P
 3. all x. ?f3(x) ^ ?z2 >> ?p : ?P
val it = () : unit
-   br p 1;
Level 4
?p : ?P
 1. !!y. y : Q(?x4(?z2)) ==> y : ?P
 2. all x. f(?x4(x)) ^ ?z2 >> ?p : ?P
val it = () : unit
-   ba 1;
Level 5
?p : Q(?x4(?z2))
 1. all x. f(?x4(x)) ^ ?z2 >> ?p : Q(?x4(?z2))
val it = () : unit
-   br allCE 1;
Level 6
?p : Q(?x4(?z2))
 1. all x. f(?x4(x)) >> ?g6 : ALL b. ?R6(b)
 2. ?g6 ^ ?z2 !> ?p : ?R6(?z2)
 3. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   br allCI 1;
Level 7
?p : Q(?x4(?z2))
 1. !!y. f(?x4(y)) >> ?g8(y) : ?R6(y)
 2. all x. ?g8(x) ^ ?z2 !> ?p : ?R6(?z2)
 3. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   bv 1;
Level 8
?p : Q(?x4(?z2))
 1. all x. f(?x4(x)) ^ ?z2 !> ?p : ?R6(?z2)
 2. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   br allR 1;
Level 9
?p : Q(?x4(?z2))
 1. f(?x4(?z2)) >> ?p : ?R10(?z2)
 2. !!y. y : ?R10(?a10) ==> y : ?P10(?a10)
 3. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?P10(?z2) =?= ?R6(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   bv 1;
Level 10
f(?x4(?z2)) : Q(?x4(?z2))
 1. !!y. y : ?R10(?a10) ==> y : ?P10(?a10)
 2. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?P10(?z2) =?= ?R6(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   ba 1;
Level 11
f(?x4(?z2)) : Q(?x4(?z2))
 1. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?R10(?a10) =?= ?P10(?a10)
  ?P10(?z2) =?= ?R6(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   ba 1;
Level 12
f(?x4(?z2)) : Q(?x4(?z2))
No subgoals!

Flex-flex pairs:
  ?x14(?a10) =?= ?x13(?a10)
  ?x13(?z2) =?= ?x12(?z2)
  ?x12(?z2) =?= ?x7(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
- val allRex = result();
val allRex =
  "(!!x. ?f(x) : ?Q(x)) ==>
   ?f(?x) : ?Q(?x)" : thm
```

Reduction rule $\exists$R is demonstrated with the following proving session

```
- val [p] =
= goal PR.thy "(!!x. f(x) : Q(x)) ==> ?p : ?P";
std_in:5.1-5.54 Warning: binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p =
  "f(?x) : Q(?x)  [!!x. f(x) : Q(x)]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br exE 1;
Level 2
?p : ?P
 1. ?p2 : EX x. ?P2(x)
 2. !!x u. u : ?P2(x) ==> ?f2(x, u) : ?P
```

```
 3. xsplit(?p2, ?f2) >> ?p : ?P
val it = () : unit
-   br exI 1;
Level 3
?p : ?P
 1. ?p3 : ?P2(?x3)
 2. !!x u. u : ?P2(x) ==> ?f2(x, u) : ?P
 3. xsplit([?x3,?p3], ?f2) >> ?p : ?P
val it = () : unit
-   br p 1;
Level 4
?p : ?P
 1. !!x u. u : Q(?x5(x)) ==> ?f2(x, u) : ?P
 2. xsplit([?x3,f(?x5(?x3))], ?f2) >> ?p : ?P
val it = () : unit
-   ba 1;
Level 5
?p : Q(?x6)
 1. xsplit([?x3,f(?x6)], %x u. u) >> ?p : Q(?x6)
val it = () : unit
-   br exCE 1;
Level 6
?p : Q(?x6)
 1. [?x3,f(?x6)] >> ?q7 : EX a. ?P7(a)
 2. !!x u.
       u : ?P7(x) ==> u >> ?g7(x, u) : Q(?x6)
 3. xsplit(?q7, ?g7) !> ?p : Q(?x6)
val it = () : unit
-   br exCI 1;
Level 7
?p : Q(?x6)
 1. f(?x6) >> ?q8 : ?P7(?x3)
 2. !!x u.
       u : ?P7(x) ==> u >> ?g7(x, u) : Q(?x6)
 3. xsplit([?x3,?q8], ?g7) !> ?p : Q(?x6)
val it = () : unit
-   bv 1;
Level 8
?p : Q(?x6)
 1. !!x u.
       u : ?P7(x) ==> u >> ?g7(x, u) : Q(?x6)
 2. xsplit([?x3,f(?x6)], ?g7) !> ?p : Q(?x6)
val it = () : unit
-   bv 1;
Level 9
?p : Q(?x11)
 1. xsplit([?x3,f(?x11)], %x u. u) !> ?p :
       Q(?x11)
val it = () : unit
-   br exR 1;
Level 10
?p : Q(?x11)
 1. f(?x11) >> ?p : Q(?x11)
val it = () : unit
-   bv 1;
Level 11
f(?x11) : Q(?x11)
No subgoals!
val it = () : unit
- val exRex = result();
val exRex =
  "(!!x. ?f(x) : ?Q(x)) ==>
   ?f(?x) : ?Q(?x)" : thm
```

Reduction rule $\vee R_1$ is demonstrated with the following proving session:

```
- val [p] = goal PR.thy "a : Q ==> ?p : ?P";
std_in:20.1-20.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : Q  [a : Q]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br disjE 1;
Level 2
?p : ?P
 1. ?a2 : ?P2 | ?Q2
 2. !!x. x : ?P2 ==> ?f2(x) : ?P
 3. !!x. x : ?Q2 ==> ?g2(x) : ?P
 4. when(?a2, ?f2, ?g2) >> ?p : ?P
val it = () : unit
-   br disjI1 1;
Level 3
?p : ?P
 1. ?a3 : ?P2
 2. !!x. x : ?P2 ==> ?f2(x) : ?P
 3. !!x. x : ?Q2 ==> ?g2(x) : ?P
 4. when(inl(?a3), ?f2, ?g2) >> ?p : ?P
val it = () : unit
-   br p 1;
Level 4
?p : ?P
 1. !!x. x : Q ==> ?f2(x) : ?P
 2. !!x. x : ?Q2 ==> ?g2(x) : ?P
 3. when(inl(a), ?f2, ?g2) >> ?p : ?P
val it = () : unit
-   ba 1;
Level 5
?p : Q
 1. !!x. x : ?Q2 ==> ?g2(x) : Q
 2. when(inl(a), %x. x, ?g2) >> ?p : Q
val it = () : unit
-   ba 1;
Level 6
?p : Q
 1. when(inl(a), %x. x, %x. x) >> ?p : Q
val it = () : unit
-   br disjCE 1;
Level 7
?p : Q
 1. inl(a) >> ?q4 : ?P4 | ?Q4
 2. !!x. x : ?P4 ==> x >> ?f'4(x) : Q
 3. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
 4. when(?q4, ?f'4, ?g'4) !> ?p : Q
val it = () : unit
-   br disjCI1 1;
Level 8
?p : Q
 1. a >> ?a'5 : ?P4
 2. !!x. x : ?P4 ==> x >> ?f'4(x) : Q
 3. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
```

```
 4. when(inl(?a'5), ?f'4, ?g'4) !> ?p : Q
val it = () : unit
-   bv 1;
Level 9
?p : Q
 1. !!x. x : ?P4 ==> x >> ?f'4(x) : Q
 2. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
 3. when(inl(a), ?f'4, ?g'4) !> ?p : Q
val it = () : unit
-   bv 1;
Level 10
?p : Q
 1. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
 2. when(inl(a), %x. x, ?g'4) !> ?p : Q
val it = () : unit
-   bv 1;
Level 11
?p : Q
 1. when(inl(a), %x. x, %x. x) !> ?p : Q
val it = () : unit
-   br disjR1 1;
Level 12
?p : Q
 1. a >> ?p : Q
val it = () : unit
-   bv 1;
Level 13
a : Q
No subgoals!
val it = () : unit
- val disjR1ex = result();
val disjR1ex = "?a : ?Q ==> ?a : ?Q" : thm
```

Reduction rule $\vee R_2$ is demonstrated with the following proving session.

```
- val [p] = goal PR.thy "a : Q ==> ?p : ?P";
std_in:35.1-35.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : Q  [a : Q]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br disjE 1;
Level 2
?p : ?P
 1. ?a2 : ?P2 | ?Q2
 2. !!x. x : ?P2 ==> ?f2(x) : ?P
 3. !!x. x : ?Q2 ==> ?g2(x) : ?P
 4. when(?a2, ?f2, ?g2) >> ?p : ?P
val it = () : unit
-   br disjI2 1;
Level 3
?p : ?P
 1. ?b3 : ?Q2
 2. !!x. x : ?P2 ==> ?f2(x) : ?P
 3. !!x. x : ?Q2 ==> ?g2(x) : ?P
 4. when(inr(?b3), ?f2, ?g2) >> ?p : ?P
val it = () : unit
-   br p 1;
Level 4
?p : ?P
 1. !!x. x : ?P2 ==> ?f2(x) : ?P
 2. !!x. x : Q ==> ?g2(x) : ?P
 3. when(inr(a), ?f2, ?g2) >> ?p : ?P
val it = () : unit
-   ba 1;
Level 5
?p : ?P
 1. !!x. x : Q ==> ?g2(x) : ?P
 2. when(inr(a), %x. x, ?g2) >> ?p : ?P
val it = () : unit
-   ba 1;
Level 6
?p : Q
 1. when(inr(a), %x. x, %x. x) >> ?p : Q
val it = () : unit
-   br disjCE 1;
Level 7
?p : Q
 1. inr(a) >> ?q4 : ?P4 | ?Q4
 2. !!x. x : ?P4 ==> x >> ?f'4(x) : Q
 3. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
 4. when(?q4, ?f'4, ?g'4) !> ?p : Q
val it = () : unit
-   br disjCI2 1;
Level 8
?p : Q
 1. a >> ?b'5 : ?Q4
 2. !!x. x : ?P4 ==> x >> ?f'4(x) : Q
 3. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
 4. when(inr(?b'5), ?f'4, ?g'4) !> ?p : Q
val it = () : unit
-   bv 1;
Level 9
?p : Q
 1. !!x. x : ?P4 ==> x >> ?f'4(x) : Q
 2. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
 3. when(inr(a), ?f'4, ?g'4) !> ?p : Q
val it = () : unit
-   bv 1;
Level 10
?p : Q
 1. !!x. x : ?Q4 ==> x >> ?g'4(x) : Q
 2. when(inr(a), %x. x, ?g'4) !> ?p : Q
val it = () : unit
-   bv 1;
Level 11
?p : Q
 1. when(inr(a), %x. x, %x. x) !> ?p : Q
val it = () : unit
-   br disjR2 1;
Level 12
?p : Q
 1. a >> ?p : Q
val it = () : unit
-   bv 1;
Level 13
a : Q
No subgoals!
val it = () : unit
- val disjR2ex = result();
val disjR2ex = "?a : ?Q ==> ?a : ?Q" : thm
```

Reduction rule substR is demonstrated with the following proving session.

```
- val [p] = goal PR.thy "a : Q ==> ?p : ?P";
std_in:17.1-17.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : Q  [a : Q]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br subst 1;
Level 2
?p : ?P
 1. ?p2 : ?x2 = ?y2
 2. !!x. ?f2(x) : ?P2(x, x)
 3. !!q. q : ?P2(?x2, ?y2) ==> q : ?P
 4. idpeel(?p2, %x. ?f2(x)) >> ?p : ?P
val it = () : unit
-   br refl 1;
Level 3
?p : ?P
 1. !!x. ?f2(x) : ?P2(x, x)
 2. !!q. q : ?P2(?x2, ?x2) ==> q : ?P
 3. idpeel(ideq(?x2), %x. ?f2(x)) >> ?p : ?P
val it = () : unit
-   br p 1;
Level 4
?p : ?P
 1. !!q. q : Q ==> q : ?P
 2. idpeel(ideq(?x2), %x. a) >> ?p : ?P
val it = () : unit
-   ba 1;
Level 5
?p : Q
 1. idpeel(ideq(?x2), %x. a) >> ?p : Q
val it = () : unit
-   br substCE 1;
Level 6
?p : Q
 1. ideq(?x2) >> ?q4 : ?a4 = ?b4
 2. !!y. y : ?P4(?a4, ?b4) ==> y : Q
 3. !!c. a >> ?g4(c) : ?P4(c, c)
 4. idpeel(?q4, ?g4) !> ?p : ?P4(?a4, ?b4)
val it = () : unit
-   br substCI 1;
Level 7
?p : Q
 1. !!y. y : ?P4(?x2, ?x2) ==> y : Q
 2. !!c. a >> ?g4(c) : ?P4(c, c)
 3. idpeel(ideq(?x2), ?g4) !> ?p : ?P4(?x2, ?x2)
val it = () : unit
- ba 1;
Level 8
?p : Q
 1. !!c. a >> ?g4(c) : Q
 2. idpeel(ideq(?x2), ?g4) !> ?p : Q
val it = () : unit
-   bv 1;
Level 9
?p : Q
 1. idpeel(ideq(?x2), %c. a) !> ?p : Q
val it = () : unit
-   br substR 1;
Level 10
?p : Q
 1. a >> ?p : ?R7(?x2, ?x2)
 2. !!y. y : ?R7(?b7, ?b7) ==> y : Q
val it = () : unit
-   bv 1;
Level 11
a : Q
 1. !!y. y : ?R7(?b7, ?b7) ==> y : Q
val it = () : unit
-   ba 1;
Level 12
a : Q
No subgoals!
val it = () : unit
- val substRex = result();
val substRex = "?a : ?Q ==> ?a : ?Q" : thm
```

A redex as argument of a destructor corresponding to 98_{205} is demonstrated with the following proving session.

```
- val [p] = goal PR.thy "a : P & Q ==> ?p : ?P";
std_in:64.1-64.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P & Q  [a : P & Q]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br conjunct1 1;
Level 2
?p : ?P
 1. ?p2 : ?P & ?Q2
 2. fst(?p2) >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 3
?p : ?P
 1. ?f3 : ?P3 --> ?P & ?Q2
 2. ?a3 : ?P3
 3. fst(?f3 ‘ ?a3) >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 4
?p : ?P
 1. !!x. x : ?P3 ==> ?f4(x) : ?P & ?Q2
 2. ?a3 : ?P3
 3. fst((lam x. ?f4(x)) ‘ ?a3) >> ?p : ?P
val it = () : unit
```

```
-   ba 1;
Level 5
?p : ?P5
 1. ?a3 : ?P5 & ?Q6
 2. fst((lam x. x) ‘ ?a3) >> ?p : ?P5
val it = () : unit
-   br p 1;
Level 6
?p : P
 1. fst((lam x. x) ‘ a) >> ?p : P
val it = () : unit
-   br conjCE1 1;
Level 7
?p : P
 1. (lam x. x) ‘ a >> ?e'7 : P & ?Q7
 2. fst(?e'7) !> ?p : P
val it = () : unit
-   br impCE 1;
Level 8
?p : P
 1. lam x. x >> ?e1'8 : ?Q8 --> P & ?Q7
 2. a >> ?e2'8 : ?Q8
 3. ?e1'8 ‘ ?e2'8 !> ?e'7 : P & ?Q7
 4. fst(?e'7) !> ?p : P
val it = () : unit
-   br impCI 1;
Level 9
?p : P
 1. !!y. y : ?Q8 ==> y >> ?g9(y) : P & ?Q7
 2. a >> ?e2'8 : ?Q8
 3. (lam x. ?g9(x)) ‘ ?e2'8 !> ?e'7 : P & ?Q7
 4. fst(?e'7) !> ?p : P
val it = () : unit
-   bv 1;
Level 10
?p : P
 1. a >> ?e2'8 : P & ?Q11
 2. (lam x. x) ‘ ?e2'8 !> ?e'7 : P & ?Q11
 3. fst(?e'7) !> ?p : P
val it = () : unit
-   bv 1;
Level 11
?p : P
 1. (lam x. x) ‘ a !> ?e'7 : P & ?Q11
 2. fst(?e'7) !> ?p : P
val it = () : unit
-   br impR 1;
Level 12
?p : P
 1. a >> ?e'7 : P & ?Q11
 2. fst(?e'7) !> ?p : P
val it = () : unit
-   bv 1;
Level 13
?p : P
 1. fst(a) !> ?p : P
val it = () : unit
-   bt 1;
Level 14
fst(a) : P
No subgoals!
val it = () : unit
- val destRedex = result();
val destRedex =
  "?a : ?P & ?Q ==> fst(?a) : ?P" : thm
```

A redex as argument of a constructor corresponding to 99_{205} is demonstrated with the following proving session.

```
-   goal PR.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 2
?p : ?P2 --> ?Q2
 1. !!x. x : ?P2 ==> ?f2(x) : ?Q2
 2. lam x. ?f2(x) >> ?p : ?P2 --> ?Q2
val it = () : unit
-   br mp 1;
Level 3
?p : ?P2 --> ?Q2
 1. !!x. x : ?P2 ==> ?f3(x) : ?P3(x) --> ?Q2
 2. !!x. x : ?P2 ==> ?a3(x) : ?P3(x)
 3. lam x. ?f3(x) ‘ ?a3(x) >> ?p : ?P2 --> ?Q2
val it = () : unit
-   br impI 1;
Level 4
?p : ?P2 --> ?Q2
 1. !!x xa.
       [| x : ?P2; xa : ?P3(x) |] ==>
       ?f4(x, xa) : ?Q2
 2. !!x. x : ?P2 ==> ?a3(x) : ?P3(x)
 3. lam x. (lam xa. ?f4(x, xa)) ‘ ?a3(x) >> ?p :
       ?P2 --> ?Q2
val it = () : unit
-   ba 1;
Level 5
?p : ?P2 --> ?P2
 1. !!x. x : ?P2 ==> ?a3(x) : ?P3(x)
 2. lam x. (lam xa. x) ‘ ?a3(x) >> ?p :
       ?P2 --> ?P2
val it = () : unit
-   back();
Level 5
?p : ?P2 --> ?Q2
 1. !!x. x : ?P2 ==> ?a3(x) : ?Q2
 2. lam x. (lam x. x) ‘ ?a3(x) >> ?p :
       ?P2 --> ?Q2
val it = () : unit
-   ba 1;
Level 6
?p : ?P2 --> ?P2
 1. lam x. (lam x. x) ‘ x >> ?p : ?P2 --> ?P2
val it = () : unit
-   br impCI 1;
Level 7
lam x. ?g5(x) : ?P2 --> ?P2
```

```
 1. !!y. y : ?P2 ==>
         (lam x. x) ‘ y >> ?g5(y) : ?P2
val it = () : unit
-   br impCE 1;
Level 8
lam x. ?g5(x) : ?P2 --> ?P2
 1. !!y. y : ?P2 ==>
         lam x. x >> ?e1’6(y) : ?Q6(y) --> ?P2
 2. !!y. y : ?P2 ==> y >> ?e2’6(y) : ?Q6(y)
 3. !!y. y : ?P2 ==>
         ?e1’6(y) ‘ ?e2’6(y) !> ?g5(y) : ?P2
val it = () : unit
-   br impCI 1;
Level 9
lam x. ?g5(x) : ?P2 --> ?P2
 1. !!y ya.
       [| y : ?P2; ya : ?Q6(y) |] ==>
       ya >> ?g7(y, ya) : ?P2
 2. !!y. y : ?P2 ==> y >> ?e2’6(y) : ?Q6(y)
 3. !!y. y : ?P2 ==>
         (lam x. ?g7(y, x)) ‘ ?e2’6(y)
            !> ?g5(y) : ?P2
val it = () : unit
-   bv 1;
Level 10
lam x. ?g5(x) : ?P2 --> ?P2
 1. !!y. y : ?P2 ==> y >> ?e2’6(y) : ?P2
 2. !!y. y : ?P2 ==>
         (lam x. x) ‘ ?e2’6(y) !> ?g5(y) : ?P2
val it = () : unit
-   bv 1;
Level 11
lam x. ?g5(x) : ?P2 --> ?P2
 1. !!y. y : ?P2 ==>
         (lam x. x) ‘ y !> ?g5(y) : ?P2
val it = () : unit
-   br impR 1;
Level 12
lam x. ?g5(x) : ?P2 --> ?P2
 1. !!y. y : ?P2 ==> y >> ?g5(y) : ?P2
val it = () : unit
-   bv 1;
Level 13
lam x. x : ?P2 --> ?P2
No subgoals!
val it = () : unit
- val constredex = result();
val constredex = "lam x. x : ?P --> ?P" : thm
```

A destructor forming a redex after but not before reduction of the destructor's arguments corresponding to *101*$_{205}$ is demonstrated with the following proving session.

```
- val [p] = goal PR.thy "a : P ==> ?p : ?P";
std_in:99.1-99.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P  [a : P]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 2
?p : ?P
 1. ?f2 : ?P2 --> ?P
 2. ?a2 : ?P2
 3. ?f2 ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br conjunct1 1;
Level 3
?p : ?P
 1. ?p3 : (?P2 --> ?P) & ?Q3
 2. ?a2 : ?P2
 3. fst(?p3) ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 4
?p : ?P
 1. ?f4 : ?P4 --> (?P2 --> ?P) & ?Q3
 2. ?a4 : ?P4
 3. ?a2 : ?P2
 4. fst(?f4 ‘ ?a4) ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 5
?p : ?P
 1. !!x. x : ?P4 ==> ?f5(x) : (?P2 --> ?P) & ?Q3
 2. ?a4 : ?P4
 3. ?a2 : ?P2
 4. fst((lam x. ?f5(x)) ‘ ?a4) ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br conjI 1;
Level 6
?p : ?P
 1. !!x. x : ?P4 ==> ?a6(x) : ?P2 --> ?P
 2. !!x. x : ?P4 ==> ?b6(x) : ?Q3
 3. ?a4 : ?P4
 4. ?a2 : ?P2
 5. fst((lam x. <?a6(x),?b6(x)>) ‘ ?a4) ‘ ?a2
       >> ?p : ?P
val it = () : unit
-   ba 1;
Level 7
?p : ?P8
 1. !!x. x : ?P7 --> ?P8 ==> ?b6(x) : ?Q3
 2. ?a4 : ?P7 --> ?P8
 3. ?a2 : ?P7
 4. fst((lam x. <x,?b6(x)>) ‘ ?a4) ‘ ?a2 >> ?p :
       ?P8
val it = () : unit
-   ba 1;
Level 8
?p : ?P10
 1. ?a4 : ?P9 --> ?P10
 2. ?a2 : ?P9
 3. fst((lam x. <x,x>) ‘ ?a4) ‘ ?a2 >> ?p : ?P10
val it = () : unit
-   br impI 1;
Level 9
?p : ?P10
 1. !!x. x : ?P9 ==> ?f11(x) : ?P10
```

```
 2. ?a2 : ?P9
 3. fst((lam x. <x,x>) ‘ (lam x. ?f11(x))) ‘
    ?a2
       >> ?p : ?P10
val it = () : unit
-   ba 1;
Level 10
?p : ?P9
 1. ?a2 : ?P9
 2. fst((lam x. <x,x>) ‘ (lam x. x)) ‘ ?a2
       >> ?p : ?P9
val it = () : unit
-   br p 1;
Level 11
?p : P
 1. fst((lam x. <x,x>) ‘ (lam x. x)) ‘ a >> ?p :
       P
val it = () : unit
-   br impCE 1;
Level 12
?p : P
 1. fst((lam x. <x,x>) ‘ (lam x. x)) >> ?e1’12 :
       ?Q12 --> P
 2. a >> ?e2’12 : ?Q12
 3. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br conjCE1 1;
Level 13
?p : P
 1. (lam x. <x,x>) ‘ (lam x. x) >> ?e’13 :
       (?Q12 --> P) & ?Q13
 2. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 3. a >> ?e2’12 : ?Q12
 4. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br impCE 1;
Level 14
?p : P
 1. lam x. <x,x> >> ?e1’14 :
       ?Q14 --> (?Q12 --> P) & ?Q13
 2. lam x. x >> ?e2’14 : ?Q14
 3. ?e1’14 ‘ ?e2’14 !> ?e’13 :
       (?Q12 --> P) & ?Q13
 4. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 5. a >> ?e2’12 : ?Q12
 6. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br impCI 1;
Level 15
?p : P
 1. !!y. y : ?Q14 ==>
         <y,y> >> ?g15(y) : (?Q12 --> P) & ?Q13
 2. lam x. x >> ?e2’14 : ?Q14
 3. (lam x. ?g15(x)) ‘ ?e2’14 !> ?e’13 :
       (?Q12 --> P) & ?Q13
 4. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 5. a >> ?e2’12 : ?Q12
 6. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br conjCI 1;
Level 16
?p : P
 1. !!y. y : ?Q14 ==> y >> ?a’16(y) : ?Q12 --> P
 2. !!y. y : ?Q14 ==> y >> ?b’16(y) : ?Q13
 3. lam x. x >> ?e2’14 : ?Q14
 4. (lam x. <?a’16(x),?b’16(x)>) ‘ ?e2’14
       !> ?e’13 : (?Q12 --> P) & ?Q13
 5. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 6. a >> ?e2’12 : ?Q12
 7. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bv 1;
Level 17
?p : P
 1. !!y. y : ?Q18 --> P ==> y >> ?b’16(y) : ?Q13
 2. lam x. x >> ?e2’14 : ?Q18 --> P
 3. (lam x. <x,?b’16(x)>) ‘ ?e2’14 !> ?e’13 :
       (?Q18 --> P) & ?Q13
 4. fst(?e’13) !> ?e1’12 : ?Q18 --> P
 5. a >> ?e2’12 : ?Q18
 6. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bv 1;
Level 18
?p : P
 1. lam x. x >> ?e2’14 : ?Q20 --> P
 2. (lam x. <x,x>) ‘ ?e2’14 !> ?e’13 :
       (?Q20 --> P) & (?Q20 --> P)
 3. fst(?e’13) !> ?e1’12 : ?Q20 --> P
 4. a >> ?e2’12 : ?Q20
 5. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br impCI 1;
Level 19
?p : P
 1. !!y. y : ?Q20 ==> y >> ?g21(y) : P
 2. (lam x. <x,x>) ‘ (lam x. ?g21(x)) !> ?e’13 :
       (?Q20 --> P) & (?Q20 --> P)
 3. fst(?e’13) !> ?e1’12 : ?Q20 --> P
 4. a >> ?e2’12 : ?Q20
 5. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bv 1;
Level 20
?p : P
 1. (lam x. <x,x>) ‘ (lam x. x) !> ?e’13 :
       (P --> P) & (P --> P)
 2. fst(?e’13) !> ?e1’12 : P --> P
 3. a >> ?e2’12 : P
 4. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br impR 1;
Level 21
?p : P
 1. <lam x. x,lam x. x> >> ?e’13 :
       (P --> P) & (P --> P)
 2. fst(?e’13) !> ?e1’12 : P --> P
 3. a >> ?e2’12 : P
 4. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br conjCI 1;
Level 22
?p : P
 1. lam x. x >> ?a’24 : P --> P
 2. lam x. x >> ?b’24 : P --> P
 3. fst(<?a’24,?b’24>) !> ?e1’12 : P --> P
 4. a >> ?e2’12 : P
 5. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br impCI 1;
Level 23
?p : P
 1. !!y. y : P ==> y >> ?g25(y) : P
 2. lam x. x >> ?b’24 : P --> P
```

```
 3. fst(<lam x. ?g25(x),?b'24>) !> ?e1'12 :
       P --> P
 4. a >> ?e2'12 : P
 5. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
-   bv 1;
Level 24
?p : P
 1. lam x. x >> ?b'24 : P --> P
 2. fst(<lam x. x,?b'24>) !> ?e1'12 : P --> P
 3. a >> ?e2'12 : P
 4. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
-   br impCI 1;
Level 25
?p : P
 1. !!y. y : P ==> y >> ?g27(y) : P
 2. fst(<lam x. x,lam x. ?g27(x)>) !> ?e1'12 :
       P --> P
 3. a >> ?e2'12 : P
 4. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
-   bv 1;
Level 26
?p : P
 1. fst(<lam x. x,lam x. x>) !> ?e1'12 : P --> P
 2. a >> ?e2'12 : P
 3. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
-   br conjR1 1;
Level 27
?p : P
 1. lam x. x >> ?e1'12 : P --> P
 2. a >> ?e2'12 : P
 3. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
-   br impCI 1;
Level 28
?p : P
 1. !!y. y : P ==> y >> ?g30(y) : P
 2. a >> ?e2'12 : P
 3. (lam x. ?g30(x)) ‘ ?e2'12 !> ?p : P
val it = () : unit
-   bv 1;
Level 29
?p : P
 1. a >> ?e2'12 : P
 2. (lam x. x) ‘ ?e2'12 !> ?p : P
val it = () : unit
-   bv 1;
Level 30
?p : P
 1. (lam x. x) ‘ a !> ?p : P
val it = () : unit
-   br impR 1;
Level 31
?p : P
 1. a >> ?p : P
val it = () : unit
-   bv 1;
Level 32
a : P
No subgoals!
val it = () : unit
- val redexAfter = result();
val redexAfter = "?a : ?P ==> ?a : ?P" : thm
```

Hereby both the context rules and the reduction rules are demonstrated and it is demonstrated that reduction works well for all of cases 97_{205}–104_{205}; hereby it should be possible to conclude that there is a certain possibility that it is possible to reduce every proof provable in object logic `FirstOrderProof` with object logic `PR`.

B.4 First Order Logic with Proofs and Program Evaluation

An object logic for evaluating proofs as they were programs, where the proofs are proofs of natural deductive intuitionistic first order logic propositions, is not implemented in Isabelle and is therefore not a part of the Isabelle distribution. Therefore an object logic `PE` corresponding to formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$ in section 4.7.4 must be constructed.

B.4.1 Implementing Theory

Formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$ specifies that program evaluation shall be reduction of proofs that are proven using formal theory $\mathcal{IFOLP}$. Therefore object logic `PE` shall be built on top of object logic `FirstOrderProof`.

Object logic `Reduction` specifies parts of proof reduction shared with object logic `PR`, object logic `PE` for program evaluation implements the parts of formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$ that is not shared with formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$. Object logic `PE` is in appendix A.6.

B.4.2 Examples

Examples demonstrating the use of object logic `PE` may cover the differences between the `PR` object logic and the `PE` object logic. No rules or axioms are added to the reduction rules and destructor context rules in object logic `Reduction`, only a tactic `try_allRules` is added which shall be used on all variables and constructors. The only thing to demonstrate is then the `try_allRules` tactic applied on constructors and variables and the functionality of program evaluation.

To demonstrate the functionality of the program evaluation the following cases shall be demonstrated:

105$_{224}$ A redex outermost of a proof; the redex shall be reduced.

106$_{224}$ A redex that is argument of a destructor; the redex shall be reduced.

107$_{224}$ A redex that is argument of a constructor; the redex shall *not* be reduced.

108$_{224}$ A destructor that does not take part in a redex before nor after reduction of the arguments of the destructor; the destructor shall not be part in a reduction.

109$_{224}$ A destructor that does not take part in a redex before reduction of the arguments of the destructor but do take part in a redex after reduction of the arguments of the destructor; the redex shall be reduced when it is created.

110$_{224}$ A constructor that does not take part in a redex; the constructor shall not be part in a reduction.

111$_{224}$ A proof that is a variable; the variable itself is a reduced proof.

Reduction of a redex outermost of a proof corresponding to *105*$_{224}$ is demonstrated with the following proving session. As well tactic `try_allRules` applied on both a constructor corresponding to *110*$_{224}$ and on a free variable corresponding to *111*$_{224}$ is demonstrated.

