Implementation of a Proof Assistant in Prolog

Andreas Hussing

Kongens Lyngby 2011
IMM-BSc-2011-12

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-BSc: ISSN 0909-3192, ISBN

Summary

In an intuitionistic logic the law of excluded middle is disregarded. However it is shown that if an intuitionistic logic accepts the axioms of choice and extensionality, then the law of excluded middle can be derived in that very logic. A logic that accepts these two axioms will therefore always be classical and cannot be intuitionistic.

In this thesis the intuitionistic higher order logic *HOLPro* is presented, and it is shown that this logic turns out classical when adding the axioms of choice and extensionality to it. It is shown by deriving the law of excluded middle and Peano arithmetic from *HOLPro*. The modus ponens rule is needed to use the axiom of choice to make *HOLPro* classical, and it is therefore derived in the logic.

An implementation of the logic has been made in Prolog. The types, terms, axioms and derived rules of the logic have been implemented to show, how the deduction rules of *HOLPro* can be used to validate a formula from one or more other formulas, which is the purpose of a proof assistant.

Resume

I en intuitionistisk logik er "the law of excluded middle" ikke gyldig. Dog kan det vises, at hvis en intuitionistisk logik accepterer "axiom of choice" og "axiom of extensionality", så kan "the law of excluded middle" alligevel udledes i logikken. En logik der accepterer disse to aksiomer vil derfor altid være klassisk og ikke intuitionistisk.

I denne opgave vil en intuitionistisk højereordenslogik *HOLPro* blive præsenteret, og det vil blive vist, at denne logik er klassisk, når "axiom of choice" og "axiom of extensionality" tilføjes til den. Det er vist ved at udlede "the law of excluded middle" og Peano aritmetik fra *HOLPro*. Modus ponens reglen er nødvendig for at kunne bruge "axiom of choice" til at gøre *HOLPro* klassisk, og den er derfor også udledt fra den opstillede logik.

En implementation af logikken er blevet lavet i Prolog. Typerne, termerne, aksiomerne og de afledte regler fra logikken er blevet implementeret for at vise, hvordan inferensreglerne i *HOLPro* kan blive brugt til at verificere en formel fra en eller flere andre formler, hvilket er formålet for en bevisassistent.

Preface

This bachelor thesis was written at the Department of Informatics and Mathematical Modelling at the Technical University of Denmark. It has been written as a part of the study for a BSc degree in software technology. The thesis was written in the period from the 1 February 2011 to the 27 June 2011. The project is for 15 ECTS points and was supervised by Jørgen Villadsen.

The reader is assumed to be a student who has completed the courses 02156 Logical Systems and Logic Programming and 02157 Functional Programming.

Andreas Hussing, June 2011

Contents

Chapter 1

Introduction

In classical logic a formula can either be true or false. In intuitionistic logic this is not taken for granted. In this special kind of logic a formula is only considered true if a proof of this exists, and a formula is only considered false if a proof of this exists. Therefore a contradicted proof of falsity cannot be used as a proof of truth for a formula and vice versa. This view on logic questions some of the most basic tautologies in classical logic:

- $p \vee \neg p$ (law of excluded middle)
- $\neg(\neg p) \Leftrightarrow p$ (double-negation)

In intuitionistic logic these formulas cannot be taken as valid. However if the axioms of choice and extensionality are accepted, it can be shown that the law of excluded middle is derivable from an intuitionistic logic (and thereby all formulas that are valid in classical logic).

The purpose of this thesis is more of an intellectual kind than oriented towards a specific problem. The thesis will present an intuitionistic higher order logic called *HOLPro*. Typed means that the variables in the logic will have types, just as in a programming language. This logic will be defined by some basic

axioms, and derived rules will be deduced from them. From *HOLPro* it should be possible to derive the law of excluded middle and Peano arithmetic by accepting the axioms of choice and extensionality.

Furthermore the logic will be used to implement a proof assistant. A proof assistant is a programme that can assist in proving a theorem. For the implemented proof assistant this will be done with the axioms and derived rules of *HOLPro*. These rules can be used to prove one formula from one or more other formulas. As an example we will show the modus ponens rule that is ultimately derived in *HOLPro*. This rule can be shown formally as

$$\frac{\Gamma \to p \Rightarrow q \qquad \Delta \to q}{\Gamma \cup \Delta \to q}\ \text{mp}$$

Using this rule it is possible to confirm the validity of the formula $\Gamma \cup \Delta \to q$ from the formulas $\Gamma \to p \Rightarrow q$ and $\Delta \to q$. When working with classical logic we often take this rule for granted, but it is an interesting challenge to verify that the modus ponens rule is also a valid deduction rule in intuitionistic logic. The purpose of the implementation is to show, that it is possible to make a proof assistant with an intuitionistic logic that can use the modus ponens rule.

The goal of the thesis is two-fold:

- Implement a proof assistant in Prolog based on an intuitionistic higher order logic that can use the modus ponens rule.
- Show that the logic used for the proof assistant is classical when the axioms of choice and extensionality are accepted.

The theory behind *HOLPro* is described in chapter 2. This includes higher order logic, intuitionistic logic and deductions in logic.
The deduction rules of *HOLPro* will be presented in chapter 3. It will be stated which axioms the logic is based on, which deduction rules can be derived from the axioms and how the usual logical connectives (for instance conjunction) can be described in *HOLPro*.
The implementation in Prolog and tests of *HOLPro* will be described in chapter 4 and 5. It is shown how the implementation should be understood and used for deriving formulas from other formulas.

HOLPro will be discussed in chapter 6. In this chapter it will be discussed how *HOLPro* can be used for deriving the law of excluded middle and be turned into a classical logic. Furthermore Peano arithmetic will be derived from *HOLPro*. In the end other subjects will be discussed to investigate if *HOLPro* could be axiomatised or implemented in a different and maybe better way.
In the end conclusions from this thesis will be presented in chapter 7.

Chapter 2

Higher-order logic

Higher-order logic is extended from first-order logic by allowing quantification on predicate, propositional and function variables. An example could be shown with the 1-ary predicate p. This predicate takes as argument an individual variable and gives a boolean value. A formula in first-order logic could then be

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

Now we introduce a new predicate q. This predicate takes as argument another predicate and gives a boolean value. A formula with q could then be

$$\forall p \forall x.q(p(x))$$

This is an example of a formula in second-order logic. We can go further and give a formula in third-order logic

$$\forall q \forall p \forall x.r(q(p(x)))$$

We can continue in this way forever. Therefore we can make arbitrarily high order logics. The logic which includes all finite-order logics is called omega-order logic or finite type theory. Any finite-order logic is a subset of this logic. The reason why it is also called finite type theory is that symbols in the logic are typed. Just like a variable in a programming language can be of type integer, real, boolean or something else, a symbol in omega-order logic has a type.
References: [2]

2.1 Intuitionistic logic

Intuitionistic logic can be viewed as a restriction of classical logic. In classical logic a formula is either true or false, even if no proof of neither of them exists. In intuitionistic logic, this assumption is not taken. Instead a formula is only considered true if a proof of this exists and only considered false if a proof of this exists. If neither exists, the formula is considered to be neither true nor false. In classical logic validity is defined by recursion over the structure of a formula. For instance in the formula $p \wedge q$ both p and q have to be true to make the overall formula true. Again these two formulas might consist of other compounded formulas that have to be evaluated recursively. In intuitionistic logic provability is considered essential. A formula in intuitionistic logic is true if and only if there exists a proof of this. A proof of the different logical connectives are defined as following:

- $\bot$ has no proof.
- A proof of $p \Rightarrow q$ is a rule that can make any proof of p into a proof of q.
- A proof of $p \wedge q$ contains both a proof of p and of q.
- A proof of $p \vee q$ is a proof of either p or q and an indication of which one.
- A proof of $\forall x.P[x]$ is a rule that can prove $P[a]$ for any object a.
- A proof of $\exists x.P[x]$ is a proof of $P[a]$ for an object a.

It is easily seen that provability of a formula in intuitionistic logic has a recursive structure similar to the evaluation of formulas in classical logic. However some formulas that is considered valid in classical logic cannot be proved in intuitionistic logic. This is for instance

- $\neg(\neg p) \Leftrightarrow p$ (double-negation)

- $p \vee \neg p$ (law of excluded middle)
- $p \vee q \Leftrightarrow \neg p \Rightarrow q$

These rules' validity in classical logic is built on the very assumption, that if a formula is not true, then it must be false and vice versa.
References: [8].

2.2 Deduction trees

The formal proof system that will be presented is built on deduction trees. A deduction tree is a notation for a rule with premises and a conclusion. The premises can be viewed as arguments for the rule and the conclusion is the result. An example of a deduction tree could be

$$\frac{A \quad B}{A \wedge B}\ \text{conj}$$

This rule says that if A is true and B is true, then $A \wedge B$ is also true. As a notation for the logical formulas we will use sequents. Sequents are on the form $\Gamma \rightarrow \Delta$, where Γ is the assumptions and Δ is the consequents. A sequent says that if its assumptions are true then its consequents are also true. We restrict Δ to be at most one formula, as this will keep the deductions simpler, but also make the logic intuitionistic.
References: [8], [1].

Chapter 3

The system $HOLPro$

In this chapter the logical principles of the implemented proof assistant will be presented. First we will look on the formulation of type theory that will be used. This formulation will be called $HOLPro$, which is short for "higher order logic in Prolog". We start by defining the type symbols. We use the greek letters α and β to denote type symbols. Furthermore we introduce functions to the system. Type symbols in $HOLPro$ are defined as follows:

- ι (iota) denotes the type of individual variables.
- o (omicron) denotes the type of truth values.
- The function which as argument takes a type symbol α and returns a type symbol β has the type $(\alpha\beta)$. This can also be written $(\alpha \to \beta)$ like functions are in many functional programming languages. α is said to be the function's domain and β its range.

By using curring it is possible to restrict the functions of $HOLPro$ to take only one argument. Curring says that if a function takes more than one argument, we can split up the function into several functions that take one argument each and use these functions as arguments themselves. An example could be

$$f(x, y) = f(g(y))$$

Therefore a function $f(x_{\alpha}{}^{1}_{1}, \ldots, x_{\alpha}{}^{n}_{n})$ will have the type $(\alpha_1(\alpha_2 \ldots (\alpha_n\beta)))$.

In *HOLPro* a formula will always have a type. Therefore we can write that a formula is of a special type. The definition of formulas is as follows:

- A variable or constant A_α is of type α. These are the most basic formulas in *HOLPro*.
- $A_{\alpha\beta}B_\beta$ is of type α. This kind of formula is called an application. The first part of the formula $A_{\alpha\beta}$ will be called the function and the second part B_β will be called the argument.
- $\lambda x_\alpha B_\beta$ is of type $(\alpha\beta)$. This kind of formula is called an abstraction.

In an application the function is of type function (as the name indicates) and is applied to the second formula that can be of any type. It is important that the types of the first and second formula match. The domain of the first formula must be the same as the range of the second formula. Whereas an application is like using a function on an argument, an abstraction is like creating a function. An abstraction consists of a variable (the binding variable) and another formula, and the abstraction is said to bind the variable in the formula. When a variable occurs as the binding variable for an abstraction, it is called a bound variable. If a variable does not occur as the binding variable in an abstraction, it is called a free variable. Notice that in the implementation a formula will be called a term.
References: [2].

3.1 Axioms for *HOLPro*

In the deduction system of *HOLPro* we will often work with theorems instead of terms. Theorems are the implementation of sequents. A theorem is a term that have an assumption list, which is a list of other terms. If all the terms in the assumption list are true, then the term is also true. The assumption list will be called the assumptions, and the term will be called the consequent. Whereas a term can be made directly from variables, constants, applications and abstractions, a theorem has to be made from one of the basic or derived rules. A theorem therefore has to be proved from terms and the deductive rules to secure that the theorem holds. The consequent t of a theorem that has no assumptions will always be true, and the theorem will be written like $\rightarrow t$. This however cannot be guaranteed fully when implemented in Prolog, because it is

possible to write anything everywhere in this language.
The deductive system is built around the equality primitive, as this is the only primitive that will occur in the rules. The first two axioms we will define are reflexivity and transitivity for equality:

$$\frac{}{\rightarrow t = t}\ \text{refl}$$

$$\frac{\Gamma \rightarrow s = t \qquad \Delta \rightarrow t = u}{\Gamma \cup \Delta \rightarrow s = u}\ \text{trans}$$

The refl axiom says, that a theorem can be made by setting any term equal to itself. Trans says, that if the rhs (right hand side) of the first premise and the lhs (left hand side) of the second are equal, then the lhs of the first premise and the rhs of the second premise are equal. In this situation equal means alpha equal. Alpha equality will be explained later.
Now we will specify two axioms that enable us to make equalities for applications and abstractions given equalities for the subcomponents. The two axioms are congruence and abstraction:

$$\frac{\Gamma \rightarrow s = t \qquad \Delta \rightarrow u = v}{\Gamma \cup \Delta \rightarrow s(u) = t(v)}\ \text{cong}$$

$$\frac{\Gamma \rightarrow s = t}{\Gamma \rightarrow (\lambda x.s) = (\lambda x.t)}\ \text{abs}$$

To be able to use the cong rule $s(u)$ and $t(v)$ must be formulas in the language of our logic. That is, s has to be of type function and the type of its domain must be the same as the range for u. This also counts for t and v. In abs we require that the variable x is not free in any of the terms in Γ.
Now we present the beta axiom and the extensionality axiom. The first one specifies the inverse of an abstraction and an application. The second one is a special case where the variable of the abstraction does not occur freely in the body of it.

$$\frac{}{\rightarrow (\lambda x.t)x = t}\ \text{beta}$$

$$\frac{}{\rightarrow \lambda x.(tx) = t}\ \text{eta}, x \notin FV(t)$$

The beta rule says, that an application consisting of an abstraction and the bound variable variable (e.g. the abstraction is applied to its binding variable) equals the body of the abstraction. Eta says that if the binding variable of an abstraction does not occur freely in the body, then this abstraction will always be equal to its body, no matter what term it is used on. The constraint of the binding variable not occuring in the body can more formally be specified like $x \notin FV(t)$, where FV denotes the free variables. This eta rule is also known as η-conversion.
The next axiom is the basic property of a sequent for a term of type boolean:

$$\frac{}{\{p_o\} \rightarrow p_o}\ \text{assume}$$

The next axiom is a modus ponens-like rule for equality:

$$\frac{\Gamma \to p = q \qquad \Delta \to p}{\Gamma \cup \Delta \to q}\ \text{eq\_mp}$$

Now an antisymmetry axiom with theorems will be presented:

$$\frac{\Gamma \to p \qquad \Delta \to q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \to p = q}\ \text{deduct\_antisym}$$

The last rule specifies instantiation of variables. Instantiation is substitution of all free occurrences of a variable with another term that has the same type as the variable.

$$\frac{\Gamma[x_1, ..., x_n] \to p[x_1, ..., x_n]}{\Gamma[t_1, ..., t_n] \to p[t_1, ..., t_n]}\ \text{inst}$$

References: The definitions of the axioms are from [8].

