Freie Universität Berlin

Automatisierung von freier Logik in Logik höherer Stufe

Automating free logic in higher-order logic

Bachelorarbeit

Irina Makarenko

7. November 2016

Name:	Irina Makarenko
Matrikelnummer:	4675619
E-Mail-Adresse:	irina.makarenko@fu-berlin.de
Fakultät:	Mathematik und Informatik
Institut:	Institut für Informatik
Studiengang:	Bachelor of Science in Informatik
Erstgutachter:	PD Dr. Christoph Benzmüller
Zweitgutachter:	M.Sc. Max Wisniewski

Zusammenfassung

Freie Logik erweitert Logik um den Aspekt der Nichtexistenz von Objekten und schafft Raum für Undefiniertheit. Logische Schlussfolgerungen über solche nicht-klassischen Logiken können über unkonventionelle Einbettungen in die klassische Logik höherer Stufe gezogen werden. Automatische Beweiser für höherstufige Logik setzen auf die TPTP-Sprache THF, eine standardisierte Kodierung für Formeln der Logik höherer Stufe. In dieser Arbeit wurde eine TPTP-konforme Kodierung für die nicht-klassische freie Logik entworfen sowie die Einbettung von freier Logik in Logik höherer Stufe diskutiert und eine automatisierte Übersetzung von freier Logik in die Logik höherer Stufe implementiert. Ziel der Arbeit war es, die native Formulierung von Formeln der freien Logik zu ermöglichen, für deren Auswertung aber trotzdem auf die Fertigkeiten von namhaften höherstufigen Theorembeweisern wie Leo-II(I) ausgewichen wird.

Die Effektivität der Übersetzung wurde anhand beispielhafter Formalisierungen untersucht. Ansatzpunkt dafür bildete das kategorientheoretische Buch *Categories, Allegories* von Freyd und Scedrov (1990).

Abstract

Free logic extends logic by including the aspect of non-existence of objects and creates space for undefinedness. Reasoning about such non-classical logics can be done using unconventional embeddings in classical higher-order logic. Automated theorem provers for higher-order logic rely on the TPTP language THF, a standardized encoding for formulae of higher-order logic. In this thesis, a TPTP compliant encoding will be formulated for non-classical free logic, the embedding of free logic in higher-order logic will be discussed and an automated translation of free logic into higher-order logic will be implemented. The goal of this thesis was to enable native formulation of free logic formulae – for their evaluation, however, the capabilities of notable higher-order theorem provers such as Leo-II(I) were used.

The effectiveness of the translation was investigated based on exemplary formalizations. The starting point was the category theory book *Categories, Allegories* by Freyd und Scedrov (1990).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Berlin, 7. November 2016 . . . . . .

Irina Makarenko

Inhaltsverzeichnis

1 Einleitung

Gibt es Dinge, die es nicht gibt? Fragt man Willard Van Quine, dann ist das keineswegs der Fall. Es gibt nichts, das es nicht gibt. Es gibt alles. Klassische Logik gibt Undefiniertheit keinen Namen, schließt diese sogar kategorisch aus. Da Undefiniertheit und Nichtexistenz aber sehr wohl übergeordnete Rollen in diversen Theorien spielen (vgl. Kapitel 1.1) reichen klassische automatische Theorembeweiser (*automated theorem prover*, ATP), Werkzeuge für die automatische Herleitung und Verifikation von logischen Formeln, nicht aus, um eben jene zu verifizieren. In dieser Arbeit soll sich der automatischen Deduktion von freier Logik auf Basis von höherstufigen Theorembeweisern angenähert werden.

1.1 Anwendungsgebiete der freien Logik

Freie Logik findet Anwendung in den Theorien der definiten Kennzeichnungen, in Sprachen, die partielle Definiertheit und nicht-strikte Funktionen erlauben, in fiktionaler Logik, in den Theorien der Prädikation, in Programmiersprachen als Abbildung von Fehlerzuständen und in der Mengenlehre (vgl. Lambert 1991; Lambert 2001; Nolt 2002). Da sich die freie Logik in der Vergangenheit bereits als nützliches Instrument zur Überprüfung der Konsistenz mathematischer Theorien erwiesen hat (vgl. Benzmüller und Scott 2016b) sei an dieser Stelle insbesondere ihre Anwendung in der Kategorientheorie und in der projektiven Geometrie hervorgehoben.

1.2 Motivation

Aufgrund der vielfältigen Einsatzmöglichkeiten ist eine native Unterstützung der freien Logik durch automatische Theorembeweiser, die in naher Vergangenheit nicht zuletzt durch ontologische Gottesbeweise große Erfolge verbuchen konnten (vgl. Benzmüller und Woltzenlogel Paleo 2015a; Benzmüller und Woltzenlogel Paleo 2016), erstrebenswert. Formeln der freien Logik sollen intuitiv, aber standardisiert eingegeben werden können und so auf ihre Konsistenz überprüft werden können. Statt auf einen eigenständigen Theorembeweiser für die freie Logik zu setzen, wird eine Einbettung nach Benzmüller (2013) vorgezogen. Mit einer solchen Einbettung kann man auf die fortgeschrittenen Fertigkeiten von klassischen höherstufigen Theorembeweisern ausweichen, ohne auf in der freien Logik natürliche Sprachkonstrukte verzichten zu müssen. Eine auf einer Übersetzung beruhende Schnittstelle mit standardisierter Ein- und Ausgabe hat den Vorteil, dass die dem der Übersetzung anschließenden Beweisschritt zu Grunde liegende Implementierung unabhängig von der Übersetzung selbst ist. Im Folgenden soll deshalb eine automatisierte Übersetzung von freier Logik in die Logik höherer Stufe realisiert werden. Dazu wird neben den theoretischen Grundlagen die Einbettung von freier Logik in Logik höherer Stufe im Detail erörtert sowie eine Umsetzung implementiert und letztendlich auch angewendet.

2 Logik höherer Stufe

Typentheorien konkretisieren die Ansätze von Logiken höherer Stufe (***higher-order logic***, HOL). Im Folgenden soll die auf Church (1940) zurückgehende einfache Typentheorie (***simple type theory***, STT), basierend auf dem einfach typisierten λ-Kalkül, aufgegriffen werden.

2.1 Syntax

Die Syntax der einfachen Typentheorie orientiert sich am λ-Kalkül und definiert sich über einfach typisierte Terme, sogenannte Ausdrücke. Typen wiederum definieren sich über eine Menge von Basistypen und dem Typkonstruktor $\to$. Zu den Basistypen zählen o für den Propositionstyp und ι als Typ für die Domäne der Individuen. Der Vollständigkeit halber sei erwähnt, dass es in der Logik höherer Stufe keinerlei Einschränkungen im Hinblick auf die Definition von weiteren Basistypen gibt (Benzmüller und Miller 2014), in dieser Arbeit aber, aus Gründen der Übersichtlichkeit, Churchs Beispiel gefolgt und sich auf die zuvor genannten beschränkt wird. Die Erweiterung der folgenden Definitionen auf eine mächtigere Menge von Basistypen ist trivial.

Definition 1. Die Menge der einfachen Typausdrücke $\mathcal{T}$ besteht aus den Basistypen o und ι und weiterhin aus Funktionstypen mit einem Domänentyp α und einem Kodomänentyp β:

$$\alpha, \beta \coloneqq \quad \iota \mid o \mid (\alpha \to \beta)\,.$$

Der Operator $\to$ ist rechtsassoziativ. Das heißt, werden Klammern ausgespart, dann ist, mit $\alpha_1, \alpha_2, \alpha_3 \in \mathcal{T}$, $(\alpha_1 \to (\alpha_2 \to \alpha_3))$ äquivalent zu $\alpha_1 \to \alpha_2 \to \alpha_3$.

Definition 2. Terme der Logik höherer Stufe sind durch folgende Grammatik gegeben:

$$\begin{aligned} s, t \coloneqq \quad & c_\alpha \mid X_\alpha \mid (s_{\alpha\to\beta}\, t_\alpha)_\beta \mid (\lambda X_\alpha.\, s_\beta)_{\alpha\to\beta} \mid ((=_{\alpha\to\alpha\to o}\ s_\alpha)\, t_\alpha)_o \mid (\neg_{o\to o}\, s_o)_o \mid \\ & ((\vee_{o\to o\to o} s_o)\, t_o)_o \mid (\forall_{(\alpha\to o)\to o}(\lambda X_\alpha.\, s_o))_o \mid (\iota_{(\alpha\to o)\to\alpha}(\lambda X_\alpha.\, s_o))_\alpha \end{aligned}$$

mit $\alpha, \beta \in \mathcal{T}$.

Konventionell sind Konstanten mit Klein- und Variablen mit Großbuchstaben bezeichnet. Der Typ eines jeden Terms wird als Subskript angegeben und ausgelassen, wenn er als irrelevant oder, zum Beispiel durch eine Bindung, als offensichtlich angesehen wird. Σ_α ist eine Menge von Konstanten des Typs α, V_α gibt die unendliche Menge aller Variablen des Typs α an. c_α ist eine syntaktische Variable, die über Σ_α iteriert, während $X_\alpha \in V_\alpha$. Terme des Typs o werden Formeln genannt. Die logischen Konstanten $\bot$ für Falsum und $\top$ für Verum sind wie folgt definiert:

$$\begin{aligned} \bot &\equiv \forall(\lambda X_o.\, X) \\ \top &\equiv \lambda X_\alpha.\, X = \lambda X_\alpha.\, X \end{aligned}$$

mit $\alpha \in \mathcal{T}$.

Für jeden binären Operator *op* kann statt auf seine Präfix-Notation $(op\ s_\alpha)\ t_\alpha$ auch auf die Infix-Notation $s_\alpha\ op\ t_\alpha$ ausgewichen werden. Durch die Negation, Disjunktion und Allquantifizierung lassen sich ergänzend weitere logische Verknüpfungen beschreiben:

$$\begin{aligned} \wedge_{o\to o\to o} &:= \lambda s_o.\, \lambda t_o.\, \neg(\neg s \vee \neg t) \\ \exists_{(\alpha\to o)\to o} &:= \lambda s_{\alpha\to o}.\, \neg\forall(\lambda X_\alpha.\, \neg s X) \\ &\ldots \end{aligned}$$

mit $\alpha \in \mathcal{T}$.

Die Kennzeichnung $\iota(\lambda X.s)$ ist optional und gibt dasjenige eindeutige X wieder, für das s gilt. Für die Kennzeichnung sowie für $\forall(\lambda X.s)$ kann auf die kompaktere Bindernotation zurückgegriffen werden: $\{\forall, \iota\}(X.s)$. Im Kontext der Arbeit wird Gleichheit als primitiv angenommen, kann aber auch alternativ durch die Leibniz-Gleichheit intendiert werden (vgl. Benzmüller, Brown und Kohlhase 2004):

$$=^L_{\alpha\to\alpha\to o} := \lambda X_\alpha.\, \lambda Y_\alpha.\, \forall P_{\alpha\to o}.\, PX \to PY$$

mit $\alpha \in \mathcal{T}$.

Ein if-then-else-Operator definiert sich mittels Kennzeichnung nach Backes (2010) wie folgt:

$$ite_{o\to\alpha\to\alpha\to\alpha} := \lambda s_o.\, \lambda X_\alpha.\, \lambda Y_\alpha.\, \iota(\lambda Z_\alpha.\ (s \wedge X = Z) \vee (\neg s \wedge Y = Z))$$

mit $\alpha \in \mathcal{T}$.

Definition 3. Eine Variable X_α ist gebunden in einem Term s_α mit $\alpha \in \mathcal{T}$, wenn sie mindestens einmal gebunden in s_α vorkommt. X_α ist frei in s_α, wenn die Variable nicht gebunden ist. Ein Term ohne freie Variablen wird geschlossen genannt.

Definition 4. Die Substitution einer Variablen X_α durch einen Term t_α in einem Term s_α, wobei $\alpha \in \mathcal{T}$, wird mit $s\,[t/X]$ angegeben.

Bei einer Substitution müssen gebundene Variablen gegebenenfalls umbenannt werden, damit niemals eine freie Variable in einem Substitut nach einer Ersetzung gebunden wird. Man spricht dann von α-Konversion. α-, β- und η-Konversion sind wie gewohnt definiert:

$$\lambda X.\, s \;=\; \lambda Y.\, (s\,[Y/X]) \quad \text{mit } Y \text{ nicht-frei in } s \qquad (\alpha)$$
$$(\lambda X.\, s)\, t \;=\; s\,[t/X] \qquad (\beta)$$
$$\lambda X.\, (s\, X) \;=\; s \quad \text{mit } X \text{ nicht-frei in } s. \qquad (\eta)$$

2.2 Semantik

Die Semantik von Logik höherer Stufe ordnet deren syntaktischen Strukturen eine formale Bedeutung zu. Dazu werden aussagenlogischen Ausdrücken Wahrheitswerte zugewiesen, indem diese in Modellen interpretiert werden. Der Modellbegriff wird zunächst anhand der Definition eines Rahmens eingeführt und mit dem Begriff der Variablenzuweisung ergänzt, um anschließend die Auswertungsfunktion für den Wert eines Terms zu erläutern. Sofern nicht anders angegeben sind $\alpha, \beta \in \mathcal{T}$.

Definition 5. Ein Rahmen D ist eine Menge $\{D_\tau\}$ bestehend aus nicht-leeren Mengen D_τ mit $\tau \in \mathcal{T}$, sodass D_ι frei wählbar, $D_o = \{wahr, falsch\}$ und $D_{\alpha\to\beta}$ Mengen von Funktionen sind, die D_α auf D_β abbilden.

