Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Logic and Automatic Proofs

Converting Higher-Order Modal Logic Problems into Classical Higher-Order Logic

Tobias Gleißner
Matrikelnummer: 4625710
gleissner.tobias@googlemail.com

Erstgutachter: Christoph Benzmüller
Zweitgutachter und Betreuer: Alexander Steen

Berlin, December 24, 2016

Abstract

Higher-order modal logic is a framework for modeling a large variety of problems and is applied in philosophy and computer science. Automating reasoning in this framework could be very beneficial. Unfortunately there exists no software which achieves this. However automatic reasoning tools for higher-order logic do exist and there is a method which allows expressing higher-order modal logic in terms of higher-order logic. Hence the method enables automatic reasoning in the desired logic. In this thesis a tool which converts a higher-order modal problem into a higher-order problem by applying this method is devised.

Keywords

Higher-Order Modal Logic, Semantical Embedding, Proof Automation

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

December 24, 2016

Tobias Gleißner

Contents

1 Introduction

1.1 Context

Formal systems are used to create abstract representations of various kinds of problems e.g. philosophical questions or logical puzzles among other things. Mathematics is a prominent example which is applied to a broad variety of problems.
The purpose of those representations enables us to gather new information about the properties or behavior of one's object of study. Once a suitable representation is fixed this information can be derived from a set of rules which is usually called calculus. One can try to do this derivation by hand which has been done very successfully since the dawn of arithmetic e.g. in the domain of mathematics.
Anyway there are certain drawbacks deriving this information manually especially for increasingly complex calculi. Subtle mistakes or sometimes even big mistakes may occur in the derivation which go by unnoticed for a long time. An inconsistency in Gödel's ontological argument may serve as an example here [1]. Other reasons are the tedious and mechanical work of transforming mathematical formulas until the desired result is obtained like in is the case in theoretical physics or testing for desired results, inconsistencies or superfluous axioms while developing formal systems. Sometimes sheer complexity of a proof may render the solution to problem too difficult to verify even for a group of experts as it is the case in the proof of Kepler's conjecture, see [2].
One possible solution comes in the form of computer assisted reasoning tools e.g. automated theorem provers (ATPs) which have emerged to overcome the obstacles mentioned above. An ATP is a computer software which takes an abstract representation of a problem as input and delivers information about the problem by constructing proofs of theorems automatically. Prominent examples of ATP's are LEO-II [3], Satallax [4] and Vampire [5]. A reasoning tool worth mentioning is Isabelle [6] which among many other things enables the user state proofs either step by step or with varying granularity and applies ATPs to take care of the details. Most ATPs are based on classical logics as formal systems in which to create the abstract representation. This works quite well for some mathematical theories. However a lot of problems can only be modeled properly or practically within non-classical logics for which only few ATPs exist.
Fortunately it is known that a lot of non-classical logics such as modal logic, free logic, conditional logic, hybrid logic, many-valued logics and others can be simulated within classical higher-order logic by the means of a semantical embedding. Using this method higher-order ATPs are applicable to solve problems modeled with certain non-classical logics. The embedding of modal logic and the application of the embedding with respect to formats offered by

the TPTP project [7] which are the standard in automated theorem proving will be the scope of this thesis.

1.2 Presentation of the problem

Higher-order theorem provers can cope with higher-order logic but not with higher-order modal logic (HOML). The embedding offers a solution which enables higher-order ATPs to proof theorems in any variation of modal logic. A conversion tool which takes the modal problem and converts it to higher-order logic is needed in order to achieve this kind of functionality.
Starting out with a modal logical problem in TPTP's machine readable THF syntax which was adapted to cover the needs of the newly proposed HMF syntax [8] the desired result is to obtain a representation of this problem in THF bearing the same meaning as before but being freed from all occurrences of non-higher-order logic statements.

1.3 Relevance

Relevant modal logic problems to which computer assisted reasoning tools could be applied exist e.g. problems dealing with the concepts of necessity and possibility like in various approaches to the ontological argument of which Gödel's version mentioned before is only one example. Other problem classes may be partial knowledge representation e.g. in the wise men puzzle [9] problems about belief situations, agent logics or problems having a temporal context [10, 11].
However there are only few reasoners for modal logic and none for higher-order modal logic. Since it is extremely tedious and labor-intensive to develop a theorem prover the semantical embedding method offers a convenient shortcut. Also modal logic has several variants e.g. through axiom schemes or different kinds of logical consequence which have to be considered by a modal logic ATP but can be easily simulated with the embedding. Embedded problems can be passed to classical higher-order ATPs which already exist and are further developed.
To motivate the use of ATPs in modal logic we will have a look at its applications. Besides various other topics modal logic is often used to represent philosophical issues. The advantage of such a formal representation despite the capability of a rigorous argumentation is to be able to leave the mechanized steps of reasoning to an ATP and concentrate on modeling a problem. Already mentioned in the introduction are errors in (modal) derivations as they occur unnoticed by the target audience in published papers. These errors could be ruled out if the paper had been formally verified by an ATP capable of handling modal logic.
Another relevant application would be consistency checks on premises called axioms in modal problems in order to confirm the integrity of one's for-

mal representation of a problem. Weaker, superfluous or redundant axioms which might be the cause of a dispute may also be discovered.

1.4 Goals of this thesis

The purpose of this thesis is to write a piece of software which takes a modal logic problem postulated in TPTP's THF syntax and converts it to classical higher-order logic represented in TPTP's THF Syntax by making use of the semantical embedding presented in *Higher-Order Modal Logics: Automation and Applications* [12]. It is intended that the software can be used as a preprocessing step in an actual ATP e.g. as an agent in the Leo III prover [13].
After parsing the problem it has to be converted according to its meaning which is referred to as semantics. For modal logics there are various choices for the semantics of formulas. Initially there will only be support for the semantical setting of rigid constants, constant domains and global consequence but also for different meanings of the modal logical connectives. The meaning of these terms will be presented later in this thesis. If there is still time left to do so other semantical settings will be implemented as well.

1.5 Structure of this Thesis

The remainder of this thesis is structured as follows: Sect. 2 introduces the higher-order logic simple type theory (STT) which is equivalent to the logics higher-order ATPs use. Subsequently higher-order modal logic (HOML) including several variations of this logic is the topic of sect. 3. Next the embedding of HOML in STT is presented in sect. 4. The encoding of logical statements in HOL, HOML and the embedding in the machine readable syntax THF is discussed in sect. 5. In sect. 6 the conversion tool which was developed is presented: Its functionality and external properties are set and a parser is devised. An algorithm and its implementation which automate the embedding process are suggested. Finally the resulting software is evaluated in terms of correctness and performance for the parser and the embedding in sect 7. The thesis concludes in sect. 8 with some further work.

1.6 Notation

There are different equality symbols in the thesis. The following table provides an overview of the symbols to in order to avoid confusion.

$\doteq^{\gamma}$	primitive equality for type γ, this is a symbol of the STT syntax
$\doteq$	same as the above but the type is omitted because the type is obvious
$\equiv$	the left and right sides are identical and can be used interchangeably
$=$	meta equality for definitions in which functions, sets, etc. are used
$\dot{=}^{\gamma}$	embedded primitive equality
$=_{\alpha}$	α-equality, symbol in the theory about STT
$=_{\beta}$	β-equality, symbol in the theory about STT
$=_{\eta}$	η-equality, symbol in the theory about STT

2 Simple Type Theory

Simple type theory (STT) is a an expansion of first order logic (FOL). Its main feature which distinguishes STT from FOL is allowing quantification over arbitrary objects. This offers the possibility to formalize statements not only about individuals but about functions, predicates, sets and relations as well. Therefore STT is called a higher-order logic (HOL). Another feature is the introduction of unnamed functions and other higher-order objects by an abstraction principle which provides simple statements for complex propositions. This will prove useful in the embedding later on.
Most of the definitions and notation are borrowed from *Higher-Order Modal Logics: Automation and Applications* [12] which are based on Church's simple theory of types [14].

2.1 Syntax

STT introduces a concept called types. A type is part of any STT term which puts restrictions on the formation of STT terms.

Definition 1 (Types in STT)
The there are two basic types o and μ and one way to construct further types:

$$\begin{aligned} o &\quad \text{is the type of truth values} \\ \mu &\quad \text{is the type of individuals} \\ \alpha \to \beta &\quad \text{is the function type where } \alpha \text{ and } \beta \text{ are types,} \\ &\quad \to \text{ is called type constructor} \end{aligned}$$

⌟

The types above are the minimal STT type setting. Further atomic types may be introduced when needed making STT a many-sorted logic. The set of types of STT is denoted by $\mathcal{T}$.

Well-formed formulas in STT which in this thesis is referred to as a terms in STT are constructed as follows:

Definition 2 (Terms in STT)
Let $\alpha, \beta, o \in \mathcal{T}$ where o is the type of truth values. The terms of STT are given by the following recursive definition where s, t are terms of STT:

c_α	typed constant
X_α	typed variable
$(\lambda X_\alpha . s_\beta)_{\alpha\to\beta}$	abstraction
$(s_{\alpha\to\beta} t_\alpha)_\beta$	application
$(\neg_{o\to o} s_o)_o$	negation
$(s_o \vee_{o\to(o\to o)} t_o)_o$	disjunction
$(\forall^\alpha_{(\alpha\to o)\to o} \lambda X_\alpha . s_o)_o$	universal quantifier
$(s_\alpha \doteq^\alpha_{\alpha\to(\alpha\to o)} t_\alpha)_o$	primitive equality

⌟

A *constant* always points to the same specific object (individual, predicate, function, connective, other higher-order construction). A *variable* may denote any object but does not represent a particular object unless stated otherwise. An *abstraction* $(\lambda X_\alpha . s_\beta)_{\alpha\to\beta}$ *binds* a variable X_α in its *scope* s_β. It is used to create a function which can take an input argument of a certain type. In order to create functions of higher arity multiple abstractions are used sequentially on its input arguments which is called *currying*:

$$f(x, y) = x + y \quad \text{is encoded as} \quad \lambda x . \lambda y . ((+x)y)$$

An *application* $(s_{\alpha\to\beta} t_\alpha)_\beta$ applies an abstraction $s_{\alpha\to\beta}$ to an input argument t_α. The *negation operator* and the *disjunction* work the same way as in propositional logic. $(\forall_{(\alpha\to o)\to o} \lambda X_\alpha . s_o)_o$ denotes quantification over all variables X_α of any type α. The symbol $\forall_{(\alpha\to o)\to o}$ is called *universal quantifier* or *for all quantifier*. The predicate s_o will be applied to each X_α. If every application evaluates to true the for all expression will evaluate to true and evaluate to false if this is not the case. $\forall X_\alpha . s_o$ can be used as an abbreviation. Primitive equality is defined later as an equivalence relation on HOL terms and their substitutes. Note that the forall quantifier is actually different a different symbol for each type in $\mathcal{T}$. Since it is clear from their subscript type signature which symbol is used the superscript index for distinguishing different symbols for different types is omitted. The same applies to primitive equality.

Using this set of primitive connectives other common connectives can be defined:

$$\begin{aligned}
\phi_o \wedge_{o\to(o\to o)} \psi_o &\equiv \neg_{o\to o}(\neg_{o\to o}\phi_o \vee_{o\to(o\to o)} \neg_{o\to o}\psi_o \\
\phi_o \to_{o\to(o\to o)} \psi_o &\equiv \neg_{o\to o}\phi_o \vee_{o\to(o\to o)} \psi_o \\
\phi_o \leftrightarrow_{o\to(o\to o)} \psi_o &\equiv (\phi_o \to_{o\to(o\to o)} \psi_o) \wedge_{o\to(o\to o)} (\psi_o \to_{o\to(o\to o)} \phi_o) \\
\exists_{(\alpha\to o)\to o}\lambda X_\alpha.s_o &\equiv \neg_{o\to o}(\forall_{(\alpha\to o)\to o}\lambda X_\alpha.\neg_{o\to o}s_o)_o
\end{aligned}$$

2.2 Semantics

The meaning of terms in a logic is in one part understood as a structure which may be restricted in any way in order to model a problem as close as possible. The notion of structure is captured by the concept of a frame which offers options about the domain of individuals, truth values and which objects can map to other objects.

Definition 3 (Frame in STT)
A Frame is a collection $\{D_\alpha\}_{\alpha\in\mathcal{T}}$ of non-empty sets D_α where $\mathcal{T}$ is the set of types such that

- D_μ the domain of individuals can be chosen freely and must not be empty
- $D_o = \{\top, \bot\}$ is the set of truth values
- $D_{\alpha\to\beta}$ are collections of functions mapping D_α to D_β. ⌟

Beware that the domains of all further introduced atomic types must not be empty and have to be part of the frame as well.
Furthermore constant symbols have to be assigned a meaning. This is achieved by adding an *interpretation function* which maps constant symbols to its meaning in a frame.

Definition 4 (Model in STT)
A *Model M* for STT is a tuple (D, I) where

- D is a frame
- I is called *interpretation function* mapping constant symbols c_α to elements of D_α. Logical connectives are given the standard denotations e.g.
  $$I(\neg_{o\to o}) = x \mapsto \begin{cases} \bot & \text{for } x = \top \\ \top & \text{for } x = \bot \end{cases}$$ ⌟

In order to talk about variables and the meaning of of STT terms it must be possible to assign objects to variables.

Definition 5 (Variable assigment)
A variable assignment g maps a variable X_α of any type α to a corresponding element of D_α, the set of possible objects of type α. Note that there is only one assignment g which takes care of all types and not multiple assignments for each type. $g[d_\alpha/Y_\alpha]$ denotes the same assignment as g, except for variable Y_α which is substituted by d_α. ⌟

In order to complete a model for a problem all the pieces from above are conflated into a *valuation function* which assigns a meaning to all terms of STT.

Definition 6 (Valuation function in STT)
The *value* $\|u_\alpha\|^{M,g}$ of a STT term u_α in a model $M = (D, I)$ under an assignment g is an element $d \in D_\alpha$ is defined by the *valuation function* $\|\cdot\|^{M,g}$ as follows:

$$
\begin{aligned}
\|c_\alpha\|^{M,g} &= I(c_\alpha) \\
\|X_\alpha\|^{M,g} &= g(X_\alpha) \\
\|(s_{\alpha\to\beta}t_\alpha)_\beta\|^{M,g} &= \|s_{\alpha\to\beta}\|^{M,g}(\|t_\alpha\|^{M,g}) \\
\|(\lambda X_\alpha.s_\beta)_{\alpha\to\beta}\|^{M,g} &= f \quad \text{where } f : D_\alpha \to D_\beta \text{ is a function} \\
&\qquad \text{s.t. } f(d) = \|s_\beta\|^{M,g[d/X_\alpha]} \text{ for } d \in D_\alpha \\
\|(\neg_{o\to o}s_o)_o\|^{M,g} &= \top \quad \text{iff } \|s_o\|^{M,g} = \bot \\
\|((\vee_{o\to(o\to o)}s_o)t_o)_o\|^{M,g} &= \top \quad \text{iff } \|s_o\|^{M,g} = \top \text{ or } \|t_o\|^{M,g} = \top \\
\|(\forall_{(\alpha\to o)\to o}(\lambda X_\alpha.s_o)_{\alpha\to o})_o\|^{M,g} &= \top \quad \text{iff } \|s_o\|^{M,g[d/X_\alpha]} = \top \text{ for } d \in D_\alpha \\
\|a_\alpha \doteq b_\alpha\|^{M,g} &= \top \quad \text{iff } \|a_\alpha\|^{M,g} = \|b_\alpha\|^{M,g}
\end{aligned}
$$
⌟

Constants c_α are evaluated by the interpretation function I of the model M. Free variables X_α are evaluated by their current assignment $g(X_\alpha)$. Application evaluates to the evaluation of the term $s_{\alpha\to\beta}$ on the left side applied to the evaluation of the argument t_α on the right side. Both $s_{\alpha\to\beta}$ and t_α may be any term and its evaluation is done by successively using the rules in this definition. Abstraction means the term $\|(\lambda X_\alpha.s_\beta)_{\alpha\to\beta}\|^{M,g}$ is exactly the function $f : D_\alpha \to D_\beta$ which takes a d_α from its input domain D_α as argument and evaluates to the $\|s_\beta\|^{M,g}$ which has all occurring X_α already substituted with d_α. This semantic makes the construction of mathematical function objects possible. Negation of true evaluates to false and vice versa. The disjunction evaluates to true if at least one operand evaluates to true and evaluates to false otherwise. The universal quantification evaluates to true if s_o evaluates to true when substituting d_α for X_α for every $d_\alpha \in D_\alpha$. It evaluates to false otherwise. Equality evaluates to true if both operands are equal under the assignment g in the model M under an assignment g.

