$\lambda {\rm P}-$: a pure type system for first order logic with automated theorum proving

Eindhoven University of Technology
Department of Mathematics and Computing Science

λP−: A Pure Type System for First Order Logic with Automated Theorem Proving

by

Michael Franssen

97/15

ISSN 0926-4515

editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 97/15
Eindhoven, October 1997

λP$^{-:}$ A Pure Type System for First Order Logic with Automated Theorem Proving

Michael Franssen

October 14, 1997

Abstract

In this document we define multi sorted first order logic and discuss several proof systems for this logic: natural deduction, tableau methods, resolution methods and the framework of Pure Type Systems. We present a Pure Type System called λP- that corresponds exactly to first order logic. We also show the correspondence between λP- and natural deduction and we present an algorithm to convert closed tableaus to λ-terms in λP-. λP-, combined with the conversion algorithm, forms the formal basis for an interactive theorem prover in which an automated theorem prover based on tableaus can be safely used.

Contents

Chapter 1

Introduction

1.1 Purpose of this Paper

The purpose of this paper is to provide a theoretical foundation for an interactive theorem prover for first order logic with a high degree of automation. This theorem prover will be used in an environment for the derivation of correct programs, but only to perform correctness proofs, not to model the semantics of the programming language.

An interactive theorem prover with a high degree of automation combines the fields of interactive theorem proving and automated theorem proving. Both approaches to theorem proving have advantages and drawbacks. We will briefly present them here.

The reason for this comparison is that we want to create a proof system that combines the advantages of interactive theorem proving with the advantages of automated theorem proving. If possible, we want to use the advantages of the first approach to eliminate the drawback of the second approach and vice versa.

1.1.1 Interactive Theorem Proving

In interactive theorem proving one often uses typed lambda calculi, since these calculi provide an easy way to represent unfinished proofs as lambda terms with typed holes. The holes then have to be filled in with terms of the correct type to complete the proof. Besides, typed lambda calculi have the following advantages:

- Verification of proofs is possible by type checking.
- Proofs can be communicated as λ-terms, since λ-terms are a standard representation of a proof.
- There is a uniform treatment of first order logics and higher order logics.
- Typed λ-calculi are suitable for interactive theorem proving, since partial proofs can be represented as λ-terms with 'holes' that represent the unsolved parts.

Examples of interactive theorem provers based on typed λ-calculi are HOL, COQ and LEGO (see [Fra97] for a more elaborate overview).

A drawback of interactive theorem provers based on typed λ-calculi is that automating the proving process is hard. One of the reasons is that typed λ-calculi are usually used to model higher order logics.

1.1.2 Automated Theorem Provers

Automated theorem provers (ATPs) are often based on the method of tableaus or on resolution methods, since these methods are powerful enough to attack all problems formulated in first order logic. Besides, it is known that for every true formula in first order logic a tableau proof and a resolution proof exist, although it may not always be computable. Both methods are based on classical first order logic[1]. ATPs can prove nontrivial theorems, but not (yet) hard theorems. They are most beneficial in proofs that are not hard but tedious (e.g. a proof consisting of many simple case distinctions in which the ATP can prove the separate cases).

Proofs constructed by an ATP are usually not suitable for a human reader. This gives rise to problems when the user has to interact with the ATP if the ATP can not find a proof fully automatically. The user will then hardly be able to see where the difficulties arise and how they should be solved. Once the user has interacted with the ATP, the proof is no longer automatically reproducible. If the ATP had to reproduce the proof, it would again not find it fully automatically. To find the same proof again, the same user-interaction as before is required.

Also, since there is no standard representation of the proof, a witness of the actual proof can not be communicated to other proof systems. These systems can therefore not be used to verify the automatically constructed proof. Tableau based theorem provers do construct a tableau as a representation of the proof, but this is not a communicatable representation. Resolution methods do not produce a representation of the proof at all, although many implementations provide an ad hoc (non-standard) internal representation for use within the same system.

A user benefits from an ATP the most if it is embedded in an interactive system which assists in proving hard theorems. The user can then invoke the ATP to deal with tedious or simple parts of the larger proof. This is exactly what is made possible with the system λP– presented in this paper: combining meaningful ATP with an interactive theorem prover based on a typed λ-calculus.

1.2 First Order Predicate Logic

Since in this paper we deal with first order predicate logic, we will formally introduce first order logic. We can then describe the proof methods for this logic more accurately.

In literature, formulas of first order predicate logic are defined as follows: Let $\mathcal{F}$ be a set of function symbols, each with a fixed arity ≥ 0. Furthermore, let $\mathcal{P}$ be a set of predicate symbols, with each a fixed arity ≥ 0. We assume the existence of an infinite set A of variables. Then the set T of terms is defined recursively as:

1. $A \subseteq T$
2. If $f \in \mathcal{F}$ with arity n and $t_1, \ldots, t_n \in T$, then $ft_1 \ldots t_n \in T$.

The set $*_p$ of formulas is defined recursively as:

1. If $P \in \mathcal{P}$ with arity n and $t_1, \ldots, t_n \in T$, then $Pt_1 \ldots t_n \in *_p$.
2. If $P, Q \in *_p$, then $P \wedge Q \in *_p$, $P \vee Q \in *_p$ and $P \Rightarrow Q \in *_p$.
3. If $P \in *_p$, then $\neg P \in *_p$.

[1] There are versions of the method of tableaus that are suitable for intuitionistic and modal first order logic, but these are not standard tableau methods. We will not consider these methods in this paper, but the interested reader is referred to [dS93]

4. If $P \in *_p$ and $x \in A$, then $\forall x.P \in *_p$ and $\exists x.P \in *_p$.

We use the variable convention, which means that:

1. Free variables will always have a name different from bound variables.
2. If formulas differ only in the names of their bound variables, they are considered to be equal (α-equality).

Substitutions

The result of substituting a term t for a variable x in P is denoted as P_t^x. The substitution is defined as follows:

x_t^x	$\equiv$	t	
y_t^x	$\equiv$	y	
$(ft_1 \ldots t_n)_t^x$	$\equiv$	$f(t_1)_t^x \ldots (t_n)_t^x$	
$(Pt_1 \ldots t_n)_t^x$	$\equiv$	$P(t_1)_t^x \ldots (t_n)_t^x$	
$(P \wedge Q)_t^x$	$\equiv$	$P_t^x \wedge Q_t^x$	
$(P \vee Q)_t^x$	$\equiv$	$P_t^x \vee Q_t^x$	
$(P \Rightarrow Q)_t^x$	$\equiv$	$P_t^x \Rightarrow Q_t^x$	
$(\neg P)_t^x$	$\equiv$	$\neg(P_t^x)$	
$(\forall x.P)_t^x$	$\equiv$	$\forall x.P$	
$(\forall y.P)_t^x$	$\equiv$	$\forall y.P_t^x$	Ok, because of variable convention.
$(\exists x.P)_t^x$	$\equiv$	$\exists x.P$	
$(\exists y.P)_t^x$	$\equiv$	$\exists y.P_t^x$	Ok, because of variable convention.

$\mathcal{F}$ and $\mathcal{P}$ are the parameters of this framework. A weakness of the definitions above is that all terms are treated equally. In practice, we often want to distinguish between terms of different "types" (for instance, booleans and integers). Therefore, we will introduce a more general definition.

In addition to $\mathcal{F}$ and $\mathcal{P}$ we have a parameter $*_s$ that represents a set of basic types. With every function symbol $f \in \mathcal{F}$ with arity n, we associate a unique tuple of types $(U_1, \ldots, U_n, U)$, where $U_1, \ldots U_n$ and U are elements of $*_s$. We denote this as $f : (U_1, \ldots U_n, U) \in \mathcal{F}$. With every predicate symbol $P \in \mathcal{P}$ with arity n, we associate a unique tuple of types $(U_1, \ldots, U_n)$, where $U_1, \ldots, U_n \in *_s$. This is denoted as $P : (U_1, \ldots, U_n) \in \mathcal{P}$. Since the arity of function and predicate symbols can now be derived from its unique associated tuple of types it will no longer be stated explicitly. The set A of variables in the extended framework contains variables a with each a unique associated type U, where $U \in *_s$. We assume that for every type there are infinitely many variables. The definition of the set T of typed terms is:

1. $a : U \in T$ for every variable a with associated type U.
2. If $f : (U_1, \ldots, U_n, U) \in \mathcal{F}$ and $t_1 : U_1, \ldots, t_n : U_n \in T$ then $ft_1 \ldots t_n : U \in T$.

The set $*_p$ of formulas for multisorted first order logic is now defined as:

1. If $P : (U_1, \ldots, U_n) \in \mathcal{P}$ and $t_1 : U_1, \ldots, t_n : U_n \in T$, then $Pt_1 \ldots t_n \in *_p$.
2. If $P, Q \in *_p$, then $P \wedge Q \in *_p$, $P \vee Q \in *_p$ and $P \Rightarrow Q \in *_p$.
3. If $P \in *_p$, then $\neg P \in *_p$.
4. If $P \in *_p$ and $x : U \in A$, then $\forall x : U.P \in *_p$ and $\exists x : U.P \in *_p$.

One important subset of $*_p$ is the set of *atomic formulas*. This is the set of all propositions $Pt_1 \ldots t_n \in *_p$, with $P \in \mathcal{P}$ and $t_1, \ldots, t_n \in T$, and their negations. Note that substitution of terms for variables is now only defined for terms and variables with the same associated type.

This framework is more general than the first one, since the original framework can be obtained by choosing $*_s \equiv \{U\}$.

1.2.1 Semantics of First Order Predicate Logic

The semantics of first order logic is such that every closed formula represents a proposition. The meaning of the proposition depends on the interpretation of the predicate symbols and function symbols the user of the logic has in mind. On the other hand, there are closed formulas that model true propositions independent of the interpretation of the user. These formulas are called tautologies. Tautologies provide self-contained information. They can be seen as legal statements that will always hold, regardless of the meaning of the predicate symbols.

The semantics of first order logic presented here consists of a mapping from the syntactical set of type symbols, function symbols and predicate symbols to real sets, functions and relations. Following the line of [dN95] page 25: Let Γ be a set of first order formulas. An interpretation I of Γ is an ordered tuple $I = (\mathcal{D}, [\,])$, where

- $\mathcal{D}$ is a set of nonempty domains.
- $[\,]$ is a function which attaches
  - to every type symbol U in $*_s$ a domain in $\mathcal{D}$. We denote this domain as $[U]$.
  - to every function symbol f occurring in Γ with associated type $(U_1, \ldots, U_n, U)$ (and hence, arity n) a total function $[U_1] \rightarrow \ldots \rightarrow [U_n] \rightarrow [U]$. We denote this function as $[f]$.
  - to every predicate symbol P occurring in Γ with associated type $(U_1, \ldots, U_n)$ (and hence, arity n) a subset of $[U_1] \times \ldots \times [U_n]$. We denote this subset as $[P]$. The elements of $[P]$ are the tuples for which the relation holds.
  - to every variable V with associated type U which is free in an $F \in \Gamma$ an element of $[U]$.

We extend the mapping $[\,]$ to attach a meaning to every term in T:

- If the term is a variable V , then $[V]$ is already defined (see above).
- If the term has the form $ft_1 \ldots t_n$, then $[ft_1 \ldots t_n] = [f]([t_1], \ldots, [t_n])$.

We also define the modified mappings $[\,]_d^{V:U}$ where V is a variable with associated type U and d is an element of $[U]$.The value of $[X]_d^{V:U}$ is defined as:

- $[X]$ if X is a type, a function symbol, a predicate symbol or a variable different form V. This includes undefined, i.e. if $[X]$ is undefined and X is a variable different from V then $[X]_d^{V:U}$ is undefined too.
- d if $X = V$.

We are now ready to define a model based on the interpretation I. We denote a model of first order logic, based on interpretation $I = (\mathcal{D}, [\,])$ as M^I. M^I is a function from propositional formulas to $\{t, f\}$ defined as:

1. $M^I(Pt_1 \ldots t_n) = t$ iff $([t_1], \ldots, [t_n]) \in [P]$.
2. $M^I(P \wedge Q) = t$ iff $M^I(P) = t$ and $M^I(Q) = t$.
3. $M^I(P \vee Q) = t$ iff $M^I(P) = t$ or $M^I(Q) = t$.
4. $M^I(P \Rightarrow Q) = t$ iff $M^I(P) = f$ or $M^I(Q) = t$.
5. $M^I(\neg P) = t$ iff $M^I(P) = f$.
6. $M^I(\forall x : U.P) = t$ iff for every $d \in [U]$ we have $M^{I'}(P) = t$ with $I' = (D, [\,]_d^{x:U})$.
7. $M^I(\exists x : U.P) = t$ iff there exists a $d \in [U]$ such that $M^{I'}(P) = t$ with $I' = (D, [\,]_d^{x:U})$.
8. $M^I(P) = f$ in all other cases.

