Embedding First-Order Tableaux into a Pure Type System

Michael Franssen

Co-operation Centre Tilburg and Eindhoven Universities
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
E-mail mfranssen@usa.net

Abstract

We consider Pure Type Systems (PTSs) extended with a mechanism for parametric terms. In this paper we introduce a PTS called $\lambda P-$ utilizing this extension. $\lambda P-$ exactly corresponds to first-order predicate logic, unlike the usual embedding of this logic in PTSs. Next, we show how tableaux-based proofs can, in a structured way, be converted into λ-terms representing proofs in $\lambda P-$. The result is an interactive theorem prover combined with powerful tableaux-based automatic proof construction.

Key words: Conversion Algorithms, First-Order Predicate Logic, Pure Type Systems, Tableaux

1 Introduction

The proof system $\lambda P-$ and the conversion algorithm presented in this paper form the basis for a proof construction system intended for use within a programming tool. This programming tool, the final goal of our project, must support the *derivation* of correct programs in a Dijkstra/Hoare like calculus [1]. Deriving programs in such a calculus involves proving many logical formulas (proof obligations), hence the quest for correct programs becomes a quest for correct proofs.

The quest for correct proofs has been addressed (among others) by de Bruijn by creating systems that can mechanically check the validity of proofs: the Automath systems [11]. Later, many systems have been classified in a framework called Pure Type Systems by Berardi [3] and Terlouw [12]. The connection between these systems and various logics is described by Geuvers in [7].

Modern implementations of such systems do not only verify proofs, but also

help to interactively construct proofs, e.g. Coq [5], LEGO [9], Yarrow [13]. However, many of the proof obligations are simple and hence, their proofs should be constructed automatically to allow the programmer to concentrate on the more difficult ones. Unfortunately, automatic proof construction is not easy in these systems [6].

Aiming at a higher degree of automatic proof construction, we looked at both tableaux and resolution methods as well as algebraic systems. Tableaux seemed to be the natural choice, since

- They support full first order predicate logic, unlike algebraic proof systems.
- They follow the syntax of the formula, unlike resolution, hence it is conceivable to construct a natural deduction proof from a closed tableau.

To easily support interactive theorem proving, we embed tableaux based theorem provers in a PTS. The system $\lambda P-$ and the conversion algorithm form exactly this: an interactive theorem prover in which automatically generated proofs can be safely used.

In section 2 we present the framework of Pure Type Systems and an extension of this framework by T.Laan [8]. Next, we use this extension to construct the system $\lambda P-$ which is the formalism of the interactive theorem prover. We argue that our system $\lambda P-$ corresponds exactly to first-order logic, unlike other type systems known from the literature. In section 3 an algorithm is presented that translates in a structured way tableau based proofs into λ-terms of our system $\lambda P-$. This allows us to use in $\lambda P-$ automatically generated proofs from a tableau-based theorem prover. The algorithm is formalized in section 4 and properties of the converted proofs are described in 5.

2 Pure Type Systems

The framework of Pure Type Systems (PTSs) is a systematic description of many typed λ-calculi found in the literature. Due to the Curry-Howard-deBruijn isomorphism between propositions and types, many of these λ-calculi can also be used to represent propositions and their proofs. Also, this isomorphism makes PTSs very suitable for interactive theorem proving, since manipulating proofs corresponds to manipulating syntactical terms in a PTS. This has led to systems like Coq, LEGO, Yarrow etc.

In this section we first give the definitions of PTSs and demonstrate the propositions-as-types isomorphism by an example. Then we introduce an extension of the PTS-framework as described in [8]. In this section we use the idea of extending PTSs with parameters to construct the system $\lambda P-$ that corresponds exactly to many-sorted first order logic.

2.1 The definition of PTSs

A PTS is specified by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ of sets, where $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The elements of $\mathcal{S}$ are called sorts, the elements of $\mathcal{A}$ are called axioms and the set $\mathcal{R}$ contains (Π-formation) rules. Given a specification of a PTS, say $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, the terms, contexts and type judgment relation of the PTS are defined as follows:

Definition 2.1 (Terms) *Given a set V of variables, the set T of PTS terms is defined by the following abstract syntax:*

$$T ::= \mathcal{S} \mid V \mid \lambda V : T.T \mid \Pi V : T.T \mid TT$$

Definition 2.2 (Contexts) *A context is a list of the form $x_1 : A_1, \ldots, x_n : A_n$, where $x_i \in V$ and A_i are terms as defined in 2.1 for $i = 1, \ldots, n$. The empty context is denoted as $<>$. By convention we use $\Gamma, \Delta, \ldots$ as meta variables for contexts. If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a context and $v \in V$, then v is called Γ-fresh if $v \notin \{x_1, \ldots, x_n\}$.*

Definition 2.3 (Type judgment relation) *The type judgment relation describes the actual PTS. A judgment always has the form $\Gamma \vdash A : B$, where A and B are terms and Γ is a context. $\Gamma \vdash A : B$ should be read as: 'A has type B in context Γ'. The type judgment relation $\vdash$ is defined by the rules in figure 1. We give a brief description of each rule.*

start *This is the only rule without premises in a PTS. It supplies, starting from the axioms in $\mathcal{A}$, basic typing judgments from which all the other typing judgments 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 derivations 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 Π-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* Π*-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 *States that we don't distinguish* β*-equal types. In several PTSs 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 (we will come back to this in our example below) 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.*