```
- val [p] = goal PE.thy "a : P ==> ?p : ?P";
std_in:76.1-76.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P  [a : P]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 2
?p : ?P
 1. ?f2 : ?P2 --> ?P
 2. ?a2 : ?P2
 3. ?f2 ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 3
?p : ?P
 1. !!x. x : ?P2 ==> ?f3(x) : ?P
 2. ?a2 : ?P2
 3. (lam x. ?f3(x)) ‘ ?a2 >> ?p : ?P
val it = () : unit
-   ba 1;
Level 4
?p : ?P
 1. ?a2 : ?P
 2. (lam x. x) ‘ ?a2 >> ?p : ?P
val it = () : unit
-   br p 1;
Level 5
?p : P
 1. (lam x. x) ‘ a >> ?p : P
val it = () : unit
-   br impCE 1;
Level 6
?p : P
 1. lam x. x >> ?e1'4 : ?Q4 --> P
 2. a >> ?e2'4 : ?Q4
 3. ?e1'4 ‘ ?e2'4 !> ?p : P
val it = () : unit
-   bta 1;
Level 7
?p : P
 1. a >> ?e2'4 : ?Q4
 2. (lam x. x) ‘ ?e2'4 !> ?p : P
val it = () : unit
-   bta 1;
Level 8
```

```
?p : P
 1. (lam x. x) ‘ a !> ?p : P
val it = () : unit
-   br impR 1;
Level 9
?p : P
 1. a >> ?p : P
val it = () : unit
-   bta 1;
Level 10
a : P
No subgoals!
val it = () : unit
- val impRex = result();
val impRex = "?a : ?P ==> ?a : ?P" : thm
```

Reduction of a redex that is argument of a destructor corresponding to 106_{224} and reduction of a destructor that does not take part in a redex corresponding to 108_{224} is demonstrated with the following proving session.

```
- val [p] = goal PE.thy "a : P & Q ==> ?p : ?P";
std_in:88.1-88.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P & Q  [a : P & Q]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br conjunct1 1;
Level 2
?p : ?P
 1. ?p2 : ?P & ?Q2
 2. fst(?p2) >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 3
?p : ?P
 1. ?f3 : ?P3 --> ?P & ?Q2
 2. ?a3 : ?P3
 3. fst(?f3 ‘ ?a3) >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 4
?p : ?P
 1. !!x. x : ?P3 ==> ?f4(x) : ?P & ?Q2
 2. ?a3 : ?P3
 3. fst((lam x. ?f4(x)) ‘ ?a3) >> ?p : ?P
val it = () : unit
-   ba 1;
Level 5
?p : ?P5
 1. ?a3 : ?P5 & ?Q6
 2. fst((lam x. x) ‘ ?a3) >> ?p : ?P5
val it = () : unit
-   br p 1;
Level 6
?p : P
 1. fst((lam x. x) ‘ a) >> ?p : P
val it = () : unit
-   br conjCE1 1;
Level 7
?p : P
 1. (lam x. x) ‘ a >> ?e'7 : P & ?Q7
 2. fst(?e'7) !> ?p : P
val it = () : unit
-   br impCE 1;
Level 8
?p : P
 1. lam x. x >> ?e1'8 : ?Q8 --> P & ?Q7
 2. a >> ?e2'8 : ?Q8
 3. ?e1'8 ‘ ?e2'8 !> ?e'7 : P & ?Q7
 4. fst(?e'7) !> ?p : P
val it = () : unit
-   bta 1;
Level 9
?p : P
 1. a >> ?e2'8 : ?Q8
 2. (lam x. x) ‘ ?e2'8 !> ?e'7 : P & ?Q7
 3. fst(?e'7) !> ?p : P
val it = () : unit
-   bta 1;
Level 10
?p : P
 1. (lam x. x) ‘ a !> ?e'7 : P & ?Q7
 2. fst(?e'7) !> ?p : P
val it = () : unit
-   br impR 1;
Level 11
?p : P
 1. a >> ?e'7 : P & ?Q7
 2. fst(?e'7) !> ?p : P
val it = () : unit
-   bta 1;
Level 12
?p : P
 1. fst(a) !> ?p : P
val it = () : unit
-   bt 1;
Level 13
fst(a) : P
No subgoals!
val it = () : unit
- val destRedex = result();
val destRedex =
  "?a : ?P & ?Q ==> fst(?a) : ?P" : thm
```

Reduction of a proof with a redex as argument of a constructor corresponding to 107_{224} is demonstrated with the following proving session.

```
-   goal PE.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 2
?p : ?P2 --> ?Q2
 1. !!x. x : ?P2 ==> ?f2(x) : ?Q2
 2. lam x. ?f2(x) >> ?p : ?P2 --> ?Q2
val it = () : unit
-   br mp 1;
Level 3
?p : ?P2 --> ?Q2
 1. !!x. x : ?P2 ==> ?f3(x) : ?P3(x) --> ?Q2
 2. !!x. x : ?P2 ==> ?a3(x) : ?P3(x)
 3. lam x. ?f3(x) ` ?a3(x) >> ?p : ?P2 --> ?Q2
val it = () : unit
-   br impI 1;
Level 4
?p : ?P2 --> ?Q2
 1. !!x xa.
       [| x : ?P2; xa : ?P3(x) |] ==>
       ?f4(x, xa) : ?Q2
 2. !!x. x : ?P2 ==> ?a3(x) : ?P3(x)
 3. lam x. (lam xa. ?f4(x, xa)) ` ?a3(x) >> ?p :
       ?P2 --> ?Q2
val it = () : unit
-   ba 1;
Level 5
?p : ?P2 --> ?P2
 1. !!x. x : ?P2 ==> ?a3(x) : ?P3(x)
 2. lam x. (lam xa. x) ` ?a3(x) >> ?p :
       ?P2 --> ?P2
val it = () : unit
-   back();
Level 5
?p : ?P2 --> ?Q2
 1. !!x. x : ?P2 ==> ?a3(x) : ?Q2
 2. lam x. (lam x. x) ` ?a3(x) >> ?p :
       ?P2 --> ?Q2
val it = () : unit
-   ba 1;
Level 6
?p : ?P2 --> ?P2
 1. lam x. (lam x. x) ` x >> ?p : ?P2 --> ?P2
val it = () : unit
-   bta 1;
Level 7
lam x. (lam x. x) ` x : ?P2 --> ?P2
No subgoals!
val it = () : unit
- val constredex = result();
val constredex =
  "lam x. (lam x. x) ` x : ?P --> ?P" : thm
```

Reduction of a destructor forming a redex after but not before reduction of the destructor's arguments corresponding to 109_{224} is demonstrated with the following proving session.

```
- val [p] = goal PE.thy "a : P ==> ?p : ?P";
std_in:127.1-127.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P  [a : P]" : thm
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 2
?p : ?P
 1. ?f2 : ?P2 --> ?P
 2. ?a2 : ?P2
 3. ?f2 ` ?a2 >> ?p : ?P
val it = () : unit
-   br conjunct1 1;
Level 3
?p : ?P
 1. ?p3 : (?P2 --> ?P) & ?Q3
 2. ?a2 : ?P2
 3. fst(?p3) ` ?a2 >> ?p : ?P
val it = () : unit
-   br mp 1;
Level 4
?p : ?P
 1. ?f4 : ?P4 --> (?P2 --> ?P) & ?Q3
 2. ?a4 : ?P4
 3. ?a2 : ?P2
 4. fst(?f4 ` ?a4) ` ?a2 >> ?p : ?P
val it = () : unit
-   br impI 1;
Level 5
?p : ?P
 1. !!x. x : ?P4 ==> ?f5(x) : (?P2 --> ?P) & ?Q3
 2. ?a4 : ?P4
 3. ?a2 : ?P2
 4. fst((lam x. ?f5(x)) ` ?a4) ` ?a2 >> ?p : ?P
val it = () : unit
-   br conjI 1;
Level 6
?p : ?P
 1. !!x. x : ?P4 ==> ?a6(x) : ?P2 --> ?P
 2. !!x. x : ?P4 ==> ?b6(x) : ?Q3
 3. ?a4 : ?P4
 4. ?a2 : ?P2
 5. fst((lam x. <?a6(x),?b6(x)>) ` ?a4) ` ?a2
       >> ?p : ?P
val it = () : unit
-   ba 1;
Level 7
?p : ?P8
 1. !!x. x : ?P7 --> ?P8 ==> ?b6(x) : ?Q3
 2. ?a4 : ?P7 --> ?P8
 3. ?a2 : ?P7
```

```
 4. fst((lam x. <x,?b6(x)>) ‘ ?a4) ‘ ?a2 >> ?p :
       ?P8
val it = () : unit
-   ba 1;
Level 8
?p : ?P10
 1. ?a4 : ?P9 --> ?P10
 2. ?a2 : ?P9
 3. fst((lam x. <x,x>) ‘ ?a4) ‘ ?a2 >> ?p : ?P10
val it = () : unit
-   br impI 1;
Level 9
?p : ?P10
 1. !!x. x : ?P9 ==> ?f11(x) : ?P10
 2. ?a2 : ?P9
 3. fst((lam x. <x,x>) ‘ (lam x. ?f11(x))) ‘
    ?a2
       >> ?p : ?P10
val it = () : unit
-   ba 1;
Level 10
?p : ?P9
 1. ?a2 : ?P9
 2. fst((lam x. <x,x>) ‘ (lam x. x)) ‘ ?a2
       >> ?p : ?P9
val it = () : unit
-   br p 1;
Level 11
?p : P
 1. fst((lam x. <x,x>) ‘ (lam x. x)) ‘ a >> ?p :
       P
val it = () : unit
-   br impCE 1;
Level 12
?p : P
 1. fst((lam x. <x,x>) ‘ (lam x. x)) >> ?e1’12 :
       ?Q12 --> P
 2. a >> ?e2’12 : ?Q12
 3. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br conjCE1 1;
Level 13
?p : P
 1. (lam x. <x,x>) ‘ (lam x. x) >> ?e’13 :
       (?Q12 --> P) & ?Q13
 2. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 3. a >> ?e2’12 : ?Q12
 4. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br impCE 1;
Level 14
?p : P
 1. lam x. <x,x> >> ?e1’14 :
       ?Q14 --> (?Q12 --> P) & ?Q13
 2. lam x. x >> ?e2’14 : ?Q14
 3. ?e1’14 ‘ ?e2’14 !> ?e’13 :
       (?Q12 --> P) & ?Q13
 4. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 5. a >> ?e2’12 : ?Q12
 6. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bta 1;
Level 15
?p : P
 1. lam x. x >> ?e2’14 : ?Q14
 2. (lam x. <x,x>) ‘ ?e2’14 !> ?e’13 :
       (?Q12 --> P) & ?Q13
 3. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 4. a >> ?e2’12 : ?Q12
 5. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bta 1;
Level 16
?p : P
 1. (lam x. <x,x>) ‘ (lam x. x) !> ?e’13 :
       (?Q12 --> P) & ?Q13
 2. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 3. a >> ?e2’12 : ?Q12
 4. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br impR 1;
Level 17
?p : P
 1. <lam x. x,lam x. x> >> ?e’13 :
       (?Q12 --> P) & ?Q13
 2. fst(?e’13) !> ?e1’12 : ?Q12 --> P
 3. a >> ?e2’12 : ?Q12
 4. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bta 1;
Level 18
?p : P
 1. fst(<lam x. x,lam x. x>) !> ?e1’12 :
       ?Q12 --> P
 2. a >> ?e2’12 : ?Q12
 3. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   br conjR1 1;
Level 19
?p : P
 1. lam x. x >> ?e1’12 : ?Q12 --> P
 2. a >> ?e2’12 : ?Q12
 3. ?e1’12 ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bta 1;
Level 20
?p : P
 1. a >> ?e2’12 : ?Q12
 2. (lam x. x) ‘ ?e2’12 !> ?p : P
val it = () : unit
-   bta 1;
Level 21
?p : P
 1. (lam x. x) ‘ a !> ?p : P
val it = () : unit
-   br impR 1;
Level 22
?p : P
 1. a >> ?p : P
val it = () : unit
-   bta 1;
Level 23
a : P
No subgoals!
val it = () : unit
- val redexAfter = result();
val redexAfter = "?a : ?P ==> ?a : ?P" : thm
```

Since tactic `try_allRules` has been demonstrated and it is demonstrated that object logic `PE`

acts like specified in 105_{224}–111_{224} then it should be possible to conclude that there is a certain possibility that it is possible to evaluate every proof provable in object logic `FirstOrderProof` as a program with object logic `PE`.

B.5 Natural Numbers

An object logic for natural numbers based on the Peano axioms is implemented in Isabelle as object logic `Nat`. This object logic will be able to implement formal theory $\mathcal{NAT}$ with a few changes.

The implementation of natural numbers will also include a rewrite rule defining the + operator on natural numbers. Such a rewriting rule does not extend the number of theorems provable by the formal theory, rewrite rules only helps specifying theorems.

B.5.1 Implementing Theory

The object logic for natural numbers can be borrowed from the Isabelle implementation, and this will work as implementation of $\mathcal{NAT}$ with the following changes:

- Function name S is used instead of the original successor function name.
- The induction rule is implemented in another way.
- The base logic will be `FirstOrder` implementing $\mathcal{IFOL}$.

The object logic representing formal theory $\mathcal{NAT}$ can be found in appendix A.10.

B.5.2 Examples

Some examples may show how the object logic can be used. Each rule shall be used at least once in the examples and the examples may show both some general examples and some specific application of a theorem on a specific natural number.

An example showing the use of the induct rule proving a more safe induction rule

$$\frac{P(0) \qquad \forall x.P(x) \rightarrow P(\mathsf{S}(x))}{\forall n.P(n)}\ \text{induction}$$

demonstrating rule induct is the following

```
- val [hb,hi] = goal Nat.thy
=   "[| P(0); ALL x. P(x) --> P(S(x)) |] ==>
      ALL n. P(n)";
std_in:5.1-6.55 Warning: binding not exhaustive
          hb :: hi :: nil = ...
Level 0
ALL n. P(n)
 1. ALL n. P(n)
val hb = "P(0)  [P(0)]" : thm
val hi = "ALL x. P(x) --> P(S(x))
          [ALL x. P(x) --> P(S(x))]" : thm
-     br induct 1;
Level 1
ALL n. P(n)
 1. P(0)
 2. !!x. P(x) ==> P(S(x))
val it = () : unit
-     br hb 1;
Level 2
ALL n. P(n)
 1. !!x. P(x) ==> P(S(x))
val it = () : unit
-     br mp 1;
Level 3
```

```
ALL n. P(n)
 1. !!x. P(x) ==> ?P1(x) --> P(S(x))
 2. !!x. P(x) ==> ?P1(x)
val it = () : unit
-     br allE 1;
Level 4
ALL n. P(n)
 1. !!x. P(x) ==> ALL xa. ?P2(x, xa)
 2. !!x. [| P(x); ?P2(x, ?x2(x)) |] ==>
         ?P1(x) --> P(S(x))
 3. !!x. P(x) ==> ?P1(x)
val it = () : unit
-     br hi 1;
Level 5
ALL n. P(n)
 1. !!x. [| P(x);
            P(?x2(x)) --> P(S(?x2(x))) |] ==>
         ?P1(x) --> P(S(x))
 2. !!x. P(x) ==> ?P1(x)
val it = () : unit
-     ba 1;
Level 6
ALL n. P(n)
 1. !!x. P(x) ==> P(x)
val it = () : unit
-     ba 1;
Level 7
ALL n. P(n)
No subgoals!
val it = () : unit
- val induction = result();
val induction =
  "[| ?P(0); ALL x. ?P(x) --> ?P(S(x)) |] ==>
   ALL n. ?P(n)" : thm
```

An example proving the inverse of inference rule S-inject being

$$\frac{m = n}{\mathsf{S}(m) = \mathsf{S}(n)}\ \text{S-surj}$$

is

```
- val [h] =
= goal Nat.thy "m = n ==> S(m) = S(n)";
std_in:21.1-21.46 Warning:
  binding not exhaustive
          h :: nil = ...
Level 0
S(m) = S(n)
 1. S(m) = S(n)
val h = "m = n  [m = n]" : thm
-   by ((rtac subst_context 1) THEN (rtac h 1));
Level 1
S(m) = S(n)
No subgoals!
val it = () : unit
- val S_surj = result();
val S_surj = "?m = ?n ==> S(?m) = S(?n)" : thm
```

An example proving theorem $(\mathsf{S}(k) = k) \rightarrow \bot$ demonstrating inference rules $\mathsf{S} \neq 0$ and S-inject is

```
-   goal Nat.thy "(S(k) = k) --> False";
Level 0
S(k) = k --> False
 1. S(k) = k --> False
val it = [] : thm list
-   br allE 1;
Level 1
S(k) = k --> False
 1. ALL x. ?P(x)
 2. ?P(?x) ==> S(k) = k --> False
val it = () : unit
-   ba 2;
Level 2
S(k) = k --> False
 1. ALL x. S(x) = x --> False
val it = () : unit
-   br induction 1;
Level 3
S(k) = k --> False
 1. S(0) = 0 --> False
 2. ALL x.
       (S(x) = x --> False) -->
       S(S(x)) = S(x) --> False
val it = () : unit
-   br impI 1;
Level 4
S(k) = k --> False
 1. S(0) = 0 ==> False
 2. ALL x.
       (S(x) = x --> False) -->
       S(S(x)) = S(x) --> False
val it = () : unit
-   br S_neq_0 1;
Level 5
S(k) = k --> False
 1. S(0) = 0 ==> S(?m3) = 0
 2. ALL x.
       (S(x) = x --> False) -->
       S(S(x)) = S(x) --> False
val it = () : unit
-   ba 1;
Level 6
S(k) = k --> False
 1. ALL x.
       (S(x) = x --> False) -->
       S(S(x)) = S(x) --> False
val it = () : unit
-   br allI 1;
Level 7
S(k) = k --> False
 1. !!x. (S(x) = x --> False) -->
         S(S(x)) = S(x) --> False
```

```
val it = () : unit
-   br impI 1;
Level 8
S(k) = k --> False
 1. !!x. S(x) = x --> False ==>
         S(S(x)) = S(x) --> False
val it = () : unit
-   br impI 1;
Level 9
S(k) = k --> False
 1. !!x. [| S(x) = x --> False;
            S(S(x)) = S(x) |] ==>
         False
val it = () : unit
-   br mp 1;
Level 10
S(k) = k --> False
 1. !!x. [| S(x) = x --> False;
            S(S(x)) = S(x) |] ==>
         ?P7(x) --> False
 2. !!x. [| S(x) = x --> False;
            S(S(x)) = S(x) |] ==>
         ?P7(x)
```

```
val it = () : unit
-   ba 1;
Level 11
S(k) = k --> False
 1. !!x. [| S(x) = x --> False;
            S(S(x)) = S(x) |] ==>
         S(x) = x
val it = () : unit
-   br S_inject 1;
Level 12
S(k) = k --> False
 1. !!x. [| S(x) = x --> False;
            S(S(x)) = S(x) |] ==>
         S(S(x)) = S(x)
val it = () : unit
-   ba 1;
Level 13
S(k) = k --> False
No subgoals!
val it = () : unit
- val S_n_not_n = result();
val S_n_not_n = "S(?k) = ?k --> False" : thm
```

Proving the theorem $0 + n = n$ which without the definition of "+" is the theorem

$$\mathsf{rec}_{n,\Pi x\ y.\ \mathsf{S}(y)}(0) = n$$

demonstrating rule rec_0 is done as follows

```
-   goal Nat.thy "0+n = n";
Level 0
0 + n = n
 1. 0 + n = n
val it = [] : thm list
-   by (rewtac add_def);
Level 1
0 + n = n
 1. rec(0, n, %x y. S(y)) = n
val it = () : unit
-   by (resolve_tac [rec_0] 1);
Level 2
0 + n = n
No subgoals!
val it = () : unit
- val add_0' = result();
val add_0' = "0 + ?n = ?n" : thm
```

```
-   goal Nat.thy "?P";
Level 0
?P
 1. ?P
val it = [] : thm list
-   br add_0' 1;
Level 1
0 + ?n1 = ?n1
No subgoals!
val it = () : unit
-   by (rewrite_tac [add_def]);
Level 2
rec(0, ?n1, %x y. S(y)) = ?n1
No subgoals!
val it = () : unit
- val add_0 = result();
val add_0 = "rec(0, ?n, %x y. S(y)) = ?n" : thm
```

Notice that the proof is in two parts – if `rewrite_tac` is used on a state where the goal is fully specified, the theorem proven is no longer the same as the proof stated.

Proving the theorem $\mathsf{S}(m) + n = \mathsf{S}(m + n)$ which without the definition of "+" is the theorem $\mathsf{rec}_{n,\Pi xy.\mathsf{S}(y)}(\mathsf{S}(m)) = \mathsf{S}(\mathsf{rec}_{n,\Pi xy.\mathsf{S}(y)}(m))$ demonstrating the rule rec_S is done as follows

```
-   goal Nat.thy "S(m)+n = S(m+n)";
Level 0
S(m) + n = S(m + n)
 1. S(m) + n = S(m + n)
val it = [] : thm list
-   by (rewtac add_def);
Level 1
S(m) + n = S(m + n)
```

```
 1. rec(S(m), n, %x y. S(y)) =
    S(rec(m, n, %x y. S(y)))
val it = () : unit
-   by (resolve_tac [rec_S] 1);
Level 2
S(m) + n = S(m + n)
No subgoals!
val it = () : unit
```

```
- val add_S' = result();
val add_S' = "S(?m) + ?n = S(?m + ?n)" : thm
-   goal Nat.thy "?P";
Level 0
?P
 1. ?P
val it = [] : thm list
-   br add_S' 1;
Level 1
S(?m1) + ?n1 = S(?m1 + ?n1)
No subgoals!
val it = () : unit
-   by (rewrite_tac [add_def]);
Level 2
rec(S(?m1), ?n1, %x y. S(y)) =
S(rec(?m1, ?n1, %x y. S(y)))
No subgoals!
val it = () : unit
- val add_S = result();
val add_S =
  "rec(S(?m), ?n, %x y. S(y)) =
   S(rec(?m, ?n, %x y. S(y)))" : thm
```

Hereby all rules have been demonstrated. To demonstrate application of a theorem on a decent value the theorem $(\mathsf{S}(\mathsf{S}(0)) = \mathsf{S}(0)) \rightarrow \bot$ is proven with the proving session

```
-   goal Nat.thy "(S(S(0)) = S(0)) --> False";
Level 0
S(S(0)) = S(0) --> False
 1. S(S(0)) = S(0) --> False
val it = [] : thm list
-   br S_n_not_n 1;
Level 1
S(S(0)) = S(0) --> False
No subgoals!
val it = () : unit
- val two_not_one = result();
val two_not_one =
  "S(S(0)) = S(0) --> False" : thm
```

B.6 Natural Numbers with Proofs

An object logic for natural numbers with proofs based on the Peano axioms is implemented in the Isabelle `FOLP` parts as object logic `Nat`. This object logic will be able to implement formal theory $\mathcal{NATP}$ with a few changes.

The implementation of natural numbers with proofs will also include a rewrite rule defining the + operator on natural numbers. Such a rewriting rule does not extend the number of theorems provable by the formal theory, rewrite rules only helps specifying theorems.

B.6.1 Implementing Theory

The object logic for natural numbers with proofs can be borrowed from the Isabelle implementation, and this will work as implementation of $\mathcal{NATP}$ with the following changes:

- Function name `S` is used instead of the original successor function name.
- The induction rule is implemented in another way.
- The base logic will be `FirstOrderProof` implementing $\mathcal{IFOLP}$.
- The implemented object logic `Nat` of the Isabelle distribution implements some reduction rules. These rules must not be a part of the object logic representing $\mathcal{NATP}$ since reduction is seperated from proof construction in this work.

Then the object logic will follow the logic presented in section 5.3.2. This object logic is though not satisfacting because that a rule corresponding to rule Induction of the examples in section B.5.2 derived in this object logic will contain redexes what may be unpleasant. If a rule Induction given by

$$\frac{b : P(0) \qquad f : \forall x.P(x) \rightarrow P(\mathsf{S}(x))}{\mathtt{nrec}(b, f) : \forall n.P(n)}\ \mathsf{induction}$$

is specified as a part of the object logic then the rule induct as specified in $\mathcal{NATP}$ can be derived hereof.

The object logic representing formal theory $\mathcal{NATP}$ can be found in appendix A.11.

B.6.2 Examples

Some examples may show how the object logic can be used. Each rule shall be used at least once in the examples and the examples may show both some general examples and some specific applications of a theorem on a specific natural number.

The rule induct of the formal theory $\mathcal{NATP}$, that is not present in the object logic, is derived as

```
- val [hb,hi] =  goal NATP.thy
=   "[| b : P(0); \
=   \   !!x xa. xa : P(x) ==> \
=   \           f(x, xa) : P(S(x)) |] ==> \
=   \ nrec(b, all x. lam xa. f(x, xa)) : \
=   \ ALL n. P(n)";
std_in:5.1-7.52 Warning: binding not exhaustive
          hb :: hi :: nil = ...
Level 0
nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)
 1. nrec(b, all x. lam xa. f(x, xa))
    : ALL n. P(n)
val hb = "b : P(0)  [b : P(0)]" : thm
val hi =
  "?xa : P(?x) ==> f(?x, ?xa) : P(S(?x))
     [!!x xa. xa : P(x) ==>
              f(x, xa) : P(S(x))]" : thm
-     br induction 1;
Level 1
nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)
 1. b : P(0)
 2. all x. lam xa. f(x, xa)
    : ALL x. P(x) --> P(S(x))
val it = () : unit
-     br hb 1;
Level 2
nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)
 1. all x. lam xa. f(x, xa)
    : ALL x. P(x) --> P(S(x))
val it = () : unit
-     br allI 1;
Level 3
nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)
 1. !!x. lam xa. f(x, xa) : P(x) --> P(S(x))
val it = () : unit
-     br impI 1;
Level 4
nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)
 1. !!x xa. xa : P(x) ==> f(x, xa) : P(S(x))
val it = () : unit
-     br hi 1;
Level 5
nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)
 1. !!x xa. xa : P(x) ==> xa : P(x)
val it = () : unit
-     ba 1;
Level 6
nrec(b, all x. lam xa. f(x, xa)) : ALL n. P(n)
No subgoals!
val it = () : unit
- val induct = result();
val induct =
  "[| ?b : ?P(0);
      !!x xa. xa : ?P(x) ==>
              ?f(x, xa) : ?P(S(x)) |] ==>
   nrec(?b, all x. lam xa. ?f(x, xa)) :
   ALL n. ?P(n)" : thm
```

An example proving the inverse of inference rule S-inject being

$$\frac{\mathtt{p} : m = n}{\mathtt{idpeel(p,\ \%x.\ ideq(S(x)))} : \mathsf{S}(m) = \mathsf{S}(n)}\ \text{S-surj}$$

is

