Bachelor thesis
Computer Science

Radboud University

A concrete deskolemization algorithm

Author:
Ramon van Sparrentak
0757276

Supervisor/assessor:
dr. Freek Wiedijk

Second assessor:
prof. dr. Herman Geuvers

July 8, 2014

Abstract

Skolemization is a common transformation in automated theorem provers. This paper presents an implementation of the reverse process, deskolemization of a proof in sequent calculus. The implementation is based on work of M. Baaz, S. Hetzl and D. Weller in *On the complexity of proof deskolemization.*

Contents

Chapter 1

Introduction

An automated theorem prover (ATP) is a computer program that tries to find a proof for a formula. Automated theorem provers are used in for example program and integrated circuit verification. These provers aid in development by proving the correctness of parts of the implementation. Finding a proof is generally not easy, and an ATP may fail to find one.

A proof generated by an ATP system is unfortunately not a proof in *natural deduction* and is difficult to understand. Natural deduction is a proof calculus that was designed to be close to actual reasoning [4].

One of the problems in transforming a proof from clausal logic to natural deduction are the Skolem functions that are introduced by the automated theorem prover.

For an user seeking a proof of a theorem an automated theorem prover gives only a proof of the Skolemized theorem. Due to the equisatisfiability of the formulas there exists a proof of the original formula. But what does the proof look like? Or how do we construct it?

Obtaining a proof for a formula from proof a of the Skolemized formula is called *deskolemization.*

This paper describes an algorithm and its implementation for deskolemizing cut-free proofs in LK. The algorithm is derived from the definitions in [7].

The remainder of this paper is organized as follows. In section 1.1 the drinker paradox is explained which is used as a running example. In section 3 the calculus is described. Section 4 explains Skolemization and finally in section 5 the deskolemization.

1.1 Drinker paradox

The drinker paradox will be used in this paper as the running example. The paradox is also called the Drinkers' Principle [10]. The drinker paradox as a first order formula is

$$\exists x\,(D(x) \rightarrow \forall y D(y))$$

In natural language the drinker paradox can be stated as *There is someone in the pub such that, if he is drinking, everyone in the pub is drinking.* The statement seems to be false. How can it be that if this person is drinking, then everybody must be drinking as well? There are two important points to see why the paradox is true. There is no time involved. The paradox does not claim if someone *starts* drinking, then everybody will drink. To show it is true, we can pick anyone we want to be the *someone.* Thus given any pub, pick someone who is not drinking. (If we can't pick someone who isn't drinking, the paradox is true since everyone is drinking). Then the paradox is true, because he or she isn't drinking. And if this person starts to drink? Just pick someone else who isn't drinking.

1.2 Automated theorem provers

The best automated theorem provers can be found at the CADE ATP System Competition, CASC, an annual competition for automated theorem provers [12][9]. CASC evaluates the performance of ATP systems on problems from Thousands of Problems for Theorem Provers, TPTP [11].

The winner of the CASC-24 in 2013 in the division of formulas in first order form was Vampire 2.6. Vampire and others like Prover9 use proof by refutation in clausal logic [1].

Most ATP systems use proof by refutation. They assume the theorem to be false and show the negated theorem to be unsatisfiable by deriving a contradiction. A proof by refutation for the drinker paradox will show there is someone who is drinking and not drinking if the drinker paradox is false.

1.3 Skolemization

Skolemization is commonly used in ATP systems before proving a formula. Skolemization replaces existential quantifiers in a formula by Skolem functions. The resulting formula is equisatisfiable with the original formula, but is easier to prove. The Skolemized formula preserves the satisfiability of the original formula. Which means that iff there is a model that makes the Skolemized formula true then there is a model that makes the original formula true. This is an useful property in a proof by refutation. A proof by refutation proves there is no model that satisfies the negated formula, thus the formula itself must be valid. A refutation proof of a Skolemized negated formula implies there is also no model for the negated formula, and thus the formula is valid.

Herbrandization is the dual of Skolemization. It eliminates universal quantifiers. Herbrandization preserves the validity of the formula.

1.4 Example

As an example we consider the case of constructing a proof in sequent calculus of the drinker paradox using Prover9, an automated theorem prover. The drinker paradox in suitable input format is

```
formulas(goals).
  (exists x (D(x) -> (all y D(y)))).
end_of_list.
```

Prover9 will negate the formula, put it in prenex normal form, Skolemize it and turn it in clausal form. The resulting Skolemized form is

```
formulas(sos).
D(x).  [deny(1)].
-D(f1(x)).  [deny(1)].
end_of_list.
```

The function f_1 is the Skolem function introduced. The intermediate steps are

$$\begin{aligned}
\neg\left(\exists x\left(D(x) \rightarrow \forall y D(y)\right)\right) \\
\neg\left(\exists x \forall y\left(D(x) \rightarrow D(y)\right)\right) \\
\forall x \exists y \neg\left(D(x) \rightarrow D(y)\right) \\
\forall x \exists y\left(D(x) \wedge \neg D(y)\right) \\
\forall x\left(D(x) \wedge \neg D(f_1(x))\right) \\
\{D(a)\}, \{\neg D(f_1(a))\}
\end{aligned}$$

The actual proof constructed by Prover9 is

```
1 (exists x (D(x) -> (all y D(y)))) # label(non_clause) # label(goal).  [goal].
2 -D(f1(x)).  [deny(1)].
3 D(x).  [deny(1)].
4 $F.  [resolve(2,a,3,a)].
```

Thus the drinker paradox is true. Why it is true is not clear from this proof. We could transform this proof in to a proof in LK. In sequent calculus the proof from Prover9 corresponds to a proof of $\forall x D(x) \wedge \forall y \neg D(f_1(y)) \vdash$

$$\begin{array}{c} [\pi] \\ \forall x D(x) \wedge \forall y \neg D(f_1(y)) \vdash \end{array}$$

With this proof we can make a refutation proof in LK for the Skolemized drinker paradox

$$\dfrac{\begin{array}{c}[\phi]\\ \neg\left(\exists x\left(D(x)\rightarrow D(f_1(x))\right)\right)\vdash\forall xD(x)\wedge\forall y\neg D(f_1(y))\end{array}\qquad\begin{array}{c}[\pi]\\ \forall xD(x)\wedge\forall y\neg D(f_1(y))\vdash\end{array}}{\neg\left(\exists x\left(D(x)\rightarrow D(f_1(x))\right)\right)\vdash}\ \text{Cut}$$

The calculus in this paper does not have a Cut rule. However, Cut rules can be eliminated from a proof [5]. CERES is a system capable of eliminating cut in first order logic [2].

The proof ϕ is simple as the antecedent and succedent are logically equivalent. The proof of π can be created from the proof by Prover9. This proof is still not a proof of the drinker paradox. There still is a Skolem function f_1.

Thus the last step necessary step to obtain a refutation proof of the drinker paradox, from the Skolemized proof of $\neg\left(\exists x\left(D(x)\rightarrow D(f_1(x))\right)\right)\vdash$, requires deskolemizing the proof. A method of deskolemizing, and its implementation, is explained in the following sections.

Chapter 2

Proof deskolemization

We now present the algorithm for deskolemization, constructing a proof for a theorem from a proof of the Skolemized theorem. The algorithm takes as input a sequent S to prove and a proof π' of S', the Skolemized S. The output is then a proof π of S. The sequent calculus, LK, for these proofs is defined in section 3.

The deskolemization of a proof consists of four steps. In the first step a compact representation of the proof, an expansion, is extracted. Secondly, this expansion is deskolemized. Thirdly, a proof of the deskolemized expansion is constructed in LK^{E} calculus. Finally, the proof in LK^{E} is transformed into proof in LK.

Let

S The sequent to proof

S' The Skolemized sequent

π' A proof of S'

E Expansion sequent of S

E' Expansion sequent of S', deskolemized S

ϕ A proof of E' in LK^{E}

π A proof of S in LK

then the process of finding a proof π of S can be depicted as

$$\begin{tikzcd}
\text{LK} & S \arrow[d, "\mathsf{sk}"] & & & \pi \\
\text{LK, Skolemized} & S' = \mathsf{sk}(S) \arrow[r, "ATP"] & \text{proof } \pi' \arrow[d, "\mathsf{e}"] & & \\
\mathrm{LK}^{\mathrm{E}} & & E' \arrow[r, "\mathsf{desk}"] & E \arrow[r, "\mathsf{Pr}"] & \text{proof } \phi \text{ of } S \arrow[uu, "\mathsf{rm,rmTF}"']
\end{tikzcd}$$

The function sk Skolemizes a sequent. Given S and π' the goal is to construct a proof π of S.

The expansion is extracted by e from the proof, section 5.1. This expansion is deskolemized by desk, section 5.3. The proof in LK^{E} is reconstructed by Pr and is transformed into a proof in LK by rm and rmTF.

2.1 Example

Let $S = \vdash \exists x(P(x) \to \forall y P(y))$ its Skolemization is $S' = \vdash \exists x\, P(x) \to P(f_0(x))$. A proof π of S' is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(f_0(a)), P(a) \vdash \exists x(P(x) \to P(f_0(x))), P(f_0(f_0(a))), P(f_0(a))}}{P(f_0(a)), P(a) \vdash P(f_0(a)), \exists x(P(x) \to P(f_0(x))), P(f_0(f_0(a)))}\,\mathrm{PR}_1}{P(a) \vdash P(f_0(a)), \exists x(P(x) \to P(f_0(x))), P(f_0(a)) \to P(f_0(f_0(a)))}\,{\to}\mathrm{R}}{P(a) \vdash P(f_0(a)), \exists x(P(x) \to P(f_0(x)))}\,\exists\mathrm{R}_{f(a)}}{P(a) \vdash \exists x(P(x) \to P(f_0(x))), P(f_0(a))}\,\mathrm{PR}_1}{\vdash \exists x(P(x) \to P(f_0(x))), P(a) \to P(f_0(a))}\,{\to}\mathrm{R}}{\vdash \exists x(P(x) \to P(f_0(x)))}\,\exists\mathrm{R}_a$$

The goal is to construct the following proof of S

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(z), P(a) \vdash \exists x(P(x) \to \forall y P(y)), \forall y P(y)), P(z)}}{P(z), P(a) \vdash P(z), \exists x(P(x) \to \forall y P(y)), \forall y P(y))}\,\mathrm{PR}_1}{P(a) \vdash P(z), \exists x(P(x) \to \forall y P(y)), P(z) \to \forall y P(y))}\,{\to}\mathrm{R}}{P(a) \vdash P(z), \exists x(P(x) \to \forall y P(y))}\,\exists\mathrm{R}_z}{P(a) \vdash \exists x(P(x) \to \forall y P(y)), P(z)}\,\mathrm{PR}_1}{P(a) \vdash \exists x(P(x) \to \forall y P(y)), \forall y P(y)}\,\forall\mathrm{R}_z}{\vdash \exists x(P(x) \to \forall y P(y)), P(a) \to \forall y P(y)}\,{\to}\mathrm{R}}{\vdash \exists x(P(x) \to \forall y P(y))}\,\exists\mathrm{R}_a$$

Chapter 3

Definitions

In this section the first order sequent calculus used in this thesis is described. It is slightly different from the calculus used in [7], $\top$ and $\bot$ are omitted.

Definition 1. Symbols

- Variables $x, y, \ldots$ and $x_0, y_0, x_1, y_1, \ldots$
- Logical connectives $\neg, \wedge, \vee, \rightarrow$
- Predicates $P_0, P_1, \ldots$
- Functions $f_0, f_1, \ldots$
- Quantifiers $\exists, \forall$

Definition 2. Terms

- A variable symbol
- An expression $f(t_1, \ldots, t_n)$ where f is a function with arity n and t_i are terms

Definition 3. Formulas

- $P(t_1, \ldots, t_n)$ where P is a predicate with arity n and t_i are terms
- $\neg A$ where A is a formula
- $(A_1 \vee A_2)$ where A_1 and A_2 are formulas
- $(A_1 \wedge A_2)$ where A_1 and A_2 are formulas
- $(A_1 \rightarrow A_2)$ where A_1 and A_2 are formulas
- $(\exists x A)$ where x is a variable and A a formula
- $(\forall x A)$ where x is a variable and A a formula

The result of a substitution $A\,[x := t]$ is A with all free occurrences of x substituted by t.

Definition 4. Every subformula of a formula is in positive or negative context. The formula itself is in positive context.

For every (sub)formula A

- If $A = \neg A_1$ is in positive context, then A_1 is in negative context
- If $A = \neg A_1$ is in negative context, then A_1 is in positive context
- If $A = A_1 \rightarrow A_2$ is in positive context, then A_1 is in negative context and A_2 is in positive context

- If $A = A_1 \to A_2$ is in negative context, then A_1 is in positive context and A_2 is in negative context
- If $A = A_1 \vee A_2$, $A = A_1 \wedge A_2$, $A = \exists x A_1$ or $A = \forall x A_1$ then A_1 and A_2 have the same context parity as A

A quantifier $\forall x$ is called strong (weak) if it occurs in a positive (negative) context, a quantifier $\exists x$ is called weak (strong) if it occurs in a positive (negative) context [7, Preliminaries]. The number of strong quantifiers in a formula A will be denoted as $\mathsf{qocc}(A)$.

Definition 5. A sequent $S = A_1, \dots, A_n \vdash B_1, \dots, B_m$ where $A_1, \dots, A_n$ and $B_1, \dots, B_m$ are sequences of formulas. The formulas $A_1, \dots, A_n$ are in negative context and $B_1, \dots, B_m$ are in positive context. This differs from [7] where sequents are multisets of formulas.