3.2 Derived rules

We can develop derived equality rules by combining the axioms of *HOLPro*. These derived rules will be presented as derivation trees. The trees are built from the axioms or derived rules already presented. The first two rules are modified versions of the cong rule. Whereas cong as arguments takes two equalities, one for the functions and one for the arguments, the two derived rules cong_function and cong_parameter takes respectively only one function and one argument.

$$\frac{\dfrac{}{\to tm = tm}\ \text{refl} \qquad \Gamma \to u = v}{\Gamma \to tm(u) = tm(v)}\ \text{cong}$$

$$\frac{\Gamma \to s = t \qquad \dfrac{}{\Gamma \to tm = tm}\ \text{refl}}{\to s(tm) = t(tm)}\ \text{cong}$$

Figure 3.1: The first is cong_function and the second is cong_parameter.

Now beta conversion will be presented. Beta conversion takes as input an application consisting of an abstraction $(\lambda x.s)$ and a variable t that the abstraction is used on. It is the purpose of the beta conversion to apply t to $(\lambda x.s)$. This is done by first use the beta rule to get $(\lambda x.s)x = s$. Now x is instantiated to t. This has no effect on the lhs of the equality as x is bound here, but x will be substituted by t on the rhs of the equality, yielding $(\lambda x.s)t = [x := t]s$:

$$\dfrac{\dfrac{}{(\lambda x.s)x = s}\ \text{beta}}{(\lambda x.s)t = [x := t]s}\ \text{inst } x \to t$$

Figure 3.2: beta_conv.

The next derived rule is symmetry. This rule says that $l = r$ is equal to $r = l$. It is shown by using the axioms refl, cong and eq_mp. For display purpose the derivation tree has been split into two. The first tree will derive $\Gamma \to (l = l) = (r = l)$, and the second one will use this as a premise for deriving the final conclusion.

First tree:

$$\dfrac{\dfrac{\dfrac{}{\to (=) = (=)}\ \text{refl} \qquad \Gamma \to l = r}{\Gamma \to (l =) = (r =)}\ \text{cong} \qquad \dfrac{}{\to l = l}\ \text{refl}}{\Gamma \to (l = l) = (r = l)}\ \text{cong}$$

Second tree:

$$\dfrac{\Gamma \to (l = l) = (r = l) \qquad \dfrac{}{\to l = l}\ \text{refl}}{\Gamma \to r = l}\ \text{eq\_mp}$$

Figure 3.3: sym.

Now we will present some derived rules used for alpha conversion. The purpose of alpha conversion is to rename a bound variable in an abstraction. The first presented alpha rule does the renaming by splitting up an abstraction $\lambda x.body$ into its binding variable and its body. The binding variable is now renamed, let us say from x to y, in the body, where it is not bound. In the end a new abstraction $\lambda y.[x := y]body$ is composed of the renamed variable and the renamed body.

$$\dfrac{y \qquad \dfrac{body}{[x := y]body}\ \text{inst } x \to y}{\lambda y.[x := y]body}\ \text{create\_abs}$$

Figure 3.4: alpha_term.

For the next derived rule we are going to use the concept alpha convertibility (also called alpha equality). When two terms are alpha convertible, it means that an alpha conversion can make the two terms exactly equal. For instance $\lambda x.x$ is alpha convertible with $\lambda y.y$. The two terms are the identity function just written with different variable names. An example of the opposite could be the terms $\lambda x.x$ and $\lambda x.y$. The body of the first abstraction contains only a bound variable, whereas the body of the second abstraction contains only a free variable. Therefore these terms can never be made equal. Notice that $\lambda x.y$ and $\lambda x.v$ will not be alpha convertible either, because alpha conversions only rename bound variables. Therefore y cannot be renamed to v or vice versa. In other words two terms are alpha convertible if they are the same terms just with different names for the bound variables. The next derived rule sets two theorems equal if they are alpha convertible:

$$\dfrac{\dfrac{}{\Gamma \to s = s}\ \text{refl} \qquad \dfrac{}{\Delta \to t = t}\ \text{refl}}{\Gamma \cup \Delta \to s = t}\ \text{trans}$$

Figure 3.5: alpha_equal.

The next derived rule alpha_conv puts alpha_term and alpha_equal together to make a complete alpha conversion on an abstraction. alpha_term v means that the bound variable will be renamed to v.

$$\dfrac{\lambda x.s \qquad \dfrac{\lambda x.s}{\lambda v.[x := v]s}\ \text{alpha\_term } v}{\to \lambda x.s = \lambda v.[x := v]s}\ \text{alpha\_equal}$$

Figure 3.6: alpha_conv.

Now we can make an alpha conversion on an abstraction. However it will be convenient to be able to make alpha conversions directly on quantifiers also (quantifiers will be introduced later). This is done by the next derived rule gen_alpha_conv. If the rule is used on an abstraction, it just uses alpha_conv. If the rule is used on an application consisting of a binder and an abstraction, then following rule is used:

$$\frac{binder \qquad \dfrac{\lambda x.s}{\to \lambda x.s = \lambda v.[x := v]s}\ \text{alpha\_conv } v}{binder(\lambda x.s) = binder(\lambda v.[x := v]s)}\ \text{cong\_function}$$

Figure 3.7: gen_alpha_conv.

Later we will need a rule called prove_ass. This rule tries to remove an assumption from a theorem by proving it from another theorem. The rule takes two theorems as premises and make a new theorem. The assumptions of the conclusion is the union of the assumptions of the two premises minus the conclusion of the first premise. The consequent of the conclusion is the consequent of the second premise. This rule is only of any value if the consequent of the first premise is alpha convertible with an assumption of the second premise and the consequent of the first premise does not occur in its assumptions. It is derived as follows:

$$\frac{\dfrac{\Gamma \to tm_1 \qquad \Delta \to tm_2}{(\Gamma - \{tm_2\}) \cup (\Delta - \{tm_1\} \to tm_1 = tm_2}\ \text{deduct\_antisym} \qquad \Gamma \to tm_1}{\Gamma \cup (\Delta - \{tm_1\}) \to tm_2}\ \text{eq\_mp}$$

Figure 3.8: prove_ass.

Reference: The definitions of the derived rules are from [9], alpha equality is from [14].

3.3 Reductions

In lambda calculus there are three kind of reductions:

- α-conversion: This conversion changes the name of bound variables.
- β-reduction: This reduction apply arguments to abstractions.
- η-conversion: This conversion expresses extensionality.

We have already described the α- and η-conversions. It counts for both these rules that they convert one or more terms into other terms in a one-to-one

manner. In opposite to this the β-reduction tries to remove one or more terms by reducing two terms into one. To understand β-reduction, we first have to introduce a β-redex. A β-redex is a term on the form $((\lambda x.A(x))t)$, where $A(x)$ might contain x. A β-reduction can be executed on this kind of terms only. A term that contains no β-redeces is said to be in beta normal form. A β-reduction will reduce a beta redex as following:

$$((\lambda x.A(x))t) \rightarrow A(t)$$

$A(t)$ is the result of substituting x with t in $A(x)$. A derivation tree for beta reduction can easily be made from beta conversion.

$$\dfrac{\dfrac{}{\rightarrow (\lambda x.A(x))t = [x := t]A(x)}\ \text{beta\_conv} \qquad \Gamma \rightarrow (\lambda x.A(x))t}{\Gamma \rightarrow [x := t]A(x)}\ \text{eq\_mp}$$

Figure 3.9: beta_red.

When the instantiation is carried out we get $[x := t]A(x) \rightarrow A(t)$. Thereby it is derived that a single beta reduction on a beta redex can be performed in *HOLPro*.
If a reduction strategy is applied, beta reduction can be implemented to keep doing beta reductions on the result of a previous beta reduction (or the premise for the first beta reduction) until the reduction strategy cannot be applied any longer. A reduction strategy is a way to prioritise which β-redex should be reduced first if more than one can be reduced. One such strategy is normal order reduction. The principle of normal order reduction is to reduce the outermost, leftmost β-redex first. An example is

$$\lambda x_0 ... \lambda x_{(i-1)}.(\lambda x_i.(A(x_i)))M_1 M_2 ... M_n \rightarrow \lambda x_0 ... \lambda x_{(i-1)}.A(M_1)M_2 ... M_n$$

When a β-redex is the outermost, leftmost β-redex, then it is said to be in head position. If a β-redex is not in head position, it is said to be internal. If a term do not contain any β-redeces in head position, then the term is said to be in head normal form. This reduction strategy is complete, which means that the result of applying a β-reduction using this strategy will always be a term in normal head form. In other words all possible reductions will have been done. As an example of another reduction strategy we can describe applicative order reduction. In this reduction strategy the internal redeces are reduced first. However this strategy is not guaranteed to terminate and therefore not complete. In *HOLPro* normal order reduction is used. The beta reduction implemented will therefore continue until it reaches a term in normal head form.
References: [14].

3.4 Logical connectives

3.4.1 Definitions of the connectives

If we can define the usual logical connectives by using equality and the earlier presented rules, we can use these connectives in *HOLPro*. Notice in the following rules that lower case letters like p are used for representing predicates (can be any kind of formula of (range) type boolean), and upper case letters like P are used for representing lambda expressions of range type boolean. Furthermore what is implemented as an application with lambda expression P applied to term t will be written as $P(t)$. Remember that the definitions will be in intuitionistic logic, which makes them more complicated than if classical logic was used. First we will define true:

$$\top \equiv (\lambda x.x) = (\lambda x.x) \tag{3.1}$$

This definition says that an abstraction equals the same abstraction will always be true.
The next definition is for conjunction:

$$\wedge \equiv \lambda p.\lambda q.(\lambda f.fpq) = (\lambda f.f\top\top) \tag{3.2}$$

The definition says that a conjunction is true when an arbitrary function applied to the two input predicates p and q is equal to the same function applied to two times true. This implies that p and q must both be true, which is the usual condition for making a conjunction true.
The next logical constant is implication:

$$\Rightarrow \equiv \lambda p.\lambda q.p \wedge q = p \tag{3.3}$$

The definition says that implication is true when $p \wedge q$ is equal to p. If p implies q, then q adds nothing to what we already have, which is p. Therefore $p \wedge q$ will be the same as just p.
Now the universal quantifier will be defined:

$$\forall \equiv \lambda P.P = \lambda x.\top \tag{3.4}$$

The universal quantifier is true if the input abstraction P is equal to the function that is always true. Notice that quantifiers are used on abstractions only.
The existential quantifier is defined as follows:

$$\exists \equiv \lambda P.\forall q.(\forall x.P(x) \Rightarrow q) \Rightarrow q \tag{3.5}$$

This definition might be the hardest one to understand. But first look at $(\forall x.P(x) \Rightarrow q)$. This formula says that if P holds for some element x, then

q also holds. We can turn this around to state that if q is true, then P holds for some element x. Therefore if q can be implied from $(\forall x.P(x) \Rightarrow q)$, then P must hold for at least one element x. This holds no matter what q is, why q is universally quantified.
Disjunction is defined as follows:

$$\vee \equiv \lambda p.\lambda q.\forall r.(p \Rightarrow r) \Rightarrow ((q \Rightarrow r) \Rightarrow r) \tag{3.6}$$

If $p \vee q \Rightarrow r$, then if both p and q implies r, r will always be implied from $p \vee q$. Again no matter what r is, this holds, why r is universally quantified.
False is defined as follows:

$$\bot \equiv \forall p.p \tag{3.7}$$

The definition for false says that if false does not exist, then false is not a boolean value and $\forall p.p$ will be true. If false does exist, then p will be false at some point and $\forall p.p$ will be false.
Now we define negation:

$$\neg \equiv \lambda t.t \Rightarrow \bot \tag{3.8}$$

When $\lambda t.t$ becomes true it implies false.
In the end we define unique existential quantifier. This quantifier is true if and only if exactly one element makes a formula true.

$$\exists! \equiv \lambda P.\exists P \wedge \forall x.\forall y.P(x) \wedge P(y) \Rightarrow (x = y) \tag{3.9}$$

The formula can be split in two parts by the first conjunction. In the first part it is checked if $\exists P$ is true. This makes sure that at least one element makes P true. In the second part it is checked that at most one element makes P true. This part says that if two elements both make P true, then they must be the same element.
References: [8]

3.4.2 Derived boolean rules

Now that the logical connectives have been defined, they can be used for making derived boolean rules. Since the derivation trees are large, they will be displayed on separate flipped pages.
For the truth constant we will make derived rules for proving

- $\rightarrow T$ from the definition of truth.
- $\Gamma \rightarrow tm$ from $\Gamma \rightarrow tm = T$. This derived rule eliminates a truth constant from a theorem.

- $\Gamma \rightarrow tm = T$ from $\Gamma \rightarrow tm$. This derived rule introduces a truth constant from a theorem.

For the conjunction constant we will implement derived rules for proving

- $\Gamma \cup \Delta \rightarrow tm_1 \wedge tm_2$ from $\Gamma \rightarrow tm_1$ and $\Delta \rightarrow tm_2$. This derived rule says that a conjunction of two theorems will be true if both the premises are true.
- $\Gamma \rightarrow l$ from $\Gamma \rightarrow l \wedge r$. This derived rule proves that if a conjunction is true then the left part of the conjunction alone is also true.
- $\Gamma \rightarrow r$ from $\Gamma \rightarrow l \wedge r$. This derived rule is the counter part of the previous for the right part of a conjunction.

The derivation trees for the last two derived rules with conjunction are very similar. Only the derivation tree for the left part of a conjunction will therefore be displayed. The counterpart for the right side of a conjunction is easy to get from this derivation tree. In the derivation tree in 3.19 replace the term at the star with the term $\lambda p.\lambda q.q$.
For the implication constant we will implement the regular modus ponens rule. This rule proves

- $\Gamma \cup \Delta \rightarrow q$ from $\Gamma \rightarrow p \Rightarrow q$ and $\Delta \rightarrow p$.

The regular modus ponens rule (which is implication-based) can be used when a term just implies another term. In the earlier presented equality-based modus ponens rule two terms had to be equal. Or in other words: both terms have to imply the other term. Hence the implication-based modus ponens rule can be used on weaker logical statements. The derivation tree 3.22 proves that the implication-based modus ponens rule is valid in $HOLPro$.
References: [9]

Derivation trees for the truth connective.

$$\dfrac{\dfrac{\dfrac{}{\to T = (\lambda p.p = \lambda p.p)}\ \text{definition}}{\to (\lambda p.p = \lambda p.p) = T}\ \text{sym} \qquad \dfrac{}{\to \lambda p.p = \lambda p.p}\ \text{refl}}{\to T}\ \text{eq\_mp}$$

Figure 3.10: Derivation tree for truth is always true (truth).

$$\dfrac{\dfrac{\Gamma \to tm = T}{\Gamma \to T = tm}\ \text{sym} \qquad \dfrac{}{\to T}\ \text{truth}}{\Gamma \to tm}\ \text{eq\_mp}$$

Figure 3.11: Derivation tree for eliminating truth (tt_elim).

$$\dfrac{\dfrac{\dfrac{}{\{t = T\} \to t = T}\ \text{assume}}{\{t = T\} \to t}\ \text{tt\_elim} \qquad \dfrac{\dfrac{}{\{t\} \to t}\ \text{assume}}{\{t\} \to t = T}\ \text{deduct\_antisym}}{\to t = (t = T)}\ \text{deduct\_antisym} \qquad \to T$$

Figure 3.12: Derivation tree for introducing truth (tt_intro_{top}).

$$\dfrac{\dfrac{tt\_intro_{top}}{\to tm = (tm = T)}\ \text{inst } t \to tm \qquad \Gamma \to tm}{\Gamma \to tm = T}\ \text{eq\_mp}$$

Figure 3.13: Derivation tree for introducing truth (tt_intro).