With this definition of PTSs, we are ready to demonstrate the propositions as types isomorphism. We consider the PTS for first order predicate logic as proposed by Berardi (presented in definition 5.4.5 of [2]). The specification is:

$$\begin{aligned}
\mathcal{S} &= \{*_s, *_p, *_f, \Box_s, \Box_p\} \\
\mathcal{A} &= \{(*_s, \Box_s), (*_p, \Box_p)\} \\
\mathcal{R} &= \{(*_s, *_s, *_f), (*_s, *_f, *_f), (*_s, \Box_p, \Box_p), (*_p, *_p, *_p), (*_s, *_p, *_p)\}
\end{aligned}$$

The sorts $\mathcal{S}$ have the following intended meaning: Terms of type $*_s$ correspond to sets of multi-sorted first order logic. Terms of type $*_f$ are themselves types of functions. Terms of type $*_p$ represent propositional formulas. Terms of type $\Box_p$ represent types of predicates. Note that since $(*_p, \Box_p) \in \mathcal{A}$, propositional formulas are predicates with arity 0.

We introduce the following shorthand: $\Gamma \vdash A : B : s$, with $s \in \mathcal{S}$ denotes that $\Gamma \vdash B : s$ and $\Gamma \vdash A : B$. Then, if we have $\Gamma \vdash A : B : *_s$, A corresponds to a term with a value in the set B. If we have $\Gamma \vdash A : B : *_f$ then A is a function with type B. If $\Gamma \vdash A : B : *_p$ then A is a proof of the propositional formula B. If $\Gamma \vdash A : B : \Box_p$ then A is a predicate.

To make these correspondences more visible, we will use different notations for various Π-types. A term $(\Pi x : A.B)$ formed by Π-*form* with $(*_p, *_p, *_p) \in \mathcal{R}$ is denoted as $A \Rightarrow B$. A term $(\Pi x : A.B)$ formed with rule $(*_s, *_p, *_p)$ is denoted as $(\forall x : A.B)$. A proof of $(\forall x : U.(Px \Rightarrow Px))$ can then be derived as depicted in figure 2.

$$\textit{start} \qquad <> \vdash s1 : s2 \qquad (s1, s2) \in \mathcal{A}$$

$$\textit{intro} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad x \text{ is } \Gamma\text{-fresh}$$

$$\textit{weaken} \qquad \frac{\Gamma \vdash A : B \;\; \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B} \qquad x \text{ is } \Gamma\text{-fresh}$$

$$\Pi\textit{-form} \qquad \frac{\Gamma \vdash A : s1 \;\; \Gamma, x{:}A \vdash B : s2}{\Gamma \vdash (\Pi x{:}A.\ B) : s3} \qquad (s1, s2, s3) \in \mathcal{R}$$

$$\Pi\textit{-intro} \qquad \frac{\Gamma, x{:}A \vdash b : B \;\; \Gamma \vdash B : s}{\Gamma \vdash (\lambda x{:}A.\ b) : (\Pi x{:}A.\ B)}$$

$$\Pi\textit{-elim} \qquad \frac{\Gamma \vdash F : (\Pi x{:}A.\ B) \;\; \Gamma \vdash a : A}{\Gamma \vdash \ Fa : B[x := a]}$$

$$\textit{conversion} \ \frac{\Gamma \vdash A : B \;\; \Gamma \vdash B' : s \;\; B =_\beta B'}{\Gamma \vdash A : B'}$$

Fig. 1. The type judgment derivation rules of a PTS.

The example above is quite dull, but already requires a derivation of 17 lines. Equally, we could derive the type judgement:

$$\begin{aligned} U : *_s, P : (\Pi x : U.*_p), Q : (\Pi x : U.*_p) \vdash \\ (\lambda p : (\forall x : U.(Px \Rightarrow Qx)).(\lambda q : (\forall x : U.Px).(\lambda x : U.px(qx)))) \\ : (\forall x : U.(Px \Rightarrow Qx)) \Rightarrow ((\forall x : U.Px) \Rightarrow (\forall x : U.Qx)) \end{aligned}$$

However, then the derivation becomes no less than 47 lines. The reason for this is the necessity to derive type correctness judgements (Π-*form*) and the step-by-step usage of weakening (see lines 3,4,5).