Definition 6. Ein Modell ist ein Tupel $M = \langle\, D, I \,\rangle$ mit einem Rahmen D und einer Menge von Interpretationsfunktionen $I = \{I_\alpha\}_{\alpha\in\mathcal{T}}$ mit I_α als Abbildung, die jeder Konstanten p_α ein Objekt aus D_α zuordnet.

Definition 7. Eine Funktion $g_\alpha \colon V_\alpha \to D_\alpha$ ist eine Variablenzuweisung, die Variablen des Typs α auf Objekte in D_α abbildet. Die Variablenzuweisung g entspricht der Menge $\{g_\alpha\}_{\alpha\in\mathcal{T}}$. $g[d/X_\alpha]$ mit $d \in D_\alpha$ ist bis auf die auf d abgebildete Variable X_α identisch zu g:

$$g[d/X_\alpha](X_\alpha) = d \text{ und } g[d/X_\alpha](Y_\alpha) = g(Y_\alpha) \in D_\alpha \text{ für alle } Y_\alpha \neq X_\alpha.$$

Definition 8. In einem Standardmodell ist eine jede Domäne $D_{\alpha\to\beta}$ definiert als die Menge $\{\, f \mid f\colon D_\alpha \to D_\beta\}$. In einem Henkin-Modell beschränkt sich diese Domäne auf eine endliche Teilmenge der möglicherweise unendlichen Gesamtmenge: $D_{\alpha\to\beta} \subseteq \{\, f \mid f\colon D_\alpha \to D_\beta\}$. Die Mächtigkeit der eingeschränkten Menge ist frei wählbar, solange die Auswertungsfunktion für die Werte von Termen der Logik höherer Stufe total bleibt.

Logik höherer Stufe mit Henkin-Semantik ist, im Gegensatz zur Standardsemantik mit ihren überabzählbaren Modellen, widerspruchsfrei und vollständig (Henkin 1950; Benzmüller, Brown et al. 2004).

Der Wert $||\, s_\alpha \,||^{M,g}$ eines Terms s_α in einem Modell M unter der Variablenzuweisung g ist ein Element $d \in D_\alpha$ und wird folgendermaßen ausgewertet:

$$
\begin{aligned}
&|| \, c_\alpha \, ||^{M,g} && := && I(c_\alpha) \\
&|| \, X_\alpha \, ||^{M,g} && := && g(X_\alpha) \\
&|| \, (s_{\alpha\to\beta} t_\alpha)_\beta \, ||^{M,g} && := && || \, s_{\alpha\to\beta} \, ||^{M,g} \, (|| \, t_\alpha \, ||^{M,g}) \\
&|| \, (\lambda X_\alpha .\, s_\beta)_{\alpha\to\beta} \, ||^{M,g} && := && \text{die Funktion } f \text{ von } D_\alpha \text{ nach } D_\beta, \text{ sodass} \\
& && && f(d) \; = \; || \, s_\beta \, ||^{M,g[d/X_\alpha]} \text{ für alle } d \in D_\alpha \\
&|| \, ((=_{\alpha\to\alpha\to o} \; s_\alpha) \, t_\alpha)_o \, ||^{M,g} && := && wahr \text{ genau dann, wenn } || \, s_\alpha \, ||^{M,g} \; = \; || \, t_\alpha \, ||^{M,g} \\
&|| \, (\neg_{o\to o} \, s_o)_o \, ||^{M,g} && := && wahr \text{ genau dann, wenn } || \, s_o \, ||^{M,g} \; = \; falsch \\
&|| \, ((\vee_{o\to o\to o} s_o) \, t_o)_o \, ||^{M,g} && := && wahr \text{ genau dann, wenn } || \, s_o \, ||^{M,g} \; = \; wahr \text{ oder} \\
& && && || \, t_o \, ||^{M,g} = \; wahr \\
&|| \, (\forall_{(\alpha\to o)\to o} (\lambda X_\alpha .\, s_o))_o \, ||^{M,g} && := && wahr \text{ genau dann, wenn für alle } d \in D_\alpha \text{ gilt:} \\
& && && || \, s_o \, ||^{M,g[d/X_\alpha]} \; = \; wahr \\
&|| \, (\iota_{(\alpha\to o)\to\alpha} (\lambda X_\alpha .\, s_o))_\alpha \, ||^{M,g} && := && d \in D_\alpha, \text{ sodass } || \, s_o \, ||^{M,g[d/X_\alpha]} \; = \; wahr \text{ und} \\
& && && \text{für alle } d' \in D_\alpha \text{ gilt:} \\
& && && \text{wenn } || \, s_o \, ||^{M,g[d'/X_\alpha]} \; = \; wahr, \text{ dann ist } d' \; = \; d
\end{aligned}
$$

mit $\alpha, \beta \in \mathcal{T}$.

Definition 9. Eine Formel s_o gilt in einem Modell M unter der Variablenzuweisung g genau dann, wenn $||s_o||^{M,g} \; = \; wahr$ und man schreibt $M, g \models s_o$. Eine Formel s_o ist gültig in M, symbolisiert durch $M \models s_o$, genau dann, wenn $M, g \models s_o$ für alle Variablenzuweisungen g erfüllt ist. Eine Formel s_o ist genau dann (allgemein-)gültig, geschrieben $\models s_o$, wenn $M \models s_o$ für alle M erfüllt ist.

Definition 10. Sei φ eine Menge von Formeln der Logik höherer Stufe, dann gilt $\models \varphi$ genau dann, wenn $\models s_o$ für alle $s_o \in \varphi$ erfüllt ist.

Definition 11. Sei φ eine Menge von Formeln der Logik höherer Stufe. Eine Formel s_o ist genau dann eine logische Konsequenz von φ, geschrieben $\varphi \models s_o$, wenn für jedes Modell M erfüllt ist, dass, wenn $M \models \varphi$, dann muss auch $M \models s_o$ gelten.

3 Freie Logik

Der Begriff der freien Logik (*free logic*) wurde von Lambert (1960) geprägt, als er damit eine Logik, die frei von jeglichen Existenzannahmen ist, beschrieb. Konkret bedeutet das, dass in der freien Logik die Quantifizierung über eine Domäne D_α in ihrer ursprünglichen Form beibehalten wird, Terme aber Objekte außerhalb der Domäne D_α denotieren können oder gar nicht definiert sein müssen. So lassen sich Aussagen über Objekte wie „Golum“ oder „die größte Primzahl“ treffen. Da klassische, fregeanische Logik annimmt, dass jeder Term ein (existentes) Objekt in D_α bezeichnet, gilt freie Logik gemeinhin als nicht-klassisch (Nolt 2014). Dieser allgemeine Begriff von freier Logik wurde von verschiedenen Autoren interpretiert und formal dargestellt. Um die Syntax und Semantik von freier Logik höherer Stufe herauszuarbeiten, wird in dieser Arbeit primär auf die Definition der freien Logik der ersten Stufe von Scott (1991) bzw. Benzmüller und Scott (2016a) zurückgegriffen, jedoch im späteren Verlauf auch auf differente Ausprägungen und Varianten von freier Logik eingegangen.

Scotts Konzept von freier Logik unterscheidet eine Domäne D_α und eine wohldefinierte Unterdomäne E_α. Die Domäne D_α enthält alle möglicherweise nicht-existenten Objekte, während E_α die Menge der tatsächlich existenten Objekte erfasst. Quantifizierungen beziehen sich per definitionem auf die Domäne E_α. Undefiniertheit wird durch ein eindeutiges Objekt $\star_\alpha \in D_\alpha \notin E_\alpha$, wie in Abbildung 1 illustriert, repräsentiert.

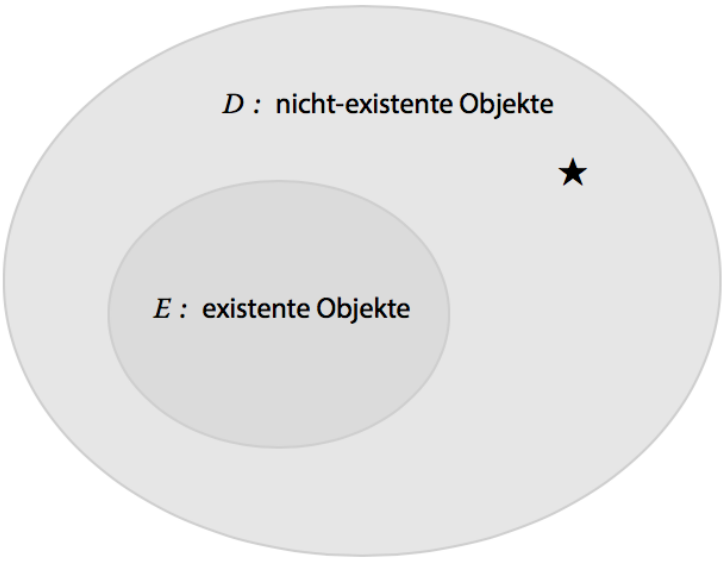

Abb. 1: Grafische Darstellung einer Domäne und ihrer Unterdomäne

3.1 Syntax

Bis auf die Grammatik von Termen, die um zwei Konstanten erweitert wird, entspricht die Syntax von freier Logik der unter Kapitel 2.1 vorgestellten Syntax der Logik höherer Stufe.

Die induktive Definition von Termen lautet nun wie folgt:

Definition 12. Terme der freien Logik sind durch folgende Grammatik gegeben:

$$\begin{aligned} s,t \;:=\; & c_\alpha \mid X_\alpha \mid \star_\gamma \mid (E!_{\alpha\to o}\, s_\alpha)_o \mid (s_{\alpha\to\beta}\, t_\alpha)_\beta \mid (\lambda X_\alpha .\, s_\beta)_{\alpha\to\beta} \mid \\ & ((=_{\alpha\to\alpha\to o}\; s_\alpha)\, t_\alpha)_o \mid (\neg_{o\to o}\, s_o)_o \mid ((\vee_{o\to o\to o} s_o)\, t_o)_o \mid \\ & (\forall_{(\alpha\to o)\to o}(\lambda X_\alpha .\, s_o))_o \mid (\iota_{(\gamma\to o)\to\gamma}(\lambda X_\gamma .\, s_o))_\gamma \end{aligned}$$

mit $\alpha, \beta, \gamma \in \mathcal{T}$ und $\gamma \neq o$.

Wie der initialen Darstellung aus Kapitel 3 zu entnehmen ist, bildet die Konstante $\star$ die Undefiniertheit von Objekten ab. *E!* ist ein Prädikat zur Überprüfung von Existenz, das als primitiv angenommen oder aber auch durch

$$E!\, s \;\equiv\; \exists t.\, t = s$$

mit einem über die Domäne der existenten Objekte quantifizierenden $\exists$ definiert werden kann (vgl. Lambert 2001: 264 ff). Die Kennzeichnung gibt dasjenige eindeutige Objekt zurück, das eine bestimmte Bedingung erfüllt. Gibt es ein solches Objekt nicht, dann ist die Kennzeichnung undefiniert. Die Einschränkung von $\star$ und der Kennzeichnung auf Typen ungleich dem Propositionstyp ist ein Seiteneffekt der auf Undefiniertheit verzichtenden Domäne D_o (siehe dazu auch die Erläuterungen zur Semantik von freier Logik in Kapitel 3.2) und nimmt der Logik nichts von ihrer Expressivität.

3.2 Semantik

Die Semantik der freien Logik unterscheidet sich je nach Art, wie singuläre Terme, die ein nicht-existentes Objekt enthalten, ausgewertet werden. Hierfür gibt es drei Herangehensweisen: die positive, negative oder neutrale Semantik. Bevor auf die einzelnen Semantiken eingegangen werden kann, muss zunächst die Definition eines Modells wie folgt abgewandelt werden:

Definition 13. Ein Rahmen D ist eine Menge $\{D_\tau\}$ bestehend aus nicht-leeren Mengen D_τ mit $\tau \in \mathcal{T}$, sodass D_ι frei wählbar, $D_o = \{wahr, falsch\}$ und $D_{\alpha\to\beta}$ Mengen von Funktionen sind, die D_α auf D_β abbilden. Jeder Menge D_τ mit $\tau \neq o$ wird das undefinierte Objekt $*_\tau$ zugeordnet.

Definition 14. Ein Unterrahmen E ist eine Menge $\{E_\tau\}$ bestehend aus nicht-leeren Mengen E_τ mit $\tau \in \mathcal{T}$, wobei jede Menge E_τ eine Untermenge von D_τ ist. Weiterhin gilt $D_o = E_o$.

Definition 15. Ein Modell ist ein Tripel $M = \langle\, D, E, I \,\rangle$ mit einem Rahmen D, einem Unterrahmen E und einer Menge von Interpretationsfunktionen $I = \{I_\alpha\}_{\alpha\in\mathcal{T}}$ mit I_α als Abbildung, die jeder Konstanten p_α ein Objekt aus D_α zuordnet.

Die übrigen Definitionen der Semantik der Logik höherer Stufe (vgl. Kapitel 2.2) behalten

ihre Gültigkeit.