Derivation trees for the conjunction connective.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\to \wedge = (\lambda p.\lambda q.((\lambda f.fpq) = (\lambda f.fTT)))}\ \text{definition}\quad p}{\to p\wedge = (\lambda p.\lambda q.((\lambda f.fpq) = (\lambda f.fTT))p)}\ \text{cong\_para}\quad q}{\to p \wedge q = (\lambda p.\lambda q.((\lambda f.fpq) = (\lambda f.fTT))pq)}\ \text{cong\_para}}{\to p \wedge q = ((\lambda f.fpq) = (\lambda f.fTT))}\ \text{beta\_red}}{\to ((\lambda f.fpq) = (\lambda f.fTT)) = p \wedge q}\ \text{sym}$$

Figure 3.14: conj_1.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{p\} \to p}\ \text{assume}}{\{p\} \to p = T}\ \text{tt\_intro}}{\{p\} \to fp = fT}\ \text{cong\_func}\qquad \dfrac{\dfrac{}{\{q\} \to q}\ \text{assume}}{\{q\} \to q = T}\ \text{tt\_intro}}{\{p\} \cup \{q\} \to fpq = fTT}\ \text{cong}}{\{p\} \cup \{q\} \to (\lambda f.fpq) = (\lambda f.fTT)}\ \text{abs 'f'}$$

Figure 3.15: conj_2.

$$\dfrac{\Delta \to tm_2 \qquad \dfrac{\Gamma \to tm_1 \qquad \dfrac{\dfrac{conj_1 \qquad conj_2}{\{p\} \cup \{q\} \to p \wedge q}\ \text{eq\_mp}}{\{tm_1\} \cup \{tm_2\} \to tm_1 \wedge tm_2}\ \text{inst } p \to tm_1, q \to tm_2}{\Gamma \cup \{tm_2\} \to tm_1 \wedge tm_2}\ \text{prove\_ass}}{\Gamma \cup \Delta \to tm_1 \wedge tm_2}\ \text{prove\_ass}$$

Figure 3.16: Derivation tree for making a conjunction from two theorems (conj).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\to \wedge = (\lambda p.\lambda q.(\lambda f.fpq) = (\lambda f.fTT)) \quad P}{\to P \wedge = (\lambda p.\lambda q.(\lambda f.fpq) = (\lambda f.fTT))P}\ \text{cong\_para}}{\to P \wedge = (\lambda q.(\lambda f.fPq) = (\lambda f.fTT))}\ \text{beta\_red} \quad Q}{\to P \wedge Q = (\lambda q.(\lambda f.fPq) = (\lambda f.fTT))Q}\ \text{cong\_para}}{\to P \wedge Q = ((\lambda f.fPQ) = (\lambda f.fTT))}\ \text{beta\_red}$$

Figure 3.17: conj_left$_{top1}$

$$\dfrac{\dfrac{\dfrac{\dfrac{conj\_left_{top1} \quad \dfrac{}{\{P \wedge Q\} \to P \wedge Q}\ \text{assume}}{\{P \wedge Q\} \to (\lambda f.fPQ) = (\lambda f.fTT)}\ \text{eq\_mp} \quad \lambda p.\lambda q.p*}{\{P \wedge Q\} \to (\lambda f.fPQ)(\lambda p.\lambda q.p) = (\lambda f.fTT)(\lambda p.\lambda q.p)}\ \text{cong\_para}}{\{P \wedge Q\} \to P = T}\ \text{beta\_red}}{\{P \wedge Q\} \to P}\ \text{tt\_elim}$$

Figure 3.18: conj_left$_{top2}$

$$\dfrac{\Gamma \to l \wedge r \quad \dfrac{conj\_left_{top2}}{\{l \wedge r\} \to l}\ \text{inst } P \to l, Q \to r}{\Gamma \to l}\ \text{prove\_ass}$$

Figure 3.19: Derivation tree for proving the left side of a conjunction (conj_left)

Derivation trees for the implication connective.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\to \Rightarrow = (\lambda p.\lambda q.p \wedge q = p)}\quad p}{\to p \Rightarrow = (\lambda p.\lambda q.p \wedge q = p)p}\ \text{cong\_para}\quad q}{\to p \Rightarrow q = (\lambda p.\lambda q.p \wedge q = p)pq}\ \text{cong\_para}}{\to p \Rightarrow q = (p \wedge q = p)}\ \text{beta\_red}\qquad \dfrac{}{\{p \Rightarrow q\} \to p \Rightarrow q}\ \text{assume}}{\{p \Rightarrow q\} \to p \wedge q = p}\ \text{eq\_mp}$$

Figure 3.20: mp_{top1}

$$\dfrac{\dfrac{\dfrac{mp_{top1}}{\{p \Rightarrow q\} \to p = p \wedge q}\ \text{sym}\qquad \dfrac{}{\{p\} \to p}\ \text{assume}}{\{p \Rightarrow q\} \cup \{p\} \to p \wedge q}\ \text{eq\_mp}}{\{p \Rightarrow q, p\} \to q}\ \text{conj\_right}$$

Figure 3.21: $\text{mp}_{top}2$

$$\dfrac{\Delta \to ant \qquad \dfrac{\Gamma \to ant \Rightarrow con \qquad \dfrac{mp_{top2}}{\{ant \Rightarrow con, ant\} \to con}\ \text{inst } p \to ant, q \to con}{(\Gamma \cup \{ant\}) \to con}\ \text{prove\_ass}}{\Delta \to con}\ \text{prove\_ass}$$

Figure 3.22: Derivation tree for the regular modus ponens rule (mp).

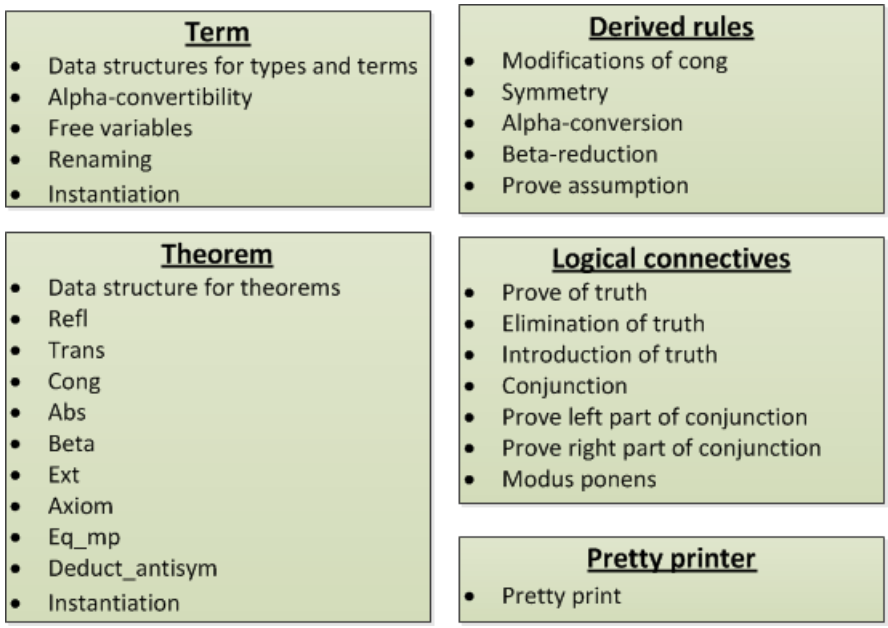
Chapter 4

Implementation

In this chapter it is explained how the proof assistant has been implemented. The chapter does not contain a complete walk through of the code, but gives a brief overview of the implementation. The implementation is in SWI-Prolog.

4.1 Design

The implementation has been split into several files. Each file contains the functionality for a part of the *HOLPro* formal proof system. On 4.1 it is shown which main predicates have been implemented for each file in the implementation of *HOLPro*. By main predicate is meant a predicate that performs some key functionality described in the presentation of the system. Often these predicates have helper predicates to help perform the desired functionality, but these predicates are not shown here to give a better overview of the system.



4.2 Data structures for terms

In this section the chosen data structures and basic predicates for terms will be presented. Terms are similar to formulas in the theory chapter. In the higher-order logic used, a term can have three types: boolean, individual and function. In the implementation this is defined as

```
% Types: booleans, individuals and functions

type(bool).
type(ind).
type(fun(X,Y)) :- type(X),type(Y).
```

The recursive definition for a function makes a term able to be of an arbitrary function. This means that the domain and range of a function can be booleans and individuals as well as other functions or a mixture.
When working with the function type in the code, it is beneficial to have some

helper predicates that can do often-used-operations. Four predicates for these have been implemented.

```
%Predicates to work with functions

create_fun(S,T,fun(S,T)).
dest_fun(fun(S,T),S,T).

domaintype(F,T) :- dest_fun(F,T,_).
rangetype(F,T) :- dest_fun(F,_,T).
```

The first two predicates respectively creates a function and breaks one down into its domain type and range type. The last two predicates find the domain or the range type.
There are four kind of terms: constant, variable, application and abstraction. This leads to the following definition of a term:

```
% Terms: constants, variables, applications and abstractions
term(const(S,X)) :- atom(S), type(X).
term(var(S,X)) :- atom(S), type(X).
term(app(T1,T2)) :- term(T1), term(T2).
term(abs(var(_ ,_ ),T2)) :- term(T2).
```

$const(S, X)$ models a constant S with the type X. $var(S, X)$ models a variable S with the type X. $app(T1, T2)$ models an application with the term $T1$ applied to the term $T2$. $abs(var(\_, \_), T2)$ represents an abstraction with a variable and the term $T2$. For all the definitions it is checked whether the given input is valid. This is done by the predicates atom (for a variable name), type (for a type) and term (for a term).
Just as there are helper predicates for types, there are also defined some helper predicates for working with terms. There is one for finding the type of a term, one for creating a term and one for breaking it down into its subcomponents.

```
%Gives the type of a term
```

```
type_of(const(_,X),X).
type_of(var(_,X),X).
type_of(app(S,_),Ty) :-
        type_of(S,Z),
        rangetype(Z,Ty).
type_of(abs(S,T),Ty) :-
        type_of(S,TyS),
        type_of(T,TyT),
        create_fun(TyS,TyT,Ty).
%Otherwise it's a logical constant
type_of(_,bool).

%Creates a term
create_const(S,Ty,Z) :-
        Z=const(S,Ty),
        term(Z).
create_var(S,Ty,Z) :-
        Z=var(S,Ty),
        term(Z).
create_app(S,T,Z) :-
        Z=app(S,T),
        term(Z),
        type_of(S,TyS),
        type_of(T,TyT),
        domaintype(TyS,DomS),
        TyT=DomS.
create_abs(S,T,Z) :-
        Z=abs(S,T),
        term(Z).

%Destroys a term
dest_const(const(S,Ty),S,Ty).
dest_var(var(S,Ty),S,Ty).
dest_app(app(S,T),S,T).
dest_abs(abs(S,T),S,T).
```

4.3 Implementation of the functionality

Besides the already shown functionality for terms we will implement:

- Alpha convertibility.
- Search for the free variables of a term.
- Renaming of a variable. This is needed for instantiation to avoid a free variable becoming bound.
- Instantiation of a variable with an arbitrary term. This is done recursively over the structure of a term.
- Constructor and destroyer for equalities. This will be convenient, since the axioms of *HOLPro* are built up around equality, and equalities therefore are used a lot.

The next in the implementation is to define the axioms and derived rules that is the backbone of the deductive system in *HOLPro*. Since theorems are used a lot, a special data structure for them has been implemented with a constructor and a destroyer.

```
create_theorem(Assumptions,Term,theorem(Assumptions,Term)).

dest_theorem(theorem(Assumptions,Term),Assumptions,Term).
```

First the axioms are implemented. Then they can be used to implement the derived rules of *HOLPro*.
Next step is to implement the usual logical connectives from the deductive system implemented so far. In general the logical connectives are implemented like constants with the following types:

- *o* for true and false
- *oo* for unary operators and binders.
- *ooo* for binary operators

Notice that a binder has to be used on an abstraction with the range type boolean. For each logical connective except true and false a constructor has been implemented that makes a term consisting of the logical connective and the arguments given. For instance for conjunction the predicate create_conj(P,Q,Result) will give as result the term $P \wedge Q$. Some of the logical connectives are used for

implementing more derived rules. For instance is implication used for implementing the ordinary modus ponens rule. Unlike the modus ponens rule used so far (eq_mp) this one is not based on equality but implication.

4.4 Pretty printer

The main task for the pretty printer is to print the types and terms in a more readable way than just printing the internal structure as it is implemented. The pretty printer outputs the different terms in the following way:

term	**print**
const(tm,type)	const_tm_type
var(tm,type)	tm_type
application(tm1,tm2)	tm1 tm2
abstraction(v,tm)	(\v.tm)

Notice that the type of the constants and variables will be printed after a _ (underscore) of the term itself. The types will be printed in the following way:

type	**print**
bool	'bool'
ind	'ind'
fun(X,Y)	X_Y

Furthermore the printer will recognize the logical connectives. Their prints will be:

constant	**print**
$\top$	tt
$\bot$	ff
$\neg tm$	$\sim tm$
$tm_1 \wedge tm_2$	$(tm_1 / \backslash tm_2)$
$tm_1 \vee tm_2$	$(tm_1 \backslash / tm_2)$
$tm_1 \Rightarrow tm_2$	$(tm_1 ==> tm_2)$
$\forall abs(v, tm)$	$(!v.tm)$
$\exists abs(v, tm)$	$(?v.tm)$
$\exists! abs(v, tm)$	$(?!v.tm)$

Chapter 5

Tests

To test the implemented formal proof system of *HOLPro* structural tests or white-box tests have been used. Structural testing requires that every little part of the system is tested. Therefore tests for all predicates but the simplest ones have been implemented. The tests are grouped according to the design of *HOLPro* as shown in 4.1.
In the following results from running the tests will be shown. The tests are formatted like the following:

Testing predicates in [file f_1]

Testing [predicate p_1]
[Test t_1]
[Test t_2]
...
[Test t_n]

A single test is formatted like

Testing [predicate p_1]

Input: [argument a_1]
[argument a_2]
...
[argument a_n]
Result: [result from p_1]

When the tests for a file is displayed, it is at the top written, which file is being tested. After this each predicate is tested. Some predicates are only tested once while others are tested several times. The test of a predicate is displayed by first stating the arguments given to the predicate. If more than one argument is needed for a test, the arguments will be split by linebreaks. After the arguments have been displayed, the result is displayed. If special types of arguments are required to test a predicate (for instance the instantiation of a theorem need a theorem to instantiate and a mapping), it is written which argument is what. For instance the tests of instantiation of a theorem is displayed like

```
Testing instantiation
Input: Mapping: (y_ind-->x_ind),(z_ind-->u_ind)
       Theorem: [x_ind]-->(\x_ind.y_ind) z_ind
Result: [x3_ind]-->(\x3_ind.x_ind) u_ind
Input: Mapping: (x_ind-->y_ind),(z_ind-->u_ind)
       Theorem: [x_ind]-->(\x_ind.x_ind) z_ind
Result: [x_ind]-->(\x_ind.x_ind) u_ind
```