Definition 6. Inference rules. A denote a single formula and Γ, Δ sequences of formulas

$$\frac{}{A, \Gamma \vdash \Delta, A}\ \mathrm{I}$$

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}\ \neg\mathrm{L} \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A}\ \neg\mathrm{R}$$

$$\frac{A_1, \Gamma \vdash \Delta \qquad A_2, \Gamma \vdash \Delta}{A_1 \vee A_2, \Gamma \vdash \Delta}\ \vee\mathrm{L} \qquad \frac{\Gamma \vdash \Delta, A_1, A_2}{\Gamma \vdash \Delta, A_1 \vee A_2}\ \vee\mathrm{R}$$

$$\frac{A_1, A_2, \Gamma \vdash \Delta}{A_1 \wedge A_2, \Gamma \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{\Gamma \vdash \Delta, A_1 \qquad \Gamma \vdash \Delta, A_2}{\Gamma \vdash \Delta, A_1 \wedge A_2}\ \wedge\mathrm{R}$$

$$\frac{\Gamma \vdash \Delta, A_1 \qquad A_2, \Gamma \vdash \Delta}{A_1 \to A_2, \Gamma \vdash \Delta}\ \to\mathrm{L} \qquad \frac{A_1, \Gamma \vdash \Delta, A_2}{\Gamma \vdash \Delta, A_1 \to A_2}\ \to\mathrm{R}$$

$$\frac{A\,[x := y]\,, \Gamma \vdash \Delta}{\exists x A, \Gamma \vdash \Delta}\ \exists\mathrm{L}_y \qquad \frac{\Gamma \vdash \Delta, \exists x A, A\,[x := t]}{\Gamma \vdash \Delta, \exists x A}\ \exists\mathrm{R}_t$$

$$\frac{A\,[x := t]\,, \forall x A, \Gamma \vdash \Delta}{\forall x A, \Gamma \vdash \Delta}\ \forall\mathrm{L}_t \qquad \frac{\Gamma \vdash \Delta, A\,[x := y]}{\Gamma \vdash \Delta, \forall x A}\ \forall\mathrm{R}_y$$

$$\frac{A_i, A_1, \dots, A_{i-1}, A_{i+1}, \dots, A_n \vdash \Delta}{A_1, \dots, A_n \vdash \Delta}\ \mathrm{PL}_i \qquad \frac{\Gamma \vdash A_1, \dots, A_{i-1}, A_{i+1}, \dots, A_n, A_i}{\Gamma \vdash A_1, \dots, A_n}\ \mathrm{PR}_i$$

y does not occur in A, Γ nor Δ.

Definition 7. A proof π in LK of a sequent S is a tree built from inference rules, with at the root of the tree S and all nodes are inferences. The length of a proof $|\pi|$ is the number of inferences in π, excluding PR and PL [7, Definition 2].

3.1 Expansions and LK$^{\mathbf{E}}$

Expansion trees are a simple representation of proofs. In a classical proof the substitutions used are the key element of a proof. These are the terms in the $\exists$R and $\forall$L inference steps. Expansion trees make the substitutions explicit, but the order of the inferences are omitted. Due to their compact representation, expansions are well suited for transformations. [8]

Definition 8. Expansions [7, Definition 4]

- $\bot$ is an expansion
- $\top$ is an expansion

- $P(t_1, \dots, t_n)$ where P is a predicate with arity n and t_i are terms
- $\neg E$ where E is an expansion
- $(E_1 \vee E_2)$ where E_1 and E_2 are expansions
- $(E_1 \wedge E_2)$ where E_1 and E_2 are expansions
- $(E_1 \to E_2)$ where E_1 and E_2 are expansions
- $\exists x A +^{t_1} E_1 + \cdots +^{t_n} E_n$ is an expansion where A is a formula and E_i are expansions
- $\forall x A +^{t_1} E_1 + \cdots +^{t_n} E_n$ is an expansion where A is a formula and E_i are expansions

The $+$ operator is commutative. A term t in an expansion $A +^t E$ is called a *selected* term.

The shallow function maps expansions to formulas [8]. It will, for example, be used to convert a proof from LK^{E} to LK in section 5.7.

$$\begin{aligned}
\mathsf{Sh}(\top) &= \top \\
\mathsf{Sh}(\bot) &= \bot \\
\mathsf{Sh}(P(t_1, \dots, t_n)) &= P(t_1, \dots, t_n) \\
\mathsf{Sh}(\neg E) &= \neg \mathsf{Sh}(E) \\
\mathsf{Sh}(E_1 \vee E_2) &= \mathsf{Sh}(E_1) \vee \mathsf{Sh}(E_2) \\
\mathsf{Sh}(E_1 \wedge E_2) &= \mathsf{Sh}(E_1) \wedge \mathsf{Sh}(E_2) \\
\mathsf{Sh}(E_1 \to E_2) &= \mathsf{Sh}(E_1) \to \mathsf{Sh}(E_2) \\
\mathsf{Sh}(\forall x A +^{t_1} E_1 \cdots +^{t_n} E_n) &= \forall x A \\
\mathsf{Sh}(\exists x A +^{t_1} E_1 \cdots +^{t_n} E_n) &= \exists x A
\end{aligned}$$

For a sequence $\Gamma = E_1, \dots, E_n$ its shallow form $\mathsf{Sh}(\Gamma) = \mathsf{Sh}(E_1), \dots, \mathsf{Sh}(E_n)$. For a sequent $\Gamma \vdash \Delta$ its shallow form $\mathsf{Sh}(\Gamma \vdash \Delta) = \mathsf{Sh}(\Gamma) \vdash \mathsf{Sh}(\Delta)$.

Definition 9. Expansion of a formula [7, Definition 4]

- $\bot$ is an expansion of every formula
- $\top$ is a dual expansion of every formula
- $P(t_1, \dots, t_n)$ is a (dual) expansion of $P(t_1, \dots, t_n)$
- $\neg E$ is a dual expansion of $\neg A$ when E is an expansion of A
- $\neg E$ is an expansion of $\neg A$ when E is a dual expansion of A
- $E_1 \vee E_2$ is an (dual) expansion of $A_1 \vee A_2$ when E_1 and E_2 are (dual) expansions of A_1 and A_2
- $E_1 \wedge E_2$ is an (dual) expansion of $A_1 \wedge A_2$ when E_1 and E_2 are (dual) expansions of A_1 and A_2
- $E_1 \to E_2$ is an expansion of $A_1 \to A_2$ when E_1 is a dual expansion of A_1 and E_2 is an expansion of A_2
- $E_1 \to E_2$ is a dual expansion of $A_1 \to A_2$ when E_1 is an expansion of A_1 and E_2 is a dual expansion of A_2
- $\exists x A +^{t_1} E_1 + \cdots +^{t_n} E_n$ is an expansion of $\exists x A$ when E_i are expansions of $A\,[x := t_i]$
- $\exists x A +^{f(t_1, \dots, t_n)} E$ is a dual expansion of $\exists x A$ when E is a dual expansion of $A\,[x := f(t_1, \dots, t_n)]$
- $\forall x A +^{t_1} E_1 + \cdots +^{t_n} E_n$ is a dual expansion of $\forall x A$ when E_i are dual expansions of $A\,[x := t_i]$
- $\forall x A +^{f(t_1, \dots, t_n)} E$ is an expansion of $\forall x A$ when E is an expansion of $A\,[x := f(t_1, \dots, t_n)]$

Definition 10. LK$^{\mathrm{E}}$ [7, Definition 8]

$$\frac{}{E, \Gamma \vdash \Delta, E}\ \mathrm{I}$$

$$\frac{\Gamma \vdash \Delta, E}{\neg E, \Gamma \vdash \Delta}\ \neg\mathrm{L} \qquad \frac{E, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg E}\ \neg\mathrm{R}$$

$$\frac{E_1, \Gamma \vdash \Delta \qquad E_2, \Gamma \vdash \Delta}{E_1 \vee E_2, \Gamma \vdash \Delta}\ \vee\mathrm{L} \qquad \frac{\Gamma \vdash \Delta, E_1, E_2}{\Gamma \vdash \Delta, E_1 \vee E_2}\ \vee\mathrm{R}$$

$$\frac{E_1, E_2, \Gamma \vdash \Delta}{E_1 \wedge E_2, \Gamma \vdash \Delta}\ \wedge\mathrm{L} \qquad \frac{\Gamma \vdash \Delta, E_1 \qquad \Gamma \vdash \Delta, E_2}{\Gamma \vdash \Delta, E_1 \wedge E_2}\ \wedge\mathrm{R}$$

$$\frac{\Gamma \vdash \Delta, E_1 \qquad E_2, \Gamma \vdash \Delta}{E_1 \to E_2, \Gamma \vdash \Delta}\ \to\mathrm{L} \qquad \frac{E_1, \Gamma \vdash \Delta, E_2}{\Gamma \vdash \Delta, E_1 \to E_2}\ \to\mathrm{R}$$

$$\frac{E, \Gamma \vdash \Delta}{\exists x A +^{t'} E, \Gamma \vdash \Delta}\ \exists\mathrm{L}_{t'} \qquad \frac{\Gamma \vdash \Delta, \exists x A + \omega, E}{\Gamma \vdash \Delta, \exists x A +^{t} E + \omega}\ \exists\mathrm{R}_t$$

$$\frac{E, \forall x A + \omega, \Gamma \vdash \Delta}{\forall x A +^{t'} E + \omega, \Gamma \vdash \Delta}\ \forall\mathrm{L}_t \qquad \frac{\Gamma \vdash \Delta, E}{\Gamma \vdash \Delta, \forall x A +^{t'} E}\ \forall\mathrm{R}_{t'}$$

$$\frac{E_i, E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n \vdash \Delta}{E_1, \ldots, E_n \vdash \Delta}\ \mathrm{PL}_i \qquad \frac{\Gamma \vdash E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n, E_i}{\Gamma \vdash E_1, \ldots, E_n}\ \mathrm{PR}_i$$

t' does not occur in A,$\mathsf{Sh}(\Gamma)$ nor $\mathsf{Sh}(\Delta)$.

The Skolem terms of an expansion $\mathsf{SkTerms}(E)$ is the set of selected terms at its strong quantifiers.

$$\begin{aligned}
\mathsf{SkTerms}^{\beta}(\bot) &= \emptyset \\
\mathsf{SkTerms}^{\beta}(\top) &= \emptyset \\
\mathsf{SkTerms}^{\beta}(P(t_1, \ldots, t_n)) &= \emptyset \\
\mathsf{SkTerms}^{+}(\neg E) &= \mathsf{SkTerms}^{-}(E) \\
\mathsf{SkTerms}^{-}(\neg E) &= \mathsf{SkTerms}^{+}(E) \\
\mathsf{SkTerms}^{\beta}(E_1 \vee E_2) &= \mathsf{SkTerms}^{\beta}(E_1) \cup \mathsf{SkTerms}^{\beta}(E_2) \\
\mathsf{SkTerms}^{\beta}(E_1 \wedge E_2) &= \mathsf{SkTerms}^{\beta}(E_1) \cup \mathsf{SkTerms}^{\beta}(E_2) \\
\mathsf{SkTerms}^{+}(E_1 \to E_2) &= \mathsf{SkTerms}^{-}(E_1) \cup \mathsf{SkTerms}^{+}(E_2) \\
\mathsf{SkTerms}^{-}(E_1 \to E_2) &= \mathsf{SkTerms}^{+}(E_1) \cup \mathsf{SkTerms}^{-}(E_2) \\
\mathsf{SkTerms}^{+}(\exists x A +^{t_1} E_1 \cdots +^{t_n} E_n) &= \mathsf{SkTerms}^{+}(E_1) \cup \cdots \cup \mathsf{SkTerms}^{+}(E_n) \\
\mathsf{SkTerms}^{-}(\exists x A +^{f(t_1, \ldots, t_n)} E) &= \{f(t_1, \ldots, t_n)\} \cup \mathsf{SkTerms}^{-}(E) \\
\mathsf{SkTerms}^{+}(\forall x A +^{f(t_1, \ldots, t_n)} E) &= \{f(t_1, \ldots, t_n)\} \cup \mathsf{SkTerms}^{+}(E) \\
\mathsf{SkTerms}^{-}(\forall x A +^{t_1} E_1 \cdots +^{t_n} E_n) &= \mathsf{SkTerms}^{-}(E_1) \cup \cdots \cup \mathsf{SkTerms}^{-}(E_n)
\end{aligned}$$

For a expansion sequent $S' = E_1, \ldots, E_n \vdash F_1, \ldots F_m$ its Skolem terms $\mathsf{SkTerms}(S) = \mathsf{SkTerms}^{-}(E_1) \cup \cdots \cup \mathsf{SkTerms}^{-}(E_n) \cup \mathsf{SkTerms}^{+}(F_1) \cup \cdots \cup \mathsf{SkTerms}^{+}(F_n)$

Definition 11. The union of two expansions $E_1 \cup E_2$ is a partial operator defined as [7, Definition 7]

- If $E_1 = \bot$ or $E_1 = \top$ then $E_1 \cup E_2 = E_2$
- If $E_2 = \bot$ or $E_2 = \top$ then $E_1 \cup E_2 = E_1$
- If $E_1 = \neg E_1'$ and $E_2 = \neg E_2'$ then $E_1 \cup E_2 = \neg(E_1' \cup E_2')$
- If $E_1 = E_1' \vee E_1''$ and $E_2 = E_2' \vee E_2''$ then $E_1 \cup E_2 = (E_1' \cup E_2') \vee (E_1'' \cup E_2'')$
- If $E_1 = E_1' \wedge E_1''$ and $E_2 = E_2' \wedge E_2''$ then $E_1 \cup E_2 = (E_1' \cup E_2') \wedge (E_1'' \cup E_2'')$
- If $E_1 = E_1' \to E_1''$ and $E_2 = E_2' \to E_2''$ then $E_1 \cup E_2 = (E_1' \cup E_2') \to (E_1'' \cup E_2'')$
- If $E_1 = \exists x A +^{t_1} E_{1,1} \cdots +^{t_k} E_{1,k} +^{s_1} F_1 \cdots +^{s_l} F_l$
  and $E_2 = \exists x A +^{t_1} E_{2,1} \cdots +^{t_k} E_{2,k} +^{r_1} G_1 \cdots +^{r_m} G_m$ and
  $\{s_1, \ldots, s_l\} \cap \{r_1, \ldots, r_m\} = \emptyset$ then
  $E_1 \cup E_2 = \exists x A +^{t_1} (E_{1,1} \cup E_{2,1}) \cdots +^{t_k} (E_{1,k} \cup E_{2,k}) +^{s_1} F_1 \cdots +^{s_l} F_l +^{r_1} G_1 \cdots +^{r_m} G_m$
- If $E_1 = \forall x A +^{t_1} E_{1,1} \cdots +^{t_k} E_{1,k} +^{s_1} F_1 \cdots +^{s_l} F_l$
  and $E_2 = \forall x A +^{t_1} E_{2,1} \cdots +^{t_k} E_{2,k} +^{r_1} G_1 \cdots +^{r_m} G_m$ and
  $\{s_1, \ldots, s_l\} \cap \{r_1, \ldots, r_m\} = \emptyset$ then
  $E_1 \cup E_2 = \forall x A +^{t_1} (E_{1,1} \cup E_{2,1}) \cdots +^{t_k} (E_{1,k} \cup E_{2,k}) +^{s_1} F_1 \cdots +^{s_l} F_l +^{r_1} G_1 \cdots +^{r_m} G_m$
- For all other cases $E_1 \cup E_2$ is undefined

For sequences of expansions $\Gamma_1 = E_1, \ldots, E_n$ and $\Gamma_2 = F_1, \ldots, F_n$ their union is $\Gamma_1 \cup \Gamma_2 = E_1 \cup F_1, \ldots, E_n \cup F_n$

3.2 Example of an expansion

The following expansion is an expansion of the drinker paradox.

$$\begin{aligned} &\exists x(P(x) \to \forall y P(y)) \\ &\qquad +^{f_0(a)} (P(f_0(a)) \to \bot) \\ &\qquad +^{a} \Big(\top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\Big) \end{aligned}$$

From this expansion a proof of the Skolemized drinker paradox can be constructed, x has to be substituted by a and $f_0(a)$, and y by $f_0(a)$.