3.2.1 Positive Semantik

In der positiven Semantik, von welcher Scott in seiner Schrift von 1991 ausgeht, können singuläre Terme des Typs o, die ein nicht-existentes Objekt enthalten, wahr sein, auch wenn diese Objekte nicht im Zusammenhang mit dem Existenzprädikat verwendet werden. Der Wert $|| s_\alpha ||^{M,g}$ eines Terms der positiven freien Logik wird wie folgt ermittelt:

$$
\begin{array}{lcl}
|| c_\alpha ||^{M,g} & := & I(c_\alpha) \\
|| \star_\gamma ||^{M,g} & := & *_\gamma \\
|| X_\alpha ||^{M,g} & := & g(X_\alpha) \\
|| (E!_{\alpha\to o}\, s_\alpha)_o ||^{M,g} & := & wahr \text{ genau dann, wenn } || s_\alpha ||^{M,g} \in E_\alpha \\
|| (s_{\alpha\to\beta}\, t_\alpha)_\beta ||^{M,g} & := & || s_{\alpha\to\beta} ||^{M,g} (|| t_\alpha ||^{M,g}) \\
|| (\lambda X_\alpha .\, s_\beta)_{\alpha\to\beta} ||^{M,g} & := & \text{die Funktion } f \text{ von } D_\alpha \text{ nach } D_\beta, \text{ sodass} \\
 & & f(d) = || s_\beta ||^{M,g[d/X_\alpha]} \text{ für alle } d \in D_\alpha \text{ [1]} \\
|| ((=_{\alpha\to\alpha\to o}\ s_\alpha)\, t_\alpha)_o ||^{M,g} & := & wahr \text{ genau dann, wenn } || s_\alpha ||^{M,g} = || t_\alpha ||^{M,g} \\
|| (\neg_{o\to o}\, s_o)_o ||^{M,g} & := & wahr \text{ genau dann, wenn } || s_o ||^{M,g} = falsch \\
|| ((\to_{o\to o\to o}\, s_o)\, t_o)_o ||^{M,g} & := & wahr \text{ genau dann, wenn } || s_o ||^{M,g} = wahr, \\
 & & \text{dann ist auch } || t_o ||^{M,g} = wahr \\
|| (\forall_{(\alpha\to o)\to o}(\lambda X_\alpha .\, s_o))_o ||^{M,g} & := & wahr \text{ genau dann, wenn für alle } e \in E_\alpha \text{ gilt:} \\
 & & || s_o ||^{M,g[e/X_\alpha]} = wahr \\
|| (\iota_{(\gamma\to o)\to\gamma}(\lambda X_\gamma .\, s_o))_\gamma ||^{M,g} & := & \begin{cases} e \in E_\gamma & \text{wenn } || s_o ||^{M,g[e/X_\gamma]} = wahr \text{ und} \\ & \text{für alle } e' \in E_\gamma \text{ gilt:} \\ & \text{wenn } || s_o ||^{M,g[e'/X_\gamma]} = wahr, \\ & \text{dann ist } e' = e \\ *_\gamma & \text{sonst} \end{cases}
\end{array}
$$

mit $\alpha, \beta, \gamma \in \mathcal{T}$ und $\gamma \neq o$.

Es sei angemerkt, dass die Auswertungsfunktion durch die Einführung eines fixen undefinierten Objekts, dargestellt durch $*$, total bleibt (Nolt 2002). Die Funktion um die Evaluierung von $|| (\iota_{(o\to o)\to o}(\lambda X_o .\, s_o))_o ||^{M,g}$ zu erweitern ist trivial, allerdings im Rahmen dieser Arbeit nicht von Belang.

[1] Mögliche bzw. erzwungene Interaktionen von Undefiniertheit über die Typhierarchien hinweg sind noch zu untersuchen.

3.2.2 Negative Semantik

In der negativen freien Logik (vgl. Burge 1974) werden alle singulären Formeln mit nichtexistenten Objekten unabhängig von der Verwendung eines Existenzprädikats bedingungslos zu falsch ausgewertet. Hierzu zählt insbesondere auch die Identität. Die Auswertungsfunktion aus Kapitel 3.2.1 ändert sich an folgenden zentralen Stellen, damit der Wert $|| s_\alpha ||^{M,g}$ eines Terms der negativen freien Logik ermittelt werden kann:

$$|| (s_{\alpha\to\gamma}\, t_\alpha)_\gamma ||^{M,g} \quad := \quad \begin{cases} || s_{\alpha\to\gamma} ||^{M,g} (|| t_\alpha ||^{M,g}) & \text{wenn } || s_{\alpha\to\gamma} ||^{M,g}, \\ & || t_\alpha ||^{M,g} \in E_\alpha \\ *_\gamma & \text{sonst} \end{cases}$$

$$|| (s_{\alpha\to o}\, t_\alpha)_o ||^{M,g} \quad := \quad \begin{cases} || s_{\alpha\to o} ||^{M,g} (|| t_\alpha ||^{M,g}) & \text{wenn } || s_{\alpha\to\gamma} ||^{M,g}, \\ & || t_\alpha ||^{M,g} \in E_\alpha \\ falsch & \text{sonst} \end{cases}$$

$$|| ((=_{\alpha\to\alpha\to o}\ s_\alpha)\, t_\alpha)_o ||^{M,g} \ := \ wahr \text{ genau dann, wenn } || s_\alpha ||^{M,g} = || t_\alpha ||^{M,g} \text{ und wenn } || s_\alpha ||^{M,g}, || t_\alpha ||^{M,g} \in E_\alpha$$

mit $\alpha \in \mathcal{T}$ und $\gamma \neq o$.

Die hier vorgestellte Semantik der negativen freien Logik beruht auf einer strikten Variante der Applikation. Alternativ kann sich Striktheit auch nur auf die Argumente von Applikationen beziehen.

Während Formeln wie $((\lambda X_\alpha.\, (X_\alpha = X_\alpha)_o)_{\alpha\to o} \star_\alpha)_o$ bzw. $Einhorn = Einhorn$ in positiver freier Logik jeweils zu $wahr$ ausgewertet werden können, sind diese in freier Logik mit negativer Semantik unabdingbar falsch und damit unerfüllbar. In der Beispieldomäne

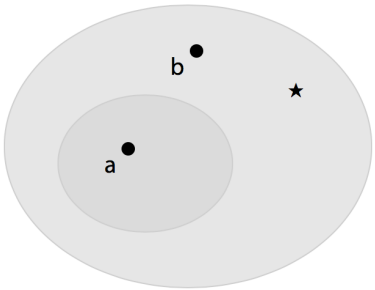

Abb. 2: Beispieldomäne des Typs ι mit drei Objekten

aus Abbildung 2 würde die Identitätsfunktion $f\colon X_\iota \mapsto X_\iota$ das Objekt a auf sich selbst und durch die Neudefinition der Gleichheit die Objekte b und $\star$ auf $*_\iota$ abbilden. Auch $|| (\lambda X.\, f(X))\, b ||^{M,g} = || \ f(b) ||^{M,g}$ wird dann zu $falsch$ ausgewertet.

3.2.3 Neutrale Semantik

In der neutralen Semantik erhalten alle singulären Formeln, die nicht der Form $E!\ s_\alpha$ entsprechen, einen unbestimmten Wahrheitswert, sie werden wahrheitswertfrei. Hier unterscheidet man zwischen zwei Typen: die gewöhnliche neutrale Semantik, bei der die Wahrheitswerte von Termen direkt auf der Basis von den wertfreien Termen ausgewertet werden und die Supervaluationen, bei denen die Werte auf Basis der übrigen Terme berechnet werden, deren Wahrheitswerte auf $wahr$ oder $falsch$ gesetzt werden, indem man temporär davon ausgeht, dass die Objekte existent statt nicht-existent sind und damit auch nicht wahrheitswertfrei sein können.

Eine neutrale Semantik ist insofern einfach gehalten, als dass singuläre Terme eindeutig wertfrei oder nicht wertfrei sind. Je komplexer die Terme werden, desto schwieriger wird auch die Auswertung. Bei einem beliebigen wahrheitswertfreien Term ist dessen Negation ebenfalls ohne Wahrheitswert, aber wie steht es um Implikationen? Ist eine Implikation $s_o \rightarrow t_o$ mit s_o wahr und t_o wahrheitswertfrei falsch oder ebenfalls wahrheitswertfrei? Auch unabhängig von solchen Entscheidungen sind viele Formeln, die in der klassischen Logik und auch in freier Logik gültig sind, dies nicht mehr in der freien Logik mit neutraler Semantik. Die Formel $\neg(s_o \wedge \neg s_o)$ ist, wenn s_o wahrheitswertfrei ist, ebenso wahrheitswertfrei und somit nicht valide. Eine Logik mit vorwiegend wahrheitswertfreien Termen würde in einer sehr schwachen Logik resultieren oder den Begriff einer schwachen Gültigkeit vorraussetzen (Lehmann 2001).

Die Supervaluationssemantik wurde erstmals von van Fraassen (1966) vorgestellt und später von Bencivenga (1986) weiter verfolgt. Bencivenga versucht in der von ihm vorgestellten Semantik die primären Eigenschaften von Objekten unabhängig von deren Existenz einzubeziehen und so die wesentlichen Probleme der einfachen neutralen freien Logik zu überwinden. Für weiterführende Erläuterungen und eine formale Ausarbeitung einer neutralen Semantik mit Supervaluationen für die freie Logik erster Stufe sei auf die genannten Quellen verwiesen.

3.3 Variationen

In der Literatur sind von Russell[2] bis heute diverse Abwandlungen der freien Logik – Morscher und Simons (2001) sprechen sogar von einer ganzen Logikfamilie – zu finden, von denen die wichtigsten kurz benannt und zusammengefasst werden sollen.

3.3.1 Meinongianische Logik

„Wer paradoxe Ausdrucksweisen liebt, könnte […] ganz wohl sagen: es gibt Gegenstände, von denen gilt, daß es dergleichen Gegenstände nicht gibt“ (Meinong 1904: 9). Die klassische, fregeanische Logik postuliert, dass es nichts gibt, was es nicht gibt. Alexius Meinong ist einer der bekanntesten Vertreter für eine gegensätzliche Meinung, nämlich die, dass sehr wohl Dinge existieren, die nicht existieren, und Namensgeber für die damit assoziierte meinongianische Logik. Sie teilt mit der freien Logik die Motivation, den Begriff der Nichtexistenz einfangen zu wollen. In der eigentlichen freien Logik müssen Objekte schlicht nicht denotieren, ein

[2] Bertrand Russell (1920: 167 ff.) schlug in seinen Ausführungen zu *Introduction to Mathematical Philosophy* einen ersten Ansatz in Richtung inklusiver Logik vor.

Umstand, der mit einer partiellen Auswertungsfunktion einhergeht. Nichtexistenz kann aber auch durch ein „Etwas“ außerhalb der Domäne ausgedrückt werden, ein „Etwas“, welches nicht über die (klassischen) Quantoren erreicht werden kann. Eine solche Auswertungsfunktion wäre total, greift allerdings über die Grenzen der Domäne der existenten Objekte hinaus (Schweizer 2015). Dies gibt genau den Ansatz von Meinongs (1904) Schrift über Existenz und Subsistenz wieder. Meinongs Idee war für viele Freilogiker Inspiration für ein zweiteiliges Domänenkonzept: eine innere Domäne für die existenten (E_α) und eine zweite, äußere Domäne, die die existenten und nicht-existenten Objekte erfasst (D_α), ergänzt durch ein weiteres Paar von Quantoren, die zusätzlich, neben den klassischen Quantoren, über die existenten und nicht-existenten Objekte iterieren. Einige Autoren gehen sogar so weit, dass die innere Domäne E_α keine Teilmenge von D_α sein muss, sondern dass es sich bei den beiden um zwei disjunkte Mengen handelt (Nolt 2014). Obwohl die Dual-Domänen-Semantik mit ihrer möglichen Quantifizierung über die äußere Domäne kontrovers diskutiert wird und ihr sogar die Eigenschaften einer freien Logik abgesprochen werden (vgl. Paśniczek 2001), wird sie in Scotts Definition und somit auch in dieser Arbeit zur Vereinfachung einer Automatisierung als implizit angenommen.

3.3.2 Inklusive Logik

Klassische Logik verlangt, dass alle Objekte Teil einer quantifizierbaren Domäne D_α sind, ebenso wie die Tatsache, dass diese Domäne D_α nicht-leer ist. Freie Logik, D_α als eine zu einer äußeren (nicht-leeren) Domäne unterschiedlichen inneren Domäne E_α interpretierend, widerspricht der ersten Annahme. Inklusive Logik (oder auch inklusive bzw. universelle freie Logik)[3] geht noch einen Schritt weiter und widerspricht beiden (Quine 1954). Der Modellbegriff wird leicht abgeändert: Statt wie in Kapitel 3.2 den Unterrahmen über nicht-leere Mengen E_τ mit $\tau \in \mathcal{T}$ zu definieren, können diese Untermengen in der inklusiven Logik möglicherweise leer sein. Während jede inklusive Logik im Allgemeinen frei ist, muss nicht jede freie Logik inklusiv sein. Inklusivität nimmt der freien Logik die letzte implizite Existenzannahme: In der klassischen Logik gilt $\forall X.\, s \vDash \exists X.\, s$, in der inklusiven Logik ist man frei von einer solchen Inferenz.