The purpose of the implementation was to show the validity of the modus ponens rule. To give an example of how the tests should be understood, we therefore consider this rule. The test of the modus ponens rule is displayed like

```
Testing modus ponens
Input: [a_ind]-->(p_bool==>q_bool)
       [b_ind]-->p_bool
Result: [a_ind,b_ind]-->q_bool
```

As arguments we give $a \to p \Rightarrow q$ and $b \to p$. From this using the modus ponens rule we can conclude $a \cup b \to q$. The predicate for the modus ponens rule is

therefore considered correctly implemented, though we can never be completely sure of this. Note that the comma in the test results should be understood as the union set operator.
To view the results of all the implemented tests would be a long and trivial reading. Hence it is only the test results for the most important predicates that will be shown. All the predicates are not tested directly below, but the ones that are not tested are used by other predicates that are. In this way all predicates are tested below, while keeping the tests needed to a minimum.

```
Testing predicates in term.pl

Testing alpha convertibility
Input: (\x_ind.x_ind)
       (\y_ind.y_ind)
Result: succes
Input: (\x_ind.x_ind)
       (\y_ind.z_ind)
Result: fail

Testing for free variables
Input: (\x_ind.y_ind)
Result: y_ind
Input: (\x_ind.x_ind)
Result:
Input: (\x_ind.x_ind) y_ind
Result: y_ind

Testing renaming of variables
Input: x_ind
Result: x13_ind

Testing instantiation of terms
Input: Mapping: (z_bool_bool-->a_bool_bool),(y_ind-->x_ind)
       Term: (\x_ind.z_bool_bool const_c_ind_bool y_ind)
Result: (\x14_ind.a_bool_bool const_c_ind_bool x_ind)
Input: Mapping: (x_ind-->y_ind),
       Term: (\x_ind.x_ind)
Result: (\x_ind.x_ind)

Testing predicates in theorem.pl

Testing refl
```

```
Input: x_ind
Result: []-->(x_ind=x_ind)

Testing trans
Input: [a_ind]-->(t_ind=u_ind)
Result: [a_ind]-->(s_ind=u_ind)

Testing cong
Input: [a_ind]-->(s_ind_ind=t_ind_ind)
       [b_ind]-->(u_ind=v_ind)
Result: [a_ind,b_ind]-->(s_ind_ind u_ind=t_ind_ind v_ind)

Testing abs
Input: Bound var: x_ind
       Theorem: [a_ind]-->(s_ind=t_ind)
Result: [a_ind]-->((\x_ind.s_ind)=(\x_ind.t_ind))

Testing beta
Input: (\x_ind.t_ind) x_ind
Result: []-->((\x_ind.t_ind) x_ind=t_ind)

Testing eta
Input: Bound var: x_ind
       Term: t_ind_ind
Result: []-->((\x_ind.t_ind_ind x_ind)=t_ind_ind)

Testing assume
Input: p_bool
Result: [p_bool]-->p_bool

Testing eq_mp
Input: [a_ind]-->(s_ind=t_ind)
       [b_ind]-->s_ind
Result: [a_ind,b_ind]-->t_ind

Testing deduct_antisym
Input: [t_ind]-->s_ind
       [s_ind]-->t_ind
Result: []-->(s_ind=t_ind)

Testing instantiation
Input: Mapping: (y_ind-->x_ind),(z_ind-->u_ind)
       Theorem: [x_ind]-->(\x_ind.y_ind) z_ind
Result: [x15_ind]-->(\x15_ind.x_ind) u_ind
```

```
Input: Mapping: (x_ind-->y_ind),(z_ind-->u_ind)
       Theorem: [x_ind]-->(\x_ind.x_ind) z_ind
Result: [x_ind]-->(\x_ind.x_ind) u_ind

Testing predicates in derived_rules.pl

Testing cong_function
Input: Function: f_ind_ind
       Equality: [a_ind]-->(u_ind=v_ind)
Result: [a_ind]-->(f_ind_ind u_ind=f_ind_ind v_ind)

Testing cong_parameter
Input: Equality: [a_ind]-->(f_ind_ind=g_ind_ind)
       Argument: u_ind
Result: [a_ind]-->(f_ind_ind u_ind=g_ind_ind u_ind)

Testing sym
Input: [a_ind]-->(l_ind=r_ind)
Result: [a_ind]-->(r_ind=l_ind)

Testing alpha conversion
Input: New name for the bound variable: y_bool
       Term: (\x_bool.x_bool)
Result: []-->((\x_bool.x_bool)=(\y_bool.y_bool))
Input: New name for the bound variable: y_bool
       Term: (!x_bool.x_bool)
Result: []-->((!x_bool.x_bool)=(!y_bool.y_bool))

Testing prove assumption
Input: [a_ind]-->t1_ind
       [t1_ind]-->t2_ind
Result: [a_ind]-->t2_ind
Input: [a_ind]-->t1_ind
       [b_ind]-->t2_ind
Result: [b_ind]-->t2_ind

Testing beta reduction
Input: [a_ind]-->(\x_ind.x_ind) y_ind
Result: [a_ind]-->y_ind
Input: [a_ind]-->(\p_bool.(\q_bool.(
\f_bool_bool_bool.f_bool_bool_bool p_bool q_bool))) s_bool t_bool
Result: [a_ind]-->(\f_bool_bool_bool.f_bool_bool_bool s_bool t_bool)

Testing predicates in logical_connectives.pl
```

```
Testing prove of truth
Result: []-->tt

Testing elimination of truth
Input: [a_ind]-->(p_bool=tt)
Result: [a_ind]-->p_bool

Testing introduction of truth
Input: [a_ind]-->p_bool
Result: [a_ind]-->(p_bool=tt)

Testing conj
Input: [a_ind]-->s_bool
       [b_ind]-->t_bool
Result: [a_ind,b_ind]-->(s_bool/\t_bool)

Testing conj_left
Input: [assumption_bool]-->(l_bool/\r_bool)
Result: [assumption_bool]-->l_bool

Testing conj_right
Input: [assumption_bool]-->(l_bool/\r_bool)
Result: [assumption_bool]-->r_bool

Testing modus ponens
Input: [a_ind]-->(p_bool==>q_bool)
       [b_ind]-->p_bool
Result: [a_ind,b_ind]-->q_bool
```

All the implemented tests can be seen in appendix B.

Chapter 6

Discussion

6.1 From intuitionistic to classical logic

So far we have been working with intuitionistic logic. To make the logic classical we need two axioms: eta-conversion and the axiom of choice. Eta-conversion can be described as

$$\lambda x.(fx) = f, x \notin FV(f) \tag{6.1}$$

where $FV(f)$ is the set of free variables in the term f. This axiom is the same as the axiom eta and therefore already present in $HOLPro$.
To help explain the axiom of choice we define a choice function f. This function is defined on a set C of nonempty sets. For every set $s \in C$, f selects an element in s. $f(s)$ is therefore an element in s. The axiom of choice says that for every set C of nonempty sets, there exists such a choice function f defined on C. In an analogy we could imagine having a set of nonempty buckets (can be infinite) containing balls. Then the axiom of choice says that we can select exactly one ball from each bucket using the choice function for each bucket. We can define the axiom of choice using this choice function and the implication connective. The axiom is defined as

$$\forall P.\forall x.P(x) \Rightarrow P(f(P)) \tag{6.2}$$

The axiom says that if any element x is contained in the set P, then it is possible to select an element in P. In the implementation a set P will be an abstraction

with boolean values as its range domain, and the elements contained in P are those that make P true. Then the axiom of choice says, that if an element x makes P true, then we can select an element contained in P that makes P true using the choice function.
Reference: [3]

6.1.1 Law of excluded middle

The implementation chapter ends with the derivation of the regular modus ponens rule. This might seem as an arbitrary place to stop the implementation. Why do we not continue deriving rules for the different logical connectives? The answer is the law of excluded middle. In intuitionistic logic the law of excluded middle is disregarded. However it can be shown that if the axiom of choice is accepted, then the law of excluded middle can be derived in intuitionistic logic using the modus ponens rule and the eta axiom. The proof of this goes as follows. First we consider two sets A and B containing a term t. t can be any formula as defined in *HOLPro*. The purpose of the proof will be to show that t is either true or false. The two sets are defined like this:

$$A = \{x | (x = \bot) \vee t)\} \tag{6.3}$$
$$B = \{x | (x = \top) \vee t\} \tag{6.4}$$

From this we can conclude that $\bot \in A$ and $\top \in B$ no matter what t is. Since we now know that both A and B contain at least one element, they are nonempty. Therefore the following formula must be true

$$\forall s.s \in \{A, B\} \Rightarrow \exists y.y \in s \tag{6.5}$$

This equation says that if s is either A or B, then an element exists in the set. But how do we get this element? This is where the axiom of choice becomes useful. We can use it to define a choice function f

$$f = \lambda s.\exists y.y \in s \tag{6.6}$$

By inserting 6.6 into 6.5 we get

$$\forall s.s \in \{A, B\} \Rightarrow f(s) \in s \tag{6.7}$$

From 6.7 we can see that $f(A) \in A$ and $f(B) \in B$. Together with 6.4 this gives us

$$\begin{aligned}(f(A) = \bot \vee t) \wedge (f(B) = \top \vee t) \Leftrightarrow \\ t \vee (f(A) = \bot \wedge f(B) = \top) \Rightarrow \\ t \vee f(A) \neq f(B)\end{aligned} \tag{6.8}$$

We can see from 6.4 that if t is true then $A = B$. Hence we got

$$t \Rightarrow (\forall x.A(x) = B(x)) \tag{6.9}$$

To get further with the proof extensionality is needed. From the eta axiom we can get the formula

$$\forall X.\forall Y.\forall F.(\forall x.X(x) = Y(x)) \Rightarrow (f(X) = f(Y)) \tag{6.10}$$

6.10 is known as extensionality of functions. It says that if two predicates always give the same result when they are applied to the same argument, then a function F will give the same result when applied to the two predicates. When 6.10 is applied to the sets A and B it says that if the two sets are equal, then $f(A) = f(B)$. Using the law of contraposition we get

$$f(A) \neq f(B) \Rightarrow \neg t \tag{6.11}$$

Note that the law of contraposition is only accepted in one direction in intuitionistic logic: $(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$. By insertion of 6.11 into 6.8 and use of the modus ponens rule we get

$$t \vee \neg t \tag{6.12}$$

Which is the law of excluded middle.
References: The proof procedure is from [5] and [3]

6.1.2 Redefining the logical connectives

With the law of excluded middle derived, classical reasoning can be used. This means that the logical connectives defined after implication can be redefined in classical logic. The benefit of defining the logical connectives in classical logic is, that the definitions are simpler for the connectives that have an opposite connective (for instance $\top$ and $\bot$), because these connectives can be defined in terms of the counterpart. To do this however, we need to define the connectives in a different order, as negation will be a much used connective in the new

definitions. The definitions in classical logic are

$$\top \equiv (\lambda x.x) = (\lambda x.x) \tag{6.13}$$
$$\wedge \equiv \lambda p.\lambda q.(\lambda f.fpq) = (\lambda f.f\top\top) \tag{6.14}$$
$$\Rightarrow \equiv \lambda p.\lambda q.p \wedge q = p \tag{6.15}$$
$$\neg \equiv \lambda t.t \Rightarrow \bot \tag{6.16}$$
$$\forall \equiv \lambda P.P = \lambda x.\top \tag{6.17}$$
$$\exists \equiv \lambda P.\neg(\forall\neg P) \tag{6.18}$$
$$\vee \equiv \lambda p.\lambda q.\neg(\neg p \wedge \neg q) \tag{6.19}$$
$$\bot \equiv \neg\top \tag{6.20}$$
$$\exists! \equiv \lambda P.\exists P \wedge \forall x.\forall y.P(x) \wedge P(y) \Rightarrow (x = y) \tag{6.21}$$

It is only $\exists$, $\vee$ and $\bot$ that have their definitions changed. When comparing with the intuitionistic definitions it is also the three connectives that are hardest to define using intuitionistic logic. Note that $\vee$ can be defined in a less intuitive but simpler formulation that only contains implication:

$$\vee \equiv \lambda p.\lambda q.(p \Rightarrow q) \Rightarrow q \tag{6.22}$$

All the classical definitions can be verified by truth tables.

6.2 The natural numbers

In *HOLPro* the natural numbers will be defined as an infinite set. Hence to get the natural numbers we need to add the axiom of infinity. The axiom expresses two properties:

- The one-to-one property: a function F exists that has to be a one-to-one function. This means that one value in the domain maps to one value in the range and vice versa.
- The onto property: There exists a value y that is not contained in the range of the function F. This means that no matter what x is in $F(x)$, $F(x) = y$ can never be true.

Based on these two properties the axiom of infinity is defined as

$$\exists F.(\forall x.\forall y.F(x) = F(y) \Rightarrow x = y) \wedge (\exists y.\forall x.y \neq F(x)) \tag{6.23}$$

The axiom of infinity can be instantiated with a successor function $S(x)$ for F. The result of $S(x)$ is one number higher than x and will be noted x'. For

instance $S(0) = 1$. The successor function meets the two properties expressed by the axiom of infinity. The function maps in a one-to-one style and 0 is not in the range of S. Now we know that an infinite set exists that contain natural numbers. But we do not know if the set contains all the natural numbers. To know this we need the axiom of induction:

$$(F(0) \wedge (\forall x.(F(x) \Rightarrow F(x'))) \Rightarrow \forall x.F(x) \tag{6.24}$$

x' is the successor of x. The axiom of induction says that $F(x)$ holds for all x if

- $F(0)$ is true. This is the base case.
- $F(x')$ can be proven true if $F(x)$ is true. This is the induction step.

To show that S can be used to produce all natural numbers, we must show the base case and induction step. The base case can be shown by

$$S(0) = 1 \tag{6.25}$$

Now to the induction step. We assume $S(x) = x'$ and have to show $S(x') = x''$. However this follows from the definition of S:

$$S(x') = x'' \tag{6.26}$$

It is thereby shown that $\forall x.S(x) = x'$. If $x \in \mathbb{N}$ then 0 has to be a natural number and the natural numbers have to be closed under S. This is guaranteed through the axioms:

$$0 \in \mathbb{N} \tag{6.27}$$

$$(x \in \mathbb{N}) \wedge (S(x) = x') \Rightarrow (x' \in \mathbb{N}) \tag{6.28}$$

We can now conclude that $S(x) = x'$ is true for all natural numbers and therefore can produce all natural numbers.

With the introduction of the natural numbers we get Heyting arithmetic if we disregard the law of excluded middle (and thereby the axiom of choice). If we accept the law of excluded middle then we get Peano arithmetic. In Peano arithmetic we can define the arithmetic operators as functions. For instance addition can be defined as the function $+_{\iota\iota\iota}$ with the definition:

$$\begin{aligned} x + 0 &= x \\ x + S(y) &= S(x + y) \end{aligned} \tag{6.29}$$

References: The axiom of infinity and definition of the natural numbers is from [11] and [9]. Peano arithmetic is from [13].

6.3 Reducing the number of axioms

HOLPro is built on ten axioms that are assumed valid without a proof. However this number can be reduced, as it is possible to derive some of the axioms from other axioms. This can be shown for the axioms trans and refl. The derivation tree for refl is quite intuitive. First a theorem is made which conclusion is $t = t$. Then the assumptions can be removed. This is shown in 6.1.

$$\dfrac{\dfrac{}{\{t\} \to t}\ \text{assume} \qquad \dfrac{}{\{t\} \to t}\ \text{assume}}{\to t = t}\ \text{deduct\_antisym}$$

Figure 6.1: The derivation tree for a derived refl.