Chapter 4

Structural Skolemization

Skolemization (and its dual Herbrandization) comes in two forms, structural and prefix Skolemization. In structural Skolemization the universal quantifiers are in place replaced by Skolem functions. Prefix Skolemization first puts the formula in prenex normal form and then replaces the quantifiers by the same method as structural Skolemization.

Structural Skolemization has the advantage that it is unique up to renaming of the Skolem terms. Prefix is not unique, since there is in general no unique prenex normal form.

Also structural Skolemization has fewer arguments in the Skolem terms as prenex normal form moves quantifiers outwards. [3]

Structural Skolemization removes all strong quantifiers, $\forall$ in positive context and $\exists$ in negative context. The number of strong quantifiers in a formula A is denoted as $\mathsf{qocc}_{+}(A)$.

$$
\begin{aligned}
\mathsf{qocc}_{\beta}(P(t_1,\dots,t_n)) &= 0\\
\mathsf{qocc}_{+}(\neg A) &= \mathsf{qocc}_{-}(A)\\
\mathsf{qocc}_{-}(\neg A) &= \mathsf{qocc}_{+}(A)\\
\mathsf{qocc}_{\beta}(A_1 \vee A_2) &= \mathsf{qocc}_{\beta}(A_1) + \mathsf{qocc}_{\beta}(A_2)\\
\mathsf{qocc}_{\beta}(A_1 \wedge A_2) &= \mathsf{qocc}_{\beta}(A_1) + \mathsf{qocc}_{\beta}(A_2)\\
\mathsf{qocc}_{+}(A_1 \to A_2) &= \mathsf{qocc}_{-}(A_1) + \mathsf{qocc}_{+}(A_2)\\
\mathsf{qocc}_{-}(A_1 \to A_2) &= \mathsf{qocc}_{+}(A_1) + \mathsf{qocc}_{-}(A_2)\\
\mathsf{qocc}_{+}(\exists x A) &= \mathsf{qocc}_{+}(A)\\
\mathsf{qocc}_{-}(\exists x A) &= 1 + \mathsf{qocc}_{-}(A)\\
\mathsf{qocc}_{+}(\forall x A) &= 1 + \mathsf{qocc}_{+}(A)\\
\mathsf{qocc}_{-}(\forall x A) &= \mathsf{qocc}_{-}(A)
\end{aligned}
$$

For a formula A its structural Skolemization $A' = \mathsf{sk}^{+,0}_{\langle\rangle}(A)$ in positive context and $A' = \mathsf{sk}^{-,0}_{\langle\rangle}(A)$ in negative context.

$$
\begin{aligned}
\mathsf{sk}^{\beta,n}_{\mu}(P(t_1,\dots,t_m)) &= P(t_1,\dots,t_m)\\
\mathsf{sk}^{+,n}_{\mu}(\neg A) &= \neg\mathsf{sk}^{-,n}_{\mu}(A)\\
\mathsf{sk}^{-,n}_{\mu}(\neg A) &= \neg\mathsf{sk}^{+,n}_{\mu}(A)\\
\mathsf{sk}^{\beta,n}_{\mu}(A_1 \vee A_2) &= \mathsf{sk}^{\beta,n}_{\mu}(A_1) \vee \mathsf{sk}^{\beta,n'}_{\mu}(A_2) \text{ when } n' = n + \mathsf{qocc}_{\beta}(A_1)\\
\mathsf{sk}^{\beta,n}_{\mu}(A_1 \wedge A_2) &= \mathsf{sk}^{\beta,n}_{\mu}(A_1) \wedge \mathsf{sk}^{\beta,n'}_{\mu}(A_2) \text{ when } n' = n + \mathsf{qocc}_{\beta}(A_1)\\
\mathsf{sk}^{+,n}_{\mu}(A_1 \to A_2) &= \mathsf{sk}^{-,n}_{\mu}(A_1) \to \mathsf{sk}^{+,n'}_{\mu}(A_2) \text{ when } n' = n + \mathsf{qocc}_{-}(A_1)\\
\mathsf{sk}^{-,n}_{\mu}(A_1 \to A_2) &= \mathsf{sk}^{+,n}_{\mu}(A_1) \to \mathsf{sk}^{-,n'}_{\mu}(A_2) \text{ when } n' = n + \mathsf{qocc}_{+}(A_1)\\
\mathsf{sk}^{+,n}_{\mu_1,\dots,\mu_l}(\exists x A) &= \exists x\, \mathsf{sk}^{+,n}_{\mu_1,\dots,\mu_l,x}(A)\\
\mathsf{sk}^{-,n}_{\mu_1,\dots,\mu_l}(\exists x A) &= \mathsf{sk}^{+,n'}_{\mu_1,\dots,\mu_l}(A\,[x := f_n(\mu_1,\dots,\mu_l)]) \text{ when } n' = n+1\\
\mathsf{sk}^{+,n}_{\mu_1,\dots,\mu_l}(\forall x A) &= \mathsf{sk}^{+,n'}_{\mu_1,\dots,\mu_l}(A\,[x := f_n(\mu_1,\dots,\mu_l)]) \text{ when } n' = n+1\\
\mathsf{sk}^{-,n}_{\mu_1,\dots,\mu_l}(\forall x A) &= \forall x\, \mathsf{sk}^{+,n}_{\mu_1,\dots,\mu_l,x}(A)
\end{aligned}
$$

The structural Skolemization of a sequent $S = A_1,\dots,A_n \vdash B_1,\dots,B_m$ is $A'_1,\dots,A'_n \vdash B'_1,\dots,B_m$

- with $B'_1,\dots,B'_m = \mathsf{sk}^{+,0}_{\langle\rangle}(B_1 \vee \cdots \vee B_m)$ if $n = 0$
- with $A'_1,\dots,A'_n = \mathsf{sk}^{-,0}_{\langle\rangle}(A_1 \wedge \cdots \wedge A_n)$ if $m = 0$
- with $A'_1,\dots,A'_n \to B'_1,\dots,B'_m = \mathsf{sk}^{+,0}_{\langle\rangle}(A_1 \wedge \cdots \wedge A_n \to B_1 \vee \cdots \vee B_m)$ if $n > 0$ and $m > 0$

4.1 Example of Skolemization

The structural Skolemization S' of the drinker paradox, $S = \vdash \exists x(P(x) \to \forall y P(y))$

$$
\begin{aligned}
\mathsf{sk}^{+,0}_{\langle\rangle}(\exists x(P(x) \to \forall y P(y))) &=\\
\exists x\, \mathsf{sk}^{+,0}_{\langle x\rangle}(P(x) \to \forall y P(y)) &=\\
\exists x \left(\mathsf{sk}^{-,0}_{\langle x\rangle}(P(x)) \to \mathsf{sk}^{+,0}_{\langle x\rangle}(\forall y P(y))\right) &=\\
\exists x \left(P(x) \to \mathsf{sk}^{+,0}_{\langle x\rangle}(\forall y P(y))\right) &=\\
\exists x \left(P(x) \to \mathsf{sk}^{+,1}_{\langle x\rangle}(P(f_0(x))\right) &=\\
\exists x\ (P(x) \to P(f_0(x)))
\end{aligned}
$$

$S' = \vdash \exists x\, P(x) \to P(f_0(x))$

Chapter 5

Proof deskolemization

In this section the actual algorithm is presented. The four steps of the deskolemization process are in the next sections. Each section is followed by example on the drinker paradox.

5.1 Expansion extraction

The first step in the deskolemization is the extraction of the tautological expansion from the proof in LK. The resulting expansion represents the proof in a simpler form. The extraction function e is recursively defined [7, Lemma 2].

For an axiom

$$\mathsf{e}\left(\frac{}{A, \Gamma \vdash \Delta, A}\right) = A, \top, \ldots, \top \vdash \bot, \ldots, \bot, A$$

Note that formulas that are irrelevant for the axiom inference are substituted by $\top$ and $\bot$. This reduces the size of the resulting expansion and the complexity of the proof deskolemization process. After the deskolemization step $\top$ and $\bot$ are substituted by the correct formula's in section 5.7.

For the inference rules

$$\mathsf{e}\left(\frac{\frac{\phi}{\Gamma \vdash \Delta, A}}{\neg A, \Gamma \vdash \Delta}\,\neg\mathrm{L}\right) = \neg E, \Gamma' \vdash \Delta'$$

$$\text{where } \Gamma' \vdash \Delta', E = \mathsf{e}\left(\frac{\phi}{\Gamma \vdash \Delta, A}\right)$$

$$\mathsf{e}\left(\frac{\frac{\phi}{A, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta, \neg A}\,\neg\mathrm{R}\right) = \Gamma' \vdash \Delta', \neg E$$

$$\text{where } E, \Gamma' \vdash \Delta' = \mathsf{e}\left(\frac{\phi}{A, \Gamma \vdash \Delta}\right)$$

$$\mathsf{e}\left(\dfrac{\dfrac{\phi_1}{A,\Gamma\vdash\Delta}\qquad\dfrac{\phi_2}{B,\Gamma\vdash\Delta}}{A\lor B,\Gamma\vdash\Delta}\lor\mathrm{L}\right) = E_1\lor E_2,\Gamma'_1\cup\Gamma'_2\vdash\Delta'_1\cup\Delta'_2$$

$$\text{where } E_1,\Gamma'_1\vdash\Delta'_1 = \mathsf{e}\left(\dfrac{\phi_1}{A,\Gamma\vdash\Delta}\right) \text{ and}$$

$$E_2,\Gamma'_2\vdash\Delta'_2 = \mathsf{e}\left(\dfrac{\phi_2}{B,\Gamma\vdash\Delta}\right)$$

$$\mathsf{e}\left(\dfrac{\dfrac{\phi}{\Gamma\vdash\Delta,A,B}}{\Gamma\vdash\Delta,A\lor B}\lor\mathrm{R}\right) = \Gamma'\vdash\Delta',E_1\lor E_2$$

$$\text{where } \Gamma'\vdash\Delta',E_1,E_2 = \mathsf{e}\left(\dfrac{\phi}{\Gamma\vdash\Delta,A,B}\right)$$

$$\mathsf{e}\left(\dfrac{\dfrac{\phi}{A,B,\Gamma\vdash\Delta}}{A\land B,\Gamma\vdash\Delta}\land\mathrm{L}\right) = E_1\land E_2,\Gamma'\vdash\Delta'$$

$$\text{where } E_1,E_2,\Gamma'\vdash\Delta' = \mathsf{e}\left(\dfrac{\phi}{A,B,\Gamma\vdash\Delta}\right)$$

$$\mathsf{e}\left(\dfrac{\dfrac{\phi_1}{\Gamma\vdash\Delta,A}\qquad\dfrac{\phi_2}{\Gamma\vdash\Delta,B}}{\Gamma\vdash\Delta,A\land B}\land\mathrm{R}\right) = \Gamma'_1\cup\Gamma'_2\vdash\Delta'_1\cup\Delta'_2,E_1\land E_2$$

$$\text{where } \Gamma'_1\vdash\Delta'_1,E_1 = \mathsf{e}\left(\dfrac{\phi_1}{\Gamma\vdash\Delta,A}\right) \text{ and}$$

$$\Gamma'_2\vdash\Delta'_2,E_2 = \mathsf{e}\left(\dfrac{\phi_2}{\Gamma\vdash\Delta,B}\right)$$

$$\mathsf{e}\left(\dfrac{\dfrac{\phi_1}{\Gamma\vdash\Delta,A}\qquad\dfrac{\phi_2}{B,\Gamma\vdash\Delta}}{A\to B,\Gamma\vdash\Delta}\to\mathrm{L}\right) = E_1\to E_2,\Gamma'_1\cup\Gamma'_2\vdash\Delta'_1\cup\Delta'_2$$

$$\text{where } \Gamma'_1\vdash\Delta'_1,E_1 = \mathsf{e}\left(\dfrac{\phi_1}{\Gamma\vdash\Delta,A}\right) \text{ and}$$

$$E_2,\Gamma'_2\vdash\Delta'_2 = \mathsf{e}\left(\dfrac{\phi_2}{B,\Gamma\vdash\Delta}\right)$$

$$\mathsf{e}\left(\dfrac{\dfrac{\phi}{A,\Gamma\vdash\Delta,B}}{\Gamma\vdash\Delta,A\to B}\to\mathrm{R}\right) = \Gamma'\vdash\Delta',E_1\to E_2$$

$$\text{where } E_1,\Gamma'\vdash\Delta',E_2 = \mathsf{e}\left(\dfrac{\phi}{A,\Gamma\vdash\Delta,B}\right)$$