Definition 7 (Truth and validity in STT)

$M, g \vDash^{STT} s_o$	A term s_o is *true* in a model M under an assignment g iff $\|s_o\|^{M,g} = \top$.
$M \vDash^{STT} s_o$	A term s_o is *valid in a model* M iff $M, g \vDash^{STT} s_o$ for all assignments g.
$\vDash^{STT} s_o$	A term s_o is *valid* iff $M \vDash^{STT} s_o$ for all models M.
$\vDash^{STT} \phi$	A set ϕ of terms is valid iff $\vDash^{STT} s_o$ for all $s_o \in \phi$. ⌟

The desire to obtain new valid statements demands a notion of logical consequence.

Definition 8 (Logical consequence in STT)
Let ϕ be a set of STT terms and s_α be a STT term. s_α is a *logical consequence* of ϕ denoted by

$$\phi \vDash^{STT} s_\alpha$$

iff it holds that for each model M that

$$M \vDash^{STT} \phi \quad \text{implies} \quad M \vDash^{STT} s_\alpha$$ ⌟

The meaning of terms in STT is composed of the definitions in this section and put together in two flavors which are called standard semantics and Henkin semantics.

Definition 9 (Standard semantics in STT)
Standard semantics captures all the definitions stated before. A Frame in standard semantics has the following properties:

D_μ	The domain of individuals.
$D_o = \{\top, \bot\}$	The set of truth values is fixed.
$D_{\alpha\to\beta} = D_\alpha \times D_\beta$	The set of predicates and functions is maximal.

The valuation function $\|\cdot\|$ has to remain total by the choice the frame. ⌟

A model with standard semantics is also called a *standard model*. A more general kind of semantics is offered by Henkin semantics.

Definition 10 (Henkin semantics in STT)
Henkin semantics captures all the definitions stated before. A Frame in Henkin semantics has the following properties:

D_μ	The domain of individuals.
$D_o = \{\top, \bot\}$	The set of truth values is fixed.
$D_{\alpha\to\beta} \subseteq D_\alpha \times D_\beta$	The set of predicates and functions.

The valuation function $\|\cdot\|$ has to remain total by the choice the frame. ⌟

A model with Henkin semantics is called a *Henkin model*. For the rest of the thesis STT will have Henkin semantics. The difference between standard and Henkin semantics lies in the set of functions. Since every standard model is obviously a Henkin model there are more valid statements in standard models as in non-standard Henkin models. Therefore valid statements in a Henkin model are also valid statements in its corresponding standard model. From a modified version of Gödel's incompleteness theorem follows that there cannot be a sound and complete calculus for STT with standard semantics [15]. However this is not the case when using STT with Henkin semantics [16].

Theorem 1 (Soundness and completeness of STT)
STT with Henkin semantics has a sound and complete calculus. ⌟

A proof is presented in *The seven virtues of simple type theory* [15].
Finally we talk about the notion of equality and introduce some definitions in order to make it a bit more practical. Since abstraction and quantification bind variables it will be useful later to know the members of the set of bound variables and its complement when talking about inference such as so called β-reduction and other aspects of STT.

Definition 11 (Free variables in STT)
Let c_α be a constant, X_α be a variable and s, t be STT terms. The meta-predicate *Free* for free variables is defined in the following way:

$$\begin{aligned} Free(c_\alpha) &= Free(\neg_{o\to o}) = Free(\vee_{o\to(o\to o)}) \\ &= Free(\forall_{(\alpha\to o)\to o}) = Free(=_{\alpha\to(\alpha\to o)}) = \emptyset \\ Free(X_\alpha) &= \{X_\alpha\} \\ Free((\lambda X_\alpha . s_\beta)_{\alpha\to\beta}) &= Free(s_\beta) \setminus \{X_\alpha\} \\ Free((s_{\alpha\to\beta} t_\alpha)_\beta) &= Free(s_{\alpha\to\beta}) \cup Free(t_\alpha) \end{aligned}$$ ⌟

Constants, primitive connectives and quantifiers do not have free variables. All occurring variables are considered free variables until ruled out by a λ-abstraction. All free variables in the scope of a λ-abstraction are the free variables of the abstraction with exception of the input argument. Free variables of function application are the free variables of the function and the free variables of the applied argument.

Definition 12 (Bound variables in STT)
Let c_α be a constant, X_α be a variable and s, t be STT terms. The meta-predicate *Bound* for bound variables is defined in the following way:

$$\begin{aligned} Bound(c_\alpha) &= Bound(\neg_{o\to o}) = Bound(\vee_{o\to(o\to o)}) \\ &= Bound(\forall_{(\alpha\to o)\to o}) = Bound(=_{\alpha\to(\alpha\to o)}) = \emptyset \\ Bound(X_\alpha) &= \emptyset \\ Bound((\lambda X_\alpha.s_\beta)_{\alpha\to\beta}) &= \{X_\alpha\} \cup Bound(s_\beta) \\ Bound((s_{\alpha\to\beta}t_\alpha)_\beta) &= Bound(s_{\alpha\to\beta}) \cup Bound(t_\alpha) \end{aligned}$$ ⌟

Constants, primitive connectives and quantifiers do not have bound variables. None of the occurring variables are considered bound variables until ruled out by a λ-abstraction. All bound variables in the body of a λ-abstraction are bound variables of the abstraction. The abstraction's argument variable is a bound variable as well. Bound variables of function application are the bound variables of the function and the bound variables of the applied argument.
The following definitions will introduce different types of inference which are used later when talking about the notion of equality. In order to do so a principle for substituting terms for variables has to be defined.

Definition 13 (Substitution in STT)
Substitution of a term s_α for a term x_α in t_β is denoted by $[s_\alpha/x_\alpha]t_\beta$. Substitution is only allowed if any term in s_α does not occur as a bound variable in t_β. We avoid hereby name clash which is also referred to as *variable capture*. Variable capture can also be avoided by applying α-conversion which is defined later to the bound variables causing the name clash. If there is no variable capture substitution is defined as follows:

$$\begin{aligned} [s_\alpha/x_\alpha]t_\beta &= t_\beta \quad \text{if } t_\beta \in \{c_\beta, \neg_{o\to o}, \vee_{o\to(o\to o)}, \forall_{(\gamma\to o)\to o}\}, \\ &\qquad c_\beta \text{ is a constant symbol} \\ [s_\alpha/x_\alpha]x_\alpha &= s_\alpha \\ [s_\alpha/x_\alpha]y_\gamma &= y_\gamma \quad \text{if } x_\alpha \text{ does not occur freely in } y_\gamma \\ [s_\alpha/x_\alpha](\lambda z_\gamma.u_\delta)_{\gamma\to\delta} &= (\lambda z_\gamma.[s_\alpha/x_\alpha]u_\delta)_{\gamma\to\delta} \\ [s_\alpha/x_\alpha](u_{\gamma\to\delta}v_\gamma)_\delta &= (([s_\alpha/x_\alpha]u_{\gamma\to\delta})([s_\alpha/x_\alpha]v_\gamma))_\delta \end{aligned}$$ ⌟

Bound variable names are considered irrelevant in STT which is formalized by the α-*conversion* rule.

Definition 14 (α-conversion)
$\lambda x_\alpha.s_\beta =_\alpha \lambda y_\alpha.[y_\alpha/x_\alpha]s_\beta$ if y_α does not occur freely or bound in s_β⌟

Applying a term to a function term is captured in the *β-reduction* rule.

Definition 15 (β-reduction)

$$(\lambda x_\alpha.s_\beta)_{\alpha\to\beta}t_\alpha =_\beta [t_\alpha/x_\alpha]s_\beta$$

$(\lambda x_\alpha.s_\beta)_{\alpha\to\beta}t_\alpha$ is called a *β-redex*. ⌟

An extensionality principle can be stated by the *eta-reduction* which formalizes two functions being equal if for each argument their results are the same.

Definition 16 (η-reduction)

$$\lambda X_\alpha.s_\beta X_\alpha =_\eta s_\beta \quad \text{iff } X_\alpha \text{ is not free in } s_\beta.$$

$\lambda X_\alpha.s_\beta X_\alpha$ is called a *η-redex*. ⌟

We are now able to talk about the notion of equality in STT.

Definition 17 (Equality in STT)
Let the symbol $=_\gamma$ for a type γ be a relation over STT terms. Let a_γ, b_γ, t_o be STT terms. We call $=_\gamma$ the *equality relation* and it is defined as:

$$\vDash^{STT} a_\gamma \doteq^\gamma a_\gamma$$
$$a_\gamma \doteq^\gamma b_\gamma, t_o \vDash^{STT} [a_\gamma/b_\gamma]t_o$$ ⌟

This axiomatization of equality is also called *primitive equality*. In order to make equality a little bit more practical one can apply the following rule: For any two formula of which one can be produced by α-conversion, β-reduction or η-reduction of the other, the two formulas are equal in the sense of primitive equality [17].

3 Higher-Order Modal Logic

Higher-order modal logic (HOML) is an extension of higher-order logic particularly STT in this definition of HOML. It offers the possibility to discuss statements under different circumstances e.g. points in time or facts which not everybody knows about.
Most of the definitions and notation are borrowed from *Higher-Order Modal Logics: Automation and Applications* [12] and *TPTP And Beyond: Representation of Quantified Non-Classical Logics* [8]. Explanations of concepts are omitted where they do not differ from STT. Definitions are omitted if they do not differ from STT but are mentioned in the text. The notation of types maybe omitted where types are obvious from now on.

3.1 Syntax

Types in HOML are similar to STT.

Definition 18 (Types in HOML)

σ	is the type of truth values
μ	is the type of individuals
$\alpha \to \beta$	is the function type where α and β are types,
	$\to$ is called type constructor

In order to avoid confusion the type of truth values σ in HOML uses an other symbol as the type of truth values o in STT. The types above are the minimal HOML type setting. Further atomic types may be introduced when needed making HOML a many-sorted logic.
The terms of HOML are an extension of STT by adding the box operator $\Box_{\sigma\to\sigma}$.

Definition 19 (Terms in HOML)
Let $\alpha, \beta, \sigma \in \mathcal{T}$ where σ is the type of truth values. The terms of HOML are given by the following recursive definition where s, t are terms of HOML:

c_α	typed constant
X_α	typed variable
$(\lambda X_\alpha . s_\beta)_{\alpha\to\beta}$	abstraction
$(s_{\alpha\to\beta} t_\alpha)_\beta$	application
$(\neg_{\sigma\to\sigma} s_\sigma)_\sigma$	negation
$(s_\sigma \vee_{\sigma\to(\sigma\to\sigma)} t_\sigma)_\sigma$	or connective
$(\forall_{(\alpha\to\sigma)\to\sigma} \lambda X_\alpha . s_\sigma)_\sigma$	forall quantifier
$(s_\alpha \doteq^{\alpha}_{\alpha\to(\alpha\to\sigma)} t_\alpha)_\sigma$	primitive equality
$(\Box_{\sigma\to\sigma})_\sigma$	box operator

The box operator will be explained in the semantic section of HOML.
Using this set of primitive connectives other common connectives like in STT and the diamond operator $\diamond_{\sigma\to\sigma}$ can be defined:

$$\diamond_{\sigma\to\sigma} s_\sigma \equiv \neg_{\sigma\to\sigma}\Box_{\sigma\to\sigma}\neg_{\sigma\to\sigma} s_\sigma$$

Free and bound variables are the same as in STT and the box operator does not contribute to either since it is a constant symbol.
Substitution is extended from STT by a substitution rule for the box operator.

Definition 20 (Substitution in HOML)
Substitution of a term s_α for a term x_α in t_β is denoted by $[s_\alpha/x_\alpha]t_\beta$. Substitution is only allowed if any term in s_α does not occur as a bound variable in t_β. We avoid hereby name clash which is also referred to as *variable capture*. Variable capture can also be avoided by applying the α-conversion which is defined later to the bound variable. If this is the case substitution is defined as follows:

$$\begin{aligned}
[s_\alpha/x_\alpha]t_\beta &= t_\beta \quad \text{if } t_\beta \in \{c_\beta, \neg_{\sigma\to\sigma}, \vee_{\sigma\to(\sigma\to\sigma)}, \forall_{(\gamma\to\sigma)\to\sigma}\},\\
&\qquad\qquad c_\beta \text{ is a constant symbol}\\
[s_\alpha/x_\alpha]x_\alpha &= s_\alpha\\
[s_\alpha/x_\alpha]y_\gamma &= y_\gamma \quad \text{if } x_\alpha \text{ does not occur freely in } y_\gamma\\
[s_\alpha/x_\alpha](\lambda z_\gamma.u_\delta)_{\gamma\to\delta} &= (\lambda z_\gamma.[s_\alpha/x_\alpha]u_\delta)_{\gamma\to\delta}\\
[s_\alpha/x_\alpha](u_{\gamma\to\delta}v_\gamma)_\delta &= (([s_\alpha/x_\alpha]u_{\gamma\to\delta})([s_\alpha/x_\alpha]v_\gamma))_\delta\\
[s_\alpha/x_\alpha]\Box_{\sigma\to\sigma}t_\sigma &= \Box_{\sigma\to\sigma}[t_\alpha/x_\alpha]t_\sigma
\end{aligned}$$

⌟

α-conversion, β-reduction, η-reduction are the same as in STT. Primitive equality is the same as in STT using the HOML substitution definition.

3.2 Semantics

The underlying structure of HOML is the same as in STT and therefore the definition of a frame in HOML is the same as in STT. The structure is extended with a directed graph. Its nodes are called worlds which represent the circumstances mentioned before and a relation which connects these worlds. This structure is called a *Kripke structure*.

Definition 21 (Kripke structure)
A *Kripke structure* K is a tuple (W, R) in which W is a set of *worlds* and R is a relation on these worlds. R is called *accessibility relation* connecting the worlds. ⌟

From the notion of frame and Kripke structure a model in HOML is defined the following way:

Definition 22 (Model in HOML)
A *Model* M for HOML is a tuple $(W, R, D, \{I_w\}_{w \in W})$ in which (W, R) comprise a Kripke structure where

- W is the set of *worlds*
- R is the *accessibility relation* connecting the worlds
- D is a frame
- $\{I_w\}_{w \in W}$ is a set of so called *interpretation functions* where I_w maps constant symbols c_α to elements of D_α in world $w \in W$. ⌟

Variable assignment is the same as in STT.

Definition 23 (Valuation function in HOML)
The *value* $\|u_\alpha\|^{M,g,w}$ of a HOML term u_α in a model $M = (W, R, D, \{I_w\}_{w \in W})$ under an assignment g is an element $d \in D_\alpha$ which is defined by the *valuation function* $\| \cdot \|^{M,g,w}$ as follows:

$$
\begin{aligned}
\|c_\alpha\|^{M,g,w} &= I_w(c_\alpha) \\
\|X_\alpha\|^{M,g,w} &= g(X_\alpha) \\
\|(s_{\alpha\to\beta} t_\alpha)_\beta\|^{M,g,w} &= \|s_{\alpha\to\beta}\|^{M,g,w}(\|t_\alpha\|^{M,g,w}) \\
\|(\lambda X_\alpha . s_\beta)_{\alpha\to\beta}\|^{M,g,w} &= f \quad \text{where } f : D_\alpha \to D_\beta \text{ is a} \\
&\quad \text{function s.t. for all } d \in D_\alpha \\
&\quad f(d) = \|s_\beta\|^{M,g[d/X_\alpha],w} \\
\|(\neg_{\sigma\to\sigma} s_\sigma)_\sigma\|^{M,g,w} &= \top \quad \text{iff } \|s_\sigma\|^{M,g,w} = \bot \\
\|((\vee_{\sigma\to(\sigma\to\sigma)} s_\sigma) t_\sigma)_\sigma\|^{M,g,w} &= \top \quad \text{iff } \|s_\sigma\|^{M,g,w} = \top \\
&\quad \text{or } \|t_\sigma\|^{M,g,w} = \top \\
\|(\forall_{(\alpha\to\sigma)\to\sigma}(\lambda X_\alpha . s_\sigma)_{\alpha\to\sigma})_\sigma\|^{M,g,w} &= \top \quad \text{iff } \|s_\sigma\|^{M,g[d/X_\alpha],w} = \top \\
&\quad \text{for all } d \in D_\alpha \\
\|a_\alpha \doteq b_\alpha\|^{M,g} &= \top \quad \text{iff } \|a_\alpha\|^{M,g} = \|b_\alpha\|^{M,g} \\
\|(\Box_{\sigma\to\sigma} s_\sigma)_\sigma\|^{M,g,w} &= \top \quad \text{iff for all } v \in W \text{ with } wRv \\
&\quad \text{it holds that } \|s_\sigma\|^{M,g,v} = \top
\end{aligned}
$$
⌟

Constants c_α in a world $w \in W$ are evaluated by the corresponding interpretation function I_w. Free variables X_α are evaluated by their current assignment $g(X_\alpha)$. Application evaluates to the evaluation of the term $s_{\alpha\to\beta}$ in the world w on the left side applied to the evaluation of the argument t_α

in the world w on the right side. Both $s_{\alpha\to\beta}$ and t_α may denote any term and its evaluation is done by successively using the rules in this definition. Abstraction means that in the world w the term $\|(\lambda X_\alpha.s_\beta)_{\alpha\to\beta}\|^{M,g,w}$ is exactly the function $f : D_\alpha \to D_\beta$ which takes a d_α from its input domain D_α as argument and evaluates to the $\|s_\beta\|^{M,g,w}$ which has all occurring X_α already substituted with d_α. This semantic makes the construction of mathematical function objects possible. Negation of true in world w evaluates to false in world w and vice versa. The or connective in world w evaluates to true if at least one operand in world w evaluates to true and it evaluates to false otherwise. The universal quantification evaluates to true if in world w, s_σ evaluates to true when substituting d_α for X_α for every $d_\alpha \in D_\alpha$. It evaluates to false otherwise. Equality evaluates to true if both operands are equal in world w under the assignment g in the model M. The box operator construction $\|(\Box_{\sigma\to\sigma}s_\sigma)_\sigma\|^{M,g,w}$ evaluates to true iff all accessible worlds v from this world w evaluate the statement s_σ in the model M under assignment g to true. It evaluates to false otherwise.