Trans is not as intuitive to derive as refl was, but it is still possible. 6.3 shows this. The derivation tree has been split into two to fit the page.

$$\dfrac{\dfrac{\dfrac{}{(=) = (=)}\ \text{refl} \qquad \dfrac{\Gamma \to s = t}{\Gamma \to t = s}\ \text{sym}}{(t =) = (s =)}\ \text{cong} \qquad \Delta \to t = u}{\Gamma \cup \Delta \to (t = t) = (s = u)}\ \text{cong}$$

Figure 6.2: trans_{top}

$$\dfrac{trans_{top} \qquad \dfrac{}{\to t = t}\ \text{refl}}{\Gamma \cup \Delta \to s = u}\ \text{deduct\_antisym}$$

Figure 6.3: The derivation tree for a derived trans.

In the derivation of trans the derived rule sym is used. This is not an axiom but is used for convenience. It can be replaced with the basic rules in its derivation tree (3.3). The only factor that can make the use of sym invalid in the derivation tree for trans, is if trans itself is used in the derivation tree for sym. Then the derivation tree for trans would be infinitely large. However this is not the case, why sym can be used as an abbreviation for a special combination of axioms.

6.4 Data structure for terms

A possibility to enhance the data structure for terms would be to use de Bruijn indexes. When using de Bruijn indexes a variable is represented by a number instead of a string name. The number is decided by which bound variable it refers to. If it is the bound variable of the innermost abstraction it is represented by a 1. For instance the identity function $\lambda x.x$ would be $\lambda 1$, as the variable refers to the bound variable closest to it. If it is the bound variable of the second innermost abstraction then it would be represented by a 2 and so on. If a variable is represented by a number higher than the number of embedded abstractions, then it is a free variable. Some examples are

- $\lambda x.\lambda y.yx$ would be $\lambda\lambda 1\ 2$.
- $\lambda x.(\lambda y.y)x$ would be $\lambda(\lambda 1)1$.
- $\lambda x.y$ would be $\lambda 2$.

A benefit from using de Bruijn indexes is that α-conversion is not needed, because two α-convertible terms would actually be exactly the same terms. Therefore α-equality would be reduced to just normal equality. Furthermore the fear of binding free variables during a substitution would be gone. However if only a number is used to represent a variable, it can not have any type. If a number is used to represent just the name of a variable, then the user-friendliness would be lessened significantly, as a variable would not be possible to name. A user would therefore not know, what a variable's purpose is.
The above issue can be handled in two ways. Either a parser can be implemented for parsing a named expression to a nameless one and back, or a mixture of named and nameless variables can be used. Implementing a parser would maintain the full advantages of using de Bruijn indexes, while still having some degree of user-friendliness. The question is how much? When the parser goes from a nameless expression to a named one, it does not know which names for the variables would give sense. Therefore a named expression generated by a parser from a nameless expression can be just an incomprehensible as the nameless expression itself.
The other possibility is mixing string names and de Bruijn indexes. When a variable is free, we do not use the number it is represented by for anything but knowing that it is free. Therefore a free variable can be represented by a string while α-equality is still reduced to normal equality, and there will be no fear of binding free variables. This principle is called the locally nameless approach. This approach also requires a parser from a named expression and back again. However as the parser translates a named expression to a locally nameless expression instead of a completely nameless expression, a program would save the

names for the free variables. These names can be used when parsing from a locally nameless expression to a named expression, reducing the user-friendliness with as little as possible while still having the advantages of using de Bruijn indexes. This approach would be a possibility to implement if future work on this project is done.
References: De Bruijn indexes is from [4], Locally named expressions is from [10].

6.5 Choice of language

The implementation of the proof assistant was done in SWI-Prolog. It was a part of the project that the implementation should be in first order Prolog. However if the implementaion could have been in any language, some good alternative exist. In this section the pros and cons of different programming languages will be discussed.

SWI-Prolog is well fitted for working with logic. But this is the only benefit of SWI-Prolog. It is not possible to implement any security in SWI-Prolog. A programmer can write whatever is wanted whenever. Because of this a programmer can create data structures without using its constructor. Furthermore the only way offered by SWI-Prolog to structure a programme is to put it in different files. This is not a problem when implementing small algorithmic procedures, but it becomes a big problem when implementing a project on the scale of this bachelor thesis. The development environment for SWI-Prolog is basically notepad++. It gives no help or hints during the implementation, which forces the programmer to remember all predicates by heart or find the definition of it, every time the predicate is used.

The most used programming language in the domain of theorem provers is ML (often in the shape of OCaml or SML). ML handles some of Prolog's disadvantages. It is possible to create a constructor for a data structure that must be used when an instance of the data structure is made. This makes sure that no invalid instances of a data structure is created, and could have been very helpful for the implementation of types, terms and theorems. Furthermore it is well-suited for implementing algorithmic procedures and can give functions as arguments for other functions (which first order Prolog cannot). For the matter of structures, it can encapsulate functions and data structures in modules, thereby avoiding illegal use of functions. If F# (Microsoft's implementation of a functional language) is used, the programmer even get IntelliSense. IntelliSense

makes the programmer able to see what functions and data structures have been implemented so far and what arguments they have. F# could therefore be a good option when deciding on the programming language.

If more structure is wanted an object-oriented language like Java is a good option. Being able to implement objects gives security for data structures and functions. If a data structure is implemented as an object, its constructor has to be used to create an instance of this data structure. This is also possible in ML, but an object-oriented programming language gives the opportunity of abstracting data structures and functions into objects. Functions for special data structures can be implemented together in an object, and some of the functions and data structures can be hidden for parts of the implementation, if they are not relevant for this part. Furthermore an object-oriented programming language often has predefined types like integers and booleans, making it unnecessary to implement them, and object-oriented languages often come with an integrated developing environment (IDE), that gives IntelliSense. On the other hand an object-oriented language is more fitted for large-scale systems like an administration system, and creating data structures are often more cumbersome than in ML. However an object-oriented language is still a good option.

So far we have only mentioned first order Prolog. A language called λProlog takes up the idea of a higher order Prolog. Instead of only having first order unification, λProlog uses higher order unification and quantification over functions and predicates. Using this language would allow us to pass functions as arguments for other functions just as in ML. Therefore using λProlog instead of SWI-Prolog for the implementation might give some benefits.

Furthermore Isar (intelligible semi-automated reasoning) could be considered. Isar does not seek to be a programming language but is more a language for writing proofs. Isar can be viewed as a language for pretty printing. It tries to build bridge from a user of a theorem prover (see appendix C) to its internal structure by giving a declarative interface to the user. When Isar is used, instead of showing the proof of a theorem as internal structures of the theorem prover, Isar is a translation of these internal structures in a format more readable for humans. However *HOLPro* is far too simple to get any benefit of using Isar. A pretty printer is implemented that does the job just fine for now.
References: λProlog is from [12], Isar is from [15].

Chapter 7

Conclusion

In chapter 2 some of the underlying theory for the thesis has been presented. *HOLPro* is a higher order logic, in which it is possible to quantify over not only individual variables but also predicates and function variables. Furthermore *HOLPro* is intuitionistic which is opposite to classical logic. In intuitionistic logic a formula is only accepted as true or false if a direct proof of this exists. Proofs by contradiction are not accepted as proofs in intuitionistic logic. In intuitionistic logic the law of excluded middle is unprovable and therefore false.

In chapter 3 the logic *HOLPro* is presented. First the ten axioms that the logic is build on were shown. Then other deduction rules were derived from the axioms. The rules for symmetry, alpha conversion and beta reduction among others were derived. In the end of the chapter the logical connectives were defined. They were later used for deriving rules themselves. The most important of these rules were the proof that the truth connective is indeed always true, the creation of a conjunction from two valid formulas and the regular modus ponens rule (based on implication and not equality).

In chapter 4 and 5 the implementation of the proof assistant is explained briefly and tested. The overall design and structure of the implementation were shown. The implementation is split into files, each file implementing the necessary data

structures and functionality for its subject. For instance in the file theorem.pl a data structure for a theorem together with the axioms have been implemented. After this the implementation was tested by showing that it could perform the main functionality described in the design. The modus ponens rule was implemented as the final step and tested.

In chapter 6 *HOLPro* and the implementation of the proof assistant were discussed. It was shown that the law of excluded middle could be derived from the implemented *HOLPro* by accepting the axiom of choice. Furthermore Peano arithmetic could also be derived by accepting the axioms of infinity and induction and assuming that 0 is contained in the set of the natural numbers and that the natural numbers are closed under a successor function. In the chapter it was also explained how the number of axioms could be lowered from ten to eight without changing the logic, because two of the axioms can actually be derived from the other axioms. The data structure for lambda expressions was also discussed. If de Bruijn indexes had been used it would be possible to avoid alpha conversion, and we would not have to worry about binding free variables, when making a substitution. However it is less user-friendly than noting variables by string names. On the other hand, this could be partly solved by letting free variables keep their string names. In the end different programming languages were discussed. The choice of the programming language was decided as a part of the thesis to be Prolog. However ML or an object-oriented programming language might had been more suited for the implementation.

In the end it can be concluded that the purpose of the thesis was achieved. A proof assistant based on the intuitionistic higher order logic *HOLPro* has been implemented in Prolog. In *HOLPro* it was possible to derive the law of excluded middle and Peano arithmetic. It has therefore been shown that if the axiom of choice is accepted in an intuitionistic logic, it is not intuitionistic anymore.

Appendix A

Source code

Terms

```
%Types: booleans, individuals and functions

type(bool).
type(ind).
type(fun(X,Y)) :- type(X),type(Y).

%Functors to work with functions

create_fun(S,T,fun(S,T)).
dest_fun(fun(S,T),S,T).

domaintype(F,T) :- dest_fun(F,T,_).
rangetype(F,T) :- dest_fun(F,_,T).

rangetype_rec(Ty,Result) :-
        Ty = fun(_,Ran),
        rangetype_rec(Ran,Result).
rangetype_rec(Ty,Ty).
```

```
%Terms: constants, variables, applications and abstractions
term(const(S,X)) :- atom(S), type(X).
term(var(S,X)) :- atom(S), type(X).
term(app(T1,T2)) :- term(T1), term(T2).
term(abs(var(_,_),T2)) :- term(T2).

%Gives the type of a term
type_of(const(_,X),X).
type_of(var(_,X),X).
type_of(app(S,_),Ty) :-
        type_of(S,Z),
        rangetype(Z,Ty).
type_of(abs(S,T),Ty) :-
        type_of(S,TyS),
        type_of(T,TyT),
        create_fun(TyS,TyT,Ty).

%Creates a term
create_const(S,Ty,Z) :-
        Z=const(S,Ty),
        term(Z).
create_var(S,Ty,Z) :-
        Z=var(S,Ty),
        term(Z).
create_app(S,T,Z) :-
        Z=app(S,T),
        term(Z),
        type_of(S,TyS),
        type_of(T,TyT),
        domaintype(TyS,DomS),
        rangetype_rec(TyT,RanT),
        RanT=DomS.
create_abs(S,T,Z) :-
        Z=abs(S,T),
        term(Z).

%Destroys a term
dest_const(const(S,Ty),S,Ty).
dest_var(var(S,Ty),S,Ty).
dest_app(app(S,T),S,T).
dest_abs(abs(S,T),S,T).

%Helper functor for alpha convertibility.
```

```
%Succeeds if two variables are alpha convertible.
aconv_var([],V1,V2) :-
        V1 = V2.
aconv_var([(Bound1,Bound2)|_],V1,V2) :-
        Bound1 = V1,
        Bound2 = V2.
aconv_var([(Bound1,Bound2)|Rest],V1,V2) :-
        Bound1 \= V1,
        Bound2 \= V2,
        aconv_var(Rest,V1,V2).

%Helper functor for alpha convertibility.
%Goes recursivily through terms.
%The environment (bound variables) is saved in a list.
aconv_rec(Env,var(V1,Ty1),var(V2,Ty2)) :-
        aconv_var(Env,var(V1,Ty1),var(V2,Ty2)).
aconv_rec(_,const(C1,Ty1),const(C2,Ty2)) :-
        const(C1,Ty1) = const(C2,Ty2).
aconv_rec(Env,app(S1,T1),app(S2,T2)) :-
        aconv_rec(Env,S1,S2),
        aconv_rec(Env,T1,T2).
aconv_rec(Env,abs(var(V1,Ty1),Body1),abs(var(V2,Ty2),Body2)) :-
        Ty1 = Ty2,
        append([(var(V1,Ty1),var(V2,Ty2))],Env,Newenv),
        aconv_rec(Newenv,Body1,Body2).

%Checks for alpha convertibility
alphac(Tm1,Tm2) :-
        aconv_rec([],Tm1,Tm2).

aconv_in_list(_,[]) :- fail.
aconv_in_list(Tm,[A|_]) :- alphac(Tm,A).
aconv_in_list(Tm,[_|Rest]) :- aconv_in_list(Tm,Rest).

%Finds free variables in a term
frees(const(_,_),[]).
frees(var(S,T),[var(S,T)]).
frees(app(S,T),Z) :-
        frees(S,L1),
        frees(T,L2),
        union(L1,L2,Z).
frees(abs(S,T),Z) :-
        frees(T,L),
        subtract(L,[S],Z).
```

```
%Checks if a given variable is free in a term
is_free(Var,Tm) :-
        frees(Tm,F),
        (\+ member(Var,F)).

%Functors for renaming a variable
rename_var(var(Oldvar,T),var(Newvar,T)) :-
        retract(var_name(Oldvar,N1)),!,
        N2 is N1+1,
        assert(var_name(Oldvar,N2)),
        atom_concat(Oldvar,N2,Newvar).
rename_var(var(Oldvar,T),var(Newvar,T)) :-
        assert(var_name(Oldvar,1)),
        atom_concat(Oldvar,1,Newvar).

reset_var :- retractall(var_name(_,_)).

rename_check(Oldvar,Avoid,Newvar) :-
        member(Oldvar,Avoid),!,
        rename_var(Oldvar,Newvar).
rename_check(Oldvar,_,Oldvar).

%For getting a map from a variable to a term
get_value(_,[],_) :- !,fail.
get_value(Var,[(Var,Value)|_],Value).
get_value(Var,[_|Rest],Value) :- get_value(Var,Rest,Value).

%Succeeds if a variable occurs freely in a mapping
collision(_,[]) :- !,fail.
collision(S,[(_,Sub)|_]) :- frees(Sub,F),member(S,F).
collision(T,[_|Rest]) :- collision(T,Rest).

%Instantiation

%For variable
inst(var(Name,Ty),Mapping,Value,Mapping) :-
        get_value(var(Name,Ty),Mapping,Value).
inst(var(Name,Ty),Mapping,var(Name,Ty),Mapping).

%For application
inst(app(S,T),Mapping,app(S2,T2),NewM) :-
        inst(S,Mapping,S2,NewM1),
        inst(T,Mapping,T2,NewM2),
```

```
        union(NewM1,NewM2,NewM).

%For abstraction
inst(abs(S,T),Mapping,abs(S2,T2),NewM) :-
        collision(S,Mapping),
        rename_var(S,S2),
        inst(T,[(S,S2)|Mapping],T2,NewM).
inst(abs(S,T),Mapping,abs(S,T2),NewM) :-
        inst(T,[(S,S)|Mapping],T2,NewM).

%For constant
inst(T,_,T,_).

%For checking if types match in a mapping
type_check([]).
type_check([(S,T)]) :-
        type_of(S,Ty),
        type_of(T,Ty).
type_check([(S,T)|Rest]) :-
        type_of(S,Ty),
        type_of(T,Ty),
        type_check(Rest).

%The final instantiate method
instantiate(T,Mapping,Result,NewM) :-
        type_check(Mapping),
        inst(T,Mapping,Result,NewM).
instantiate(T,_,T,_) :-
        write('Invalid mapping: types do not match.'),
        nl,fail.

%Equality
create_eq(S,T,Z) :-
        type_of(S,TyS),
        type_of(T,TyT),
        create_const('=',fun(TyS,fun(TyT,bool)),Eq),
        create_app(Eq,S,Z1),
        create_app(Z1,T,Z).

dest_eq(app(app(const(=,_),S),T),S,T).
```