Existenzquantifizierte Formeln wie

$$\exists X_\alpha.\, X = X$$

$$\exists X_\alpha.\, sX \to sX$$

sind wahr in einer nicht-leeren Domäne und somit auch wahr in klassischer und freier Logik. In der inklusiven Logik sind diese aber aufgrund der möglicherweise leeren Domäne E_α nicht allgemeingültig. Im Gegenzug sind Allquantifizierungen in der leeren Domäne uneingeschränkt gültig, sodass in der inklusiven Logik auch kontroverse Formeln – beispielsweise $\forall X.\, X \wedge \neg X$ – wahr sein können. Zudem werden in der klassischen und freien Logik allgemeingültige Formeln wie $(\forall X.\, s) \to s$ mit X nicht-frei in s ungültig in der inklusiven Logik,

[3] Inklusive Logik existiert auch als eigenständiger, von der freien Logik losgelöster Begriff. In dieser Arbeit wird er synonym zu inklusiver freier Logik verwendet.

womit auch die Möglichkeiten für Quantorenverschiebungen eingeschränkt werden. Die folgende Äquivalenz ist nicht allgemeingültig in der inklusiven Logik:

$$(\forall X_\alpha.\,(s \wedge t)) \;\leftrightarrow\; (s \wedge \forall X_\alpha.\,t) \quad \text{mit } X \text{ nicht-frei in } s$$

Trotzdem ist die inklusive Logik weit verbreitet unter Freilogikern, die diese Eigenschaften für sich nutzen können (Nolt 2002).

3.3.3 Kripke-Semantik

Auch die Mögliche-Welten-Semantik, insbesondere in Kombination mit der Identität über mehrere Welten hinweg, berührt den Existenzbegriff: Objekte existieren nicht nur in einer Welt, sondern in einer Menge von zueinander in Relation stehenden Welten. Um solche Modalitäten abzubilden, konstruiert die Semantik von Kripke (1963) einen Rahmen, der aus einer Menge K von sogenannten Welten mit jeweils variierenden Domänen und einer binären Zugänglichkeitsrelation $R \subseteq K \times K$ besteht, die diese verbindet (vgl. Abbildung 3). Logiken

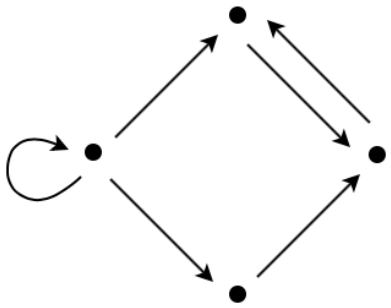

Abb. 3: Grafische Darstellung eines Kripke-Rahmens

mit Kripke-Semantik, insbesondere die Modallogik, tendieren dazu, frei zu sein. Nach Garson (1991) ist der Gebrauch von freier Logik für eine adäquate semantische Behandlung von quantifizierter Modallogik sogar unumgänglich. Das ist der Tatsache geschuldet, dass Objekte in einer oder mehreren Welten existieren können, in anderen Welten aber nicht zwangsläufig auch existieren müssen. Zum Beispiel kann ein Objekt „Johann Wolfgang von Goethe“ in einigen Welten, in der Temporallogik beispielsweise in der Vergangenheit, existieren, in der Gegenwart und in allen zukünftigen Zeitepochen indessen nicht.

4 Einbettung von freier Logik in Logik höherer Stufe

Während in den vorhergehenden Kapiteln Syntax und Semantik der beiden an der Übersetzung beteiligten Logiken definiert wurden, wird nun die semantische Einbettung der freien Logik in die Logik höherer Stufe beschrieben und deren Kodierung betrachtet.

4.1 Semantische Einbettung

Die im Folgenden gezeigte semantische Einbettung orientiert sich formal an der von Benzmüller und Woltzenlogel Paleo (2015b) vorgestellten Einbettung von Modallogik in die Logik höherer Stufe, die sich wiederum auf die Techniken von Gabbay (1996) stützt. Grundlage für die in dieser Arbeit vorgestellte Einbettung bildet der Ansatz von Benzmüller und Scott (2016a), welcher die Einbettung von positiver freier Logik in Isabelle/HOL[4] behandelt.

Freie Logik wird in die Logik höherer Stufe eingebettet, indem ein Prädikat eingeführt wird, das die Existenz von Objekten überprüft und so die Unterdomäne der existenten Objekte einer Domäne D_α abbildet. Der Allquantor und der Existenzquantor iterieren unter Verwendung dieses Existenzprädikats über die Domäne der existenten Objekte, über E_α. Die Kennzeichnung gibt ein Objekt aus E_α oder, falls ein solches Objekt nicht existiert, ein undefiniertes Objekt aus der Domäne $D_\alpha \setminus E_\alpha$ zurück. Somit sind – ausgehend von den Erläuterungen aus den Kapiteln 2 und 3 – die logischen Verknüpfungen der positiven freien Logik wie folgt in die Logik höherer Stufe eingebettet:

$$
\begin{aligned}
\dot{\neg}_{o\to o} &:= \lambda s_o.\, \neg s \\
\dot{\to}_{o\to o\to o} &:= \lambda s_o.\, \lambda t_o.\, s \to t \\
\dot{=}_{\alpha\to\alpha\to o} &:= \lambda X_\alpha.\lambda Y_\alpha.\, X = Y \\
\dot{\forall}_{(\alpha\to o)\to o} &:= \lambda s_{\alpha\to o}.\, \forall(\lambda X_\alpha.\, E!\, X \to sX) \\
\dot{\iota}_{(\alpha\to o)\to\alpha} &:= \lambda s_{\alpha\to o}.\, ite\,(\exists(\lambda X_\alpha.\, E!\, X \wedge sX \wedge \forall(\lambda Y_\alpha.\, (E!\, Y \wedge sX) \to (X = Y)))) \\
&\qquad\qquad (\iota(\lambda X_\alpha.\, E!\, X \wedge\ sX)) \\
&\qquad\qquad \star_\alpha
\end{aligned}
$$

mit *E!* als Existenzprädikat und $\star_\alpha$ als Konstante, die die Undefiniertheit in der Domäne D_α symbolisiert. Die verbleibenden logischen Verknüpfungen können wie üblich abhängig von $\dot{\neg}$, $\dot{\to}$ und $\dot{\forall}$ definiert werden:

[4] Bei Isabelle/HOL handelt es sich um eine interaktive Beweisumgebung für die Logik höherer Stufe (vgl. Nipkow, Paulson und Wenzel 2002).

$$
\begin{aligned}
\dot{\vee}_{o\to o\to o} &:= \lambda s_o.\, \lambda t_o.\, (\dot{\neg} s) \dot{\to} t \\
\dot{\wedge}_{o\to o\to o} &:= \lambda s_o.\, \lambda t_o.\, \dot{\neg}(\dot{\neg} s \,\dot{\vee}\, t) \\
\dot{\leftrightarrow}_{o\to o\to o} &:= \lambda s_o.\lambda t_o.\, (s \dot{\to} t) \,\dot{\wedge}\, (t \dot{\to} s) \\
\dot{\exists}_{(\alpha\to o)\to o} &:= \lambda s_{\alpha\to o}.\, \dot{\neg}\dot{\forall}(\lambda X_\alpha.\, \dot{\neg} s X)
\end{aligned}
$$

Anhand dieser Einbettung können Probleme der freien Logik so erweitert und umschrieben werden, dass sie vollständig in der Syntax der Logik höherer Stufe ausgedrückt werden können. Nachdem ein Problem der freien Logik in ein äquivalentes Problem der Logik höherer Stufe konvertiert wurde, kann dieses an einen höherstufigen Theorembeweiser weitergegeben werden, der das Problem auf seine Konsistenz hin überprüft. Das Ergebnis kann schlussendlich zum ursprünglichen Problem zurückprogagiert werden.

Für die Automatisierung der Übersetzung von freier Logik in Logik höherer Stufe muss die Einbettung zunächst in ein maschinenlesbares Format gebracht werden.

4.2 Kodierung der Einbettung

Um die zuvor gezeigte Einbettung in ein praxisnahes Format zu überführen, wird die Einbettung in TPTP-konforme logische Formeln übersetzt. Dazu soll zunächst initial das TPTP-Projekt und die TPI-Sprache vorgestellt sowie auf das THF-Format, das TPTP-Sprachpendant zur Logik höherer Stufe, und auf eine Adaption dessen für die freie Logik eingegangen werden. Zudem wird eine Ergänzung der TPI-Sprache vorgeschlagen, mit welcher spezifische Parameter für die Einbettung von freier Logik festgesetzt werden können.

4.2.1 TPTP

Die 2009 erstmals veröffentlichte Bibliothek der Tausend Probleme für Theorembeweiser (*Thousands of Problems for Theorem Provers*, TPTP) sieht sich als eine Infrastruktur, aufgebaut, um die Entwicklung, Forschung und Lehre von automatischen Theorembeweisern voranzutreiben (Sutcliffe 2009). Die Infrastruktur umfasst die Probleme selbst, die TPTP-Sprache, die Bibliothek der Tausend Lösungen von Theorembeweisern (*Thousands of Solutions from Theorem Provers*, TSTP) und weitere mit den Bibliotheken verknüpfte Tools sowie den jährlich stattfindenden CADE-Wettbewerb für automatische Theorembeweissysteme (*CADE ATP System Competition*, CASC).[5]

In den Anfängen, um 1993, wurde zunächst nur die konjunktive Normalform (*clause normal form*, CNF) als Untersprache der Logik erster Stufe unterstützt und 1997 mit FOF, einer Form für die (vollständige) Logik erster Stufe, komplettiert. 2008 wurde THF, eine typisierte Form von Logik höherer Stufe, entwickelt, der auch eine typisierte Form von Logik erster Stufe, genannt TFF, folgte. Die THF-Probleme, die in der TPTP enthalten sind, sind größtenteils in TH0 geschrieben, einem minimalen, aber ausreichend ausdrucksstarken Kernteil von THF, der

[5] Alle Komponenten der Infrastruktur sind frei verfügbar und zu finden unter `http://www.tptp.org`.

auf Churchs einfach typisiertes λ-Kalkül zurückzuführen ist. Daneben steht die polymorphe Form TH1 zur Verfügung (vgl. Kaliszyk, Sutcliffe und Rabe 2016).

Die TPTP-Sprachfamilie ist menschenlesbar, leicht parsebar, flexibel und beliebig erweiterbar, wodurch sie als ein effizientes Werkzeug für das Schreiben von Problemen und deren Lösungen gilt. Sie findet breite Anwendung im Bereich des automatischen Theorembeweisens und bildet auch die Grundlage für die Umsetzung der Einbettung, die im Rahmen dieser Arbeit erarbeitet wurde. Im Folgenden wird kurz auf die Sprachkonstrukte TPI und THF eingegangen. Eine vollständige Beschreibung der Syntax kann zum Beispiel in der Übersicht von Sutcliffe und Benzmüller (2010) nachgeschlagen oder unter `http://www.tptp.org` eingesehen werden.

4.2.2 TPI

Die Sprache für TPTP-Prozessanweisungen (*TPTP Process Instruction*, TPI) kodiert Kontrollbefehle an automatische Theorembeweiser, mit welchen logische Formeln aktiv beeinflusst werden können. Eine Eingabe in TPI sieht beispielsweise wie folgt aufgebaut aus:

```
tpi(name, command, command details, [source, [useful-info]]).
```

Eingaben in TPI können direkt an Theorembeweiser weitergegeben oder durch externe Programme gelesen werden, die letztendlich selbst Theorembeweiser aufrufen. Solche Systeme, die TPI interpretieren können, werden TPI-Systeme genannt. Dateien, die an TPI-Systeme weitergegeben werden, können logische Formeln und Befehle gleichermaßen enthalten: Formeln werden ausgewertet, Befehle ausgeführt.

Die aus der Umsetzung der Einbettung resultierende Anwendung interpretiert für die Übersetzung relevante TPI-Befehle. Da ein solcher Befehl in solch einer Form noch nicht existiert, wird dem Beispiel von Wisniewski, Steen und Benzmüller (2016) folgend eine Erweiterung der TPI-Sprache um nachfolgendes Konstrukt vorgeschlagen:

```
tpi(1, set_logic, free('$E' = '$empty', '$choice' = '$yes', '$ite' = '$yes')).
```

respektive

```
tpi(1, set_logic, free('$E' = '$non_empty', '$choice' = '$no', '$ite' = '$no')).
```

Durch den Parameter `$E` kann zwischen einer freien Logik (`$E = $non-empty`) oder einer inklusiven freien Logik (`$E = $empty`) gewählt werden. Da die Einbettung die Kennzeichnung berücksichtigt und für diese Opertoreneinbettung spezielle THF-Sprachkonstrukte notwendig sind (vgl. dazu Kapitel 4.2.3), die nicht von jedem Theorembeweiser unterstützt werden, soll festgelegt werden können, ob diese Sprachkonstrukte in der Einbettung Verwendung finden sollen oder ob auf eine alternative Einbettung über Axiomatisierungen zurückgegriffen werden soll. Die betroffenen Sprachkonstrukte – der choice-Operator `@+` und der if-then-else-Operator `$ite` – können über die Parameter `$choice` und `$ite` jeweils mit `$yes` oder `$no` aktiviert bzw. deaktiviert werden. Die Reihenfolge der Schlüsselwörter spielt keine Rolle, jedoch müssen alle drei über den TPI-Befehl festgelegt werden. Die Anwendung wird standardmäßig von der Belegung `$non-empty`/`$yes`/`$yes` ausgehen, sodass der TPI-Befehl auch weggelassen werden kann.