For $\exists\mathrm{L}_y$ and $\forall\mathrm{R}_y$ the function e is undefined as these inferences can not occur in a proof of a Skolemized formula.

$$\mathsf{e}\left(\cfrac{\cfrac{\phi}{A\left[x:=y\right],\Gamma\vdash\Delta}}{\exists xA,\Gamma\vdash\Delta}\,\exists\mathrm{L}_y\right)\text{is undefined}$$

$$\mathsf{e}\left(\cfrac{\cfrac{\phi}{\Gamma\vdash\Delta,\exists xA,A\left[x:=t\right]}}{\Gamma\vdash\Delta,\exists xA}\,\exists\mathrm{R}_t\right)=\Gamma'\vdash\Delta',E_1\cup\left(\exists xA+^t E_2\right)$$

$$\text{where }\Gamma'\vdash\Delta',E_1,E_2=\mathsf{e}\left(\cfrac{\phi}{\Gamma\vdash\Delta,\exists xA,A\left[x:=t\right]}\right)$$

$$\mathsf{e}\left(\cfrac{\cfrac{\phi}{A\left[x:=t\right],\forall xA,\Gamma\vdash\Delta}}{\forall xA,\Gamma\vdash\Delta}\,\forall\mathrm{L}_t\right)=E_1\cup\left(\forall xA+^t E_2\right),\Gamma'\vdash\Delta'$$

$$\text{where }E_2,E_1,\Gamma'\vdash\Delta'=\mathsf{e}\left(\cfrac{\phi}{A\left[x:=t\right],\forall xA,\Gamma\vdash\Delta}\right)$$

$$\mathsf{e}\left(\cfrac{\cfrac{\phi}{\Gamma\vdash\Delta,A\left[x:=y\right]}}{\Gamma\vdash\Delta,\forall xA}\,\forall\mathrm{R}_y\right)\text{is undefined}$$

$$\mathsf{e}\left(\cfrac{\cfrac{\phi}{A,\Gamma_1,\Gamma_2\vdash\Delta}}{\Gamma_1,A,\Gamma_2\vdash\Delta}\,\mathrm{PL}\right)=\Gamma_1',E,\Gamma_2'\vdash\Delta'$$

$$\text{where }E,\Gamma_1',\Gamma_2'\vdash\Delta'=\mathsf{e}\left(\cfrac{\phi}{A,\Gamma_1,\Gamma_2\vdash\Delta}\right)$$

$$\mathsf{e}\left(\cfrac{\cfrac{\phi}{\Gamma\vdash\Delta_1,\Delta_2,A}}{\Gamma\vdash\Delta_1,A,\Delta_2}\,\mathrm{PR}\right)=\Gamma'\vdash\Delta_1',E,\Delta_2'$$

$$\text{where }\Gamma'\vdash\Delta_1',\Delta_2',E=\mathsf{e}\left(\cfrac{\phi}{\Gamma\vdash\Delta_1,\Delta_2,A}\right)$$

5.2 Example of expansion extraction

Applying the extraction e on the proof π from 2.1 gives

$$\mathsf{e}\left(\overline{P(f_0(a)),P(a)\vdash\exists x(P(x)\to P(f_0(x))),P(f_0(f_0(a))),P(f_0(a))}\right)=P(f_0(a)),\top\vdash\bot,\bot,P(f_0(a))$$

$$\mathsf{e}\left(\cfrac{\phi}{P(f_0(a)),P(a)\vdash P(f_0(a)),\exists x(P(x)\to P(f_0(x))),P(f_0(f_0(a)))}\right)=P(f_0(a)),\top\vdash P(f_0(a)),\bot,\bot$$

$$\mathsf{e}\left(\cfrac{\phi}{P(a)\vdash P(f_0(a)),\exists x(P(x)\to P(f_0(x))),P(f_0(a))\to P(f_0(f_0(a)))}\right)=$$
$$\top\vdash P(f_0(a)),\bot,P(f_0(a))\to\bot$$

$$\mathsf{e}\left(\cfrac{\phi}{P(a)\vdash P(f_0(a)),\exists x(P(x)\to P(f_0(x)))}\right)=\Gamma'\vdash\Delta',E_1\cup\left(\exists xA+^t E_2\right)$$

$$\begin{aligned}\text{where }\Gamma'\vdash\Delta',E_1,E_2&=\top\vdash P(f_0(a)),\bot,P(f_0(a))\to\bot\\&=\top\vdash P(f_0(a)),\bot\cup\left(\exists x(P(x)\to P(f_0(x)))+^{f_0(a)}P(f_0(a))\to\bot\right)\\&=\top\vdash P(f_0(a)),\left(\exists x(P(x)\to P(f_0(x)))+^{f_0(a)}P(f_0(a))\to\bot\right)\end{aligned}$$

$$\mathsf{e}\Big(\frac{\phi}{P(a) \vdash \exists x(P(x) \to P(f_0(x))), P(f_0(a))}\Big) = \top \vdash \big(\exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot\big), P(f_0(a))$$

$$\mathsf{e}\Big(\frac{\phi}{\vdash \exists x(P(x) \to P(f_0(x))), P(a) \to P(f_0(a))}\Big) = \vdash \big(\exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot\big), \top \to P(f_0(a))$$

$$\begin{aligned}
\mathsf{e}\Big(\frac{\phi}{\vdash \exists x(P(x) \to P(f_0(x)))}\Big) &= \Gamma' \vdash \Delta', E_1 \cup \big(\exists x A +^{t} E_2\big) \\
\text{where } \Gamma' \vdash \Delta', E_1, E_2 &= \vdash \big(\exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot\big), \top \to P(f_0(a)) \\
&= \vdash \big(\exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot\big) \cup \\
&\qquad (\exists x(P(x) \to P(f_0(x))) +^{a} \top \to P(f_0(a))) \\
&= \vdash \exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot +^{a} \top \to P(f_0(a))
\end{aligned}$$

Thus the Skolemized expansion is $E' = \vdash \exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot +^{a} \top \to P(f_0(a))$.

5.3 Skolemized expansion to deskolemized expansion

Now that the proof is in expansion form, it can be easily deskolemized. The strong $\exists$ and $\forall$ quantifiers, removed by the Skolemization, have to be restored at the correct place. The Skolem terms corresponding to these strong quantifiers are added as a selected term. The selected term is $f(s_1, \ldots, s_n)$ where f is the Skolem functions for the strong quantifier and $s_1, \ldots, s_n$ are the selected terms in its scope. The process is similar to that of the Skolemization. In the Skolemization, sk, the quantified variables $\forall x$ are recorded to create the correct function. During the deskolemization here the actual selected terms, s_i, are kept to create the correct Skolem term.

The selected terms $+^{t}$ will still contain Skolem functions, these will be replaced by fresh terms in section 5.7. [7, Lemma 3]

$$\begin{aligned}
\mathsf{desk}^{\beta,m}_{s_1,\ldots,s_n}(A, \top) &= \top \\
\mathsf{desk}^{\beta,m}_{s_1,\ldots,s_n}(A, \bot) &= \bot \\
\mathsf{desk}^{\beta,m}_{s_1,\ldots,s_n}(P(t_1, \ldots, t_n), P(t'_1, \ldots, t'_n)) &= P(t'_1, \ldots, t'_n) \\
\mathsf{desk}^{+,m}_{s_1,\ldots,s_n}(\neg A, \neg E) &= \neg \mathsf{desk}^{-,m}_{s_1,\ldots,s_n}(A, E) \\
\mathsf{desk}^{-,m}_{s_1,\ldots,s_n}(\neg A, \neg E) &= \neg \mathsf{desk}^{+,m}_{s_1,\ldots,s_n}(A, E) \\
\mathsf{desk}^{\beta,m}_{s_1,\ldots,s_n}(A_1 \lor A_2, E_1 \lor E_2) &= \mathsf{desk}^{\beta,m}_{s_1,\ldots,s_n}(A_1, E_1) \lor \mathsf{desk}^{\beta,m'}_{s_1,\ldots,s_n}(A_2, E_2) \\
&\quad \text{when } m' = m + \mathsf{qocc}_{\beta}(A_1) \\
\mathsf{desk}^{\beta,m}_{s_1,\ldots,s_n}(A_1 \land A_2, E_1 \land E_2) &= \mathsf{desk}^{\beta,m}_{s_1,\ldots,s_n}(A_1, E_1) \land \mathsf{desk}^{\beta,m'}_{s_1,\ldots,s_n}(A_2, E_2) \\
&\quad \text{when } m' = m + \mathsf{qocc}_{\beta}(A_1) \\
\mathsf{desk}^{+,m}_{s_1,\ldots,s_n}(A_1 \to A_2, E_1 \to E_2) &= \mathsf{desk}^{-,m}_{s_1,\ldots,s_n}(A_1, E_1) \to \mathsf{desk}^{+,m'}_{s_1,\ldots,s_n}(A_2, E_2) \\
&\quad \text{when } m' = m + \mathsf{qocc}_{+}(A_1) \\
\mathsf{desk}^{-,m}_{s_1,\ldots,s_n}(A_1 \to A_2, E_1 \to E_2) &= \mathsf{desk}^{+,m}_{s_1,\ldots,s_n}(A_1, E_1) \to \mathsf{desk}^{-,m'}_{s_1,\ldots,s_n}(A_2, E_2) \\
&\quad \text{when } m' = m + \mathsf{qocc}_{-}(A_1) \\
\mathsf{desk}^{+,m}_{s_1,\ldots,s_n}(\exists x A, \exists x A' +^{t_1} E_1 \cdots +^{t_n} E_n) &= \exists x A +^{t_1} \mathsf{desk}^{+,m}_{s_1,\ldots,s_n,t_1}(A[x := t_1], E_1) \cdots +^{t_n} \mathsf{desk}^{+,m}_{s_1,\ldots,s_n,t_n}(A[x := t_n], E_n) \\
\mathsf{desk}^{-,m}_{s_1,\ldots,s_n}(\exists x A, E) &= \exists x A +^{f_m(s_1,\ldots,s_n)} \mathsf{desk}^{-,m'}_{s_1,\ldots,s_n}(A[x := f_m(s_1, \ldots, s_n)], E) \\
&\quad \text{when } m' = m + 1 \\
\mathsf{desk}^{+,m}_{s_1,\ldots,s_n}(\forall x A, E) &= \forall x A +^{f_m(s_1,\ldots,s_n)} \mathsf{desk}^{+,m'}_{s_1,\ldots,s_n}(A[x := f_m(s_1, \ldots, s_n)], E) \\
&\quad \text{when } m' = m + 1 \\
\mathsf{desk}^{-,m}_{s_1,\ldots,s_n}(\forall x A, \forall x A' +^{t_1} E_1 \cdots +^{t_n} E_n) &= \forall x A +^{t_1} \mathsf{desk}^{-,m}_{s_1,\ldots,s_n,t_1}(A[x := t_1], E_1) \cdots +^{t_n} \mathsf{desk}^{-,m}_{s_1,\ldots,s_n,t_n}(A[x := t_n], E_n)
\end{aligned}$$

The deskolemization of an expansion sequent $E_1, \dots, E_n \vdash F_1, \dots, F_m$ for a sequent $S = A_1, \dots, A_n \vdash B_1, \dots, B_m$ is $\mathsf{desk}^-(A_1, E_1), \dots, \mathsf{desk}^-(A_n, E_n) \vdash \mathsf{desk}^+(B_1, F_1), \dots, \mathsf{desk}^+(B_m, F_m)$

5.4 Example of expansion deskolemization

The deskolemized expansion E is

$$\begin{aligned}\mathsf{desk}^+(\exists x(P(x) \to \forall y P(y)), \exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot +^a \top \to P(f_0(a))) =\\ \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} \mathsf{desk}^+(P(f_0(a)) \to \forall y P(y)), P(f_0(a)) \to \bot)\\ +^a \mathsf{desk}^+(P(a) \to \forall y P(y)), \top \to P(f_0(a)))\end{aligned}$$

$$\begin{aligned}\mathsf{desk}^+(P(f_0(a)) \to \forall y P(y)), P(f_0(a)) \to \bot) =\\ \mathsf{desk}^-(P(f_0(a)), P(f_0(a))) \to \mathsf{desk}^+(\forall y P(y), \bot) = P(f_0(a)) \to \bot\end{aligned}$$

$$\begin{aligned}\mathsf{desk}^+(P(a) \to \forall y P(y)), \top \to P(f_0(a))) =\\ \mathsf{desk}^-(P(a), \top) \to \mathsf{desk}^+(\forall y P(y), P(f_0(a))) =\\ \top \to \forall y P(y) +^{f_0(a))} \mathsf{desk}^+(P(f_0(a)), P(f_0(a))) =\\ \top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\end{aligned}$$

$$\begin{aligned}\mathsf{desk}^+(\exists x(P(x) \to \forall y P(y)), \exists x(P(x) \to P(f_0(x))) +^{f_0(a)} P(f_0(a)) \to \bot +^a \top \to P(f_0(a))) =\\ \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot) +^a \left(\top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\right)\end{aligned}$$

5.5 Proof in LK^{E} from deskolemized expansion

In this section the proof is reconstructed from the expansion. The key point is that the selected terms in the expansions do not have an explicit order. For the $\exists \mathrm{R}_t$ and $\forall \mathrm{L}_t$ rules the terms may not be in the $\mathsf{SkTerms}()$. This ensures the correct ordering, and the eigenvariable restrictions for the inferences rules.[7, Lemma 6]