To reason about the correctness of reasoning methods like natural deduction, tableau- and resolution methods, we introduce the relation $\models$ between sets of closed formulas. As usual, the definition is as follows:

$\Gamma \models \Delta$ iff for all models M^I for which $M^I(P) = t$ for all $P \in \Gamma$ there is at least one $Q \in \Delta$ with $M^I(Q) = t$

where Γ and Δ are sets of closed formulas. If Γ is empty then for all models M^I at least one formula in Δ is true. This is denoted as $\models \Delta$. If, in addition, Δ contains only one element Q, this element is a tautology since $M^I(Q) = t$ for all models M^I. If Δ is empty then $\Gamma \models \Delta$ is false, even if Γ is empty too, since there never is an element in Δ that is mapped to t in the interpretation I. If there are no models for which $M^I(P) = t$ for every formula P in Γ, then the set Γ is called inconsistent. This is denoted by $\Gamma \models \bot$. Obviously, if $\Gamma \models \bot$ then $\Gamma \models Q$ for every closed formula Q.

Chapter 2

Proof Methods for First Order Predicate Logic

In this chapter a brief description of several proof methods is given. The descriptions serve mainly as a gently introduction to the notations used and as a basis for the conversions between the proofs given in chapter 3.

2.1 Natural Deduction

Natural deduction is a method for reasoning with logical formulas. To define natural deduction, we introduce a relation $\vdash_{nd}$ between sets of formulas and a single formula. In this section $\vdash_{nd}$ is simply denoted as $\vdash$. The set of formulas is denoted without the curly brackets. If Γ is a set of formulas and P is a formula, then $\Gamma \vdash P$ holds if it can be derived by the following set of axioms ($\bot \in *_p$ is such that for all interpretations I the value $M^I(\bot)$ is f. Hence, $\bot$ has now become a syntactical formula):

$$\textit{start}\quad \Gamma, p \vdash p \qquad \textit{weakening}\quad \frac{\Gamma \vdash p}{\Gamma, \Delta \vdash p}$$

$$\Rightarrow I\quad \frac{\Gamma, p \vdash q}{\Gamma \vdash p \Rightarrow q} \qquad \Rightarrow E\quad \frac{\Gamma \vdash p \Rightarrow q \quad \Gamma \vdash p}{\Gamma \vdash q}$$

$$\vee I1\quad \frac{\Gamma \vdash p}{\Gamma \vdash p \vee q} \qquad \vee I2\quad \frac{\Gamma \vdash q}{\Gamma \vdash p \vee q} \qquad \vee E\quad \frac{\Gamma \vdash p \Rightarrow r \quad \Gamma \vdash q \Rightarrow r}{\Gamma \vdash p \vee q \Rightarrow r}$$

$$\wedge I\quad \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q} \qquad \wedge E1\quad \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p} \qquad \wedge E2\quad \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash q}$$

$$\neg I\quad \frac{\Gamma, p \vdash \bot}{\Gamma \vdash \neg p} \qquad \neg E\quad \frac{\Gamma \vdash \neg p \quad \Gamma \vdash p}{\Gamma \vdash \bot} \qquad \textit{falsum}\quad \frac{\Gamma \vdash \bot}{\Gamma \vdash p}$$

$$\forall I\quad \frac{\Gamma \vdash p^x_\alpha}{\Gamma \vdash \forall x : U.p} \qquad \forall E\quad \frac{\Gamma \vdash \forall x : U.p}{\Gamma \vdash p^x_t}$$

$$\exists I\quad \frac{\Gamma \vdash p^x_t}{\Gamma \vdash \exists x : U.p} \qquad \exists E\quad \frac{\Gamma \vdash \exists x : U.p \quad \Gamma \vdash (\forall x : U.p \Rightarrow q)}{\Gamma \vdash (\exists x : U.p) \Rightarrow q}$$

$$\text{classic}\quad \frac{\Gamma \vdash \neg\neg p}{\Gamma \vdash p}$$

where $\alpha \in A$ does not occur in Γ or p, x does not occur in q and term $t : U \in T$. A few remarks should be made to explain the rule $\forall I$:

- The type U is fixed, since x is a variable and with every variable we associate one fixed type.
- The type of α then also has to be U, since otherwise p^x_α would not be defined.

Since every variable α is also a term we can apply $\exists I$ whenever we can apply $\forall I$. In our semantics this is true, since every set U represents a non-empty domain and hence $(\forall x : U.P) \Rightarrow (\exists x : U.P)$.

In intuitionistic logic we do not have the rule *classic*. However, in this document we only consider classical logic. Note that the relation $\vdash$ is defined syntactically, while $\models$ is defined semantically.

As an example we will give an annotated derivation of $\vdash ((\exists x : U.p) \wedge (\forall x : U.p \Rightarrow q)) \Rightarrow (\exists x : U.q)$:

(0)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash (\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q))$	*start*
(1)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash (\exists x : U.p)$	$\wedge E1$ on (0)
(2)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash (\forall x : U.(p \Rightarrow q))$	$\wedge E2$ on (0)
(3)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash p^x_\alpha \Rightarrow q^x_\alpha$	$\forall E$ on (2)
(4)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)), p^x_\alpha \vdash p^x_\alpha$	*start*
(5)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)), p^x_\alpha \vdash p^x_\alpha \Rightarrow q^x_\alpha$	*weakening* on (3)
(6)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)), p^x_\alpha \vdash q^x_\alpha$	$\Rightarrow E$ on (4) and (5)
(7)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)), p^x_\alpha \vdash (\exists x : U.q)$	$\exists I$ on (6)
(8)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash p^x_\alpha \Rightarrow (\exists x : U.q)$	$\Rightarrow I$ on (7)
(9)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash (\forall x : U.(p \Rightarrow (\exists x : U.q)))$	$\forall I$ on (8)
(10)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash (\exists x : U.p) \Rightarrow (\exists x : U.q)$	$\exists E$ on (1) and (9)
(11)	$(\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q)) \vdash (\exists x : U.q)$	$\Rightarrow E$ on (1) and (10)
(12)	$\vdash ((\exists x : U.p) \wedge (\forall x : U.(p \Rightarrow q))) \Rightarrow (\exists x : U.q)$	$\Rightarrow I$ on (11)

$$\frac{\neg\neg P}{P}$$

$$\frac{P \wedge Q}{P,\ Q} \qquad \frac{\neg(P \Rightarrow Q)}{P,\ \neg Q} \qquad \frac{\neg(P \vee Q)}{\neg P,\ \neg Q}$$

$$\frac{\neg(P \wedge Q)}{\neg P \mid \neg Q} \qquad \frac{P \Rightarrow Q}{\neg P \mid Q} \qquad \frac{P \vee Q}{P \mid Q}$$

$$\frac{\exists x : U.P}{P^x_\alpha} \qquad \frac{\neg\forall x : U.P}{\neg P^x_\alpha}$$

$$\frac{\neg\exists x : U.P}{\neg\exists x : U.P,\ \neg P^x_t} \qquad \frac{\forall x : U.P}{\forall x : U.P,\ P^x_t}$$

Figure 2.1: Rules for the construction of tableaus for 1st order classical logic. α represents a fresh variable of type U. t represents an arbitrary term of type U.

There is a relation between $\models$ and $\vdash$, which is formulated in two theorems: the soundness theorem and the completeness theorem. For the formulation of these theorems, we assume that Γ is a set of closed formulas and P is a single closed formula.

Theorem 2.1.1 (soundness) *If* $\Gamma \vdash P$ *then* $\Gamma \models P$.

Theorem 2.1.2 (completeness) *If* $\Gamma \models P$ *then* $\Gamma \vdash P$.

Proving these properties is beyond the scope of this paper. The interested reader is referred to [dS93].

2.2 The Method of Tableaus

Another method to prove formulas in first order logic is the method of semantic tableaus. Tableau methods attempt to prove $\models P$ for a formula P by decomposition of P. Tableau methods are constructive in the sense that they build a representation of the proof. This representation is a labeled tree called the tableau. The labels are sets of formulas.

A tableau is constructed in the following manner:

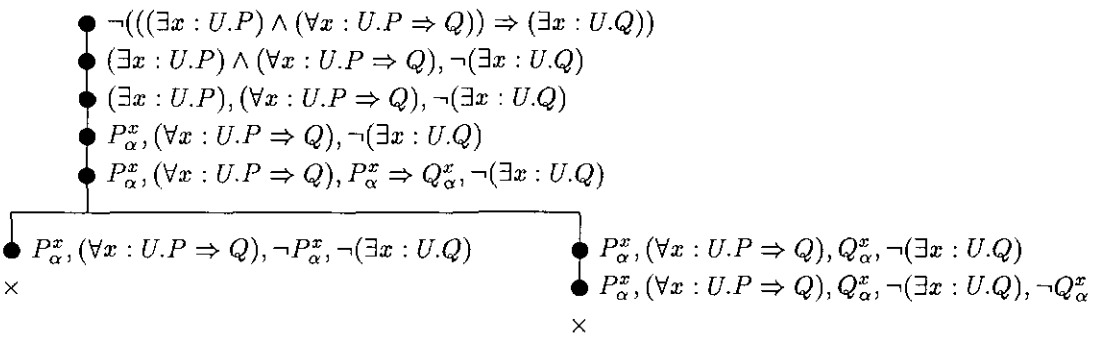
1. Start with a single node, labeled with $\{\neg P\}$, where P is the formula to be proved.
2. Select a leaf from the partially constructed tree. Select from the corresponding label L a formula X to which one of the rules from figure 2.1 can be applied. Extend the leaf with a number of nodes equal to the number of conclusions of the rule. Conclusions are separated by a '|' and can contain several formulas separated by a ','. A successor node is labeled with $(L \setminus \{X\}) \cup Y$, where Y is the conclusion for which the successor was created.
3. Repeat step 2 until for every leaf there is a formula P such that both P and $\neg P$ occur in its label. Such a leaf is called closed.

The process of constructing a tableau may be non-terminating. If the construction terminates and all leaves are closed, the tableau itself is called closed. If the construction process terminates and

there are leaves that are not closed, we can construct a counterexample for P. A counterexample for P is a consistent model, induced by an interpretation I, for which $M^I(P) = f$.

2.2.1 An example of a Tableau Proof

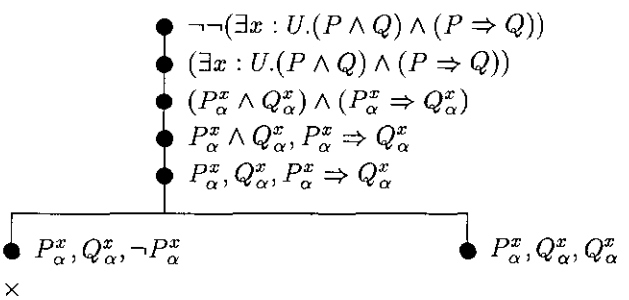
Below an example of a tableau is given that proves the formula $((\exists x : U.P) \wedge (\forall x : U.P \Rightarrow Q)) \Rightarrow (\exists x : U.Q)$.



The $\times$ below a leaf means that the leaf is closed.

2.2.2 Deriving a Counterexample from an open Tableau

From the following tableau we will derive a counterexample for $\neg(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q))$:



In general, a counterexample derived from a tableau is an interpretation in which the formula we tried to prove does not hold. To derive such an interpretation from an open branch in a tableau we only have to create an interpretation in which all the formulas in the label of the open branch are true. Since the search for a closed tableau terminated we know that all formulas in the label are atomic formulas and hence, we can easily construct the required interpretation.

In our example, we derive the following interpretation from the open branch with label $P^x_\alpha, Q^x_\alpha, Q^x_\alpha$: Take an arbitrary closed term $u : U \in T$. Choose for the interpretation I that $[P^x_u] = t$ and $[Q^x_u] = t$. This is possible, since by the nature of an open branch P is a predicate symbol. That the chosen interpretation is indeed a counterexample can easily be checked: In the model M^I we have $M^I((P^x_u \wedge Q^x_u) \wedge (P^x_u \Rightarrow Q^x_u)) = t$ and hence, $M^I(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q)) = t$ and thus, $M^I(\neg(\exists x : U.(P \wedge Q) \wedge (P \Rightarrow Q))) = f$. It follows that the theorem we were trying to prove was not in all interpretations true.