4.2.3 THF und HFF

Bevor die Einbettung im Detail erläutert werden kann, wird zunächst die Syntax von THF und HFF, eine eigens entworfene Anpassung von THF, die den Ansprüchen der freien Logik gerecht wird, aufgezeigt. Die Semantik von THF entspricht der Logik höherer Stufe mit Henkin-Semantik und einem choice-Operator, einer Konstanten, die dasjenige Objekt zurückgibt, welches eine bestimmte Bedingung erfüllt (Sutcliffe und Benzmüller 2010). Das folgende Beispiel für die Struktur einer THF-Formel, das die Vereinigung definiert, ist dem Papier von Sutcliffe und Benzmüller (2010) entnommen:

```
thf(union, definition,
    ( union = ( ^ [X: $i > $o,Y: $i > $o,U: $i] : ( ( X @ U ) | ( Y @ U ) ) ) )
).
```

Der Angabe des Bezeichners und des Formeltyps (im Beispiel: `definition`) folgt die Formel selbst. Neben logischen Verknüpfungen wie `~`, `&`, `|`, `=>` und `<=>` für $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ wird die λ-Abstraktion mit dem Symbol `^` und die Applikation mit `@` angegeben. `!` bezeichnet die Allquantifizierung über bestimmte typisierte Variablen, `?` den Existenzquantor. Für nähere Informationen zu der Verwendung dieser Sprachelemente sei wieder auf die bereits erwähnten Quellen in Bezug auf TPTP verwiesen.

HFF, eine Form für freie Logik höherer Stufe, benannt nach der in Wisniewski et al. (2016) eingeführten Konvention, übernimmt alle Eigenschaften von THF und fügt dessen Sprachspektrum weitere Konstanten ähnlich `$true` und `$false` hinzu, sodass diese nativ und ohne eine vorherige Definition genutzt werden können. Die Definitionen werden bei der Übersetzung von HFF nach THF nachgeliefert. Für die freie Logik werden Konstanten für das Existenzprädikat und $\star_\alpha$ benötigt, sowie ein Satz zusätzlicher Quantoren. Das Existenzprädikat wird durch `$e` repräsentiert und wie folgt angewendet:

```
hff(eq, definition,
    ( eq = ( ^ [X: $i, Y: $i] :
                ( ( $e @ $i @ X ) & ( $e @ $i @ Y ) & ( X = Y ) ) ) )
).
```

`$e` erhält zwei Parameter, von denen der erste der Variablentyp ist, auf den das Prädikat angewendet wird (im Beispiel: `$i`). Als zweites Argument wird die Variable selbst übergeben. Das Undefiniertheit repräsentierende Symbol $\star_\alpha$ wird durch die Konstante `$star` umgesetzt, welche als Argument ebenfalls ihren Domänentyp erhält und im folgenden Beispiel zusammen mit der Kennzeichnung `THE` angewendet wird:

```
hff(lem, conjecture, ( THE [X: $i] : ( X = ( $star @ $i ) ) ) ).
```

Die Funktionsweise von `THE` ähnelt der Funktionsweise des choice-Operators `@+` aus THF. Die Quantoren bleiben erhalten, iterieren in ihrer ursprünglichen Form allerdings nur über die Domäne der existenten Objekte:

```
hff(lem, conjecture, ( ! [X: $i] : ( $e @ $i @ X ) ) ).
```

Für eine Iteration über die Gesamtdomäne, über die existenten und nicht-existenten Objekte, wird `!+` bzw. `?+` vorgeschlagen:

```
hff(lem, conjecture,
    ( !+ [X: $i] : ( ( $e @ $i @ X ) | ( ~ ( $e @ $i @ X ) ) ) )
).
```

Die syntaktische Unterscheidung dieser beiden Quantorenpaare wurde aufgrund der beschränkten Popularität der Dual-Domänen-Semantik so gewählt, wie zuvor beschrieben. Die Idee von freier Logik gründet auf der Existenz einer einzelnen Domäne, der Domäne E_α. Objekte können außerhalb oder innerhalb dieser liegen. Um der allgemeinen Auffassung von freier Logik so nah wie möglich zu kommen, werden für die Quantifizierung über die Domäne E_α weiterhin die primären Quantoren `!` und `?` eingesetzt und `!+` bzw. `?+` nur bei Bedarf herangezogen.

4.2.4 Einbettung von HFF in THF

Um eine Übersetzung von HFF in THF zu ermöglichen, müssen die unter Kapitel 4.2.3 vorgestellten freie Logik-spezifischen Konstanten in THF definiert werden. Zudem müssen die in den Formeln verwendeten HFF-Konstanten so angepasst werden, dass sie mit der Syntax von THF und den Definitionen konform gehen. Die nachfolgende Einbettung wurde beispielhaft für Objekte des Typs `$i` skizziert, dieser kann aber durch beliebige Typkonstrukte ausgetauscht werden.

Die Domäne E_α wird, wie bereits aus der semantischen Einbettung hervorging, durch ein Existenzprädikat dargestellt. Dieses wird wie folgt definiert:

```
thf(freelogic_existence_type, type, ( eE: ( $i > $o ) ) ).
```

Jede Kombination `$e @ $i`, die in der HFF-Eingabedatei verwendet wird, wird in das Prädikat `eE` übersetzt. Die Domäne E_α wird durch folgendes Axiom nicht-leer gesetzt und ausgelassen, wenn eine inklusive Logik gewünscht ist:

```
thf(freelogic_nonemptyE_axiom, axiom, ( ? [X: $i] : ( eE @ X ) ) ).
```

Die Konstante `$star @ $i` wird als THF-Konstante `star` anhand folgender Typdefinition eingebettet:

```
thf(freelogic_star_type, type, ( star: $i ) ).

thf(freelogic_star_axiom, axiom, ( ~ ( eE @ star ) ) ).
```

Aufgrund der Präsumtion, dass die äußere Domäne nur nicht-leer sein kann, wird `star` als Teil der Domäne D_α deklariert. Die Einbettung des Allquantors wird ebenfalls über das Existenzprädikat als Wächter erreicht:

```
thf(freelogic_forall_type, type, ( fforall: ( ( $i > $o ) > $o ) ) ).

thf(freelogic_forall, definition,
        ( fforall =
          ( ^ [Phi: $i > $o] : ! [X: $i] : ( ( eE @ X ) => ( Phi @ X ) ) ) )
).
```

Eine Übersetzung von `!` im Sinne von HFF in ein äquivalentes THF-Konstrukt erfordert eine λ-Abstraktion und eine Applikation, damit die Neudefinition des Allquantors von den Theorembeweisern verarbeitet werden kann. Eines der Beispiele aus Kapitel 4.2.3 übersetzt sich dann wie folgt:

```
hff(lem, conjecture, ( ! [X: $i] : ( $e @ $i @ X ) ) ).
```

$\equiv$

```
thf(lem, conjecture, ( fforall @ ^ [X: $i] : ( eE @ X ) ) ).
```

Hierbei ist es wichtig zu erwähnen, dass in der Übersetzung dem `:` zwingend Klammern folgen müssen, um die Norm der THF-Syntax zu erfüllen. Die Ersetzung des Existenzquantors erfolgt äquivalent. Dieser wird, wie üblich, abhängig vom Allquantor definiert:

```
thf(freelogic_exists_type, type, ( fexists: ( ( $i > $o ) > $o ) ) ).

thf(freelogic_exists, definition,
        ( fexists =
          ( ^ [Phi: $i > $o] : ~ ( fforall @ ^ [X: $i] : ( ~ ( Phi @ X ) ) ) ) )
).
```

Das neue Quantorenpaar `!+` und `?+` iteriert ohne Zuhilfenahme des Existenzprädikats über die Gesamtdomäne D_α und wird damit durch `!` und `?` in ihrer eigentlichen Funktion ersetzt. Für die Einbettung der Kennzeichnung werden die Operatoren `$ite` und `@+` verwendet:

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

thf(ffthat, definition,
    ( i =
      ( ^ [Phi: $i > $o] :
          ( $ite
            @ ( ? [X: $i] :
                  ( ( eE @ X )
                  & ( Phi @ X )
                  & ( ! [Y: $i] :
                        ( ( ( eE @ Y ) & ( Phi @ Y ) ) => ( Y = X ) ) ) ) )
```

```
            @ ( @+[6] [X : $i] :
                    ( ( eE @ X ) & ( Phi @ X ) ) )
            @ star ) ) )
).
```

Sofern `$ite` und `@+` vermieden werden sollen, wird folgende alternative Einbettung auf der Grundlage von axiomatisierten Definitionen für choice- und if-then-else-Operatoren nach Backes (2010) und Backes und Brown (2011) gewählt:

```
thf(the_type, type, ( the: ( $i > $o ) > $i ) ).

thf(the, axiom,
    ( ! [P: $i > $o, A: $i] :
        ( ( P @ A ) => ( ( ! [X: $i] :
                             ( ( P @ X ) => ( X = A ) ) )
                             => ( P @ ( the @ P ) ) ) ) )
).

thf(if_type, type, ( if: $o > $i > $i > $i ) ).

thf(if, axiom,
    ( if =
      ( ^ [P: $o] :
          ( ^ [X: $i] :
               ( ^ [Y: $i] :
                   ( the @ ( ^ [Z: $i] :
                               ( ( ( P ) => ( Z = X ) )
                                 & ( ( ~ ( P ) ) => ( Z = Y ) ) ) ) ) ) ) ) )
).

thf(if_ax1, axiom,
    ( ! [P: $o] : ( ( P = $true ) | ( P = $false ) ) ) ).

thf(if_ax2_1,axiom,
    ( ! [X: $i,Y: $i] : ( ( if @ $false @ X @ Y ) = Y ) ) ).

thf(if_ax2_2,axiom,
    ( ! [X: $i,Y: $i] : ( ( if @ $true @ X @ Y ) = X ) ) ).

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

[6] Der choice-Operator `@+` gibt per Definition ein Objekt wieder, das eine bestimmte Bedingung erfüllt. Dieses Objekt muss nicht eindeutig sein. Für die Einbettung der Kennzeichnung ist Eindeutigkeit aber unablässig. In dieser Kodierung wird dies durch die Prämisse gewährleistet, dass es sich bei allen Objekte in der Domäne, die die Bedingung ebenfalls erfüllen, um dasselbe Objekt handelt.

```
thf(freelogic_fthat, definition,
    ( i =
      ( ^ [Phi: $i > $o] :
          ( if
            @ ( ? [X: $i] :
                  ( ( eE @ X )
                    & ( Phi @ X )
                    & ( ! [Y: $i] :
                          ( ( ( eE @ Y ) & ( Phi @ Y ) )
                            => ( Y = X ) ) ) ) )
            @ ( the @ ( ^ [X: $i] :
                          ( ( eE @ X ) & ( Phi @ X ) ) ) )
            @ star ) ) )
).
```

Soll nur einer der beiden vordefinierten Operatoren umgangen werden, dann wird eine Mischform der zwei Einbettungsalternativen gewählt. Jedoch sollte die Verwendung von `$ite` und `@+` bevorzugt werden, wenn der Zieltheorembeweiser die Möglichkeit dazu bietet, da die interne Behandlung der Operatoren im Allgemeinen als effektiver angesehen werden kann als die Verarbeitung von Axiomen.

Wie der formalen Einbettung aus Kapitel 4.1 zu entnehmen ist ändert sich die semantische Bedeutung der logischen Verknüpfungen $\neg$, $\rightarrow$ und $=$ für die positive freie Logik nicht, weswegen auf eine Kodierung dieser verzichtet wird. Eine Übersetzung ist in diesem Sinne nicht notwendig. Die Einbettung ist in ihrer Vollständigkeit auch als Anhang hinterlegt.

5 Umsetzung der Einbettung

Der Entwurf einer Einbettung ist nur der erste Schritt zu einer Übersetzung von freier Logik in Logik höherer Stufe. Die Einbettung wird nun in einen Gesamtkontext eingebunden und die Automatisierung der Übersetzung umfassend beschrieben.

5.1 Leo-III

Leo-III ist ein sich zum aktuellen Zeitpunkt in Entwicklung befindlicher „state-of-the-art" Theorembeweiser für die Logik höherer Stufe und der Nachfolger des Leo-II-Beweisers (Benzmüller, Theiss, Paulson und Fietzke 2008). Leo-III basiert auf Konzepten wie geordneter Paramodulation/Superposition und setzt im Gegensatz zu seinem Vorgänger auf ein polymorphes Typsystem (Wisniewski, Steen und Benzmüller 2015). Zudem wird großer Wert auf die Integrierbarkeit von semantischen Einbettungen nach Benzmüller (2013) gelegt, um sich dem logischen Schließen von nicht-klassischen Logiken aufbauend auf der Mächtigkeit von gewöhnlichen höherstufigen Beweisern annähern zu können.

Die vorliegende Arbeit wurde im Kontext von diesem Projekt initiiert, um einen ersten Schritt in Richtung der Unterstützung für nicht-klassische Logiken zu gehen. Da Leo-III in der funktionalen, objektorientierten Sprache Scala geschrieben wird, erfolgte die Implementierung der Umsetzung ebenfalls in Scala. Die zum Zeitpunkt der Implementierung aktuelle Version von Scala trug die Versionsnummer 2.11.7.

5.2 Implementierung

Die Übersetzung der einzelnen HFF-spezifischen Elemente wurde bereits in Kapitel 4.2.4 im Detail betrachtet. Im Folgenden wird die Anwendung beschrieben, die diese Elemente anhand einer differenzierten Syntaxanalyse lokalisiert und deren Übersetzung durchführt. Die Anwendung erhält eine beliebige Eingabedatei, idealerweise eine .p- oder .tpi-Datei, die alle Arten von TPTP-konformen Anweisungen inklusive HFF-Formeln enthalten kann. Für die resultierende Ausgabedatei werden die identifizierten HFF-Anweisungen übersetzt sowie adäquate TPI-Anweisungen interpretiert. Die übrigen Anweisungen werden bis auf minimale Modifikationen an den Pfaden von inkludierten Eingabedateien unverändert in die Ausgabedatei übernommen. Das Diagramm in Abbildung 4 veranschaulicht den exakten Ablauf der Anwendung.