```
- val [h] = goal NATP.thy
= "r : m = n ==> ?p : S(m) = S(n)";
std_in:19.1-19.56 Warning:
  binding not exhaustive
          h :: nil = ...
Level 0
?p : S(m) = S(n)
 1. ?p : S(m) = S(n)
val h = "r : m = n  [r : m = n]" : thm
-   by ((rtac subst_context 1) THEN (rtac h 1));
Level 1
idpeel(r, %x. ideq(S(x))) : S(m) = S(n)
No subgoals!
val it = () : unit
- val S_surj = result();
val S_surj =
 "?r : ?m = ?n ==>
  idpeel(?r, %x. ideq(S(x))) : S(?m) = S(?n)"
  : thm
```

An example proving theorem

$$\texttt{nrec(lam x. nneq(x), all x. lam x xa. x ` ninj(xa))} \mathbin{\hat{}} \texttt{k} : (\mathsf{S}(\texttt{k}) = \texttt{k}) \to \bot$$

demonstrating inference rules $\mathsf{S} \neq 0$ and S-inject is

```
-   goal NATP.thy "?p : (S(k) = k) --> False";
Level 0
?p : S(k) = k --> False
 1. ?p : S(k) = k --> False
val it = [] : thm list
-   br allE 1;
Level 1
?p1 ^ ?z1 : S(k) = k --> False
 1. ?p1 : ALL x. ?P1(x)
 2. !!y. y : ?P1(?z1) ==> y : S(k) = k --> False
val it = () : unit
-   ba 2;
Level 2
?p1 ^ k : S(k) = k --> False
 1. ?p1 : ALL x. S(x) = x --> False
val it = () : unit
-   br induction 1;
Level 3
nrec(?b2, ?f2) ^ k : S(k) = k --> False
 1. ?b2 : S(0) = 0 --> False
 2. ?f2
    : ALL x.
         (S(x) = x --> False) -->
         S(S(x)) = S(x) --> False
val it = () : unit
-   br impI 1;
Level 4
nrec(lam x. ?f3(x), ?f2) ^ k
: S(k) = k --> False
 1. !!x. x : S(0) = 0 ==> ?f3(x) : False
 2. ?f2
    : ALL x.
         (S(x) = x --> False) -->
         S(S(x)) = S(x) --> False
val it = () : unit
-   br S_neq_0 1;
Level 5
nrec(lam x. nneq(?p4(x)), ?f2) ^ k
: S(k) = k --> False
 1. !!x. x : S(0) = 0 ==> ?p4(x) : S(?m4(x)) = 0
 2. ?f2
    : ALL x.
         (S(x) = x --> False) -->
         S(S(x)) = S(x) --> False
val it = () : unit
-   ba 1;
Level 6
nrec(lam x. nneq(x), ?f2) ^ k
: S(k) = k --> False
 1. ?f2
    : ALL x.
         (S(x) = x --> False) -->
         S(S(x)) = S(x) --> False
val it = () : unit
-   br allI 1;
Level 7
nrec(lam x. nneq(x), all x. ?f5(x)) ^ k
: S(k) = k --> False
 1. !!x. ?f5(x)
         : (S(x) = x --> False) -->
           S(S(x)) = S(x) --> False
val it = () : unit
-   br impI 1;
Level 8
nrec
 (lam x. nneq(x), all x. lam xa. ?f6(x, xa)) ^
k
: S(k) = k --> False
 1. !!x xa.
       xa : S(x) = x --> False ==>
       ?f6(x, xa) : S(S(x)) = S(x) --> False
val it = () : unit
-   br impI 1;
Level 9
nrec
 (lam x. nneq(x),
  all x. lam xa xb. ?f7(x, xa, xb)) ^
k
: S(k) = k --> False
 1. !!x xa xb.
       [| xa : S(x) = x --> False;
          xb : S(S(x)) = S(x) |] ==>
       ?f7(x, xa, xb) : False
val it = () : unit
-   br mp 1;
Level 10
nrec
 (lam x. nneq(x),
  all x.
     lam xa xb.
        ?f8(x, xa, xb) ` ?a8(x, xa, xb)) ^
k
: S(k) = k --> False
 1. !!x xa xb.
       [| xa : S(x) = x --> False;
          xb : S(S(x)) = S(x) |] ==>
       ?f8(x, xa, xb) : ?P8(x, xa, xb) --> False
 2. !!x xa xb.
       [| xa : S(x) = x --> False;
          xb : S(S(x)) = S(x) |] ==>
       ?a8(x, xa, xb) : ?P8(x, xa, xb)
val it = () : unit
-   ba 1;
Level 11
nrec
 (lam x. nneq(x),
  all x. lam xa xb. xa ` ?a8(x, xa, xb)) ^
k
: S(k) = k --> False
 1. !!x xa xb.
       [| xa : S(x) = x --> False;
          xb : S(S(x)) = S(x) |] ==>
       ?a8(x, xa, xb) : S(x) = x
val it = () : unit
-   br S_inject 1;
Level 12
nrec
 (lam x. nneq(x),
  all x. lam xa xb. xa ` ninj(?p9(x, xa, xb))) ^
k
```

```
: S(k) = k --> False
 1. !!x xa xb.
       [| xa : S(x) = x --> False;
          xb : S(S(x)) = S(x) |] ==>
       ?p9(x, xa, xb) : S(S(x)) = S(x)
val it = () : unit
-   ba 1;
Level 13
nrec
 (lam x. nneq(x),
  all x. lam x xa. x ‘ ninj(xa)) ^
k
: S(k) = k --> False
No subgoals!
val it = () : unit
- val S_n_not_n = result();
val S_n_not_n =
  "nrec(lam x. nneq(x),
        all x. lam x xa. x ‘ ninj(xa)) ^ ?k
   : S(?k) = ?k --> False" : thm
```

Proving the theorem `rec0` : $0 + n = n$ which without the definition of "+" is the theorem

$$\texttt{rec0} : \mathsf{rec}_{n,\Pi x\ y.\ \mathsf{S}(y)}(0) = n$$

demonstrating rule rec_0 is done as follows

```
-   goal NATP.thy "?p : 0+n = n";
Level 0
?p : 0 + n = n
 1. ?p : 0 + n = n
val it = [] : thm list
-   by (rewtac add_def);
Level 1
?p : 0 + n = n
 1. ?p : rec(0, n, %x y. S(y)) = n
val it = () : unit
-   by (resolve_tac [rec_0] 1);
Level 2
rec0 : 0 + n = n
No subgoals!
val it = () : unit
- val add_0' = result();
val add_0' = "rec0 : 0 + ?n = ?n" : thm
-   goal NATP.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br add_0' 1;
Level 1
rec0 : 0 + ?n1 = ?n1
No subgoals!
val it = () : unit
-   by (rewrite_tac [add_def]);
Level 2
rec0 : rec(0, ?n1, %x y. S(y)) = ?n1
No subgoals!
val it = () : unit
- val add_0 = result();
val add_0 =
  "rec0 : rec(0, ?n, %x y. S(y)) = ?n" : thm
```

Notice that the proof is in two parts – if `rewrite_tac` is used on a state where the goal is fully specified, the theorem proven is no longer the same as the proof stated.

Proving the theorem `recS` : $\mathsf{S}(m) + n = \mathsf{S}(m + n)$ which without the definition of "+" is the theorem `recS` : $\mathsf{rec}_{n,\Pi xy.\mathsf{S}(y)}(\mathsf{S}(m)) = \mathsf{S}(\mathsf{rec}_{n,\Pi xy.\mathsf{S}(y)}(m))$ demonstrating the rule rec_S is done as follows

```
-   goal NATP.thy "?p : S(m)+n = S(m+n)";
Level 0
?p : S(m) + n = S(m + n)
 1. ?p : S(m) + n = S(m + n)
val it = [] : thm list
-   by (rewtac add_def);
Level 1
?p : S(m) + n = S(m + n)
 1. ?p
    : rec(S(m), n, %x y. S(y)) =
      S(rec(m, n, %x y. S(y)))
val it = () : unit
-   by (resolve_tac [rec_S] 1);
Level 2
recS : S(m) + n = S(m + n)
No subgoals!
val it = () : unit
- val add_S' = result();
val add_S' =
  "recS : S(?m) + ?n = S(?m + ?n)" : thm
-   goal NATP.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br add_S' 1;
Level 1
recS : S(?m1) + ?n1 = S(?m1 + ?n1)
No subgoals!
val it = () : unit
-   by (rewrite_tac [add_def]);
Level 2
recS
: rec(S(?m1), ?n1, %x y. S(y)) =
  S(rec(?m1, ?n1, %x y. S(y)))
No subgoals!
val it = () : unit
- val add_S = result();
```

```
val add_S =                                              S(rec(?m, ?n, %x y. S(y)))"
  "recS : rec(S(?m), ?n, %x y. S(y)) =                : thm
```

Hereby all rules have been demonstrated. To demonstrate application of a theorem on a decent value a theorem with proposition part $(\mathsf{S}(\mathsf{S}(\mathsf{0})) = \mathsf{S}(\mathsf{0})) \to \bot$ is proven with

```
-   goal NATP.thy                                        all x. lam x xa. x ‘ ninj(xa)) ^
= "?p : (S(S(0)) = S(0)) --> False";                    S(0)
Level 0                                                 : S(S(0)) = S(0) --> False
?p : S(S(0)) = S(0) --> False                           No subgoals!
 1. ?p : S(S(0)) = S(0) --> False                       val it = () : unit
val it = [] : thm list                                  - val two_not_one = result();
-   br S_n_not_n 1;                                     val two_not_one =
Level 1                                                   "nrec(lam x. nneq(x),
nrec                                                            all x. lam x xa. x ‘ ninj(xa)) ^ S(0)
 (lam x. nneq(x),                                          : S(S(0)) = S(0) --> False" : thm
```

Notice that the above example demonstrates exactly the proving method described in section 4.1.1 that gives redundant proofs. As demonstrated in section B.7.2 the proof for a theorem having proposition part $(\mathsf{S}(\mathsf{S}(\mathsf{0})) = \mathsf{S}(\mathsf{0})) \to \bot$ can be as small as `lam x. nneq(ninj(x))`.

B.7 Natural Numbers with Proofs and Proof Reduction

An object logic for reducing proofs of propositions including natural numbers shall be created as object logic `PRNAT` corresponding to formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ defined in section 5.3.2.

B.7.1 Implementing Theory

Formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ specifies that proof reduction shall be reduction of proofs that are proven using formal theory $\mathcal{NATP}$ with the reduction methods specified in formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$. Therefore object logic `PRNAT` must be built on top of object logics `PR` and `NATP`.

Since there shall be two object logics implementing normalization of proofs including natural numbers the common terms of these two object logics is placed in object logic `NATReduction` presented in section A.12.

Object logic `PRNAT` for proof reduction of proofs including natural numbers implements the parts of formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ that is not shared with formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$. Object logic `PRNAT` is in appendix A.13.

B.7.2 Examples

Some examples may show how the object logic can be used. Each rule must be used at least ince in the examples and the examples must show both some general examples and reduction of some specific application of a theorem on a specific natural number.

An example demonstrating context rule $\mathsf{rec}_0\mathsf{Cl}$ is

```
-   goal PRNAT.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br add_0 1;
Level 2
?p : rec(0, ?n2, %x y. S(y)) = ?n2
 1. rec0 >> ?p : rec(0, ?n2, %x y. S(y)) = ?n2
val it = () : unit
-   br rec_0CI 1;
Level 3
rec0 : rec(0, ?n2, %x y. S(y)) = ?n2
No subgoals!
val it = () : unit
- val add_0Red = result();
val add_0Red =
  "rec0 : rec(0, ?n, %x y. S(y)) = ?n" : thm
```

An example demonstrating context rule $\mathsf{rec_S CI}$ is

```
-   goal PRNAT.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br add_S 1;
Level 2
?p
: rec(S(?m2), ?n2, %x y. S(y)) =
  S(rec(?m2, ?n2, %x y. S(y)))
 1. recS >> ?p :
       rec(S(?m2), ?n2, %x y. S(y)) =
       S(rec(?m2, ?n2, %x y. S(y)))
val it = () : unit
-   br rec_SCI 1;
Level 3
recS
: rec(S(?m2), ?n2, %x y. S(y)) =
  S(rec(?m2, ?n2, %x y. S(y)))
No subgoals!
val it = () : unit
- val add_SRed = result();
val add_SRed =
  "recS : rec(S(?m), ?n, %x y. S(y)) =
          S(rec(?m, ?n, %x y. S(y)))"
  : thm
```

An example demonstrating context rules $\mathsf{induction CI}$, $\mathsf{S} \neq \mathsf{0 CI}$ and $\mathsf{S\text{-}inject CI}$ followed by reduction rules $\mathsf{induction R_S}$ and $\mathsf{induction R_0}$ is reduction of the proof of proposition $(\mathsf{S}(\mathsf{S}(0)) = \mathsf{S}(0)) \to \bot$. This reduction is shown by the following example below. The example is the only example in full of a reduction of an application of a term since the proving session prints gets enormous.

```
-   goal PRNAT.thy
= "?p : (S(S(0)) = S(0)) --> False";
Level 0
?p : S(S(0)) = S(0) --> False
 1. ?p : S(S(0)) = S(0) --> False
val it = [] : thm list
-   br redE 1;
Level 1
?p : S(S(0)) = S(0) --> False
 1. ?a1 : S(S(0)) = S(0) --> False
 2. ?a1 >> ?p : S(S(0)) = S(0) --> False
val it = () : unit
-   br S_n_not_n 1;
Level 2
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    S(0)
       >> ?p : S(S(0)) = S(0) --> False
val it = () : unit
-   br allCE 1;
Level 3
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa))
       >> ?g3 : ALL b. ?R3(b)
 2. ?g3 ^ S(0) !> ?p : ?R3(S(0))
 3. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br inductionCI 1;
Level 4
?p : S(S(0)) = S(0) --> False
 1. lam x. nneq(x) >> ?b'4 : ?R3(0)
 2. all x. lam x xa. x ` ninj(xa) >> ?f'4 :
       ALL x. ?R3(x) --> ?R3(S(x))
 3. nrec(?b'4, ?f'4) ^ S(0) !> ?p : ?R3(S(0))
 4. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br impCI 1;
Level 5
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?P7(0) ==>
         nneq(y) >> ?g5(y) : ?Q6(0)
 2. all x. lam x xa. x ` ninj(xa) >> ?f'4 :
       ALL x.
          (?P7(x) --> ?Q6(x)) -->
          ?P7(S(x)) --> ?Q6(S(x))
```

```
 3. nrec(lam x. ?g5(x), ?f'4) ^ S(0) !> ?p :
       ?P7(S(0)) --> ?Q6(S(0))
 4. !!y. y : ?P7(S(0)) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br S_neq_0CI 1;
Level 6
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?P7(0) ==>
         y >> ?p'8(y) : S(?m8(y)) = 0
 2. all x. lam x xa. x ` ninj(xa) >> ?f'4 :
       ALL x.
          (?P7(x) --> ?Q6(x)) -->
          ?P7(S(x)) --> ?Q6(S(x))
 3. nrec(lam x. nneq(?p'8(x)), ?f'4) ^ S(0)
       !> ?p : ?P7(S(0)) --> ?Q6(S(0))
 4. !!y. y : ?P7(S(0)) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   bv 1;
Level 7
?p : S(S(0)) = S(0) --> False
 1. all x. lam x xa. x ` ninj(xa) >> ?f'4 :
       ALL x.
          (S(?m11(x)) = x --> ?Q6(x)) -->
          S(?m11(S(x))) = S(x) --> ?Q6(S(x))
 2. nrec(lam x. nneq(x), ?f'4) ^ S(0) !> ?p :
       S(?m11(S(0))) = S(0) --> ?Q6(S(0))
 3. !!y. y : S(?m11(S(0))) = S(0) -->
             ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br allCI 1;
Level 8
?p : S(S(0)) = S(0) --> False
 1. !!y. lam x xa. x ` ninj(xa) >> ?g13(y) :
            (S(?m11(y)) = y --> ?Q6(y)) -->
            S(?m11(S(y))) = S(y) --> ?Q6(S(y))
 2. nrec(lam x. nneq(x), all x. ?g13(x)) ^ S(0)
       !> ?p :
       S(?m11(S(0))) = S(0) --> ?Q6(S(0))
 3. !!y. y : S(?m11(S(0))) = S(0) -->
             ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br impCI 1;
Level 9
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       ya : S(?m11(y)) = y --> ?Q6(y) ==>
       lam x. ya ` ninj(x) >> ?g14(y, ya) :
          S(?m11(S(y))) = S(y) --> ?Q6(S(y))
 2. nrec
     (lam x. nneq(x),
      all x. lam xa. ?g14(x, xa)) ^
    S(0)
       !> ?p :
       S(?m11(S(0))) = S(0) --> ?Q6(S(0))
 3. !!y. y : S(?m11(S(0))) = S(0) -->
             ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br impCI 1;
Level 10
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       ya ` ninj(yb) >> ?g15(y, ya, yb) :
          ?Q6(S(y))
 2. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g15(x, xa, xb)) ^
    S(0)
       !> ?p :
       S(?m11(S(0))) = S(0) --> ?Q6(S(0))
 3. !!y. y : S(?m11(S(0))) = S(0) -->
             ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br impCE 1;
Level 11
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       ya >> ?e1'17(y, ya, yb) :
          ?Q17(y, ya, yb) --> ?Q6(S(y))
 2. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       ninj(yb) >> ?e2'17(y, ya, yb) :
          ?Q17(y, ya, yb)
 3. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       ?e1'17(y, ya, yb) ` ?e2'17(y, ya, yb)
          !> ?g15(y, ya, yb) : ?Q6(S(y))
 4. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g15(x, xa, xb)) ^
    S(0)
       !> ?p :
       S(?m11(S(0))) = S(0) --> ?Q6(S(0))
 5. !!y. y : S(?m11(S(0))) = S(0) -->
             ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   bv 1;
Level 12
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       ninj(yb) >> ?e2'17(y, ya, yb) :
          S(?m11(y)) = y
 2. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       ya ` ?e2'17(y, ya, yb)
          !> ?g15(y, ya, yb) : ?Q6(S(y))
 3. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g15(x, xa, xb)) ^
    S(0)
       !> ?p :
       S(?m11(S(0))) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(?m11(S(0))) = S(0) -->
             ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
```

```
-   br S_injectCI 1;
Level 13
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       yb >> ?p'20(y, ya, yb) :
          S(S(?m11(y))) = S(y)
 2. !!y ya yb.
       [| ya : S(?m11(y)) = y --> ?Q6(y);
          yb : S(?m11(S(y))) = S(y) |] ==>
       ya ` ninj(?p'20(y, ya, yb))
          !> ?g15(y, ya, yb) : ?Q6(S(y))
 3. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g15(x, xa, xb)) ^
    S(0)
       !> ?p :
       S(?m11(S(0))) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(?m11(S(0))) = S(0) -->
             ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 14
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(y) = y --> ?Q6(y);
          yb : S(S(y)) = S(y) |] ==>
       ya ` ninj(yb) !> ?g15(y, ya, yb) :
          ?Q6(S(y))
 2. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g15(x, xa, xb)) ^
    S(0)
       !> ?p : S(S(0)) = S(0) --> ?Q6(S(0))
 3. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bt 1;
Level 15
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    S(0)
       !> ?p : S(S(0)) = S(0) --> ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br inductionRS 1;
Level 16
?p : S(S(0)) = S(0) --> False
 1. (all x. lam x xa. x ` ninj(xa) ^ 0) `
    (nrec
      (lam x. nneq(x),
       all x. lam x xa. x ` ninj(xa)) ^
     0)
       >> ?p : S(S(0)) = S(0) --> ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCE 1;
Level 17
?p : S(S(0)) = S(0) --> False
 1. all x. lam x xa. x ` ninj(xa) ^ 0
       >> ?e1'25 :
       ?Q25 --> S(S(0)) = S(0) --> ?Q6(S(0))
 2. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : ?Q25
 3. ?e1'25 ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br allCE 1;
Level 18
?p : S(S(0)) = S(0) --> False
 1. all x. lam x xa. x ` ninj(xa) >> ?g26 :
       ALL b. ?R26(b)
 2. ?g26 ^ 0 !> ?e1'25 : ?R26(0)
 3. !!y. y : ?R26(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : ?Q28(0)
 5. ?e1'25 ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br allCI 1;
Level 19
?p : S(S(0)) = S(0) --> False
 1. !!y. lam x xa. x ` ninj(xa) >> ?g29(y) :
            ?R26(y)
 2. all x. ?g29(x) ^ 0 !> ?e1'25 : ?R26(0)
 3. !!y. y : ?R26(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : ?Q28(0)
 5. ?e1'25 ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
```

```
Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 20
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       ya : ?P30(y) ==>
       lam x. ya ‘ ninj(x) >> ?g30(y, ya) :
          ?Q30(y)
 2. all x. lam xa. ?g30(x, xa) ^ 0 !> ?e1’25 :
       ?P30(0) --> ?Q30(0)
 3. !!y. y : ?P30(0) --> ?Q30(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 5. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 21
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : ?P30(y); yb : ?P35(y) |] ==>
       ya ‘ ninj(yb) >> ?g31(y, ya, yb) :
          ?Q34(y)
 2. all x. lam xa xb. ?g31(x, xa, xb) ^ 0
       !> ?e1’25 :
       ?P30(0) --> ?P35(0) --> ?Q34(0)
 3. !!y. y : ?P30(0) --> ?P35(0) --> ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 5. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCE 1;
Level 22
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : ?P30(y); yb : ?P35(y) |] ==>
       ya >> ?e1’36(y, ya, yb) :
          ?Q36(y, ya, yb) --> ?Q34(y)
 2. !!y ya yb.
       [| ya : ?P30(y); yb : ?P35(y) |] ==>
       ninj(yb) >> ?e2’36(y, ya, yb) :
          ?Q36(y, ya, yb)
 3. !!y ya yb.
       [| ya : ?P30(y); yb : ?P35(y) |] ==>
       ?e1’36(y, ya, yb) ‘ ?e2’36(y, ya, yb)
          !> ?g31(y, ya, yb) : ?Q34(y)
 4. all x. lam xa xb. ?g31(x, xa, xb) ^ 0
       !> ?e1’25 :
       ?P30(0) --> ?P35(0) --> ?Q34(0)
 5. !!y. y : ?P30(0) --> ?P35(0) --> ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 6. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 7. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 8. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 23
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : ?Q41(y) --> ?Q34(y);
          yb : ?P35(y) |] ==>
       ninj(yb) >> ?e2’36(y, ya, yb) : ?Q41(y)
 2. !!y ya yb.
       [| ya : ?Q41(y) --> ?Q34(y);
          yb : ?P35(y) |] ==>
       ya ‘ ?e2’36(y, ya, yb)
          !> ?g31(y, ya, yb) : ?Q34(y)
 3. all x. lam xa xb. ?g31(x, xa, xb) ^ 0
       !> ?e1’25 :
       (?Q41(0) --> ?Q34(0)) -->
       ?P35(0) --> ?Q34(0)
 4. !!y. y : (?Q41(0) --> ?Q34(0)) -->
             ?P35(0) --> ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 5. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 6. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 7. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_injectCI 1;
Level 24
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : ?m46(y) = ?n45(y) --> ?Q34(y);
          yb : ?P35(y) |] ==>
       yb >> ?p’42(y, ya, yb) :
```

```
         S(?m46(y)) = S(?n45(y))
 2. !!y ya yb.
       [| ya : ?m46(y) = ?n45(y) --> ?Q34(y);
          yb : ?P35(y) |] ==>
       ya ‘ ninj(?p’42(y, ya, yb))
          !> ?g31(y, ya, yb) : ?Q34(y)
 3. all x. lam xa xb. ?g31(x, xa, xb) ^ 0
       !> ?e1’25 :
       (?m46(0) = ?n45(0) --> ?Q34(0)) -->
       ?P35(0) --> ?Q34(0)
 4. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             ?P35(0) --> ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 5. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 6. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 7. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 25
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : ?m46(y) = ?n45(y) --> ?Q34(y);
          yb : S(?m46(y)) = S(?n45(y)) |] ==>
       ya ‘ ninj(yb) !> ?g31(y, ya, yb) :
          ?Q34(y)
 2. all x. lam xa xb. ?g31(x, xa, xb) ^ 0
       !> ?e1’25 :
       (?m46(0) = ?n45(0) --> ?Q34(0)) -->
       S(?m46(0)) = S(?n45(0)) --> ?Q34(0)
 3. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 5. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bt 1;
Level 26
?p : S(S(0)) = S(0) --> False
 1. all x. lam x xa. x ‘ ninj(xa) ^ 0
       !> ?e1’25 :
       (?m46(0) = ?n45(0) --> ?Q34(0)) -->
       S(?m46(0)) = S(?n45(0)) --> ?Q34(0)
 2. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 3. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 4. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 5. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br allR 1;
Level 27
?p : S(S(0)) = S(0) --> False
 1. lam x xa. x ‘ ninj(xa) >> ?e1’25 : ?R52(0)
 2. !!y. y : ?R52(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 3. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 5. ?e1’25 ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 28
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?P61(0) ==>
         lam x. y ‘ ninj(x) >> ?g59(y) : ?Q60(0)
 2. !!y. y : ?P61(?x52) --> ?Q60(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 3. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
```

```
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 5. (lam x. ?g59(x)) ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 29
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       [| y : ?P61(0); ya : ?P64(0) |] ==>
       y ‘ ninj(ya) >> ?g62(y, ya) : ?Q63(0)
 2. !!y. y : ?P61(?x52) -->
             ?P64(?x52) --> ?Q63(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 3. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 5. (lam x xa. ?g62(x, xa)) ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCE 1;
Level 30
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       [| y : ?P61(0); ya : ?P64(0) |] ==>
       y >> ?e1’67(y, ya) :
          ?Q67(y, ya) --> ?Q63(0)
 2. !!y ya.
       [| y : ?P61(0); ya : ?P64(0) |] ==>
       ninj(ya) >> ?e2’67(y, ya) : ?Q67(y, ya)
 3. !!y ya.
       [| y : ?P61(0); ya : ?P64(0) |] ==>
       ?e1’67(y, ya) ‘ ?e2’67(y, ya)
          !> ?g62(y, ya) : ?Q63(0)
 4. !!y. y : ?P61(?x52) -->
             ?P64(?x52) --> ?Q63(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 5. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 6. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 7. (lam x xa. ?g62(x, xa)) ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 8. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 31
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       [| y : ?Q71(0) --> ?Q70(0);
          ya : ?P64(0) |] ==>
       ninj(ya) >> ?e2’67(y, ya) : ?Q71(0)
 2. !!y ya.
       [| y : ?Q71(0) --> ?Q70(0);
          ya : ?P64(0) |] ==>
       y ‘ ?e2’67(y, ya) !> ?g62(y, ya) :
          ?Q63(0)
 3. !!y. y : (?Q71(?x52) --> ?Q70(?x52)) -->
             ?P64(?x52) --> ?Q63(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 4. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 5. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’25 : ?Q28(0)
 6. (lam x xa. ?g62(x, xa)) ‘ ?e2’25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 7. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False
```

```
Flex-flex pairs:
  ?Q70(0) =?= ?Q63(0)
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_injectCI 1;
Level 32
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       [| y : ?m75(0) = ?n74(0) --> ?Q70(0);
          ya : ?P64(0) |] ==>
       ya >> ?p'73(y, ya) :
          S(?m75(0)) = S(?n74(0))
 2. !!y ya.
       [| y : ?m75(0) = ?n74(0) --> ?Q70(0);
          ya : ?P64(0) |] ==>
       y ` ninj(?p'73(y, ya)) !> ?g62(y, ya) :
          ?Q63(0)
 3. !!y. y : (?m75(?x52) = ?n74(?x52) -->
              ?Q70(?x52)) -->
             ?P64(?x52) --> ?Q63(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 4. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 5. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : ?Q28(0)
 6. (lam x xa. ?g62(x, xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 7. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q70(0) =?= ?Q63(0)
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 33
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       [| y : ?m75(0) = ?n74(0) --> ?Q70(0);
          ya : S(?m80(0)) = S(?n79(0)) |] ==>
       y ` ninj(ya) !> ?g62(y, ya) : ?Q63(0)
 2. !!y. y : (?m75(?x52) = ?n74(?x52) -->
              ?Q70(?x52)) -->
             S(?m80(?x52)) = S(?n79(?x52)) -->
             ?Q63(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 3. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : ?Q28(0)
 5. (lam x xa. ?g62(x, xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?m80(0) =?= ?m75(0)
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bt 1;
Level 34
?p : S(S(0)) = S(0) --> False
 1. !!y. y : (?m75(?x52) = ?n74(?x52) -->
              ?Q70(?x52)) -->
             S(?m80(?x52)) = S(?n79(?x52)) -->
             ?Q63(?x52) ==>
         y : (?m58(?x52) = ?n57(?x52) -->
              ?Q56(?x52)) -->
             S(?m55(?x52)) = S(?n54(?x52)) -->
             ?Q53(?x52)
 2. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 3. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : ?Q28(0)
 4. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 5. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?m80(0) =?= ?m75(0)
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
```

```
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   ba 1;
Level 35
?p : S(S(0)) = S(0) --> False
 1. !!y. y : (?m46(0) = ?n45(0) --> ?Q34(0)) -->
             S(?m46(0)) = S(?n45(0)) -->
             ?Q34(0) ==>
         y : ?Q28(0) -->
             S(S(0)) = S(0) --> ?Q27(0)
 2. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : ?Q28(0)
 3. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?n54(?x52)
  ?m80(?x52) =?= ?m55(?x52)
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?n57(?x52)
  ?m75(?x52) =?= ?m58(?x52)
  ?m80(0) =?= ?m75(0)
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?m58(0) =?= ?m46(0)
  ?n57(0) =?= ?n45(0)
  ?Q56(0) =?= ?Q34(0)
  ?m55(0) =?= ?m46(0)
  ?n54(0) =?= ?n45(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   ba 1;
Level 36
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa)) ^
    0
       >> ?e2'25 : S(0) = 0 --> ?Q85(0)
 2. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 3. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br allCE 1;
Level 37
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ` ninj(xa))
       >> ?g86 : ALL b. ?R86(b)
 2. ?g86 ^ 0 !> ?e2'25 : ?R86(0)
 3. !!y. y : ?R86(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 4. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 5. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br inductCI 1;
Level 38
?p : S(S(0)) = S(0) --> False
 1. lam x. nneq(x) >> ?b'88 : ?R86(0)
 2. !!y ya.
       ya : ?R86(y) ==>
       lam x. ya ` ninj(x) >> ?g88(y, ya) :
          ?R86(S(y))
 3. nrec(?b'88, all x. lam xa. ?g88(x, xa)) ^ 0
       !> ?e2'25 : ?R86(0)
 4. !!y. y : ?R86(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 5. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
```