In λ-calculus the proof object $(\lambda x : U.(\lambda p : Px.p))$ is a function that returns for each element x of U a proof of $Px \Rightarrow Px$. This proof is given by $(\lambda p : Px.p)$,

1 $<>$	$\vdash *_s : \Box_s$	(*start*)
2 $U : *_s$	$\vdash U : *_s$	(*intro* on 1)
3 $<>$	$\vdash *_p : \Box_p$	(*start*)
4 $U : *_s$	$\vdash *_p : \Box_p$	(*weaken* on 1,3)
5 $U : *_s, x : U$	$\vdash *_p : \Box_p$	(*weaken* on 2,4)
6 $U : *_s$	$\vdash (\Pi x : U.*_p) : \Box_p$	(Π-*form* on 2,5)
7 $U : *_s, P : (\Pi x : U.*_p)$	$\vdash P : (\Pi x : U.*_p)$	(*intro* on 6)
8 $U : *_s, P : (\Pi x : U.*_p)$	$\vdash U : *_s$	(*weaken* on 2,6)
9 $U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash P : (\Pi x : U.*_p)$	(*weaken* on 7,8)
10 $U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash x : U$	(*intro* on 8)
11 $U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash Px : *_p$	(Π-*elim* on 9,10)
12 $U : *_s, P : (\Pi x : U.*_p), x : U, p : Px$	$\vdash Px : *_p$	(*weaken* on 11,11)
13 $U : *_s, P : (\Pi x : U.*_p), x : U, p : Px$	$\vdash p : Px$	(*intro* on 11)
14 $U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash Px \Rightarrow Px : *_p$	(Π-*form* on 11,12)
15 $U : *_s, P : (\Pi x : U.*_p), x : U$	$\vdash (\lambda p : Px.p) : Px \Rightarrow Px$	(Π-*intro* on 13,14)
16 $U : *_s, P : (\Pi x : U.*_p)$	$\vdash (\forall x : U.(Px \Rightarrow Px)) : *_p$	(Π-*form* on 8,14)
17 $U : *_s, P : (\Pi x : U.*_p)$	$\vdash (\lambda x : U.(\lambda p : Px.p)) : (\forall x : U.(Px \Rightarrow Px))$	(Π-*intro* on 15,16)

Fig. 2. A sample derivation in a Pure Type System.

which in turn is a function that given a proof of Px returns a proof of Px (the identity function). Intuitively the existence of such a function is indeed a proof of $(\forall x : U.(Px \Rightarrow Px))$. Given a context Γ, a proof term p and a proposition P, an entire derivation of $\Gamma \vdash p : P$ can be automatically constructed (see e.g. [4]), hence the proof-term represents the entire proof. This has two advantages:

(i) Even if a large and complex proof system is used, correctness of the proofs is assured by type-checking. This algorithm is relatively simple and can be proved to be correct.

(ii) Communicating proofs corresponds to communicating a syntactical proof term. This proof term can then be checked by another proof system based on λ-calculus.

Note that the set U and the unary predicate P occur explicitly in the context of the PTS. Hence, the functions and predicates of the logic can be modeled by elements in the context and do not need to be defined beforehand. This allows flexible logics to be handled by proof systems based on PTSs.

However, this PTS does not model first order logic exactly. There are a few differences that are not always desirable:

(i) Constants, like the natural number 0, are modeled in a context by $\Gamma_1, \mathbb{N} : *_s, 0 : \mathbb{N}, \Gamma_2$. Therefore they are indistinguishable from ordinary variables like the $x : U$ in line 9 of our example derivation.

(ii) Functions themselves have types. More precisely, a binary function f with arguments from sets A and B yielding a value from C has type $(\Pi x : A.(\Pi y : B.C))$. If f is applied to an argument $a : A$ then fa has type $(\Pi y : B.C)$, while in first order logic f applied to just one argument does not have a meaning at all. The same holds for predicates.

(iii) A single proposition corresponds to several types. For instance: in context $U : *_s, P : *_p, a : U$ the term P represents a predicate of the logic with arity 0, but in this context the same predicate is represented by $(\lambda x : U.P)a$. This is why the rule *conversion* is needed: a proof $p : P$ should also be a proof of $(\lambda x : U.P)a$, since it represents the same proposition. The problem appears to be caused by the rule $(*_s, \Box_p, \Box_p)$, which allows the creation of such λ-terms. This rule is absolutely necessary, however, to construct types of predicates of arities larger than zero.

2.2 Extending PTSs with parametric constants

The awkward properties of the PTS for first order logic given in the previous part of this section can be avoided by using an extension of the PTS definition described in [8]. The extension introduces parametric constants added to the terms of a PTS. A parametric constant is kept in the context and can only be used if all the required parameters are supplied at once. This corresponds to the way predicates and functions are used in first order logic. PTSs extended with parametric constants are called PPTSs.

A PPTS is specified by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$, where $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ are the sorts, axioms and rules of a regular PTS and $\mathcal{P}$ is a subset of $\mathcal{S} \times \mathcal{S}$. $\mathcal{P}$ is called the set of parametric rules.

Definition 2.4 (Parametric Terms) *Given a set V of variables and a set C of constants, the set $T_{\mathcal{P}}$ of PPTS terms is defined by the following abstract*

syntax:

$$T_{\mathcal{P}} ::= \mathcal{S} \mid V \mid \lambda V : T_{\mathcal{P}}.T_{\mathcal{P}} \mid \Pi V : T_{\mathcal{P}}.T_{\mathcal{P}} \mid T_{\mathcal{P}}T_{\mathcal{P}} \mid C(L_{\mathcal{P}})$$
$$L_{\mathcal{P}} ::= \varepsilon \mid < L_{\mathcal{P}}, T_{\mathcal{P}} >$$

The lists of terms produced by $L_{\mathcal{P}}$ are usually denoted as $< A_1, \ldots, A_n >$ or $A_1, \ldots, A_n$ instead of $< \ldots << \varepsilon, A_1 >, A_2 > \ldots A_n >$.