Definition 24 (Truth and validity in HOML)

$M, g, w \models^{HOML} s_\sigma$	A term s_σ is *true* in a model M for a world w under an assignment g iff $\|s_\sigma\|^{M,g,w} = \top$.
$M, w \models^{HOML} s_\sigma$	A term s_σ is *locally valid* in a model M for a world w iff $M, g, W \models^{HOML} s_\sigma$ for all assignments g.
$M \models^{HOML} s_\sigma$	A term s_σ is *globally valid* in a model M iff $M, w \models^{HOML} s_\sigma$ for all worlds w.
$\models^{HOML} s_\sigma$	A term s_σ is *valid* iff $M \models^{HOML} s_\sigma$ for all models M.
$\models^{HOML} \phi$	A set ϕ of terms is valid iff $\models^{HOML} s_\sigma$ for all $s_\sigma \in \phi$. ⌟

There are two different kinds of logical consequence which will be discussed in section 3.4.2.

Definition 25 (Local logical consequence in HOML)
Let ϕ be a set of HOML terms and s_α be a HOML term. s_α is a *local logical consequence* of ϕ denoted by

$$\phi \models^{HOML}_{local} s_\alpha$$

iff it holds that for each model M and each world w that

$$M, w \models^{HOML} \phi \quad \text{implies} \quad M, w \models^{HOML} s_\alpha$$ ⌟

Definition 26 (Global logical consequence in HOML)
Let ϕ be a set of HOML terms and s_α be a HOML term. s_α is a *local logical consequence* of ϕ denoted by

$$\phi \vDash^{HOML}_{global} s_\alpha$$

iff it holds that for each model M that

$$M \vDash^{HOML} \phi \quad \text{implies} \quad M \vDash^{HOML} s_\alpha$$ ⌟

Standard and Henkin semantics are defined in the same way as in STT. For the rest of the thesis STT will have Henkin semantics. HOML with Henkin semantics is also called *Kripke semantics*.

Theorem 2 (Soundness and completeness of HOML)
HOML with Henkin semantics has a sound and complete calculus. ⌟

A proof is presented in *Higher-Order Modal Logic* by Reinhard Muskens [18].

3.3 Variations

There are a lot of different ways to setup HOML in order to represent a problem adequately. This section will look into the manipulation of the accessibility relation as well as the variations concerning logical consequence, constants and domains and show what options are available. Note that different variations can be combined.

3.3.1 Axiom Systems

The graph structure of HOML allows the formulation of properties which model or a class of models should have through the accessibility relation altering their Kripke structure. This is done by introducing axioms along with the problem.

Name	Axiom	Relation condition
K	$\Box(s \to t) \to (\Box s \to \Box t)$	-
T	$\Box s \to s$	reflexive
B	$s \to \Box \diamond s$	symmetric
D	$\Box s \to \diamond s$	serial
4	$\Box s \to \Box\Box s$	transitive
5	$\diamond s \to \Box \diamond s$	euclidean

Axiom K holds for all terms s, t since if $s \to t$ is true in all accessible worlds then t is true in all accessible worlds whenever s is true in all accessible worlds. This is not a coincidence because originally the axiom K together with the rule of necessitation were added to classical logic in order to create

modal logic. Since the extension from classical logic to modal logic in this thesis is done by adding a Kripke structure with its semantical definitions axiom K is not really an axiom but a consequence of these definitions. The rule of necessitation is implied by the definition of global logical consequence which is shown in the next section. Therefore the rule of necessitation is not an axiom as well in the setting of this thesis. The reason for not including this rule as an axiom is the discovery of local logical consequence after the original extension from classical logic to modal logic.
Let $M = (W, R, D, \{I_w\}_{w \in W})$ be a model in HOML. The accessibility relation is

reflexive	iff wRw for all $w \in W$
symmetric	iff wRv implies vRw for all $w, v \in W$
transitive	iff for all $w, v, u \in W$ (wRv and vRu) implies wRu
serial	iff for each $w \in W$ there is some $v \in W$ such that wRv
euclidean	iff for all $w, v, u \in W$ (wRv and wRu) implies vRu, which also implies uRv

The equivalences of the axioms and the relation conditions are proofed in correspondence theorems [19, Chapter 4] [20, Chapter 9] [21, Chapter 7].
These are popular axiom systems which are very often used for modelling problems:

System	Axioms	Relation condition
K	K	-
T	K + T	reflexive
D	K + D	serial
S4	K + T + 4	reflexive, transitive
S5	K + T + 5	reflexive, euclidean, transitive (implicitly), symmetric (implicitly)

Note that the properties euclidean + reflexivity lead to symmetry and transitivity. The property euclidean results from symmetry and transitivity. Beware of the naming confusion between an axiom name and a system name.

Example 1 (Simple time model)
An example for the application of axiom schemes is a model for time. Points in time are represented by worlds and the connections in between are advances in time. In this model time can advance in arbitrary leaps. The box operator models that an event s takes place at every point after the current point in time which is captured by $\Box s \rightarrow \Box\Box s$ and therefore must be transitive. Since the diamond operator is defined by $\diamond s = \neg\Box\neg s$ the operator means it is not the case that the negation of s is taking place at all future points in time. In other words s takes place at some point in the future. The axiom scheme in this example is 4. ⌟

3.3.2 Variations concerning logical consequence

The difference between local and global logical consequence is located in the quantification over worlds in its definitions. Local consequence has the world quantification over both premises and consequence at the same time. There is only a consequence if the premises are valid in the current world w and the consequence is valid in w. Note that for the implication only ϕ in the current world w is used in order to obtain the consequence s_α. Global consequence needs the premises to be globally valid ($M \vDash^{HOML} \phi$) and the consequence will be globally valid again ($M \vDash^{HOML} s_\alpha$). For each world in which s_α should be valid in order to be globally valid the premises in all worlds can be used to draw this implication regardless of the world for which it should hold. Therefore global consequence is the strictly weaker statement compared to local consequence. As a side effect the deduction theorem does not hold if global logical consequence is considered [22].

Theorem 3 (Deduction theorem)
Let Δ be a set of HOML formulas of type σ and $A_\sigma, B\sigma$ HOML formulas. If B_σ is a local logical consequence of $\Delta \cup \{A_\sigma\}$ the following holds:

$$\Delta \cup \{A_\sigma\} \vdash B_\sigma \quad \text{implies} \quad \Delta \vdash A_\sigma \rightarrow B_\sigma$$ ⌟

A proof is presented in *Does the deduction theorem fail for modal logic?* [22]. In order to point out the difference between the two choices of logical consequence one can look at the rule of necessitation:

Example 2 (Rule of necessitation)
$p \vDash^{HOML}_{global} \Box p$ is a theorem because p already holds in all worlds and therefore it holds in all accessible worlds of any world which is the definition of global logical consequence. However $p \vDash^{HOML}_{local} \Box p$ is a non-theorem. A counter model is easily found having two worlds w_1, w_2, accessibility relation $R = \{(w_1, w_2)\}$ and p is valid in w_1 but not in w_2. The world w_2 which is accessible from w_1 must have p as a valid theorem otherwise one would not have local logical consequence but this is not the case. ⌟

3.3.3 Variations concerning constants

Constants usually denote the same term while studying a model or a class of models regardless of the current world. This property is captured by the definition of rigid constants.

Definition 27 (Rigid constant in HOML)
Let $M = (W, R, D, \{I_w\}_{w \in W})$ be a model in HOML. A constant c_α in this model M is called *rigid* iff for all worlds $v, w \in W$ it holds that

$$\|c_\alpha\|^{M,g,w} = \|c_\alpha\|^{M,g,v}$$ ⌟

Example 3 (Character roles in games)
In the party game mafia exist the character roles bar man or doctor which are controlled by human players [23]. Therefore these characters are constants denoting an individual which is a human player. Since it is the same person playing the character for the entire game these constants denote the same individuals (players) throughout the entire game and therefore are rigid constants. ⌟

However a constant does not have to denote the same object in every world but can allow some flexibility.

Definition 28 (Flexible constant in HOML)
Let $M = (W, R, D, \{I_w\}_{w \in W})$ be a model in HOML. A constant c_α in this model M is called *flexible* iff it is not the case that the constant is rigid. ⌟

Example 4 (Rulers of countries over time)
Having a king in a country F may be represented by a constant *king* which denotes a specific human individual. Assuming the kind of government does not change over time and humans are mortal the *king* of country F will change at some point in the future. Therefore the denotation of the constant *king* has to change as well. This is possible by making *king* a flexible constant. ⌟

3.3.4 Variations concerning domains

The domain of individuals is usually the same set in any world. This property is captured by the definition of constant domains

Definition 29 (Constant domains)
Let $M = (W, R, D, \{I_w\}_{w \in W})$ be a model in HOML. Since all domains in the Frame D are fixed M is a model with *constant domains*. ⌟

Example 5 (Country development)
The development e.g. economically or socially of countries over time can be modelled asserting a fixed set of countries which will not perish. This set provides the constant domain of individuals. ⌟

The quantifier ranges on any type may not be constant but depend on a world.

Definition 30 (Varying domains)
Let $M = (W, R, D, \{I_w\}_{w \in W})$ be a model in HOML. The definition of a frame will be altered to depend on worlds in order to obtain *varying domains*. A Frame $\{D_w\}_{w \in W}$ is a family of collections $D_w = \{D_\alpha^w\}_{\alpha \in \mathcal{T}}$ of sets D_α where $\mathcal{T}$ is the set of types and $w \in W$ such that

- D_μ^w the domain of individuals can be chosen freely and must not be empty
- $D_\sigma^w = \{\top, \bot\}$ is the set of truth values
- $D_{\alpha \to \beta}^w$ are collections of functions mapping D_α^w to D_β^w.

The quantifiers are now world-dependant and therefore get a world superscript. The valuation function on the quantifiers is extended accordingly. The tuple $(W, R, \{D_w\}_{w \in W}, \{I_w\}_{w \in W})$ is now a model in HOML with varying domains. ⌟

Beware that the domains of all further introduced atomic types must not be empty and have to be part of the frame as well.

Example 6 (Population of a country)
The population of a country over time can be modelled with its citizens as domain of individuals which varies over time. ⌟

The varying domains model can be restricted to having a certain order of variation in domains. The domains may increase or decrease on their Kripke structure.

Definition 31 (Increasing domains)
Let $M = (W, R, \{D_w\}_{w \in W}, \{I_w\}_{w \in W})$ be a model in HOML with varying domains. Its domains are *cumulative* or *decreasing* if for all D_w, D_v where $w, v \in W$ it holds that

$$wRv \quad \textit{implies} \quad D_w \subseteq D_v$$ ⌟

Example 7 (Stamp collection)
Hannibal has got a stamp collection. He receives a lot of letters from around the world and keeps every stamp except for duplicates which he throws away because he is irritated by owning things he cannot distinguish. In such a way his collection of stamps increases over time. He thinks about modelling his stamp collection using modal logic and cumulative domains which fit modelling the collection and his needs for insight about the properties of his stamp collection perfectly. ⌟

Definition 32 (Decreasing domains)
Let $M = (W, R, \{D_w\}_{w \in W}, \{I_w\}_{w \in W})$ be a model in HOML with varying domains. Its domains are *cumulative* or *decreasing* if for all D_w, D_v where $w, v \in W$ it holds that

$$wRv \quad implies \quad D_v \subseteq D_w$$ ⌟

Example 8 (Process modelling)
Starting out with a certain number of tasks for a problem the process of coping with these tasks maybe modelled with modal logic having decreasing domains where the domains consists of the tasks. This maybe useful where parallelism plays an important role and certain tasks can only be started,finished or continued under certain circumstances. ⌟

3.3.5 Generalizations

A generalization of modal logic which comes to mind is to have not only one but multiple graph structures with the same worlds. This is achieved by having multiple relations in a modal logic model. Each relation then corresponds to an indexed box operator $\Box_i$. This not only allows to have multiple structures but also a different set of properties on each structure. This logic is called *multimodal logic*.

Example 9 (Partial knowledge)
In order to model knowledge of truths of multiple perfect reasoning agents one can define a box operator for the knowledge of each agent and one for common knowledge. A world is then a state of an agent in which he knows certain things. The connections between these states can be interpreted as ways or paths through time in order to obtain more knowledge. The box operator will be assigned certain properties in order to achieve a good model for partial knowledge [24]. ⌟

A further generalization would be to let the nodes vary across the structures instead of having the exact same worlds in every structure.

4 Embedding of Higher-Order Modal Logic in Simple Type Theory

The idea behind embedding a HOML term into HOL is to introduce a new type ι for the worlds and abstract the HOML expressions from the worlds. This process is called *lifting*. It is achieved by lifting the types first of all. Most of the definitions and notation are borrowed from *Higher-Order Modal Logics: Automation and Applications* [12] and *Tutorial on Reasoning in Expressive Non-Classical Logics with Isabelle/HOL* [9].
First a new type ι is introduced which is associated with the domain of worlds. From this any HOML type can be mapped to a certain HOL type.

Definition 33 (Associated HOL type of a HOML type)
Let α, β be HOML types and μ denote the type of individuals and σ denote the type of truth values of HOML. For each HOML type γ the *associated raised HOL type* $\lceil\gamma\rceil$.

$$\begin{aligned} \lceil\mu\rceil &= \mu \\ \lceil\sigma\rceil &= \iota \to \sigma = \rho \\ \lceil\alpha \to \beta\rceil &= \lceil\alpha\rceil \to \lceil\beta\rceil \end{aligned}$$ ⌟

Note that the definition of $\lceil\mu\rceil$ makes constants rigid. Other base types may be defined similarly. Defining $\lceil\sigma\rceil$ like this makes truth values of HOML terms dependent on the world. ρ is used as an abbreviation for $\lceil\sigma\rceil$.

The definition of the valuation function shows all classes of terms which have to be type-raised in order to type-raise all HOML terms.