```
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 39
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?P91(0) ==>
         nneq(y) >> ?g89(y) : ?Q90(0)
 2. !!y ya.
       ya : ?P91(y) --> ?Q90(y) ==>
       lam x. ya ‘ ninj(x) >> ?g88(y, ya) :
          ?P91(S(y)) --> ?Q90(S(y))
 3. nrec
     (lam x. ?g89(x),
      all x. lam xa. ?g88(x, xa)) ^
    0
       !> ?e2'25 : ?P91(0) --> ?Q90(0)
 4. !!y. y : ?P91(0) --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 5. (lam x xa. x ‘ ninj(xa)) ‘ ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_neq_0CI 1;
Level 40
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?P91(0) ==>
         y >> ?p'92(y) : S(?m92(y)) = 0
 2. !!y ya.
       ya : ?P91(y) --> ?Q90(y) ==>
       lam x. ya ‘ ninj(x) >> ?g88(y, ya) :
          ?P91(S(y)) --> ?Q90(S(y))
 3. nrec
     (lam x. nneq(?p'92(x)),
      all x. lam xa. ?g88(x, xa)) ^
    0
       !> ?e2'25 : ?P91(0) --> ?Q90(0)
 4. !!y. y : ?P91(0) --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 5. (lam x xa. x ‘ ninj(xa)) ‘ ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 41
?p : S(S(0)) = S(0) --> False
 1. !!y ya.
       ya : S(?m95(y)) = y --> ?Q90(y) ==>
       lam x. ya ‘ ninj(x) >> ?g88(y, ya) :
          S(?m95(S(y))) = S(y) --> ?Q90(S(y))
 2. nrec
     (lam x. nneq(x),
      all x. lam xa. ?g88(x, xa)) ^
    0
       !> ?e2'25 : S(?m95(0)) = 0 --> ?Q90(0)
 3. !!y. y : S(?m95(0)) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 4. (lam x xa. x ‘ ninj(xa)) ‘ ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 5. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 42
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       ya ‘ ninj(yb) >> ?g97(y, ya, yb) :
          ?Q90(S(y))
 2. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g97(x, xa, xb)) ^
    0
```

```
      !> ?e2'25 : S(?m95(0)) = 0 --> ?Q90(0)
 3. !!y. y : S(?m95(0)) = 0 --> ?Q90(0) ==>
        y : S(0) = 0 --> ?Q87(0)
 4. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
      S(S(0)) = S(0) --> ?Q6(S(0))
 5. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
        y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCE 1;
Level 43
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       ya >> ?e1'99(y, ya, yb) :
          ?Q99(y, ya, yb) --> ?Q90(S(y))
 2. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       ninj(yb) >> ?e2'99(y, ya, yb) :
          ?Q99(y, ya, yb)
 3. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       ?e1'99(y, ya, yb) ` ?e2'99(y, ya, yb)
          !> ?g97(y, ya, yb) : ?Q90(S(y))
 4. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g97(x, xa, xb)) ^
    0
       !> ?e2'25 : S(?m95(0)) = 0 --> ?Q90(0)
 5. !!y. y : S(?m95(0)) = 0 --> ?Q90(0) ==>
        y : S(0) = 0 --> ?Q87(0)
 6. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
      S(S(0)) = S(0) --> ?Q6(S(0))
 7. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
        y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 44
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       ninj(yb) >> ?e2'99(y, ya, yb) :
          S(?m95(y)) = y
 2. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       ya ` ?e2'99(y, ya, yb)
          !> ?g97(y, ya, yb) : ?Q90(S(y))
 3. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g97(x, xa, xb)) ^
    0
       !> ?e2'25 : S(?m95(0)) = 0 --> ?Q90(0)
 4. !!y. y : S(?m95(0)) = 0 --> ?Q90(0) ==>
        y : S(0) = 0 --> ?Q87(0)
 5. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
      S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
        y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_injectCI 1;
Level 45
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       yb >> ?p'102(y, ya, yb) :
          S(S(?m95(y))) = S(y)
 2. !!y ya yb.
       [| ya : S(?m95(y)) = y --> ?Q90(y);
          yb : S(?m95(S(y))) = S(y) |] ==>
       ya ` ninj(?p'102(y, ya, yb))
          !> ?g97(y, ya, yb) : ?Q90(S(y))
 3. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g97(x, xa, xb)) ^
```

```
    0
       !> ?e2'25 : S(?m95(0)) = 0 --> ?Q90(0)
 4. !!y. y : S(?m95(0)) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 5. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 6. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 46
?p : S(S(0)) = S(0) --> False
 1. !!y ya yb.
       [| ya : S(y) = y --> ?Q90(y);
          yb : S(S(y)) = S(y) |] ==>
       ya ` ninj(yb) !> ?g97(y, ya, yb) :
          ?Q90(S(y))
 2. nrec
     (lam x. nneq(x),
      all x. lam xa xb. ?g97(x, xa, xb)) ^
    0
       !> ?e2'25 : S(0) = 0 --> ?Q90(0)
 3. !!y. y : S(0) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 4. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 5. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bt 1;
Level 47
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam xa x. xa ` ninj(x)) ^
    0
       !> ?e2'25 : S(0) = 0 --> ?Q90(0)
 2. !!y. y : S(0) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 3. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br inductionR0 1;
Level 48
?p : S(S(0)) = S(0) --> False
 1. lam x. nneq(x) >> ?e2'25 :
       S(0) = 0 --> ?Q90(0)
 2. !!y. y : S(0) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 3. (lam x xa. x ` ninj(xa)) ` ?e2'25 !> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 49
```

```
?p : S(S(0)) = S(0) --> False
 1. !!y. y : S(0) = 0 ==>
         nneq(y) >> ?g107(y) : ?Q90(0)
 2. !!y. y : S(0) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 3. (lam x xa. x ‘ ninj(xa)) ‘
    (lam x. ?g107(x))
       !> ?p : S(S(0)) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_neq_0CI 1;
Level 50
?p : S(S(0)) = S(0) --> False
 1. !!y. y : S(0) = 0 ==>
         y >> ?p'108(y) : S(?m108(y)) = 0
 2. !!y. y : S(0) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 3. (lam x xa. x ‘ ninj(xa)) ‘
    (lam x. nneq(?p'108(x)))
       !> ?p : S(S(0)) = S(0) --> ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 51
?p : S(S(0)) = S(0) --> False
 1. !!y. y : S(0) = 0 --> ?Q90(0) ==>
         y : S(0) = 0 --> ?Q87(0)
 2. (lam x xa. x ‘ ninj(xa)) ‘ (lam x. nneq(x))
       !> ?p : S(S(0)) = S(0) --> ?Q6(S(0))
 3. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   ba 1;
Level 52
?p : S(S(0)) = S(0) --> False
 1. (lam x xa. x ‘ ninj(xa)) ‘ (lam x. nneq(x))
       !> ?p : S(S(0)) = S(0) --> ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impR 1;
Level 53
?p : S(S(0)) = S(0) --> False
 1. lam x. (lam x. nneq(x)) ‘ ninj(x) >> ?p :
       S(S(0)) = S(0) --> ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
```

```
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 54
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         (lam x. nneq(x)) ‘ ninj(y)
            >> ?g112(y) : ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCE 1;
Level 55
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         lam x. nneq(x) >> ?e1’113(y) :
            ?Q113(y) --> ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) ==>
         ninj(y) >> ?e2’113(y) : ?Q113(y)
 3. !!y. y : S(S(0)) = S(0) ==>
         ?e1’113(y) ‘ ?e2’113(y) !> ?g112(y) :
            ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impCI 1;
Level 56
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y ya.
       [| y : S(S(0)) = S(0);
          ya : ?Q113(y) |] ==>
       nneq(ya) >> ?g115(y, ya) : ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) ==>
         ninj(y) >> ?e2’113(y) : ?Q113(y)
 3. !!y. y : S(S(0)) = S(0) ==>
         (lam x. ?g115(y, x)) ‘ ?e2’113(y)
            !> ?g112(y) : ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_neq_0CI 1;
Level 57
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y ya.
       [| y : S(S(0)) = S(0);
          ya : ?Q113(y) |] ==>
       ya >> ?p’117(y, ya) : S(?m117(y, ya)) = 0
 2. !!y. y : S(S(0)) = S(0) ==>
         ninj(y) >> ?e2’113(y) : ?Q113(y)
 3. !!y. y : S(S(0)) = S(0) ==>
         (lam x. nneq(?p’117(y, x))) ‘
         ?e2’113(y)
            !> ?g112(y) : ?Q6(S(0))
 4. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
```

```
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 58
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         ninj(y) >> ?e2'113(y) : S(?m121(y)) = 0
 2. !!y. y : S(S(0)) = S(0) ==>
         (lam x. nneq(x)) ` ?e2'113(y)
            !> ?g112(y) : ?Q6(S(0))
 3. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_injectCI 1;
Level 59
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         y >> ?p'122(y) : S(S(?m121(y))) = S(0)
 2. !!y. y : S(S(0)) = S(0) ==>
         (lam x. nneq(x)) ` ninj(?p'122(y))
            !> ?g112(y) : ?Q6(S(0))
 3. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 60
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         (lam x. nneq(x)) ` ninj(y)
            !> ?g112(y) : ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br impR 1;
Level 61
lam x. ?g112(x) : S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         nneq(ninj(y)) >> ?g112(y) : ?Q6(S(0))
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_neq_0CI 1;
Level 62
```

```
lam x. nneq(?p'126(x))
: S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         ninj(y) >> ?p'126(y) : S(?m126(y)) = 0
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   br S_injectCI 1;
Level 63
lam x. nneq(ninj(?p'128(x)))
: S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) ==>
         y >> ?p'128(y) : S(S(?m126(y))) = S(0)
 2. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   bv 1;
Level 64
lam x. nneq(ninj(x)) : S(S(0)) = S(0) --> False
 1. !!y. y : S(S(0)) = S(0) --> ?Q6(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q90(0) =?= ?Q87(0)
  (%y ya yb. ?Q90(y)) =?= (%y ya yb. ?Q90(S(y)))
  ?Q87(0) =?= ?Q85(0)
  ?Q85(0) =?= ?Q34(0)
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?Q34(0) =?= ?Q27(0)
  ?Q63(?x52) =?= ?Q53(?x52)
  ?n79(?x52) =?= ?x52
  ?Q70(?x52) =?= ?Q56(?x52)
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
  ?Q70(0) =?= ?Q63(0)
  ?Q56(0) =?= ?Q34(0)
  ?Q53(0) =?= ?Q34(0)
  ?Q27(0) =?= ?Q6(S(0))
  (%y ya yb. ?Q6(y)) =?= (%y ya yb. ?Q6(S(y)))
val it = () : unit
-   ba 1;
Level 65
lam x. nneq(ninj(x)) : S(S(0)) = S(0) --> False
No subgoals!

Flex-flex pairs:
  ?x84(?x52) =?= ?x52
  ?x84(0) =?= ?x83(0)
  ?x83(?x52) =?= ?x52
  ?n79(?x52) =?= ?x52
  ?n74(?x52) =?= ?x52
  ?n79(0) =?= ?n74(0)
val it = () : unit
- val two_not_oneRed = result();
val two_not_oneRed =
  "lam x. nneq(ninj(x)) :
   S(S(0)) = S(0) --> False" : thm
```

An example showing why it is problematic working with inductive defined datatypes is the following reduction of the proof

```
nrec(idpeel
      (idpeel(idpeel(rec0, %x. ideq(S(x))), %x. ideq(x)),
       %x. lam x. x) `
     rec0,
     all x.
        lam xa.
           idpeel
            (idpeel
              (idpeel
```

```
                (idpeel
                  (idpeel
                    (idpeel(recS, %x. ideq(x)), %x. lam x. x) `
                   idpeel
                    (idpeel
                      (ideq(rec(x, n, %x y. S(y))), %x. ideq(x)),
                     %x. ideq(S(x))),
                    %x. lam x. x) `
                  xa,
                  %x. ideq(x)),
                 %x. ideq(S(x))),
               %x. lam x. x) `
             recS) ^
m
```

of proposition $m + \mathsf{S}(n) = \mathsf{S}(m + n)$ where at first all subgoals that can be solved using a context rule, tactic `assume_tac` or tactic `vartac` *without* Isabelle instantiating variables in an unpleasant way, yields the below state

```
Level 54
?p : m + S(n) = S(m + n)
 1. !!c. ideq(S(c)) >> ?g32(c) :
           ?a35(c, c) = ?b34(c, c)
 2. idpeel(rec0, ?g32) !> ?q31 :
       ?a35(rec(0, ?b32, ?f33), ?b32) =
       ?b34(rec(0, ?b32, ?f33), ?b32)
 3. !!y. y : ?P31
              (?a35(rec(0, ?b32, ?f33), ?b32),
               ?b34
                (rec(0, ?b32, ?f33), ?b32)) ==>
        y : ?a27 = ?b27
 4. !!c. ideq(c) >> ?g31(c) : ?P31(c, c)
 5. idpeel(?q31, ?g31) !> ?q27 :
       ?P31
        (?a35(rec(0, ?b32, ?f33), ?b32),
         ?b34(rec(0, ?b32, ?f33), ?b32))
 6. !!y. y : ?P30(?a27, ?b27) -->
             ?Q29(?a27, ?b27) ==>
        y : rec(0, ?a26, ?f26) = ?a26 -->
            ?R21(0)
 7. idpeel(?q27, %c. lam x. x) !> ?e1'25 :
       ?P30(?a27, ?b27) --> ?Q29(?a27, ?b27)
 8. ?e1'25 ` rec0 !> ?b'22 : ?R21(0)
 9. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P58
             (y, ya,
              rec(S(?m59(y, ya)), ?a59(y, ya),
                  ?f59(y, ya)),
              ?f59
               (y, ya, ?m59(y, ya),
                rec(?m59(y, ya), ?a59(y, ya),
                    ?f59(y, ya)))) |] ==>
      yb : ?a52(y, ya) = ?b52(y, ya)
10. !!y ya c.
       ya : ?R21(y) ==>
       ideq(c) >> ?g58(y, ya, c) :
          ?P58(y, ya, c, c)
11. !!y ya.
       ya : ?R21(y) ==>
       idpeel(recS, ?g58(y, ya))
                !> ?q52(y, ya) :
                ?P58
                 (y, ya,
                  rec(S(?m59(y, ya)), ?a59(y, ya),
                      ?f59(y, ya)),
                  ?f59
                   (y, ya, ?m59(y, ya),
                    rec(?m59(y, ya), ?a59(y, ya),
                        ?f59(y, ya))))
12. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P55
             (y, ya, ?a52(y, ya),
              ?b52(y, ya)) -->
            ?Q54
             (y, ya, ?a52(y, ya),
              ?b52(y, ya)) |] ==>
       yb
       : ?Q51(y, ya) -->
         ?a50(y, ya) = ?b50(y, ya)
13. !!y ya.
       ya : ?R21(y) ==>
       idpeel(?q52(y, ya), %c. lam x. x)
          !> ?e1'51(y, ya) :
          ?P55
           (y, ya, ?a52(y, ya),
            ?b52(y, ya)) -->
          ?Q54(y, ya, ?a52(y, ya), ?b52(y, ya))
14. !!y ya.
       ya : ?R21(y) ==>
       ideq(rec(y, n, %x y. S(y)))
          >> ?q61(y, ya) :
          ?a61(y, ya) = ?b61(y, ya)
15. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P61
             (y, ya, ?a61(y, ya),
              ?b61(y, ya)) |] ==>
       yb : ?a60(y, ya) = ?b60(y, ya)
16. !!y ya c.
       ya : ?R21(y) ==>
```

```
       ideq(c) >> ?g61(y, ya, c) :
          ?P61(y, ya, c, c)
17. !!y ya.
       ya : ?R21(y) ==>
       idpeel(?q61(y, ya), ?g61(y, ya))
          !> ?q60(y, ya) :
          ?P61(y, ya, ?a61(y, ya), ?b61(y, ya))
18. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P60
             (y, ya, ?a60(y, ya),
              ?b60(y, ya)) |] ==>
       yb : ?Q51(y, ya)
19. !!y ya c.
       ya : ?R21(y) ==>
       ideq(S(c)) >> ?g60(y, ya, c) :
          ?P60(y, ya, c, c)
20. !!y ya.
       ya : ?R21(y) ==>
       idpeel(?q60(y, ya), ?g60(y, ya))
          !> ?e2'51(y, ya) :
          ?P60(y, ya, ?a60(y, ya), ?b60(y, ya))
21. !!y ya.
       ya : ?R21(y) ==>
       ?e1'51(y, ya) ` ?e2'51(y, ya)
          !> ?q50(y, ya) :
          ?a50(y, ya) = ?b50(y, ya)
22. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P64
             (y, ya, ?a50(y, ya),
              ?b50(y, ya)) -->
            ?Q63
             (y, ya, ?a50(y, ya),
              ?b50(y, ya)) |] ==>
       yb
       : ?Q49(y, ya) -->
         ?a48(y, ya) = ?b48(y, ya)
23. !!y ya.
       ya : ?R21(y) ==>
       idpeel(?q50(y, ya), %c. lam x. x)
          !> ?e1'49(y, ya) :
          ?P64
           (y, ya, ?a50(y, ya),
            ?b50(y, ya)) -->
          ?Q63(y, ya, ?a50(y, ya), ?b50(y, ya))
24. !!y ya.
       ya : ?R21(y) ==>
       ya >> ?e2'49(y, ya) : ?Q49(y, ya)
25. !!y ya.
       ya : ?R21(y) ==>
       ?e1'49(y, ya) ` ?e2'49(y, ya)
          !> ?q48(y, ya) :
          ?a48(y, ya) = ?b48(y, ya)
26. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P48
             (y, ya, ?a48(y, ya),
              ?b48(y, ya)) |] ==>
       yb : ?a47(y, ya) = ?b47(y, ya)
27. !!y ya c.
       ya : ?R21(y) ==>
       ideq(c) >> ?g48(y, ya, c) :
          ?P48(y, ya, c, c)
28. !!y ya.
       ya : ?R21(y) ==>
       idpeel(?q48(y, ya), ?g48(y, ya))
          !> ?q47(y, ya) :
          ?P48(y, ya, ?a48(y, ya), ?b48(y, ya))
29. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P47
             (y, ya, ?a47(y, ya),
              ?b47(y, ya)) |] ==>
       yb : ?a41(y, ya) = ?b41(y, ya)
30. !!y ya c.
       ya : ?R21(y) ==>
       ideq(S(c)) >> ?g47(y, ya, c) :
          ?P47(y, ya, c, c)
31. !!y ya.
       ya : ?R21(y) ==>
       idpeel(?q47(y, ya), ?g47(y, ya))
          !> ?q41(y, ya) :
          ?P47(y, ya, ?a47(y, ya), ?b47(y, ya))
32. !!y ya yb.
       [| ya : ?R21(y);
          yb
          : ?P44
             (y, ya, ?a41(y, ya),
              ?b41(y, ya)) -->
            ?Q43
             (y, ya, ?a41(y, ya),
              ?b41(y, ya)) |] ==>
       yb
       : rec(S(?m40(y, ya)), ?a40(y, ya),
             ?f40(y, ya)) =
         ?f40
          (y, ya, ?m40(y, ya),
           rec(?m40(y, ya), ?a40(y, ya),
               ?f40(y, ya))) -->
         ?R21(S(y))
33. !!y ya.
       ya : ?R21(y) ==>
       idpeel(?q41(y, ya), %c. lam x. x)
          !> ?e1'38(y, ya) :
          ?P44
           (y, ya, ?a41(y, ya),
            ?b41(y, ya)) -->
          ?Q43(y, ya, ?a41(y, ya), ?b41(y, ya))
34. !!y ya.
       ya : ?R21(y) ==>
       ?e1'38(y, ya) ` recS !> ?g24(y, ya) :
          ?R21(S(y))
35. nrec(?b'22, all x. lam xa. ?g24(x, xa)) ^
    m
       !> ?p : ?R21(m)
36. !!y. y : ?R21(m) ==>
         y : rec(m, S(n), %x y. S(y)) =
             S(rec(m, n, %x y. S(y)))

Flex-flex pairs:
  (%y ya c. ?P64(y, ya, c, c)) =?=
  (%y ya c. ?Q63(y, ya, c, c))
  (%y ya c. ?P55(y, ya, c, c)) =?=
  (%y ya c. ?Q54(y, ya, c, c))
  (%y ya c. ?P44(y, ya, c, c)) =?=
  (%y ya c. ?Q43(y, ya, c, c))
  (%c. ?P30(c, c)) =?= (%c. ?Q29(c, c))
val it = () : unit
```

where in fact all subgoals that should be solved with axiom substCI is left unsolved, and these subgoals are the only unsolved subgoals that must be solved with a context rule. None of these subgoals that must be solved by either axiom substCI, tactic `vartac` or tactic `assume_tac` can be solved without invoking an unpleasant instantiation done by Isabelle. Therefore these subgoals may all be solved simultaneously to get the correct instantiations of variables. This is done by

```
- by ((assume_tac 36) THEN (assume_tac 32) THEN
=     (rtac substCI 30) THEN (assume_tac 29)
=     THEN (rtac substCI 27) THEN (assume_tac
=                           26)
=     THEN (vartac 24) THEN (assume_tac 22)
=     THEN  (rtac substCI 19) THEN (assume_tac
=                            18)
=     THEN (rtac substCI 16) THEN (assume_tac
=                           15) THEN
=     (rtac substCI 14) THEN (assume_tac
=                       12) THEN (rtac
=                                 substCI
=                                 10)
=     THEN (assume_tac 9) THEN (assume_tac 6)
=     THEN (rtac substCI 4) THEN (assume_tac 3)
=     THEN (rtac substCI 1));
Level 55
?p : m + S(n) = S(m + n)
 1. idpeel(rec0, %c. ideq(S(c))) !> ?q31 :
       S(n) = S(rec(0, n, %u ua. S(ua)))
 2. idpeel(?q31, %c. ideq(c)) !> ?q27 :
       S(n) = S(rec(0, n, %u ua. S(ua)))
 3. idpeel(?q27, %c. lam x. x) !> ?e1'25 :
       rec(0, S(rec(0, n, %x y. S(y))),
           %x y. S(y)) =
       S(rec(0, n, %x y. S(y))) -->
       rec(0, S(n), %x y. S(y)) =
       S(rec(0, n, %x y. S(y)))
 4. ?e1'25 ` rec0 !> ?b'22 :
       rec(0, S(n), %x y. S(y)) =
       S(rec(0, n, %x y. S(y)))
 5. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel(recS, %c. ideq(c))
          !> ?q52(y, ya) :
          S(rec(y, n, %u ua. S(ua))) =
          rec(S(y), n, %u ua. S(ua))
 6. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel(?q52(y, ya), %c. lam x. x)
          !> ?e1'51(y, ya) :
          S(rec(y, n, %x y. S(y))) =
          S(rec(y, n, %x y. S(y))) -->
          S(rec(y, n, %x y. S(y))) =
          rec(S(y), n, %x y. S(y))
 7. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel
        (ideq(rec(y, n, %x y. S(y))),
         %c. ideq(c))
          !> ?q60(y, ya) :
          rec(y, n, %x y. S(y)) =
          rec(y, n, %x y. S(y))
 8. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel(?q60(y, ya), %c. ideq(S(c)))
          !> ?e2'51(y, ya) :
          S(rec(y, n, %x y. S(y))) =
          S(rec(y, n, %x y. S(y)))
 9. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       ?e1'51(y, ya) ` ?e2'51(y, ya)
          !> ?q50(y, ya) :
          S(rec(y, n, %x y. S(y))) =
          rec(S(y), n, %x y. S(y))
10. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel(?q50(y, ya), %c. lam x. x)
          !> ?e1'49(y, ya) :
          rec(y, S(n), %u ua. S(ua)) =
          S(rec(y, n, %x y. S(y))) -->
          rec(y, S(n), %u ua. S(ua)) =
          rec(S(y), n, %x y. S(y))
11. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       ?e1'49(y, ya) ` ya !> ?q48(y, ya) :
          rec(y, S(n), %u ua. S(ua)) =
          rec(S(y), n, %x y. S(y))
12. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel(?q48(y, ya), %c. ideq(c))
          !> ?q47(y, ya) :
          rec(y, S(n), %u ua. S(ua)) =
          rec(S(y), n, %x y. S(y))
13. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel(?q47(y, ya), %c. ideq(S(c)))
          !> ?q41(y, ya) :
          S(rec(y, S(n), %u ua. S(ua))) =
          S(rec(S(y), n, %x y. S(y)))
14. !!y ya.
       ya
       : rec(y, S(n), %x y. S(y)) =
         S(rec(y, n, %x y. S(y))) ==>
       idpeel(?q41(y, ya), %c. lam x. x)
          !> ?e1'38(y, ya) :
          rec(S(y), S(n), %x y. S(y)) =
          S(rec(y, S(n), %u ua. S(ua))) -->
          rec(S(y), S(n), %x y. S(y)) =
          S(rec(S(y), n, %x y. S(y)))
15. !!y ya.
```

```
ya
: rec(y, S(n), %x y. S(y)) =
  S(rec(y, n, %x y. S(y))) ==>
?e1'38(y, ya) ‘ recS !> ?g24(y, ya) :
   rec(S(y), S(n), %x y. S(y)) =
   S(rec(S(y), n, %x y. S(y)))
16. nrec(?b'22, all x. lam xa. ?g24(x, xa)) ^
    m
       !> ?p :
       rec(m, S(n), %x y. S(y)) =
       S(rec(m, n, %x y. S(y)))
val it = () : unit
```

Solving the rest of the subgoals gives the reduced proof

```
val add_S_rightCheck =
  "nrec
    (idpeel(idpeel(idpeel(rec0, %c. ideq(S(c))), %c. ideq(c)), %c. lam x. x) ‘
     rec0,
     all x.
        lam xa.
           idpeel
            (idpeel
              (idpeel
                (idpeel
                  (idpeel(idpeel(recS, %c. ideq(c)), %c. lam x. x) ‘
                   ideq(S(rec(x, ?n, %x y. S(y)))), %c. lam x. x) ‘ xa,
                 %c. ideq(c)), %c. ideq(S(c))), %c. lam x. x) ‘ recS) ^ ?m
   : ?m + S(?n) = S(?m + ?n)" : thm
```

B.8 Natural Numbers with Proofs and Program Evaluation

An object logic for evaluating proofs including natural numbers as they were programs shall be created as object logic PENAT corresponding to formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ defined in section 5.3.2.

B.8.1 Implementing Theory

Formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ specifies that program evaluation shall be reduction of proofs that are proven using formal theory $\mathcal{NATP}$ with the reduction methods specified in formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$. Therefore object logic PENAT must be built on top of object logics PE and NATP.

As the second of the two object logics implementing normalization of proofs including natural numbers also this object logic's common terms with object logic PRNAT is placed in object logic NATReduction presented in section A.12.

Object logic PENAT for evaluation of proofs including natural numbers as they were programs implements the parts of formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{NAT}}$ that is not shared with formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{NAT}}$. Object logic PENAT is in appendix A.14.

B.8.2 Examples

The only thing to demonstrate of object logic PENAT is the rules added to existing theories which is reduction rules $\mathsf{inductionR}_0$ and $\mathsf{inductionR}_S$. These are demonstrated with object logic PRNAT.

A full reduction of an application of a theorem on a term is much shorter with program evaluation than with proof reduction, which is shown with the below example that program evaluates the proof of proposition $(\mathsf{S}(\mathsf{S}(\mathsf{0})) = \mathsf{S}(\mathsf{0})) \rightarrow \bot$ that is proof reduced in section B.7.2.