Definition 2.5 (Contexts of PPTSs) *A context is a list of the form $x_1 : A_1, \ldots, x_n : A_n$, such that every A_i is a term as defined in definition 2.4 and either $x_i \in V$ or x_i has the form $c(y_1 : B_1, \ldots, y_m : B_m)$, where $c \in C$, $y_1, \ldots, y_m \in V$ and $B_1, \ldots, B_m$ are terms as defined in definition 2.4. A constant c is called Γ-fresh if it does not occur in Γ.*

Definition 2.6 (Type judgment relation of a PPTS) *The type judgment relation of a PPTS uses all rules of a regular PTS (see figure 1) and two additional rules to make use of parametric constants. Let Δ denote $x_1 : B_1, \ldots, x_n : B_n$ and Δ_i denote $x_1 : B_1, \ldots, x_{i-1} : B_{i-1}$. Then the additional rules are:*

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

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

We give a brief description of the additional rules:

C-weaken *The C-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.*

C-application *Since a parametric constant itself cannot be typed in a PPTS it cannot be used with the usual* Π-elim *(sometimes called* application*) rule. The rule C-application allows us to use a parametric constant, but only if we supply all the required arguments at once. This corresponds to the use of 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.*

2.3 λP−: A PPTS for First Order Logic

We are now ready to introduce the system $\lambda P-$. $\lambda P-$ is a PPTS that exactly models many sorted first order predicate logic.

Definition 2.7 ($\lambda P-$) *$\lambda P-$ is the PPTS specified by:*

$$\begin{aligned}\mathcal{S} &= \{*_s, *_p, \Box_s, \Box_p\}\\ \mathcal{A} &= \{(*_s, \Box_s), (*_p, \Box_p)\}\\ \mathcal{R} &= \{(*_p, *_p, *_p), (*_s, *_p, *_p)\}\\ \mathcal{P} &= \{(*_s, *_s), (*_s, \Box_p)\}\end{aligned}$$

Note that the sort $*_f$, used by Berardi to model function types, is not present in $\lambda P-$. Also, the only rules in $\lambda P-$ are those corresponding to implication and universal quantification.

Functions and predicates are now added to the context by using the rule *C-weaken*, using parametric rule $(*_s, *_s)$ for functions and $(*_s, \Box_p)$ for predicates. A function or a predicate can only be used to form a proposition using the rule *C-application*. For instance, a function of arity 2 can only be used when it is applied to 2 arguments at once.

Essentially the propositions-as-types isomorphism and the intended meanings of the sorts of this system are equal to those of the regular PTS of Berardi. However, $\lambda P-$ corresponds more closely to first order logic:

(i) Constants are now modeled by a parametric constant with zero parameters. The natural number 0 is then modeled in a context as $\Gamma_1, \mathbb{N} : *_s, 0() : \mathbb{N}, \Gamma_2$. Since the 0 is now a constant from C, it cannot be confused with a parameter from V, since it is not possible to build a term like $(\lambda 0() : \mathbb{N}.X)$.

(ii) Functions themselves do not have types. A binary function f with arguments from sets A and B yielding a value from C occurs in the context as $f(x : A, y : B) : C$. Since f is a parametric constant with 2 arguments it cannot be applied to a single argument $a : A$. The same holds for predicates.

(iii) A single proposition corresponds to a single type. The rule $(*_s, \Box_p, \Box_p)$ allowing the typing of lambda terms representing predicates is no longer available. Therefore, a predicate P is no longer represented by $(\lambda x : U.P)a$, where U corresponds to a set of first order logic and $a : U$. The rule *conversion* is no longer needed, allowing a simpler and faster implementation.

Proofs of these properties are given by T. Laan and the author of this paper in [10].

So far, we were considering *minimal* first order logic with only implication and universal quantification. To model negation, conjunction, disjunction and existential quantification we would need a more powerful PTS allowing higher order constructs. However, this would destroy our close correspondence with first order logic. Another possibility is to further extend the abstract syntax of λ-terms and adding more rules to the type judgment relation. These extended λ-terms can then easily be translated into regular λ-terms of a PTS allowing higher order logic. However, our proof system itself then keeps its close correspondence to first order logic. The required extensions are given in appendix A.