First, the definition of two helper functions that handle the correct permutation inferences.

$$\begin{aligned}\mathsf{permL}_i((E_i, \Gamma \vdash \Delta), \pi) &= \pi\\ \mathsf{permL}_i((\Gamma_1, E_i, \Gamma_2 \vdash \Delta), \pi) &= \frac{\pi}{\Gamma_1, E_i, \Gamma_2 \vdash \Delta}\mathrm{PL}_i\\ \mathsf{permR}_i((\Gamma \vdash \Delta, E_i), \pi) &= \pi\\ \mathsf{permR}_i((\Gamma \vdash \Delta_1, E_i, \Delta_2), \pi) &= \frac{\pi}{\Gamma \vdash \Delta_1, E_i, \Delta_2}\mathrm{PR}_i\end{aligned}$$

The actual reconstruction of the proof is done by Pr. It takes an expansion sequent as input and produces a proof in LK^{E} by recursion. Pr is not uniquely defined, yet every possible branch will result in a correct proof. For example $A \wedge B \vdash A \vee B$ matches the cases for $E_1 \wedge E_2$ and $E_1 \vee E_2$.

$$
\begin{aligned}
\mathsf{Pr}\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta_1, E_j, \Delta_2\right) &= \mathsf{permL}_i\Bigg(\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta_1, E_j, \Delta_2\right), \\
&\qquad \mathsf{permR}_j\left(\left(E_i, \Gamma_1, \Gamma_2 \vdash \Delta_1, E_j, \Delta_2\right), \frac{}{E_i, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, E_j}\right)\Bigg) \\
&\text{when } E = E_i = E_j \\
\mathsf{Pr}\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right) &= \mathsf{permL}_i\left(\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right), \frac{\mathsf{Pr}\left(\Gamma_1, \Gamma_2 \vdash \Delta, E\right)}{E_i, \Gamma_1, \Gamma_2 \vdash \Delta}\,\neg\mathrm{L}\right) \\
&\text{when } E_i = \neg E \\
\mathsf{Pr}\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right) &= \mathsf{permL}_i\left(\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right), \frac{\mathsf{Pr}\left(E, \Gamma \vdash \Delta_1, \Delta_2\right)}{\Gamma \vdash \Delta_1, \Delta_2, E_i}\,\neg\mathrm{R}\right) \\
&\text{when } E_i = \neg E \\
\mathsf{Pr}\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right) &= \mathsf{permL}_i\left(\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right), \frac{\mathsf{Pr}\left(E_1, \Gamma_1, \Gamma_2 \vdash \Delta\right) \qquad \mathsf{Pr}\left(E_2, \Gamma_1, \Gamma_2 \vdash \Delta\right)}{E_i, \Gamma_1, \Gamma_2 \vdash \Delta}\,\vee\mathrm{L}\right) \\
&\text{when } E_i = E_1 \vee E_2 \\
\mathsf{Pr}\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right) &= \mathsf{permL}_i\left(\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right), \frac{\mathsf{Pr}\left(\Gamma \vdash \Delta_1, \Delta_2, E_1, E_2\right)}{\Gamma \vdash \Delta_1, \Delta_2, E_i}\,\vee\mathrm{R}\right) \\
&\text{when } E_i = E_1 \vee E_2 \\
\mathsf{Pr}\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right) &= \mathsf{permL}_i\left(\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right), \frac{\mathsf{Pr}\left(E_1, E_2, \Gamma_1, \Gamma_2 \vdash \Delta\right)}{E_i, \Gamma_1, \Gamma_2 \vdash \Delta}\,\wedge\mathrm{L}\right) \\
&\text{when } E_i = E_1 \wedge E_2 \\
\mathsf{Pr}\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right) &= \mathsf{permL}_i\left(\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right), \frac{\mathsf{Pr}\left(\Gamma \vdash \Delta_1, \Delta_2, E_1\right) \qquad \mathsf{Pr}\left(\Gamma \vdash \Delta_1, \Delta_2, E_2\right)}{\Gamma \vdash \Delta_1, \Delta_2, E_i}\,\wedge\mathrm{R}\right) \\
&\text{when } E_i = E_1 \wedge E_2 \\
\mathsf{Pr}\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right) &= \mathsf{permL}_i\left(\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right), \frac{\mathsf{Pr}\left(\Gamma_1, \Gamma_2 \vdash \Delta, E_1\right) \qquad \mathsf{Pr}\left(E_2, \Gamma_1, \Gamma_2 \vdash \Delta\right)}{E_i, \Gamma_1, \Gamma_2 \vdash \Delta}\,{\rightarrow}\mathrm{L}\right) \\
&\text{when } E_i = E_1 \rightarrow E_2 \\
\mathsf{Pr}\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right) &= \mathsf{permL}_i\left(\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right), \frac{\mathsf{Pr}\left(E_1, \Gamma \vdash \Delta_1, \Delta_2, E_2\right)}{\Gamma \vdash \Delta_1, \Delta_2, E_i}\,{\rightarrow}\mathrm{R}\right) \\
&\text{when } E_i = E_1 \rightarrow E_2 \\
\mathsf{Pr}\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right) &= \mathsf{permL}_i\left(\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right), \frac{\mathsf{Pr}\left(E, \Gamma_1, \Gamma_2 \vdash \Delta\right)}{E_i, \Gamma_1, \Gamma_2 \vdash \Delta}\,\exists\mathrm{L}_{f(\bar{t})}\right) \\
&\text{when } E_i = \exists x A +^{f(\bar{t})} E \\
\mathsf{Pr}\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right) &= \mathsf{permL}_i\left(\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right), \frac{\mathsf{Pr}\left(\Gamma \vdash \Delta_1, \Delta_2, \exists x A + \omega, E\right)}{\Gamma \vdash \Delta_1, \Delta_2, E_i}\,\exists\mathrm{R}_t\right) \\
&\text{when } E_i = \exists x A +^{t} E + \omega \\
&\text{and } t \notin \mathsf{SkTerms}(\Gamma \vdash \Delta_1, E_i, \Delta_2) \\
\mathsf{Pr}\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right) &= \mathsf{permL}_i\left(\left(\Gamma_1, E_i, \Gamma_2 \vdash \Delta\right), \frac{\mathsf{Pr}\left(E, \forall x A + \omega, \Gamma \vdash \Delta\right)}{E_i, \Gamma_1, \Gamma_2 \vdash \Delta}\,\forall\mathrm{L}_t\right) \\
&\text{when } E_i = \forall x A +^{t} E + \omega \\
&\text{and } t \notin \mathsf{SkTerms}(\Gamma_1, E_i, \Gamma_2 \vdash \Delta) \\
\mathsf{Pr}\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right) &= \mathsf{permL}_i\left(\left(\Gamma \vdash \Delta_1, E_i, \Delta_2\right), \frac{\mathsf{Pr}\left(\Gamma, \vdash \Delta_1, \Delta_2, E\right)}{\Gamma \vdash \Delta_1, \Delta_2, E_i}\,\forall\mathrm{R}_{f(\bar{t})}\right) \\
&\text{when } E_i = \forall x A +^{f(\bar{t})} E
\end{aligned}
$$

5.6 Example of proof construction in $\mathbf{LK^E}$

From the previous section we have $\vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot) +^{a} \big(\top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\big)$ to construct the proof we simply apply Pr.

$$\mathsf{Pr}\left(\vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot) +^{a} \left(\top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\right)\right)$$

Only the rule $\exists$ at the right matches, with $t = f_0(a)$ or $t = a$. However $f_0(a) \in \mathsf{SkTerms}(S) = f_0(a)$ since $\mathsf{SkTerms}^{+}(\forall y P(y) +^{f_0(a))} P(f_0(a))) = f_0(a)$. Thus the first step is $\exists \mathrm{R}_a$.

$$\dfrac{\mathsf{Pr}\left(\vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot)\,, \top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\right)}{\vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot) +^{a} \left(\top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\right)}\,\exists \mathrm{R}_a$$

The next possible step is $\to$R, since $\exists \mathrm{R}_{f_0(a)}$ is still in $\mathsf{SkTerms}()$ and can not be applied. The final result of Pr is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(f_0(a)), \top \vdash \exists x(P(x) \to \forall y P(y)), \bot, P(f_0(a))}}{P(f_0(a)), \top \vdash P(f_0(a)), \exists x(P(x) \to \forall y P(y)), \bot}\,\mathrm{PR}_1}{\top \vdash P(f_0(a)), \exists x(P(x) \to \forall y P(y)), (P(f_0(a)) \to \bot)}\,\to\!\mathrm{R}}{\top \vdash P(f_0(a)), \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot)}\,\exists \mathrm{R}_{f_0(a)}}{\top \vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot)\,, P(f_0(a))}\,\mathrm{PR}_1}{\top \vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot)\,, \forall y P(y) +^{f_0(a))} P(f_0(a))}\,\forall \mathrm{R}_{f_0(a)}}{\vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot)\,, \top \to \forall y P(y) +^{f_0(a))} P(f_0(a))}\,\to\!\mathrm{R}}{\vdash \exists x(P(x) \to \forall y P(y)) +^{f_0(a)} (P(f_0(a)) \to \bot) +^{a} \left(\top \to \forall y P(y) +^{f_0(a))} P(f_0(a))\right)}\,\exists \mathrm{R}_a$$

5.7 $\mathbf{LK^E}$ to LK

The proof generated is not yet a proof in LK. The sequents are made from expansions and not formulas, the terms in for example $\exists$R may have Skolem functions. The Skolem terms are removed in the first step [7, Lemma 7]

$$\begin{aligned}
\mathsf{rm}\left(\overline{\Gamma \vdash \Delta}\right) &= \overline{\mathsf{Sh}(\Gamma \vdash \Delta)} \\
\mathsf{rm}\left(\dfrac{\pi}{\Gamma \vdash \Delta}\,\mathcal{R}\right) &= \dfrac{\mathsf{rm}\,(\pi)}{\mathsf{Sh}(\Gamma \vdash \Delta)}\,\mathcal{R} \\
&\qquad \text{when } \mathcal{R} \in \{\neg\mathrm{L}, \neg\mathrm{R}, \vee\mathrm{R}, \wedge\mathrm{L}, \to\!\mathrm{R}, \exists\mathrm{R}_x, \forall\mathrm{L}_y, \mathrm{PL}_n, \mathrm{PR}_m\} \\
\mathsf{rm}\left(\dfrac{\pi_1 \qquad \pi_2}{\Gamma \vdash \Delta}\,\mathcal{R}\right) &= \dfrac{\mathsf{rm}\,(\pi_1) \qquad \mathsf{rm}\,(\pi_2)}{\mathsf{Sh}(\Gamma \vdash \Delta)}\,\mathcal{R} \qquad \text{when } \mathcal{R} \in \{\vee\mathrm{L}, \wedge\mathrm{R}, \to\!\mathrm{L}\} \\
\mathsf{rm}\left(\dfrac{\pi}{\exists x A +^{f(\bar{t})} E, \Gamma \vdash \Delta}\exists\mathrm{L}_{f(\bar{t})}\right) &= \dfrac{\mathsf{rm}(\pi\,[f(\bar{t}) := y])}{\exists x A, \mathsf{Sh}(\Gamma) \vdash \mathsf{Sh}(\Delta)}\exists\mathrm{L}_y \\
&\qquad \text{when } y \text{ does not occur in } \exists x A, \mathsf{Sh}(\Gamma) \vdash \mathsf{Sh}(\Delta) \\
\mathsf{rm}\left(\dfrac{\pi}{\Gamma \vdash \Delta, \forall x A +^{f(\bar{t})} E}\forall\mathrm{R}_{f(\bar{t})}\right) &= \dfrac{\mathsf{rm}(\pi\,[f(\bar{t}) := y])}{\mathsf{Sh}(\Gamma) \vdash \mathsf{Sh}(\Delta), \forall x A}\forall\mathrm{R}_y \\
&\qquad \text{when } y \text{ does not occur in } \mathsf{Sh}(\Gamma) \vdash \mathsf{Sh}(\Delta), \forall x A
\end{aligned}$$

The final step is to remove $\top$ and $\bot$ from the formulas. This is simply done by reconstructing the proof by applying the inference steps.

$$\begin{aligned}
\mathsf{rmTF}\left((A,\Gamma\vdash\Delta,A)\,,\overline{A,\top,\ldots,\top\vdash\bot,\ldots,\bot,A}\right) &= A,\Gamma\vdash\Delta,A\\
\mathsf{rmTF}\left((\Gamma\vdash\Delta)\,,\frac{\pi}{S}\,\mathcal{R}\right) &= \frac{\mathsf{rmTF}((\Gamma'\vdash\Delta'),\pi)}{\Gamma\vdash\Delta}\,\mathcal{R}\\
&\quad\text{when } \frac{\Gamma'\vdash\Delta'}{\Gamma\vdash\Delta}\,\mathcal{R}\\
\mathsf{rmTF}\left((\Gamma\vdash\Delta)\,,\frac{\pi_1 \qquad \pi_2}{S}\,\mathcal{R}\right) &= \frac{\mathsf{rmTF}((\Gamma'\vdash\Delta'),\pi_1) \qquad \mathsf{rmTF}((\Gamma''\vdash\Delta''),\pi_2)}{\Gamma\vdash\Delta}\,\mathcal{R}\\
&\quad\text{when } \frac{\Gamma'\vdash\Delta' \qquad \Gamma''\vdash\Delta''}{\Gamma\vdash\Delta}\,\mathcal{R}
\end{aligned}$$

Now the deskolemization process is completed. An expansion was extracted, e, deskolemized, desk, a proof was reconstructed, Pr and finally rewritten as a proof in LK by rm and rmTF. The appendix A lists an actual implementation in Haskell of the described deskolemization process.