2.2.3 An Interpretation of the Method of Tableau

One can consider the construction of a tableau to be a search for an interpretation I, such that $M^I(\neg P) = t$ and hence, $M^I(P) = f$. If such an interpretation is found, we have a proof that not in all interpretations $M^I(P) = t$. Branches in the tableau indicate that there are two kinds of interpretations that are candidates for the search. For instance, if $\neg(A \wedge B)$ must hold for the interpretation I, then either $\neg A$ must hold for I or $\neg B$ must hold for I. This corresponds to the tableau rule

$$\frac{\neg(A \wedge B)}{\neg A \mid \neg B}$$

If a leaf is closed then the search for an interpretation failed. No interpretation can make a formula P to hold and make $\neg P$ hold at the same time. If all leaves are closed, the search for I failed altogether. We can then conclude that no interpretation I exists for which $M^I(\neg P) = t$, hence $M^I(\neg P) = f$ for all interpretations I and therefore $M^I(P) = t$ for all interpretations I. In short, we conclude $\models P$.

Tableau methods are suitable for automating. The rules are then applied in a specific order and several heuristics are used to obtain closed tableaus. These algorithms are usually referred to as *systematic tableaus*. An implementation of a systematic tableau method is presented in [SO94] and in [dS94]. Here, de Swart and Ophelders use abstract variables instead of concrete terms when applying a rule to $(\forall x : U.P)$ or $\neg(\exists x : U.P)$. A unification algorithm that respects certain restrictions is used to determine a substitute for the abstract variables such that the leaves of the tableau become closed.

The tableau method can be proved to be sound and complete. That is, there exists a closed tableau for P if and only if $\models P$. Unfortunately, it is not decidable for arbitrary P whether or not a closed tableau for P exists[1]. This is due to the nature of the algorithm: suppose we build tableaus in order to prove P. After we have built all possible trees with heights at most n without finding a closed tableau, we cannot conclude that a closed tableau does not exist. It is still possible that a closed tableau exists with height greater than n. The interested reader may read these proofs in [Smu68]. Soundness and completeness for an intuitionistic version of semantic tableaus can be proved fully intuitionistically (see [dS93]).

2.3 Resolution Methods

Resolution methods work with predicates modulo conjunction and disjunction. They do not distinguish the between the formulas $A \vee (B \vee C)$ and $(A \vee B) \vee C$, nor do they distinguish between $A \vee B$ and $B \vee A$. We present here only the basic techniques of resolution. More powerful techniques that can be used in resolution based theorem provers are found among others in [dN95].

To prove the validity of a formula P with resolution methods we first convert $\neg P$ into a suitable normal form Q. The structure of Q must be of the form $B_1 \wedge \ldots \wedge B_n$, where each B_i consists of a series of disjunctions of atomic formulas. The B_i's are called clauses. Clauses are an important concept of resolution methods. They may not contain any quantifications, but they can contain free variables.

To obtain the formula Q from $\neg P$ four transformations are applied to $\neg P$ consecutively. The first is a rewrite system that eliminates all negation symbols in front of non-atomic formulas

[1] For propositional logic (i.e. predicate logic without quantifications) it is decidable for arbitrary P whether or not there exists a closed tableau.

and all implication symbols. The second transformation rewrites the formula into a form with all quantifiers at the head of the formula. Then skolemization is used to eliminate all quantifiers, using skolem-functions for variables that are existentially quantified and free variables for variables that are universally quantified. The fourth transformation rewrites the formula into the required conjunctive normal form.

2.3.1 Eliminating Negations that are not part of Atomic Formulas and Implications:

The first rewrite-system eliminates negations that are not part of atomic formulas and implications. The rules defining this rewrite system are:

$$\begin{array}{lcl} A \Rightarrow B & \rightarrow & \neg A \vee B \\ \neg(\forall x : U.P) & \rightarrow & (\exists x : U.\neg P) \\ \neg(\exists x : U.P) & \rightarrow & (\forall x : U.\neg P) \\ \neg(A \wedge B) & \rightarrow & \neg A \vee \neg B \\ \neg(A \vee B) & \rightarrow & \neg A \wedge \neg B \\ \neg(\neg A) & \rightarrow & A \end{array}$$

After the first rewrite system has been applied as far as possible (i.e. no more rewrite steps can be applied) negation symbols only occur as part of atomic formulas and implication symbols do not occur at all. Note that application of the first rewrite system terminates in a unique resulting formula.

2.3.2 Moving Quantifiers to the Head of the Formula

The second rewrite system results in a formula that has all quantifiers at the head of the formula. This form is called Head Normal Form (HNF). The rules for this rewrite system are (x does not occur in Q):

$$\begin{array}{lcl} (\forall x : U.P) \wedge Q & \rightarrow & (\forall x : U.(P \wedge Q)) \\ (\forall x : U.P) \vee Q & \rightarrow & (\forall x : U.(P \vee Q)) \\ Q \wedge (\forall x : U.P) & \rightarrow & (\forall x : U.(Q \wedge P)) \\ Q \vee (\forall x : U.P) & \rightarrow & (\forall x : U.(Q \vee P)) \\ (\exists x : U.P) \wedge Q & \rightarrow & (\exists x : U.(P \wedge Q)) \\ (\exists x : U.P) \vee Q & \rightarrow & (\exists x : U.(P \vee Q)) \\ Q \wedge (\exists x : U.P) & \rightarrow & (\exists x : U.(Q \wedge P)) \\ Q \vee (\exists x : U.P) & \rightarrow & (\exists x : U.(Q \vee P)) \end{array}$$

Before rewriting a formula into HNF we rename the bound variables such that they all have a different name. Otherwise, we could rewrite $(\forall x : U.(\forall x : U.P(x)) \vee Q(x))$ to $(\forall x : U.(\forall x : U.P(x) \vee Q(x)))$, which is not valid. The correct result should read $(\forall x : U.(\forall y : U.P(y) \vee Q(x)))$. Note that this rewrite system again terminates with a unique result.

2.3.3 Eliminating Quantifiers

Skolemization is used to eliminate the quantifiers in the formula. Universally quantified variables are replaced by free variables, which are different iff they were bound by different quantifiers. These free variables are considered to be *implicitly* universally quantified. Existentially quantified variables are replaced by a so-called skolem function. The number and types of the arguments of

this function depend on the context in which the existentially quantified formula occurs. Formally, skolemisation is defined as follows:

Let $Q_1 \ldots Q_{n+m}.C$ be a formula in HNF, where Q_i is either a universal or an existential quantifier and C is quantifier-free. We then rename bound variables of universal quantifications from left to right to $x_1 : U_1, \ldots, x_n : U_n$ respectively. Also, we rename bound variables of existential quantifications from left to right to $y_1 : V_1, \ldots, y_m : V_m$ respectively. We denote the number of universal quantifiers that appear to the left of $\exists y_i : V_i$ in the original formula as a_i $(i = 1, \ldots m)$. For every y_i, we introduce a *skolem function* f_i . The skolem function f_i will have arity a_i and associated type $(U_1, \ldots, U_{a_i}, V_i)$. The skolemized representation of $Q_1 \ldots Q_{n+m}.C$ then reads C' , where C' is equal to C with all y_i replaced by $f_i(x_1, \ldots x_{a_i})$. Note that C' does no longer contain any quantifiers and that the free x_j are implicitly universally quantified.

Intuitively one can think of the skolem function as a replacement of the $\exists$-elimination. In a natural deduction style proof we used $(\exists y : V.P)$ formulas by introducing a fresh $\alpha : V$ and assuming that P^y_α for this particular α. The skolem function that now replaces α can be thought to produce exactly the α for which P^y_α holds. The arguments of the function make the dependencies of α from its context explicit.

We give a few examples:

Original formula	Skolemized formula
$\forall x_1 : U_1.\forall x_2 : U_2.\exists y_1 : V_1.$ $Px_1x_2 \wedge (\neg Px_1y_1 \vee \neg Py_1x_2)$	$Px_1x_2 \wedge (\neg Px_1f_1(x_1,x_2) \vee \neg Pf_1(x_1,x_2)x_2)$
$\forall x_1 : U_1.\forall x_2 : U_2.\forall x_3 : U_3.$ $(\neg Px_1x_2 \wedge \neg Px_2x_3) \vee Px_1x_3$	$(\neg Px_1x_2 \wedge \neg Px_2x_3) \vee Px_1x_3$
$\forall x_1 : U_1.\exists y_1 : V_1.$ $Px_1y_1 \wedge \neg Qy_1(gy_1)$	$Px_1f_1(x_1) \wedge \neg Qf_1(x_1)(g(f_1(x_1)))$
$\forall x_1 : U_1.\exists y_1 : V_1.\forall x_2 : U_2.$ $Px_1y_1 \wedge Py_1x_2 \Rightarrow Qx_1x_2$	$Px_1f_1(x_1) \wedge Pf_1(x_1)x_2 \Rightarrow Qx_1x_2$

2.3.4 Computing the Conjunctive Normal Form

The final transformation takes the quantifier free formula and transforms it into a conjunctive normal form. In conjunctive normal form, no conjunction symbols occur in any subexpression of a disjunction. The conjunctive normal form is obtained by application of a rewrite system with only 2 rules:

$$\begin{array}{rcl} A \vee (B \wedge C) & \rightarrow & (A \vee B) \wedge (A \vee C) \\ (A \wedge B) \vee C & \rightarrow & (A \vee C) \wedge (B \vee C) \end{array}$$

The formula obtained after all transformations has to form $C_1 \wedge \ldots \wedge C_n$, where every C_i has the form $A_i1 \vee \ldots \vee A_in_i$ and every A_ij is an atomic formula. A formula in such a form is suitable for resolution. Therefore, if we want to prove the validity of P, we perform the above transformations on $\neg P$ and then use resolution on the result Q. Every C_i is called a *clause*. Resolution treats a formula Q as a set of clauses, rather than a conjunction of clauses.

2.3.5 Applying Resolution

The actual resolution uses only one rule to derive a contradiction called the cut-rule:

$$\frac{A \vee X \quad \neg A \vee Y}{X \vee Y}$$

A is an atomic formula and X or Y may be disjunctions of several atomic formulas or negations of atomic formulas; they may even be absent, in which case we read e.g. X for $X \vee Y$. If X and Y are both empty we have an empty conclusion which represents our contradiction. The empty conclusion is denoted as $\square$.

As we stated before, resolution methods deal with formulas modulo commutativity and associativity. Conjuncts will be treated as separate formulas in the premises. Also, the P of the derivation rule can occur at any place in a series of disjunctions.

In order to apply the cut-rule on the result $Q \equiv C_1 \wedge \ldots \wedge C_n$ obtained by transformation of $\neg P$, we need two clauses from Q, say $C_1 \equiv A_1 \vee \ldots \vee A_n$ and $C_2 \equiv B_1 \vee \ldots \vee B_m$, with all A's and B's atomic formulas. Als we need a substitution, say θ. $C_1\theta$ must then contain an atomic formula $D = A_i\theta$ such that $C_2\theta$ contains $\neg D$, say $B_j\theta$. Without loss of generality, assume $i = j = 1$, then result of the cut-rule will be $R \equiv (A_2 \vee \ldots \vee A_n \vee B_2 \vee \ldots \vee B_m)\theta$. R is added as a new clause to the set of clauses. This process is continued until an empty clause $R \equiv \square$ is derived.