Except for the extensions for propositional constructs, we also need a context containing the set-, function- and predicate symbols of the logic. This context is defined as follows:

Definition 2.8 ($\Gamma_{\mathcal{L}}$) *Let $\mathcal{L}$ be a logic with set symbols $U_1, \ldots, U_k$, function symbols $f_1, \ldots, f_p$ and predicate symbols $P_1, \ldots, P_n$. Furthermore, let $V_{i,j}$ denote the set symbol representing the type of the j'th argument of function f_i and let V_i denote the set symbol representing the type of the result of function f_i. Finally, let $T_{i,j}$ denote the set symbol corresponding to the type of the j'th argument of predicate P_i. Then the context $\Gamma_{\mathcal{L}}$, modeling this first order logic in $\lambda P-$, is defined as:*

$$
\begin{aligned}
&U_1 : *_s, \ldots, U_k : *_s,\\
&f_1(x_1 : V_{1,1}, \ldots, x_{s_1} : V_{1,s_1}) : V_1, \ldots, f_p(x_1 : V_{p,1}, \ldots, x_{s_p} : V_{p,s_p}) : V_p,\\
&P_1(x_1 : T_{1,1}, \ldots, x_{r_1} : T_{1,r_1}) : *_p, \ldots, P_n(x_1 : T_{n,1}, \ldots, x_{r_n} : T_{n,r_n}) : *_p
\end{aligned}
$$

s_i and r_j are the arities of f_i and P_j respectively.

The close correspondence of logic $\mathcal{L}$ to $\lambda P-$ with context $\Gamma_{\mathcal{L}}$ is given by the following theorems:

Theorem 2.9 *$\Gamma_{\mathcal{L}} \vdash U : *_s$ if and only if U is a set symbol of $\mathcal{L}$.*

Theorem 2.10 *For any set symbol U of $\mathcal{L}$ we have $\Gamma_{\mathcal{L}} \vdash t : U$ if and only if t is a term in $\mathcal{L}$ whose type is represented by set symbol U.*

Theorem 2.11 *$\Gamma_{\mathcal{L}} \vdash P : *_p$ if and only if P is a proposition of $\mathcal{L}$.*

Theorem 2.12 *For any proposition P of $\mathcal{L}$ we have $\Gamma_{\mathcal{L}} \vdash p : P$ if and only if $\models_{\mathcal{L}} P$.*

Theorems 2.9 till 2.11 are proved by induction on the term structure. The completeness part of theorem 2.12 follows from the algorithm we present in the next section: the method of tableaux is complete and every closed tableau can be converted to a proof in $\lambda P-$ in context $\Gamma_{\mathcal{L}}$.

The converse is also true: if Γ is a valid context of $\lambda P-$, then there exists a logic $\mathcal{L}$ such that theorems 2.9 till 2.11 with $\Gamma_{\mathcal{L}}$ replaced by Γ hold. Hence, $\lambda P-$ has a one-to-one correspondence with many-sorted first-order predicate logic (for a proof see [10]).

To present the conversion algorithm, we need the following two theorems:

Theorem 2.13 *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 2.14 *Let $\Delta_1, A : B, \Delta_2$ be a legal context such that Δ_1, Δ_2 is also a legal context. If $\Delta_1, \Delta_2 \vdash C : D$ then $\Delta_1, A : B, \Delta_2 \vdash C : D$.*

Theorem 2.13 is proved by induction on the length of the context, using *intro* and *weaken* rules. Theorem 2.14 is proved by induction on the derivation of $C : D$.

3 From Closed Tableaux to λ-terms

In this section we will describe an algorithm to convert closed tableaux into λ-terms of $\lambda P-$. These λ-terms can easily be transformed into λ-terms of other PTSs, provided that these other PTSs are powerful enough. $\lambda P-$ merely states the minimal requirements for the conversion.

The closed tableau may be produced by any tableau-based theorem prover, allowing us to use existing tableau-based theorem provers as a module in an implementation of $\lambda P-$. This yields more powerful automated theorem proving than the usual exhaustive search used in PTSs (e.g. Coq's 'Auto' and Isabelle's 'fast_tac' tactics.). 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 automated theorem prover (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.

Using λ-terms to encode proofs allows us to concentrate on the structure of the conversion: for similar rules of the tableau method, similar conversion steps are performed. The classes of similar rules of the tableau method are the usual α-, β-, γ- and δ-rules. In figure 3 for each class the structure of the

rules is depicted. Our conversion algorithm will have one case for every class of rules. Without λ-terms, every rule would have to be treated explicitly.

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

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

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

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

Fig. 3. Structure of the different classes of tableau rules.

We will now show how to model each step of a tableau-proof of the formula A in $\lambda P-$. We assume that we have the basic context $\Gamma_{\mathcal{L}}$ (definition 2.8) to model the first order logic. Also we will intensively use the axiomatic extension for first-order constructs given in appendix A. In the conversion algorithm the labels of the tree correspond roughly to a context for $\lambda P-$. 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 $\lambda P-$ we will intensively use theorem 2.13 and theorem 2.14.

3.1 Converting the Initial Tableau

The tableau starts with a node labeled by $\neg A$ and the initial context for $\lambda P-$ will be $\Gamma_{\mathcal{L}}, p : \neg A$. The tableau represents a contradiction derived from $\neg A$ and hence, converting the tableau should result in a contradiction $c : \bot$ derived from the context $\Gamma_{\mathcal{L}}, p : \neg A$. The validity of A in $\lambda P-$ is then given by $\Gamma_{\mathcal{L}} \vdash classic\ A\ (\lambda p : \neg A.c) : A$ (see appendix A). ($classic\ A\ (\lambda p : \neg A.c)$ is read as A is proved in classical logic by $(\lambda p : \neg A.c)$, which actually is a proof of $\neg\neg A$).