Für die Verarbeitung der Eingabe wird die Eingabedatei zuerst in separate Anweisungen segmentiert, um aufgrund der syntaktischen Unabhängigkeit dieser die Option einer parallelisiert durchgeführten Übersetzung aller Einzelanweisungen offen zu halten. Als Trennsymbol dient der Punkt, der Abschluss einer jeden nicht-leeren Anweisung. Punkte in Kommentaren und Zahlen sowie Punkte zwischen einfachen Anführungszeichen werden übersprungen. Kommentare unmittelbar vor einer Anweisung werden syntaktisch zu dieser gezählt. Die letzte Anweisung muss nicht mit einem Punkt abschließen. In diesem Fall handelt es sich um eine leere Anweisung, die nur typografischen Weißraum oder Kommentare umfasst. Im Anschluss wird jede Anweisung für sich analysiert. Hierfür wurden sechs verschiedene Parser entworfen, einer für jede Art von TPTP-Anweisung, die in der Eingabe enthalten sein

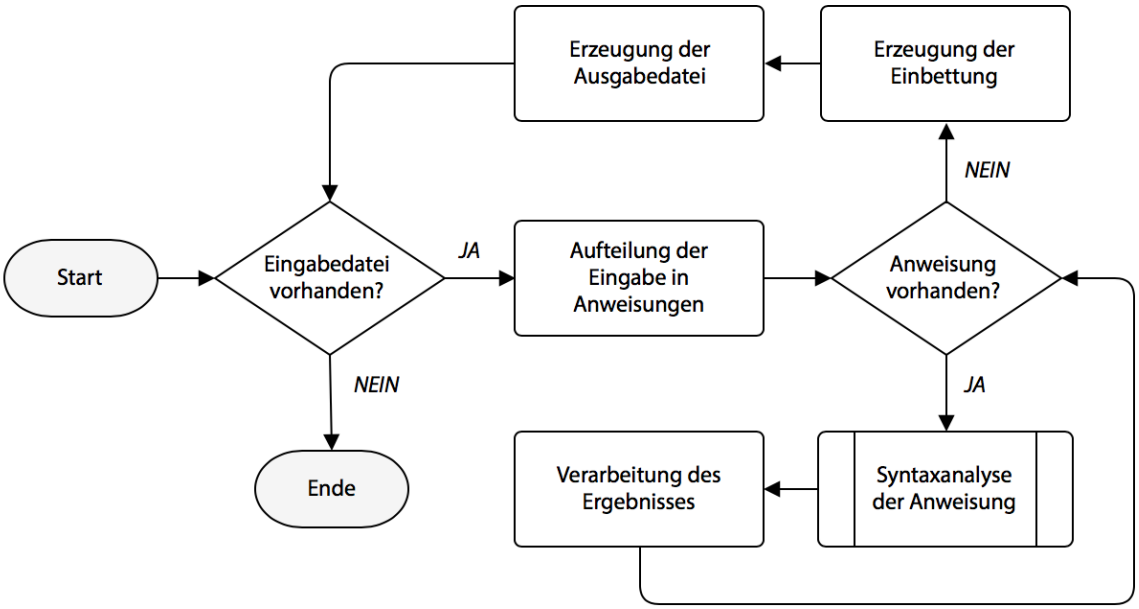

Abb. 4: Ablaufdiagramm der Anwendung

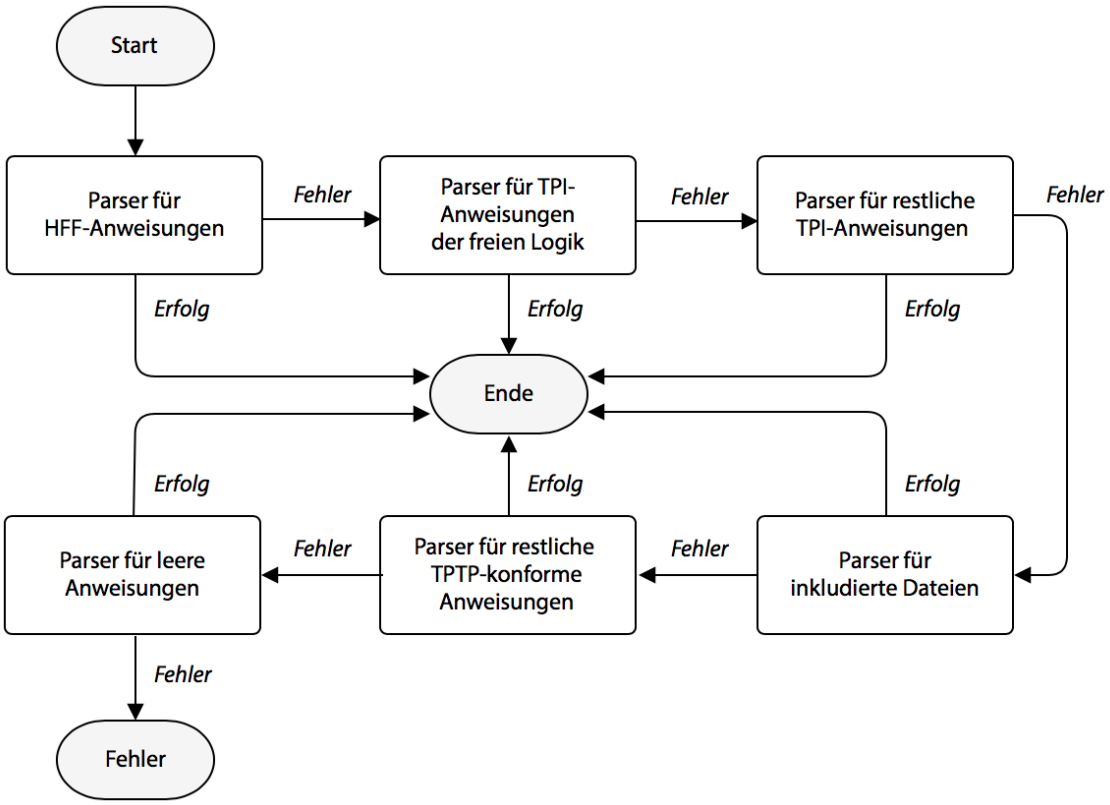

Abb. 5: Ablaufdiagramm der Syntaxanalyse einer Anweisung

kann. Diese werden nacheinander durchlaufen bis einer die Anweisung erfolgreich analysiert hat (vgl. Abbildung 5) und einen entsprechenden Rückgabewert für die Weiterverarbeitung zurückliefert. Wenn keiner der Parser die Anweisung verifizieren konnte, dann ist die Anweisung fehlerhaft und der Benutzer erhält unter Angabe einer Zeilen- und Spaltennummer sowie der stattdessen erwarteten Eingabe die Fehlermeldung des Parsers, der die Eingabe am weitesten vergleichen konnte, angezeigt.

Jeder der sechs Parser analysiert auf der Basis von PEG (*parsing expression grammar*), einer leichtgewichtigen Alternative zu kontextfreien Grammatiken, die Backtracking unterstützt. Ein solcher Parser operiert in der Regel direkt auf der Eingabezeichenfolge und verzichtet auf eine vorherige lexikalische Analyse. Für die Implementierung dieser Parser wurde auf Parsergeneratoren gesetzt, wofür die Bibliothek Parboiled2 (Doenitz und Myltsev 2013) herangezogen wurde. Diese wurde den Parserkombinatoren von Scala vorgezogen, da der Funktionsumfang der Bibliothek insbesondere im Hinblick auf die Flexibilität der domänenspezifischen Sprache mehr überzeugen konnte. Zudem setzt Parboiled2 zur Steigerung der Leistungfähigkeit auf effizientere Strukturen sowie Scala Macros (vgl. Béguet und Jonnalagedda 2014; Kurš, Vraný, Ghafari, Lungu und Nierstrasz 2016) und vergleicht damit die Performance seiner zur Kompilierzeit generierten Parser mit der von handgeschriebenen Parsern. Der Aufruf eines solchen Parsers und das Abfangen seiner Ergebnisse wird wie folgt durchgeführt:

```
parser.Statement_HFF.run() match {
    case Success(result) =>
        return result
    case Failure(new_parseError: ParseError) =>
        handleParseError
    case Failure(new_parseError) =>
        debug("unexpected error during parsing run")
}
```

Die Entscheidung für die Aufteilung der Syntaxanalyse auf verschiedene Parser ermöglicht die Anbindung externer Parser, um so zum Beispiel auch THF oder andere Logiksprachen direkt aus der Anwendung heraus analysieren zu können ohne an die bereits bestehenden Parser gebunden zu sein. Die Parser bleiben übersichtlich und können einfach ergänzt oder ausgetauscht werden.

Die Grammatik, die dem Parser für HFF-Anweisungen zu Grunde liegt, lautet wie folgt:

```
statement    ::=    "hff" "(" <id> "," <formula_type> "," <formula>
                    ")" "."
id           ::=    [a-z] [a-zA-Z0-9_]* | [0-9]+
formula_type ::=    "axiom" | "conjecture" | "definition"| "lemma" | …
formula      ::=    "(" <formula>? ")" <formula>?
                    | "'" <formula>? "'" <formula>?
                    | <definitions>
```

```
                    | "eE_" <formula>?
                    | "$e" "@" "$"? [a-zA-Z0-9]+ "@" <formula>
                    | "star_" <formula>?
                    | "$star" "@" "$"? [a-zA-Z0-9]+ <formula>
                    | <quantifier_outerdomain>
                    | <quantifier_innerdomain>
                    | <description>
                    | ^[()'.] <formula>?
comment      ::=    "%" ^[\n] "\n"
definitions  ::=    "[" variable_definition ( ","
                    variable_definition )* "]" ":" <formula>
quantifier_outerdomain ::= ( "!+" | "?+" ) "[" variable_definition
                    ( "," variable_definition )* "]" ":" <formula>
quantifier_innerdomain ::= ( "!" | "?" ) "[" variable_definition
                    variables_list
description  ::=    "THE" "[" variable_definition variables_list
variables_list ::=  "," variable_definition variables_list | "]" ":"
                    <formula>
variable_definition ::= ^[:,[].]+ ":" <type>
type         ::=    "(" <type> ")" <type>? | ^[()[]:,.]+ <type>?
```

Token sind mit Anführungszeichen markiert, weiterführende Regeln mit spitzen Klammern und reguläre Ausdrücke sind in eckige Klammern gesetzt. Die in der Notation verwendete Syntax für reguläre Ausdrücke entspricht dem allgemeinen Standard. Die Grammatik wurde leicht adaptiert, um ihre Lesbarkeit zu vereinfachen. Zum Beispiel können an diversen Stellen Leerraum und Kommentare eingesetzt werden, die hier der Übersichtlichkeit halber ausgelassen wurden. Die Grammatik konzentriert sich darauf, Quantoren und Konstanten der freien Logik zu finden und deren syntaktische Verwendung, insbesondere die zugehörigen Variablendefinitonen und Applikationen, auf ihre Korrektheit zu überprüfen. Sollte eine der Definitionen, die in der anschließenden Übersetzung hinzugefügt werden, bereits in der eigentlichen Eingabe definiert worden sein, so werden diese ebenfalls aufgespürt und umbenannt. Zusätzlich werden zusammengehörende Klammern und einfache Anführungszeichen erkannt und geprüft. Bis auf einige wenige Ausnahmen wie die grundlegende Struktur von TPTP-konformen Anweisungen und die Syntaxregeln für Bezeichner wurde die exakte Syntax von TPTP außer Acht gelassen, da deren Beachtung nicht zwingend für die Übersetzung notwendig ist. Die syntaktische Korrektheit der Eingabe wird während dem der Übersetzung obligatorisch nachstehenden Beweisvorgang sichergestellt, sodass die Übersetzung nur als eine Art Vorverarbeitungsschritt anzusehen ist. Ein Ausschnitt des Parboiled2-Parsers, der diese Grammatik erkennen kann, sieht wie folgt aus:

```
def Statement_HFF: Rule1String = rule {
    clearSB ~ quiet( W ) ~ HFF_Language.named("hff") ~
    quiet( W ) ~ leftParenthesis.named("left parenthesis") ~
```

```
    quiet( W ) ~ ID ~ quiet( W ) ~ comma ~ quiet( W ) ~
    FormulaType ~ quiet( W ) ~ comma ~ quiet( W ) ~
    Formula.named("formula") ~ ( comma ~ Annotation ).? ~
    rightParenthesis.named("right parenthesis") ~ quiet( W ) ~
    dot ~ EOI ~ push(sb.toString) }

def comma: Rule0 = rule { CharPredicate(',') ~ appendSB(",") }

def VariableDefinition(quantifier: String): Rule0 = rule {
    ( noneOf(":,.").+ ).named("variable name") ~ ( colon ~
    ( capture( Type ) ~> { (var_type: String) =>
    { appendSB( handleFoundVarType(quantifier, var_type) ) } } |
    fail("type") ) | fail("colon") ) }
```

Neben altbekannten Operatoren – `.+` für sich mindestens einmal wiederholende und `.?` für optionale Regeln – und der Regelkonkatenation durch Tilden unterstützt Parboiled2s domänenspezifische Sprache auch weitere nützliche Parseraktionen: Beispielsweise gibt `.named` den Namen der Regel an, der im Fehlerfall ausgegeben wird und `quiet` sorgt gegenteilig dafür, dass die Regel im Verborgenen bleibt. Mit `fail` kann ein mögliches Backtracking des Parsers verhindert werden, indem die Anwendung der Regel im Falle einer missglückten Übereinstimmung fehlschlägt und damit den Parsevorgang beendet. `capture` fängt die Analyse der darunterliegenden Regel auf und stellt sie zur Weiterverarbeitung durch `~>` zur Verfügung. In dem gezeigten Parserauschnitt wird die Erkennung des Typs in einer Variablendefinition mitgeschnitten und der Typ dann anhand der Methode `handleFoundVarType` ausgewertet. `SB` ist eine Erweiterung des Parboiled2-Parsers mit der direkt während dem Parsevorgang eine Zeichenkette zusammengesetzt werden kann. Für die Übersetzung wird dies so genutzt, dass ein gelesener Buchstabe oder eine gelesene Zeichenkombination sofort wieder an die Ausgabezeichenkette angehängt wird. Wenn dabei eine der HFF-spezifischen Regeln ausgelöst wird, dann wird nicht die gleiche Zeichenkette an die Ausgabezeichenkette angefügt, welche auch gelesen wurde, sondern deren korrespondierende Übersetzung in THF. Solche `SB`-Methoden sind im oberen Beispiel kursiv markiert, befinden sich aber größtenteils in untergeordneten Regeln. Sofern eine Anweisung komplett von dem Parser gelesen werden konnte wird die übersetzte Eingabezeichenfolge zurückgegeben, die vervollständigt durch eine Einbettung an die Ausgabedatei angefügt werden kann.