Basic deductive system

```
:- ensure_loaded('term.pl').

%Axioms with theorems

create_theorem(Assumptions,Term,theorem(Assumptions,Term)).

dest_theorem(theorem(Assumptions,Term),Assumptions,Term).

%Prints an error message and will fail
write_invalid(I) :-
        write('Theorems for '),
        write(I),write(' are invalid.'),
        nl,fail.

%Refl
refl(T,Result) :-
        create_eq(T,T,E),
        create_theorem([],E,Result).

%Trans
trans(theorem(A1,T1),theorem(A2,T2),Result) :-
        dest_eq(T1,L1,R1),
        dest_eq(T2,L2,R2),
        alphac(R1,L2),
        union(A1,A2,A),
        create_eq(L1,R2,E),
        create_theorem(A,E,Result).
trans(_,_,_) :- write_invalid('trans').

%Cong
cong(theorem(A1,T1),theorem(A2,T2),Result) :-
        dest_eq(T1,F,G),
        dest_eq(T2,X,Y),
        create_app(F,X,FX),
        create_app(G,Y,GY),
        create_eq(FX,GY,FXGY),
        union(A1,A2,A),
        create_theorem(A,FXGY,Result).
cong(_,_,_) :- write_invalid('cong').
```

```
%Abs
abs(X,theorem(A1,_),_) :-
        collision(X,A1),
        write('abs: there is a free variable.'),nl.
abs(X,theorem(A,E),Result) :-
        dest_eq(E,S,T),
        create_abs(X,S,XS),
        create_abs(X,T,XT),
        create_eq(XS,XT,XSXT),
        create_theorem(A,XSXT,Result).
abs(_,_,_) :- write_invalid('abs').

%Beta
beta(T,Result) :-
        dest_app(T,F,X1),
        dest_abs(F,X2,T2),
        X1=X2,
        create_eq(T,T2,E),
        create_theorem([],E,Result).
beta(_,_) :- write_invalid('beta').

%Eta
eta(X,T,_) :-
        frees(T,F),
        member(X,F),
        write('eta: the bound variable occurs freely'),
        nl,!,fail.
eta(X,T,Result) :-
        create_app(T,X,TX),
        create_abs(X,TX,XTX),
        create_eq(XTX,T,E),
        create_theorem([],E,Result).
eta(_,_,_) :- write_invalid('eta').

%Assume
assume(T,Result) :-
        type_of(T,bool),
        create_theorem([T],T,Result).
assume(_,_) :- write_invalid('assume').

%Eq_mp
eq_mp(theorem(A1,T1),theorem(A2,T2),Result) :-
        dest_eq(T1,Left,Right),
        alphac(Left,T2),
```

```
        union(A1,A2,A),
        create_theorem(A,Right,Result).
eq_mp(_,_,_) :- write_invalid('eq_mp').

%Subtract from list if alpha convertible.
subtract_alpha([],_,[]).
subtract_alpha([E|Rest],Term,Result) :-
        alphac(E,Term),
        Result = Rest.
subtract_alpha([E|Rest],Term,Result) :-
        subtract_alpha(Rest,Term,L),
        append([E],L,Result).

%Deduct_antisym
deduct_antisym(theorem(A1,T1),theorem(A2,T2),Result) :-
        create_eq(T1,T2,E),
        subtract_alpha(A2,T1,B2),
        subtract_alpha(A1,T2,B1),
        union(B1,B2,A),
        create_theorem(A,E,Result).

%Instantiate
inst_assumptions(_,[],[]).
inst_assumptions(Mapping,[T|Rest],Result) :-
        inst_assumptions(Mapping,Rest,Rest2),
        instantiate(T,Mapping,T2,_),
        append([T2],Rest2,Result).

% The final instantiation rule
inst_rule(Mapping,theorem(A,T),Result) :-
        instantiate(T,Mapping,T2,NewM),
        inst_assumptions(NewM,A,A2),
        create_theorem(A2,T2,Result).
```

Derived rules

```
:- ensure_loaded('theorem').
```

```
%Implementation of the derived rules.

%cong function: special cong with only one function that is
%given as a term.
%cong_function(+term/function,+theorem/parameters,-result)
cong_function(Tm,E,Result) :-
        refl(Tm,Refl_Tm),
        cong(Refl_Tm,E,Result).

%cong parameter: special cong with only one parameter that is
%given as a term
%cong_parameter(+theorem/functions,+term/parameter,-result)
cong_parameter(E,Tm,Result) :-
        refl(Tm,Refl_Tm),
        cong(E,Refl_Tm,Result).

%Beta-conversion
beta_conv(Tm,Result) :-
        dest_app(Tm,F,T),
        dest_abs(F,X,_),
        create_app(F,X,FX),
        beta(FX,FX2),
        inst_rule([(X,T)],FX2,Result).

%Symmetri
sym(theorem(A,T),Result) :-
        dest_app(T,Eq_left,_),
        dest_app(Eq_left,Eq,Left),
        refl(Eq,Refl_eq),
        cong(Refl_eq,theorem(A,T),Cong1),
        refl(Left,Refl_left),
        cong(Cong1,Refl_left,Cong2),
        eq_mp(Cong2,Refl_left,Result).

%Alpha for term: renames a bound variable
alpha_term(Var,Tm,Tm) :-
        dest_abs(Tm,Var0,_),
        Var = Var0.
alpha_term(Var,Tm,Result) :-
        dest_abs(Tm,Var0,Body),
        type_of(Var,Ty),
        type_of(Var0,Ty0),
        Ty=Ty0,
        create_theorem([],Body,Thm),
```

```
        inst_rule([(Var0,Var)],Thm,theorem(_,Renamed_Body)),
        create_abs(Var,Renamed_Body,Result).

%Alpha equal: Makes an equal with two theorems if they are equal
alpha_equal(Tm1,Tm2,Result) :-
        refl(Tm1,Refl1),
        refl(Tm2,Refl2),
        trans(Refl1,Refl2,Result).

%Alpha conversion: renames the bound variable in an anbstraction
%and sets it equal to the original term
alpha_conv(Var,Tm,Result) :-
        alpha_term(Var,Tm,Tm2),
        alpha_equal(Tm,Tm2,Result).

% Genereic alpha conversion: alpha conversion for expressions with
% quantifiers
gen_alpha_conv(Var,abs(S,T),Result) :-
        alpha_conv(Var,abs(S,T),Result).
gen_alpha_conv(Var,app(Binder,Tm),Result) :-
        alpha_conv(Var,Tm,Renamed),
        cong_function(Binder,Renamed,Result).

% Creates a new theorem with the union of the two input theorems'
% assumptions minus the consequent of the first
% theorem as the new assumptions and the consequent of the second
% theorem as the new consequent.
prove_ass(Th1,Th2,Result) :-
        dest_theorem(Th1,_,Tm1),
        dest_theorem(Th2,A2,_),
        aconv_in_list(Tm1,A2),
        deduct_antisym(Th1,Th2,Z),
        eq_mp(Z,Th1,Result).
prove_ass(_,Th2,Th2).

% Beta reduction: Reduces an application consisting of
% an abstraction and an arbitrary term.
% Uses normal order reduction.

% Case of an application and an abstraction
beta_red(T,Result) :-
        T = app(abs(_,_),_),
        beta_conv(T,BetaTh),
        dest_theorem(BetaTh,_,BetaTm),
```

```
        create_theorem([],BetaTm,Th1),
        eq_mp(Th1,theorem([],T),Th),
        dest_theorem(Th,_,Result).
% Case of an abstraction
beta_red(T,Result) :-
        T = abs(X,Tm),
        beta_red(Tm,BetaTm),
        create_abs(X,BetaTm,Result).
% Case of an application
beta_red(T,Result) :-
        T = app(Tm1,Tm2),
        beta_red(Tm1,BetaTm1),
        Tm1 \= BetaTm1,
        create_app(BetaTm1,Tm2,Result).
beta_red(T,Result) :-
        T = app(Tm1,Tm2),
        beta_red(Tm2,BetaTm2),
        create_app(Tm1,BetaTm2,Result).
% Other cases
beta_red(T,T).

% Wrapper for beta reduction
beta_reduction(Th,Result) :-
        dest_theorem(Th,A,Tm),
        beta_red(Tm,BetaTm),
        Tm \= BetaTm,
        create_theorem(A,BetaTm,NewTh),
        beta_reduction(NewTh,Result).
beta_reduction(Th,Result) :-
        dest_theorem(Th,A,Tm),
        beta_red(Tm,BetaTm),
        create_theorem(A,BetaTm,Result).
```

Definitions of logical connectives

```
:- ensure_loaded('derived_rules').

% Implementation of the logical constants and predicates
% associated with them.

%Creates a binary operator
create_binary(String,Tm1,Tm2,Result) :-
        create_const(String,fun(bool,fun(bool,bool)),C),
        create_app(C,Tm1,App),
        create_app(App,Tm2,Result).

%Creates a binder
create_binder(String,Abs,Result) :-
        dest_abs(Abs,_,_), %Abs must be a lambda expression
        type_of(Abs,Ty),
        rangetype_rec(Ty,bool),
        create_const(String,fun(bool,bool),Binder),
        create_app(Binder,Abs,Result).

%True

truth_def(Result) :-
        create_const('tt',bool,T),
        create_var('p',bool,P),
        create_abs(P,P,Abs),
        create_eq(Abs,Abs,Eq),
        create_eq(T,Eq,Tm),
        create_theorem([],Tm,Result).

truth(Result) :-
        create_var('p',bool,P),
        create_abs(P,P,Abs),
        refl(Abs,R),
        truth_def(Tdef),
        sym(Tdef,S),
        eq_mp(S,R,Result).

truth_elim(Th,Result) :-
        truth(T),
        sym(Th,Sym),
```

```
        eq_mp(Sym,T,Result).

truth_intro(Th,Result) :-
        create_var('t',bool,T),
        assume(T,T2),
        truth(Truth),
        deduct_antisym(T2,Truth,T3),
        dest_theorem(T3,_,Tm3),
        assume(Tm3,T4),
        truth_elim(T4,T5),
        deduct_antisym(T5,T3,T6),
        dest_theorem(Th,_,Tm),
        inst_rule([(T,Tm)],T6,Inst),
        eq_mp(Inst,Th,Result).


%Conjunction

create_conj(Tm1,Tm2,Result) :-
        create_binary('/\\',Tm1,Tm2,Result).

conj_def(Result) :-
        create_const('/\\',fun(bool,fun(bool,bool)),C),
        create_var('f',fun(bool,fun(bool,bool)),F),
        create_var('p',bool,P),
        create_var('q',bool,Q),
        create_app(F,P,FP),
        create_app(FP,Q,FPQ),
        create_abs(F,FPQ,Left_abs),
        truth(TT),
        dest_theorem(TT,_,T),
        create_app(F,T,FT),
        create_app(FT,T,FTT),
        create_abs(F,FTT,Right_abs),
        create_eq(Left_abs,Right_abs,E),
        create_abs(Q,E,QE),
        create_abs(P,QE,PQE),
        create_eq(C,PQE,Conj),
        create_theorem([],Conj,Result).

%conj(+lhs,+rhs,-conjunction)
conj(Tm1,Tm2) :-
        (term(Tm1);term(Tm2)),
        write('conj takes theorems as arguments.').
```

```
conj(Th1,Th2,Result) :-
        create_var('p',bool,P),
        create_var('q',bool,Q),
        create_var('f',fun(bool,fun(bool,bool)),F),
        assume(P,AxiomP),
        truth_intro(AxiomP,PTruth),
        cong_function(F,PTruth,CongFP),
        assume(Q,AxiomQ),
        truth_intro(AxiomQ,QTruth),
        cong(CongFP,QTruth,CongFPQ),
        abs(F,CongFPQ,AbsFPQ),

        conj_def(Conjunction),
        cong_parameter(Conjunction,P,CongConP),
        cong_parameter(CongConP,Q,CongConPQ),
        beta_reduction(CongConPQ,ConjQ),
        beta_reduction(ConjQ,Conj),
        sym(Conj,SymConj),
        eq_mp(SymConj,AbsFPQ,ConjPQ),
        dest_theorem(Th1,_,Tm1),
        dest_theorem(Th2,_,Tm2),
        inst_rule([(P,Tm1),(Q,Tm2)],ConjPQ,InstConj),
        prove_ass(Th1,InstConj,ProveTh1),
        prove_ass(Th2,ProveTh1,Result).

dest_conj(app(app(const('/\\',fun(bool,fun(bool,bool))),P),Q),P,Q).

% For extracting one literal of a conjunction.
% Var should be var('left',bool) or var('right',bool)
% for any effect, otherwise the result will be meaningless.
conj_extract(Th,Var,Result) :-
        dest_theorem(Th,_,Tm),
        dest_conj(Tm,L,R),
        create_var('p',bool,P),
        create_theorem([],P,ThP),
        create_var('q',bool,Q),
        create_theorem([],Q,ThQ),
        conj_def(Conj),

        cong_parameter(Conj,P,ConjP),
        beta_reduction(ConjP,Beta),
        cong_parameter(Beta,Q,BetaQ),
        beta_reduction(BetaQ,Beta2),
        conj(ThP,ThQ,PandQ),
```

```
        dest_theorem(PandQ,_,TmPQ),
        assume(TmPQ,ThPQ),
        eq_mp(Beta2,ThPQ,EpMp),

        create_var('left',bool,Left),
        create_var('right',bool,Right),
        create_abs(Right,Var,RV),
        create_abs(Left,RV,LRV),

        cong_parameter(EpMp,LRV,CongPara),
        beta_reduction(CongPara,Beta3),
        truth_elim(Beta3,TElim),
        inst_rule([(P,L),(Q,R)],TElim,Inst),
        prove_ass(Th,Inst,Result).

% Extracts the lhs of a conjunction
conj_left(Th,Result) :-
        create_var('left',bool,Left),
        conj_extract(Th,Left,Result).

% Extracts the rhs of a conjunction
conj_right(Th,Result) :-
        create_var('right',bool,Right),
        conj_extract(Th,Right,Result).

% Extracts both the lhs and rhs of a conjunction
conj_pair(Th,Lhs,Rhs) :-
        conj_left(Th,Lhs),
        conj_right(Th,Rhs).

%Implication

create_impl(Tm_left,Tm_right,Result) :-
        create_binary('==>',Tm_left,Tm_right,Result).

dest_impl(app(app(const('==>',
                        fun(bool,fun(bool,bool))),P),Q),P,Q).

impl_def(Result) :-
        create_const('==>',fun(bool,fun(bool,bool)),C),
        create_var('p',bool,P),
        create_var('q',bool,Q),
        create_conj(P,Q,And),
        create_eq(And,P,E),
```

```
        create_abs(Q,E,Abs1),
        create_abs(P,Abs1,Abs2),
        create_eq(C,Abs2,E2),
        create_theorem([],E2,Result).

%Modus ponens
mp(Ith,Th,Result) :-
        create_var('p',bool,P),
        impl_def(Imp),
        cong_parameter(Imp,P,Cong1),
        create_var('q',bool,Q),
        cong_parameter(Cong1,Q,Cong2),
        beta_reduction(Cong2,Beta1),
        create_impl(P,Q,Impl2),
        assume(Impl2,Axiom1),
        eq_mp(Beta1,Axiom1,EqMp1),
        sym(EqMp1,Sym1),
        assume(P,Axiom2),
        eq_mp(Sym1,Axiom2,EqMp2),
        conj_right(EqMp2,Conj1),

        dest_theorem(Ith,_,Impl3),
        dest_impl(Impl3,Ant,Con),
        dest_theorem(Th,_,Tm),
        alphac(Ant,Tm),
        inst_rule([(P,Ant),(Q,Con)],Conj1,Inst1),
        prove_ass(Ith,Inst1,ProveHyp1),
        prove_ass(Th,ProveHyp1,Result).

%Universal quantifier

create_forall(P,Result) :-
        create_binder('!',P,Result).

forall_def(Result) :-
        create_var('x',ind,X),
        truth(Th),
        dest_theorem(Th,_,T),
        create_abs(X,T,XT),
        create_var('p',fun(ind,bool),P),
        create_eq(P,XT,E),
        create_abs(P,E,Abs),
        create_const('!',fun(bool,bool),C),
        create_eq(C,Abs,E2),
```

```
        create_theorem([],E2,Result).

%Existential quantifier

create_exists(P,Result) :-
        create_binder('?',P,Result).

exists_def(Result) :-
        create_const('?',fun(bool,bool),C),
        create_var('p',fun(ind,bool),P),
        create_var('x',ind,X),
        create_var('q',bool,Q),
        create_app(P,X,PX),
        create_impl(PX,Q,Impl),
        create_abs(X,Impl,A1),
        create_forall(A1,Forall),
        create_impl(Forall,Q,Impl2),
        create_abs(Q,Impl2,A2),
        create_forall(A2,Forall2),
        create_abs(P,Forall2,Abs),
        create_eq(C,Abs,E),
        create_theorem([],E,Result).

%Disjunction

create_disj(Tm1,Tm2,Result) :-
        create_binary('\\/',Tm1,Tm2,Result).

disj_def(Result) :-
        create_const('\\/',fun(bool,fun(bool,bool)),C),
        create_var('p',bool,P),
        create_var('q',bool,Q),
        create_var('r',bool,R),
        create_impl(P,R,PR),
        create_impl(Q,R,QR),
        create_impl(QR,R,QRR),
        create_impl(PR,QRR,PRQRR),
        create_abs(R,PRQRR,A),
        create_forall(A,Forall),
        create_abs(Q,Forall,Abs1),
        create_abs(P,Abs1,Abs2),
        create_eq(C,Abs2,E),
        create_theorem([],E,Result).
```

```
%Falsity

false_def(Result) :-
        create_const('ff',bool,F),
        create_var('p',bool,P),
        create_abs(P,P,PP),
        create_forall(PP,Forall),
        create_eq(F,Forall,E),
        create_theorem([],E,Result).

%Negation

neg_def(Result) :-
        create_const('~',fun(bool,bool),C),
        create_const('ff',bool,F),
        create_var('p',bool,P),
        create_impl(P,F,Impl),
        create_abs(P,Impl,Abs),
        create_eq(C,Abs,E),
        create_theorem([],E,Result).

create_neg(P,Result) :-
        type_of(P,bool),
        create_const('~',fun(bool,bool),Neg),
        create_app(Neg,P,Result).

%Unique existential quantifier

uexists_def(Result) :-
        create_const('?!',fun(bool,bool),C),
        create_var('x',ind,X),
        create_var('y',ind,Y),
        create_var('p',fun(ind,bool),P),
        create_eq(X,Y,XY),
        create_app(P,X,PX),
        create_app(P,Y,PY),
        create_conj(PX,PY,PXPY),
        create_impl(PXPY,XY,Impl),
        create_abs(Y,Impl,Abs_y),
        create_forall(Abs_y,Forall),
        create_abs(X,Forall,Abs_x),
        create_forall(Abs_x,Part2),
        create_abs(X,PX,Abs_x2),
        create_exists(Abs_x2,Part1),
```

```
        create_conj(Part1,Part2,Conj),
        create_abs(P,Conj,Abs1),
        create_eq(C,Abs1,E),
        create_theorem([],E,Result).

create_uexists(P,Result) :-
        type_of(P,Ty),
        Ty = fun(_,bool),
        create_const('?!',fun(Ty,bool),Uexists),
        create_app(Uexists,P,Result).
```