3.2 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 $\Gamma_{\mathcal{L}}, \Delta_1, x : X, \Delta_2$, 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.

3.2.1 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 $\Gamma_{\mathcal{L}}, \Delta_1, o : \neg\neg P, \Delta_2$. For the successor-node we create the corresponding context $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P$. By recursion, we derive a contradiction c from this successor node, hence we have $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P \vdash c : \bot$. Then the contradiction we seek is derived as follows:

(0)	$\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P \vdash c : \bot$	induction
(1)	$\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda p : P.c) : P \Rightarrow \bot$	Π-*intro* on (0)
(2)	$\Gamma_{\mathcal{L}}, \Delta_1, o : (P \Rightarrow \bot) \Rightarrow \bot, \Delta_2 \vdash o : (P \Rightarrow \bot) \Rightarrow \bot$	see remark below and theorem 2.13
(3)	$\Gamma_{\mathcal{L}}, \Delta_1, o : (P \Rightarrow \bot) \Rightarrow \bot, \Delta_2 \vdash o\ (\lambda p : P.c) : \bot$	Π-*elim* on (1) and (2)

Remark: Formally we also need to derive types in order to apply the PTS-rules. For instance in step (2), we use theorem 2.13 to insert a variable of the type $(P \Rightarrow \bot) \Rightarrow \bot$ in the context, but for this we also need a type judgment saying $\Gamma_{\mathcal{L}}, \Delta_1 \vdash (P \Rightarrow \bot) \Rightarrow \bot : *_p$. Such a type judgment can be derived by:

(*a*)	$\Gamma_{\mathcal{L}}, \Delta_1 \vdash P : *_p$	Theorem 2.11 and $P \in *_p^L$
(*b*)	$\Gamma_{\mathcal{L}}, \Delta_1, p : P \vdash \bot : *_p$	Axiom of $\lambda P-$ and repeated weaken
(*c*)	$\Gamma_{\mathcal{L}}, \Delta_1 \vdash P \Rightarrow \bot : *_p$	Π-*form* on (a) and (b)
(*d*)	$\Gamma_{\mathcal{L}}, \Delta_1, p : (P \Rightarrow \bot) \vdash \bot : *_p$	Axiom of $\lambda P-$ and repeated weaken
(*e*)	$\Gamma_{\mathcal{L}}, \Delta_1 \vdash (P \Rightarrow \bot) \Rightarrow \bot : *_p$	Π-*form* on (c) and (d)

For reasons of space and simplicity, we will omit these type derivations. Usu-

ally it will be evident that the types are correct.

3.2.2 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 $\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash ? : \bot$. To the successor node, we assign the context $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P, q : Q$. By recursion, we get from this context a contradiction: $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P, q : Q \vdash c : \bot$. To derive a contradiction from the original context we use the following derivation:

(0)	$\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P, q : Q \vdash c : \bot$	induction
(1)	$\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P \vdash (\lambda q : Q.c) : Q \Rightarrow \bot$	Π-*intro* on (0)
(2)	$\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c)) : P \Rightarrow (Q \Rightarrow \bot)$	Π-*intro* on (1)
(3)	$\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash o : P \wedge Q$	theorem 2.13
(4)	$\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash \pi_1(o) : P$	$\wedge$-*elim*$_1$ on (3)
(5)	$\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash \pi_2(o) : Q$	$\wedge$-*elim*$_2$ on (3)
(6)	$\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c)) : P \Rightarrow (Q \Rightarrow \bot)$	theorem 2.14 on (2)
(7)	$\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o) : Q \Rightarrow \bot$	Π-*elim* on (4,6)
(8)	$\Gamma_{\mathcal{L}}, \Delta_1, o : P \wedge Q, \Delta_2 \vdash (\lambda p : P.(\lambda q : Q.c))\ \pi_1(o)\ \pi_2(o) : \bot$	Π-*elim* on (5,7)

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 $\Gamma_{\mathcal{L}}, \Delta_1, o : E(P, Q), \Delta_2$. To the successor node we assign the context $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : E_1(P), q : E_2(Q)$ 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 $\Gamma_{\mathcal{L}}, \Delta_1, o : E(P, Q), \Delta_2 \vdash ?' :$

$E_1(P) \wedge E_2(Q)$. 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 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 1
Conversion functions for α-rules.