Da sich die Einbettung nicht auf einen Variablentyp im Speziellen beschränken soll, sondern die Operatoren und Konstanten für alle Typen, die in der Eingabedatei Verwendung gefunden haben, eingebettet werden sollen, wird bei der Syntaxanalyse der Typ jeder gelesenen Variablendefinition verarbeitet, indem er mit den Daten der bisher verwerteten Typen verglichen wird. Sofern der Typ noch nicht zusammen mit diesem Operator verwendet wurde, dann wird er den Daten hinzugefügt. Hierbei wird jeder Typ mit einem bestimmten globalen Index identifiziert, der bei der Ersetzung der Zeichenkette Beachtung findet. Das Konstrukt `$e @ $i @` ersetzt sich so zum Beispiel zu `eE_1 @` während das Konstrukt `$e @ ($i>$o) @` zu `eE_2 @` ersetzt wird, wobei der Typ `$i` intern den Index eins erhält und `$i>$o` den Index

zwei. Eine hypothetische Allquantifizierung `![X:$i>$o]` wird dann in der selben Ausgabedatei zu `(fforall_2 @ ^[X:$i>$o] ... )` ersetzt.[7] Zu diesem Zweck wurde die Einbettung bereits so vorbereitet, dass nach Beendigung der Syntaxanalyse aller Anweisungen jeder Einbettungsteil wiederholt zu der Gesamteinbettung hinzugefügt werden kann, abhängig von den Typen, die für genau diesen Einbettungspart gefunden wurden. Die sich ergebende Einbettung enthält somit nur exakt die Definitionen, die tatsächlich auch benötigt werden.[8] So wird die Einbettung schlank gehalten und die Theorembeweiser müssen nicht mehr Input verarbeiten als notwendig. Bei der Zusammensetzung der Einbettung werden die einzelnen Teile abhängig von bestimmten Parameter gewählt (vgl. Kapitel 4.2.4). Die TPI-Anweisung, die in Kapitel 4.2.2 speziell für die freie Logik vorgeschlagen wurde und die die Parameter für die Einbettung kontrolliert, wird anhand folgender Grammatik und zugehörigem Parser ausgelesen:

```
statements.  ::=   "tpi" "(" <id> "," "set_logic" "," "free"
                   "(" <parameters> ")" ")" "."
id           ::=   [a-z] [a-zA-Z0-9_]* | [0-9]+
parameters   ::=   <e> "," ( <choice> "," <ite> | <ite> "," <choice> )
                   | <choice>  "," ( <e> "," <ite> | <ite> "," <e> )
                   | <ite> "," ( <e> "," <choice> | <choice> "," <e> )
e            ::=   "'" <para_e> "'" "=" "'" <value_e> "'"
choice       ::=   "'" <para_choice> "'" "=" "'" <value_yes_no> "'"
ite          ::=   "'" <para_ite> "'" "=" "'" <value_yes_no> "'"
value_e      ::=   "$empty" | "$non_empty"
value_yes_no ::=   "$yes" | "$no"
para_e       ::=   "$E"
para_choice  ::=   "$choice"
para_ite     ::=   "$ite"
```

Wie bereits erwähnt wird die Eingabe aller drei Parameter verlangt, jedoch wäre eine Anpassung kein Problem, sollten zukünftig im Kontext von TPTP Standardwerte vorgesehen sein. Der Parser gibt eine Liste der ausgelesenen Parameter zurück sowie eine Liste der gefundenen Leerzeilen und Kommentare. Die Parameter werden für die der Syntaxanalyse anschließende Erzeugung der Einbettung verarbeitet und die Anweisung nach deren Interpretation verworfen. Kommentare und Leerzeilen bleiben erhalten und werden der Ausgabedatei hinzugefügt, um die Zeilenzuordnung größtenteils beizubehalten und somit eine etwaige Fehlersuche zu vereinfachen.

Anweisungen, die weitere Eingabedateien inkludieren, haben folgende Form:

```
include('Axioms/Ax001.p').
```

[7] Es sei auch erwähnt, dass eine Aneinanderreihung wie in `![X:$i,Y:a]` aus syntaktischen Gründen zu `(fforall_1 @ ^[X:$i]: (fforall_3 @ ^[Y:a] ... ))` aufgeteilt werden muss.

[8] Das Axiom, das die Existenz des nicht-definierten Objekts postuliert, wird für jeden Typ eingebettet.

Diese Anweisungen werden ausgelesen, die spezifizierte Datei zu der Liste der noch zu verarbeitenden Eingabedateien hinzugefügt und die Anweisung zusammen mit dem neuen Pfad zu der in Kürze ebenfalls übersetzten Datei weitergegeben, um diese modifizierte Anweisung an die Ausgabedatei anzuhängen.

Die Anwendung ist als Kommandozeilenwerkzeug konzipiert worden und beherrscht folgende Optionen, die neben der Eingebadatei durch einen Befehl der Form `tff2thf [options] <file>` mitgegeben werden können:

```
-o, --out <file>        -e, --empty_e <boolean>   -u, --use_include
-w, --overwrite         -c, --choice <boolean>    -r, --result <logic>
--verbose <level>       -i, --ite <boolean>       --help
--debug                 --more_axioms
```

Die Standardoptionen werden um die Optionen `--empty_e`, `--choice` und `--ite` ergänzt, eine weitere Möglichkeit die Parameter der Einbettung unabhängig von einer TPI-Anweisung zu beeinflussen. Sollte jedoch trotzdem eine TPI-Anweisung gefunden werden, dann werden die über die Kommandozeile festgelegten Parameter überschrieben. Die Option `--more_axioms` setzt zusätzliche Axiome für Einbettungen mit einem axiomatisierten choice-Operator. Sie wird nach Beendigung des Übersetzungsvorgangs vorgeschlagen, falls für die Übersetzung der choice-Operator mittels Axiomen eingebettet wurde. Die Option `--use_include` steuert, ob die Einbettung an den Anfang jeder Eingabedatei eingefügt werden soll, oder ob stattdessen eine separate Einbettungsdatei angelegt und pro Eingabedatei eine einzeilige Anweisung für die Inkludierung dieser Datei geschrieben werden soll. `--result` legt die Logiksprache fest, in die die HFF-Anweisungen übersetzt werden sollen. Prinzipiell wird eine Übersetzung in THF bevorzugt, in einigen Fällen kann die gewünschte Zielsprache, zum Beispiel für eine Folgeübersetzung von (freier) Modallogik nach THF, abweichen. Die Anwendung hält sich an gängige Ein- und Ausgabekonventionen. Eingabedateien können als relativer oder absoluter Pfad angegeben werden, wobei auch die TPTP-Umgebungsvariable `$TPTP` ausgelesen wird, sofern diese gesetzt ist. Da möglicherweise mehrere Dateien nacheinander für eine erfolgreiche Ausgabe beschrieben werden müssen, wird gewährleistet, dass bei einem Fehler alle Ausgabedateien in ihren Ausgangszustand zurückversetzt werden. Bereits vorhandene Dateien, die überschrieben werden sollten, werden anhand einer Sicherheitskopie wiederhergestellt, neue gelöscht. So wird sichergestellt, dass die Ausgabe konsistent bleibt.

Die Implementierung wurde anhand von 401 einfachen Problemen der TPTP-Bibliothek getestet sowie bei der Formalisierung von Kategorientheorie angewendet. Die Auswertung wird in Kapitel 7 diskutiert.

6 Anwendung am Beispiel der Kategorientheorie

Die Kategorientheorie ist ein Zweig der Mathematik, der die Gemeinsamkeiten von mathematischen Strukturen wie Gruppen, Körper und Ringe, sogenannte Kategorien, erörtert und deren strukturelle Unterschiede zu erfassen versucht. Das 1990 erschienene Buch *Categories, Allegories* von Freyd und Scedrov über Kategorientheorie basiert auf der Definition einer partiellen Operation, sodass sich die freie Logik im Speziellen für die Formalisierung dieses Buches eignet. Kategorientheorie nach Freyd und Scedrov soll im Folgenden (teil-)formalisiert werden, um die erfolgreiche Automatisierung von freier Logik in Logik höherer Stufe im Rahmen dieser Arbeit auszutesten.

6.1 Formalisierungen

Die folgenden Formalisierungen stützen sich alle auf die Sprachelemente, die in Kapitel 4.2.3 vorgestellt wurden.

Freyd und Scedrov definieren zu Beginn drei Operatoren, um die Theorie von Kategorien einzuleiten: die Quelle $\square X_\iota$ und das Ziel $X_\iota \square$ eines Morphismus X_ι mit Individuentyp ι sowie die Komposition zweier Morphismen $X_\iota \cdot Y_\iota$. Diese grundlegenden Definitionen werden wie folgt als Konstanten formalisiert, als zwei unäre und eine binäre Operation:

```
hff(source_type, type, ( source: $i > $i ) ).

hff(target_type, type, ( target: $i > $i ) ).

hff(composition_type, type, ( comp: $i > $i > $i ) ).
```

X_ι und Y_ι werden als Variablen eines Typs `$i` umgesetzt, die im Sinne der Kategorientheorie auch Morphismen genannt werden. Das ursprünglich auf diesen Operatoren aufbauende, von Freyd und Scedrov angegebene Axiomensystem konnte von Benzmüller und Scott (2016b) bereits als inkonsistent nachgewiesen werden. Der Beweis hierfür lässt sich einfach anhand eines Theorembeweisers nachvollziehen und wird an dieser Stelle ausgespart. Für die Formalisierungen wird stattdessen auf ein äquivalentes, aber konsistentes Axiomensystem von Scott (1979) zurückgegriffen, um die Definitionen von Freyd und Scedrov gegenzuprüfen. Dieses wird wie folgt formalisiert:

```
hff(scott_s1_axiom, axiom,
    ( !+ [X: $i] :
         ( ( $e @ $i @ ( source @ X ) ) => ( $e @ $i @ X ) ) )
).

hff(scott_s2_axiom, axiom,
    ( !+ [X: $i] :
         ( ( $e @ $i @ ( target @ X ) ) => ( $e @ $i @ X ) ) )
).
```

```
hff(scott_s3_axiom, axiom,
    ( !+ [X: $i] :
         ( !+ [Y: $i] :
              ( ( $e @ $i @ ( comp @ X @ Y ) )
                <=> ( eq1 @ ( source @ X ) @ ( target @ Y ) ) ) ) )
).

hff(scott_s4_axiom, axiom,
    ( !+ [X: $i] :
         ( !+ [Y: $i] :
             ( !+ [Z: $i] :
                  ( eq2
                    @ ( comp @ X @ ( comp @ Y @ Z ) )
                    @ ( comp @ ( comp @ X @ Y ) @ Z ) ) ) ) )
).

hff(scott_s5_axiom, axiom,
    ( !+ [X: $i] :
         ( eq2 @ ( comp @ ( source @ X ) @ X ) @ X ) )
).

hff(scott_s6_axiom, axiom,
    ( !+ [X: $i] :
         ( eq2 @ ( comp @ X @ ( target @ X ) ) @ X ) )
).
```

Im Folgenden sind die hervorgehobenen (Kapitel-)Nummerierungen denen in *Categories, Allegories* nachempfunden, um eine ähnliche Struktur für die Orientierung beizubehalten. $\equiv_{\text{HFF}}$ gibt die äquivalente Umformung einer Formel in HFF an.