2.3.6 An Example of a Resolution Proof

An example: We want to prove:

$$((\exists x : U.Px) \wedge (\forall x : U.Px \Rightarrow Qx)) \Rightarrow (\exists x : U.Qx)$$

We start by transforming the negation of the above formula into the correct form:

$\neg(((\exists x : U.Px) \wedge (\forall x : U.Px \Rightarrow Qx)) \Rightarrow (\exists x : U.Qx))$
{removing $\Rightarrow$ and pushing negation towards atomic formulas}
$((\exists x : U.Px) \wedge (\forall x : U.\neg Px \vee Qx)) \wedge (\forall x : U.\neg Qx)$
{creating the head normal form}
$\exists x : U.\forall y : U.\forall z : U.(Px \wedge (\neg Py \vee Qy)) \wedge \neg Qz$
{skolemization: eliminating quantifiers, using more practical names than x_i and f_j}
$(Pf() \wedge (\neg Py \vee Qy)) \wedge \neg Qz$
{computing the conjunctive normal form. In this case nothing changes}
$(Pf() \wedge (\neg Py \vee Qy)) \wedge \neg Qz$

In order to make the resolution proof more readable, we denote all the clauses separately and use numbers to refer to them. The resolution proof of this formula reads:

(0)	$Pf()$	
(1)	$\neg Py \vee Qy$	
(2)	$\neg Qz$	
(3)	$Qf()$	[derived from (0) and (1), substitution $y := f()$]
(4)	$\square$	[derived from (2) and (3), substitution $z := f()$]

2.3.7 Unification

The required substitutions are obtained by unification algorithms. A unification algorithm usually returns a substitution called the most general unifier (mgu). The most general unifier has the following property: Let P and Q be formulas. Let θ be the mgu of P and Q. Since θ is a unifier, $P\theta = Q\theta$. The 'most general' character of θ is established by: Let σ be any substitution such that $P\sigma = Q\sigma$. Then there exists a σ' such that $P\theta\sigma' = P\sigma$ and $Q\theta\sigma' = Q\sigma$. Conversely, if two formulas can be made equal by performing a substitution for the free variables, then there exists a most general unifier with the mentioned property. Intuitively one can say that the mgu unifies two formulas while being as little specific as possible. Computing the most general unifier is decidable. An efficient unification algorithm is described in [MM82].

2.4 Pure Type Systems

Pure Type Systems (PTSs) emerged as a result of typed λ-calculi. In the untyped λ-calculus of Church one builds terms from:

1. an infinite list of variables and
2. two constructs to build more complex terms: abstraction and application.

Abstraction is meant to be a general procedure for function construction; application codes a possible function application.

A rewrite system called β-reduction is defined such that function application is actually 'executed'. Repeated rewriting may lead to a unique normal form, i.e. a λ-term to which none of the re-write rules can be applied anymore (a *normal form*). β-reduction is used to model the computation of a function when it is applied to arguments and the normal form represents the result of this computation. However, there are λ-terms that do not reduce to any normal form. Typed lambda calculi were introduced to define a subset of λ-terms in such a way that all λ-terms in this subset reduce to normal forms. Later, Barendregt, Berardi and Terlouw introduced Pure Type Systems: a parameterized framework in which many variants of typed λ-calculi can be described (see [Bar92]).

Due to the Curry-Howard-de Bruijn isomorphism between propositions and types, PTSs can also be used as a parameterized framework to model various kinds of logic. A type P then corresponds to a proposition in the logic. A term of type P encodes a proof of P, hence to find a proof of P it suffices to find a term of type P. In this section we will determine the parameters that produce a PTS which corresponds to the first order logic we introduced in section 1.2. But before we determine these parameters, we will introduce the Pure Type Systems.

2.4.1 The Definition of Pure Type Systems

The parameters of a pure type system are $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$. $\mathcal{S}$ is a set of sorts, $\mathcal{A}$ is a set of axioms being pairs of sorts and $\mathcal{R}$ is a set of rules being triples of sorts which defines the Π-formations that are allowed. The rules of a PTS define a relation $\vdash_\lambda$ between lists of typed λ-terms and a single typed λ-term. In this section $\vdash_\lambda$ is simply denoted as $\vdash$. The rules of a PTS are such that it is not necessary to define what terms are allowed: the terms themselves are formed by the rules. We only need one set of variable symbols to represent constants, functions, propositions etc. The rules of a PTS are ($s \in \mathcal{S}$):

start	$<> \vdash s1 : s2$	$(s1, s2) \in \mathcal{A}$
intro	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$	x is Γ-fresh
weaken	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}$	x is Γ-fresh
Π-*form*	$\dfrac{\Gamma \vdash A : s1 \quad \Gamma, x{:}A \vdash B : s2}{\Gamma \vdash (\Pi x{:}A.\ B) : s3}$	$(s1, s2, s3) \in \mathcal{R}$
Π-*intro*	$\dfrac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash B : s}{\Gamma \vdash (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$	
Π-*elim*	$\dfrac{\Gamma \vdash F : (\Pi x{:}A.\ B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$	
conversion	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$	

We give a brief comment on each rule:

start This is the only rule without premises in a PTS. It supplies, starting from the axioms in $\mathcal{A}$, basic typing judgements from which all the other typing judgements are derived.

intro Intro is used in a much more general sense than the *intro*-rule in natural deduction. In natural deduction intro allows one to add assumptions to the context. In a PTS intro allows one to add assumptions, constants (which in a PTS are equal to variables), functions and propositional variables (including predicates) to the context. This depends on the form of A. The type of the introduced item x depends on s, which is the type of the type of x.

weaken Weaken is needed to preserve existing deriviations in extended contexts. It states that everything that can be derived in a certain context can also be derived in a more extended context.

Π-*form* This rule allows the construction of function types, predicates, universal quantifications etc. The set of rules $\mathcal{R}$ of a PTS determines the ways in which $\Pi - form$ can be used. Actually, the set $\mathcal{R}$ states which abstractions are allowed.

Π-*intro* One needs this rule to actually construct terms of a type built with the previous rule. Without this rule, we could only *assume* that there are terms of this type by using *intro*.

Π-*elim* Once a term with a Π-type is constructed or assumed, it can be used to create a term with a more concrete type. The $\Pi - elim$ rule, also referred to as the application rule, instantiates the body of an abstract Π-type by substituting a term for the bound abstract variable.

conversion In several PTSs there is no unicity of types, i.e. a term A can have type B where B can be rewritten to B' by β-reduction. In the propositions-as-types isomorphism, B and B' then represent the same propositional formula and hence, A is a proof of B' just as well as it is a proof of B. To support this switch of representation the *conversion* rule is needed. $B =_\beta B'$ is read as B is β-equal to B', which means that there exists a B'' such that B and B' can both be reduced to B'' by β-reduction. A problem with the conversion rule is that it does not affect the term A, which makes type-checking more difficult.

For example, the system $\lambda_{\rightarrow}$ of Church is modeled by the PTS with parameters $\mathcal{S} \equiv \{*, \Box\}$, $\mathcal{A} \equiv \{(*, \Box)\}$ and $\mathcal{R} \equiv \{(*, *, *)\}$. The type $A \rightarrow B$ in the original system $\lambda_{\rightarrow}$ is now represented by $(\Pi x : A.B)$. For $\lambda_{\rightarrow}$ the rule *conversion* is superfluous, since no type can ever contain a λ and hence, no type can ever β-reduce to anything else but itself.

The relation between PTSs and logic will become more clear in the next part, in which we introduce a PTS that corresponds to our version of first order logic.

2.4.2 The System λP–

We want to have a PTS to which we can add automatic proof engines based on tableau methods. We therefore wish that if $\Gamma \vdash P : *_p$ that P is a formula in first order predicate logic. In most PTSs the structure of $P : *_p$ can be much more complex than formulas in first order logic. Therefore it is not possible to add the proof engines in such a way that they can be used on arbitrary propositions of these PTSs. The system λP– is designed in such a way that tableau- and resolution methods can be applied to any proposition that can be formed in λP–.

The PTS-parameters $\mathcal{S}$ and $\mathcal{A}$ of λP– are:

$$\begin{array}{rcl} \mathcal{S} & \equiv & \{*_s, \Box_s, *_p, \Box_p\} \\ \mathcal{A} & \equiv & \{(*_s, \Box_s), (*_p, \Box_p)\} \end{array}$$

As one would expect, $*_s$ will be used to represent the type of all sets and $*_p$ will be used to represent the type of all formulas. The sorts $\Box_s$ and $\Box_p$ are the types of $*_s$ and $*_p$ respectively. They enable the intro-rule to be used for introduction of set variables and propositional variables, which otherwise could only be introduced to the context by separate rules.

The definition of $\mathcal{R}$ is a little harder to give. It might appear that a variant of the system λP, obtained by $\mathcal{R} \equiv \{(*_p, *_p, *_p), (*_s, *_p, *_p), (*_s, *_s, *_s), (*_s, \Box_p, \Box_p)\}$, is a good candidate. The Π-types formed with these rules then correspond to our concepts of first order predicate logic in the following way ($U, V, W : *_s$ and $P, Q, R : *_p$)

type	represents	obtained by using
$(\Pi x : P.Q) : *_p$	$P \Rightarrow Q$	$(*_p, *_p, *_p)$
$(\Pi x : U.Q) : *_p$	$(\forall x : U.Q)$	$(*_s, *_p, *_p)$
$(\Pi x : U.V) : *_s$	the type (U, V) of a unary function symbol	$(*_s, *_s, *_s)$
$(\Pi x : U.(\Pi y : V.W)) : *_s$	the type (U, V, W) of a binary function symbol	$(*_s, *_s, *_s)$ twice
$(\Pi x : U.*_p) : \Box_p$	the type (U) of a unary predicate symbol	$(*_s, \Box_p, \Box_p)$
$(\Pi x : U.(\Pi y : V.*_p)) : \Box_p$	the type (U, V) of a binary predicate symbol	$(*_s, \Box_p, \Box_p)$ twice

Unfortunately, the type $(\Pi x : (\Pi y : U.V).*_p) : \Box_p$, being a predicate over a function, can also be formed in this version of λP. But quantifying over functions is not allowed in first order predicate logic and is not supported by tableau and resolution methods.

λP– in Regular PTS-style

We could solve this problem by adding the element $*_f$ to the set of sorts and change the set of rules to $\mathcal{R} = \{(*_s, *_s, *_f), (*_s, *_f, *_f), (*_s, *_p, *_p), (*_p, *_p, *_p), (*_s, \Box_p, \Box_p)\}$. The rule $(*_s, *_s, *_f)$ establishes that the type of functions with arity 1 have type $*_f$ instead of $*_s$. The rule $(*_s, *_f, *_f)$ allows us to build functions of arity greater than 1, but only functions which have elements of basic types (basic types have type $*_s$) as arguments. Hence, all Π-types that were of type $*_s$ in the previous variant are of type $*_f$ and therefore it is not possible to quantify over them. This

variant still has a drawback: if we have a context Γ such that $\Gamma \vdash F : (\Pi x : U.(\Pi y : V.W))$ and $\Gamma \vdash a : U$ then by the $\Pi - elim$ rule $\Gamma \vdash Fa : (\Pi y : V.W)$. But in our definition of first order predicate logic a function with arity 2 applied to a single argument does not even occur. Apparently, this pure type system has more terms than the first order predicate logic we want to model.

λP- in a PTS with Parametric Constants

A more elegant solution to the problem of unwanted quantifications over functions is found in chapter six of [Laa97]: a pure type system with parametric constants. Like in our definition of first order predicate logic a parametric pure type system has terms that only have a meaning after all the required arguments have been provided. That is, there are terms that cannot be typed unless they are applied to a number of parameters. Such a PTS is called a PPTS, where the extra P stands for parametric. To define a PPTS we need the set $\mathcal{P}$ of parameter rules in addition to the sets $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$ of a regular PTS. $\mathcal{P}$ contains pairs of sorts. In the following rules Δ stands for a list $x_1 : B_1, \ldots, x_n : B_n$ of named variables and Δ_i stands for the first $i-1$ elements of this list. In general B_i can depend on elements x_j with $j < i$ but in $\lambda P-$ this will not be the case. The additional rules of a PPTS are:

$$P\text{-}weaken \qquad \frac{\Gamma \vdash b : B \quad \Gamma, \Delta_i \vdash B_i : s_i \quad \Gamma, \Delta \vdash A : s}{\Gamma, c(\Delta){:}A \vdash b : B} \qquad (s_i, s) \in \mathcal{P}$$

$$P\text{-}app \qquad \frac{\begin{array}{ll}\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash b_i : B_i[x_j := b_j]_{j=1}^{i-1} & \text{for } i = 1, \ldots, n \\ \Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash A : s & \text{if } n = 0\end{array}}{\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash c(b_1, \ldots, b_n) : A[x_j := b_j]_{j=1}^{n}}$$

Again, we give a brief comment on these rules:

P-weaken The $P-weaken$ rule allows us to add a parametric constant to the context. In contrast to other extensions of the context this rule does not allow us to type the parametric constant itself, while the intro-rule (used for regular extensions of the context) allows the typing of every newly added item.