5.8 Example of LK$^{\mathbf{E}}$ to LK

Applying rm to the generated proof is simple. In most cases only Sh is needed, which turns the expansions into formulas. The first step is as follows

$$\begin{aligned}
&\mathsf{rm}\left(\frac{\pi}{\vdash\exists x(P(x)\to\forall yP(y)) +^{f_0(a)} (P(f_0(a))\to\bot) +^{a} \left(\top\to\forall yP(y) +^{f_0(a))} P(f_0(a))\right)}\,\exists\mathrm{R}_a\right) =\\
&\frac{\mathsf{rm}\,(\pi)}{\mathsf{Sh}(\vdash\exists x(P(x)\to\forall yP(y)) +^{f_0(a)} (P(f_0(a))\to\bot) +^{a} \left(\top\to\forall yP(y) +^{f_0(a))} P(f_0(a))\right))}\,\exists\mathrm{R}_a =\\
&\frac{\mathsf{rm}\,(\pi)}{\vdash\exists x(P(x)\to\forall yP(y))}\,\exists\mathrm{R}_a =
\end{aligned}$$

The next step for the $\Rightarrow$R inference rule is similar. For the third step at the inference rule $\forall\mathrm{R}_{f_0(a)}$ a fresh variable is needed, for example x_2. All occurrences of $f_0(a)$ in the proof have to be substituted by x_2 because of $\mathsf{rm}(\pi\,[f(\bar{t}) := y])$. Thus

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{P(f_0(a)),\top\vdash\exists x(P(x)\to\forall yP(y)),\bot,P(f_0(a))}}{P(f_0(a)),\top\vdash P(f_0(a)),\exists x(P(x)\to\forall yP(y)),\bot}\,\mathrm{PR}_1}{\top\vdash P(f_0(a)),\exists x(P(x)\to\forall yP(y)),(P(f_0(a))\to\bot)}\,\to\mathrm{R}}{\top\vdash P(f_0(a)),\exists x(P(x)\to\forall yP(y)) +^{f_0(a)} (P(f_0(a))\to\bot)}\,\exists\mathrm{R}_{f_0(a)}}{\top\vdash\exists x(P(x)\to\forall yP(y)) +^{f_0(a)} (P(f_0(a))\to\bot)\,,P(f_0(a))}\,\mathrm{PR}_1$$

becomes

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{P(x_2),\top\vdash\exists x(P(x)\to\forall yP(y)),\bot,P(x_2)}}{P(x_2),\top\vdash P(x_2),\exists x(P(x)\to\forall yP(y)),\bot}\,\mathrm{PR}_1}{\top\vdash P(x_2),\exists x(P(x)\to\forall yP(y)),(P(x_2)\to\bot)}\,\to\mathrm{R}}{\top\vdash P(x_2),\exists x(P(x)\to\forall yP(y)) +^{x_2} (P(x_2)\to\bot)}\,\exists\mathrm{R}_{x_2}}{\top\vdash\exists x(P(x)\to\forall yP(y)) +^{x_2} (P(x_2)\to\bot)\,,P(x_2)}\,\mathrm{PR}_1$$

Note that also the inference $\exists\mathrm{R}_{f_0(a)}$ changed to $\exists\mathrm{R}_{x_2}$. The final result of rm is

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$P(x_2), \top \vdash \exists x(P(x) \to \forall y P(y)), \bot, P(x_2)$}
\RightLabel{PR$_1$}
\UnaryInfC{$P(x_2), \top \vdash P(x_2), \exists x(P(x) \to \forall y P(y)), \bot$}
\RightLabel{$\to$R}
\UnaryInfC{$\top \vdash P(x_2), \exists x(P(x) \to \forall y P(y)), P(x_2) \to \bot$}
\RightLabel{$\exists$R$_{x_2}$}
\UnaryInfC{$\top \vdash P(x_2), \exists x(P(x) \to \forall y P(y))$}
\RightLabel{PR$_1$}
\UnaryInfC{$\top \vdash \exists x(P(x) \to \forall y P(y)), P(x_2)$}
\RightLabel{$\forall$R$_{x_2}$}
\UnaryInfC{$\top \vdash \exists x(P(x) \to \forall y P(y)), \forall y P(y)$}
\RightLabel{$\to$R}
\UnaryInfC{$\vdash \exists x(P(x) \to \forall y P(y)), \top \to \forall y P(y)$}
\RightLabel{$\exists$R$_a$}
\UnaryInfC{$\vdash \exists x(P(x) \to \forall y P(y))$}
\end{prooftree}
$$

The final function to produce the proof in LK is rmTF. This function simply applies the inferences rules starting at the root of the proof. The result of rmTF is

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$P(x_2), P(a) \vdash \exists x(P(x) \to \forall y P(y)), \forall y P(y)), P(x_2)$}
\RightLabel{PR$_1$}
\UnaryInfC{$P(x_2), P(a) \vdash P(x_2), \exists x(P(x) \to \forall y P(y)), \forall y P(y))$}
\RightLabel{$\to$R}
\UnaryInfC{$P(a) \vdash P(x_2), \exists x(P(x) \to \forall y P(y)), P(x_2) \to \forall y P(y))$}
\RightLabel{$\exists$R$_{x_2}$}
\UnaryInfC{$P(a) \vdash P(x_2), \exists x(P(x) \to \forall y P(y))$}
\RightLabel{PR$_1$}
\UnaryInfC{$P(a) \vdash \exists x(P(x) \to \forall y P(y)), P(x_2)$}
\RightLabel{$\forall$R$_{x_2}$}
\UnaryInfC{$P(a) \vdash \exists x(P(x) \to \forall y P(y)), \forall y P(y)$}
\RightLabel{$\to$R}
\UnaryInfC{$\vdash \exists x(P(x) \to \forall y P(y)), P(a) \to \forall y P(y)$}
\RightLabel{$\exists$R$_a$}
\UnaryInfC{$\vdash \exists x(P(x) \to \forall y P(y))$}
\end{prooftree}
$$

This indeed is a proof of the drinker paradox $\vdash \exists x(P(x) \to \forall y P(y))$.

Chapter 6

Conclusion

We have shown an algorithm to deskolemize a proof. The algorithm takes a sequent a proof of the skolemized sequent and produces a proof of the original sequent. The implementation can be found in appendix A.

The algorithm works for cut-free proofs. The expansions trees that are extracted from the proof do not support a cut rule. One way to overcome this limitation is to use a cut elimination. However, an extension to expansions trees, expansions trees with cut[6] is a promising result to deskolemization with cut rules.

Bibliography

[1] ALEXANDRE RIAZANOV, A. V. The design and implementation of vampire. *AI Communications 15* (2002), 91–110.

[2] BAAZ, M., HETZL, S., LEITSCH, A., RICHTER, C., AND SPOHR, H. Cut-elimination: Experiments with ceres. In *Logic for Programming, Artificial Intelligence, and Reasoning*, F. Baader and A. Voronkov, Eds., vol. 3452 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 481–495.

[3] BAAZ, M., AND LEITSCH, A. On skolemization and proof complexity. *Fundamenta Informaticae 20*, 4 (1994), 353–379.

[4] GENTZEN, G. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift 39* (1934), 176–210.

[5] GENTZEN, G. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift 39* (1934), 405–431.

[6] HETZL, S., AND WELLER, D. Expansion trees with cut. *CoRR abs/1308.0428* (2013).

[7] M. BAAZ, S. H., AND WELLER, D. On the complexity of proof deskolemization. *The Journal of Symbolic Logic 77*, 2 (2012).

[8] MILLER, D. A. A compact representation of proofs. *Studia Logica 46*, 4 (1987), 347–370.

[9] PELLETIER, F., SUTCLIFFE, G., AND SUTTNER, C. The Development of CASC. *AI Communications 15*, 2-3 (2002), 79–90.

[10] SMULLYAN, R. M. *What is the name of this book?* Prentice-Hall Englewood Cliffs, New Jersey, 1978.

[11] SUTCLIFFE, G. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning 43*, 4 (2009), 337–362.

[12] SUTCLIFFE, G., AND SUTTNER, C. The State of CASC. *AI Communications 19*, 1 (2006), 35–48.

Appendices

Appendix A

Haskell implementation

The implementation includes a tablaux prover for testing purposes. The core functionality, `extract`, `desk`, `pr`, `rm` and `rmTF` can be found in Expansion.hs. Function.hs contains a function sk that implements the Skolemization of a formula.

Formula.hs

```
module Formula (allVariables,substituteTerm,Formula(..),Term(..),Function,
        Variable,Predicate,Symbol(..),Sequent,substitute,fnot,for,fand,fimp,fpred,
        Proof(..),LR(..),Rule(..),drinkerParadox,RuleResult,termsS,freshVariable,
        freeTerms)where


import Prelude hiding(Left,Right)
import Data.List
import Data.Maybe

type Function = String
type Variable = String
type Predicate = String

data Term = Fun Function [Term] | Var Variable deriving (Eq)

instance Show Term where
        show (Var name) = name
        show (Fun name terms) = name ++ "" ++ (show terms) ++ ""

data Symbol = Not | Or | And | Imp | Exists | Forall  deriving (Eq)

instance Show Symbol where
        show Not = "\xac"
        show Or = "\x2228␣"
        show And = "\x2227␣"
        show Imp = "\x2192␣"
        show Exists = "\x2203"
        show Forall = "\x2200"


data Formula = TopBot | Pred Predicate [Term] | Unary Symbol Formula |
        Binary Symbol Formula Formula | Quantifier Symbol Variable Formula
        deriving (Eq)

instance Show Formula where
        show (Pred p terms) = p ++  (show terms)
```

```haskell
        show (Unary s f) = show s ++ (show f)
        show (Binary s f1 f2) = "(" ++ (show f1) ++ (show s) ++ (show f2) ++ ")"
        show (Quantifier s x f) = (show s) ++ x ++ (show f)

type Sequent = ([Formula],[Formula])

ant :: Sequent -> [Formula]
ant (gamma,delta) = gamma

suc :: Sequent -> [Formula]
suc (gamma,delta) = delta

data LR = Left | Right deriving (Eq,Show)
data Rule = Axiom | SymbolRule LR Symbol | QuantifierRule LR Symbol Term |
        Perm LR Int deriving (Show)

type RuleResult = (Rule,[Sequent])

data Proof = Proof Rule Sequent [Proof]

indent :: Int -> String
indent n = (replicate (n*4) ' ')

printProof :: (Int,Proof) -> String
printProof (n,(Proof rule s proofs))    = concat
        (map printProof (zip [n..(length proofs)] proofs)) ++
        "\n" ++ (indent n) ++ "--------------" ++ (show rule) ++ "\n" ++ (show s)

instance Show Proof where
        show proof = printProof (0,proof)

breakFormula :: Formula -> [Formula]
breakFormula (Pred _ _) = []
breakFormula (Unary s f) = [f]
breakFormula (Quantifier s _ f) = [f]
breakFormula (Binary s f1 f2) = [f1,f2]

formulaSymbol :: Formula -> Maybe Symbol
formulaSymbol (Pred _ _) = Nothing
formulaSymbol (Unary s _) = Just s
formulaSymbol (Quantifier s _ _) = Just s
formulaSymbol (Binary s _ _) = Just s

fpred :: String -> [Term] -> Formula
fpred s terms = Pred s terms

fnot :: Formula -> Formula
fnot f = Unary Not f

fand :: Formula -> Formula -> Formula
fand f1 f2 = Binary And f1 f2

for :: Formula -> Formula -> Formula
for f1 f2 = Binary Or f1 f2

fimp :: Formula -> Formula -> Formula
fimp f1 f2 = Binary Imp f1 f2

fexists :: Variable -> Formula -> Formula
fexists x f = Quantifier Exists x f
```

```
fforall :: Variable -> Formula -> Formula
fforall x f = Quantifier Forall x f

-- |A list of all variables x1, x2, ...
allVariables :: [Variable]
allVariables = vars 1
        where
        vars :: Int -> [Variable]
        vars n = ("x" ++ show(n)):vars (n+1)

freshVariable :: Sequent -> Variable
freshVariable (ant,suc) = head
                [x | x <- allVariables, (not (elem (Var x) sequentterms))]
        where
        sequentterms = concat (map terms ant) ++ concat (map terms suc)

freeTerms :: Formula -> [Term]
freeTerms f = (terms f) \\ (binderTerms f)

binderTerms :: Formula -> [Term]
binderTerms f = nub (binderTerms' f)
        where
        binderTerms' (Pred _ t) = []
        binderTerms' (Quantifier Exists x f) = (Var x):(binderTerms f)
        binderTerms' (Quantifier Forall x f) = (Var x):(binderTerms f)
        binderTerms' (Unary _ f) = terms f
        binderTerms' (Binary _ f1 f2) = (binderTerms f1) ++ (binderTerms f2)


termsS :: Sequent -> [Term]
termsS (gamma,delta) = (concat $ map terms gamma) ++ (concat $ map terms delta)
terms :: Formula -> [Term]
terms f = nub (terms' f)
        where
        terms' (Pred _ t) = t
        terms' (Quantifier Exists x f) = (Var x):(terms f)
        terms' (Quantifier Forall x f) = (Var x):(terms f)
        terms' (Unary _ f) = terms f
        terms' (Binary _ f1 f2) = (terms f1) ++ (terms f2)

substituteTerm :: Term -> Term -> Term -> Term
substituteTerm a@(Var _) b t            = if a == b then t else a
substituteTerm a@(Fun f terms) x t      = if a == x then t else
        Fun f (map (\y -> (substituteTerm y x t)) terms)

substitute :: Formula -> Term -> Term -> Formula
substitute (Pred p terms) var t = Pred p
        (map (\x -> substituteTerm x var t) terms)
substitute err@(Quantifier Exists x f) var t =
        if (Var x) == var
        then error ("Oops␣"++(show var)++"␣is␣bound␣in␣" ++(show err))
        else Quantifier Exists x (substitute f var t)
substitute err@(Quantifier Forall x f) var t =
        if (Var x) == var
        then error ("Oops␣"++(show var)++"␣is␣bound␣in␣" ++(show err))
        else Quantifier Forall x (substitute f var t)
substitute (Unary b f) var t = Unary b (substitute f var t)
substitute (Binary b f1 f2) var t =
        Binary b (substitute f1 var t) (substitute f2 var t)

drinkerParadoxPart :: Formula
drinkerParadoxPart = fimp (fpred "p" [Var "x1"])
        (fforall "x2" (fpred "p" [Var "x2"]))

drinkerParadox :: Formula
drinkerParadox = fexists "x1" (drinkerParadoxPart)

drinkerParadoxAlmost :: Sequent
drinkerParadoxAlmost = ([],
        [substitute drinkerParadoxPart (Var "x1") (Var "x"),drinkerParadox])
```