Definition 34 (Type-raised HOML terms)
The type-raised HOML term $\lceil u_\gamma \rceil$ of a HOML term u_γ is defined as follows:

$$\begin{aligned}
\lceil c_\alpha \rceil &= c_{\lceil \alpha \rceil} \\
\lceil X_\alpha \rceil &= X_{\lceil \alpha \rceil} \\
\lceil (s_{\alpha\to\beta} t_\alpha)_\beta \rceil &= \lceil s_{\alpha\to\beta} \rceil \lceil t_\alpha \rceil \\
\lceil (\lambda X_\alpha . s_\beta)_{\alpha\to\beta} \rceil &= (\lambda \lceil X_\alpha \rceil . \lceil s_\beta \rceil \\
\lceil (\neg_{\sigma\to\sigma} s_\sigma)_\sigma \rceil &= (\dot{\neg}_{\rho\to\rho}\ \lceil s_\alpha \rceil) \\
\lceil ((\vee_{\sigma\to(\sigma\to\sigma)} s_\sigma) t_\sigma)_\sigma \rceil &= ((\dot{\vee}_{\rho\to(\rho\to\rho)}\ \lceil s_\alpha \rceil) \lceil t_\alpha \rceil) \\
\lceil ((\wedge_{\sigma\to(\sigma\to\sigma)} s_\sigma) t_\sigma)_\sigma \rceil &= ((\dot{\wedge}_{\rho\to(\rho\to\rho)}\ \lceil s_\alpha \rceil) \lceil t_\alpha \rceil) \\
\lceil (\forall_{(\alpha\to\sigma)\to\sigma} (\lambda X_\alpha . s_\sigma)_{\alpha\to\sigma})_\sigma \rceil &= (\dot{\forall}_{(\alpha\to\rho)\to\rho}\ (\lambda \lceil X_\alpha \rceil . \lceil s_\beta \rceil)) \\
\lceil (\exists_{(\alpha\to\sigma)\to\sigma} (\lambda X_\alpha . s_\sigma)_{\alpha\to\sigma})_\sigma \rceil &= (\dot{\exists}_{(\alpha\to\rho)\to\rho}\ (\lambda \lceil X_\alpha \rceil . \lceil s_\beta \rceil)) \\
\lceil ((\doteq^\alpha_{\alpha\to(\alpha\to\sigma)}\ s_\sigma) t_\sigma)_\sigma \rceil &= ((\dot{=}_{\alpha\to(\alpha\to\rho)}\ \lceil s_\alpha \rceil) \lceil t_\alpha \rceil) \\
\lceil (\Box_{\sigma\to\sigma} s_\sigma)_\sigma \rceil &= (\dot{\Box}_{\rho\to\rho}\ \lceil s_\alpha \rceil) \\
\lceil (\Diamond_{\sigma\to\sigma} s_\sigma)_\sigma \rceil &= (\dot{\Diamond}_{\rho\to\rho}\ \lceil s_\alpha \rceil)
\end{aligned}$$ ⌟

This process is similar as to what happens in the definition of the valuation function. The dotted connectives are meant to be lifted and will be defined below. Next new HOML constant symbol $r_{\iota\to(\iota\to\sigma)}$ is introduced in order to deal with the accessibility relation.

Definition 35 (Accessibility relation embedding)
Given an accessibility relation R of a model $M = (W, R, D, \{I_w\}_{w\in W})$ and two worlds w^1_ι, w^2_ι the constant symbol $r_{\iota\to(\iota\to\sigma)}$ which models the accessibility relation is defined in the following way:

$$r_{\iota\to(\iota\to\sigma)} w^1_\iota w^2_\iota = \begin{cases} \top & \text{iff } (w^1_\iota, w^2_\iota) \in R \\ \bot & \text{otherwise} \end{cases}$$ ⌟

With the accessibility relation defined as HOL term the lifted box operator can now be defined. The lifted connectives are defined in the following way:

Definition 36 (Lifted connectives)

$$\begin{aligned}
\dot{\neg}_{\rho\to\rho} &= \lambda s_\rho.\lambda w_\iota.\neg(s_\rho w_\iota)\\
\dot{\vee}_{\rho\to(\rho\to\rho)} &= \lambda s_\rho.\lambda t_\rho\lambda w_\iota.(s_\rho w_\iota \vee t_\rho w_\iota)\\
\dot{\wedge}_{\rho\to(\rho\to\rho)} &= \lambda s_\rho.\lambda t_\rho\lambda w_\iota.(s_\rho w_\iota \wedge t_\rho w_\iota)\\
\dot{\forall}_{(\alpha\to\rho)\to\rho} &= \lambda s_{\alpha\to\rho}.\lambda w_\iota.\forall_{(\alpha\to\sigma)\to\sigma}X_\alpha.s_{\alpha\to\rho}X_\alpha w_\iota\\
\dot{\exists}_{(\alpha\to\rho)\to\rho} &= \lambda s_{\alpha\to\rho}.\lambda w_\iota.\exists_{(\alpha\to\sigma)\to\sigma}X_\alpha.s_{\alpha\to\rho}X_\alpha w_\iota\\
\dot{=}_{\alpha\to(\alpha\to\rho)} &= \lambda s_\alpha.\lambda t_\alpha.\lambda w_\iota.(s_\alpha w_\iota \doteq^{\alpha}_{\alpha\to(\alpha\to\rho)} t_\alpha w_\iota)\\
\dot{\Box}_{\rho\to\rho} &= \lambda s_\rho.\lambda w_\iota\forall_{(\iota\to\sigma)\to\sigma}v_\iota.(\neg(r_{\iota\to(\iota\to\sigma)}w_\iota v_\iota) \vee s_\rho v_\iota)\\
\dot{\Diamond}_{\rho\to\rho} &= \lambda s_\rho.\lambda w_\iota\exists_{(\iota\to\sigma)\to\sigma}v_\iota.((r_{\iota\to(\iota\to\sigma)}w_\iota v_\iota) \wedge s_\rho v_\iota)
\end{aligned}$$ ⌟

Negation takes a term s_ρ which value depends on the world what is represented in its type $\rho = \iota \to \sigma$. It also takes a world w_ι and returns the negation of the term $s_{\iota\to\sigma}$ in the world w_ι. The same happens in the lifted or symbol except it takes two terms s_ρ, t_ρ and applies the or connective instead of negation to both of them via currying. The lifted and operator works the same way as the lifted or operator. Forall takes a term $s_{\alpha\to\rho}$ and substitutes all occurrences of X_α in this term $s_{\alpha\to\rho}$. If this results to true for all X_α in the input world w_ι the forall term evaluates to true. Exist works the same way as forall except the substitute has to evaluate to true for only one X_α in the domain of type α. Equality takes two terms of the same type and checks for equality in the world w_ι. The box operator works takes a term s_ρ and a world w_ι and checks for all worlds v_ι whether there is no connection between w_ι and v_ι or the term s_ρ is valid in this world v_ι. If one or the other is true for all worlds v_ι the box operator evaluates to true for the world w_ι. The diamond operator takes a term s_ρ and a world w_ι and checks if there exists a accessible world v_ι from world w_ι in which s_ρ is valid. The expression evaluates to true if this is the case and to false otherwise.

Definition 37 (Validity)
The operator *valid* checking for validity in model of a type-raised HOML term $s_{\iota\to\sigma}$ is defined as

$$valid = \lambda s_{\iota\to\sigma}.\forall w_\iota.s_{\iota\to\sigma}w_\iota$$ ⌟

The operator *valid* takes a type-raised HOML term and returns true if that term is true when applied to any world w_ι. In other words the non-type-raised HOML term is globally valid if it is locally valid in all worlds of a given model.

Theorem 4 (Soundness and completeness of the embedding)
For any HOML term s_σ it holds that

$$\models^{HOML} s_\sigma \quad \text{if and only if} \quad \models^{STT} valid\ \lceil s_\sigma \rceil_\rho \qquad \lrcorner$$

A proof sketch is presented in *Higher-Order Modal Logics: Automation and Applications* [12, Section 3.3].

4.1 Embedding of Axiom Systems

There are two ways of embedding the axiom systems. Using the properties of the accessibility relation e.g. reflexive is the easier way since its definitions are already given in informal HOL. The other way is to translate the axiomatic form of a property e.g. axiom T which is $\Box s \rightarrow s$ using the rules in the section before. The method which is used here is the first one described. The properties list as

reflexive	iff wRw for all $w \in W$
symmetric	iff wRv implies vRw for all $w, v \in W$
transitive	iff for all $w, v, u \in W$ (wRv and vRu) implies wRu
serial	iff for each $w \in W$ there is some $v \in W$ such that wRv
euclidean	iff for all $w, v, u \in W$ (wRv and wRu) implies vRu, which also implies uRv

Translating these informal properties to formal HOL results in

reflexive	$\forall X_\iota . r X_\iota X_\iota$
symmetric	$\forall X_\iota . \forall Y_\iota . (r X_\iota Y_\iota \rightarrow r Y_\iota X_\iota)$
transitive	$\forall X_\iota . \forall Y_\iota . \forall Z_\iota . ((r X_\iota Y_\iota \wedge r Y_\iota Z_\iota) \rightarrow r X_\iota Z_\iota)$
serial	$\forall X_\iota . \exists Y_\iota . r X_\iota Y_\iota$
euclidean	$\forall X_\iota . \forall Y_\iota . \forall Z_\iota . ((r X_\iota Y_\iota \wedge r X_\iota Z_\iota) \rightarrow r Y_\iota Z_\iota)$

The embedding can be easily extended for multimodal logic by introducing further relations and abstraction of the properties from the relations e.g.

reflexive	$\lambda r . \forall X_\iota . r X_\iota X_\iota$

5 TPTP's Machine Readable Syntax

5.1 THF for STT and HOML

TPTP provides a machine readable format for logics called *typed higher form* (THF)[7]. THF has built in all symbols of STT and HOML except for lambda abstraction which is introduced in the THF extension TH0. The following shows exemplary how TH0 can be applied to encode these logics.

THF sentence

A *THF sentence* has the following form

```
% this is a comment
thf(some_name,some_role,some_thf_formula).
```

The name can be chosen arbitrarily and the most important values for role are *type*, *definition*, *axiom*, *conjecture* and *logic*. A *thf formula* is, dependent on the role of the THF sentence, the declaration of a type or of a constant of some type, the definition of a constant or the statement of an axioms, a conjecture or the semantics of the logic currently used.

Types

THF comes with some built-in types of which the most important are $o for the truth type and $i for type of individuals. Additional types can be introduced.

```
thf( my_type_declaration , type , ( my_fancy_type : $tType ) ).
```

Constants need to be declared and are assigned a type. Note that all constants begin with a lower case letter.

```
thf( my_symbol_declaration , type , (my_symbol : $o ) ).
```

Types can be constructed from other types using the greater symbol as type constructor which associates to the right.

```
thf( my_symbol_declaration , type , (my_symbol : ($i>$o) ) ).
```

Definitions

After declaring the type of a constant it can be defined

```
thf( my_symbol_declaration , type , ( my_symbol : $o ) ).
thf( my_symbol_definition , definition , (my_symbol = ($true) ) ).
```

Connectives

There are already predefined logical connectives like not, and, implication, etc.

```
thf( false_implies_true , conjecture , ( $false => $true ) ).
```

Supported standard connectives are not, equal, or, not-equal, not-and, not-or, implication, reverse-implication, equivalence and not-equivalence as denoted by the following symbols.

```
~ = & | != ~& ~| => <= <=> <~>
```

Lambda abstraction

A lambda abstraction is created by the use of the circumflex symbol followed by a list of bound variables followed by a colon and the body of the abstraction. Bound variables begin with a capital letter and have a certain type which is specified by using a colon after the variable name.

```
thf( my_abstraction_type , type , ( my_abstraction :
( alpha_type > beta_type > $o ) ) ).
thf( my_abstraction_definition , definition , my_abstraction =
( ^ [A:alpha_type,B:beta_type] : ($true) ) ).
```

Application

A term can be applied to some other term using the at symbol as the function application symbol. Just as in lambda calculus application associates to the left.

```
thf( my_abstraction_type , type , ( my_abstraction :
( $o > $o ) ) ).
thf( my_abstraction_definition , definition , my_abstraction =
( ^ [A:$o] : ( ~ (A) ) ) ).
thf( my_conjecture , conjecture , ( my_abstraction @ $false ) ).
```

Quantifiers

A quantification starts with an exclamation mark for universal quantification or a question mark for existential quantification. Similarly to lambda abstraction the symbol is followed by a list of typed variables which is followed by a colon and the body of the quantification.

```
thf( my_conjecture , conjecture , ( ! [A:$o]) : (A) ).
```

Modalities

For HOML with one modality exist the keywords $box and $dia which can be applied to a term of type $o.

```
thf( excluded_middle_is_possible , conjecture ,
( ! [ A : $o ] : ( $dia @ ( A | ~(A) ) ) ) ).
```

5.2 Embedded HOML in THF

Most of the concepts of the embedding can be translated to THF in a straightforward manner since they are already formulated in STT. There are few technical difficulties which occur with the polymorphic operators for quantification and equality since for each operator the embedded pendant is dependent on the type of the operands. Such a dependency which is called polymorphism can be stated in TH1 which extends TH0 but most ATPs cannot deal with this extension end therefore TH0 has to be used.
First a type *w_type* for possible worlds is defined.

```
thf( w , type , ( w_type:$tType ) ).
```

Next the accessibility relation r which takes two worlds can be declared. By definition r returns true if and only if the second world can be reached from the first world.

```
thf( r , type , (r:w_type>w_type>$o) ).
```

The accessibility relation may have certain properties e.g. it might be reflexive. The property definitions of section 4.1 'Embedding of Axiom Systems' can be directly encoded in THF and applied to the relation.

```
thf( mreflexive_type , type , ( mreflexive : (w_type>w_type>$o)>$o ) ).
thf( mreflexive , definition , ( mreflexive =
( ^ [R:w_type>w_type>$o] : ! [A:w_type] : ( R @ A @ A ) ) ) ).
thf( r_mreflexive , axiom , ( mreflexive @ r ) ).
```

The valid operator which grounds formulas of type $\iota \to \sigma$ is directly encoded as well.

```
thf( mvalid_type , type , ( mvalid : (w_type>$o)>$o ) ).
thf( mvalid , definition , ( mvalid =
( ^ [S:w_type>$o] : ! [W:w_type] : ( S @ W ) ) ) ).
```

The encoding of the embedding of logical connectives is straightforward and shown here exemplary for negation and disjunction.

```
thf( mnot_type , type , ( mnot: (w_type>$o)>w_type>$o) ).
thf( mnot , definition , ( mnot = ( ^ [A:w_type>$o,W:w_type] : ~(A@W)))).
thf( mor_type , type , ( mor: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mor , definition , ( mor =
( ^ [A:w_type>$o,B:w_type>$o,W:w_type] : ( (A@W) | (B@W) )))).
```

Quantifier embedding can be easily done in TH1 according to the embedding definition. The symbol exclamation mark followed by the symbol greater means that T is a type which is polymorphic. This is a convenient abbreviation since the definition of the quantification is the same for all types.

```
thf( mforall , type , ( mforall_const :
( !> [T:$tType] : (T>w_type>$o)>w_type>$o ) ) ).
thf( mforall , definition , ( mforall_const =
( ^ [A:T>w_type>$o,W:w_type] : ! [X:T] : (A @ X @ W) ))).
```

However TH0 offers no support for defining symbols that way. Therefore a quantifier constant for each type which occurs as the type of a bound variable has to be defined.

```
thf( mforall_mytype_type , type , ( mforall_mytype :
( ( ( w_type>$o ) > ( w_type > $o ) ) > w_type > $o ) ) ).
thf( mforall_mytype_definition , definition , ( mforall_mytype_type = (
^ [A:(w_type>$o)>w_type>$o,W:w_type] :
( ! [X:((w_type>$o))] : (A @ X @ W) ) ) ) ).
```

The same problem as for quantifiers occurs with the equality symbol which is polymorphic implicitly for the embedding. One solution would be to collect all types of the operands of equality and define the equality symbol for every type as it is the case with quantification. But this would involve expanding all definitions of the problem and calculating the types for all operands which. This can be omitted by exploiting the equality symbol which is polymorphic even in TH0 and the derivability of equality from $\alpha\beta\eta$-equality in Henkin semantics. The idea is that the polymorphic equality quantifies over all possible instantiations of the type of the operands. This means there is an implicit quantification over all worlds if their operands depend on worlds.
The embedded equality symbol

$$(\dot{=}_{\alpha\to(\alpha\to\rho)}\ a_\alpha b_\alpha)_\rho$$

which is defined as

$$(\lambda s_\alpha.\lambda t_\alpha.\lambda w_\iota.(s_\alpha w_\iota \doteq^{\alpha}_{\alpha\to(\alpha\to\rho)}\ t_\alpha w_\iota) a_\alpha b_\alpha)_\rho$$

is equivalent to

$$(\lambda s_\alpha.\lambda t_\alpha.\lambda w_\iota.s_\alpha \doteq^{\alpha}_{\alpha\to(\alpha\to o)}\ t_\alpha)_\rho$$

where ρ is the truth type of the embedding and defined as $\iota \to \sigma$. Note that the lambda abstraction for the world is only there to maintain the embedding's truth type and is not applied anywhere. It is crucial that the bound variable W of the world abstraction does not occur anywhere else since it would have to be α-renamed in order to avoid any variable capture. In TH0 a formula $s = t$ can be translated to

```
^ [W:w_type] : ( s = t )
```

Note that this method could be applied to embedding Quantifiers, too.
An exhaustive list of all embedding definitions including all connectives can be found in Appendix B.