```
- goal PENAT.thy
= "?p :  S(S(0)) = S(0) --> False";
Level 0
?p : S(S(0)) = S(0) --> False
 1. ?p : S(S(0)) = S(0) --> False
val it = [] : thm list
-   br redE 1;
Level 1
?p : S(S(0)) = S(0) --> False
 1. ?a1 : S(S(0)) = S(0) --> False
 2. ?a1 >> ?p : S(S(0)) = S(0) --> False
val it = () : unit
-   br S_n_not_n 1;
Level 2
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    S(0)
       >> ?p : S(S(0)) = S(0) --> False
val it = () : unit
-   br allCE 1;
Level 3
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa))
       >> ?g3 : ALL b. ?R3(b)
 2. ?g3 ^ S(0) !> ?p : ?R3(S(0))
 3. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   bta 1;
Level 4
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    S(0)
       !> ?p : ?R3(S(0))
 2. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br inductionRS 1;
Level 5
?p : S(S(0)) = S(0) --> False
 1. (all x. lam x xa. x ‘ ninj(xa) ^ 0) ‘
    (nrec
      (lam x. nneq(x),
       all x. lam x xa. x ‘ ninj(xa)) ^
     0)
       >> ?p : ?R3(S(0))
 2. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   br impCE 1;
Level 6
?p : S(S(0)) = S(0) --> False
 1. all x. lam x xa. x ‘ ninj(xa) ^ 0 >> ?e1’6 :
       ?Q6 --> ?R3(S(0))
 2. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’6 : ?Q6
 3. ?e1’6 ‘ ?e2’6 !> ?p : ?R3(S(0))
 4. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False
val it = () : unit
-   bta 1;
Level 7
?p : S(S(0)) = S(0) --> False
 1. all x. lam x xa. x ‘ ninj(xa) >> ?g7 :
       ALL b. ?R7(b)
 2. ?g7 ^ 0 !> ?e1’6 : ?R7(0)
 3. !!y. y : ?R7(0) ==> y : ?Q9(0) --> ?R8(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’6 : ?Q9(0)
 5. ?e1’6 ‘ ?e2’6 !> ?p : ?R3(S(0))
 6. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   bta 1;
Level 8
?p : S(S(0)) = S(0) --> False
 1. all x. lam x xa. x ‘ ninj(xa) ^ 0 !> ?e1’6 :
       ?R7(0)
 2. !!y. y : ?R7(0) ==> y : ?Q9(0) --> ?R8(0)
 3. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’6 : ?Q9(0)
 4. ?e1’6 ‘ ?e2’6 !> ?p : ?R3(S(0))
 5. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   br allR 1;
Level 9
?p : S(S(0)) = S(0) --> False
 1. lam x xa. x ‘ ninj(xa) >> ?e1’6 : ?R11(0)
 2. !!y. y : ?R11(?x11) ==> y : ?P11(?x11)
 3. !!y. y : ?R7(0) ==> y : ?Q9(0) --> ?R8(0)
 4. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’6 : ?Q9(0)
 5. ?e1’6 ‘ ?e2’6 !> ?p : ?R3(S(0))
 6. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?P11(0) =?= ?R7(0)
  ?R8(0) =?= ?R3(S(0))
```

```
val it = () : unit
-   bta 1;
Level 10
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?R11(?x11) ==> y : ?P11(?x11)
 2. !!y. y : ?R7(0) ==> y : ?Q9(0) --> ?R8(0)
 3. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’6 : ?Q9(0)
 4. (lam x xa. x ‘ ninj(xa)) ‘ ?e2’6 !> ?p :
       ?R3(S(0))
 5. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?P11(0) =?= ?R7(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   ba 1;
Level 11
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?R7(0) ==> y : ?Q9(0) --> ?R8(0)
 2. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’6 : ?Q9(0)
 3. (lam x xa. x ‘ ninj(xa)) ‘ ?e2’6 !> ?p :
       ?R3(S(0))
 4. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?R11(?x11) =?= ?P11(?x11)
  ?P11(0) =?= ?R7(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   ba 1;
Level 12
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       >> ?e2’6 : ?Q9(0)
 2. (lam x xa. x ‘ ninj(xa)) ‘ ?e2’6 !> ?p :
       ?R3(S(0))
 3. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   br allCE 1;
Level 13
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa))
       >> ?g19 : ALL b. ?R19(b)
 2. ?g19 ^ 0 !> ?e2’6 : ?R19(0)
 3. !!y. y : ?R19(0) ==> y : ?P19(0)
 4. (lam x xa. x ‘ ninj(xa)) ‘ ?e2’6 !> ?p :
       ?R3(S(0))
 5. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   bta 1;
Level 14
?p : S(S(0)) = S(0) --> False
 1. nrec
     (lam x. nneq(x),
      all x. lam x xa. x ‘ ninj(xa)) ^
    0
       !> ?e2’6 : ?R19(0)
 2. !!y. y : ?R19(0) ==> y : ?P19(0)
 3. (lam x xa. x ‘ ninj(xa)) ‘ ?e2’6 !> ?p :
       ?R3(S(0))
 4. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   br inductionR0 1;
Level 15
?p : S(S(0)) = S(0) --> False
 1. lam x. nneq(x) >> ?e2’6 : ?R19(0)
 2. !!y. y : ?R19(0) ==> y : ?P19(0)
 3. (lam x xa. x ‘ ninj(xa)) ‘ ?e2’6 !> ?p :
       ?R3(S(0))
 4. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   bta 1;
Level 16
?p : S(S(0)) = S(0) --> False
 1. !!y. y : ?R19(0) ==> y : ?P19(0)
 2. (lam x xa. x ‘ ninj(xa)) ‘ (lam x. nneq(x))
```

```
      !> ?p : ?R3(S(0))
 3. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   ba 1;
Level 17
?p : S(S(0)) = S(0) --> False
 1. (lam x xa. x ` ninj(xa)) ` (lam x. nneq(x))
       !> ?p : ?R3(S(0))
 2. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?R19(0) =?= ?P19(0)
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   br impR 1;
Level 18
?p : S(S(0)) = S(0) --> False
 1. lam x. (lam x. nneq(x)) ` ninj(x) >> ?p :
       ?R3(S(0))
 2. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?R19(0) =?= ?P19(0)
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   bta 1;
Level 19
lam x. (lam x. nneq(x)) ` ninj(x)
: S(S(0)) = S(0) --> False
 1. !!y. y : ?R3(S(0)) ==>
         y : S(S(0)) = S(0) --> False

Flex-flex pairs:
  ?R19(0) =?= ?P19(0)
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?R17(?x11) =?= ?R15(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?R15(0) =?= ?R13(0)
  ?Q14(0) =?= ?Q9(0)
  ?R13(0) =?= ?R8(0)
  ?R8(0) =?= ?R3(S(0))
val it = () : unit
-   ba 1;
Level 20
lam x. (lam x. nneq(x)) ` ninj(x)
: S(S(0)) = S(0) --> False
No subgoals!

Flex-flex pairs:
  ?x26(?x11) =?= ?x11
  ?x25(?x11) =?= ?x11
  ?R19(0) =?= ?P19(0)
  ?P19(0) =?= ?Q9(0)
  ?Q18(?x11) =?= ?Q16(?x11)
  ?Q16(0) =?= ?Q14(0)
  ?Q14(0) =?= ?Q9(0)
val it = () : unit
- val t1 = result();
val t1 =
  "lam x. (lam x. nneq(x)) ` ninj(x) :
   S(S(0)) = S(0) --> False" : thm
```

The outcoming proof is not on normal form but only on weak head normal form, exactly as expected since program evaluation is used.

B.9 Lists

An object logic for lists of elements of some type based on the Peano axioms can be created as what looks like the implemented object logic `Nat` for natural numbers where the constructor list `C` is regarded as the constructor `S` annotated with some element and `Nil` is list terminator like `0` is the natural number terminator.

The implementation of lists of elements of some type will also include rewriting rules defining the length of a list and appending of lists.

B.9.1 Implementing Theory

The object logic `List` representing formal theory $\mathcal{LIST}$ can be found in appendix A.15.

B.9.2 Examples

Some examples may show how the object logic can be used. Each rule shall be used at least once in the examples and the examples may show both some general examples and some specific application of a theorem on a specific list.

An example showing the use of the inductl rule proving a more safe induction rule

$$\frac{P(\mathsf{Nil}) \qquad \forall x\ xs.P(xs) \rightarrow P(\mathsf{C}(x,xs))}{\forall l.P(l)}\ \text{induction}$$

is the following

```
- val [hb,hi] = goal List.thy
=   "[| P(Nil);
=       ALL x xs. P(xs) --> P(C(x,xs)) |] ==>
=    ALL l. P(l)";
std_in:8.1-9.64 Warning: binding not exhaustive
         hb :: hi :: nil = ...
Level 0
ALL l. P(l)
 1. ALL l. P(l)
val hb = "P(Nil)  [P(Nil)]" : thm
val hi = "ALL x xs. P(xs) --> P(C(x, xs))
          [ALL x xs. P(xs) --> P(C(x, xs))]"
  : thm
-     br inductl 1;
Level 1
ALL l. P(l)
 1. P(Nil)
 2. !!x xs. P(xs) ==> P(C(x, xs))
val it = () : unit
-     br hb 1;
Level 2
ALL l. P(l)
 1. !!x xs. P(xs) ==> P(C(x, xs))
val it = () : unit
-     br mp 1;
Level 3
ALL l. P(l)
 1. !!x xs. P(xs) ==> ?P1(x, xs) --> P(C(x, xs))
 2. !!x xs. P(xs) ==> ?P1(x, xs)
val it = () : unit
-     br allE 1;
Level 4
ALL l. P(l)
 1. !!x xs. P(xs) ==> ALL xa. ?P2(x, xs, xa)
 2. !!x xs.
       [| P(xs); ?P2(x, xs, ?x2(x, xs)) |] ==>
       ?P1(x, xs) --> P(C(x, xs))
 3. !!x xs. P(xs) ==> ?P1(x, xs)
val it = () : unit
-     br allE 1;
Level 5
ALL l. P(l)
 1. !!x xs. P(xs) ==> ALL xa. ?P3(x, xs, xa)
 2. !!x xs.
       [| P(xs); ?P3(x, xs, ?x3(x, xs)) |] ==>
       ALL xa. ?P2(x, xs, xa)
 3. !!x xs.
       [| P(xs); ?P2(x, xs, ?x2(x, xs)) |] ==>
       ?P1(x, xs) --> P(C(x, xs))
 4. !!x xs. P(xs) ==> ?P1(x, xs)
val it = () : unit
-     br hi 1;
Level 6
ALL l. P(l)
 1. !!x xs.
       [| P(xs);
          ALL xsa.
             P(xsa) -->
             P(C(?x3(x, xs), xsa)) |] ==>
       ALL xa. ?P2(x, xs, xa)
 2. !!x xs.
       [| P(xs); ?P2(x, xs, ?x2(x, xs)) |] ==>
       ?P1(x, xs) --> P(C(x, xs))
 3. !!x xs. P(xs) ==> ?P1(x, xs)
val it = () : unit
-     ba 1;
Level 7
ALL l. P(l)
 1. !!x xs.
       [| P(xs);
          P(?x2(x, xs)) -->
          P(C(?x3(x, xs), ?x2(x, xs))) |] ==>
       ?P1(x, xs) --> P(C(x, xs))
 2. !!x xs. P(xs) ==> ?P1(x, xs)
val it = () : unit
-     ba 1;
Level 8
ALL l. P(l)
 1. !!x xs. P(xs) ==> P(xs)
val it = () : unit
-     ba 1;
Level 9
ALL l. P(l)
No subgoals!
val it = () : unit
- val induction = result();
val induction =
  "[| ?P(Nil);
      ALL x xs. ?P(xs) --> ?P(C(x, xs)) |] ==>
   ALL l. ?P(l)" : thm
```

An example proving the inverse of inference rule $\mathsf{C}_{\mathsf{inj1}}$ being

$$\frac{x = y}{\mathsf{C}(x,l) = \mathsf{C}(y,l)}\ \mathsf{C}_{\mathsf{surj1}}$$

is

```
- val [h] =
= goal List.thy "x = y ==> C(x,l) = C(y,l)";
std_in:24.1-24.51 Warning:
  binding not exhaustive
          h :: nil = ...
Level 0
C(x, l) = C(y, l)
 1. C(x, l) = C(y, l)
val h = "x = y  [x = y]" : thm
```

```
-   by ((rtac subst_context 1) THEN (rtac h 1));
Level 1
C(x, l) = C(y, l)
No subgoals!
val it = () : unit
- val C_surj1 = result();
val C_surj1 =
  "?x = ?y ==> C(?x, ?l) = C(?y, ?l)" : thm
```

An example proving the inverse of inference rule $\mathsf{C}_{\mathsf{inj2}}$ being

$$\frac{xs = ys}{\mathsf{C}(a,xs) = \mathsf{C}(a,ys)}\ \mathsf{C}_{\mathsf{surj2}}$$

is

```
- val [h] =
  goal List.thy "xs = ys ==> C(a,xs) = C(a,ys)";
std_in:27.1-27.55 Warning:
  binding not exhaustive
          h :: nil = ...
Level 0
C(a, xs) = C(a, ys)
 1. C(a, xs) = C(a, ys)
val h = "xs = ys  [xs = ys]" : thm
```

```
-   by ((rtac subst_context 1) THEN (rtac h 1));
Level 1
C(a, xs) = C(a, ys)
No subgoals!
val it = () : unit
- val C_surj2 = result();
val C_surj2 =
  "?xs = ?ys ==> C(?a, ?xs) = C(?a, ?ys)" : thm
```

Because natural deduction is used it is not possible to create a rule stating that if two lists are equal then the components of the list are equal. The other way, represented by inference rule

$$\frac{x_1 = x_2 \qquad xs_1 = xs_2}{\mathsf{C}(x_1,xs_1) = \mathsf{C}(x_2,xs_2)}\ \mathsf{C}_{\mathsf{surj}}$$

can be proved by

```
- val [h1,h2] = goal List.thy
=   "[| x = (y::'a) ;
=       xs = (ys::'a list) |] ==>
=    C(x,xs) = C(y,ys)";
std_in:30.1-31.64 Warning:
  binding not exhaustive
          h1 :: h2 :: nil = ...
Level 0
C(x, xs) = C(y, ys)
 1. C(x, xs) = C(y, ys)
val h1 = "x = y  [x = y]" : thm
val h2 = "xs = ys  [xs = ys]" : thm
-   br subst 1;
Level 1
C(x, xs) = C(y, ys)
 1. ?x = ?y
 2. !!x. ?P(x, x)
 3. ?P(?x, ?y) ==> C(x, xs) = C(y, ys)
val it = () : unit
-   br h1 1;
```

```
Level 2
C(x, xs) = C(y, ys)
 1. !!x. ?P(x, x)
 2. ?P(x, y) ==> C(x, xs) = C(y, ys)
val it = () : unit
-   ba 2;
Level 3
C(x, xs) = C(y, ys)
 1. !!x. C(x, xs) = C(x, ys)
val it = () : unit
-   br S_surj2 1;
Level 4
C(x, xs) = C(y, ys)
 1. !!x. xs = ys
val it = () : unit
-   br h2 1;
Level 5
C(x, xs) = C(y, ys)
No subgoals!
val it = () : unit
```

```
- val C_surj = result();
val C_surj =
  "[| ?x = ?y; ?xs = ?ys |] ==>
   C(?x, ?xs) = C(?y, ?ys)" : thm
```

An example proving theorem $\forall l\ x.\ (\mathsf{C}(x, l) = l) \rightarrow \bot$ demonstrating inference rules $\mathsf{C} \neq \mathsf{Nil}$ and $\mathsf{C}_{\mathsf{inj1}}$ is

```
- goal List.thy
= "ALL xs x . (C(x,xs) = xs) --> False";
Level 0
ALL xs x. C(x, xs) = xs --> False
 1. ALL xs x. C(x, xs) = xs --> False
val it = [] : thm list
- br induction 1;
Level 1
ALL xs x. C(x, xs) = xs --> False
 1. ALL x. C(x, Nil) = Nil --> False
 2. ALL x xs.
       (ALL x. C(x, xs) = xs --> False) -->
       (ALL xa.
           C(xa, C(x, xs)) = C(x, xs) --> False)
val it = () : unit
- br allI 1;
Level 2
ALL xs x. C(x, xs) = xs --> False
 1. !!x. C(x, Nil) = Nil --> False
 2. ALL x xs.
       (ALL x. C(x, xs) = xs --> False) -->
       (ALL xa.
           C(xa, C(x, xs)) = C(x, xs) --> False)
val it = () : unit
- br impI 1;
Level 3
ALL xs x. C(x, xs) = xs --> False
 1. !!x. C(x, Nil) = Nil ==> False
 2. ALL x xs.
       (ALL x. C(x, xs) = xs --> False) -->
       (ALL xa.
           C(xa, C(x, xs)) = C(x, xs) --> False)
val it = () : unit
- br C_neq_Nil 1;
Level 4
ALL xs x. C(x, xs) = xs --> False
 1. !!x. C(x, Nil) = Nil ==>
         C(?x3(x), ?xs3(x)) = Nil
 2. ALL x xs.
       (ALL x. C(x, xs) = xs --> False) -->
       (ALL xa.
           C(xa, C(x, xs)) = C(x, xs) --> False)
val it = () : unit
- ba 1;
Level 5
ALL xs x. C(x, xs) = xs --> False
 1. ALL x xs.
       (ALL x. C(x, xs) = xs --> False) -->
       (ALL xa.
           C(xa, C(x, xs)) = C(x, xs) --> False)
val it = () : unit
- br allI 1;
Level 6
ALL xs x. C(x, xs) = xs --> False
 1. !!x. ALL xs.
            (ALL x. C(x, xs) = xs --> False) -->
            (ALL xa.
                C(xa, C(x, xs)) = C(x, xs) -->
                False)
val it = () : unit
- br allI 1;
Level 7
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs.
       (ALL x. C(x, xs) = xs --> False) -->
       (ALL xa.
           C(xa, C(x, xs)) = C(x, xs) --> False)
val it = () : unit
- br impI 1;
Level 8
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs.
       ALL x. C(x, xs) = xs --> False ==>
       ALL xa.
          C(xa, C(x, xs)) = C(x, xs) --> False
val it = () : unit
- br allI 1;
Level 9
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       ALL x. C(x, xs) = xs --> False ==>
       C(xa, C(x, xs)) = C(x, xs) --> False
val it = () : unit
- br impI 1;
Level 10
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs) |] ==>
       False
val it = () : unit
- br mp 1;
Level 11
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs) |] ==>
       ?P9(x, xs, xa) --> False
 2. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs) |] ==>
       ?P9(x, xs, xa)
val it = () : unit
- br allE 1;
Level 12
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs) |] ==>
       ALL xb. ?P10(x, xs, xa, xb)
 2. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs);
          ?P10
           (x, xs, xa, ?x10(x, xs, xa)) |] ==>
       ?P9(x, xs, xa) --> False
 3. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
```

```
         C(xa, C(x, xs)) = C(x, xs) |] ==>
      ?P9(x, xs, xa)
val it = () : unit
- ba 1;
Level 13
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs);
          C(?x10(x, xs, xa), xs) = xs -->
          False |] ==>
       ?P9(x, xs, xa) --> False
 2. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs) |] ==>
       ?P9(x, xs, xa)
val it = () : unit
- ba 1;
Level 14
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs) |] ==>
      C(?x10(x, xs, xa), xs) = xs
val it = () : unit
- br C_inject1 1;
Level 15
ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       [| ALL x. C(x, xs) = xs --> False;
          C(xa, C(x, xs)) = C(x, xs) |] ==>
       C(?x11(x, xs, xa),
         C(?x10(x, xs, xa), xs)) =
       C(?y11(x, xs, xa), xs)
val it = () : unit
- ba 1;
Level 16
ALL xs x. C(x, xs) = xs --> False
No subgoals!
val it = () : unit
- val C_l_not_l = result();
val C_l_not_l =
  "ALL xs x. C(x, xs) = xs --> False" : thm
```

Proving the theorem $\mathsf{Append}(\mathsf{Nil}, l) = l$ which without the definition of "Append" is the theorem

$$\mathsf{recl}_{l,\Pi x\ xs\ r.\ \mathsf{C}(x,r)}(\mathsf{Nil}) = l$$

demonstrating rule $\mathsf{recl}_{\mathsf{Nil}}$ is done as follows

```
-   goal List.thy "Append(Nil,l) = l";
Level 0
Append(Nil, l) = l
 1. Append(Nil, l) = l
val it = [] : thm list
-   by (rewtac app_def);
Level 1
Append(Nil, l) = l
 1. recl(Nil, l, %x xs r. C(x, r)) = l
val it = () : unit
-   by (resolve_tac [recl_Nil] 1);
Level 2
Append(Nil, l) = l
No subgoals!
val it = () : unit
- val app_Nil' = result();
val app_Nil' = "Append(Nil, ?l) = ?l" : thm
-   goal List.thy "?P";
Level 0
?P
 1. ?P
val it = [] : thm list
-   br app_Nil' 1;
Level 1
Append(Nil, ?l1) = ?l1
No subgoals!
val it = () : unit
-   by (rewrite_tac [app_def]);
Level 2
recl(Nil, ?l1, %x xs r. C(x, r)) = ?l1
No subgoals!
val it = () : unit
- val app_Nil = result();
val app_Nil =
  "recl(Nil, ?l, %x xs r. C(x, r)) = ?l" : thm
```

Note that the proof is in two parts like when using tactic `rewrite_tac` for other object logics.

Proving the theorem $\mathsf{Append}(\mathsf{C}(x, xs), ys) = \mathsf{C}(x, \mathsf{Append}(xs, ys))$ which without the definition of "Append" is the theorem

$$\mathsf{recl}_{ys,\Pi x\ xs\ r.\ \mathsf{C}(x,r)}(\mathsf{C}(x, xs)) = \mathsf{C}(x, \mathsf{recl}_{ys,\Pi x\ xs\ r.\ \mathsf{C}(x,r)}(xs))$$

demonstrating the rule $\mathsf{recl}_{\mathsf{C}}$ is done as follows

```
- goal List.thy
= "Append(C(x,xs),ys) = C(x,Append(xs,ys))";
Level 0
Append(C(x, xs), ys) = C(x, Append(xs, ys))
 1. Append(C(x, xs), ys) = C(x, Append(xs, ys))
val it = [] : thm list
-   by (rewtac app_def);
Level 1
Append(C(x, xs), ys) = C(x, Append(xs, ys))
 1. recl(C(x, xs), ys, %x xs r. C(x, r)) =
```

```
    C(x, recl(xs, ys, %x xs r. C(x, r)))
val it = () : unit
-   by (resolve_tac [recl_C] 1);
Level 2
Append(C(x, xs), ys) = C(x, Append(xs, ys))
No subgoals!
val it = () : unit
- val app_C' = result();
val app_C' =
  "Append(C(?x, ?xs), ?ys) =
   C(?x, Append(?xs, ?ys))" : thm
-   goal List.thy "?P";
Level 0
?P
 1. ?P
val it = [] : thm list
-   br app_C' 1;
Level 1
Append(C(?x1, ?xs1), ?ys1) =
C(?x1, Append(?xs1, ?ys1))
No subgoals!
val it = () : unit
-   by (rewrite_tac [app_def]);
Level 2
recl(C(?x1, ?xs1), ?ys1, %x xs r. C(x, r)) =
C(?x1, recl(?xs1, ?ys1, %x xs r. C(x, r)))
No subgoals!
val it = () : unit
- val app_C = result();
val app_C =
  "recl(C(?x, ?xs), ?ys, %x xs r. C(x, r)) =
   C(?x,
     recl(?xs, ?ys, %x xs r. C(x, r)))" : thm
```

Hereby all rules have been demonstrated. To demonstrate an application of a theorem on a decent value the theorem $((\mathsf{C}(\mathsf{S}(0)), \mathsf{Nil}) = \mathsf{Nil}) \rightarrow \bot$ is proven with

```
- goal List.thy "(C(S(0),Nil) = Nil) --> False";
Level 0
C(S(0), Nil) = Nil --> False
 1. C(S(0), Nil) = Nil --> False
val it = [] : thm list
-   br allE 1;
Level 1
C(S(0), Nil) = Nil --> False
 1. ALL x. ?P(x)
 2. ?P(?x) ==> C(S(0), Nil) = Nil --> False
val it = () : unit
-   br allE 1;
Level 2
C(S(0), Nil) = Nil --> False
 1. ALL x. ?P1(x)
 2. ?P1(?x1) ==> ALL x. ?P(x)
 3. ?P(?x) ==> C(S(0), Nil) = Nil --> False
val it = () : unit
-   br C_1_not_1 1;
Level 3
C(S(0), Nil) = Nil --> False
 1. ALL x. C(x, ?x1) = ?x1 --> False ==>
    ALL x. ?P(x)
 2. ?P(?x) ==> C(S(0), Nil) = Nil --> False
val it = () : unit
-   ba 1;
Level 4
C(S(0), Nil) = Nil --> False
 1. C(?x, ?x1) = ?x1 --> False ==>
    C(S(0), Nil) = Nil --> False
val it = () : unit
-   ba 1;
Level 5
C(S(0), Nil) = Nil --> False
No subgoals!
val it = () : unit
- val singleton_not_empty = result();
val singleton_not_empty =
  "C(S(0), Nil) = Nil --> False" : thm
```

B.10 Lists with Proofs

An object logic for reasoning about proofs including lists of elements of some type defined using the Peano axioms shall be created as specified in formal theory $\mathcal{LISTP}$.

The impelementation of proofs of lists of elements of some type will also include rewriting rules defining the length of a list and appending of lists.

B.10.1 Implementing Theory

The object logic `LISTP` representing formal theory $\mathcal{LISTP}$ can be found in appendix A.16

B.10.2 Examples

Some examples may show how the object logic can be used. Each rule shall be used at least once in the examples and the examples may show both some general examples and some specific application of a theorem on a specific list.

An example showing the use of the induction rule proving the inference rule inductl defined in formal theory $\mathcal{LISTP}$ is

```
- val [hb,hi] = goal LISTP.thy
=   "[| b : P(Nil); \
=   \   !! x xs u. u : P(xs) ==> \
=   \      f(x,xs,u) : P(C(x,xs)) |] ==> \
=   \?p : ALL l. P(l)";
std_in:5.1-9.21 Warning: binding not exhaustive
          hb :: hi :: nil = ...
Level 0
?p : ALL l. P(l)
 1. ?p : ALL l. P(l)
val hb = "b : P(Nil)  [b : P(Nil)]" : thm
val hi =
  "?u : P(?xs) ==>
   f(?x, ?xs, ?u) : P(C(?x, ?xs))
   [!!x xs u. u : P(xs) ==>
              f(x, xs, u) : P(C(x, xs))]" : thm
-     br induction 1;
Level 1
lrec(?b1, ?f1) : ALL l. P(l)
 1. ?b1 : P(Nil)
 2. ?f1 : ALL x xs. P(xs) --> P(C(x, xs))
val it = () : unit
-     br hb 1;
Level 2
lrec(b, ?f1) : ALL l. P(l)
 1. ?f1 : ALL x xs. P(xs) --> P(C(x, xs))
val it = () : unit
-     br allI 1;
Level 3
lrec(b, all x. ?f2(x)) : ALL l. P(l)
 1. !!x. ?f2(x) : ALL xs. P(xs) --> P(C(x, xs))
val it = () : unit
-     br allI 1;
Level 4
lrec(b, all x xa. ?f3(x, xa)) : ALL l. P(l)
 1. !!x xs. ?f3(x, xs) : P(xs) --> P(C(x, xs))
val it = () : unit
-     br impI 1;
Level 5
lrec(b, all x xa. lam xb. ?f4(x, xa, xb))
: ALL l. P(l)
 1. !!x xs xa.
       xa : P(xs) ==>
       ?f4(x, xs, xa) : P(C(x, xs))
val it = () : unit
-     br hi 1;
Level 6
lrec
 (b,
  all x xa. lam xb. f(x, xa, ?u5(x, xa, xb)))
: ALL l. P(l)
 1. !!x xs xa.
       xa : P(xs) ==> ?u5(x, xs, xa) : P(xs)
val it = () : unit
-     ba 1;
Level 7
lrec(b, all x xa. lam xb. f(x, xa, xb))
: ALL l. P(l)
No subgoals!
val it = () : unit
- val inductl = result();
val inductl =
  "[| ?b : ?P(Nil);
      !!x xs u. u : ?P(xs) ==>
                ?f(x, xs, u) :
                ?P(C(x, xs)) |] ==>
   lrec(?b, all x xa. lam xb. ?f(x, xa, xb)) :
   ALL l. ?P(l)" : thm
```

An example proving the inverse of inference rule $\mathsf{C}_{\mathsf{inj1}}$ being

$$\frac{\mathtt{p} : x = y}{\mathtt{idpeel(p,\ \%x.\ ideq(C(x,\ 1)))} : \mathsf{C}(x,\mathtt{1}) = \mathsf{C}(y,\mathtt{1})}\ \mathsf{C}_{\mathsf{surj1}}$$

is

```
- val [h] = goal LISTP.thy
= "p : x = y ==> ?q : C(x,1) = C(y,1)";
std_in:20.1-20.61 Warning:
  binding not exhaustive
          h :: nil = ...
Level 0
?q : C(x, 1) = C(y, 1)
 1. ?q : C(x, 1) = C(y, 1)
val h = "p : x = y  [p : x = y]" : thm
-   by ((rtac subst_context 1) THEN (rtac h 1));
Level 1
idpeel(p, %x. ideq(C(x, 1))) : C(x, 1) = C(y, 1)
No subgoals!
val it = () : unit
- val C_surj1 = result();
val C_surj1 =
  "?p : ?x = ?y ==>
   idpeel(?p, %x. ideq(C(x, ?1))) :
   C(?x, ?1) = C(?y, ?1)"
   : thm
```

An example proving the inverse of inference rule $\mathsf{C}_{\mathsf{inj2}}$ being

$$\frac{\mathtt{p} : xs = ys}{\mathtt{idpeel(p,\ \%x.\ ideq(C(a,\ x)))} : \mathsf{C}(\mathtt{a}, xs) = \mathsf{C}(\mathtt{a}, ys)}\ \mathsf{C}_{\mathsf{surj2}}$$

is