Pretty printer

```
:- ensure_loaded('classic').

%Pretty printer

%Types
pretty_print(bool) :- write('bool').
pretty_print(ind) :- write('ind').
pretty_print(fun(X,Y)) :-
        write(X),
        write('_'),
        pretty_print(Y).

%Special constants
%For true
pretty_print(const('tt',bool)) :-
        write('tt').
%For false
pretty_print(const('ff',bool)) :-
        write('ff').
%For binary operators
pretty_print(app(app(const('=',_),S),T)) :-
        write('('),pretty_print(S),
        write('='),pretty_print(T),
        write(')').
pretty_print(app(app(const('/\\',_),S),T)) :-
```

```
        write('('),pretty_print(S),
        write('/\\'),
        pretty_print(T),
        write(')').
pretty_print(app(app(const('\\/',_),S),T)) :-
        write('('),pretty_print(S),
        write('\\/'),
        pretty_print(T),
        write(')').
pretty_print(app(app(const('==>',_),S),T)) :-
        write('('),
        pretty_print(S),
        write('==>'),
        pretty_print(T),
        write(')').
%For select operator
pretty_print(app(const('@',_),Tm)) :-
        write('(('),
        write('@'),
        write(') '),
        pretty_print(Tm),
        write(')').
%For binders
pretty_print(app(const('!',_),abs(S,T))) :-
        write('('),
        write('!'),
        pretty_print(S),
        write('.'),
        pretty_print(T),
        write(')').
pretty_print(app(const('?',_),abs(S,T))) :-
        write('('),
        write('?'),
        pretty_print(S),
        write('.'),
        pretty_print(T),
        write(')').
pretty_print(app(const('?!',_),abs(S,T))) :-
        write('('),
        write('?!'),
        pretty_print(S),
        write('.'),
        pretty_print(T),
        write(')').
```

```
%For negation
pretty_print(app(const('~',_),P)) :-
        write('~'),
        pretty_print(P).

%Basic terms
pretty_print(const(S,X)) :-
        write('const_'),
        write(S),
        write('_'),
        pretty_print(X).
pretty_print(var(S,X)) :-
        write(S),
        write('_'),
        pretty_print(X).
pretty_print(app(T1,T2)) :-
        pretty_print(T1),
        write(' '),
        pretty_print(T2).
pretty_print(abs(S,X)) :-
        write('(\\'),
        pretty_print(S),
        write('.'),
        pretty_print(X),
        write(')').

%Instantiation list
pretty_print([(X,Y)]) :-
        write('('),
        pretty_print(X),
        write('-->'),
        pretty_print(Y),
        write(')').
pretty_print([(X,Y)|Rest]) :-
        write('('),
        pretty_print(X),
        write('-->'),
        pretty_print(Y),
        write('),'),
        pretty_print(Rest).

%Assumption list
pretty_print([]).
pretty_print([Tm]) :-
```

```
        pretty_print(Tm).
pretty_print([Tm|Rest]) :-
        pretty_print(Tm),
        write(','),
        pretty_print(Rest).

%For printing a theorem
pretty_print(theorem(A,Tm)) :-
        write('['),
        pretty_print(A),
        write(']-->'),
        pretty_print(Tm).
```