5.3 Example

The following example demonstrates the encoding of a HOML problem and its corresponding embedding in THF . It contains a modal operator, a quantification and connectives of arity one and two.

$$\forall A_\sigma.\, \Diamond_{\sigma\to(\sigma\to\sigma)}\, (A_\sigma \vee_{\sigma\to(\sigma\to\sigma)} \neg_{\sigma\to\sigma} A_\sigma)$$

We assume this problem has the semantics rigid constants, constant domains, global consequence and axiom system S5 for the modality.
The original problem encoded in THF is

```
% define semantics
thf ( simple_s5 , logic , ( $modal := [
    $constants := $rigid ,
    $quantification := $constant ,
    $consequence := $global ,
    $modalities := $modal_system_S5 ] ) ) .

% state problem
thf ( excluded_middle , conjecture ,
    ( ! [ A : $o ] : ( $dia @ ( A | ~ ( A ) ) ) ) ) .
```

Embedding the problem using the steps shown in the previous section results in

```
% declare type for possible worlds
thf( w , type , ( w_type:$tType ) ).

% declare accessibility relations
thf( r , type , (r:w_type>w_type>$o) ).

% define accessibility relation properties
thf( meuclidean_type , type , ( meuclidean : (w_type>w_type>$o)>$o ) ).
thf( meuclidean , definition , ( meuclidean = ( ^ [R:w_type>w_type>$o] :
    ( ! [A:w_type,B:w_type,C:w_type] :
    ( ( (R@A@B) & (R@A@C) ) => (R@B@C) ) ) ) ) ).
thf( mreflexive_type , type , ( mreflexive : (w_type>w_type>$o)>$o ) ).
thf( mreflexive , definition , ( mreflexive = ( ^ [R:w_type>w_type>$o] :
    ( ! [A:w_type] : (R@A@A) ) ) ) ).

% assign properties to accessibility relations
thf( r_meuclidean , axiom , ( meuclidean @ r ) ).
thf( r_mreflexive , axiom , ( mreflexive @ r ) ).

% define valid operator
thf( mvalid_type , type , ( mvalid: (w_type>$o)>$o ) ).
thf( mvalid , definition , ( mvalid = ( ^ [S:w_type>$o] : ! [W:w_type] :
    (S@W) ) ) ).

% define nullary, unary and binary operators which are not quantifiers or
% valid operator
thf( mor_type , type , ( mor: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mor , definition , ( mor = ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] :
    ( (A@W) | (B@W) ) ) ) ).
thf( mnot_type , type , ( mnot: (w_type>$o)>w_type>$o) ).
thf( mnot , definition , ( mnot = ( ^ [A:w_type>$o,W:w_type] :
    ~(A@W) ) ) ).
thf( mdia_type , type , ( mdia: (w_type>$o)>w_type>$o) ).
thf( mdia , definition , ( mdia = ( ^ [A:w_type>$o,W:w_type] :
    ? [V:w_type] : ( (r@W@V) & (A@V) ) ) ) ).

% define for all quantifiers
thf( mforall_const_type__o_w_type_t__d_o_c_ , type ,
    ( mforall_const__o_w_type_t__d_o_c_ : ( ( (w_type>$o) >
    ( w_type > $o ) ) > w_type > $o ) ) ).
thf( mforall_const__o_w_type_t__d_o_c_ , definition ,
    ( mforall_const__o_w_type_t__d_o_c_ =
    ( ^ [A:(w_type>$o)>w_type>$o,W:w_type] :
    ( ! [X:(w_type>$o)] : (A @ X @ W) ) ) ) ).

% transformed problem
thf ( excluded_middle , conjecture , ( mvalid @ ( (
    mforall_const__o_w_type_t__d_o_c_ @ ( ^ [ A : (w_type>$o) ] :
    ( mdia @ ( mor @ ( A ) @ ( mnot @ ( A ) ) ) ) ) ) ) ) ) ) .
```

6 Implementation

6.1 Functionality

The aim is to create a conversion tool which takes a file containing a modal problem consisting of TH0 sentences as input and embed it into STT. The output will be a file containing the corresponding embedded modal problem consisting of TH0 sentences as shown in the section before.
Supported semantics include one modality which may have any of the axiom schemes presented in section '3.3.1 Axiom Systems'. Other variations are not considered hence constants are rigid, domains are constant and logical consequence is global.

6.2 Environment

The tool will be available as a command line tool taking parameters. It should be easily extendable for the variations of modal logic itself and for other logical embeddings.
Since logical formulas can be represented as trees the tool will need a parser which creates an abstract syntax tree (AST) and a procedure for tree manipulations in order to achieve the embedding. Reasonable choices for the parser to create an AST for the input are as follows:

- Custom software with parser generator and tree manipulations for the embedding
- Integration into the TPTP2X tool which converts different formats of logics using translation files for parsing and transformation.
- Making use of the Leo-III Parser and tree manipulations for the embedding

TPTP2X would offer the benefit of a nice integration into the TPTP infrastructure. However converting between different formats and making a semantical embedding are quite different things to do. Therefore separation of concerns by creating a different program is more desirable. The LEO-III Parser is written in Scala and demands learning the programming language in order to be able to use it which cannot be appointed time for in this thesis. All three options involve creating some kind of grammar for parsing modal logical formulas. Using a parser generator such as ANTLR one is not only able to skip the development of the parser itself but to automatically produce a parser for modal logic in various programming languages which might be beneficial for other purposes. For these reasons a custom piece of software using the ANTLR parser generator will be the choice here. The programming language is going to be Java because the tool can easily be incorporated into the LEO-III ATP which is developed in Scala [25] or into

Isabelle which is written in Java and Scala [26]. Other considerations are platform independence as well as compatibility with the LLVM compiler suite.
For an easy build and deployment process Apache Maven is used as build system.

6.3 Command Line Interface

A file containing a modal problem in TH0 including its semantical definitions can be embedded using the tool *EmbedModal*. The input and output file are specified using parameters *-i* and *-o*. These may also be nested directories for transforming multiple problems.
If the modal problem does not include a semantical definition it may be added via the option *-semantics*. The choices are *standard_s5* for rigid, constant, global S5 semantics and *all* for creating multiple embeddings on all available modal systems.
There is also support for Graphviz creating a graphical visualization of the parse trees with options *-dotin*, *-dotout*, *-dotbin* for the Graphviz files for the input problem, the transformed problem and the Graphviz binary for creating the images of both.
An overview and exemplary use can be found in appendix D.

6.4 Architecture

The library for the embedding is divided up into the packages *exceptions*, *parser*, *transformation* and *util*. Figure 1 should give an impression on how the architecture is organized.
An executable may call *parser.ThfAstGen.parse()* in order to receive a *parser.ParseContext* containing the parse tree which is represented as *util.tree.Node* datastructure.
From this tree a *transformation.ModalTransformator* can be constructed that analyzes semantics and performs the embedding.
To make the use of the library a bit more practical there exist wrappers for this whole process which are located in *transformation.Wrappers* which not only allow to pass additional semantical definitions (*transformation.Wrappers.convertModal*) but can also create multiple transformations for a set of semantical definitions (*transformation .Wrappers.convertModalMultipleSemantics*) or traverse a directory (*transformation .Wrappers.convertModalMultipleSemanticsTraverseDirectory*) and output embeddings for problems in this directory and its nested directories. The main class *EmbedModal* actually uses these wrappers for different kinds of inputs. Additional semantical definitions are delivered by the class *transformation.SemanticsGenerator*.

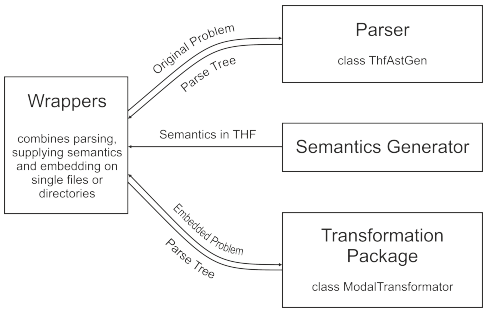


Figure 1: Architecture

The upcoming sections go into detail of parsing and transforming a modal logical problem and point out pieces in the software which are involved.

6.5 ANTLR Generated Parser

The parser generator ANTLR creates a parser from an ANTLR grammar which consists of BNF rules for lexical analysis or tokenization which are stated in capital letters and rules for parsing in lower case letters. The grammar for parsing THF extended with modal operators and semantical definitions is very similar to the original TPTP grammar [7] which looks like this:

```
<thf_formula>          ::= <thf_logic_formula> | <thf_sequent>
<thf_logic_formula>    ::= <thf_binary_formula> | <thf_unitary_formula> |
                           <thf_type_formula> | <thf_subtype>
<thf_binary_formula>   ::= <thf_binary_pair> | <thf_binary_tuple>
<thf_binary_pair>      ::= <thf_unitary_formula> <thf_pair_connective>
                           <thf_unitary_formula>
<thf_binary_tuple>     ::= <thf_or_formula> | <thf_and_formula> |
                           <thf_apply_formula>
<thf_or_formula>       ::= <thf_unitary_formula> <vline> <thf_unitary_formula>
                           | <thf_or_formula> <vline> <thf_unitary_formula>
```

The corresponding ANTLR grammar which is located in *lib_embedding/src/main/antlr4/parser/Hmf.g4* looks like this:

```
thf_formula         : thf_logic_formula
                    | thf_sequent
                    ;
thf_logic_formula   : thf_binary_formula
                    | thf_unitary_formula
                    | thf_type_formula
                    | thf_subtype
                    | logic_defn_element
                    ;
thf_binary_formula  : thf_binary_pair
                    | thf_binary_tuple
                    ;
```

```
thf_binary_pair     : thf_unitary_formula thf_pair_connective
                      thf_unitary_formula
                    ;
thf_binary_tuple    : thf_or_formula
                    | thf_and_formula
                    | thf_apply_formula
                    ;
thf_or_formula      : thf_unitary_formula VLINE thf_unitary_formula
                    | thf_or_formula VLINE thf_unitary_formula
                    ;
```

In most cases the translation from TPTP BNF to ANTLR grammar is straightforward except for nested rules which use the same delimiters for different applications and the lexical rules have to be constructed from scratch in order to avoid conflicts. The resulting grammar can deal with THF formulas without annotations, TPI, numbers, comments which are not in one line and formula selection in include statements. These restrictions on the resulting parser may look inconvenient but they only affect a tiny set of actual THF problems as is shown in the evaluation section.
Using this grammar the ANTLR Maven plugin creates the four classes *parser.HmfBaseListener*, *parser.HmfLexer*, *parser.HmfListener* and *parser.HmfParser*. These classes are used by the method *parser.ThfAstGen .parse* which takes a start rule for parsing and an *ANTLRInputStream* which is a file handler to the problem as arguments. Lexical analysis is performed on this input stream and the result is fed into *HmfParser*. The *HmfParser* also gets a *parser.DefaultTreeListener* which contains callbacks on how to act on finding parser rules, terminal symbols and errors. The start rule is executed on the parser and a tree is constructed from the callbacks. All parse artifacts including the tree's root node and errors are put into a *parser.ParseContext* and returned to the caller. The following outline portrays this parsing process:

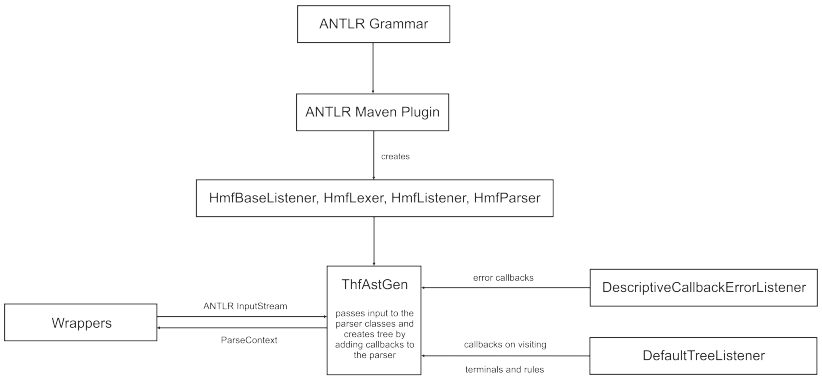


Figure 2: Parsing process

6.6 Embedding Algorithm

An algorithm which corresponds to an embedding with rigid constants, constant domains and global logical consequence should perform the following steps:

i) Sort THF sentences

Group THF sentences by role since type declarations, symbol definitions, logic definitions and the remaining roles are treated differently. Non-THF roles like fi_domain, fi_functors, etc. are ignored.

ii) Extract semantics

Semantics are extracted from THF sentences with the role logic. These sentences are not affected by the remaining steps.

iii) Lift types

Lift all occurring truth type usages in every thf sentence in order to make it depend on the world type. This means the type σ becomes $\iota \to \sigma$. Other types are not lifted since the semantical setting is rigid, constant, global-

iv) Replace operators

Replace all occurring operators e.g. $\top, \neg, \vee, \forall, \Box$ by their embedded pendant in THF sentences of all roles except type declarations because these symbols should not occur there.

v) Apply valid operator

Apply the valid operator to all THF formulas which are neither types nor definitions which means the formulas are axioms, conjectures, negated_conjectures, etc. As a consequence all THF formulas of type $\iota \to \sigma$ are grounded to σ by checking their validity on every world.

vi) Declare and define embedding symbols and types

This includes a type for the possible worlds, the accessibility relation, accessibility relation properties, the valid operator and all other operators.

vii) Assign properties to the accessibility relation according to the extracted semantics

6.7 Embedding Algorithm Implementation

The algorithm from the subsection before is implemented in the *transformation* package. Figure 3 provides an overview of all classes which are part

of the embedding process. All definitions of world type, accessibility relation, accessibility relation properties, embedded operators and the valid operator are stored in *transformation.EmbeddingDefinitions*. An executable can create a *transformation.ModalTransformator* supplying the root node of the problem to transform from a *parser.ParseContext* object. Calling the method *transform* on this transformator applies the following steps:

i) A *transformation.ThfAnalyzer* is created which groups the THF sentences by their roles.

ii) Semantics are extracted from THF sentences of role *logic* by a *transformation.SemanticsAnalyzer* which provides Maps from constants, domains, axioms and modalities to their corresponding semantical properties, e.g. a specific constant *king_of_france* might be flexible, the domain of *population* might be varying, etc. These maps also have default properties if specified by a THF sentence of role *logic*.

The actual transformation is done in the method *transformation.ModalTransformator.actualTransformation()* which traverses the parse tree and applies the steps of the algorithm as follows:

iii) Lift types

All leafs in the parse tree which have the TPTP truth type $\$o$ as payload are replaced by the lifted type $(w\_type > \$o)$ as defined in *EmbeddingDefinitions* for formulas of all roles except for *logic*.

The next steps only affect formulas of all types except for *logic* and *type*.

iv) Substitute all operators by their embedded pendants

Substitute nullary and unary operators

Leafs may contain the nullary or unary operators $\$true, \$false, \$box, \$dia, \sim$ which are replaced in the method *ModalTransformator .replaceNullaryAndUnaryOperators* by their modal version. Truth values are not applied to an expression and can be left as they are in their modal embedding. Since $\$box$ and $\$dia$ are already followed the application operator @ there is no need for further change. The negation operator does not have a trailing @ and therefore has to supply one.

Exemplary use of nullary operator:

```
($true)    becomes    (mtrue)
```

Definition of mtrue:

```
thf( mtrue_type , type , ( mtrue: w_type>$o ) ).
thf( mtrue , definition , ( mtrue = ( ^ [W:w_type] : $true) ) ).
```

Exemplary use of an unary operator:

```
$box @ A   becomes    mbox @ A
~(A)       becomes    (mnot @ A)
```

Definitions of mbox and mnot:

```
thf( mbox_type , type , ( mbox: (w_type>$o)>w_type>$o) ).
thf( mbox , definition , ( mbox = ( ^ [A:w_type>$o,W:w_type] :
    ! [V:w_type] : ( (r@W@V) => (A@V) ) ) ) ).
thf( mnot_type , type , ( mnot: (w_type>$o)>w_type>$o) ).
thf( mnot , definition , ( mnot = ( ^ [A:w_type>$o,W:w_type] :
    ~(A@W) ) ) ).
```

Substitute binary connectives

Binary connectives follow the grammar rules *thf_binary_tuple* or *thf_binary_pair* and are addressed by the methods *ModalTransformator.embed_thf_binary_tuple* and *ModalTransformator .embed_thf_binary_pair*. All connectives except for equality and inequality are treated in the same way. The connective symbol is identified and the embedding symbol is added to the front. Application symbols are places in between the new symbol and the left operand and in place of the connective symbol which is fine since application associates to the left and braces surrounding both operands are inserted. Conjunction and disjunction may occur nested as *thf_binary_tuple* in the parse tree since their compounding does not matter and is omitted in that case. The right child is the right operand which is not nested any further while the left child may be a conjunction or disjunction again.

```
A & B
mand A & B
mand @ A @ B
mand @ (A) @ (B)
```

Substitute equality connective

If a binary connective an equality or inequality connective it is wrapped in brackets and abstracted from an unused variable name of the world type and wrapped in brackets again. The validity of this step is discussed in section 5.2.

```
A = B
( ^ [W:w_type] : (A = B) )
```

Substitute quantifiers

A node of with grammar rule *thf_quantified_formula* triggers the method *ModalTransformator.embed_thf_quantified_formula*. The quantifier symbol is retrieved and removed from the formula just as the variable list

brackets and the colon. The remaining tree is a nested structure of typed variables in which the left child is a typed variable and the right child is the rest of the nested structure. After the comma is removed the typed variable is removed and instead a lambda abstraction for this variable is introduced. Braces are added to avoid conflicts and comply with the standard and the application of the quantification constant for the type of the variable is put in front. These steps are repeated for every typed variable in the nested structure.

```
! [A:a_type,b:b_type] : body
A:a_type , b:b_type body
b:b_type body
^ [A:a_type] : b:b_type body
( ^ [A:a_type] : ( b:b_type body ) )
mforall_a_type @ ( ^ [A:a_type] : ( b:b_type body ) )
mforall_a_type @ ( ^ [A:a_type] : ( ^ [b:b_type] : body ) )
mforall_a_type @ ( ^ [A:a_type] : ( ( ^ [b:b_type] : ( body ) ) ) )
mforall_a_type @ ( ^ [A:a_type] : ( mforall_b_type @ ( ^ [b:b_type] :
    ( body ) ) ) )
```

v) The valid operator is applied to all formulas which are neither definitions nor type declarations and the resulting formulas are surrounded by brackets again.

```
( ( mforall_const__d_i @ ( ^ [ A : $i ] : ( ( ( ^ [ W0 : w_type ] :
    ( A = A ) ) ) ) ) ) )
( mvalid @ ( ( mforall_const__d_i @ ( ^ [ A : $i ] : ( (
    ( ^ [ W0 : w_type ] : ( A = A ) ) ) ) ) ) ) )
```

vi) During these tree manipulations all used embedding symbols are collected and their declarations and definitions which are exemplary used in this sections are added to the resulting problem in the end by the method *TransformModal.getModalDefinitions*.

vii) The accessibility relation properties which were extracted by *transformation.SemanticsAnalyzer* are applied to the relation as already shown in section 5.2 by the method *TransformModal .getModalDefinitions*.