```
- val [h] = goal LISTP.thy
= "p : xs = ys ==> ?q : C(a,xs) = C(a,ys)";
std_in:23.1-23.65 Warning:
  binding not exhaustive
          h :: nil = ...
Level 0
?q : C(a, xs) = C(a, ys)
 1. ?q : C(a, xs) = C(a, ys)
val h = "p : xs = ys  [p : xs = ys]" : thm
-   by ((rtac subst_context 1) THEN (rtac h 1));
Level 1
idpeel(p, %x. ideq(C(a, x)))
: C(a, xs) = C(a, ys)
No subgoals!
val it = () : unit
- val C_surj2 = result();
val C_surj2 =
  "?p : ?xs = ?ys ==>
   idpeel(?p, %x. ideq(C(?a, x))) :
   C(?a, ?xs) = C(?a, ?ys)"
   : thm
```

Because natural deduction is used it is not possible to create a rule stating that if two lists are equal then the components of the list are equal. The other way, represented by inference rule

$$\frac{\mathtt{p} : x_1 = x_2 \qquad \mathtt{q} : xs_1 = xs_2}{\mathtt{idpeel(p,\ \%x.\ idpeel(q,\ \%xa.\ ideq(C(x,\ xa))))} : \mathsf{C}(x_1, xs_1) = \mathsf{C}(x_2, xs_2)}\ \mathsf{C}_{\mathsf{surj}}$$

can be proved by

```
- val [h1,h2] = goal LISTP.thy
= "[| p : x = (y::'a) ; \
= \q : xs = (ys::'a list) |] ==> \
= \?r : C(x,xs) = C(y,ys)";
std_in:26.1-27.77 Warning:
  binding not exhaustive
          h1 :: h2 :: nil = ...
Level 0
?r : C(x, xs) = C(y, ys)
 1. ?r : C(x, xs) = C(y, ys)
val h1 = "p : x = y  [p : x = y]" : thm
val h2 = "q : xs = ys  [q : xs = ys]" : thm
-   br subst 1;
Level 1
idpeel(?p1, %x. ?f1(x)) : C(x, xs) = C(y, ys)
 1. ?p1 : ?x1 = ?y1
 2. !!x. ?f1(x) : ?P1(x, x)
 3. !!q. q : ?P1(?x1, ?y1) ==>
         q : C(x, xs) = C(y, ys)
val it = () : unit
-   br h1 1;
Level 2
idpeel(p, %x. ?f1(x)) : C(x, xs) = C(y, ys)
 1. !!x. ?f1(x) : ?P1(x, x)
 2. !!q. q : ?P1(x, y) ==>
         q : C(x, xs) = C(y, ys)
val it = () : unit
-   ba 2;
Level 3
idpeel(p, %x. ?f1(x)) : C(x, xs) = C(y, ys)
 1. !!x. ?f1(x) : C(x, xs) = C(x, ys)
val it = () : unit
-   br C_surj2 1;
Level 4
idpeel
 (p, %x. idpeel(?p2(x), %xa. ideq(C(x, xa))))
: C(x, xs) = C(y, ys)
 1. !!x. ?p2(x) : xs = ys
val it = () : unit
-   br h2 1;
Level 5
idpeel(p, %x. idpeel(q, %xa. ideq(C(x, xa))))
: C(x, xs) = C(y, ys)
No subgoals!
val it = () : unit
- val C_surj = result();
val C_surj =
  "[| ?p : ?x = ?y; ?q : ?xs = ?ys |] ==>
   idpeel(?p,
          %x. idpeel(?q, %xa. ideq(C(x, xa)))) :
   C(?x, ?xs) = C(?y, ?ys)"
   : thm
```

An example proving theorem

$$\mathtt{lrec(all\ x.\ lam\ x.\ lneq(x),\ all\ x\ xa.\ lam\ xa.\ all\ xb.\ lam\ xb.\ (xa\ \hat{}\ x)\ `\ linj(xb))} : \forall l\ x.\ (\mathsf{C}(x, l) = l) \to \bot$$

demonstrating inference rules $\mathsf{C} \neq \mathsf{Nil}$ and $\mathsf{C}_{\mathsf{inj1}}$ is

```
- goal LISTP.thy
= "?p : ALL xs x . (C(x,xs) = xs) --> False";
Level 0
?p : ALL xs x. C(x, xs) = xs --> False
 1. ?p : ALL xs x. C(x, xs) = xs --> False
val it = [] : thm list
- br induction 1;
Level 1
lrec(?b1, ?f1)
: ALL xs x. C(x, xs) = xs --> False
 1. ?b1 : ALL x. C(x, Nil) = Nil --> False
 2. ?f1
    : ALL x xs.
         (ALL x. C(x, xs) = xs --> False) -->
         (ALL xa.
             C(xa, C(x, xs)) = C(x, xs) -->
             False)
val it = () : unit
- br allI 1;
Level 2
lrec(all x. ?f2(x), ?f1)
: ALL xs x. C(x, xs) = xs --> False
 1. !!x. ?f2(x) : C(x, Nil) = Nil --> False
 2. ?f1
    : ALL x xs.
         (ALL x. C(x, xs) = xs --> False) -->
         (ALL xa.
             C(xa, C(x, xs)) = C(x, xs) -->
             False)
val it = () : unit
- br impI 1;
Level 3
lrec(all x. lam xa. ?f3(x, xa), ?f1)
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xa.
       xa : C(x, Nil) = Nil ==>
       ?f3(x, xa) : False
 2. ?f1
    : ALL x xs.
         (ALL x. C(x, xs) = xs --> False) -->
         (ALL xa.
             C(xa, C(x, xs)) = C(x, xs) -->
             False)
val it = () : unit
- br C_neq_Nil 1;
Level 4
lrec(all x. lam xa. lneq(?p4(x, xa)), ?f1)
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xa.
       xa : C(x, Nil) = Nil ==>
       ?p4(x, xa)
       : C(?x4(x, xa), ?xs4(x, xa)) = Nil
 2. ?f1
    : ALL x xs.
         (ALL x. C(x, xs) = xs --> False) -->
         (ALL xa.
             C(xa, C(x, xs)) = C(x, xs) -->
             False)
val it = () : unit
- ba 1;
Level 5
lrec(all x. lam x. lneq(x), ?f1)
: ALL xs x. C(x, xs) = xs --> False
 1. ?f1
    : ALL x xs.
         (ALL x. C(x, xs) = xs --> False) -->
         (ALL xa.
             C(xa, C(x, xs)) = C(x, xs) -->
             False)
val it = () : unit
- br allI 1;
Level 6
lrec(all x. lam x. lneq(x), all x. ?f5(x))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x. ?f5(x)
         : ALL xs.
              (ALL x.
                  C(x, xs) = xs --> False) -->
              (ALL xa.
                  C(xa, C(x, xs)) = C(x, xs) -->
                  False)
val it = () : unit
- br allI 1;
Level 7
lrec
 (all x. lam x. lneq(x), all x xa. ?f6(x, xa))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs.
       ?f6(x, xs)
       : (ALL x. C(x, xs) = xs --> False) -->
         (ALL xa.
             C(xa, C(x, xs)) = C(x, xs) -->
             False)
val it = () : unit
- br impI 1;
Level 8
lrec
 (all x. lam x. lneq(x),
  all x xa. lam xb. ?f7(x, xa, xb))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa.
       xa : ALL x. C(x, xs) = xs --> False ==>
       ?f7(x, xs, xa)
       : ALL xa.
            C(xa, C(x, xs)) = C(x, xs) --> False
val it = () : unit
- br allI 1;
Level 9
lrec
 (all x. lam x. lneq(x),
  all x xa. lam xb. all xc. ?f8(x, xa, xb, xc))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa xb.
       xa : ALL x. C(x, xs) = xs --> False ==>
       ?f8(x, xs, xa, xb)
       : C(xb, C(x, xs)) = C(x, xs) --> False
val it = () : unit
- br impI 1;
Level 10
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xb.
        all xc. lam xd. ?f9(x, xa, xb, xc, xd))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?f9(x, xs, xa, xb, xc) : False
val it = () : unit
- br mp 1;
Level 11
lrec
 (all x. lam x. lneq(x),
```

```
  all x xa.
     lam xb.
        all xc.
           lam xd.
              ?f10(x, xa, xb, xc, xd) `
              ?a10(x, xa, xb, xc, xd))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?f10(x, xs, xa, xb, xc)
       : ?P10(x, xs, xa, xb, xc) --> False
 2. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?a10(x, xs, xa, xb, xc)
       : ?P10(x, xs, xa, xb, xc)
val it = () : unit
- br allE 1;
Level 12
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xb.
        all xc.
           lam xd.
              (?p11(x, xa, xb, xc, xd) ^
               ?z11(x, xa, xb, xc, xd)) `
              ?a10(x, xa, xb, xc, xd))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?p11(x, xs, xa, xb, xc)
       : ALL xd. ?P11(x, xs, xa, xb, xc, xd)
 2. !!x xs xa xb xc y.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs);
          y : ?P11
               (x, xs, xa, xb, xc,
                ?z11(x, xs, xa, xb, xc)) |] ==>
       y : ?P10(x, xs, xa, xb, xc) --> False
 3. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?a10(x, xs, xa, xb, xc)
       : ?P10(x, xs, xa, xb, xc)
val it = () : unit
- ba 1;
Level 13
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xb.
        all xc.
           lam xd.
              (xb ^ ?z11(x, xa, xb, xc, xd)) `
              ?a10(x, xa, xb, xc, xd))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa xb xc y.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs);
          y : C(?z11(x, xs, xa, xb, xc), xs) =
              xs -->
              False |] ==>
       y : ?P10(x, xs, xa, xb, xc) --> False
 2. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?a10(x, xs, xa, xb, xc)
       : ?P10(x, xs, xa, xb, xc)
val it = () : unit
- ba 1;
Level 14
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xb.
        all xc.
           lam xd.
              (xb ^ ?z12(x, xa, xb, xc, xd)) `
              ?a10(x, xa, xb, xc, xd))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?a10(x, xs, xa, xb, xc)
       : C(?z12(x, xs, xa, xb, xc), xs) = xs
val it = () : unit
- br C_inject1 1;
Level 15
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xb.
        all xc.
           lam xd.
              (xb ^ ?z12(x, xa, xb, xc, xd)) `
              linj(?p13(x, xa, xb, xc, xd)))
: ALL xs x. C(x, xs) = xs --> False
 1. !!x xs xa xb xc.
       [| xa : ALL x. C(x, xs) = xs --> False;
          xc : C(xb, C(x, xs)) = C(x, xs) |] ==>
       ?p13(x, xs, xa, xb, xc)
       : C(?x13(x, xs, xa, xb, xc),
           C(?z12(x, xs, xa, xb, xc), xs)) =
         C(?y13(x, xs, xa, xb, xc), xs)
val it = () : unit
- ba 1;
Level 16
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xa.
        all xb. lam xb. (xa ^ x) ` linj(xb))
: ALL xs x. C(x, xs) = xs --> False
No subgoals!
val it = () : unit
- val C_l_not_l = result();
val C_l_not_l =
  "lrec
    (all x. lam x. lneq(x),
     all x xa. lam xa. all xb. lam xb.
                (xa ^ x) ` linj(xb))
   : ALL xs x. C(x, xs) = xs --> False" : thm
```

Proving the theorem

$$\texttt{recNil} : \mathrm{Append}(\mathsf{Nil}, l) = l$$

which without the definition of "Append" is the theorem

$$\texttt{recNil} : \mathsf{recl}_{l,\Pi x\ xs\ r.\ \mathsf{C}(x,r)}(\mathsf{Nil}) = l$$

demonstrating rule $\mathsf{recl}_{\mathsf{Nil}}$ is done as follows

```
-   goal LISTP.thy "?p : Append(Nil,l) = l";
Level 0
?p : Append(Nil, l) = l
 1. ?p : Append(Nil, l) = l
val it = [] : thm list
-   by (rewtac app_def);
Level 1
?p : Append(Nil, l) = l
 1. ?p : recl(Nil, l, %x xs r. C(x, r)) = l
val it = () : unit
-   by (resolve_tac [recl_Nil] 1);
Level 2
recNil : Append(Nil, l) = l
No subgoals!
val it = () : unit
- val app_Nil' = result();
val app_Nil' =
  "recNil : Append(Nil, ?l) = ?l" : thm
-   goal LISTP.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br app_Nil' 1;
Level 1
recNil : Append(Nil, ?l1) = ?l1
No subgoals!
val it = () : unit
-   by (rewrite_tac [app_def]);
Level 2
recNil : recl(Nil, ?l1, %x xs r. C(x, r)) = ?l1
No subgoals!
val it = () : unit
- val app_Nil = result();
val app_Nil =
  "recNil :
   recl(Nil, ?l, %x xs r. C(x, r)) = ?l" : thm
```

Notice that the proof is in two parts like when using tactic `rewrite_tac` for other object logics.

Proving the theorem

$$\texttt{recC} : \mathrm{Append}(\mathsf{C}(x, xs), ys) = \mathsf{C}(x, \mathrm{Append}(xs, ys))$$

which without the definition of "Append" is the theorem

$$\texttt{recC} : \mathsf{recl}_{ys,\Pi x\ xs\ r.\ \mathsf{C}(x,r)}(\mathsf{C}(x, xs)) = \mathsf{C}(x, \mathsf{recl}_{ys,\Pi x\ xs\ r.\ \mathsf{C}(x,r)}(xs))$$

demonstrating the rule $\mathsf{recl}_{\mathsf{C}}$ is done as follows

```
- goal LISTP.thy
= "?p : Append(C(x,xs),ys) = \
= \C(x,Append(xs,ys))";
Level 0
?p : Append(C(x, xs), ys) = C(x, Append(xs, ys))
 1. ?p
    : Append(C(x, xs), ys) =
      C(x, Append(xs, ys))
val it = [] : thm list
-   by (rewtac app_def);
Level 1
?p : Append(C(x, xs), ys) = C(x, Append(xs, ys))
 1. ?p
    : recl(C(x, xs), ys, %x xs r. C(x, r)) =
      C(x, recl(xs, ys, %x xs r. C(x, r)))
val it = () : unit
-   by (resolve_tac [recl_C] 1);
Level 2
recC
: Append(C(x, xs), ys) = C(x, Append(xs, ys))
No subgoals!
val it = () : unit
- val app_C' = result();
val app_C' =
  "recC :
   Append(C(?x, ?xs), ?ys) =
   C(?x, Append(?xs, ?ys))" : thm
-   goal LISTP.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br app_C' 1;
Level 1
recC
: Append(C(?x1, ?xs1), ?ys1) =
  C(?x1, Append(?xs1, ?ys1))
No subgoals!
val it = () : unit
-   by (rewrite_tac [app_def]);
Level 2
recC
: recl(C(?x1, ?xs1), ?ys1, %x xs r. C(x, r)) =
  C(?x1, recl(?xs1, ?ys1, %x xs r. C(x, r)))
No subgoals!
val it = () : unit
- val app_C = result();
val app_C =
  "recC
   : recl(C(?x, ?xs), ?ys, %x xs r. C(x, r)) =
```

```
    C(?x,                                              recl(?xs, ?ys, %x xs r. C(x, r)))" : thm
```

Hereby all rules have been demonstrated. To demonstrate an application of a theorem on a decent value a theorem with the logic part $((\mathsf{C}(\mathsf{S}(0)), \mathsf{Nil}) = \mathsf{Nil}) \rightarrow \bot$ is proven with

```
- goal LISTP.thy
= "?p : (C(S(0),Nil) = Nil) --> False";
Level 0
?p : C(S(0), Nil) = Nil --> False
 1. ?p : C(S(0), Nil) = Nil --> False
val it = [] : thm list
-   br allE 1;
Level 1
?p1 ^ ?z1 : C(S(0), Nil) = Nil --> False
 1. ?p1 : ALL x. ?P1(x)
 2. !!y. y : ?P1(?z1) ==>
         y : C(S(0), Nil) = Nil --> False
val it = () : unit
-   br allE 1;
Level 2
?p2 ^ ?z2 ^ ?z1 : C(S(0), Nil) = Nil --> False
 1. ?p2 : ALL x. ?P2(x)
 2. !!y. y : ?P2(?z2) ==> y : ALL x. ?P1(x)
 3. !!y. y : ?P1(?z1) ==>
         y : C(S(0), Nil) = Nil --> False
val it = () : unit
-   br C_l_not_l 1;
Level 3
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xa.
        all xb. lam xb. (xa ^ x) ‘ linj(xb)) ^
?z2 ^
?z1
: C(S(0), Nil) = Nil --> False
 1. !!y. y : ALL x.
                C(x, ?z2) = ?z2 --> False ==>
         y : ALL x. ?P1(x)
 2. !!y. y : ?P1(?z1) ==>
         y : C(S(0), Nil) = Nil --> False
val it = () : unit
-   ba 1;
Level 4
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xa.
        all xb. lam xb. (xa ^ x) ‘ linj(xb)) ^
?z5 ^
?z1
: C(S(0), Nil) = Nil --> False
 1. !!y. y : C(?z1, ?z5) = ?z5 --> False ==>
         y : C(S(0), Nil) = Nil --> False
val it = () : unit
-   ba 1;
Level 5
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xa.
        all xb. lam xb. (xa ^ x) ‘ linj(xb)) ^
Nil ^
S(0)
: C(S(0), Nil) = Nil --> False
No subgoals!
val it = () : unit
- val singleton_not_empty = result();
val singleton_not_empty =
  "lrec
    (all x. lam x. lneq(x),
     all x xa. lam xa. all xb. lam xb.
                 (xa ^ x) ‘ linj(xb)) ^ Nil ^ S(0)
   : C(S(0), Nil) = Nil --> False" : thm
```

B.11 List with Proofs and Proof Reduction

An object logic for reducing proofs of propositions including lists of elements of some type shall be created as object logic PRLIST corresponding to formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ defined in section 5.3.3.

B.11.1 Implementing Theory

Formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ specifies that proof reduction is reduction of proofs that are proven using formal theory $\mathcal{LISTP}$ with the reduction methods specified in formal theory $\mathcal{PR}_{\mathcal{IFOLP}}$. Therefore object logic PRLIST must be built on top of object logics PR and LISTP.

Since there shall be two object logics implementing normalization of proofs including lists of elements of some type the common terms of these two object logics is placed in object logic LISTReduction presented in appendix A.17.

Object logic PRLIST for proof reduction of proofs including lists of elements of some type implements the parts of formal theory $\mathcal{PR}_{\mathcal{IFOLP},\mathcal{LIST}}$ that is not shared with the formal theory $\mathcal{PE}_{\mathcal{IFOLP},\mathcal{LIST}}$. Object logic PRLIST is implemented in appendix A.18.

B.11.2 Examples

An example demonstrating context rule $\mathsf{rec_{Nil}Cl}$ is

```
- goal PRLIST.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
- br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
- br app_Nil 1;
Level 2
?p : recl(Nil, ?l2, %x xs r. C(x, r)) = ?l2
 1. recNil >> ?p :
       recl(Nil, ?l2, %x xs r. C(x, r)) = ?l2
val it = () : unit
- br recl_NilCI 1;
Level 3
recNil : recl(Nil, ?l2, %x xs r. C(x, r)) = ?l2
No subgoals!
val it = () : unit
- val app_NilRed = result();
val app_NilRed =
  "recNil : recl(Nil, ?l, %x xs r. C(x, r)) =
            ?l" : thm
```

An example demonstrating context rule $\mathsf{rec_C Cl}$ is

```
-   goal PRLIST.thy "?p :?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br app_C 1;
Level 2
?p
: recl(C(?x2, ?xs2), ?ys2, %x xs r. C(x, r)) =
  C(?x2, recl(?xs2, ?ys2, %x xs r. C(x, r)))
 1. recC >> ?p :
       recl
        (C(?x2, ?xs2), ?ys2, %x xs r. C(x, r)) =
       C(?x2,
         recl(?xs2, ?ys2, %x xs r. C(x, r)))
val it = () : unit
-   br recl_CCI 1;
Level 3
recC
: recl(C(?x2, ?xs2), ?ys2, %x xs r. C(x, r)) =
  C(?x2, recl(?xs2, ?ys2, %x xs r. C(x, r)))
No subgoals!
val it = () : unit
- val app_CRed = result();
val app_CRed =
  "recC
    : recl(C(?x, ?xs), ?ys, %x xs r. C(x, r)) =
      C(?x,
        recl(?xs, ?ys, %x xs r. C(x, r)))" : thm
```

An example demonstrating contextrules $\mathsf{inductionCl}$, $\mathsf{C} \neq \mathsf{NilCl}$, $\mathsf{C_{inj1}}$ followed by reduction rules $\mathsf{inductionR_C}$ and $\mathsf{inductionR_{Nil}}$ is reduction of a proof of proposition

$$\mathsf{C}(\mathsf{S}(0), \mathsf{C}(\mathsf{S}(0), \mathsf{Nil})) = \mathsf{C}(\mathsf{S}(0), \mathsf{Nil}) \rightarrow \bot$$

A part of the reduction is shown below. States that results of an application of a $\mathcal{PR}_{\mathcal{IFOLP}}$ rule is cutted out since the proof session prints get enormous because a lot of context rule applications are used to state that a proof subtree does not contain any redexes.

First a proof of the proposition is created by applying theorem C_1_not_1 on list $\mathsf{C}(\mathsf{S}(0), \mathsf{Nil})$ and number $\mathsf{S}(0)$ illustrated by the proving session

```
-  goal LISTP.thy
= "?p : (C(S(0),C(S(0),Nil)) = C(S(0),Nil)) \
  \     --> False";
Level 0
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. ?p
    : C(S(0), C(S(0), Nil)) = C(S(0), Nil) -->
      False
val it = [] : thm list
-   br allE 1;
Level 1
?p1 ^ ?z1
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. ?p1 : ALL x. ?P1(x)
 2. !!y. y : ?P1(?z1) ==>
         y : C(S(0), C(S(0), Nil)) =
             C(S(0), Nil) -->
             False
val it = () : unit
-   br allE 1;
Level 2
?p2 ^ ?z2 ^ ?z1
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. ?p2 : ALL x. ?P2(x)
 2. !!y. y : ?P2(?z2) ==> y : ALL x. ?P1(x)
 3. !!y. y : ?P1(?z1) ==>
         y : C(S(0), C(S(0), Nil)) =
             C(S(0), Nil) -->
             False
val it = () : unit
-   br C_l_not_l 1;
Level 3
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xa.
        all xb. lam xb. (xa ^ x) ‘ linj(xb)) ^
?z2 ^
?z1
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. !!y. y : ALL x.
                C(x, ?z2) = ?z2 --> False ==>
         y : ALL x. ?P1(x)
 2. !!y. y : ?P1(?z1) ==>
         y : C(S(0), C(S(0), Nil)) =
             C(S(0), Nil) -->
             False
val it = () : unit
-   ba 1;
Level 4
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xa.
        all xb. lam xb. (xa ^ x) ‘ linj(xb)) ^
?z5 ^
?z1
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. !!y. y : C(?z1, ?z5) = ?z5 --> False ==>
         y : C(S(0), C(S(0), Nil)) =
             C(S(0), Nil) -->
             False
val it = () : unit
-   ba 1;
Level 5
lrec
 (all x. lam x. lneq(x),
  all x xa.
     lam xa.
        all xb. lam xb. (xa ^ x) ‘ linj(xb)) ^
C(S(0), Nil) ^
S(0)
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
No subgoals!
val it = () : unit
- val twoelt_not_oneelt = result();
val twoelt_not_oneelt =
  "lrec
    (all x. lam x. lneq(x),
     all x xa. lam xa. all xb. lam xb.
         (xa ^ x) ‘ linj(xb)) ^ C(S(0), Nil) ^
    S(0) : C(S(0), C(S(0), Nil)) = C(S(0), Nil)
          --> False" : thm
```

The created proof `twoelt_not_oneelt` is then reduced in a proving session from which an abstract is below.

```
-   goal PRLIST.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br twoelt_not_oneelt 1;
Level 2
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. lrec
     (all x. lam x. lneq(x),
      all x xa.
         lam xa.
            all xb.
               lam xb. (xa ^ x) ‘ linj(xb)) ^
    C(S(0), Nil) ^
    S(0)
       >> ?p :
       C(S(0), C(S(0), Nil)) = C(S(0), Nil) -->
       False
val it = () : unit
-   br allCE 1;
-   br allCE 1;
Level 4
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. lrec
     (all x. lam x. lneq(x),
      all x xa.
         lam xa.
            all xb.
               lam xb. (xa ^ x) ‘ linj(xb))
```

```
       >> ?g4 : ALL b. ?R4(b)
...
-   br inductionCI 1;
Level 5
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. all x. lam x. lneq(x) >> ?b'6 : ?R4(Nil)
 2. all x xa.
       lam xa.
          all xb. lam xb. (xa ^ x) ‘ linj(xb)
       >> ?f'6 :
       ALL x xs. ?R4(xs) --> ?R4(C(x, xs))
 3. lrec(?b'6, ?f'6) ^ C(S(0), Nil) !> ?g3 :
       ?R4(C(S(0), Nil))
...
-   br allCI 1;
-   br impCI 1;
Level 7
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. !!y ya.
       ya : ?P11(Nil, y) ==>
       lneq(ya) >> ?g9(y, ya) : ?Q10(Nil, y)
...
-   br C_neq_NilCI 1;
Level 8
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. !!y ya.
       ya : ?P11(Nil, y) ==>
       ya >> ?p'12(y, ya) :
          C(?x12(y, ya), ?xs12(y, ya)) = Nil
...
-   bv 1;
-   br allCI 1;
-   br allCI 1;
-   br impCI 1;
-   br allCI 1;
-   br impCI 1;
-   br impCE 1;
-   br allCE 1;
-   bv 1;
-   bt 1;
Level 18
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 2. !!y ya yb yc yd.
       [| yb
          : ALL z.
               C(?x16(ya, z), ?xs15(ya, z)) =
               ya -->
               ?Q10(ya, z);
          yd
          : C(?x16(C(y, ya), yc),
              ?xs15(C(y, ya), yc)) =
            C(y, ya) |] ==>
       linj(yd) >> ?e2'25(y, ya, yb, yc, yd) :
          ?Q31(y, y, ya, yb, yc, yd)
...
-   br C_inject1CI 2;
Level 19
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 2. !!y ya yb yc yd.
       [| yb
          : ALL z.
               C(?x16(ya, z), ?xs15(ya, z)) =
               ya -->
               ?Q10(ya, z);
          yd
          : C(?x16(C(y, ya), yc),
              ?xs15(C(y, ya), yc)) =
            C(y, ya) |] ==>
       yd >> ?p'34(y, ya, yb, yc, yd) :
          C(?x34(y, ya, yb, yc, yd),
            ?xs36(y, y, ya, yb, yc, yd)) =
          C(?y34(y, ya, yb, yc, yd),
            ?ys35(y, y, ya, yb, yc, yd))
...
-   bv 2;
-   bt 2;
Level 21
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 2. lrec
     (all x. lam x. lneq(x),
      all x xa.
         lam xa.
           all xb.
              lam xb. (xa ^ x) ‘ linj(xb)) ^
    C(S(0), Nil)
       !> ?g3 :
       ALL z.
          C(?x16(C(S(0), Nil), z),
            ?xs15(C(S(0), Nil), z)) =
          C(S(0), Nil) -->
          ?Q10(C(S(0), Nil), z)
...
-   br inductionRC 2;
Level 22
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 2. (all x xa.
        lam xa.
           all xb. lam xb. (xa ^ x) ‘ linj(xb) ^
     S(0) ^
     Nil) ‘
    (lrec
      (all x. lam x. lneq(x),
       all x xa.
          lam xa.
             all xb.
                lam xb. (xa ^ x) ‘ linj(xb)) ^
      Nil)
       >> ?g3 :
       ALL z.
          C(?x16(C(S(0), Nil), z),
            ?xs15(C(S(0), Nil), z)) =
          C(S(0), Nil) -->
          ?Q10(C(S(0), Nil), z)
...
-   br impCE 2;
-   br allCE 2;
-   br allCE 2;
-   br allCI 2;
-   br allCI 2;
-   br impCI 2;
-   br allCI 2;
-   br impCI 2;
-   br impCE 2;
```