P-app Since a parametric constant itself cannot be typed in a PPTS it cannot be used with the usual application (or $\Pi-elim$) rule. The rule $P-app$ allows us to use a parametric constant, but only if we supply all the required arguments at once. This corresponds to functions and predicates in first order logic: these too can only be used after all the arguments have been supplied. The special premise for the case $n = 0$ is needed to assure that the context $\Gamma_1, c(\Delta) : A, \Gamma_2$ is a valid one.

We now can state the following definitions of $\mathcal{R}$ and $\mathcal{P}$ for λP-:

$$\begin{array}{rcl}\mathcal{R} & \equiv & \{(*_s, *_p, *_p), (*_p, *_p, *_p)\} \\ \mathcal{P} & \equiv & \{(*_s, *_s), (*_s, \Box_p)\}\end{array}$$

For notational convenience, we will denote the type $(\Pi x : U.P)$, introduced by application of the rule $(*_s, *_p, *_p)$, as $(\forall x : U.P)$ and the type $(\Pi x : P.Q)$, introduced by application of the rule $(*_p, *_p, *_p)$, as $P \Rightarrow Q$.

Simplifying the Parameter Mechanism

Since in $\lambda P-$ types of kind $*_s$ cannot depend on any term (i.e. we only have basic types) the rules $P - weaken$ and $P - app$ can be simplified. Also, all rules in $\mathcal{P}$ start with $*_s$. We then

get the following simplified version of the additional rules:

$$P-weaken \qquad \frac{\Gamma \vdash b : B \quad \Gamma, \Delta \vdash B_i : *_s \quad \Gamma, \Delta \vdash A : s}{\Gamma, c(\Delta){:}A \vdash b : B} \qquad (*_s, s) \in \mathcal{P}$$

$$P-app \qquad \frac{\begin{array}{ll}\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash b_i : B_i & \text{for } i = 1, \ldots, n \\ \Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash A : s & \text{if } n = 0\end{array}}{\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash c(b_1, \ldots, b_n) : A}$$

Note that these rules still contain superfluous elements: the names of the parameters (hidden in Δ) are not needed, since the types are not depending on them. This is similar to the term $(\Pi p : P.Q)$ obtained by $\Pi - form$ with rule $(*_p, *_p, *_p)$: In this case one uses the abbreviation $P \Rightarrow Q$, since the P is not used in Q and hence needs not to be denoted. However, in the case of parametric constants we will maintain the variable names, since this is more closely to the way mathematicians denote function definitions and the way in which functions are declared in programming languages.

Extending λP– to First Order Logic

Also, it can be seen by the set of rules of λP– that the system is a refinement of $\lambda_{\rightarrow}$. As a consequence, the *conversion*-rule is superfluous: If $\Gamma \vdash A : B$ and $\Gamma \vdash B' : s$ such that $B =_\beta B'$ then $B = B'$. As a consequence, unification, and thereby type-checking, will be much easier in λP– than in a regular PTS version that models first order predicate logic.

On the other hand, we do not yet have conjunction, disjunction, negation and existential quantification in λP–. The rules for these constructs do not belong to the standard PTS rules, because there exist higher order codings for these constructs that can be built in more powerful PTSs. However, we will add the constructs to λP– by adding rules, since the codings cannot be built within λP–. The rules are:

$\bot$-*intro* $$<> \vdash \bot : *_p$$

falsum $$\frac{\Gamma \vdash p : \bot \quad \Gamma \vdash P : *_p}{\Gamma \vdash pP : P}$$

classic $$\frac{\Gamma \vdash p : (P \Rightarrow \bot) \Rightarrow \bot}{\Gamma \vdash classic\ P\ p : P}$$

$\wedge$-*form* $$\frac{\Gamma \vdash P : *_p \quad \Gamma \vdash Q : *_p}{\Gamma \vdash P \wedge Q : *_p}$$

$\wedge$-*intro* $$\frac{\Gamma \vdash p : P \quad \Gamma \vdash q : Q \quad \Gamma \vdash P \wedge Q : *_p}{\Gamma \vdash (p, q) : P \wedge Q}$$

$\wedge$-$elim_1$ $$\frac{\Gamma \vdash p : P \wedge Q}{\Gamma \vdash \pi_1(p) : P}$$

$\wedge$-$elim_2$ $$\frac{\Gamma \vdash p : P \wedge Q}{\Gamma \vdash \pi_2(p) : Q}$$

$\vee$-*form* $$\frac{\Gamma \vdash P : *_p \quad \Gamma \vdash Q : *_p}{\Gamma \vdash P \vee Q : *_p}$$

$\vee$-$intro_1$ $$\frac{\Gamma \vdash p : P \quad \Gamma \vdash P \vee Q : *_p}{\Gamma \vdash injl\ (P \vee Q)\ p : P \vee Q}$$

$\vee$-$intro_2$ $$\frac{\Gamma \vdash q : Q \quad \Gamma \vdash P \vee Q : *_p}{\Gamma \vdash injr\ (P \vee Q)\ q : P \vee Q}$$

$\vee$-*elim* $$\frac{\Gamma \vdash p : P \Rightarrow R \quad \Gamma \vdash q : Q \Rightarrow R}{\Gamma \vdash (p \nabla q) : (P \vee Q) \Rightarrow R}$$

$\exists$-*form* $$\frac{\Gamma \vdash U : *_s \quad \Gamma, x{:}U \vdash P : *_p}{\Gamma \vdash (\exists x : U.P) : *_p}$$

$\exists$-*intro* $$\frac{\Gamma \vdash p : P_t^x \quad \Gamma \vdash (\exists x : U.P) : *_p}{\Gamma \vdash inj\ (\exists x : U.P)\ p\ t : (\exists x : U.P)}$$

$\exists$-*elim* $$\frac{\Gamma \vdash Q : *_p \quad \Gamma \vdash (\exists x : U.P) : *_p \quad \Gamma \vdash p : (\forall x : U.(P \Rightarrow Q))}{\Gamma \vdash \Diamond\ (\exists x : U.P)\ p : (\exists x : U.P) \Rightarrow Q}$$

Negation does not occur in the rules above, but we can model the negation of P by $P \Rightarrow \bot$. For convenience we will denote this as $\neg P$.

Adding all the rules above makes λP– a relatively large system compared to usual PTSs. This does not mean that the system is truly more complex: the rules for $\wedge$, $\vee$ and $\exists$ appear in groups with each a *form, intro* and *elim* part. There may be many rules, but they are not difficult to verify.

To model a first order predicate logic[2] in λP–, we need a basic context for λP–. Let $*_s^L$, $\mathcal{F}^L$ and

[2] To distinguish between the parameters of the first order logic framework and the parameters and sorts of the

$\mathcal{P}^L$ be the set of basic sets, the set of function-symbols and the set of predicate symbols of the first order logic respectively. Furthermore, let T^L be the set of terms defined by $*_s^L$ and $\mathcal{F}^L$. Then the basic context is build in the following way:

1. For every $U \in *_s^L$ the term $U : *_s$ is added to the context by application of the *intro*-rule of the PTS where $\Box_s$ is used for s.
2. For every $f : (U_1, \ldots, U_n, U) \in \mathcal{F}^L$ we add the parametric constant $f(x_1 : U_1, \ldots, x_n : U_n) : U$ to the context. These elements are introduced by application of the $P - weaken$-rule of the PTS by taking for s the sort $*_s$.
3. For every $P : (U_1, \ldots, U_n) \in \mathcal{P}^L$ we add the parametric constant $P(y_1 : U_1, \ldots, y_n : U_n) : *_p$ to the context. Once more, this is done by application of the $P - weaken$-rule, now with sort $\Box_p$ substituted for s.

The context obtained by the steps above is denoted as Γ_L. The relation between a first order logic L and λP– with context Γ_L is expressed in the following theorems:

Theorem 2.4.1 $\Gamma_L \vdash U : *_s$ *iff* $U \in *_s^L$.

Theorem 2.4.2 *For any* U *with* $\Gamma_L \vdash U : *_s$ *we have* $\Gamma_L \vdash t : U$ *iff* $t : U \in T^L$.

Theorem 2.4.3 $\Gamma_L \vdash P : *_p$ *iff* $P \in *_p^L$.

Theorem 2.4.4 $\Gamma_L \vdash p : P$ *iff* $\models P$.

Theorem 2.4.1 till theorem 2.4.3 are proved by induction on the structure of the terms. Theorem 2.4.4 follows from the conversion algorithm presented in the next chapter: we can convert a closed tableau into a λ-term in λP– and the tableau method is complete, hence λP– is complete.

To present the conversion algorithm, we also need two other theorems:

Theorem 2.4.5 *Let* $\Delta_1, A : B, \Delta_2$ *be a legal context (i.e. it is possible to derive* $\Delta_1, A : B, \Delta_2 \vdash *_s : \Box_s$*). Then* $\Delta_1, A : B, \Delta_2 \vdash A : B$.

Theorem 2.4.6 *Let* $\Delta_1, A : B, \Delta_2$ *be a legal context such that* Δ_1, Δ_2 *is also a legal context. Then if* $\Delta_1, \Delta_2 \vdash C : D$ *then* $\Delta_1, A : B, \Delta_2 \vdash C : D$.

This concludes our introduction to λP–. We will use λP– later to encode proofs found by tableau based proof engines in λ-terms. This way, we can use the power of ATP in a theorem prover based on a PPTS, without extending the logic. Also the proofs found by these proof engines can then be communicated and checked by other theorem provers.

PTS we will add a 'L' in superscript to the parameters and sets of the first order logic framework.

Chapter 3

Conversion of Proofs

This chapter describes (in a way) relations between several proof systems. The main relevance of the chapter is that it is shown that several proof styles can be combined without violating the logic of the system. The logic is not extended or limited by combining the systems (e.g. adding a tableau-based ATP to λP–). The advantages of combined systems is that the user of an implementation can use his own favorite proof style, while the implementation uses a single formalism.

We are particularly interested in combing a PTS-based theorem prover with a tableau based theorem prover. This combination adds powerful ATP to a PTS in which usually everything had to be proved manually. On the other hand, it provides a good basis to implement user interaction with an ATP, which usually would be very hard.

In order to show that λP–corresponds to classical first order logic on which tableau methods are based, we also present the Curry-Howard-deBruijn isomorphism of propositions as types for λP–.

3.1 λ-terms as Natural Deduction Proofs

λ-terms can be seen as encodings of natural deduction proofs. In fact this does not even require a real transformation. Just erasing the terms, all $-form$-rules and all premises considering well-formedness conditions (i.e. premises ending with "$: *_p$" or "$: *_s$") from the axioms of λP– gives us an axiomatic system that looks surprisingly much like the natural deduction axioms. For example:

$$\frac{\Gamma \vdash p : P \Rightarrow R \quad \Gamma \vdash q : Q \Rightarrow R}{\Gamma \vdash (p \nabla q) : (P \vee Q) \Rightarrow R} \text{ becomes } \frac{\Gamma \vdash P \Rightarrow R \quad \Gamma \vdash Q \Rightarrow R}{\Gamma \vdash (P \vee Q) \Rightarrow R}$$

The λ-terms are constructed in such a way that every λ-term corresponds to exactly one derivation in natural deduction. For instance, the term $(p \nabla q)$ encodes that proofs p and q of $P \Rightarrow R$ and $Q \Rightarrow R$ respectively are used to proof $(P \vee Q) \Rightarrow R$. The ∇-symbol indicates the way in which the proofs are combined. In fact, the structure of λ-terms administrates the order in which axioms of the PTS are applied in order to get a proof of the proposition represented by their type.

We will not show how every λ-construct can be converted to the corresponding step in natural deduction, since most of this is trivial. However, we will comment on two translations that are less direct:

1. A λ-abstraction corresponds to either an application of $\Rightarrow I$ or an application of $\forall I$ depending on its type (λ-abstractions always have a Π-type). In case of a λ-abstraction, checking

the type of the term provides us the information we need in order to translate (or better: read) the proof encoded in the λ-term.

- If the type is obtained by using the rule $(*_s, *_p, *_p)$ then the λ-term corresponds to $\forall I$.
- If we need $(*_p, *_p, *_p)$ to construct the type then the λ-term corresponds to $\Rightarrow I$.