All declarations and definitions are put in order in such a way as they the concerning symbols do not occur in formulas before they are stated because most ATPs cannot sort them themselves. The original semantical statements are removed from the problem and a *transformation.TransformContext* is passed from *ModalTransformator.actualTransformation* to *ModalTransformator.transform* and returned to the caller. The *TransformContext* object contains the root node of the original problem, the root node of the transformed problem and a string with the applied embedding's declarations and definitions

in the right order. *TransformContext* provides a method to combine these fields into a string resulting in the complete embedded problem which can be passed to an ATP system.

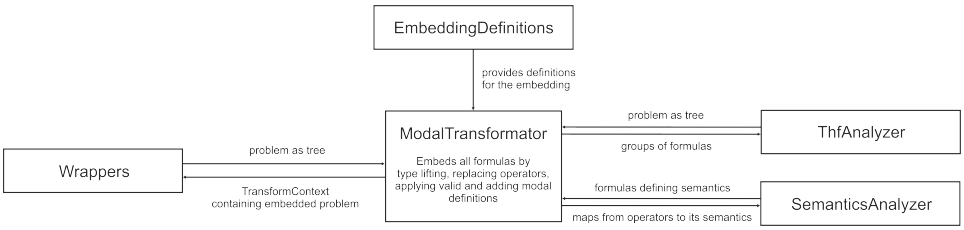


Figure 3: Embedding process

7 Evaluation

In this section correctness and performance tests of both the parser and the embedding process are presented, applied and their results interpreted.
The evaluation largely depends on the use of the ATP Satallax [4] for which the embedding library provides a wrapper *util.external_software_wrappers .SatallaxWrapper*. The wrapper can traverse a directory and check for syntax and type errors, satisfiability or theorem status of a THF problem.

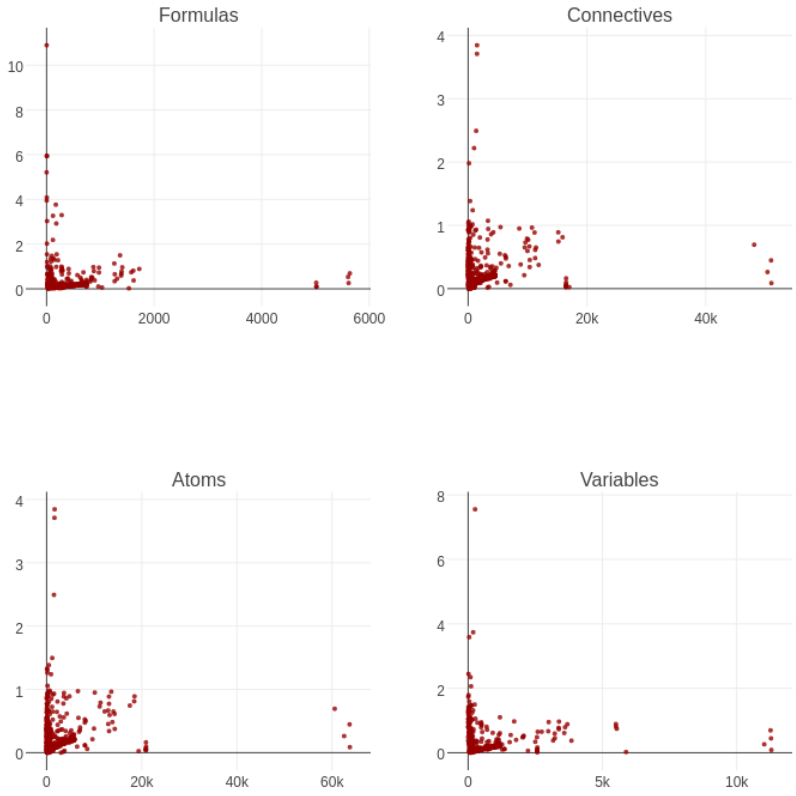
7.1 Parser Correctness

The test of the correctness of the parser mainly relies on the number of problems it can parse. Since the grammar was in most cases translated in a straightforward manner and the parser generator ANTLR is widely applied and tested tool this is a good indicator of correctness. All 3061 THF problems which are currently part of the TPTP can be parsed with the exception of two problems. Two errors occur in file *SYN000^1.p* in which functors with single quotation marks and block comments were identified as not working. In file *SYN000^2.p* distinct objects with their double quotation marks and the exponential notation of real numbers do not work. When removing respecting statements both files work properly. The escaping mechanisms of single and double quoted tokens are expected to malfunction too since this feature was not implemented.
The parse trees were also inspected manually at random for ten different problems by comparing the graphical parse tree outputs with the expected trees. Using this method which is at best an indicator of correctness showed no errors.
Since the development of the parser was not the main topic no further tests were conducted. If one would want to inquire further a comparison test between the parse tree of this parser and the parse tree of a well tested parser e.g. of some ATP system could be performed. The Graphviz format could act as a representation of the parse trees for comparison in particular Graphviz support is already implemented in this parser and would only have to be added to the other parser.

7.2 Parser Performance

The binary *test_parser* also reads the meta data of the problems which is provided by TPTP and measures the performance of the parser for each THF problem of the TPTP. A python script is automatically built which creates plots of the performance for different problem properties provided by the meta data. Since not all meta data is available for all problems the plots may not be directly comparable to each other but they reveal the absolute performance. The test was conducted on a Intel(R) Core(TM) i7-4750HQ

CPU @ 3.2 GHz with four cores and 2x8GB DDR3 RAM @ 1600 MHz .
The performance test shows an average parsing time or 0.055 seconds per problem. The median parsing time is 0.011 seconds The parsing time depending on the number of formulas, connectives, atoms and variables shows a very slow and linear increase. For almost all problems the time is below 1.5 seconds.



Further performance plots can be found in appendix E. This test could be improved by counting the number of formulas, connectives, atoms and variables instead of relying on the meta data.
In a multicore environment the parser's performance could be improved by splitting the problem into THF sentences or even more granular parts and assign them to multiple parsers. This could be achieved by either splitting the problem itself or splitting the token stream which is generated by the lexer before the beginning of every THF sentence.

7.3 Embedding Correctness

To ensure the transformation is implemented correctly different test methods are applied to the embedded pendants of a set of modal logical problems. First of all the problem is checked for syntax and type errors. Furthermore the axioms will be checked for satisfiability. If the problem has a model the embedding should be satisfiable and if there is no model it should be unsatisfiable. Finally problems which are theorems will be identified and their embeddings must be theorems again. The same should be true concerning non-theorems.
All tests are performed using Satallax. The binary *test_problems* which takes an input directory and an output directory as arguments calls the Satallax wrapper which traverses the input directory and creates files containing the concerning problem filenames for all tests which are performed.
The set of modal logical problems consists of some simple conjectures and a formalization of the ontological argument by Gödel.

Simple conjectures

the transitivity of equality

$$\forall A, B, C.((A = B) \wedge (B = C)) \rightarrow (A = C)$$

an equivalence of material implication

$$\forall A, B.(A \rightarrow B) \leftrightarrow (\neg A \vee B)$$

the law of excluded middle

$$\forall A.A \vee \neg A$$
$$\Box\forall A.A \vee \neg A \quad \textit{modal version}$$

the frame properties stated using modal operators e.g. reflexivity

$$\forall A.\Box A \rightarrow A$$

among many others. All these problems are stored in the directory *resources/tptp_files/tiny_problems* of the project.

Gödel's ontological argument

Gödel's ontological argument is a higher-order proof of the existence of god which is understood quite well on a logical level. The proof contains type declarations of propositions and a second-order functor, definitions, both modal operators, second-order quantification and various logical connectives. Therefore it can serve as a good example to demonstrate that the conversion tool is capable of handling higher-order arguments properly. The supplied problem is actually Scott's version of Gödel's ontological argument since the original version by Gödel contains an error [27]:

A1 Either a property or its negation is positive, but not both:
$\forall \phi.P\ (\lambda x.\neg(\phi\ x)) \leftrightarrow \neg(P\ \phi)$

A2 A property necessarily implied by a positive property is positive:
$\forall \phi.\forall \psi.(P\ \phi \wedge \Box\forall x.\phi\ x \rightarrow \psi\ x) \rightarrow P\ \psi$

T1 Positive properties are possibly exemplified:
$\forall \phi.P\ \phi \rightarrow \Diamond\exists x.\phi\ x$

D1 A God-like being possesses all positive properties:
$G\ x \equiv \forall \phi.P\ \phi \rightarrow \phi\ x$

A3 The property of being God-like is positive:
$P(G)$

C Possibly, God exists:
$\Diamond\exists x.G\ x$

A4 Positive properties are necessarily positive:
$\forall \phi.P\ \phi \rightarrow \Box\ P\ \phi$

D2 An essence of an individual is a property possessed by it and necessarily implying any of its properties:
$ess\ \phi\ x \equiv \phi\ x \wedge \forall \psi.(\psi\ x \rightarrow \Box\forall y.(\phi\ y \rightarrow \psi\ y))$

T2 Being God-like is an essence of any God-like being:
$\forall x.G\ x \rightarrow ess\ G\ x$

D3 Necessary existence of an individual is the necessary exemplification of all its essences:
$NE\ x \equiv \forall \phi.ess\ \phi\ x \rightarrow \Box\exists y.\phi\ y$

A5 Necessary existence is a positive property:
$P\ NE$

T3 Necessarily, God exists:
$\Box\exists x.G\ x$

The formalization in TH0 with modal operators is as follows:

```
% positive constant - maps a proposition to boolean
thf( positive_type , type , ( positive: ( $i > $o ) > $o ) ).

% godlike constant - maps an individual to boolean
thf( godlike_type , type , ( godlike: $i > $o ) ).

% essence constant - maps a proposition and an individual to boolean
thf( essence_type , type , ( essence: ( $i > $o ) > $i > $o ) ).
```

```
% necessary existence constant - maps an individual to boolean
thf( ne_type , type , ( ne: $i > $o ) ).

% A1: Either the property or its negation are positive, but not both.
thf( a1 , axiom , ( ! [Phi:$i>$o] : ( ( positive @ ( ^ [X:$i] :
    ( ~ ( Phi @ X ) ) ) ) <=> ( ~ ( positive @ Phi ) ) ) ) ).

% A2: A property necessarily implied by a positive property is positive.
thf( a2 , axiom , ( ! [Phi:$i>$o,Psi:$i>$o] :  ( ( ( positive @ Phi ) &
    ( $box @ ( ! [X:$i] : ( ( ( Phi @ X ) => ( Psi @ X ) ) ) ) ) ) ) =>
    ( positive @ Psi ) ) ) ).

% T1: Positive properties are possibly exemplified.
thf( t1 , conjecture , ( ! [Phi:$i>$o] : ( ( positive @ Phi ) => ( $dia @
    ( ? [X:$i] : ( Phi @ X ) ) ) ) ) ).

% D1: A God-like being possesses all positive properties.
thf( d1 , definition , ( godlike = ( ^ [X:$i] :
( ! [Phi:$i>$o] : ( ( positive @ Phi ) => ( Phi @ X ) ) ) ) ) ).

% A3: The property of being God-like is positive.
thf( a3 , axiom , ( positive @ godlike ) ).

% C: Possibly, God exists.
thf ( c , conjecture , ( $dia @ ( ? [X:$i] : ( godlike @ X ) ) ) ).

% A4: Positive properties are necessary positive properties.
thf( a4 , axiom , ( ! [Phi:$i>$o] :
    ( ( positive @ Phi ) => ( $box @ ( positive @ Phi ) ) ) ) ).

% D2: An essence of an individual is a property possessed by it and
% necessarily implying any of its properties.
thf( d2 , definition , ( essence = ( ^ [Phi:$i>$o,X:$i] :
    ( ( Phi @ X ) & ( ! [Psi:$i>$o] : ( ( Psi @ X ) => ( $box @
    ( ! [Y:$i] : ( ( Phi @ Y ) => ( Psi @ Y ) ) ) ) ) ) ) ) ) ).

% T2: Being God-like is an essence of any God-like being.
thf( t2 , conjecture , ( ![X:$i] : ( ( godlike @ X ) =>
    ( essence @ godlike @ X ) ) ) ).

% D3: Necessary existence of an individual is the necessary exemplification
% of all its essences
thf( d3 , definition , ( ne = ( ^ [X:$i] : ( ! [Phi:$i>$o] :
    ( ( essence @ Phi @ X ) => ( $box @ ( ? [Y:$i] :
    ( Phi @ Y ) ) ) ) ) ) ) ).

% A5: Necessary existence is positive.
thf( a5 , axiom , ( positive @ ne ) ).

% T3: Necessarily God exists.
thf( t3 , conjecture , ( $box @ ( ? [X:$i] : ( godlike @ X ) ) ) ).
```