3.2.3 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 $\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2$ then its successors will have contexts $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P$ and $\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, q : Q$ 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:

$(0)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, p : P \vdash c_1 : \perp$	induction
$(1)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, q : Q \vdash c_2 : \perp$	induction
$(2)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda p : P.c_1) : P \Rightarrow \perp$	Π-*intro* on (0)
$(3)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda q : Q.c_2) : Q \Rightarrow \perp$	Π-*intro* on (1)
$(4)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash ((\lambda p : P.c_1) \nabla (\lambda q : Q.c_2)) : (P \vee Q) \Rightarrow \perp$	$\vee$-*elim* on (2,3)
$(5)\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2 \vdash o : P \vee Q$	theorem 2.13
$(6)\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2 \vdash ((\lambda p : P.c_1) \nabla (\lambda q : Q.c_2)) : (P \vee Q) \Rightarrow \perp$	theorem 2.14 on (4)
$(7)\Gamma_{\mathcal{L}}, \Delta_1, o : P \vee Q, \Delta_2 \vdash ((\lambda p : P.c_1) \nabla (\lambda q : Q.c_2))\ o : \perp$	Π-*elim* on (5,6)

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 2 but their derivation is omitted. Again, the transformation functions T produce λ-terms but are not λ-terms themselves.

$E(P,Q)$	$E_1(P)$	$E_2(Q)$	$T(o) : E_1(P) \vee E_2(Q)$ with $o : E(P,Q)$
$\neg(P \wedge Q)$	$\neg P$	$\neg Q$	$classic\ (\neg P \vee \neg Q)\ \lambda r : \neg(\neg P \vee \neg Q).$
			$\quad r(injl\ (\neg P \vee \neg Q)\ (\lambda p : P.r(injr\ (\neg P \vee \neg Q)$
			$\quad (\lambda q : Q.o(p,q)))))$
$P \Rightarrow Q$	$\neg P$	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 2
Conversion functions for β-rules.

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

3.2.4 Conversion for γ-rules

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

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

The current context is $\Gamma_\mathcal{L}, \Delta_1, o : (\exists x : U.P), \Delta_2$. For γ-rules 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 $\Gamma_\mathcal{L}, \Delta_1, \Delta_2, \theta : U, p : P$. By recursion we have derived a

contradiction c from this context. A contradiction from the current context is derived as follows:

$$
\begin{array}{ll}
(0)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, \theta : U, p : P \vdash c : \bot & \text{induction} \\
(1)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2, \theta : U \vdash (\lambda p : P.c) : P \Rightarrow \bot & \Pi\text{-}intro \text{ on } (0) \\
(2)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash (\lambda \theta : U.(\lambda p : P.c)) : (\forall \theta : U.(P \Rightarrow \bot)) & \Pi\text{-}intro \text{ on } (1)
\end{array}
$$

$$
\begin{array}{ll}
(3)\Gamma_{\mathcal{L}}, \Delta_1, \Delta_2 \vdash \Diamond\, (\exists x : U.P)\ (\lambda \theta : U.(\lambda p : P.c)) & \exists\text{-}elim \text{ on } (2) \text{ for } \bot \\
\qquad : (\exists x : U.P) \Rightarrow \bot & \\
(4)\Gamma_{\mathcal{L}}, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash o : (\exists x : U.P) & \text{theorem 2.13} \\
(5)\Gamma_{\mathcal{L}}, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash \Diamond\, (\exists x : U.P)\ (\lambda \theta : U.(\lambda p : P.c)) & \text{theorem 2.14 on } (3) \\
\qquad : (\exists x : U.P) \Rightarrow \bot & \\
(6)\Gamma_{\mathcal{L}}, \Delta_1, o : (\exists x : U.P), \Delta_2 \vdash (\Diamond\, (\exists x : U.P) & \Pi\text{-}elim \text{ on } (4,5) \\
\qquad (\lambda \theta : U.(\lambda p : P.c)))\ o : \bot &
\end{array}
$$

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.

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

Table 3
Conversion functions for γ-rules.

3.2.5 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 $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2$ and the term t used to extend the tableau, we construct for the successor node the context $\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2, p : P_t^x$. 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:

$(0)\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2, p : P_t^x \vdash c : \bot$	induction
$(1)\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash (\lambda p : P_t^x.c) : P_t^x \Rightarrow \bot$	Π-*intro* on (0)
$(2)\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash o : (\forall x : U.P)$	theorem 2.13
$(3)\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash t : U$	theorem 2.10
$(4)\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash o\ t : P_t^x$	Π-*elim* on (2,3)
$(5)\Gamma_{\mathcal{L}}, \Delta_1, o : (\forall x : U.P), \Delta_2 \vdash (\lambda p : P_t^x.c)\ (o\ t) : \bot$	Π-*elim* on (1,4)

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 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 4
Conversion functions for δ-rules.

3.3 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 $\lambda P-$ negation is modeled by implication and $\bot$. Hence, in the context $\Gamma_{\mathcal{L}}, \Delta_1, p : P, \Delta_2, p' : \neg P, \Delta_3$ we can derive the contradiction $p'p$.

4 Formalizing the Algorithm

The conversion algorithm can also be described as a function C from closed tableaux to λ-terms. Although this allows us to formally prove correctness of the conversion, we chose for the previous presentation since it is more descriptive in how the λ-terms are obtained. For the sake of completeness, we will now illustrate how the formal definitions are constructed.