Expansions.hs

```haskell
module Expansion(Expansion,extract,desk,pr,sh,skolemTerms,shS,skolemTermsS,
        rm,rmTF) where

import Prelude hiding (Left,Right)

import Formula
import Prover
import Data.List
import Data.Maybe
import Functions

data Expansion = Top | Bot | Pred Predicate [Term] | Unary Symbol Expansion |
        Binary Symbol Expansion Expansion |
        Quantifier Symbol Variable Formula [(Term,Expansion)] deriving (Eq)


instance Show Expansion where
        show Top = "Top"
        show Bot = "Bot"
        show (Expansion.Pred p terms) = p ++  (show terms)
        show (Expansion.Unary s f) = show s ++ (show f)
        show (Expansion.Binary s f1 f2) = "(" ++ (show f1) ++ (show s) ++
                (show f2) ++ ")"
        show (Expansion.Quantifier s x f omega) = (show s) ++ x ++ (show f) ++
                "+" ++ (show omega)

type ExpSequent = ([Expansion],[Expansion])

data LKEProof = LKEProof Rule ExpSequent [LKEProof]

indent :: Int -> String
indent n = (replicate (n*4) ' ')

printProof :: (Int,LKEProof) -> String
printProof (n,(LKEProof rule s proofs))         = concat (map printProof
        (zip [n..(length proofs)] proofs)) ++ "\n" ++ (indent n) ++
                "--------------" ++ (show rule) ++ "\n" ++ (show s)

instance Show LKEProof where
        show proof = printProof (0,proof)

shS :: ExpSequent -> ExpSequent
shS (gamma,delta) = (map sh gamma,map sh delta)

sh :: Expansion -> Expansion
sh Top = Top
sh Bot = Bot
sh (Expansion.Pred p terms) = Expansion.Pred p terms
sh (Expansion.Unary s e) = Expansion.Unary s (sh e)
sh (Expansion.Binary s e1 e2) = Expansion.Binary s (sh e1) (sh e2)
sh (Expansion.Quantifier s x f omega) = Expansion.Quantifier s x f []

skolemTermsS :: ExpSequent -> [Term]
skolemTermsS (gamma,delta) = union (concat $ map (skolemTerms False) gamma)
        (concat $ map (skolemTerms True) delta)

skolemTerms :: Bool -> Expansion -> [Term]
skolemTerms _ Top = []
skolemTerms _ Bot = []
skolemTerms _ (Expansion.Pred _ _) = []
skolemTerms b (Expansion.Unary Not e) = skolemTerms (not b) e
skolemTerms b (Expansion.Binary And e1 e2) =
        union (skolemTerms b e1) (skolemTerms b e2)
skolemTerms b (Expansion.Binary Or e1 e2) =
        union (skolemTerms b e1) (skolemTerms b e2)
skolemTerms b (Expansion.Binary Imp e1 e2) =
        union (skolemTerms (not b) e1) (skolemTerms b e2)
skolemTerms True (Expansion.Quantifier Exists _ _ omega) =
        foldl1 union (map (skolemTerms True) [e | (t,e) <- omega])
skolemTerms False (Expansion.Quantifier Exists _ _ [(t,e)]) =
        union [t] (skolemTerms False e)
skolemTerms True (Expansion.Quantifier Forall _ _ [(t,e)]) =
        union [t] (skolemTerms True e)
skolemTerms False (Expansion.Quantifier Forall _ _ omega) =
        foldl1 union (map (skolemTerms False) [e | (t,e) <- omega])


eterms :: Expansion -> [Term]
eterms e = nub (terms' e)
        where
        terms' (Top) = []
        terms' (Bot) = []
        terms' (Expansion.Pred _ t) = t
        terms' (Expansion.Quantifier Exists x f omega) =
                (Var x):(concat $ map eterms [e | (t,e) <- omega])
        terms' (Expansion.Quantifier Forall x f omega) =
                (Var x):(concat $ map eterms [e | (t,e) <- omega])
        terms' (Expansion.Unary _ e) = eterms e
        terms' (Expansion.Binary _ e1 e2) = (eterms e1) ++ (eterms e2)

efreshVariable :: ExpSequent -> Variable
efreshVariable (ant,suc) =
                head [x | x <- allVariables, (not (elem (Var x) sequentterms))]
        where
        sequentterms = concat (map eterms ant) ++ concat (map eterms suc)

substituteTE :: (Term,Expansion) -> Term -> Term -> (Term,Expansion)
substituteTE (t,e) x t'
        | t == x                = (t',esubstitute e x t')
        | otherwise             = (t,esubstitute e x t')

esubstituteS :: ExpSequent -> Term -> Term -> ExpSequent
esubstituteS (gamma,delta) var t =
        (map (\x -> esubstitute x var t) gamma,
        map (\x -> esubstitute x var t) delta)

esubstituteP :: LKEProof -> Term -> Term -> LKEProof
esubstituteP (LKEProof (QuantifierRule lr symbol term) s proofs) x y =
        (LKEProof (QuantifierRule lr symbol (substituteTerm term x y))
                (esubstituteS s x y) (map (\p -> esubstituteP p x y) proofs))
esubstituteP (LKEProof rule s proofs) x y =
        (LKEProof rule (esubstituteS s x y)
                (map (\p -> esubstituteP p x y) proofs))

esubstitute :: Expansion -> Term -> Term -> Expansion
esubstitute Top _ _ = Top
esubstitute Bot _ _ = Bot
esubstitute (Expansion.Pred p terms) var t              =
        Expansion.Pred p (map (\x -> substituteTerm x var t) terms)
```

```
esubstitute err@(Expansion.Quantifier Exists x f omega) var t =
        if (Var x) == var
        then error ("Oops␣"++(show var)++"␣is␣bound␣in␣" ++(show err))
        else Expansion.Quantifier Exists x (substitute f var t) omega'
                where omega' = [substituteTE te var t | te <- omega]
esubstitute err@(Expansion.Quantifier Forall x f omega) var t =
        if (Var x) == var
        then error ("Oops␣"++(show var)++"␣is␣bound␣in␣" ++(show err))
        else Expansion.Quantifier Forall x (substitute f var t) omega'
        where omega' = [substituteTE te var t | te <- omega]
esubstitute (Expansion.Unary b e) var t                          =
        Expansion.Unary b (esubstitute e var t)
esubstitute (Expansion.Binary b e1 e2) var t            =
        Expansion.Binary b (esubstitute e1 var t) (esubstitute e2 var t)

toExpansion :: Formula -> Expansion
toExpansion (Formula.Pred p terms) = Expansion.Pred p terms
toExpansion (Formula.Unary s f) = Expansion.Unary s (toExpansion f)
toExpansion (Formula.Binary s f1 f2) =
        Expansion.Binary s (toExpansion f1) (toExpansion f2)
toExpansion (Formula.Quantifier s x f) = Expansion.Quantifier s x f []

extract :: Proof -> ExpSequent
extract (Proof Axiom (a:gamma,b:delta) proofs) =
        (toExpansion a : replicate (length gamma) Top,
        toExpansion b : replicate (length delta) Bot)
extract (Proof (SymbolRule Left symbol) sequent proofs) =
        extractLeft symbol (map extract proofs)
extract (Proof (SymbolRule Right symbol) sequent proofs) =
        extractRight symbol (map extract proofs)
extract (Proof (Perm Left n) sequent [proof]) = (fromFirst n gamma',delta')
        where (gamma',delta') = extract proof
extract (Proof (Perm Right n) sequent [proof]) = (gamma',fromFirst n delta')
        where (gamma',delta') = extract proof
extract (Proof (QuantifierRule Right Exists term)
        sequent@(_,f@(Formula.Quantifier Exists x a):_) [proof]) =
        (gamma,eunion e1 e2':delta)
        where
        (gamma,e2:e1:delta) = extract proof
        e2' = Expansion.Quantifier Exists x a [(term,e2)]

eunion :: Expansion -> Expansion -> Expansion
eunion Top e2   = e2
eunion Bot e2   = e2
eunion e1 Top   = e1
eunion e1 Bot   = e1
eunion (Expansion.Pred p1 terms1) (Expansion.Pred p2 terms2)
        | and [p1 == p2,terms1 == terms2] = Expansion.Pred p1 terms1
        | otherwise = error "No␣union"
eunion (Expansion.Unary s1 e1) (Expansion.Unary s2 e2)
        | s1 == s2      = Expansion.Unary s1 (eunion e1 e2)
        | otherwise = error "No␣union"
eunion (Expansion.Binary s1 e1 e1') (Expansion.Binary s2 e2 e2')
        | s1 == s2      = Expansion.Binary s1 (eunion e1 e2) (eunion e1' e2')
        | otherwise = error "No␣union"
eunion (Expansion.Quantifier s1 x1 a1 e1) (Expansion.Quantifier s2 x2 a2 e2)
        | and [s1 == s2,x1 == x2,a1 == a2]      =
                Expansion.Quantifier s1 x1 a1 (eunion' e1 e2)
        | otherwise = error "No␣union"
        where
        eunion' :: [(Term,Expansion)] -> [(Term,Expansion)] -> [(Term,Expansion)]
        eunion' [] b            = b
        eunion' (a@(t,e1):as) b
                | (elemIndex t bterms) == Nothing = a:eunion' as b
                | otherwise                         = (t,eunion e1 e2) : eunion' as b
                where
                        (bterms,_) = unzip b
                        (t2,e2) = b !! (fromJust (elemIndex t bterms))
eunion e1 e2 = error $ "No␣union␣for␣" ++ (show e1) ++ "␣and␣"++ (show e2)

enot :: Expansion -> Expansion
enot e = Expansion.Unary Not e

eimp :: Expansion -> Expansion -> Expansion
eimp e1 e2 = Expansion.Binary Imp e1 e2

extractLeft :: Symbol -> [ExpSequent] -> ExpSequent
extractLeft Not [(gamma,e:delta)] = (enot e:gamma,delta)

extractRight :: Symbol -> [ExpSequent] -> ExpSequent
extractRight Not [(e:gamma,delta)] = (gamma,enot e:delta)
extractRight Imp [(e1:gamma,e2:delta)] = (gamma,eimp e1 e2:delta)



desk :: Bool -> Int -> [Term] -> Formula -> Expansion -> Expansion
desk _ _ _ _ Top = Top
desk _ _ _ _ Bot = Bot
desk _ _ _ f@(Formula.Pred p terms) e@(Expansion.Pred p2 terms2)
        | and [p == p2] = e
        | otherwise             = error ("Predicates␣don't␣match:␣" ++
                (show f) ++ "␣" ++ (show e))
desk b n mu (Formula.Unary Not f) (Expansion.Unary Not e) =
        desk (not b) n mu f e -- TODO check?
desk True n mu (Formula.Binary Imp f1 f2) (Expansion.Binary Imp e1 e2) =
        Expansion.Binary Imp (desk False n mu f1 e1) (desk True m mu f2 e2)
        where m = n + (qocc False f1)
desk True n mu (Formula.Quantifier Exists x a)
                (Expansion.Quantifier Exists _ _ omega) =
        (Expansion.Quantifier Exists x a omega')
        where
        omega' = [(term,desk True n (mu ++ [term]) a e) | (term,e) <- omega]
desk True n mu (Formula.Quantifier Forall x a) e =
        Expansion.Quantifier Forall x a [(Fun (skolemFunc n) mu,e')]
        where e' = desk True (n+1) mu a e
desk b _ _ a _ = error $ "Error␣" ++ (show b) ++ (show a)

permL :: Int -> ExpSequent -> LKEProof -> LKEProof
permL 0 _ proof = proof
permL index s proof = LKEProof (Perm Left index) s [proof]

permR :: Int -> ExpSequent -> LKEProof -> LKEProof
permR 0 _ proof = proof
permR index s proof = LKEProof (Perm Right index) s [proof]

symbolIndex :: Symbol -> [Expansion] -> Maybe Int
symbolIndex s e = symbolIndex' 0 s e
        where
        symbolIndex' :: Int -> Symbol -> [Expansion] -> Maybe Int
        symbolIndex' _ symbol [] = Nothing
        symbolIndex' index symbol ((Expansion.Binary symbol' _ _):e)
                | symbol == symbol'     = Just index
                | otherwise             = symbolIndex' (index+1) symbol e
        symbolIndex' index symbol ((Expansion.Unary symbol' _):e)
                | symbol == symbol'     = Just index
                | otherwise             = symbolIndex' (index+1) symbol e
        symbolIndex' index symbol ((Expansion.Quantifier symbol' _ _ _):e)
                | symbol == symbol'     = Just index
                | otherwise             = symbolIndex' (index+1) symbol e
        symbolIndex' index symbol (_:e) = symbolIndex' (index+1) symbol e

pr :: ExpSequent -> LKEProof
pr s@(gamma,delta)
        | length (intersect gamma' delta') > 0  =
                axiom gamma delta
        | leftRight == Left                            =
                permL index s (prLeft (setFirst index gamma,delta))
        | leftRight == Right                           =
                permR index s (prRight (gamma,setFirst index delta))
                where
                gamma' = deleteBy (==) Bot (deleteBy (==) Top gamma)
                delta' = deleteBy (==) Bot (deleteBy (==) Top delta)
                axiom gamma delta = let
                        e = head (intersect gamma' delta')
                        li = fromJust (elemIndex e gamma)
                        ri = fromJust (elemIndex e delta)
                        in permL li (gamma,setFirst ri delta)
                                (permR ri s (LKEProof Axiom
                                        (setFirst li gamma,setFirst ri delta) []))
                (leftRight,index) = head $ catMaybes
                        (map (hasExpansionOf s) [Or,And,Imp,Forall,Exists])

                prLeft :: ExpSequent -> LKEProof
                prLeft ((Expansion.Binary Imp e1 e2):gamma,delta) =
                        LKEProof (SymbolRule Left Imp) s
                                [pr (gamma,e1:delta),pr (e2:gamma,delta)]
                prLeft (e:gamma,delta) = error $ "Left␣" ++ show e

                prRight :: ExpSequent -> LKEProof
                prRight (gamma,(Expansion.Quantifier Exists x a omega):delta) =
                        LKEProof (QuantifierRule Right Exists t) s
                                [(pr (gamma,e:e2:delta))]
                        where
                        (t,e) = head [(t',e') | (t',e') <- omega,
                                elemIndex t' (skolemTermsS s) == Nothing ]
                        omega' = delete (t,e) omega
                        e2 = Expansion.Quantifier Exists x a omega'
                prRight (gamma,(Expansion.Binary Imp e1 e2):delta) =
                        LKEProof (SymbolRule Right Imp) s [pr (e1:gamma,e2:delta)]
                prRight (gamma,(Expansion.Quantifier Forall x a [(t,e)]):delta) =
                        LKEProof (QuantifierRule Right Forall t) s [pr (gamma,e:delta)]
                prRight (gamma,e:delta) = error $ "Right" ++ show e

hasExpansionOf :: ExpSequent -> Symbol -> Maybe (LR,Int)
hasExpansionOf (gamma,delta) s
        | symbolIndex s gamma /= Nothing        =
                Just (Left,fromJust(symbolIndex s gamma))
        | symbolIndex s delta /= Nothing        =
                Just (Right,fromJust(symbolIndex s delta))
        | otherwise                             = Nothing

rm :: LKEProof -> LKEProof
rm (LKEProof Axiom s []) = (LKEProof Axiom (shS s) [])
rm (LKEProof rule@(SymbolRule lr symbol) s proofs)                      =
        (LKEProof rule (shS s) (map rm proofs))
rm (LKEProof (QuantifierRule Left Exists term) s [proof])       =
        (LKEProof (QuantifierRule Left Exists y)
                (esubstituteS (shS s) term y) [esubstituteP (rm proof) term y])
        where y = Var (efreshVariable s)
rm (LKEProof (QuantifierRule Right Forall term) s [proof])      =
        (LKEProof (QuantifierRule Right Forall y)
                (esubstituteS (shS s) term y) [esubstituteP (rm proof) term y])
        where y = Var (efreshVariable s)
rm (LKEProof rule@(QuantifierRule _ _ term) s [proof])          =
        (LKEProof rule (shS s) [rm proof])
rm (LKEProof rule@(Perm _ _) s [proof])                                         =
        (LKEProof rule (shS s) [rm proof])
rm p = error ("Error" ++ (show p))

rmTF :: Sequent -> LKEProof -> Proof
rmTF s (LKEProof Axiom _ []) = Proof Axiom s []
rmTF s (LKEProof rule _ proofs) = Proof rule s proofs'
        where
        (_,nextsequents) = fromJust $ applyRule s rule
        r = zip nextsequents proofs
        proofs' = [rmTF s' p' | (s',p') <- r ]
```