Running the conversion tool on the problem results in the following embedding:

```
% ----------------------------------------------------------------------
% modal definitions
% ----------------------------------------------------------------------

% declare type for possible worlds
thf( w , type , ( w_type:$tType ) ).

% declare accessibility relations
thf( r , type , (r:w_type>w_type>$o) ).

% define accessibility relation properties
thf( mreflexive_type , type , ( mreflexive : (w_type>w_type>$o)>$o ) ).
thf( mreflexive , definition , ( mreflexive = ( ^ [R:w_type>w_type>$o] :
    ! [A:w_type] : (R@A@A) ) ) ).
thf( meuclidean_type , type , ( meuclidean : (w_type>w_type>$o)>$o ) ).
thf( meuclidean , definition , ( meuclidean = ( ^ [R:w_type>w_type>$o] :
    ! [A:w_type,B:w_type,C:w_type] : ( ( (R@A@B) & (R@A@C) ) =>
    (R@B@C) ) ) ) ).

% assign properties to accessibility relations
thf( r_mreflexive , axiom , ( mreflexive @ r ) ).
thf( r_meuclidean , axiom , ( meuclidean @ r ) ).

% define valid operator
thf( mvalid_type , type , ( mvalid: (w_type>$o)>$o ) ).
thf( mvalid , definition , ( mvalid = ( ^ [S:w_type>$o] : ! [W:w_type] :
    (S@W) ) ) ).

% define nullary, unary and binary operators which are not quantifiers or
% valid operator
thf( mimplies_type , type , ( mimplies:
    (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mimplies , definition , ( mimplies =
    ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] : ( (A@W) => (B@W) ) ) ) ).
thf( mnot_type , type , ( mnot: (w_type>$o)>w_type>$o) ).
thf( mnot , definition , ( mnot =
    ( ^ [A:w_type>$o,W:w_type] : ~(A@W) ) ) ).
thf( mand_type,type , ( mand: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mand , definition , ( mand = ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] :
    ( (A@W) & (B@W) ) ) ) ).
thf( mdia_type , type , ( mdia: (w_type>$o)>w_type>$o) ).
thf( mdia , definition , ( mdia =
    ( ^ [A:w_type>$o,W:w_type] : ? [V:w_type] : ( (r@W@V) & (A@V) ) ) ) ).
thf( mequiv_type , type , ( mequiv: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mequiv , definition , ( mequiv =
    ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] : ( (A@W) <=> (B@W) ) ) ) ).
thf( mbox_type , type , ( mbox: (w_type>$o)>w_type>$o) ).
thf( mbox , definition , ( mbox = ( ^ [A:w_type>$o,W:w_type] :
    ! [V:w_type] : ( (r@W@V) => (A@V) ) ) ) ).
```

```
% define exists quantifiers
thf( mexists_const_type__d_i , type , ( mexists_const__d_i :
    ( ( ( $i ) > ( w_type > $o ) ) > w_type > $o ) ) ).
thf( mexists_const__d_i , definition , ( mexists_const__d_i =
    ( ^ [A:($i)>w_type>$o,W:w_type] : ? [X:($i)] :
    (A @ X @ W) ) ) ).

% define for all quantifiers
thf( mforall_const_type__d_i , type , ( mforall_const__d_i :
    ( ( ( $i ) > ( w_type > $o ) ) > w_type > $o ) ) ).
thf( mforall_const__d_i , definition , ( mforall_const__d_i =
    ( ^ [A:($i)>w_type>$o,W:w_type] : ! [X:($i)] : (A @ X @ W) ) ) ).
thf( mforall_const_type__d_i_t__o_w_type_t__d_o_c_ , type ,
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ :
    ( ( ( $i>(w_type>$o) ) > ( w_type > $o ) ) > w_type > $o ) ) ).
thf( mforall_const__d_i_t__o_w_type_t__d_o_c_ , definition ,
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ =
    ( ^ [A:($i>(w_type>$o))>w_type>$o,W:w_type] : ! [X:($i>(w_type>$o))] :
    (A @ X @ W) ) ) ).

% ------------------------------------------------------------------------
% transformed problem
% ------------------------------------------------------------------------

% positive constant
% maps a proposition to boolean
thf ( positive_type , type , ( positive :
    ( $i > (w_type>$o) ) > (w_type>$o) ) ).

% godlike constant
% maps an individual to boolean
thf ( godlike_type , type , ( godlike : $i > (w_type>$o) ) ).

% essence constant
% maps a proposition and an individual to boolean
thf ( essence_type , type , ( essence :
    ( $i > (w_type>$o) ) > $i > (w_type>$o) ) ).

% necessary existence constant
% maps an individual to boolean
thf ( ne_type , type , ( ne : $i > (w_type>$o) ) ).

% A1: Either the property or its negation are positive, but not both.
thf ( a1 , axiom , ( mvalid @ (
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Phi : $i > (w_type>$o) ] : ( ( mequiv @ ( ( positive @
    ( ^ [ X : $i ] : ( mnot @ ( Phi @ X ) ) ) ) ) @ ( mnot @
    ( positive @ Phi ) ) ) ) ) ) ) ) ).
```

```
% A2: A property necessarily implied by a positive property is positive.
thf ( a2 , axiom , ( mvalid @ (
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Phi : $i > (w_type>$o) ] :
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Psi : $i > (w_type>$o) ] : ( ( mimplies @ ( mand @
    ( positive @ Phi ) @ ( ( mbox @ ( mforall_const__d_i @
    ( ^ [ X : $i ] : ( mimplies @ ( Phi @ X ) @ ( Psi @ X ) ) ) ) ) ) ) @
    ( positive @ Psi ) ) ) ) ) ) ) ) ).

% T1: Positive properties are possibly exemplified.
thf ( t1 , conjecture , ( mvalid @ (
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Phi : $i > (w_type>$o) ] : ( ( mimplies @ ( positive @ Phi ) @
    ( mdia @ ( mexists_const__d_i @ ( ^ [ X : $i ] :
    ( Phi @ X ) ) ) ) ) ) ) ) ) ).

% D1: A God-like being possesses all positive properties.
thf ( d1 , definition , ( godlike = ( ^ [ X : $i ] :
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Phi : $i > (w_type>$o) ] :
    ( mimplies @ ( positive @ Phi ) @ ( Phi @ X ) ) ) ) ) ) ).

% A3: The property of being God-like is positive.
thf ( a3 , axiom , ( mvalid @ ( positive @ godlike ) ) ).

% C: Possibly, God exists.
thf ( c , conjecture , ( mvalid @ ( mdia @ ( mexists_const__d_i @
    ( ^ [ X : $i ] : ( godlike @ X ) ) ) ) ) ).

% A4: Positive properties are necessary positive properties.
thf ( a4 , axiom , ( mvalid @ (
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Phi : $i > (w_type>$o) ] : ( mimplies @ ( positive @ Phi ) @
    ( mbox @ ( positive @ Phi ) ) ) ) ) ) ).

% D2: An essence of an individual is a property possessed by it and
% necessarily implying any of its properties.
thf ( d2 , definition , ( essence =
    ( ^ [ Phi : $i > (w_type>$o) , X : $i ] :( mand @ ( Phi @ X ) @
    ( ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Psi : $i > (w_type>$o) ] : ( ( mimplies @ ( Psi @ X ) @
    ( mbox @ ( mforall_const__d_i @ ( ^ [ Y : $i ] :
    ( mimplies @ ( Phi @ Y ) @ ( Psi @ Y ) ) ) ) ) ) ) ) ) ) ) ) ).

% T2: Being God-like is an essence of any God-like being.
thf ( t2 , conjecture , ( mvalid @ (
    ( mforall_const__d_i @ ( ^ [ X : $i ] : ( mimplies @
    ( godlike @ X ) @ ( essence @ godlike @ X ) ) ) ) ) ) ).
```

```
% D3: Necessary existence of an individual is the necessary exemplification
% of all its essences
thf ( d3 , definition , ( ne = ( ^ [ X : $i ] :
    ( mforall_const__d_i_t__o_w_type_t__d_o_c_ @
    ( ^ [ Phi : $i > (w_type>$o) ] : ( mimplies @
    ( essence @ Phi @ X ) @ ( mbox @ ( mexists_const__d_i @
    ( ^ [ Y : $i ] : ( Phi @ Y ) ) ) ) ) ) ) ) ) ).

% A5: Necessary existence is positive.
thf ( a5 , axiom , ( mvalid @ ( positive @ ne ) ) ).

% T3: Necessarily God exists.
thf ( t3 , conjecture , ( mvalid @ ( mbox @ ( mexists_const__d_i @
    ( ^ [ X : $i ] : ( godlike @ X ) ) ) ) ) ) .
```

Note that this output is actually not valid in THF since it contains multiple conjectures. Removing all conjectures except for one from this literal makes this output valid in THF. In order to proof the intermediate conjectures one has to make axioms from the preceding conjectures and remove at least all following conjectures. The axioms of the embedding are shown to be satisfiable by passing them to the model finder Nitpick [28]. All conjectures are confirmed theorems by Satallax. For the proof of the last theorem T3 all axioms except A5 are removed, theorem T1 is removed and both C and T2 are stated as axioms since the prover was not able to solve the final conjecture T3 without these intermediary results.

7.4 Embedding Performance

A proper performance test could not be conducted since there was no large set of TH0 problems with modal extension available. For all problems from the previous section which are quite small problems the conversion time was less than 0.1 seconds.
The implementation of the transformation process is not efficient at this moment. The reason for this is that the code should be overseeable and all functionality stated explicitly in order to be able to debug the software properly and expand it later to support more modal logical variations and a multimodal setting. Hence there is much room for various improvements of the running time:
A major step would be to shift the identification of nodes which are interesting to the embedding process to the callbacks of the parse tree creation. This includes sorting THF sentences of different roles, collecting the nodes of logical symbols and storing all type nodes. When this is the case it is not necessary anymore to completely traverse the parse tree.
Also the conversion process does a lot of string matching which could be replaced by the way faster matching of integers for fixed expressions e.g. grammar rules and logical connectives. They could be represented as enumerations created by the parser.

Last but not least tweaking the embedding rules may provide performance benefits e.g. by exploring only relevant paths of a subtree when searching for certain elements etc.

8 Conclusion

This thesis introduced the subjects of higher-order logic, higher-order modal logic and their representations in the machine readable syntax TH0.
The existing grammar of TH0 was rewritten to an ANTLR grammar which can be used to automatically create parsers for various programming languages. The parser constructed from this grammar by the software ANTLR was shown to work correctly for most inputs and even all inputs concerning this thesis. Missing features of the parser concern the implementation of numbers, block comments, annotations in annotated formulas, TPI and formula selection in include statements. These can be added subsequently by slightly changing the grammar. The performance of the parser was shown to be very good for all input sizes.
After reading a modal logical formula in TH0 format with the afore mentioned parser it can be converted to TH0 without modal operators. This is achieved by using the embedding approach from *Higher-Order Modal Logics: Automation and Applications* [12] which was implemented by the conversion tool. The resulting TH0 problem can then be given to an ordinary higher-order ATP. Hence the software can be used as a preprocessing tool which enables higher-order ATPs to address higher-order modal problems. It can cope with one modality and all common axiom systems of modal logic in a setting of rigid constants, constant domains and global consequences. Its correctness was demonstrated by translating small problems and Gödel's ontological argument and analyzing the resulting problems using ATP systems.

Further Work

To further develop the software the parser grammar should be extended to have full THF compliance. As for the embedding part more semantical variations like flexible constants, varying domains, local consequences and multimodal settings could be supported. Adding multicore support to both the parser end the embedding would speed up the process significantly for large inputs. More testing on the resulting software artifacts has to be done in order to ensure their correct implementation.

A Definitions, Theorems and Examples

List of Definitions

List of Theorems

List of Examples

B Embedding Definitions in THF

```
% declare world type
thf( w , type , ( w_type:$tType ) ).

% declare relation
thf( r , type , (r:w_type>w_type>$o) ).

% accessibility relation properties definitions
thf( mreflexive_type , type , ( mreflexive : (w_type>w_type>$o)>$o ) ).
thf( mreflexive , definition , ( mreflexive = ( ^ [R:w_type>w_type>$o] :
    ! [A:w_type] : (R@A@A)))).
thf( meuclidean_type , type , ( meuclidean : (w_type>w_type>$o)>$o ) ).
thf( meuclidean , definition , ( meuclidean = ( ^ [R:w_type>w_type>$o] :
    ! [A:w_type,B:w_type,C:w_type] :
    ( ( (R@A@B) & (R@A@C) ) => (R@B@C) ) ) ) ).
thf( msymmetric_type , type , ( msymmetric : (w_type>w_type>$o)>$o ) ).
thf( msymmetric , definition , ( msymmetric = ( ^ [R:w_type>w_type>$o] :
    ! [A:w_type,B:w_type] : ( (R@A@B) => (R@B@A) ) ) ) ).
thf( mserial_type , type , ( mserial : (w_type>w_type>$o)>$o ) ).
thf( mserial , definition , ( mserial = ( ^ [R:w_type>w_type>$o] :
    ! [A:w_type] : ? [B:w_type] : (R@A@B) ) ) ).
thf( mtransitive_type , type , ( mtransitive : (w_type>w_type>$o)>$o ) ).
thf( mtransitive , definition , ( mtransitive = ( ^ [R:w_type>w_type>$o] :
    ! [A:w_type,B:w_type,C:w_type] :
    ( ( (R@A@B) & (R@B@C) ) => (R@A@C) ) ) ) ).
```

```
% assign properties to relations
thf( r_mreflexive , axiom , ( mreflexive @ r ) ).
thf( r_meuclidean , axiom , ( meuclidean @ r ) ).
thf( r_msymmetric , axiom , ( msymmetric @ r ) ).
thf( r_mserial , axiom , ( mserial @ r ) ).
thf( r_mtransitive , axiom , ( mtransitive @ r ) ).

% define valid
thf( mvalid_type , type , ( mvalid: (w_type>$o)>$o ) ).
thf( mvalid , definition , ( mvalid = ( ^ [S:w_type>$o] : ! [W:w_type] :
    (S@W) ) ) ).

% define embedded operators
thf( mtrue_type , type , ( mtrue: w_type>$o ) ).
thf( mtrue , definition , ( mtrue = ( ^ [W:w_type] : $true) ) ).
thf( mfalse_type , type , ( mfalse: w_type>$o ) ).
thf( mfalse , definition , ( mfalse = ( ^ [W:w_type] : $false))).
thf( mnot_type , type , ( mnot: (w_type>$o)>w_type>$o) ).
thf( mnot , definition , ( mnot =
    ( ^ [A:w_type>$o,W:w_type] : ~(A@W) ) ) ).
thf( mnand_type,type , ( mnand: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mnand , definition , ( mn1and =
    ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] : ( (A@W) ~& (B@W) ) ) ) ).
thf( mor_type , type , ( mor: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mor , definition , ( mor = ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] :
    ( (A@W) | (B@W) ) ) ) ).
thf( mnequiv_type , type , ( mnequiv: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mnequiv , definition , ( mnequiv =
    ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] : ( (A@W) <~> (B@W) ) ) ) ).
thf( mnor_type , type , ( mnor: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mnor , definition , ( mnor = ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] :
    ( (A@W) ~| (B@W) ) ) ) ).
thf( mand_type,type , ( mand: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mand , definition , ( mand = ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] :
    ( (A@W) & (B@W) ) ) ) ).
thf( mimplies_type , type , ( mimplies: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mimplies , definition ,
    ( mimplies = ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] :
    ( (A@W) => (B@W) ) ) ) ).
thf( mimpliesreversed_type , type , ( mimpliesreversed:
    (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mimpliesreversed , definition , ( mimpliesreversed =
    ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] : ( (A@W) <= (B@W) ) ) ) ).
thf( mequiv_type , type , ( mequiv: (w_type>$o)>(w_type>$o)>w_type>$o) ).
thf( mequiv , definition , ( mequiv =
    ( ^ [A:w_type>$o,B:w_type>$o,W:w_type] : ( (A@W) <=> (B@W) ) ) ) ).
thf( mbox_type , type , ( mbox: (w_type>$o)>w_type>$o) ).
thf( mbox , definition , ( mbox = ( ^ [A:w_type>$o,W:w_type] :
    ! [V:w_type] : ( (r@W@V) => (A@V) ) ) ) ).
thf( mdia_type , type , ( mdia: (w_type>$o)>w_type>$o) ).
thf( mdia , definition , ( mdia = ( ^ [A:w_type>$o,W:w_type] :
    ? [V:w_type] : ( (r@W@V) & (A@V) ) ) ) ).
```

C List of Figures

D CLI Parameters

Option	Values	Effect	Required
-f	modal | free	specify input logic	yes
-i	existing file or directory	specify input file or directory	yes
-o	file if input is a file, directory if input is a directory	specify output file or directory	yes
-semantics	standard_s5 | all	add semantics to a problem	no
-dotin	file if input is a file, nothing otherwise	creates a Graphviz representation of the input problem	no
-dotout	file if input is a file, nothing otherwise	creates a Graphviz representation of the embedded problem	no
-dotbin	Graphviz binary	creates images from the Graphviz representations specified by -dotin and -dotout	no
-log	file	save log to the specified file	no
-h	n/a	displays help	no

Exemplary usage

Embed one problem

```
java -jar embed-1.0-SNAPSHOT-shaded.jar
    -f modal
    -i my_input_problem.p
    -o my_output_problem.p
```

Embed one problem, create Graphviz output and images from these

```
java -jar embed-1.0-SNAPSHOT-shaded.jar
    -f modal
    -i my_input_problem.p
    -o my_output_problem.p
    -dotin dot_input_problem.dot
    -dotout dot_output_problem.dot
    -dotbin dot
```

Embed one problem which has no semantical definitions for standard S5 semantics (creates one problem)

```
java -jar embed-1.0-SNAPSHOT-shaded.jar
    -f modal
    -i my_input_problem.p
    -o my_output_problem.p
    -semantics standard_s5
```

Embed one problem which has no semantical definitions for all available semantics (creates multiple problems in directory B)

```
java -jar embed-1.0-SNAPSHOT-shaded.jar
    -f modal
    -i my_input_problem.p
    -o B
    -semantics all
```

```
java -jar embed-1.0-SNAPSHOT-shaded.jar
    -f modal
    -i my_input_problem.p
    -o my_output_problem.p
```

Embed the problems of a directory A and its subdirectories and store the embedded problems in a directory B

```
java -jar embed-1.0-SNAPSHOT-shaded.jar
    -f modal
    -i A
    -o B
```

Embed the problems which must not have semantical definitions of a directory A and its subdirectories and store the embedded problems in a directory B Each problem will be created for all semantics available and graphiz representations and images will be created.

```
java -jar embed-1.0-SNAPSHOT-shaded.jar
    -f modal
    -i A
    -o B
    -semantics all
    -dotin
    -dotout
    -dotbin dot
```