2. Application terms correspond to an application of $\Rightarrow E$ or an application of $\forall E$. For application Fa we check the type of F.
   - If it is constructed using $(*_s, *_p, *_p)$ then the application corresponds to using the $\forall E$-rule.
   - If the type of F was constructed using $(*_p, *_p, *_p)$ it corresponds to using $\Rightarrow E$.

We finish explaining the correspondence between natural deduction proofs and λ-terms with two examples.

Example 1: From Natural Deduction to λP– The following derivation in λP– follows the proof of $((\exists x : U.p) \wedge (\forall x : U.p \Rightarrow q)) \Rightarrow (\exists x : U.q)$, which was given in natural deduction on page 9.

(0)	$\Gamma_L, p : (\exists x : U.P) \wedge (\forall x : U.(P \Rightarrow Q)) \vdash p : (\exists x : U.P) \wedge (\forall x : U.(P \Rightarrow Q))$	note 1
(1)	$\Gamma_L, \Gamma \vdash \pi_1(p) : (\exists x : U.P)$	
(2)	$\Gamma_L, \Gamma \vdash \pi_2(p) : (\forall x : U.(P \Rightarrow Q))$	
(3)	$\Gamma_L, \Gamma, \alpha : U \vdash \pi_2(p)\alpha : P^x_\alpha \Rightarrow Q^x_\alpha$	note 2
(4)	$\Gamma_L, \Gamma, \alpha : U, p' : P^x_\alpha \vdash p' : P^x_\alpha$	
(5)	$\Gamma_L, \Gamma, \alpha : U, p' : P^x_\alpha \vdash \pi_2(p)\alpha : P^x_\alpha \Rightarrow Q^x_\alpha$	
(6)	$\Gamma_L, \Gamma, \alpha : U, p' : P^x_\alpha \vdash \pi_2(p)\alpha p' : Q^x_\alpha$	
(7)	$\Gamma_L, \Gamma, \alpha : U, p' : P^x_\alpha \vdash inj(\exists x : U.Q)(\pi_2(p)\alpha p')\alpha : (\exists x : U.Q)$	
(8)	$\Gamma_L, \Gamma, \alpha : U \vdash (\lambda p' : P^x_\alpha.(inj(\exists x : U.Q)(\pi_2(p)\alpha p')\alpha)) : P^x_\alpha \Rightarrow (\exists x : U.Q)$	
(9)	$\Gamma_L, \Gamma \vdash (\lambda x : U.(\lambda p' : P.(inj(\exists x : U.Q)(\pi_2(p)xp')x))) : (\forall x : U.P \Rightarrow (\exists x : U.Q))$	note 3
(10)	$\Gamma_L, \Gamma \vdash (\Diamond(\exists x : U.P)(\lambda x : U.(\lambda p' : P.(inj(\exists x : U.Q)(\pi_2(p)xp')x))))$ $: (\exists x : U.P) \Rightarrow (\exists x : U.Q)$	
(11)	$\Gamma_L, \Gamma \vdash (\Diamond(\exists x : U.P)(\lambda x : U.(\lambda p' : P.(inj(\exists x : U.Q)(\pi_2(p)xp')x)))) \ \pi_1(p)$ $: (\exists x : U.Q)$	
(12)	$\Gamma_L \vdash \lambda p : (\exists x : U.P) \wedge (\forall x : U.(P \Rightarrow Q))$ $.(\Diamond(\exists x : U.P)(\lambda x : U.(\lambda p' : P.(inj(\exists x : U.Q)(\pi_2(p)xp')x)))) \ \pi_1(p)$ $: ((\exists x : U.P) \wedge (\forall x : U.(P \Rightarrow Q))) \Rightarrow (\exists x : U.Q)$	

note 1 We use Γ as shorthand for $p : (\exists x : U.P) \wedge (\forall x : U.(P \Rightarrow Q))$. We omit derivations of the types because of space.

note 2 In a PTS the variable α has to occur in the context explicitly.

note 3 We rename α to x, which is possible, since α does not occur within the body of $(\exists x : U.Q)$. This also allows us to write P for P^x_x.

Example 2: From λP– to Natural Deduction

The second example will show how to construct a natural deduction proof from two different λ-terms that both have type $(P \wedge Q) \Rightarrow (P \vee Q)$. The terms are $(\lambda p : P \wedge Q.injl\ (P \vee Q)\ \pi_1(p))$ and $(\lambda p : P \wedge Q.injr\ (P \vee Q)\ \pi_2(p))$ respectively. Even though both terms have the same type they are different and hence, they encode different proofs.

Decoding the first λ-term

We will decode the first λ-term in detail. Decoding results in a proof (or derivation) of $\vdash (P \wedge Q) \Rightarrow (P \vee Q)$ in natural deduction.

The first term is an abstraction of proofs p of $P \wedge Q$ over $injl\ (P \vee Q)\ \pi_1(p)$. Its type $(P \wedge Q) \Rightarrow (P \vee Q)$ is obtained by using $(*_p, *_p, *_p)$, hence the abstraction indicates application of the rule $\Rightarrow I$ from the natural deduction system. The premise of $\Rightarrow I$ is $\Gamma, p \vdash q$. Since the conclusion should read $(P \wedge Q) \Rightarrow (P \vee Q)$ we must now find a derivation of $P \wedge Q \vdash P \vee Q$.

The next step is analyzing the body of the abstraction: $injl\ (P \vee Q)\ \pi_1(p)$. The $injl$ indicates that in natural deduction the rule $\vee I1$ should be used. The result should read $P \wedge Q \vdash P \vee Q$ and hence, we require a derivation of $P \wedge Q \vdash P$. This proof is encrypted by $\pi_1(p)$.

$\pi_1(p)$ indicates that we should use natural deduction rule $\wedge E1$. Since in the PTS-system p has type $P \wedge Q$, we have to apply $\wedge E1$ to $P \wedge Q$. Hence, we have to derive $P \wedge Q \vdash P \wedge Q$. This is trivial, since it is equal to the axiom *start* of natural deduction.

Altogether the proof of $\vdash (P \wedge Q) \Rightarrow (P \vee Q)$ encoded by the first λ-term reads:

(0)	$P \wedge Q \vdash P \wedge Q$	*intro*
(1)	$P \wedge Q \vdash P$	$\wedge E1$ on (0)
(2)	$P \wedge Q \vdash P \vee Q$	$\vee I1$ on (1)
(3)	$\vdash (P \wedge Q) \Rightarrow (P \vee Q)$	$\Rightarrow I$ on (2)

Decoding the second λ-term

The proof of the same formula as encoded by the second λ-term reads:

(0)	$P \wedge Q \vdash P \wedge Q$	*intro*
(1)	$P \wedge Q \vdash Q$	$\wedge E2$ on (0)
(2)	$P \wedge Q \vdash P \vee Q$	$\vee I2$ on (1)
(3)	$\vdash (P \wedge Q) \Rightarrow (P \vee Q)$	$\Rightarrow I$ on (2)

Which is slightly different because the λ-term is slightly different.

3.2 From Closed Tableaus to λ-terms

In this section we will describe an algorithm to convert closed tableaus into λ-terms of λP–. These λ-terms can easily be transformed into λ-terms of other PTSs, provided that these other PTS's are powerful enough. The closed tableau may be produced by any tableau-based theorem prover. This gives us the capability to use existing theorem provers as a module in an implementation of λP– and thereby adding powerful automated theorem proving to an interactive proof system, without the danger of extending our logic in an unforeseen way. If there is enough trust in the correctness of the implementation of the automatic theorem prover we can also use a special token to encode that the proof can be constructed using the ATP. We then do not have to actually convert the tableau and store the large λ-term that is the result of converting the tableau. The ATP can then reconstruct the tableau and convert it into a λ-term on request; for instance, if we want to communicate our proof to somebody using a different theorem prover based on λ-calculus.

The conversion is done in a structured way: for similar rules of the tableau method, similar conversion steps are performed. The classes of similar rules of the tableau method are usually called α-, β-, γ- and δ-rules. In figure 2.1 on page 10 each class is depicted in one row. The top row displays the special rule, which is very simple to convert. In figure 3.1 for each class the

structure of the rules is depicted. Our conversion algorithm will have one case for every class of rules.

$$special \quad \frac{\neg\neg P}{P}$$

$$\alpha \quad \frac{E(P,Q)}{E_1(P),\ E_2(Q)} \qquad \beta \quad \frac{E(P,Q)}{E_1(P) \mid E_2(Q)}$$

$$\gamma \quad \frac{E(U,P)}{E'(P)^x_\theta} \qquad \delta \quad \frac{E(U,P)}{E(U,P), E'(P)^x_t}$$

θ new variable of type U $\qquad$ t a term of type U

Figure 3.1: Structure of the different classes of tableau rules.

We will now show how to model each step of a tableau-proof of the formula P in λP–. We assume that we have the basic context Γ_L (see 2) to model the first order logic. In the conversion algorithm the labels of the tree correspond roughly to a context for λP–. The propositions in a label are used as types of assumptions in the context, but we will also have a few variables. To modify contexts of λP– we will intensively use theorem 2.4.5 and theorem 2.4.6 on page 22.

3.2.1 Converting the Initial Tableau

The tableau starts with a node labeled by $\neg P$ and the initial context for λP– will be $\Gamma_L, p : \neg P$. As explained in section 2.2 the tableau represents a contradiction derived from $\neg P$ and hence, converting the tableau should result in a contradiction $c : \bot$ derived from the context $\Gamma_L, p : \neg P$. The validity of P in λP– is then given by $\Gamma_L \vdash classic\ P\ (\lambda p : \neg P.c) : P$.

3.2.2 Converting Applications of Tableau Rules

Derivation of the contradiction is done recursively: first a contradiction is derived from the successor nodes and then a term is constructed for the current node. How this final construction of the contradiction is done depends on the tableau rule used to extend the node. We denote the context corresponding to the current node as $\Gamma_L, \Delta_1, x : X, \Delta_2$, where X is the proposition to which the tableau rule was applied. The context of the successor-node(s) will be stated for each case separately. For each type of node we will describe the construction of the contradiction.

Conversion for the Special Rule

Our first case will deal with the special tableau-rule:

$$\frac{\neg\neg P}{P}$$

We have to derive a contradiction c from a node with context $\Gamma_L, \Delta_1, o : \neg\neg P, \Delta_2$. For the successor-node we create the corresponding context $\Gamma_L, \Delta_1, \Delta_2, p : P$. By recursion, we derive a contradiction c from this successor node, hence we have $\Gamma_L, \Delta_1, \Delta_2, p : P \vdash c : \bot$. Then the contradiction we seek is derived as follows:

(0)	$\Gamma_L, \Delta_1, \Delta_2, p : P \vdash c : \bot$	induction hypothesis
(1)	$\Gamma_L, \Delta_1, \Delta_2 \vdash (\lambda p : P.c) : P \Rightarrow \bot$	Π-*intro* on (0)
(2)	$\Gamma_L, \Delta_1, o : (P \Rightarrow \bot) \Rightarrow \bot, \Delta_2 \vdash o : (P \Rightarrow \bot) \Rightarrow \bot$	see remark 3.2.1 and theorem 2.4.5
(3)	$\Gamma_L, \Delta_1, o : (P \Rightarrow \bot) \Rightarrow \bot, \Delta_2 \vdash \mathit{classic}\ P\ o : P$	*classic* on (2)
(4)	$\Gamma_L, \Delta_1, o : (P \Rightarrow \bot) \Rightarrow \bot, \Delta_2 \vdash (\lambda p : P.c) : P \Rightarrow \bot$	theorem 2.4.6 on (1)
(5)	$\Gamma_L, \Delta_1, o : (P \Rightarrow \bot) \Rightarrow \bot, \Delta_2 \vdash (\lambda p : P.c)\ (\mathit{classic}\ P\ o) : \bot$	Π-*elim* on (3) and (4)

Remark 3.2.1 Formally we also need to derive types in order to apply the PTS-rules. For instance in step (2), we indirectly apply intro on the type $(P \Rightarrow \bot) \Rightarrow \bot$ by using theorem 2.4.5, but for this we also need a type judgment saying $\Gamma_L, \Delta_1 \vdash (P \Rightarrow \bot) \Rightarrow \bot : *_p$. Such a type judgment can be derived by:

(*a*)	$\Gamma_L, \Delta_1 \vdash P : *_p$	Theorem 2.4.3 and $P \in *_p^L$
(*b*)	$\Gamma_L, \Delta_1, p : P \vdash \bot : *_p$	Axiom of λP$-$ and repeated weaken
(*c*)	$\Gamma_L, \Delta_1 \vdash P \Rightarrow \bot$	Π-*form* on (a) and (b)
(*d*)	$\Gamma_L, \Delta_1, p : (P \Rightarrow \bot) \vdash \bot : *_p$	Axiom of λP$-$ and repeated weaken
(*e*)	$\Gamma_L, \Delta_1 \vdash (P \Rightarrow \bot) \Rightarrow \bot : *_p$	Π-*form* on (a) and (b)

For reasons of space and simplicity, we will omit these type derivations. Usually it will be evident that the types are correct.