Appendix B

Tests

```
:- ensure_loaded('pp').

%Tests for terms

test_funvar(Name,S,T,X) :-
        create_fun(S,T,F),
        create_var(Name,F,X).

test_create_term(X) :-
        create_var('x',ind,S),
        create_fun(ind,bool,F),
        create_const('c',F,T),
        create_var('y',ind,Y),
        create_app(T,Y,U),
        create_fun(bool,bool,F2),
        create_var('z',F2,Z),
        create_app(Z,U,ZU),
        create_abs(S,ZU,X).

test_alphac_succes :-
        create_var('x',ind,X),
```

```
        create_var('y',ind,Y),
        create_abs(X,X,XX),
        create_abs(Y,Y,YY),
        write('Input: '),
        pretty_print(XX),nl,
        write('       '),
        pretty_print(YY),nl,
        alphac(XX,YY).

test_alphac_fail :-
        create_var('x',ind,X),
        create_var('y',ind,Y),
        create_var('z',ind,Z),
        create_abs(X,X,XX),
        create_abs(Y,Z,YZ),
        write('Input: '),
        pretty_print(XX),nl,
        write('       '),
        pretty_print(YZ),nl,
        alphac(XX,YZ).

test_mapping([(Z,T1),(Y,T2)]) :-
        create_var('z',fun(bool,bool),Z),
        create_var('a',fun(bool,bool),T1),
        create_var('y',ind,Y),
        create_var('x',ind,T2).

test_frees1 :-
        create_var('x',ind,X),
        create_var('y',ind,Y),
        create_abs(X,Y,Abs),
        write('Input: '),
        pretty_print(Abs),nl,
        frees(Abs,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_frees2 :-
        create_var('x',ind,X),
        create_abs(X,X,XX),
        write('Input: '),
        pretty_print(XX),nl,
        frees(XX,Result),
        write('Result: '),
```

```
        pretty_print(Result),nl.

test_frees3 :-
        create_var('x',ind,X),
        create_abs(X,X,XX),
        create_var('y',ind,Y),
        create_app(XX,Y,XXY),
        write('Input: '),
        pretty_print(XXY),nl,
        frees(XXY,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_rename_var :-
        create_var('x',ind,X),
        write('Input: '),
        pretty_print(X),nl,
        rename_var(X,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_inst1 :-
        test_create_term(Oldterm),
        test_mapping(M),
        write('Input: '),
        write('Mapping: '),
        pretty_print(M),nl,
        write('       Term: '),
        pretty_print(Oldterm),nl,
        instantiate(Oldterm,M,Result,_),
        write('Result: '),
        pretty_print(Result),nl.

test_inst2 :-
        create_var('x',ind,X),
        create_abs(X,X,XX),
        create_var('y',ind,Y),
        M = [(X,Y)],
        write('Input: '),
        write('Mapping: '),
        pretty_print(M),nl,
        write('       Term: '),
        pretty_print(XX),nl,
        instantiate(XX,M,Result,_),
```

```
        write('Result: '),
        pretty_print(Result),nl.

test_create_eq(X) :- create_eq(var('s',ind),var('t',ind),X).

test_dest_eq(X1,X2) :- test_create_eq(Y),dest_eq(Y,X1,X2).

%Tests for axioms/basic rules

test_theorem(X) :-
        test_create_eq(T),
        create_var('a',ind,Ass),
        create_theorem([Ass],T,X).

test_refl :-
        create_var('x',ind,T),
        write('Input: '),
        pretty_print(T),nl,
        refl(T,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_trans :-
        test_theorem(T1),
        create_eq(var('t',ind),var('u',ind),E),
        create_var('a',ind,Ass),
        create_theorem([Ass],E,T2),
        write('Input: '),
        pretty_print(T2),nl,
        trans(T1,T2,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_cong :-
        test_funvar('s',ind,ind,S),
        test_funvar('t',ind,ind,T),
        create_eq(S,T,E1),
        create_var('a',ind,Ass1),
        create_theorem([Ass1],E1,T1),
        create_var('u',ind,U),
        create_var('v',ind,V),
        create_eq(U,V,E2),
        create_var('b',ind,Ass2),
        create_theorem([Ass2],E2,T2),
```

```
        write('Input: '),
        pretty_print(T1),nl,
        write('       '),
        pretty_print(T2),nl,
        cong(T1,T2,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_abs :-
        test_theorem(T),
        create_var('x',ind,V),
        write('Input: '),
        write('Bound var: '),
        pretty_print(V),nl,
        write('       '),
        write('Theorem: '),
        pretty_print(T),nl,
        abs(V,T,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_beta :-
        create_var('x',ind,S),
        create_var('t',ind,T),
        create_abs(S,T,ST),
        create_app(ST,S,STS),
        write('Input: '),
        pretty_print(STS),nl,
        beta(STS,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_eta :-
        create_var('x',ind,S),
        create_var('t',fun(ind,ind),T),
        write('Input: '),
        write('Bound var: '),
        pretty_print(S),nl,
        write('       '),
        write('Term: '),
        pretty_print(T),nl,
        eta(S,T,Result),
        write('Result: '),
        pretty_print(Result),nl.
```

```
test_assume :-
        create_var('p',bool,P),
        write('Input: '),
        pretty_print(P),nl,
        assume(P,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_eq_mp :-
        test_create_eq(E),
        create_var('a',ind,Ass1),
        create_theorem([Ass1],E,T1),
        create_var('s',ind,S),
        create_var('b',ind,Ass2),
        create_theorem([Ass2],S,T2),
        write('Input: '),
        pretty_print(T1),nl,
        write('       '),
        pretty_print(T2),nl,
        eq_mp(T1,T2,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_subtract_alpha :-
        create_var('x',ind,X),
        create_var('y',ind,Y),
        create_abs(X,X,XX),
        create_abs(Y,Y,YY),
        subtract_alpha([X,YY],XX,Result),
        pretty_print(Result).

test_deduct_antisym :-
        create_var('s',ind,S),
        create_var('t',ind,T),
        create_theorem([T],S,S2),
        create_theorem([S],T,T2),
        write('Input: '),
        pretty_print(S2),nl,
        write('       '),
        pretty_print(T2),nl,
        deduct_antisym(S2,T2,Result),
        write('Result: '),
        pretty_print(Result),nl.
```

```
test_inst_rule1 :-
        create_var('x',ind,X),
        create_var('y',ind,Y),
        create_var('z',ind,Z),
        create_var('u',ind,U),
        create_abs(X,Y,Abs),
        create_app(Abs,Z,AbsZ),
        create_theorem([X],AbsZ,T),
        M = [(Y,X),(Z,U)],
        write('Input: '),
        write('Mapping: '),
        pretty_print(M),nl,
        write('       '),
        write('Theorem: '),
        pretty_print(T),nl,
        inst_rule(M,T,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_inst_rule2 :-
        create_var('x',ind,X),
        create_var('y',ind,Y),
        create_var('z',ind,Z),
        create_var('u',ind,U),
        create_abs(X,X,Abs),
        create_app(Abs,Z,AbsZ),
        create_theorem([X],AbsZ,T),
        M = [(X,Y),(Z,U)],
        write('Input: '),
        write('Mapping: '),
        pretty_print(M),nl,
        write('       '),
        write('Theorem: '),
        pretty_print(T),nl,
        inst_rule(M,T,Result),
        write('Result: '),
        pretty_print(Result),nl.

%Test derived rules
test_beta_conv(Result) :-
        create_var('x',ind,X),
        create_var('s',ind,S),
        create_var('t',ind,T),
```

```
        create_abs(X,S,XS),
        create_app(XS,T,XST),
        beta_conv(XST,Result).

test_sym :-
        create_var('l',ind,S),
        create_var('r',ind,T),
        create_eq(S,T,ST),
        create_var('a',ind,Ass),
        create_theorem([Ass],ST,Th),
        write('Input: '),
        pretty_print(Th),nl,
        sym(Th,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_cong_function :-
        create_var('f',fun(ind,ind),F),
        create_var('u',ind,U),
        create_var('v',ind,V),
        create_eq(U,V,E),
        create_var('a',ind,Ass),
        create_theorem([Ass],E,Thm),
        write('Input: '),
        write('Function: '),
        pretty_print(F),nl,
        write('       '),
        write('Equality: '),
        pretty_print(Thm),nl,
        cong_function(F,Thm,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_cong_parameter :-
        create_var('f',fun(ind,ind),F),
        create_var('g',fun(ind,ind),G),
        create_var('u',ind,U),
        create_eq(F,G,E),
        create_var('a',ind,Ass),
        create_theorem([Ass],E,Thm),
        write('Input: '),
        write('Equality: '),
        pretty_print(Thm),nl,
        write('       '),
```

```
        write('Argument: '),
        pretty_print(U),nl,
        cong_parameter(Thm,U,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_alpha_term(Result) :-
        create_var('y',ind,Y),
        create_var('x',ind,X),
        create_abs(X,X,XX),
        alpha_term(Y,XX,Result).

test_alpha_equal(Result) :-
        create_var('s',ind,S),
        alpha_equal(S,S,Result).

test_alpha_conv(Result) :-
        create_var('y',bool,Y),
        create_var('x',bool,X),
        create_abs(X,X,XX),
        alpha_conv(Y,XX,Result).

test_gen_alpha_conv1 :-
        create_var('y',bool,Y),
        create_var('x',bool,X),
        create_abs(X,X,XX),
        write('Input: '),
        write('New name for the bound variable: '),
        pretty_print(Y),nl,
        write('       '),
        write('Term: '),
        pretty_print(XX),nl,
        gen_alpha_conv(Y,XX,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_gen_alpha_conv2 :-
        create_var('y',bool,Y),
        create_var('x',bool,X),
        create_abs(X,X,XX),
        create_forall(XX,Forall),
        write('Input: '),
        write('New name for the bound variable: '),
        pretty_print(Y),nl,
```

```
        write('       '),
        write('Term: '),
        pretty_print(Forall),nl,
        gen_alpha_conv(Y,Forall,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_prove_ass1 :-
        create_var('a',ind,A),
        create_var('t1',ind,T1),
        create_var('t2',ind,T2),
        create_theorem([A],T1,Th1),
        create_theorem([T1],T2,Th2),
        write('Input: '),
        pretty_print(Th1),nl,
        write('       '),
        pretty_print(Th2),nl,
        prove_ass(Th1,Th2,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_prove_ass2 :-
        create_var('a',ind,A),
        create_var('b',ind,B),
        create_var('t1',ind,T1),
        create_var('t2',ind,T2),
        create_theorem([A],T1,Th1),
        create_theorem([B],T2,Th2),
        write('Input: '),
        pretty_print(Th1),nl,
        write('       '),
        pretty_print(Th2),nl,
        prove_ass(Th1,Th2,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_beta_red1 :-
        create_var('x',ind,X),
        create_var('y',ind,Y),
        create_abs(X,X,XX),
        create_app(XX,Y,XXY),
        create_var('a',ind,Ass),
        create_theorem([Ass],XXY,Th),
        write('Input: '),
```

```
        pretty_print(Th),nl,
        beta_reduction(Th,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_beta_red2 :-
        create_var('p',bool,P),
        create_var('q',bool,Q),
        create_var('s',bool,S),
        create_var('t',bool,T),
        create_var('f',fun(bool,fun(bool,bool)),F),
        create_app(F,P,FP),
        create_app(FP,Q,FPQ),
        create_abs(F,FPQ,FFPQ),
        create_abs(Q,FFPQ,QFFPQ),
        create_abs(P,QFFPQ,PQFFPQ),
        create_app(PQFFPQ,S,PQFFPQS),
        create_app(PQFFPQS,T,PQFFPQPST),
        create_var('a',ind,Ass),
        create_theorem([Ass],PQFFPQPST,Th),
        write('Input: '),
        pretty_print(Th),nl,
        beta_reduction(Th,Result),
        write('Result: '),
        pretty_print(Result),nl.

%Test logical constants

test_create_binary(Result) :-
        create_var('p',bool,P),
        create_var('q',bool,Q),
        create_binary('/\\',P,Q,Result),
        pretty_print(Result).

%Truth
test_truth_def(Result) :-
        truth_def(Result),
        pretty_print(Result).

test_truth :-
        truth(Result),
        write('Result: '),
        pretty_print(Result),nl.
```

```
test_truth_elim :-
        truth(T),
        dest_theorem(T,_,Tm),
        create_var('p',bool,P),
        create_eq(P,Tm,Eq),
        create_var('a',ind,Ass),
        create_theorem([Ass],Eq,Th),
        write('Input: '),
        pretty_print(Th),nl,
        truth_elim(Th,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_truth_intro :-
        create_var('p',bool,P),
        create_var('a',ind,Ass),
        create_theorem([Ass],P,Th),
        write('Input: '),
        pretty_print(Th),nl,
        truth_intro(Th,Result),
        write('Result: '),
        pretty_print(Result),nl.

%Conjunction

test_create_conj :-
        create_var('p',bool,P),
        create_var('q',bool,Q),
        write('Input: '),
        pretty_print(P),nl,
        write('       '),
        pretty_print(Q),nl,
        create_conj(P,Q,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_conj_def(Result) :-
        conj_def(Result),
        pretty_print(Result).

test_conj :-
        create_var('s',bool,S),
        create_var('a',ind,Ass1),
        create_theorem([Ass1],S,ThS),
```

```
        create_var('t',bool,T),
        create_var('b',ind,Ass2),
        create_theorem([Ass2],T,ThT),
        write('Input: '),
        pretty_print(ThS),nl,
        write('       '),
        pretty_print(ThT),nl,
        conj(ThS,ThT,Result),
        write('Result: '),
        pretty_print(Result),nl.

testhelper_conj(Result) :-
        create_var('l',bool,L),
        create_theorem([],L,Th1),
        create_var('r',bool,R),
        create_var('assumption',bool,A),
        create_theorem([A],R,Th2),
        conj(Th1,Th2,Result).

test_conj_left :-
        testhelper_conj(Conj),
        write('Input: '),
        pretty_print(Conj),nl,
        conj_left(Conj,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_conj_right :-
        testhelper_conj(Conj),
        write('Input: '),
        pretty_print(Conj),nl,
        conj_right(Conj,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_conj_pair(L,R) :-
        testhelper_conj(Conj),
        conj_pair(Conj,L,R),
        pretty_print(L),
        pretty_print(R).


%Implication
```

```
test_create_impl :-
        create_var('p',bool,P),
        create_var('q',bool,Q),
        write('Input: '),
        pretty_print(P),nl,
        write('       '),
        pretty_print(Q),nl,
        create_impl(P,Q,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_impl_def(Result) :-
        impl_def(Result),
        pretty_print(Result).

test_mp :-
        create_var('p',bool,P),
        create_var('q',bool,Q),
        create_impl(P,Q,Impl),
        create_var('a',ind,Ass1),
        create_var('b',ind,Ass2),
        create_theorem([Ass1],Impl,I),
        create_theorem([Ass2],P,Th),
        write('Input: '),
        pretty_print(I),nl,
        write('       '),
        pretty_print(Th),nl,
        mp(I,Th,Result),
        write('Result: '),
        pretty_print(Result),nl.

%Universal quantifier

test_create_forall :-
        create_var('p',bool,P),
        create_abs(P,P,Abs),
        write('Input: '),
        pretty_print(Abs),nl,
        create_forall(Abs,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_forall_def(Result) :-
        forall_def(Result),
```

```
        pretty_print(Result).

%Existential quantifier

test_create_exists :-
        create_var('p',bool,P),
        create_abs(P,P,Abs),
        write('Input: '),
        pretty_print(Abs),nl,
        create_exists(Abs,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_exists_def(Result) :-
        exists_def(Result),
        pretty_print(Result).

%Disjunction

test_create_disj :-
        create_var('p',bool,P),
        create_var('q',bool,Q),
        write('Input: '),
        pretty_print(P),nl,
        write('       '),
        pretty_print(Q),nl,
        create_disj(P,Q,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_disj_def(Result) :-
        disj_def(Result),
        pretty_print(Result).

%False

test_false_def :-
        false_def(Result),
        write('Result: '),
        pretty_print(Result),nl.

%Negation

test_create_neg(Result) :-
```

```
        create_var('p',bool,P),
        write('Input: '),
        pretty_print(P),nl,
        create_neg(P,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_neg_def(Result) :-
        neg_def(Result),
        pretty_print(Result).


%Unique existential quantifier

test_create_uexists(Result) :-
        create_var('p',bool,P),
        create_abs(P,P,Abs),
        write('Input: '),
        pretty_print(Abs),nl,
        create_uexists(Abs,Result),
        write('Result: '),
        pretty_print(Result),nl.

test_uexists_def(Result) :-
        uexists_def(Result),
        pretty_print(Result).

%Overall tests

test_term :-
        write('Testing predicates in term.pl'),
        nl,nl,
        write('Testing alpha convertibility'),nl,
        test_alphac_succes,
        write('Result: succes'),nl,
        \+ test_alphac_fail,
        write('Result: fail'),nl,
        nl,
        write('Testing for free variables'),nl,
        test_frees1,
        test_frees2,
        test_frees3,
        nl,
        write('Testing renaming of variables'),nl,
```

```
        test_rename_var,
        nl,
        write('Testing instantiation of terms'),nl,
        test_inst1,
        test_inst2,
        nl.

test_theorem :-
        write('Testing predicates in theorem.pl'),
        nl,nl,
        write('Testing refl'),nl,
        test_refl,
        nl,
        write('Testing trans'),nl,
        test_trans,
        nl,
        write('Testing cong'),nl,
        test_cong,
        nl,
        write('Testing abs'),nl,
        test_abs,
        nl,
        write('Testing beta'),nl,
        test_beta,
        nl,
        write('Testing eta'),nl,
        test_eta,
        nl,
        write('Testing assume'),nl,
        test_assume,
        nl,
        write('Testing eq_mp'),nl,
        test_eq_mp,
        nl,
        write('Testing deduct_antisym'),nl,
        test_deduct_antisym,
        nl,
        write('Testing instantiation'),nl,
        test_inst_rule1,
        test_inst_rule2,
        nl.

test_derived_rules :-
        write('Testing predicates in derived_rules.pl'),
```

```
        nl,nl,
        write('Testing cong_function'),nl,
        test_cong_function,
        nl,
        write('Testing cong_parameter'),nl,
        test_cong_parameter,
        nl,
        write('Testing sym'),nl,
        test_sym,
        nl,
        write('Testing alpha conversion'),nl,
        test_gen_alpha_conv1,
        test_gen_alpha_conv2,
        nl,
        write('Testing prove assumption'),nl,
        test_prove_ass1,
        test_prove_ass2,
        nl,
        write('Testing beta reduction'),nl,
        test_beta_red1,
        test_beta_red2,
        nl.

test_logical_connectives :-
        write('Testing predicates in logical_connectives.pl'),
        nl,nl,
        write('Testing prove of truth'),nl,
        test_truth,
        nl,
        write('Testing elimination of truth'),nl,
        test_truth_elim,
        nl,
        write('Testing introduction of truth'),nl,
        test_truth_intro,
        nl,
        write('Testing conj'),nl,
        test_conj,
        nl,
        write('Testing conj_left'),nl,
        test_conj_left,
        nl,
        write('Testing conj_right'),nl,
        test_conj_right,
        nl,
```

```
        write('Testing modus ponens'),nl,
        test_mp,
        nl.

test_all :-
        test_term,
        test_theorem,
        test_derived_rules,
        test_logical_connectives.
```

Appendix C

HOL Light

The proof assistant of this thesis is built on the logic of the theorem prover HOL Light. HOL Light also sets up an intuitionistic higher order logic that turns out to be classical when accepting the axioms of choice and extentionality. The HOL Light project has therefore served as a guideline for this thesis. The purpose of a theorem prover is to prove a statement from the logic that the theorem prover is based on. Which statements depend on the application of the theorem prover. The applications are mostly in the areas of

- Verifying mathematical theorems.
- Verifying software and hardware.

In particular HOL Light is used for verifying floating point algorithms for hardware at Intel. Among these floating-point algorithms are algorithms for division and square root. Furthermore HOL Light has been used for proving error bounds in transcendental functions. Transcendental functions are functions which cannot be expressed by a finite sequence of the usual algebraic functions addition, multiplication and root operations (for instance square root). Therefore an approximation of these functions must be used. Some examples of transcendental functions are logarithm, sine and cosine.
Besides being used at Intel, HOL Light is also used for formalising mathematical

theorems. The theorems can be simple ones like the irrationality of the square root of two and the Pythagorean theorem. However more complex theorems can also be verified by HOL Light. At the moment the Flyspeck project is trying to verify the proof of Kepler conjecture using HOL Light.
References: [6],[7].

Bibliography

[1] Jamie Andrews. A proof assistant for a weakly-typed higher order logic. *Newsletter of the Association for Logic Programming (ALP)*, 17(2), May 2004.

[2] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, 2nd edition, 2002.

[3] John L. Bell. The axiom of choice. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2009 edition, 2009.

[4] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

[5] John Harrison. HOL done right. Extracted from http://www.cl.cam.ac.uk/ jrh13/papers/holright.html, August 1995.

[6] John Harrison. Floating-point verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 529–532. Springer-Verlag, 2005.

[7] John Harrison. The HOL Light theorem prover, June 2010. http://www.cl.cam.ac.uk/ jrh13/hol-light/index.html.

[8] John Harrison. Chapter 9: Further glimpses. Notes received by email, February 2011.

[9] John Harrison. Source code for HOL Light, June 2011. http://code.google.com/p/hol-light/source/checkout.

[10] Conor Mcbride and James Mckinna. I am not a number: I am a free variable. In *In Haskell workshop*, 2004.

[11] Joan Moschovakis. Intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2010 edition, 2010.

[12] Gopalan Nadathur. Teyjus: A λprolog implementation. *Association of Logic Programming Newsletter*, May 2009.

[13] Alvaro Lozano Robledo. Peano arithmetic, February 2004. http://planetmath.org/encyclopedia/PeanoArithmetic.html.

[14] Peter Selinger. Lecture notes on the lambda calculus, 2007.

[15] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Y. Bertot et al., editor, *Theorem Proving in Higher Order Logics, 12th International Conference*. Springer, 1999.