Functions.hs

```haskell
module Functions where
import Formula

qocc :: Bool -> Formula -> Int
qocc b (Pred _ _ ) = 0
qocc b (Unary Not f) = qocc (Prelude.not b) f
qocc b (Binary Or f1 f2) = (qocc b f1) + (qocc b f2)
qocc b (Binary And f1 f2) = (qocc b f1) + (qocc b f2)
qocc True (Binary Imp f1 f2) = (qocc False f1) + (qocc True f2)
qocc False (Binary Imp f1 f2) = (qocc True f1) + (qocc False f2)
qocc True (Quantifier Forall x f) = 1 + (qocc True f)
qocc False (Quantifier Forall x f) = qocc False f
qocc True (Quantifier Exists x f) = qocc True f
qocc False (Quantifier Exists x f) = 1 + (qocc False f)


skolemFunc :: Int -> Function
skolemFunc n = ("f" ++ (show n))

sequentAsFormula :: Sequent -> Formula
sequentAsFormula ([],[]) = error "Empty␣sequent"
sequentAsFormula ([],delta) = foldl1 for delta
sequentAsFormula (gamma,[]) = foldl1 fand gamma
sequentAsFormula (gamma,delta) = fimp (foldr1 fand gamma) (foldr1 for delta)

unfoldFormula :: Int -> Symbol -> Formula -> [Formula]
unfoldFormula 1 _ f = [f]
unfoldFormula n symbol (Binary symbol' f1 f2)
        | symbol == symbol'     = f1: unfoldFormula (n-1) symbol f2
        | otherwise             = error "Wrong␣symbol"
unfoldFormula _ _ f = error $ "Can't␣unfold␣" ++ (show f)


formulaAsSequent :: Int -> Int -> Formula -> Sequent
formulaAsSequent 0 0 _ = error "Empty␣formula"
formulaAsSequent n 0 f = (unfoldFormula n And f,[])
formulaAsSequent 0 m f = ([],unfoldFormula m Or f)
formulaAsSequent n m (Binary Imp f1 f2) =
        (unfoldFormula n And f1,unfoldFormula m Or f2)

skSequent :: Sequent -> Sequent
skSequent s@(ant,suc) = formulaAsSequent (length ant) (length suc)
        $ sk True 0 [] $ sequentAsFormula s

sk :: Bool -> Int -> [Term] -> Formula -> Formula
sk b n mu f@(Pred _ _) = f
sk b n mu (Unary Not f) = Unary Not (sk (not b) n mu f)
sk b n mu (Binary Or f1 f2) = Binary Or (sk b n mu f1) (sk b m mu f2)
        where m = n + (qocc b f1)
sk b n mu (Binary And f1 f2) = Binary And (sk b n mu f1) (sk b m mu f2)
        where m = n + (qocc b f1)
sk b n mu (Binary Imp f1 f2) = Binary Imp (sk (not b) n mu f1) (sk b m mu f2)
        where m = n + (qocc (not b) f1)
sk True n mu (Quantifier Forall x f) = sk True m mu f' where
    m = n + 1
    f' = substitute f (Var x) (Fun (skolemFunc n) mu)
sk False n mu (Quantifier Forall x f) =
        Quantifier Forall x (sk False n (mu ++ [Var x]) f)
```

```haskell
sk True n mu (Quantifier Exists x f) =
        Quantifier Exists x (sk True n (mu ++ [Var x]) f)
sk False n mu (Quantifier Exists x f) = sk False m mu f' where
    m = n + 1
    f' = substitute f (Var x) (Fun (skolemFunc n) mu)
```

Prover.hs

```haskell
module Prover (allVariables,substituteTerm,allRules,applyRule,Formula(..),
        Term(..),Function,Variable,Predicate,Symbol(..),Sequent,fromFirst,
        setFirst,applyLeftRight,substitute,fnot,for,fand,fimp,fpred,Proof(..),
        LR(..),Rule(..),drinkerParadox,proof)where

import Formula
import Prelude hiding(Left,Right)
import Data.List
import Data.Maybe


proof :: Int -> Sequent -> Maybe Proof
proof max s     | len == Nothing                = Nothing
                        | otherwise             = Just (makeProof (fromJust len) s)
        where len = proofLength max s

makeProof :: Int -> Sequent -> Proof
makeProof n s = c goodStep
        where
        goodStep :: RuleResult
        goodStep = head [t | t <- applyLeftRight s, hasProofRule (n-1) t]
        c :: RuleResult -> Proof
        c (rule,sequents) = Proof rule s (map (makeProof (n-1)) sequents)

proofLength :: Int -> Sequent -> Maybe Int
proofLength max s | not (hasProof max s) = Nothing
                        | and [hasProof max s,not (hasProof (max-1) s)] = Just max
                        | otherwise = proofLength (max - 1) s

hasProofRule :: Int -> RuleResult -> Bool
hasProofRule max (_,[]) = True
hasProofRule max (_,sequents) = and (map (hasProof max) sequents)

hasProof :: Int -> Sequent -> Bool
hasProof 0 _ = False
hasProof max s = or ((map (branchProvable (max-1))) branches)
        where
        break :: (Rule,[Sequent]) -> [Sequent]
        break (r,s) = s
        branches :: [[Sequent]]
        branches = map break (applyLeftRight s)
        branchProvable :: Int -> [Sequent] -> Bool
        branchProvable _ [] = True
        branchProvable m sequents = and (map (hasProof m) sequents)

applyRule :: Sequent -> Rule -> Maybe RuleResult
applyRule s r   | result == Nothing = Nothing
                        | otherwise = Just (r, fromJust result)
                        where result = applyRule' s r

applyRule' :: Sequent -> Rule -> Maybe [Sequent]
applyRule' ((Unary Not a):gamma,delta) (SymbolRule Left Not)
        = Just [(gamma,a:delta)]
applyRule' ((Binary Or a b):gamma,delta) (SymbolRule Left Or)
        = Just [(a:gamma,delta),(b:gamma,delta)]
applyRule' ((Binary And a b):gamma,delta) (SymbolRule Left And)
        = Just [(a:b:gamma,delta)]
applyRule' ((Binary Imp a b):gamma,delta) (SymbolRule Left Imp)
```

```
        = Just [(gamma,a:delta),(b:gamma,delta)]
applyRule' s@((Quantifier Exists x a):gamma,delta) (QuantifierRule Left Exists t)
        | elem t (termsS s)     = Nothing
        | otherwise             = Just [(a':gamma,delta)]
                where a' = substitute a (Var x) t
applyRule' (f@(Quantifier Forall x a):gamma,delta) (QuantifierRule Left Forall t)
        = Just [(a':f:gamma,delta)]
                where a' = substitute a (Var x) t
applyRule' (gamma,delta) (Perm Left index)
        = Just [(setFirst index gamma,delta)]

applyRule' (gamma,(Unary Not a):delta) (SymbolRule Right Not)
        = Just [(a:gamma,delta)]
applyRule' (gamma,(Binary Or a b):delta) (SymbolRule Right Or)
        = Just [(gamma,a:b:delta)]
applyRule' (gamma,(Binary And a b):delta) (SymbolRule Right And)
        = Just [(gamma,a:delta),(gamma,b:delta)]
applyRule' (gamma,(Binary Imp a b):delta) (SymbolRule Right Imp)
        = Just [(a:gamma,b:delta)]
applyRule' (gamma,f@(Quantifier Exists x a):delta) (QuantifierRule Right Exists t)
        = Just [(gamma,a':f:delta)]
                where a' = substitute a (Var x) t
applyRule' s@(gamma,(Quantifier Forall x a):delta) (QuantifierRule Right Forall t)
        | elem t (termsS s)     = Nothing
        | otherwise                     = Just [(gamma,a':delta)]
                where a' = substitute a (Var x) t
applyRule' (gamma,delta) (Perm Right index)
        = Just [(gamma,setFirst index delta)]
applyRule' _ _ = Nothing

applyLeftRight :: Sequent -> [RuleResult]
applyLeftRight s@(ant,suc)
        | and [ant /= [],suc /=[],head ant == head suc] = [(Axiom,[])]
        | otherwise = catMaybes $ map (applyRule s) (allRules s)

allRules :: Sequent -> [Rule]
allRules s@(gamma,delta) = allSymbolRules ++ allPerms ++ allQuantifiers
        where
        allSymbolRules =
                [(SymbolRule lr symbol) | lr <- [Left,Right], symbol <- [Not,Or,And,Imp]]
        allPerms = [(Perm Left index) | index <- [1..length(gamma)-1]]
                ++ [(Perm Right index) | index <- [1..length(delta)-1]]
        sequentFreeTerms = (concat (map freeTerms gamma))
                ++ (concat (map freeTerms delta))
        terms = (Var (freshVariable s)):sequentFreeTerms
        allQuantifiers = [(QuantifierRule lr symbol term)
                | lr <- [Left,Right], symbol <- [Exists,Forall], term <- terms]

setFirst :: Int -> [a] -> [a]
setFirst 0 as = as
setFirst index as
        | (index - 1) == (length as)    = (as !! index) : (take (index - 1) as)
        | otherwise                     = (as !! index) :
                (take index as) ++ (drop (index+1) as)

fromFirst :: Int -> [a] -> [a]
fromFirst 0 as = as
fromFirst index (x:as) = (take index as) ++ [x] ++ (drop index as)

permLeft :: Sequent -> [(Rule,[Sequent])]
permLeft (gamma,delta) =
        [(permLeft' (gamma,delta) i) | i <- [1..((length gamma)-1)]]
        where
        permLeft' :: Sequent -> Int -> (Rule,[Sequent])
        permLeft' (gamma,delta) index =
                (Perm Left index,[(setFirst index gamma,delta)])

permRight :: Sequent -> [(Rule,[Sequent])]
permRight (gamma,delta) =
        [(permRight' (gamma,delta) i) | i <- [1..((length delta)-1)]]
        where
        permRight' :: Sequent -> Int -> (Rule,[Sequent])
        permRight' (gamma,delta) index =
                (Perm Right index,[(gamma,setFirst index delta)])
```

Main.hs

```haskell
module Main(main) where
import Functions
import Formula
import Expansion
import Data.Maybe
import Prover

f = Pred "p" []
f2 = Pred "q" []
f3 = Pred "r" []
f4 = Pred "s" []

skDrinker = skSequent ([],[drinkerParadox])
proofDrinker = fromJust (proof 10 $ skDrinker)
([],expDrinker:[]) = extract proofDrinker


expDeskDrinker = desk True 0 [] drinkerParadox expDrinker

test = rmTF ([],[drinkerParadox]) (rm $ pr ([],[expDeskDrinker]))




main = do
  putStrLn $ show $ test
```