Conversion for α-rules

Before we present the general scheme to convert α-rules, we describe the conversion of the typical case of an α-rule: conjunction. The tableau rule is:

$$\frac{P \wedge Q}{P, Q}$$

We have to derive $\Gamma_L, \Delta_1, o : P \wedge Q, \Delta_2 \vdash ? : \bot$. To the successor node, we assign the context $\Gamma_L, \Delta_1, \Delta_2, p : P, q : Q$. By recursion, we get from this context a contradiction: $\Gamma_L, \Delta_1, \Delta_2, p : P, q : Q \vdash c : \bot$. To derive a contradiction from the original context we use the following derivation:

(0)	$\Gamma_L, \Delta_1, \Delta_2, p : P, q : Q \vdash c : \bot$	induction hypothesis
(1)	$\Gamma_L, \Delta_1, \Delta_2, p : P \vdash (\lambda q : Q.c) : Q \Rightarrow \bot$	Π-*intro* on (0)
(2)	$\Gamma_L, \Delta_1, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c)) : P \Rightarrow (Q \Rightarrow \bot)$	Π-*intro* on (1)
(3)	$\Gamma_L, \Delta_1, o : P \wedge Q, \Delta_2 \vdash o : P \wedge Q$	theorem 2.4.5
(4)	$\Gamma_L, \Delta_1, o : P \wedge Q, \Delta_2 \vdash \pi_1(o) : P$	$\wedge$-*elim*$_1$ on (3)
(5)	$\Gamma_L, \Delta_1, o : P \wedge Q, \Delta_2 \vdash \pi_2(o) : Q$	$\wedge$-*elim*$_2$ on (3)
(6)	$\Gamma_L, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c)) : P \Rightarrow (Q \Rightarrow \bot)$	theorem 2.4.6 on (2)
(7)	$\Gamma_L, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o) : Q \Rightarrow \bot$	Π-*elim* on (4) and (6)
(8)	$\Gamma_L, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o)\ \pi_2(o) : \bot$	Π-*elim* on (5) and (7)

Hence, the solution is given by $? := (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o)\ \pi_2(o)$.

In the general case we consider the tableau rule:

$$\frac{E(P, Q)}{E_1(P), E_2(Q)}$$

We have to derive a contradiction from the context $\Gamma_L, \Delta_1, o : E(P, Q), \Delta_2$. To the successornode we assign the context $\Gamma_L, \Delta_1, \Delta_2, p : E_1(P), q : E_2(Q)$ from which we get a contradiction $c : \bot$ by

recursion. In order to obtain a contradiction from the original context, we use a modified version of the scheme given above: Steps (0) till (2) remain unchanged, except that P has now become $E_1(P)$ and Q has become $E_2(Q)$. In step (3) we introduce $o : E(P,Q)$, but to continue with steps (4) till (8) we need $\Gamma_L, \Delta_1, o : E(P,Q), \Delta_2 \vdash ?' : E_1(P) \wedge E_2(Q)$. How this is accomplished depends on the actual rule that is applied. For every rule we can construct a derivation and hence a λ-term to fill in for $?'$. The derivation of the individual λ-terms is omitted here, but the results are given in table 3.1. In this table, the conversion function T gives for a term $o : E(P,Q)$ a term with type $E_1(P) \wedge E_2(Q)$. Since steps (4) till (8) are performed after using the conversion function T, the appearances of o in these steps become $T(o)$. Note that the conversion functions produce λ-terms and that they are not λ-terms themselves.

$E(P,Q)$	$E_1(P)$	$E_2(Q)$	$T(o) : E_1(P) \wedge E_2(Q)$ with $o : E(P,Q)$
$P \wedge Q$	P	Q	o
$\neg(P \Rightarrow Q)$	P	$\neg Q$	$(classic\ P\ (\lambda p : \neg P.o(\lambda q : P.p\ q\ Q)), (\lambda q : Q.o(\lambda p : P.q)))$
$\neg(P \vee Q)$	$\neg P$	$\neg Q$	$(\lambda p : P.o\ (injl\ (P \vee Q)\ p), \lambda q : Q.o\ (injr\ (P \vee Q)\ q))$

Table 3.1: Conversion functions for α-rules.

Conversion for β-rules

Again, we start with the typical case as an example. For β-rules the typical case is a disjunction, which has the tableau rule:

$$\frac{P \vee Q}{P \mid Q}$$

If the current context is $\Gamma_L, \Delta_1, o : P \vee Q, \Delta_2$ then its successors will have contexts $\Gamma_L, \Delta_1, \Delta_2, p : P$ and $\Gamma_L, \Delta_1, \Delta_2, q : Q$ respectively. From the successor contexts we have derived contradictions c_1 and c_2 by recursion. The derivation of a contradiction from the current context is then given by:

(0)	$\Gamma_L, \Delta_1, \Delta_2, p : P \vdash c_1 : \bot$	induction hypothesis
(1)	$\Gamma_L, \Delta_1, \Delta_2, q : Q \vdash c_2 : \bot$	induction hypothesis
(2)	$\Gamma_L, \Delta_1, \Delta_2 \vdash (\lambda p : P.c_1) : P \Rightarrow \bot$	Π-*intro* on (0)
(3)	$\Gamma_L, \Delta_1, \Delta_2 \vdash (\lambda q : Q.c_2) : Q \Rightarrow \bot$	Π-*intro* on (1)
(4)	$\Gamma_L, \Delta_1, \Delta_2 \vdash ((\lambda p : P.c_1) \nabla (\lambda q : Q.c_2)) : (P \vee Q) \Rightarrow \bot$	$\vee$-*elim* on (2) and (3)
(5)	$\Gamma_L, \Delta_1, o : P \vee Q, \Delta_2 \vdash o : P \vee Q$	theorem 2.4.5
(6)	$\Gamma_L, \Delta_1, o : P \vee Q, \Delta_2 \vdash ((\lambda p : P.c_1) \nabla (\lambda q : Q.c_2)) : (P \vee Q) \Rightarrow \bot$	theorem 2.4.6 on (4)
(7)	$\Gamma_L, \Delta_1, o : P \vee Q, \Delta_2 \vdash ((\lambda p : P.c_1) \nabla (\lambda q : Q.c_2))\ o : \bot$	Π-*elim* on (5) and (6)

To convert the general case we consider the tableau rule:

$$\frac{E(P,Q)}{E_1(P) \mid E_2(Q)}$$

We use the same strategy we used for α-rules: The derivation above is used as a scheme in which we have to replace P by $E_1(P)$ and Q by $E_2(Q)$ in lines (0) to (4). Instead of introducing $o : P \vee Q$ in line (5), we introduce $o : E(P,Q)$ and then insert a derivation between line (5) and line (6) that results in a λ-term of type $E_1(P) \vee E_2(Q)$. These λ-terms depend on o and can be obtained by applying a transformation function T to o. The transformation functions for β-rules are given in table 3.2 but their derivation is omitted. Again, the transformation functions T produce λ-terms but are not λ-terms themselves.

The remainder of the general case (the new lines (6) and (7)) then follows easily.

$E(P,Q)$	$E_1(P)$	$E_2(Q)$	$T(o) : E_1(P) \vee E_2(Q)$ with $o : E(P,Q)$
$\neg(P \wedge Q)$	$\neg P$	$\neg Q$	$classic\ (\neg P \vee \neg Q)\ \lambda r : \neg(\neg P \vee \neg Q).$ $\quad r(injl\ (\neg P \vee \neg Q)\ (\lambda p : P.r(injr\ (\neg P \vee \neg Q)\ (\lambda q : Q.o(p,q)))))$
$P \Rightarrow Q$	$\neg P$	Q	$classic\ (\neg P \vee Q)\ \lambda r : \neg(\neg P \vee Q).$ $\quad r(injl\ (\neg P \vee Q)\ (\lambda p : P.r(injr\ (\neg P \vee Q)\ (o\ p))))$
$P \vee Q$	P	Q	o

Table 3.2: Conversion functions for β-rules.

Conversion for γ-rules

The typical case for a γ-rule is existential quantification, with the tableau rule:

$$\frac{\exists x : U.P}{P^x_\alpha}$$

The current context is $\Gamma_L, \Delta_1, o : (\exists x : U.P), \Delta_2$. For γ-rule we have to extend the context more than for the other cases: we do not only add $p : P$ to the successor's context, but also a fresh variable $\theta : U$. The successor's context then reads $\Gamma_L, \Delta_1, \Delta_2, \theta : U, p : P$. By recursion we have derived a contradiction c from this context. A contradiction from the current context is derived as follows:

(0)	$\Gamma_L, \Delta_1, \Delta_2, \theta : U, p : P \vdash c : \bot$	induction hypothesis
(1)	$\Gamma_L, \Delta_1, \Delta_2, \theta : U \vdash (\lambda p : P.c) : P \Rightarrow \bot$	Π-*intro* on (0)
(2)	$\Gamma_L, \Delta_1, \Delta_2 \vdash (\lambda\theta : U.(\lambda p : P.c)) : (\forall\theta : U.(P \Rightarrow \bot))$	Π-*intro* on (1)
(3)	$\Gamma_L, \Delta_1, \Delta_2 \vdash \Diamond\ (\exists x : U.P)\ (\lambda\theta : U.(\lambda p : P.c)) : (\exists x : U.P) \Rightarrow \bot$	$\exists$-*elim* on (2) for $\bot$
(4)	$\Gamma_L, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash o : (\exists x : U.P)$	theorem 2.4.5
(5)	$\Gamma_L, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash \Diamond\ (\exists x : U.P)\ (\lambda\theta : U.(\lambda p : P.c))$ $: (\exists x : U.P) \Rightarrow \bot$	theorem 2.4.6 on (3)
(6)	$\Gamma_L, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash (\Diamond\ (\exists x : U.P)\ (\lambda\theta : U.(\lambda p : P.c)))\ o : \bot$	Π-*elim* on (4) and (5)

Like before, we use the above derivation to obtain a scheme for the general case. The tableau rule is:

$$\frac{E(U,P)}{E'(P)^x_\theta}$$

First, we replace P in the derivation above by $E'(P)^x_\theta$ in lines (0) to (2). In line (3), P is replaced by just $E'(P)$, which is allowed, since the occurrences of x in P that were bound within $E(U,P)$ are now explicitly bound by the $\exists x : U \ldots$ occurring before $E'(P)$. Next, we change the *intro* in line (4) to an introduction of $o : E(U,P)$. Finally, we insert a derivation of a λ-term of type $(\exists x : U.E'(P))$ between line (4) and line (5). Also like before, these λ-terms are given by a transformation function T. The transformation functions for γ-rules are given in table 3.3.

$E(U,P)$	$E'(P)$	$T(o) : (\exists x : U.E'(P))$ with $o : E(U,P)$
$(\exists x : U.P)$	P	o
$\neg(\forall x : U.P)$	$\neg P$	$classic\ (\exists x : U.\neg P)\ (\lambda r : \neg(\exists x : U.\neg P).$ $\quad o(\lambda x : U.classic\ P\ (\lambda p : \neg P.r(inj\ (\exists x : U.\neg P)\ p\ x))))$

Table 3.3: Conversion functions for γ-rules.