Definition 4.1 (Conversion function C) *Let C :Closed Tableaux $\rightarrow T$ be the conversion function defined as*

$$C(T) = classic\ P\ (\lambda p : \neg P.C'(\Gamma_{\mathcal{L}}, p : \neg P; T))$$

where $L(T) = \{\neg P\}$ is the label of the root of the tableau and C' :Contexts$\times$ Tableaux$\rightarrow T$ is an auxiliary function to be defined next.

Definition 4.2 (Auxiliary function C') *The auxiliary function C' :Contexts$\times$ Tableaux$\rightarrow T$ is defined recursively by distinction between the type of rule applied to the label of the tableaux. The definition of C' follows the description given in the presentation of the algorithm.*

We can now state the correctness of the algorithm by the following theorems. We only give sketches of the proofs of these theorems, since the proofs can easily be extracted from the presentation of the algorithm.

Theorem 4.3 (Correctness of contradictions) *Let T be a closed tableau and let Γ be a valid context such that $L(T) = \{P \in \mathcal{P} \mid \exists p \in V.(p : P) \in \Gamma\}$ ($\mathcal{P}$ denotes the set of propositional formulas of the logic $\mathcal{L}$) then*

$$\Gamma \vdash C'(\Gamma, T) : \bot$$

Proof. By induction on the depth of the tableau. We will need cases for leafs and cases for nodes to which the special rule or one of the α-, β-, δ- or γ-rules is applied. It is easy to verify that the premises hold for the recursive function calls. □

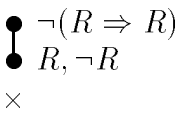
Theorem 4.4 (Correctness of conversion) *Let T be a closed tableau for P (i.e. $L(T) = \{\neg P\}$) then*

$$\Gamma_{\mathcal{L}} \vdash C(T) : P$$

Proof. See converting the initial tableau and use theorem 4.3. □

5 Properties of Converted Tableaux

The converted proof may be much longer than a proof that is constructed directly in $\lambda P-$. For example: a direct proof of $R \Rightarrow R$ in $\lambda P-$ looks like $\Gamma_{\mathcal{L}} \vdash (\lambda p : R.p) : R \Rightarrow R$. However, if we convert the tableau



we get a much larger λ-term. Following the algorithm, we start with $\Gamma_{\mathcal{L}} \vdash$ $classic\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).c) : R \Rightarrow R$, where c is a contradiction extracted from the initial context $\Gamma_{\mathcal{L}}, o : \neg(R \Rightarrow R)$. 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 $\Gamma_{\mathcal{L}}, p : E_1(P), q : E_2(Q)$. 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

$$classic\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).(\lambda p : R.\lambda q : \neg R.c')$$
$$\pi_1(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))$$
$$\pi_2(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))) : R \Rightarrow R$$

and c' is the contradiction derived from the context $\Gamma_{\mathcal{L}}, p : R, q : \neg R$. 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:

$$\Gamma_{\mathcal{L}} \vdash\ classic\ (R \Rightarrow R)\ (\lambda o : \neg(R \Rightarrow R).(\lambda p : R.\lambda q : \neg R.qp)$$
$$\pi_1(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))$$
$$\pi_2(classic\ R\ (\lambda p : \neg R.o(\lambda q : R.pqR)), (\lambda q : R.o(\lambda p : R.q)))) : R \Rightarrow R$$

Note that if during the construction of a tableau needless steps are taken these will also be translated, which makes matters even worse.

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.

A Type Judgement Rules for additional Logical Constructs

$$\bot\text{-}intro \qquad <> \vdash \bot : *_p$$

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

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

$$\wedge\text{-}form \qquad \frac{\Gamma \vdash P : *_p \; \Gamma \vdash Q : *_p}{\Gamma \vdash P \wedge Q : *_p}$$

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

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

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

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

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

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

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

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

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

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

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 $\lambda P-$ 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.

References

[1] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part I. *ACM Transactions on Programming Languages and Systems*, 3(4):432–483, October 1981.

[2] H.P. Barendregt. *Lambda Calculi with Types*, volume 2 of *Handbook of Logic in Computer Science*, chapter 2, pages 118–310. Oxford Science Publications, 1992.

[3] S Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Dept. of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, 1988.

[4] L.S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Propositions: International Workshop TYPES'93*, volume 806 of *LNCS*, pages 19–61, Nijmegen, May 1993. Springer-Verlag 1994.

[5] Coq. The Coq proof assistant. In *URL:* `http://pauillac.inria.fr/coq/`, 1997.

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

[7] J.H. Geuvers. *Logics and Type Systems*. PhD thesis, Catholic University of Nijmegen, 1993.

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

[9] LEGO. The LEGO proof assistant. In *URL:* `http://www.dcs.ed.ac.uk/home/lego/`, 1997.

[10] Twan Laan and Michael Franssen. Embedding first-order logic in a pure type system with parameters. Submitted for publication.

[11] Rob Nederpelt, Herman Geuvers, and Roel de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, 1994.

[12] J Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmegen, 1989.

[13] Jan Zwanenburg. The Yarrow home page. In *URL:* `http://www.win.tue.nl/cs/pa/janz/yarrow/`, 1997.