E Parser Performance

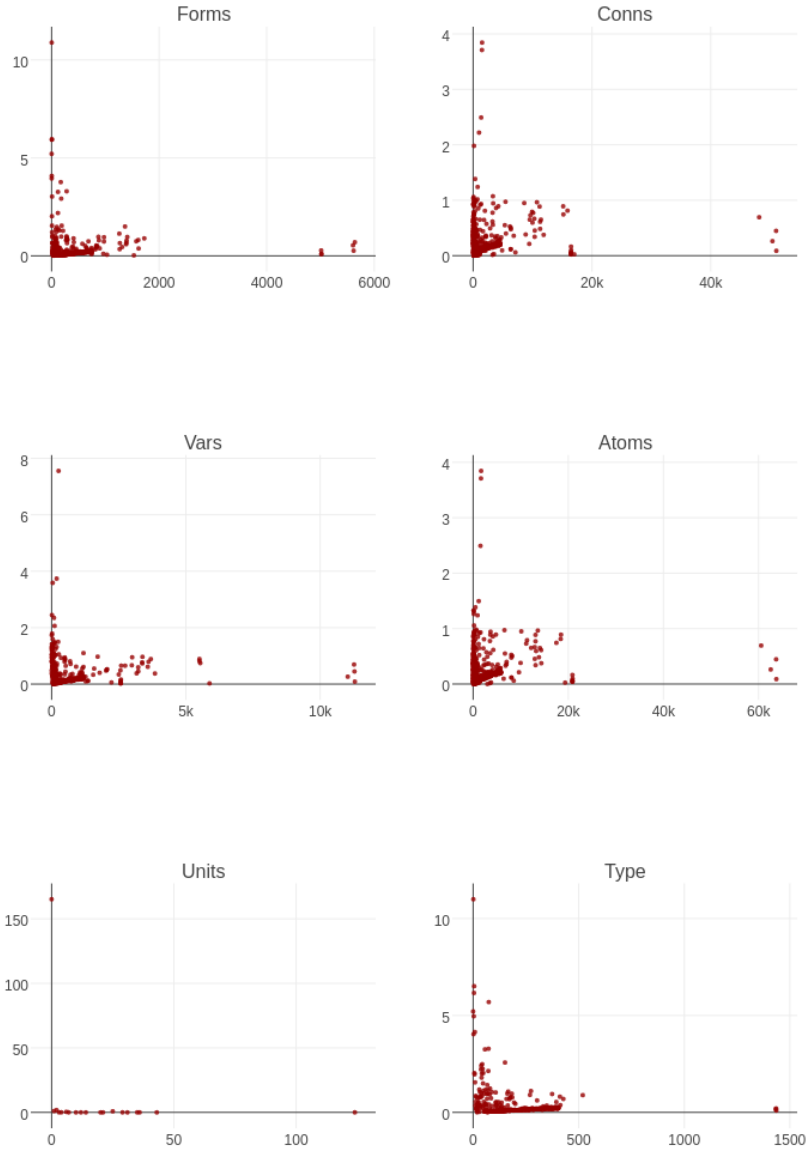
Depending on Category

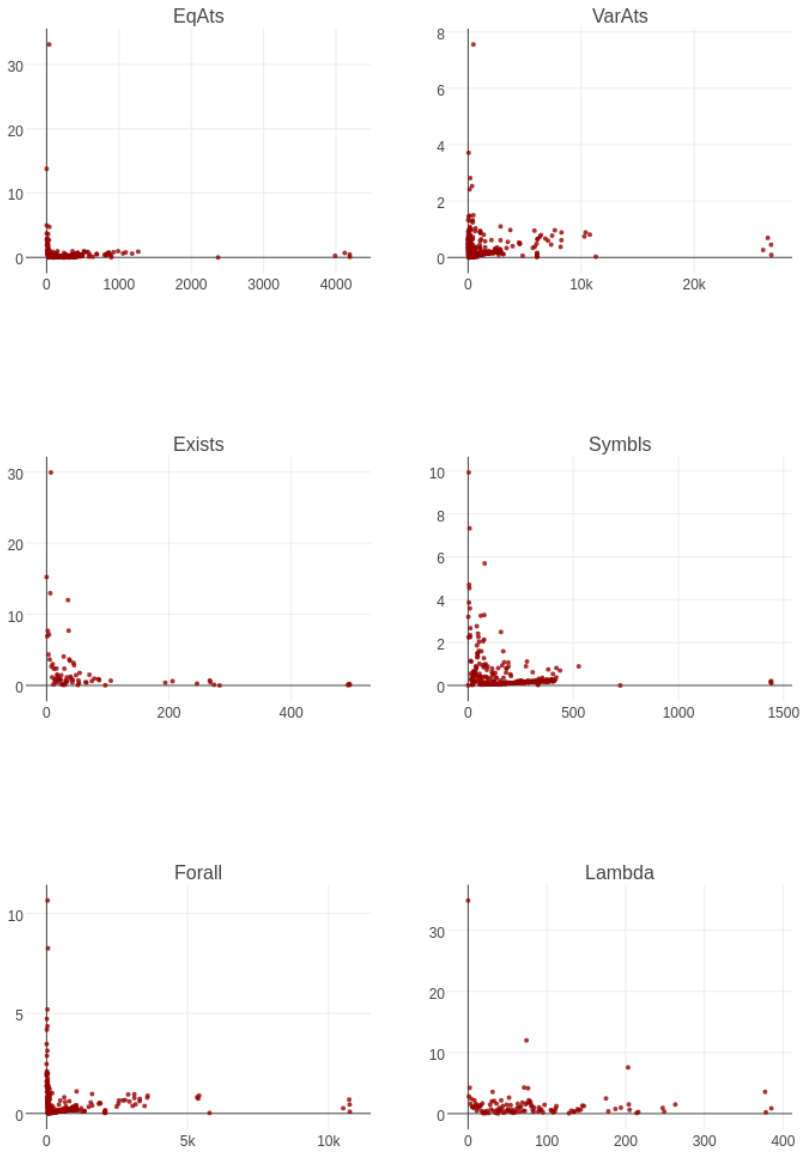
Problem category vs average parse time in seconds

AGT	0.223	NUM	0.038
ALG	0.072	PHI	0.088
CAT	0.011	PLA	0.090
COM	0.007	PUZ	0.034
CSR	0.045	QUA	0.026
DAT	0.041	SCT	0.522
GEG	0.054	SEV	0.041
GRA	0.025	SEU	0.069
GRP	0.007	SET	0.050
KRS	0.077	SWC	0.085
LCL	0.074	SWV	0.169
MSC	0.041	SWW	0.443
NLP	0.478	SYN	0.031
NUN	0.008	SYO	0.015

Depending on Meta Data: Non-Average

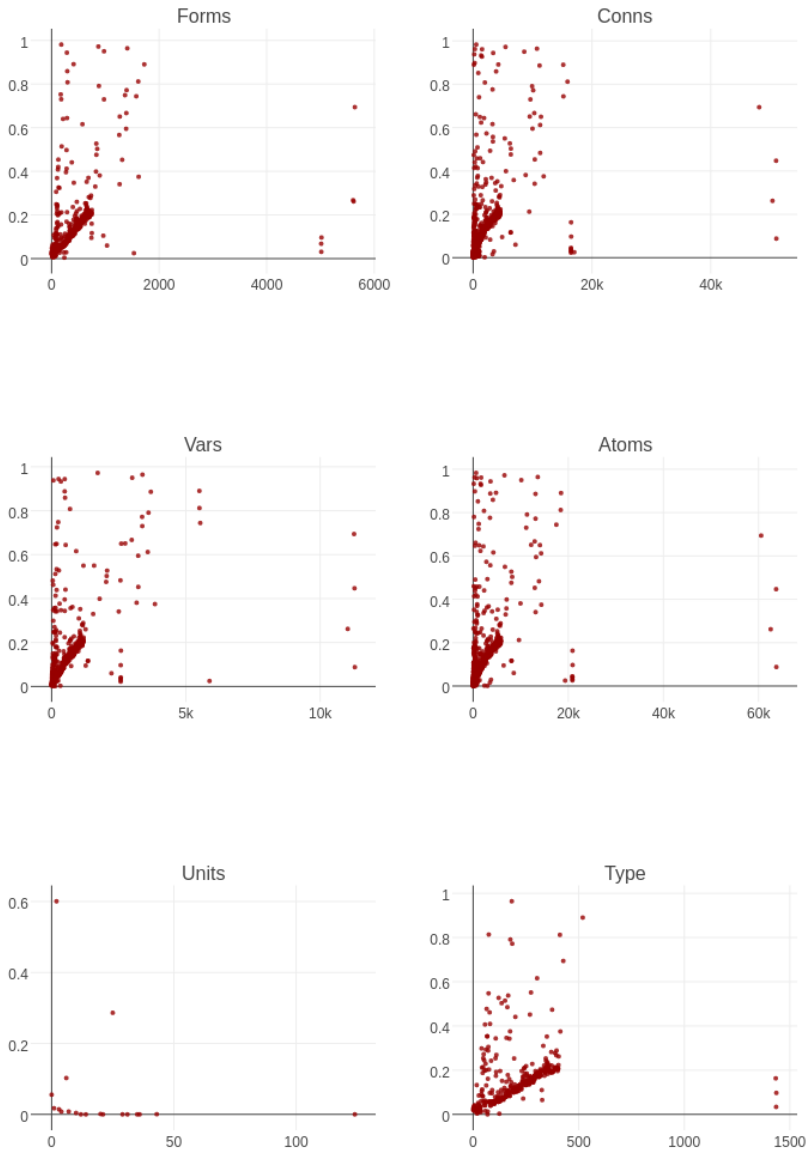
The parsing time for each problem's meta data property value is plotted. Not all meta data is available for all problems hence there might be problems missing in any plot.

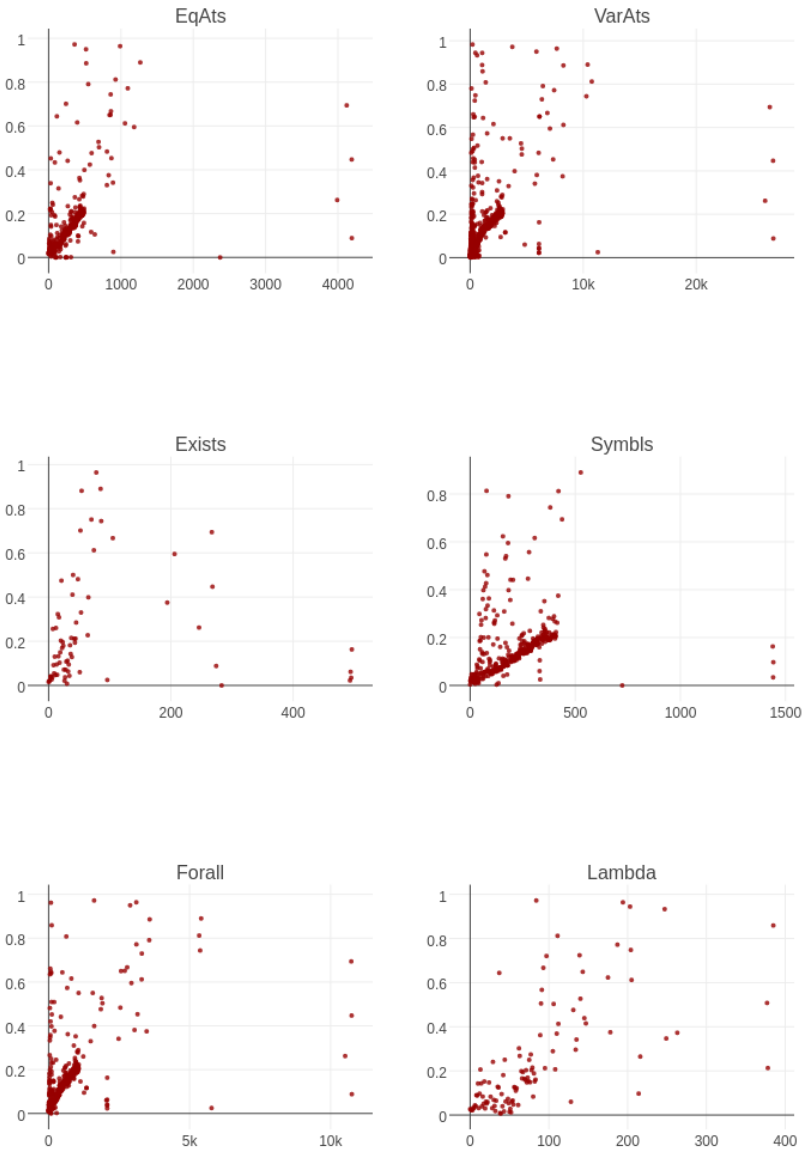



EqAts
0
10
20
30
0
1000
2000
3000
4000
VarAts
0
2
4
6
8
0
10k
20k
Exists
0
10
20
30
0
200
400
Symbls
0
2
4
6
8
10
0
500
1000
1500
Forall
0
5
10
0
5k
10k
Lambda
0
10
20
30
0
100
200
300
400

Depending on Meta Data: Average

The average parsing time is calculated for each meta data property value and plotted. Not all meta data is available for all problems hence there might be problems missing in any plot.




EqAts
VarAts
Exists
Symbls
Forall
Lambda

References

[1] Christoph Benzmüller and Bruno Woltzenlogel Paleo. Automating Gödel's ontological proof of God's existence with higher-order automated theorem provers. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 93 – 98. IOS Press, 2014.

[2] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture, 2015.

[3] Christoph Benzmüller, Lawrence C. Paulson, Nik Sultana, and Frank Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.

[4] Chad E. Brown. *Satallax: An Automatic Higher-Order Prover*, pages 111–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[6] L.C. Paulson T. Nipkow and M. Wenzel. Isabelle/hol: A proof assistant for higher-order logic. In *Number 2283 in LNCS*, 2002.

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

[8] Max Wisniewski, Alexander Steen, and Christoph Benzmüller. Tptp and beyond: Representation of quantified non-classical logics. In Christoph Benzmüller and Jens Otten, editors, *ARQNL 2016. Automated Reasoning in Quantified Non-Classical Logics*, 2016.

[9] Alexander Steen, Max Wisniewski, and Christoph Benzmüller. Tutorial on reasoning in expressive non-classical logics with isabelle/hol.

[10] Wikipedia. Modal logic — Wikipedia, the free encyclopedia, 2016. [Online; accessed 20-August-2016].

[11] Frank Blackburn Patrick Benthem, Johan F.A.K. van Wolter. *Handbook of Modal Logic*. Burlington Elsevier, Burlington, 2007.

[12] Christoph Benzmüller and Bruno Woltzenlogel Paleo. Higher-order modal logics: Automation and applications. In Adrian Paschke and Wolfgang Faber, editors, *Reasoning Web 2015*, number 9203 in LNCS, pages 32–74, Berlin, Germany, 2015. Springer. (Invited paper, mildly reviewed).

[13] Max Wisniewski, Alexander Steen, and Christoph Benzmüller. The Leo-III project. In Alexander Bolotov and Manfred Kerber, editors, *Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014.

[14] A. Church. A formulation of the simple theory of types. In *Journal of Symbolic Logic*, page 5:56–68, 1940.

[15] William M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6(3):267 – 286, 2008.

[16] Leon Henkin. Completeness in the theory of types. *J. Symbolic Logic*, 15(2):81–91, 06 1950.

[17] H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[18] Reinhard Muskens. 10 higher order modal logic. In Johan Van Benthem Patrick Blackburn and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 621 – 653. Elsevier, 2007.

[19] Sally Popkorn. *First steps in modal logic*. Cambridge [u.a.] : Cambridge Univ. Press, Cambridge [u.a.], 1. publ. edition, 1994.

[20] Johan van Benthem. *Modal Logic for Open Minds*. 2010.

[21] Rosalie Iemhoff. Modal logic facts. This text contains some basic facts about modal logic., November 2007.

[22] Raul Hakli Sara Negri. Does the deduction theorem fail for modal logic? 2010.

[23] Wikipedia. Mafia (party game) — Wikipedia, the free encyclopedia, 2016. [Online; accessed 10-August-2016].

[24] Peter Vanderschraaf and Giacomo Sillari. Common knowledge. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

[25] Max Wisniewski, Alexander Steen, and Christoph Benzmüller. Leopard - A generic platform for the implementation of higher-order reasoners. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *LNCS*, pages 325–330. Springer, 2015.

[26] Isabelle overview
https://isabelle.in.tum.de/overview.html.

[27] J.H. Sobel. Appx. b: Notes in dana scott's hand. In *Logic and Theism: Arguments for and Against Beliefs in God*, page 145–146. Cambridge University Press, 2004.

[28] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A ounterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172, pages 131–146, 2010.