Conversion for δ-rules

In case of δ-rules the most typical example is the rule for universal quantification, with tableau rule:

$$\frac{\forall x : U.P}{\forall x : U.P,\ P_t^x}$$

Given the current context $\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2$ and the term t used to extend the tableau, we construct for the successor node the context $\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2, p : P_t^x$. Note that the original universal quantifier is still present in this context. After the contradiction c has been derived from the successor's context by recursion we derive a contradiction from the original context as follows:

(0)	$\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2, p : P_t^x \vdash c : \bot$	induction hypothesis
(1)	$\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash (\lambda p : P_t^x.c) : P_t^x \Rightarrow \bot$	Π-*intro* on (0)
(2)	$\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash o : (\forall x : U.P)$	theorem 2.4.5
(3)	$\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash t : U$	ok because of theorem 2.4.2 on page 22
(4)	$\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash o\ t : P_t^x$	Π-*elim* on (2) and (3)
(5)	$\Gamma_L, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash (\lambda p : P_t^x.c)\ (o\ t) : \bot$	Π-*elim* on (1) and (4)

To make this derivation suitable for the general case, consider the rule:

$$\frac{E(U,P)}{E(U,P), E'(P)_t^x}$$

We replace $(\forall x : U.P)$ by $E(U,P)$ and P_t^x by $E'(P)_t^x$ in the entire derivation. We then have to insert a derivation of a term of type $(\forall x : U.E'(P))$ from $o : E(U,P)$ after line (2). The resulting λ-term of this derivation is given by the transformation functions T given in table 3.4.

$E(U,P)$	$E'(P)$	$T(o) : (\forall x : U.E'(P))$ with $o : E(U,P)$
$(\forall x : U.P)$	P	o
$\neg(\exists x : U.P)$	$\neg P$	$(\lambda x : U.(\lambda p : P.o\ (inj\ (\exists x : U.P)\ p\ x)))$

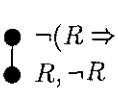
Table 3.4: Conversion functions for δ-rules.

3.2.3 Conversion of Closed Leafs

Recursion ends when we convert a leaf of the tableau. At a leaf we cannot use a contradiction derived from successor-nodes, since there are no successor-nodes. However, at a closed leaf we have a context in which both a variable of type P and a variable of type $\neg P$ occur. In λP– negation is modeled by implication and $\bot$. Hence, in the context $\Gamma_L, \Delta_1, p : P, \Delta_2, p' : \neg P, \Delta_3$ we can derive the contradiction $p'p$.

This concludes the conversion algorithm. Note that if during the construction of a tableau needless steps are taken these will also be translated.

The converted proof may be much longer than a proof that is constructed directly in λP–. For example: a direct proof of $R \Rightarrow R$ in λP– looks like $\Gamma_L \vdash (\lambda p : R.p) : R \Rightarrow R$. However, if we convert the tableau



we get a much larger λ-term. Following the algorithm, we start with $\Gamma_L \vdash classic\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).c) : R \Rightarrow R$, where c is a contradiction extracted from the initial context $\Gamma_L, o : \neg(R \Rightarrow R)$. The tableau rule applied is an α-rule for implication. The resulting λ-term of this conversion in general is $(\lambda p : E_1(P).(\lambda q : E_2(Q).c'))\ \pi_1(T(o))\ \pi_2(T(o)) : \bot$, in which c' is the contradiction derived from the successor's context $\Gamma_L, p : E_1(P), q : E_2(Q)$. If we fill in P, Q, E_1, E_2 and T for our example and then use the result of this substitution in our proof, we get

$$\begin{aligned}
&classic\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).(\lambda p : R.\lambda q : \neg R.c')\\
&\pi_1(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))\\
&\pi_2(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))) : R \Rightarrow R
\end{aligned}$$

and c' is the contradiction derived from the context $\Gamma_L, p : R, q : \neg R$. This corresponds to the context in which the tableau gets closed by R and $\neg R$, hence the algorithm gives us $c' \equiv qp$. The final proof then reads:

$$\begin{aligned}
\Gamma_L \vdash\ &classic\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).(\lambda p : R.\lambda q : \neg R.qp)\\
&\pi_1(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))\\
&\pi_2(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))) : R \Rightarrow R
\end{aligned}$$

This 'explosion' of the proof term is certainly a drawback of this proof method. However, we do not need to really convert each proof. We can use a short representation in a λ-term to indicate that the required term can be found with the tableau prover built in the system. We can then construct the λ-term on request, by reconstructing the tableau and then convert it according to the method we described.

Chapter 4

Conclusions

4.1 Advantages

In this document we presented a pure type system $\lambda P-$ that closely models first order logic. We compared it to natural deduction and tableau methods and showed that these can be performed within $\lambda P-$. These results offer the possibility to construct an interactive proof system based on pure type systems with an in-built automated theorem prover for first order logic. This has the following advantages:

- Pure type systems offer a compact, yet powerful way to prove theorems.
- Due to its parameter mechanism $\lambda P-$ does not contain the *conversion*-rule of a regular PTS. This makes designing the unification algorithm, and hence the type checker simpler and the implementation faster.
- Proofs are encoded with λ-terms and hence, can be type-checked for correctness by a simple program. This increases the reliability of the system.
- λ-terms can be communicated to other theorem provers based on Pure Type Systems like Coq, Lego or HOL. Verifying the constructed proofs on several systems increases the reliability of the system even more.
- Providing an automated theorem prover based on tableau methods offers a high degree of automation that is usually not present in interactive theorem provers (see [Fra97]). Converting the tableau into a λ-term offers a safe verification method of the proof. Detecting errors in a large tableau based theorem prover becomes easier.

4.2 Disadvantages

A drawback of the method of tableau conversion is the 'explosion' of the λ-term. The proof obtained in this manner is not suitable for a human reader. However, we plan to use the theorem prover in an environment for proving program correctness. In such an environment the user of the system is a programmer who will only be interested in whether or not a theorem is correct. Usually programmers do not want to read every detail of their proof. Also, it is possible not to convert the proof at all, but add an axiom to the derivation system that looks like:

$$\frac{\Gamma \vdash P : *_p \quad \text{'We have a closed tableau for } P\text{'}}{\Gamma \vdash tab\ P : P} \tag{4.1}$$

,where tab is just an extension of the λ-term syntax to encode that P was proved by a tableaux-based theorem prover. This axioms allows us to use the theorem prover without getting extremely large λ-terms. Since we can reconstruct the tableau and convert it to a λ-term whenever we want to,, we can still communicate our proofs to other theorem provers.

Another drawback is that λP– is only suitable for first order logic, not higher order logic. If we use a pure type system for higher order logic, e.g. the calculus of constructions, we cannot apply the automated theorem prover to every propositional formula we can derive in the pure type system.

4.3 What about resolution?

We also presented resolution methods, which like the previous systems is sound and complete with respect to the semantics of first order logic. Therefore it is also possible to use an axiom like axiom 4.1 if we want to include a resolution based theorem prover in our system. However, we then no longer can communicate our proofs to other systems, since we do not have a method to convert a resolution proof to a λ-term.

Our attempts to find a conversion algorithm for resolution proofs have not succeeded. If one attempts to simulate the steps of a resolution proof directly in λP–, then problems arise during conversion of the resolution proof after the formula has been skolemized. Suppose we want to prove $\Gamma_L \vdash X : *_p$ by simulating a given resolution proof of X. First, we use the *classic* rule and simulate all the rewrite steps that lead to the head normal form (HNF) of $\neg X$. Second, we simulate the rewrite steps that convert the body of the HNF of $\neg X$ into conjunctive normal form, e.g. we get something like:

$$o : \forall x : U.\exists s : U.((Px \vee Qc_3 \vee Rs) \wedge (Sx \vee \neg Qc_3 \vee Rs) \wedge (\neg Pc_1) \wedge (\neg Sc_2) \wedge (\neg Rx))$$

,where c_i denote constants. However, we cannot simulate skolemization. Simulating skolemization requires the introduction of fresh functions to the context, which in λP– are parametric terms. We could then never get rid of these functions anymore, since we cannot make λ-abstractions over parametric terms. Therefore we could never transform a proof of the skolemized formula into a proof derivable from the context Γ_L. Hence, something else is needed here.

Simulating the applications of the cut-rules in a resolution proof is now not possible in general, since in the resolution proof the order of applying these cut-rules causes problems during simulation: Several values may be substituted for a free variable in the same clause in several resolution steps. The results obtained by these successive steps can then be combined again later in the proof with yet another substitution. For instance, consider the following resolution proof of our example:

(0)	$Px \vee Qc_3 \vee Rs(x)$	
(1)	$Sx \vee \neg Qc_3 \vee Rs(x)$	
(2)	$\neg Pc_1$	
(3)	$\neg Sc_2$	
(4)	$\neg Rx$	
(5)	$Qc_3 \vee Rs(c_1)$	[derived from (0) and (2), substitution $x := c_1$]
(6)	$\neg Qc_3 \vee Rs(c_2)$	[derived from (1) and (3), substitution $x := c_2$]
(7)	$Rs(c_1) \vee Rs(c_2)$	[derived from (5) and (6), no substitution]
(8)	$Rs(c_2)$	[derived from (4) and (7), substitution $x := s(c_1)$]
(9)	$\square$	[derived from (4) and (8), substitution $x := s(c_2)$]

In clauses (0) till (4) the skolem function $s(x)$ refers to the same existential binder (see the representation of the initial clauses in λP- given before)

However, in clauses (5) and (6) we have $s(c_1)$ and $s(c_2)$ respectively, which refer to different existential quantifiers, since they are obtained by applications of o to c_1 and c_2 respectively:

$$o\ c_1 : \exists s : U.((Pc_1 \vee Qc_3 \vee Rs) \wedge (Sc_1 \vee \neg Qc_3 \vee Rs) \wedge (\neg Pc_1) \wedge (\neg Sc_2) \wedge (\neg Rc_1))$$
$$o\ c_2 : \exists s : U.((Pc_2 \vee Qc_3 \vee Rs) \wedge (Sc_2 \vee \neg Qc_3 \vee Rs) \wedge (\neg Pc_1) \wedge (\neg Sc_2) \wedge (\neg Rc_2))$$

This obscures the conversion, since the partial results contain skolem functions, which in the λ-terms are explicitly existentially quantified. The bodies of these quantifications then have to be combined to obtain a new (partial) result (see derived clause (7)). I.e. we need the following rule that is not true in general (the existential quantifiers refer to the same skolem-function in the resolution proof):

$$\frac{\Gamma \vdash (\exists x : U.(P \vee Q)) \quad \Gamma \vdash (\exists x : U.(\neg P \vee R))}{\Gamma \vdash (\exists x : U.(Q \vee R))}$$

Individual proofs always seem to be convertible by using a different order of eliminating conjuncts from the clauses, but we have failed to find a general algorithm that does the job.

Bibliography

[Bar92] H.P. Barendregt. *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, chapter 2, pages 118–310. Oxford Science Publications, 1992.

[dN95] Hans de Nivelle. *Ordering Refinements of Resolution.* PhD thesis, Delft University of Technology, 1995.

[dS93] H.C.M. de Swart. *LOGIC; Mathematics, Language, Computer Science and Philosophy*, volume I. Peter Lang, Frankfurt, 1993.

[dS94] H.C.M. de Swart. *LOGIC; Logic and Computer Science*, volume II. Peter Lang, Frankfurt, 1994.

[Fra97] Michael Franssen. Tools for the construction of correct programs: an overview. Technical Report Report 97-06, Eindhoven University of Technology, 1997.

[Laa97] T. Laan. *The Evolution of Type Theory in Logic and Mathematics.* PhD thesis, Eindhoven University of Technology, 1997.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2), April 1982.

[Smu68] Raymond M. Smullyan. *First-Order Logic.* Ergebnisse der Mathematik und Ihrer Grenzgebiete, Band 43. Springer, 1968.

[SO94] H.C.M. de Swart and W.M.J. Ophelders. Tableaux, resolution, and complexity of formulas. *Methods of Logic in Computer Science*, 1:241–260, 1994.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The I^2C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.
96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
96/21	L. Bijlsma and R. Nederpelt	Predicate calculus: concepts and misconceptions, p. 26.
96/22	M.C.A. van de Graaf and G.J. Houben	Designing Effective Workflow Management Processes, p. 22.
96/23	W.M.P. van der Aalst	Structural Characterizations of sound workflow nets, p. 22.
96/24	M. Voorhoeve and W. van der Aalst	Conservative Adaption of Workflow, p.22
96/25	M. Vaccari and R.C. Backhouse	Deriving a systolic regular language recognizer, p. 28
97/01	B. Knaack and R. Gerth	A Discretisation Method for Asynchronous Timed Systems.
97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.

97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.
97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.
97/12	D. Hauschildt, E. Verbeek and W. van der Aalst	WOFLAN: A Petri-net-based Workflow Analyzer, p. 30.
97/13	W.M.P. van der Aalst	Exploring the Process Dimension of Workflow Management, p. 56.
97/14	J.F. Groote, F. Monin and J. Springintveld	A computer checked algebraic verification of a distributed summation algorithm, p. 28