```
-   br allCE 2;
-   bv 2;
-   bt 2;
Level 34
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 3. !!y ya yb yc yd.
       [| yb : ALL b. ?R74(y, b, ya);
          yd : ?P61(y, ya, yc) |] ==>
       linj(yd) >> ?e2'62(y, ya, yb, yc, yd) :
          ?Q68(y, y, ya, yb, yc, yd)
...
-   br C_inject1CI 3;
Level 35
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 3. !!y ya yb yc yd.
       [| yb : ALL b. ?R74(y, b, ya);
          yd : ?P61(y, ya, yc) |] ==>
       yd >> ?p'75(y, ya, yb, yc, yd) :
          C(?x75(y, ya, yb, yc, yd),
            ?xs77(y, y, ya, yb, yc, yd)) =
          C(?y75(y, ya, yb, yc, yd),
            ?ys76(y, y, ya, yb, yc, yd))
...
-   bv 3;
-   bt 3;
-   br allR 3;
-   br allCI 3;
-   br impCI 3;
-   br allCI 3;
-   br impCI 3;
-   br impCE 3;
-   br allCE 3;
-   bv 3;
-   bt 3;
Level 46
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 4. !!y ya yb yc.
       [| ya : ALL b. ?R117(S(0), y, b);
          yc : ?P107(S(0), y, yb) |] ==>
       linj(yc) >> ?e2'110(y, ya, yb, yc) :
          ?Q114(y, ya, yb, yc, S(0))
...
-   br C_inject1CI 4;
Level 47
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 4. !!y ya yb yc.
       [| ya : ALL b. ?R117(S(0), y, b);
          yc : ?P107(S(0), y, yb) |] ==>
       yc >> ?p'121(y, ya, yb, yc) :
          C(?x121(y, ya, yb, yc),
            ?xs123(y, ya, yb, yc, S(0))) =
          C(?y121(y, ya, yb, yc),
            ?ys122(y, ya, yb, yc, S(0)))
...
-   bv 4;
-   bt 4;
-   br allR 6;
-   br impCI 6;
-   br allCI 6;
-   br impCI 6;
-   br impCE 6;
-   br allCE 6;
-   bv 6;
-   bt 6;
Level 57
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 7. !!y ya yb.
       [| y : ALL b. ?R154(Nil, b);
          yb : ?P144(Nil, ya) |] ==>
       linj(yb) >> ?e2'147(y, ya, yb) :
          ?Q151(y, ya, yb, S(0))
...
-   br C_inject1CI 7;
Level 58
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 7. !!y ya yb.
       [| y : ALL b. ?R154(Nil, b);
          yb : ?P144(Nil, ya) |] ==>
       yb >> ?p'158(y, ya, yb) :
          C(?x158(y, ya, yb),
            ?xs160(y, ya, yb, S(0))) =
          C(?y158(y, ya, yb),
            ?ys159(y, ya, yb, S(0)))
...
-   bv 7;
-   bt 7;
-   br allCE 9;
Level 61
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 9. lrec
     (all x. lam x. lneq(x),
      all x xa.
         lam xa.
            all xb.
               lam xb. (xa ^ x) ` linj(xb))
       >> ?g172 : ALL b. ?R172(b)
...
-   br inductionCI 9;
Level 62
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 9. all x. lam x. lneq(x) >> ?b'173 : ?R172(Nil)
...
-   br allCI 9;
-   br impCI 9;
Level 64
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 9. !!y ya.
       ya : ?P178(Nil, y) ==>
       lneq(ya) >> ?g176(y, ya) : ?Q177(Nil, y)
...
-   br C_neq_NilCI 9;
Level 65
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 9. !!y ya.
```

```
       ya : ?P178(Nil, y) ==>
       ya >> ?p'179(y, ya) :
          C(?x179(y, ya), ?xs179(y, ya)) = Nil
...
-   bv 9;
-   br allCI 9;
-   br allCI 9;
-   br impCI 9;
-   br allCI 9;
-   br impCI 9;
-   br impCE 9;
-   br allCE 9;
-   bv 9;
-   bt 9;
Level 75
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 10. !!y ya yb yc yd.
        [| yb
           : ALL z.
                C(?x183(ya, z), ?xs182(ya, z)) =
                ya -->
                ?Q177(ya, z);
           yd
           : C(?x183(C(y, ya), yc),
               ?xs182(C(y, ya), yc)) =
             C(y, ya) |] ==>
        linj(yd) >> ?e2'192(y, ya, yb, yc, yd) :
           ?Q198(y, y, ya, yb, yc, yd)
...
-   br C_inject1CI 10;
Level 76
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 10. !!y ya yb yc yd.
        [| yb
           : ALL z.
                C(?x183(ya, z), ?xs182(ya, z)) =
                ya -->
                ?Q177(ya, z);
           yd
           : C(?x183(C(y, ya), yc),
               ?xs182(C(y, ya), yc)) =
             C(y, ya) |] ==>
        yd >> ?p'201(y, ya, yb, yc, yd) :
           C(?x201(y, ya, yb, yc, yd),
             ?xs203(y, y, ya, yb, yc, yd)) =
           C(?y201(y, ya, yb, yc, yd),
             ?ys202(y, y, ya, yb, yc, yd))
...
-   bv 10;
-   bt 10;
Level 78
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 10. lrec
      (all x. lam x. lneq(x),
       all x xa.
          lam xa.
             all xb.
                lam xb. (xa ^ x) ` linj(xb)) ^
     Nil
        !> ?e2'43 :
        ALL z.
           C(?x183(Nil, z), ?xs182(Nil, z)) =
           Nil -->
           ?Q177(Nil, z)
...
-   br inductionRNil 10;
Level 79
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 10. all x. lam x. lneq(x) >> ?e2'43 :
        ALL z.
           C(?x183(Nil, z), ?xs182(Nil, z)) =
           Nil -->
           ?Q177(Nil, z)
...
-   br allCI 10;
-   br impCI 10;
Level 81
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 10. !!y ya.
        ya
        : C(?x183(Nil, y), ?xs182(Nil, y)) =
          Nil ==>
        lneq(ya) >> ?g211(y, ya) : ?Q177(Nil, y)
...
-   br C_neq_NilCI 10;
Level 82
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 10. !!y ya.
        ya
        : C(?x183(Nil, y), ?xs182(Nil, y)) =
          Nil ==>
        ya >> ?p'212(y, ya) :
           C(?x212(y, ya), ?xs212(y, ya)) = Nil
...
-   bv 10;
-   br impR 11;
-   br allCI 11;
-   br impCI 11;
-   br impCE 11;
-   br allCE 11;
-   br allCI 11;
-   br impCI 11;
Level 90
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 11. !!y ya yb yc.
        [| ya
           : C(?x16(C(S(0), Nil), y),
               ?xs15(C(S(0), Nil), y)) =
             C(S(0), Nil);
           yc : ?P227(y, ya, yb) |] ==>
        lneq(yc) >> ?g227(y, ya, yb, yc) :
           ?Q227(y, ya, yb)
...
-   br C_neq_NilCI 11;
Level 91
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 11. !!y ya yb yc.
        [| ya
```

```
          : C(?x16(C(S(0), Nil), y),
              ?xs15(C(S(0), Nil), y)) =
            C(S(0), Nil);
          yc : ?P227(y, ya, yb) |] ==>
       yc >> ?p'228(y, ya, yb, yc) :
          C(?x228(y, ya, yb, yc),
            ?xs228(y, ya, yb, yc)) =
          Nil
...
-   bv 11;
-   br allR 11;
-   br impCI 11;
Level 94
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 11. !!y ya yb.
        [| ya
          : C(?x16(C(S(0), Nil), y),
              ?xs15(C(S(0), Nil), y)) =
            C(S(0), Nil);
          yb : ?P241(y, ya, S(0)) |] ==>
       lneq(yb) >> ?g239(y, ya, yb) :
          ?Q240(y, ya, S(0))
...
-   br C_neq_NilCI 11;
Level 95
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 11. !!y ya yb.
        [| ya
          : C(?x16(C(S(0), Nil), y),
              ?xs15(C(S(0), Nil), y)) =
            C(S(0), Nil);
          yb : ?P241(y, ya, S(0)) |] ==>
       yb >> ?p'242(y, ya, yb) :
          C(?x242(y, ya, yb),
            ?xs242(y, ya, yb)) =
          Nil
...
-   bv 11;
Level 96
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 13. !!y ya.
        ya
        : C(?x16(C(S(0), Nil), y),
            ?xs15(C(S(0), Nil), y)) =
          C(S(0), Nil) ==>
        linj(ya) >> ?e2'220(y, ya) :
          ?Q224(y, ya, S(0))
...
-   br C_inject1CI 13;
-   bv 13;
-   br impR 13;
Level 99
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 13. !!y ya.
        ya
        : C(?x16(C(S(0), Nil), y),
            ?xs15(C(S(0), Nil), y)) =
          C(S(0), Nil) ==>
        lneq(linj(ya)) >> ?g219(y, ya) :
          ?Q10(C(S(0), Nil), y)
...
-   br C_neq_NilCI 13;
Level 100
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 13. !!y ya.
        ya
        : C(?x16(C(S(0), Nil), y),
            ?xs15(C(S(0), Nil), y)) =
          C(S(0), Nil) ==>
        linj(ya) >> ?p'257(y, ya) :
          C(?x257(y, ya), ?xs257(y, ya)) = Nil
...
-   br C_inject1CI 13;
Level 101
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 13. !!y ya.
        ya
        : C(?x16(C(S(0), Nil), y),
            ?xs15(C(S(0), Nil), y)) =
          C(S(0), Nil) ==>
        ya >> ?p'259(y, ya) :
          C(?x259(y, ya),
            C(?x257(y, ya), ?xs257(y, ya))) =
          C(?y259(y, ya), Nil)
...
-   bv 13;
-   br allR 14;
-   br impCI 14;
Level 104
lam x. ?g263(x)
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 14. !!y. y : ?P265(S(0)) ==>
          lneq(linj(y)) >> ?g263(y) :
            ?Q264(S(0))
..
-   br C_neq_NilCI 14;
Level 105
lam x. lneq(?p'266(x))
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 14. !!y. y : ?P265(S(0)) ==>
          linj(y) >> ?p'266(y) :
            C(?x266(y), ?xs266(y)) = Nil
...
-   br C_inject1CI 14;
Level 106
lam x. lneq(linj(?p'268(x)))
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
...
 14. !!y. y : ?P265(S(0)) ==>
          y >> ?p'268(y) :
            C(?x268(y),
              C(?x266(y), ?xs266(y))) =
            C(?y268(y), Nil)
...
-   bv 14;
Level 107
lam x. lneq(linj(x))
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. !!y ya yb yc yd ye.
       [| yb
```

```
        : ALL z.
             C(?x16(ya, z), ya) = ya -->
             ?Q10(ya, z);
        yd
        : C(?x16(C(y, ya), yc), C(y, ya)) =
          C(y, ya);
        ye
        : C(?x16(ya, y), ya) = ya -->
          ?Q10(ya, y) |] ==>
     ye : C(y, ya) = ya --> ?Q30(y, y, ya, yc)
2. !!y ya yb yc yd ye.
     [| yb : ALL b. ?R96(y, ya, b);
        yd
        : C(?x86(y, ya, yc),
            ?xs85(y, y, ya, yc)) =
          C(?y84(y, ya, yc),
            ?ys83(y, y, ya, yc));
        ye : ?R96(y, ya, y) |] ==>
     ye
     : ?xs85(y, y, ya, yc) =
       ?ys83(y, y, ya, yc) -->
       ?Q67(y, y, ya, yc)
3. !!y ya yb yc yd.
     [| ya : ALL b. ?R117(S(0), y, b);
        yc
        : C(?x128(S(0), y, yb),
            ?xs127(S(0), y, yb)) =
          C(?y126(S(0), y, yb),
            ?ys125(S(0), y, yb));
        yd : ?R117(S(0), y, S(0)) |] ==>
     yd
     : ?xs131(y, yb, S(0)) =
       ?ys129(y, yb, S(0)) -->
       ?Q115(y, yb, S(0))
4. !!y. y : ALL z.
              (ALL b. ?R117(?a89, z, b)) -->
              (ALL za.
                  C(?x128(?a89, z, za),
                    ?xs127(?a89, z, za)) =
                  C(?y126(?a89, z, za),
                    ?ys125(?a89, z, za)) -->
                  ?Q106(?a89, z, za)) ==>
       y : ALL z.
              (ALL b. ?R95(?a89, z, b)) -->
              (ALL za.
                  C(?x94(?a89, z, za),
                    ?xs93(?a89, z, za)) =
                  C(?y92(?a89, z, za),
                    ?ys91(?a89, z, za)) -->
                  ?Q90(?a89, z, za))
5. !!y. y : ALL z.
              (ALL b. ?R96(S(0), z, b)) -->
              (ALL za.
                  C(?x86(S(0), z, za),
                    ?xs85
                      (S(0), S(0), z, za)) =
                  C(?y84(S(0), z, za),
                    ?ys83
                      (S(0), S(0), z, za)) -->
                  ?Q67(S(0), S(0), z, za)) ==>
       y : ALL b. ?R50(S(0), b)
6. !!y ya yb yc.
     [| y : ALL b. ?R154(Nil, b);
        yb
        : C(?x165(Nil, ya), ?xs164(Nil, ya)) =
          C(?y163(Nil, ya), ?ys162(Nil, ya));
        yc : ?R154(Nil, S(0)) |] ==>
        yc
        : ?xs168(ya, S(0)) = ?ys166(ya, S(0)) -->
          ?Q152(ya, S(0))
7. !!y. y : (ALL b. ?R154(?a135, b)) -->
            (ALL z.
                C(?x165(?a135, z),
                  ?xs164(?a135, z)) =
                C(?y163(?a135, z),
                  ?ys162(?a135, z)) -->
                ?Q143(?a135, z)) ==>
        y : ?P135(?a135)
8. !!y. y : ?R50(S(0), Nil) ==>
        y : ?Q48(Nil) -->
            (ALL z.
                C(?x47(Nil, z), C(S(0), Nil)) =
                C(S(0), Nil) -->
                ?Q45(Nil, z))
9. !!y ya yb yc yd ye.
     [| yb
        : ALL z.
             C(?x183(ya, z), ?xs182(ya, z)) =
             ya -->
             ?Q177(ya, z);
        yd
        : C(?x183(C(y, ya), yc),
            ?xs182(C(y, ya), yc)) =
          C(y, ya);
        ye
        : C(?x183(ya, y), ?xs182(ya, y)) =
          ya -->
          ?Q177(ya, y) |] ==>
     ye
     : ?xs206(y, y, ya, yc) = ya -->
       ?Q197(y, y, ya, yc)
10. !!y. y : ALL z.
               C(?x183(Nil, z),
                 ?xs182(Nil, z)) =
               Nil -->
               ?Q177(Nil, z) ==>
        y : ?P172(Nil)
11. !!y ya yb.
      [| ya
         : C(?x16(C(S(0), Nil), y),
             C(S(0), Nil)) =
           C(S(0), Nil);
         yb
         : C(?x246(y, ya, ?a235(y, ya)),
             ?xs245(y, ya, ?a235(y, ya))) =
           Nil -->
           ?Q240(y, ya, ?a235(y, ya)) |] ==>
      yb
      : C(?x238(y, ya, ?a235(y, ya)),
          ?xs237(y, ya, ?a235(y, ya))) =
        Nil -->
        ?Q236(y, ya, ?a235(y, ya))
12. !!y ya yb.
      [| ya
         : C(?x16(C(S(0), Nil), y),
             C(S(0), Nil)) =
           C(S(0), Nil);
         yb
         : C(?x234(y, ya, S(0)),
             ?xs233(y, ya, S(0))) =
           Nil -->
           ?Q227(y, ya, S(0)) |] ==>
      yb
      : C(S(0), Nil) = Nil --> ?Q225(y, S(0))
```

```
 13. !!y. y : ALL z.
                C(?x16(C(S(0), Nil), z),
                  C(S(0), Nil)) =
                C(S(0), Nil) -->
                ?Q10(C(S(0), Nil), z) ==>
          y : ALL b. ?R5(C(S(0), Nil), b)
 14. !!y. y : C(?x273(?a262),
                C(?x272(?a262),
                  ?xs271(?a262))) =
              C(?y270(?a262), Nil) -->
              ?Q264(?a262) ==>
          y : ?P262(?a262)
 15. !!y. y : ?R5(C(S(0), Nil), S(0)) ==>
          y : C(S(0), C(S(0), Nil)) =
              C(S(0), Nil) -->
              False

Flex-flex pairs:
...
val it = () : unit
-   by ((atac 15) THEN (atac 14) THEN (atac 13)
=       THEN (atac 12) THEN (atac 11) THEN
=       (atac 10) THEN (atac 9) THEN (atac 8)
=       THEN (atac 7) THEN (atac 6) THEN
=       (atac 5) THEN (atac 4) THEN (atac 3)
=       THEN (atac 2) THEN (atac 1));
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
Enter MATCH
...
***Unification bound exceeded
Enter MATCH
...
Enter MATCH
...
Level 108
lam x. lneq(linj(x))
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
No subgoals!

Flex-flex pairs:
...
val it = () : unit
- val test = result();
val test =
  "lam x. lneq(linj(x)) :
   C(S(0), C(S(0), Nil)) = C(S(0), Nil) -->
   False" : thm
```

The above proving session reveals the same problems as the reduction example in section B.7.2, exhibiting that Isabelle may do some unwanted instantiations. These instantiations are then done as the last part of the proof. Another problem revealed by the above proving session is that unification may be a problem since Isabelle can not handle it.

B.12 List with Proofs and Program Evaluation

An object logic for evaluating proofs including lists of elements of some type as the proofs were programs is created as object logic PELIST corresponding to formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ defined in section 5.3.3.

B.12.1 Implementing Theory

Formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ specifies that program evaluation is reduction of proofs that are proven using formal theory $\mathcal{LISTP}$ with the reduction methods specified in the formal theory $\mathcal{PE}_{\mathcal{IFOLP}}$. Therefore object logic PELIST must be built on top of object logics PE and LISTP. Since object logic LISTP also includes natural number object logic NATP this object logic and the program evaluation object logic PENAT must also be included in the object logic PELIST.

As the second of the two object logics implementing normalization of proofs including lists of elements of some type also this object logic's common terms with object logic PRLIST is placed in object logic LISTReduction presented in section A.17.

Object logic PELIST for evaluation of proofs, including lists of elements of some type, as the proofs

were programs implements the parts of formal theory $\mathcal{PE}_{\mathcal{IFOLP},\,\mathcal{LIST}}$ that is not shared with formal theory $\mathcal{PR}_{\mathcal{IFOLP},\,\mathcal{LIST}}$. Object logic PELIST is in appendix A.19.

B.12.2 Examples

The only thing to demonstrate of object logic PELIST is the rules added to existing object logics which is rules $\mathsf{inductionR}_{\mathsf{Nil}}$ and $\mathsf{inductionR}_{\mathsf{C}}$. These rules are demonstrated with object logic PRLIST.

To demonstrate the difference between proof reduction and program evaluation of proofs including lists the proof of a theorem with logic part

$$\mathsf{C}(\mathsf{S}(0),\mathsf{C}(\mathsf{S}(0),\mathsf{Nil})) = \mathsf{C}(\mathsf{S}(0),\mathsf{Nil}) \to \bot$$

that is reduced in section B.11.2 is evaluated as a program below. Parts not dealing with lists are cutted out to save space.

```
-   goal PELIST.thy "?p : ?P";
Level 0
?p : ?P
 1. ?p : ?P
val it = [] : thm list
-   br redE 1;
Level 1
?p : ?P
 1. ?a1 : ?P
 2. ?a1 >> ?p : ?P
val it = () : unit
-   br twoelt_not_oneelt 1;
Level 2
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. lrec
     (all x. lam x. lneq(x),
      all x xa.
         lam xa.
            all xb.
               lam xb. (xa ^ x) ‘ linj(xb)) ^
    C(S(0), Nil) ^
    S(0)
       >> ?p :
       C(S(0), C(S(0), Nil)) = C(S(0), Nil) -->
       False
val it = () : unit
-   br allCE 1;
-   br allCE 1;
-   bta 1;
Level 5
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. lrec
     (all x. lam x. lneq(x),
      all x xa.
         lam xa.
            all xb.
               lam xb. (xa ^ x) ‘ linj(xb)) ^
    C(S(0), Nil)
       !> ?g3 : ?R4(C(S(0), Nil))
...
-   br inductionRC 1;
Level 6
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. (all x xa.
        lam xa.
           all xb. lam xb. (xa ^ x) ‘ linj(xb) ^
     S(0) ^
     Nil) ‘
    (lrec
      (all x. lam x. lneq(x),
       all x xa.
          lam xa.
             all xb.
                lam xb. (xa ^ x) ‘ linj(xb)) ^
     Nil)
       >> ?g3 : ?R4(C(S(0), Nil))
...
-   br impCE 1;
-   br allCE 1;
-   by flexflex_tac;
-   br allCE 1;
-   bta 1;
-   br allR 1;
-   by flexflex_tac;
-   bta 1;
-   ba 1;
-   ba 1;
-   by flexflex_tac;
-   br allR 1;
-   bta 1;
-   ba 1;
-   by flexflex_tac;
-   ba 1;
-   br allCE 1;
-   by flexflex_tac;
-   bta 1;
Level 25
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. lrec
     (all x. lam x. lneq(x),
      all x xa.
         lam xa.
            all xb.
               lam xb. (xa ^ x) ‘ linj(xb)) ^
    Nil
       !> ?e2'8 : ?R26(Nil)
```

```
...
-   br inductionRNil 1;
Level 26
?p
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
 1. all x. lam x. lneq(x) >> ?e2'8 : ?R26(Nil)
...
-   bta 1;
-   ba 1;
-   br impR 1;
-   bta 1;
-   ba 1;
-   br allR 1;
-   by flexflex_tac;
-   bta 1;
-   ba 1;
-   ba 1;
Level 36
lam x. (all x. lam x. lneq(x) ^ S(0)) ‘ linj(x)
: C(S(0), C(S(0), Nil)) = C(S(0), Nil) --> False
No subgoals!
val it = () : unit
- val redExample = result();
val redExample =
  "lam x. (all x. lam x. lneq(x) ^ S(0)) ‘
   linj(x)
   : C(S(0), C(S(0), Nil)) = C(S(0), Nil) -->
     False" : thm
```

Notice that there is a big difference in the number of tactics applied to states to reach the final state and that the reduced proof only is on weak head normal form as expected.

B.13 Automatic Proof Reduction

An object logic for automatic proof normalization is created as object logic `PRauto`.

B.13.1 Implementing Normalizer

The normalizer is implemented as an object logic using the meta normalizer specified in object logic `auto`. Inference rules added are the context elimination rules of figure 4.4 and the context introduction *inference* rules. Axiom added is rule (4.24) and the context tactic is `vartac`. Reduction rules of first order logic reduction redexes are included since it is a part of the meta normalizer. A normalizer tactic returned is `retac`.

B.13.2 Examples

The examples below illustrate the goals that are solved as examples for object logic `PR`.

First, it must be illustrated that the normalizer do the job correctly. All context subgoals must be solved before one reduction is done. This is illustrated with the below proving session, automating the reduction of the example starting at page 221.

```
- val [p] = goal PR.thy "a : P ==> ?p : ?P";
std_in:32.1-32.41 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0
?p : ?P
 1. ?p : ?P
val p = "a : P  [a : P]" : thm

...

Level 11
?p : P
 1. fst((lam x. <x,x>) ‘ (lam x. x)) ‘ a >>
    ?p : P
val it = () : unit
val it = () : unit
- by retac;
Level 12
?p : P
 1. fst((lam x. <x,x>) ‘ (lam x. x)) >> ?e1'12 :
       ?Q12 --> P
 2. a >> ?e2'12 : ?Q12
 3. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 13
?p : P
 1. (lam x. <x,x>) ‘ (lam x. x) >> ?e'13 :
       (?Q12 --> P) & ?Q13
 2. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 3. a >> ?e2'12 : ?Q12
```

```
 4. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 14
?p : P
 1. lam x. <x,x> >> ?e1'14 :
       ?Q14 --> (?Q12 --> P) & ?Q13
 2. lam x. x >> ?e2'14 : ?Q14
 3. ?e1'14 ‘ ?e2'14 !> ?e'13 :
    (?Q12 --> P) & ?Q13
 4. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 5. a >> ?e2'12 : ?Q12
 6. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 15
?p : P
 1. !!y. y : ?Q14 ==>
         <y,y> >> ?g15(y) : (?Q12 --> P) & ?Q13
 2. lam x. x >> ?e2'14 : ?Q14
 3. (lam x. ?g15(x)) ‘ ?e2'14 !> ?e'13 :
       (?Q12 --> P) & ?Q13
 4. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 5. a >> ?e2'12 : ?Q12
 6. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 16
?p : P
 1. !!y. y : ?Q14 ==> y >> ?a'16(y) : ?Q12 --> P
 2. !!y. y : ?Q14 ==> y >> ?b'16(y) : ?Q13
 3. lam x. x >> ?e2'14 : ?Q14
 4. (lam x. <?a'16(x),?b'16(x)>) ‘ ?e2'14 !>
    ?e'13 : (?Q12 --> P) & ?Q13
 5. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 6. a >> ?e2'12 : ?Q12
 7. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 17
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==>
         y >> ?a'16(y) : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y >> ?b'16(y) :
         ?Q13
 3. !!y. y : ?P17 ==> y >> ?g17(y) : ?Q17
 4. (lam x. <?a'16(x),?b'16(x)>) ‘
    (lam x. ?g17(x))
       !> ?e'13 : (?Q12 --> P) & ?Q13
 5. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 6. a >> ?e2'12 : ?Q12
 7. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 18
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y >> ?b'16(y) :
         ?Q13
 3. !!y. y : ?P17 ==> y >> ?g17(y) : ?Q17
 4. (lam x. <x,?b'16(x)>) ‘ (lam x. ?g17(x))
       !> ?e'13 : (?Q12 --> P) & ?Q13
 5. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 6. a >> ?e2'12 : ?Q12
 7. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 19
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?Q13
 3. !!y. y : ?P17 ==> y >> ?g17(y) : ?Q17
 4. (lam x. <x,x>) ‘ (lam x. ?g17(x)) !> ?e'13 :
       (?Q12 --> P) & ?Q13
 5. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 6. a >> ?e2'12 : ?Q12
 7. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 20
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?Q13
 3. !!y. y : ?P17 ==> y : ?Q17
 4. (lam x. <x,x>) ‘ (lam x. x) !> ?e'13 :
       (?Q12 --> P) & ?Q13
 5. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 6. a >> ?e2'12 : ?Q12
 7. ?e1'12 ‘ ?e2'12 !> ?p : P
val it = () : unit
- by retac;
Level 21
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?Q13
 3. !!y. y : ?P17 ==> y : ?Q17
 4. (lam x. <x,x>) ‘ (lam x. x) !> ?e'13 :
       (?Q12 --> P) & ?Q13
 5. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 6. a : ?Q12
 7. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 22
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?Q13
 3. !!y. y : ?P17 ==> y : ?Q17
 4. <lam x. x,lam x. x> >> ?e'13 :
    (?Q12 --> P) & ?Q13
 5. fst(?e'13) !> ?e1'12 : ?Q12 --> P
 6. a : ?Q12
 7. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 23
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?Q13
 3. !!y. y : ?P17 ==> y : ?Q17
 4. lam x. x >> ?a'27 : ?Q12 --> P
 5. lam x. x >> ?b'27 : ?Q13
 6. fst(<?a'27,?b'27>) !> ?e1'12 : ?Q12 --> P
 7. a : ?Q12
 8. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 24
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?Q13
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y >> ?g28(y) : P
 5. lam x. x >> ?b'27 : ?Q13
```

```
 6. fst(<lam x. ?g28(x),?b'27>) !>
    ?e1'12 : ?Q12 --> P
 7. a : ?Q12
 8. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 25
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y >> ?g28(y) : P
 5. !!y. y : ?P29 ==> y >> ?g29(y) : ?Q29
 6. fst(<lam x. ?g28(x),lam x. ?g29(x)>) !>
       ?e1'12 : ?Q12 --> P
 7. a : ?Q12
 8. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 26
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y : P
 5. !!y. y : ?P29 ==> y >> ?g29(y) : ?Q29
 6. fst(<lam x. x,lam x. ?g29(x)>) !> ?e1'12 :
       ?Q12 --> P
 7. a : ?Q12
 8. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 27
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y : P
 5. !!y. y : ?P29 ==> y : ?Q29
 6. fst(<lam x. x,lam x. x>) !> ?e1'12 :
    ?Q12 --> P
 7. a : ?Q12
 8. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 28
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y : P
 5. !!y. y : ?P29 ==> y : ?Q29
 6. lam x. x >> ?e1'12 : ?Q12 --> P
 7. a : ?Q12
 8. ?e1'12 ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 29
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y : P
 5. !!y. y : ?P29 ==> y : ?Q29
 6. !!y. y : ?Q12 ==> y >> ?g35(y) : P
 7. a : ?Q12
 8. (lam x. ?g35(x)) ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 30
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y : P
 5. !!y. y : ?P29 ==> y : ?Q29
 6. !!y. y : ?Q12 ==> y : P
 7. a : ?Q12
 8. (lam x. x) ‘ a !> ?p : P
val it = () : unit
- by retac;
Level 31
?p : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y : P
 5. !!y. y : ?P29 ==> y : ?Q29
 6. !!y. y : ?Q12 ==> y : P
 7. a : ?Q12
 8. a >> ?p : P
val it = () : unit
- by retac;
Level 32
a : P
 1. !!y. y : ?P17 --> ?Q17 ==> y : ?Q12 --> P
 2. !!y. y : ?P17 --> ?Q17 ==> y : ?P29 --> ?Q29
 3. !!y. y : ?P17 ==> y : ?Q17
 4. !!y. y : ?Q12 ==> y : P
 5. !!y. y : ?P29 ==> y : ?Q29
 6. !!y. y : ?Q12 ==> y : P
 7. a : ?Q12
 8. a : P
val it = () : unit
- by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 33
a : P
No subgoals!
val it = () : unit
```

The example shows that all context subgoals are solved before any reduction subgoals are solved and any hypotheses are used. Then the normalizer applies tactics in the correct order. Then, it is just to show that the normalizer returns the same proofs as the examples in appendix B.3.

```
- goal PRauto.thy
= "?p : (P | Q) & R --> (P & R) | (Q & R)";

...
```

```
Level 2
?p : (P | Q) & R --> P & R | Q & R
 1. lam x.
       when
        (fst(x), %xa. inl(<xa,snd(x)>),
```

```
        %xa. inr(<xa,snd(x)>))
      >> ?p : (P | Q) & R --> P & R | Q & R
val it = () : unit
- by (REPEAT retac);
Level 3
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(<xa,snd(x)>))
: (P | Q) & R --> P & R | Q & R
 1. !!y. y : (P | Q) & R ==>
         y : (?P4(y) | ?Q4(y)) & ?Q5(y)
 2. !!y x. [| y : (P | Q) & R; x : ?P4(y) |]
           ==> x : P
 3. !!y x.
       [| y : (P | Q) & R; x : ?P4(y) |] ==>
       y : ?P8(y, x) & R
 4. !!y x. [| y : (P | Q) & R; x : ?Q4(y) |]
           ==> x : Q
 5. !!y x.
       [| y : (P | Q) & R; x : ?Q4(y) |] ==>
       y : ?P11(y, x) & R
val it = () : unit
- by (ALLGOALS assume_tac);
Level 4
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(<xa,snd(x)>))
: (P | Q) & R --> P & R | Q & R
No subgoals!
val it = () : unit
- val propEx' = result();
val propEx' =
  "lam x. when(fst(x),
               %xa. inl(<xa,snd(x)>),
               %xa. inr(<xa,snd(x)>))
   : (?P | ?Q) & ?R --> ?P & ?R | ?Q & ?R" : thm


- goal PRauto.thy "?p : P --> (P --> Q) --> Q";
Level 0

...

Level 2
?p : P --> (P --> Q) --> Q
 1. lam x xa. xa ` x >> ?p :
    P --> (P --> Q) --> Q
val it = () : unit
- by (REPEAT retac);
Level 3
lam x xa. xa ` x : P --> (P --> Q) --> Q
 1. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       ya : ?Q5(y, ya) --> Q
 2. !!y ya.
       [| y : P; ya : P --> Q |] ==>
       y : ?Q5(y, ya)
val it = () : unit
- by (ALLGOALS assume_tac);
Level 4
lam x xa. xa ` x : P --> (P --> Q) --> Q
No subgoals!
val it = () : unit
- val impEx' = result{};
val impEx' =
  "lam x xa. xa ` x :
   ?P --> (?P --> ?Q) --> ?Q" : thm


- goal PRauto.thy
= "?p : (EX (x::'a). (ALL (y::'b). P(x,y))) \
= \ --> (ALL (y::'b). (EX (x::'a). P(x,y)))";

...

Level 2
?p : (EX x. ALL y. P(x, y)) -->
     (ALL y. EX x. P(x, y))
 1. lam x.
      all xa. xsplit(x, %xaa u. [xaa,u ^ xa])
       >> ?p :
       (EX x. ALL y. P(x, y)) -->
       (ALL y. EX x. P(x, y))
val it = () : unit
- by (REPEAT retac);
Level 3
lam x. all xa. xsplit(x, %x u. [x,u ^ xa])
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. !!y ya.
       y : EX x. ALL y. P(x, y) ==>
       y : EX a. ?P5(y, ya, a)
 2. !!y ya x u.
       [| y : EX x. ALL y. P(x, y);
          u : ?P5(y, ya, x) |] ==>
       u : ALL b. ?R13(y, ya, b, x, u)
 3. !!y ya x u yb.
       [| y : EX x. ALL y. P(x, y);
          u : ?P5(y, ya, x);
          yb : ?R13(y, ya, ya, x, u) |] ==>
       yb : P(x, ya)
val it = () : unit
- by (ALLGOALS assume_tac);
Level 4
lam x. all xa. xsplit(x, %x u. [x,u ^ xa])
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
No subgoals!
val it = () : unit
- val QuantEx' = result();
val QuantEx' =
  "lam x. all xa. xsplit(x, %x u. [x,u ^ xa])
   : (EX x. ALL y. ?P(x, y)) -->
     (ALL y. EX x. ?P(x, y))" : thm


- val [prems] = goal PRauto.thy
=   "q : a=b ==> ?p : b =a";
std_in:127.1-128.25 Warning:
  binding not exhaustive
          prems :: nil = ...
Level 0

...

Level 3
?p : b = a
 1. idpeel(q, %x. ideq(x)) >> ?p : b = a
val it = () : unit
- by (REPEAT retac);
Level 4
idpeel(q, %c. ideq(c)) : b = a
```

```
 1. q : ?a3 = ?b3
 2. !!y. y : ?P3(?a3, ?b3) ==> y : b = a
 3. !!c y. y : c = c ==> y : ?P3(c, c)
val it = () : unit
- by (ALLGOALS (assume_tac ORELSE'
=               (rtac prems)));
Level 5
idpeel(q, %c. ideq(c)) : b = a
No subgoals!
val it = () : unit
- val sym' = result();
val sym' =
 "?q : ?a = ?b ==>
  idpeel(?q, %c. ideq(c)) : ?b = ?a" : thm


- val [p1,p2] = goal PRauto.thy
=  "[| p:a=b;  q:b=c |] ==> ?d:a=c";
std_in:135.1-136.33 Warning:
  binding not exhaustive
          p1 :: p2 :: nil = ...
Level 0

...

Level 4
?d : a = c
 1. idpeel(q, %x. lam x. x) ` p >> ?d : a = c
val it = () : unit
-  by (REPEAT retac);
Level 5
idpeel(q, %c. lam x. x) ` p : a = c
 1. q : ?a4 = ?b4
 2. !!y. y : ?P7(?a4, ?b4) --> ?Q6(?a4, ?b4) ==>
         y : ?Q3 --> a = c
 3. !!c y. y : ?P7(c, c) ==> y : ?Q6(c, c)
 4. p : ?Q3
val it = () : unit
-  by (ALLGOALS (assume_tac ORELSE'
=                (resolve_tac [p1,p2])));
Level 6
idpeel(q, %c. lam x. x) ` p : a = c
No subgoals!
val it = () : unit
- val trans' = result();
val trans' =
  "[| ?p : ?a = ?b; ?q : ?b = ?c |] ==>
   idpeel(?q, %c. lam x. x) ` ?p
   : ?a = ?c" : thm


- val [p] = goal PRauto.thy
=   "[| p : a=b |]  ==>  ?q : t(a)=t(b)";
std_in:144.1-145.38 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 2
?q : t(a) = t(b)
 1. idpeel(p, %x. ideq(t(x))) >> ?q
    : t(a) = t(b)
val it = () : unit
-  by (REPEAT retac);
Level 3
idpeel(p, %c. ideq(t(c))) : t(a) = t(b)
 1. p : ?a3 = ?b3
 2. !!y. y : ?P3(?a3, ?b3) ==> y : t(a) = t(b)
 3. !!c y. y : t(c) = t(c) ==> y : ?P3(c, c)
val it = () : unit
-  by (ALLGOALS (assume_tac ORELSE'
=                (resolve_tac [p])));
Level 4
idpeel(p, %c. ideq(t(c))) : t(a) = t(b)
No subgoals!
val it = () : unit
- val subst_context' = result();
val subst_context' =
  "?p : ?a = ?b ==>
   idpeel(?p, %c. ideq(?t(c))) :
   ?t(?a) = ?t(?b)" : thm


- val [p] = goal PRauto.thy "a : P ==> ?p : ?P";
std_in:151.1-151.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : P
 1. (lam x. x) ` a >> ?p : P
val it = () : unit
-  by (REPEAT retac);
Level 6
a : P
 1. !!y. y : ?Q4 ==> y : P
 2. a : ?Q4
 3. a : P
val it = () : unit
-  by (ALLGOALS (assume_tac ORELSE'
=                (resolve_tac [p])));
Level 7
a : P
No subgoals!
val it = () : unit
- val impRex = result();
val impRex = "?a : ?P ==> ?a : ?P" : thm


- val [p] =
= goal PRauto.thy "a : P & Q ==> ?p : ?P";
std_in:160.1-160.49 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 6
?p : P
 1. fst((lam x. x) ` a) >> ?p : P
val it = () : unit
-  by (REPEAT retac);
Level 7
fst(a) : P
 1. !!y. y : ?Q8 ==> y : P & ?Q7
 2. a : ?Q8
 3. a : P & ?Q7
val it = () : unit
```

```
-  by (ALLGOALS (assume_tac ORELSE'
=                (resolve_tac [p])));
Level 8
fst(a) : P
No subgoals!
val it = () : unit
- val destRedex = result();
val destRedex =
  "?a : ?P & ?Q ==> fst(?a) : ?P" : thm


-   goal PRauto.thy "?p : ?P";
Level 0

...

Level 3
?p : ?P2 --> ?P2
 1. lam x. x >> ?p : ?P2 --> ?P2
val it = () : unit
-   by (REPEAT retac);
Level 4
lam x. x : ?P2 --> ?P2
 1. !!y. y : ?P2 ==> y : ?P2
val it = () : unit
-   by (ALLGOALS assume_tac);
Level 5
lam x. x : ?P2 --> ?P2
No subgoals!
val it = () : unit
- val exRed3 = result();
val exRed3 = "lam x. x : ?P --> ?P" : thm


- val [p] = goal PRauto.thy "a : P ==> ?p : ?P";
std_in:177.1-177.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 8
?p : P
 1. fst((lam x. <x,x>) ` a) >> ?p : P
val it = () : unit
-   by (REPEAT retac);
Level 9
a : P
 1. !!y. y : ?Q7 ==> y : P
 2. !!y. y : ?Q7 ==> y : ?Q6
 3. a : ?Q7
 4. a : P
 5. a : ?Q6
 6. a : P
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 10
a : P
No subgoals!
val it = () : unit
- val exRed4 = result();
val exRed4 = "?a : ?P ==> ?a : ?P" : thm


-   goal PRauto.thy "?p : ?P";
Level 0
```

```
...

Level 6
?p : ?P2 --> ?P2
 1. lam x. (lam x. x) ` x >> ?p : ?P2 --> ?P2
val it = () : unit
-   by (REPEAT retac);
Level 7
lam x. x : ?P2 --> ?P2
 1. !!y ya. [| y : ?P2; ya : ?Q6(y) |]
            ==> ya : ?P2
 2. !!y. y : ?P2 ==> y : ?Q6(y)
 3. !!y. y : ?P2 ==> y : ?P2
val it = () : unit
-   by (ALLGOALS assume_tac);
Level 8
lam x. x : ?P2 --> ?P2
No subgoals!
val it = () : unit
- val constredex = result();
val constredex = "lam x. x : ?P --> ?P" : thm


- val [p] =
= goal PRauto.thy " a : P ==> ?p : ?P";
std_in:200.1-200.46 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : P
 1. fst(<a,a>) >> ?p : P
val it = () : unit
-   by (REPEAT retac);
Level 6
a : P
 1. a : P
 2. a : ?Q4
 3. a : P
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 7
a : P
No subgoals!
val it = () : unit
- val conjR1ex = result();
val conjR1ex = "?a : ?P ==> ?a : ?P" : thm


- val [p] =
= goal PRauto.thy " a : P ==> ?p : ?P";
std_in:209.1-209.46 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : P
 1. snd(<a,a>) >> ?p : P
val it = () : unit
-   by (REPEAT retac);
```

```
Level 6
a : P
 1. a : ?P4
 2. a : P
 3. a : P
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 7
a : P
No subgoals!
val it = () : unit
- val conjR2ex = result();
val conjR2ex = "?a : ?P ==> ?a : ?P" : thm


- val [p] = goal PRauto.thy
= "(!!x. f(x) : Q(x)) ==> ?p : ?P";
std_in:218.1-218.58 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : Q(?x4(?z2))
 1. all x. f(?x4(x)) ^ ?z2 >> ?p : Q(?x4(?z2))
val it = () : unit
-   by (REPEAT retac);
Level 6
f(?x4(?z2)) : Q(?x4(?z2))
 1. !!y. f(?x4(y)) : ?R6(y)
 2. f(?x4(?z2)) : ?R11(?z2)
 3. !!y. y : ?R11(?a11) ==> y : ?P11(?a11)
 4. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?P11(?z2) =?= ?R6(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 7
f(?x4(?z2)) : Q(?x4(?z2))
No subgoals!

Flex-flex pairs:
  ?x16(?z2) =?= ?x4(?z2)
  ?x16(?a11) =?= ?x15(?a11)
  ?x15(?z2) =?= ?x4(?z2)
  ?x4(?z2) =?= ?x7(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
- val allRex = result();
val allRex =
  "(!!x. ?f(x) : ?Q(x)) ==>
         ?f(?x) : ?Q(?x)" : thm


- val [p] = goal PRauto.thy
= "(!!x. f(x) : Q(x)) ==> ?p : ?P";
std_in:227.1-227.58 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : Q(?x6)
 1. xsplit([?x3,f(?x6)], %x u. u) >> ?p : Q(?x6)
val it = () : unit
-   by (REPEAT retac);
Level 6
f(?x6) : Q(?x6)
 1. f(?x6) : ?P7(?x3)
 2. !!x u. u : ?P7(x) ==> u : Q(?x6)
 3. f(?x6) : Q(?x6)
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 7
f(?x17) : Q(?x17)
No subgoals!
val it = () : unit
- val exRex = result();
val exRex =
  "(!!x. ?f(x) : ?Q(x)) ==>
         ?f(?x) : ?Q(?x)" : thm


- val [p] = goal PRauto.thy "a : Q ==> ?p : ?P";
std_in:236.1-236.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 6
?p : Q
 1. when(inl(a), %x. x, %x. x) >> ?p : Q
val it = () : unit
-   by (REPEAT retac);
Level 7
a : Q
 1. a : ?P4
 2. !!x. x : ?P4 ==> x : Q
 3. !!x. x : ?Q4 ==> x : Q
 4. a : Q
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 8
a : Q
No subgoals!
val it = () : unit
- val disjR1ex = result();
val disjR1ex = "?a : ?Q ==> ?a : ?Q" : thm


- val [p] = goal PRauto.thy "a : Q ==> ?p : ?P";
std_in:246.1-246.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 6
?p : Q
 1. when(inr(a), %x. x, %x. x) >> ?p : Q
val it = () : unit
-   by (REPEAT retac);
Level 7
a : Q
 1. a : ?Q4
```

```
 2. !!x. x : ?P4 ==> x : Q
 3. !!x. x : ?Q4 ==> x : Q
 4. a : Q
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 8
a : Q
No subgoals!
val it = () : unit
- val disjR2ex = result();
val disjR2ex = "?a : ?Q ==> ?a : ?Q" : thm


- val [p] = goal PRauto.thy "a : Q ==> ?p : ?P";
std_in:256.1-256.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : Q
 1. idpeel(ideq(?x2), %x. a) >> ?p : Q
val it = () : unit
-   by (REPEAT retac);
Level 6
a : Q
 1. !!y. y : ?x2 = ?x2 ==> y : ?a4 = ?b4
 2. !!y. y : ?P4(?a4, ?b4) ==> y : Q
 3. !!c. a : ?P4(c, c)
 4. a : ?R9(?x2, ?x2)
 5. !!y. y : ?R9(?b9, ?b9) ==> y : ?P9(?b9, ?b9)

Flex-flex pairs:
  ?P9(?x2, ?x2) =?= ?P4(?a4, ?b4)
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 7
a : Q
No subgoals!
val it = () : unit
- val substRex = result();
val substRex = "?a : ?Q ==> ?a : ?Q" : thm
```

B.14 Automatic Program Evaluation

An object logic for automatic program evaluation is created as object logic `PEauto`.

B.14.1 Implementing Normalizer

The normalizer is implemented as an object logic using the meta normalizer specified in object logic `auto`. Inference rules added are the context elimination rules of figure 4.4 and the context tactic is `try_allRules`. Reduction rules of first order logic reduction redexes are included since it is a part of the meta normalizer. A normalizer tactic returned is `retac`.

B.14.2 Examples

The examples below illustrate the goals that are solved as examples for object logic `PE`.

It is illustrated in B.13.2 that the normalizer do apply rules in the correct ordern and this is a property of the meta normalizer. Therefore it is enough to show that the normalizer returns the same proofs as the examples in appendix B.13, and also that the context rules are handled correctly.

```
- val [p] = goal PEauto.thy "a : P ==> ?p : ?P";
std_in:25.1-25.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 11
?p : P
 1. fst((lam x. <x,x>) ` (lam x. x)) ` a >> ?p :
       P
val it = () : unit
-   by (REPEAT retac);
Level 12
a : P
No subgoals!
val it = () : unit
- val redexAfter = result();
val redexAfter = "?a : ?P ==> ?a : ?P" : thm


- goal PEauto.thy
= "?p : (P | Q) & R --> (P & R) | (Q & R)";
Level 0
```

```
...

Level 2
?p : (P | Q) & R --> P & R | Q & R
 1. lam x.
       when
        (fst(x), %xa. inl(<xa,snd(x)>),
         %xa. inr(<xa,snd(x)>))
       >> ?p : (P | Q) & R --> P & R | Q & R
val it = () : unit
- by (REPEAT retac);
Level 3
lam x.
   when
    (fst(x), %xa. inl(<xa,snd(x)>),
     %xa. inr(<xa,snd(x)>))
: (P | Q) & R --> P & R | Q & R
No subgoals!
val it = () : unit
- val propEx' = result();
val propEx' =
  "lam x. when(fst(x),
               %xa. inl(<xa,snd(x)>),
               %xa. inr(<xa,snd(x)>))
   : (?P | ?Q) & ?R --> ?P & ?R | ?Q & ?R" : thm


- goal PEauto.thy "?p : P --> (P --> Q) --> Q";
Level 0

...

Level 2
?p : P --> (P --> Q) --> Q
 1. lam x xa. xa ` x >> ?p :
       P --> (P --> Q) --> Q
val it = () : unit
- by (REPEAT retac);
Level 3
lam x xa. xa ` x : P --> (P --> Q) --> Q
No subgoals!
val it = () : unit
- val impEx' = result{};
val impEx' =
  "lam x xa. xa ` x :
   ?P --> (?P --> ?Q) --> ?Q" : thm


- goal PEauto.thy
= "?p : (EX (x::'a). (ALL (y::'b). P(x,y))) \
= \ --> (ALL (y::'b). (EX (x::'a). P(x,y)))";
Level 0

...

Level 2
?p
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
 1. lam x.
       all xa. xsplit(x, %xaa u. [xaa,u ^ xa])
       >> ?p :
       (EX x. ALL y. P(x, y)) -->
       (ALL y. EX x. P(x, y))
val it = () : unit
- by (REPEAT retac);
Level 3
lam x. all xa. xsplit(x, %xaa u. [xaa,u ^ xa])
: (EX x. ALL y. P(x, y)) -->
  (ALL y. EX x. P(x, y))
No subgoals!
val it = () : unit
- val QuantEx' = result();
val QuantEx' =
  "lam x.
     all xa. xsplit(x, %xaa u. [xaa,u ^ xa])
   : (EX x. ALL y. ?P(x, y)) -->
     (ALL y. EX x. ?P(x, y))" : thm


- val [prems] = goal PEauto.thy
=   "q : a=b ==> ?p : b =a";
std_in:61.1-62.25 Warning:
  binding not exhaustive
          prems :: nil = ...
Level 0

...

Level 3
?p : b = a
 1. idpeel(q, %x. ideq(x)) >> ?p : b = a
val it = () : unit
- by (REPEAT retac);
Level 4
idpeel(q, %c. ideq(c)) : b = a
 1. !!y. y : ?P3(?a3, ?b3) ==> y : b = a
val it = () : unit
- by (ALLGOALS (assume_tac ORELSE'
=               (rtac prems)));
Level 5
idpeel(q, %c. ideq(c)) : b = a
No subgoals!
val it = () : unit
- val sym' = result();
val sym' =
  "?q : ?a = ?b ==>
   idpeel(?q, %c. ideq(c)) : ?b = ?a" : thm


- val [p1,p2] = goal PEauto.thy
=  "[| p:a=b;  q:b=c |] ==> ?d:a=c";
std_in:69.1-70.33 Warning:
  binding not exhaustive
          p1 :: p2 :: nil = ...
Level 0

...

Level 4
?d : a = c
 1. idpeel(q, %x. lam x. x) ` p >> ?d : a = c
val it = () : unit
-  by (REPEAT retac);
Level 5
idpeel(q, %c. lam x. x) ` p : a = c
 1. !!y. y : ?P4(?a4, ?b4) ==> y : ?Q3 --> a = c
val it = () : unit
-  by (ALLGOALS (assume_tac ORELSE'
=                (resolve_tac [p1,p2])));
Level 6
idpeel(q, %c. lam x. x) ` p : a = c
No subgoals!
val it = () : unit
```

```
- val trans' = result();
val trans' =
  "[| ?p : ?a = ?b; ?q : ?b = ?c |] ==>
   idpeel(?q, %c. lam x. x) ` ?p
   : ?a = ?c" : thm


- val [p] = goal PEauto.thy
=   "[| p : a=b |]  ==>  ?q : t(a)=t(b)";
std_in:78.1-79.38 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 2
?q : t(a) = t(b)
 1. idpeel(p, %x. ideq(t(x))) >> ?q :
    t(a) = t(b)
val it = () : unit
-  by (REPEAT retac);
Level 3
idpeel(p, %c. ideq(t(c))) : t(a) = t(b)
 1. !!y. y : ?P3(?a3, ?b3) ==> y : t(a) = t(b)
val it = () : unit
-  by (ALLGOALS (assume_tac ORELSE'
=                (resolve_tac [p])));
Level 4
idpeel(p, %c. ideq(t(c))) : t(a) = t(b)
No subgoals!
val it = () : unit
- val subst_context' = result();
val subst_context' =
  "?p : ?a = ?b ==>
   idpeel(?p, %c. ideq(?t(c))) :
   ?t(?a) = ?t(?b)" : thm


- val [p] = goal PEauto.thy "a : P ==> ?p : ?P";
std_in:85.1-85.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : P
 1. (lam x. x) ` a >> ?p : P
val it = () : unit
-  by (REPEAT retac);
Level 6
a : P
No subgoals!
val it = () : unit
- val impRex = result();
val impRex = "?a : ?P ==> ?a : ?P" : thm


- val [p] =
= goal PEauto.thy "a : P & Q ==> ?p : ?P";
std_in:94.1-94.49 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 6
?p : P
 1. fst((lam x. x) ` a) >> ?p : P
val it = () : unit
-  by (REPEAT retac);
Level 7
fst(a) : P
No subgoals!
val it = () : unit
- val destRedex = result();
val destRedex =
  "?a : ?P & ?Q ==> fst(?a) : ?P" : thm


-   goal PEauto.thy "?p : ?P";
Level 0

...

Level 3
?p : ?P2 --> ?P2
 1. lam x. x >> ?p : ?P2 --> ?P2
val it = () : unit
-   by (REPEAT retac);
Level 4
lam x. x : ?P2 --> ?P2
No subgoals!
val it = () : unit
- val exRed3 = result();
val exRed3 = "lam x. x : ?P --> ?P" : thm


- val [p] = goal PEauto.thy "a : P ==> ?p : ?P";
std_in:111.1-111.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 8
?p : P
 1. fst((lam x. <x,x>) ` a) >> ?p : P
val it = () : unit
-   by (REPEAT retac);
Level 9
a : P
No subgoals!
val it = () : unit
- val exRed4 = result();
val exRed4 = "?a : ?P ==> ?a : ?P" : thm


-   goal PEauto.thy "?p : ?P";
Level 0

...

Level 6
?p : ?P2 --> ?P2
 1. lam x. (lam x. x) ` x >> ?p : ?P2 --> ?P2
val it = () : unit
-   by (REPEAT retac);
Level 7
lam x. (lam x. x) ` x : ?P2 --> ?P2
```

```
No subgoals!
val it = () : unit
- val constredex = result();
val constredex =
  "lam x. (lam x. x) ‘ x : ?P --> ?P" : thm


- val [p] =
= goal PEauto.thy " a : P ==> ?p : ?P";
std_in:134.1-134.46 Warning:
 binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : P
 1. fst(<a,a>) >> ?p : P
val it = () : unit
-   by (REPEAT retac);
Level 6
a : P
No subgoals!
val it = () : unit
- val conjR1ex = result();
val conjR1ex = "?a : ?P ==> ?a : ?P" : thm


- val [p] =
= goal PEauto.thy " a : P ==> ?p : ?P";
std_in:143.1-143.46 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : P
 1. snd(<a,a>) >> ?p : P
val it = () : unit
-   by (REPEAT retac);
Level 6
a : P
No subgoals!
val it = () : unit
- val conjR2ex = result();
val conjR2ex = "?a : ?P ==> ?a : ?P" : thm


- val [p] = goal PEauto.thy
= "(!!x. f(x) : Q(x)) ==> ?p : ?P";
std_in:152.1-152.58 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : Q(?x4(?z2))
 1. all x. f(?x4(x)) ^ ?z2 >> ?p : Q(?x4(?z2))
val it = () : unit
-   by (REPEAT retac);
Level 6
f(?x4(?z2)) : Q(?x4(?z2))
 1. !!y. y : ?R10(?a10) ==> y : ?P10(?a10)
 2. !!y. y : ?R6(?z2) ==> y : Q(?x7(?z2))

Flex-flex pairs:
  ?P10(?z2) =?= ?R6(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 7
f(?x4(?z2)) : Q(?x4(?z2))
No subgoals!

Flex-flex pairs:
  ?x15(?a10) =?= ?x14(?a10)
  ?x14(?z2) =?= ?x13(?z2)
  ?x13(?z2) =?= ?x7(?z2)
  ?x7(?z2) =?= ?x4(?z2)
val it = () : unit
- val allRex = result();
val allRex =
  "(!!x. ?f(x) : ?Q(x)) ==>
         ?f(?x) : ?Q(?x)" : thm

- val [p] = goal PEauto.thy
= "(!!x. f(x) : Q(x)) ==> ?p : ?P";
std_in:161.1-161.58 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : Q(?x6)
 1. xsplit([?x3,f(?x6)], %x u. u) >> ?p : Q(?x6)
val it = () : unit
-   by (REPEAT retac);
Level 6
f(?x6) : Q(?x6)
No subgoals!
val it = () : unit
- val exRex = result();
val exRex =
  "(!!x. ?f(x) : ?Q(x)) ==>
         ?f(?x) : ?Q(?x)" : thm


- val [p] = goal PEauto.thy "a : Q ==> ?p : ?P";
std_in:170.1-170.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 6
?p : Q
 1. when(inl(a), %x. x, %x. x) >> ?p : Q
val it = () : unit
-   by (REPEAT retac);
Level 7
a : Q
No subgoals!
val it = () : unit
- val disjR1ex = result();
val disjR1ex = "?a : ?Q ==> ?a : ?Q" : thm
- val [p] = goal PEauto.thy "a : Q ==> ?p : ?P";
```

```
std_in:180.1-180.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 6
?p : Q
 1. when(inr(a), %x. x, %x. x) >> ?p : Q
val it = () : unit
-   by (REPEAT retac);
Level 7
a : Q
No subgoals!
val it = () : unit
- val disjR2ex = result();
val disjR2ex = "?a : ?Q ==> ?a : ?Q" : thm


- val [p] = goal PEauto.thy "a : Q ==> ?p : ?P";
std_in:190.1-190.45 Warning:
  binding not exhaustive
          p :: nil = ...
Level 0

...

Level 5
?p : Q
 1. idpeel(ideq(?x2), %x. a) >> ?p : Q
val it = () : unit
-   by (REPEAT retac);
Level 6
a : Q
 1. !!y. y : ?P4(?a4, ?b4) ==> y : Q
 2. !!y. y : ?R9(?b9, ?b9) ==> y : ?P9(?b9, ?b9)

Flex-flex pairs:
  ?P9(?x2, ?x2) =?= ?P4(?a4, ?b4)
val it = () : unit
-   by (ALLGOALS (assume_tac ORELSE' (rtac p)));
Level 7
a : Q
No subgoals!
val it = () : unit
- val substRex = result();
val substRex = "?a : ?Q ==> ?a : ?Q" : thm
```

Bibliography

[1] Anderson, P. *Program Derivation by Proof Transformation.* PhD thesis, Carnegie Mellon University, Oct. 1993. Available as Technical Report CMU-CS-93-206.

[2] Berg, C. *Matematik 3GT Generel Topologi.* Københavns Universitet, Matematisk Institut, 1996.

[3] Boyer, C. B., and Merzbach, U. C. *A History of Mathematics.* John Wiley & Sons Inc., 1989.

[4] Gallier, J. Constructive logics. part i: A tutorial on proof systems and typed λ-calculi. Tech. rep., Department of Computer and Information Science, University of Pennsylvania, 1992.

[5] Gallier, J. On the correspondence between proofs and λ-terms. Tech. rep., Department of Computer and Information Science, University of Pennsylvania, 1993.

[6] Girard, J., Lafont, Y., and Taylor, P. *Proofs and Types.* Cambridge University Press, 1989.

[7] Kjellerup, B. unpublished note on how to write proofs in a readable manner. For the dat0 course.

[8] Komagata, Y., and Schmidt, D. A. Implementation of intuitionistic type theory and realizability theory. Tech. rep., Kansas State University, 1995.

[9] Mendelson, E. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole Advanced Books & Software, 1987.

[10] Parigot, M. Recursive programming with proofs. *Theoretical Computer Science 94*, 2 (9 Mar. 1992), 335–356.

[11] Paulson, L. C. Isabelle: The next 700 theorem provers. In *Logic and Computer Science*, P. Odifreddi, Ed. Academic Press, 1990, pp. 361–386.

[12] Paulson, L. C. *ML for the Working Programmer.* Cambridge University Press, 1993.

[13] Paulson, L. C. *Introduction to Isabelle.* Computer Laboratory, University of Cambridge, 1995.

[14] Paulson, L. C. *The Isabelle Reference Manual.* Computer Laboratory, University of Cambridge, 1996.

[15] Paulson, L. C. *Isabelle's Object Logics.* Computer Laboratory, University of Cambridge, 1996.

[16] Pfenning, F. Computation and deduction. Unpublished lecture notes, 277 pp. Revised May 1994, April 1995, May 1992.

[17] PLASMEIJER, R., AND VAN EEKELEN, M. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley Publishing Company, 1993.

[18] SCMIDT, D. A. *The Structure of Typed Programming Languages.* MIT Press, 1994.

[19] Bible online.