1.11. Gleichheit, in diesem Fall wie die von Freyd und Scedrov vorausgesetzte Kleene-Gleichheit, ist wie folgt definiert:

$$X_\iota \overset{\text{s}}{=} Y_\iota \;\equiv\; (E!X_\iota \vee E!Y_\iota) \to (X_\iota \overset{\text{w}}{=} Y_\iota)$$

$\equiv_{\text{HFF}}$

```
hff(eq2_type, type, ( eq2: $i > $i > $o ) ).

hff(eq2, definition,
    ( eq2 =
      ( ^ [X: $i, Y: $i] :
          ( ( ( $e @ $i @ X ) | ( $e @ $i @ Y ) )
            => ( eq1 @ X @ Y ) ) ) )
).
```

Scott nutzt zudem eine zweite, schwächere, nicht-reflexive Form von Gleichheit für die Definition der Kleene-Gleichheit und auch für das dritte Axiom, um ein konsistentes System zu erlangen:

$$X_\iota \stackrel{\text{w}}{=} Y_\iota \equiv E!X_\iota \wedge E!Y_\iota \wedge (X_\iota = Y_\iota)$$

$\equiv_{\text{HFF}}$

```
hff(eq1_type, type, ( eq1: $i > $i > $o ) ).

hff(eq1, definition,
    ( eq1 =
      ( ^ [X: $i, Y: $i] :
          ( ( $e @ $i @ X ) & ( $e @ $i @ Y ) & ( X = Y ) ) ) )
).
```

1.12. Ergänzt werden diese Definitionen um eine dritte Gleichheit, die gerichtete Kleene-Gleichheit:

$$X_\iota \stackrel{\text{d}}{=} Y_\iota \equiv E!X_\iota \rightarrow (X_\iota \stackrel{\text{w}}{=} Y_\iota)$$

$\equiv_{\text{HFF}}$

```
hff(eq3_type, type, ( eq3: $i > $i > $o ) ).

hff(eq3, definition,
    ( eq3 =
      ( ^ [X: $i, Y: $i] :
          ( ( $e @ $i @ X ) => ( eq1 @ X @ Y ) ) ) )
).
```

Das nachstehende Lemma ist eines der ersten im Buch und beschäftigt sich mit der gerichteten Gleichheit:

$$\Box(X_\iota \cdot Y_\iota) \stackrel{\text{s}}{=} \Box(X_\iota \cdot (\Box Y_\iota)) \leftrightarrow \Box(X_\iota \cdot Y_\iota) \stackrel{\text{d}}{=} \Box X_\iota$$

$\equiv_{\text{HFF}}$

```
hff(lem1_12, conjecture,
    ( ! [X: $i] :
        ( ! [Y: $i] :
            ( ( eq2
                @ ( source @ ( comp @ X @ Y ) )
                @ ( source @ ( comp @ X @ ( source @ Y ) ) ) )
              <=> ( eq3
                    @ ( source @ ( comp @ X @ Y ) )
                    @ ( source @ X ) ) ) ) )
).
```

1.13. Die kommenden zwei Lemmata gelten, da $\Box(\Box X_\iota) \overset{s}{=} \Box((\Box X_\iota)\Box) \overset{s}{=} (\Box X_\iota)\Box \overset{s}{=} \Box X_\iota$ und äquivalent für $X_\iota\Box$.

$\Box(\Box X_\iota) \overset{s}{=} \Box X_\iota$

$\equiv_{\text{HFF}}$

```
hff(lem1_13_1, conjecture,
    ( ! [X: $i] :
        ( eq2 @ ( source @ ( source @ X ) ) @ ( source @ X ) ) )
).
```

$(X_\iota\Box)\Box \overset{s}{=} X_\iota\Box$

$\equiv_{\text{HFF}}$

```
hff(lem1_13_2, conjecture,
    ( ! [X: $i] :
        ( eq2 @ ( target @ ( target @ X ) ) @ ( target @ X ) ) )
).
```

1.13. Die folgenden Aussagen sind gleichwertige Eigenschaften eines Morphismus E_ι:

$\exists X_\iota.\ E_\iota \overset{s}{=} \Box X_\iota$
$\exists X_\iota.\ E_\iota \overset{s}{=} X_\iota\Box$
$E_\iota \overset{s}{=} \Box E_\iota$
$E_\iota \overset{s}{=} E_\iota\Box$
$\forall X_\iota.\ E_\iota \cdot X_\iota \overset{d}{=} X_\iota$
$\forall X_\iota.\ X_\iota \cdot E_\iota \overset{d}{=} X_\iota\,.$

Ein solches E_ι wird Identitätsmorphismus genannt. Die zugehörige Formalisierung lautet:

```
hff(idM_type, type, ( idM: $i > $o ) ).

hff(idM, definition,
    ( idM =
      ( ^ [X: $i] :
          ( ( $e @ $i @ X ) & ( eq2 @ X @ ( source @ X ) ) ) ) )
).

hff(lem_identityMorphism, conjecture,
    ( ! [X: $i] :
        ( ( ( idM @ X ) <=> ( ? [Y: $i] : ( eq2 @ X @ ( source @ Y ) ) ) )
          & ( ( idM @ X ) <=> ( ? [Y: $i] : ( eq2 @ X @ ( target @ Y ) ) ) ) )
```

```
          & ( ( idM @ X ) <=> ( eq2 @ X @ ( source @ X ) ) )
          & ( ( idM @ X ) <=> ( eq2 @ X @ ( target @ X ) ) )
          & ( ( idM @ X ) <=> ( ! [Y: $i] : ( eq3 @ ( comp @ X @ Y ) @ Y ) ) )
          & ( ( idM @ X ) <=> ( ! [Y: $i] :
                                  ( eq3 @ ( comp @ Y @ X ) @ Y ) ) ) ) )
).
```

1.17. Ein Morphismus X_ι ist linksinvertierbar, wenn es einen Morphismus Y_ι gibt, sodass $Y_\iota \cdot X_\iota$ ein Identitätsmorphismus ist, und rechtsinvertierbar, wenn es einen Morphismus Z_ι gibt, sodass $X_\iota \cdot Z_\iota$ ein Identitätsmorphismus ist. Ein Isomorphismus ist links- und rechtsinvertierbar. Diese Definitionen werden wie folgt formalisiert:

```
hff(idM_type, type, ( idM: $i > $o ) ).

hff(idM, axiom,
    ( idM =
      ( ^ [X: $i] :
          ( ( $e @ $i @ X ) & ( eq2 @ X @ ( source @ X ) ) ) ) )
).

hff(lI_type, type, ( lI: $i > $o ) ).

hff(lI, definition,
    ( lI =
      ( ^ [X: $i] :
          ( ( $e @ $i @ X ) & ( ? [Y: $i] : ( idM @ ( comp @ Y @ X ) ) ) ) ) )
).

hff(rI_type, type, ( rI: $i > $o ) ).

hff(rI, definition,
    ( rI =
      ( ^ [X: $i] :
          ( ( $e @ $i @ X ) & ( ? [Z: $i] : ( idM @ ( comp @ X @ Z ) ) ) ) ) )
).

hff(iso_type, type, ( iso: $i > $o ) ).

hff(iso, definition,
    ( iso =
      ( ^ [X: $i] : ( ( lI @ X ) & ( rI @ X ) ) ) )
).
```

Ein Isomorphismus hat eine eindeutige Linksinverse und eine eindeutige Rechtsinverse und diese sind identisch, da $Y_\iota \overset{s}{=} Y_\iota \cdot (X_\iota \cdot Z_\iota) \overset{s}{=} (Y_\iota \cdot X_\iota) \cdot Z_\iota \overset{s}{=} Z_\iota$:

```
hff(isomorphism, conjecture,
    ( ! [X: $i] :
        ( iso @ X )
        => ( ? [Y: $i] :
               ( ? [Z: $i] :
                   ( ( idM @ ( comp @ X @ Z ) )
                     & ( idM @ ( comp @ Y @ X ) )
                     & ( eq2 @ Y @ ( comp @ Y @ ( comp @ X @ Z ) ) )
                     & ( eq2
                         @ ( comp @ Y @ ( comp @ X @ Z ) )
                         @ ( comp @ ( comp @ Y @ X ) @ Z ) )
                     & ( eq2 @ ( comp @ ( comp @ Y @ X ) @ Z ) @ Z ) ) ) ) )
).
```

1.18. Gegeben zwei Kategorien A und B ist eine Funktion $f\colon A \mapsto B$ ein Funktor genau dann, wenn für drei Morphismen X_a, Y_a, Z_a mit a als Individuentyp gilt:

$$\begin{aligned}
&\Box X_a \overset{s}{=} Y_a \;\rightarrow\; \Box(f(X_a)) \overset{s}{=} f(Y_a)\\
&X_a\Box \overset{s}{=} Y_a \;\rightarrow\; (f(X_a)\Box) \overset{s}{=} f(Y_a)\\
&X_a \cdot Y_a \overset{s}{=} Z_a \;\rightarrow\; f(X_a)\cdot f(Y_a) \overset{s}{=} f(Z_a)\,.
\end{aligned}$$

Für diese Formalisierung müssen alle bisherigen Definitionen auf zwei verschiedene Typen erweitert werden. Dies wird hier nicht im Detail ausgeführt, es sei lediglich erwähnt, dass die neuen Operatoren, am Beispiel des Operators `source`, die Namen `source_a` bzw. `source_b` für zwei Individuentypen `a` und `b` tragen. Das gesamte Problem mit allen erneuerten Operatoren kann im Anhang eingesehen werden. Demnach sind Funktoren gegeben durch folgende Definition:

```
hff(functor_type, type, ( functor: ( a > b ) > $o ) ).

hff(functor, definition,
    ( functor =
      ( ^ [F: a > b] :
          ( ( ! [A: a] :
                ( ! [B: a] :
                    ( eq2_a @ ( source_a @ A ) @ B )
                    => ( eq2_b @ ( source_b @ ( F @ A ) ) @ ( F @ B ) ) ) )
            & ( ! [A: a] :
                  ( ! [B: a] :
                      ( eq2_a @ ( target_a @ A ) @ B )
                      => ( eq2_b @ ( target_b @ ( F @ A ) ) @ ( F @ B ) ) ) )
            & ( ! [A: a] :
                  ( ! [B: a] :
                      ( ! [C: a] :
```

```
                        ( eq2_a @ ( comp_a @ A @ B ) @ C )
                        => ( eq2_b
                             @ ( comp_b @ ( F @ A ) @ ( F @ B ) )
                             @ ( F @ C ) ) ) ) ) ) ) )
).
```

Der Funktorenbegriff kann laut Freyd und Scedrov alternativ auch durch folgende drei Eigenschaften beschrieben werden:

$$f(\Box X_\iota) \overset{s}{=} \Box(f(X_\iota))$$
$$f(X_\iota \Box) \overset{s}{=} (f(X_\iota))\Box$$
$$f(X_\iota \cdot Y_\iota) \overset{d}{=} (f(X_\iota) \cdot f(Y_\iota)) \,.$$

Für die Formularisierung der Äquivalenz wird zunächst nur versucht, zu zeigen, dass eine Richtung gilt:

```
hff(functor_equivalence, conjecture,
    ( ! [F: a > b] :
         ( ( functor @ F )
           => ( ( ! [X: a] :
                    ( eq2_b
                      @ ( F @ ( source_a @ X ) )
                      @ ( source_b @ ( F @ X ) ) ) )
                & ( ! [X: a] :
                      ( eq2_b
                        @ ( F @ ( target_a @ X ) )
                        @ ( target_b @ ( F @ X ) ) ) )
                & ( ! [X: a] :
                      ( ! [Y: a] :
                          ( eq3_b
                            @ ( F @ ( comp_a @ X @ Y ) )
                            @ ( comp_b @ ( F @ X ) @ ( F @ Y ) ) ) ) ) ) ) )
).
```

Doch leider konnte diese Annahme nicht bewiesen werden, die Beweissuche wurde nach einer angemessenen Zeitspanne abgebrochen. Die Gegenrichtung indes konnte ohne Probleme gezeigt werden, genauso wie die folgende Adaption:

```
hff(functor_equivalence, conjecture,
    ( ! [F: a > b] :
        ( ( functor @ F )
           => ( ( ! [X: a] :
                    ( eq2_b
```

```
                @ ( F @ ( source_a @ X ) )
                @ ( source_b @ ( F @ X ) ) ) )
          & ( ! [X: a] :
                ( eq2_b
                  @ ( F @ ( target_a @ X ) )
                  @ ( target_b @ ( F @ X ) ) ) ) ) ) ) )
).
```

Alle anderen der bisher gezeigten Lemmata und Konjekturen konnten nach deren Übersetzung in die Logik höherer Stufe von einem automatischen Theorembeweiser als valide bestätigt werden. Weitere Lemmata, unter anderem auch eines, das auf die Kennzeichnung zurückkommt, sowie eine weiter fortgeschrittene Formalisierung der Kategorientheorie in Isabelle/HOL befinden sich im Projektordner der Anwendung.

7 Evaluation

Im vorherigen Kapitel wurde festgestellt, dass einfache kategorientheoretische Probleme mithilfe diverser Definitionen und Axiome ohne nennenswerte Schwierigkeiten übersetzt und anschließend auch bewiesen werden können. Einige wenige Probleme sind nicht entscheidbar, doch ist fragwürdig, ob die Lemmata gemäß den Vorstellungen von Freyd und Scedrov umgesetzt wurden. Eine intuitive Formularisierung der Kategorientheorie auf dem Level von HFF ist also erfolgreich realisierbar. Zusätzliche Tests mit verschiedenen Problemen aus der TPTP waren im Gegensatz dazu weniger vielversprechend. Die erste Version der Implementierung setzte auf eine vollständige Einbettung für alle Variablentypen der Datei, sodass die Kennzeichnung, obwohl selten gebraucht, immer angefügt wurde. Jedoch brachten die Axiome für die choice- und if-then-else-Operatoren die Beweiser in einigen Fällen an ihre Grenzen. Erst als die Anzahl der Operatordefinitionen auf die tatsächlich notwendigen beschränkt wurde, konnte über eine relevante Menge von Problemen eine Aussage getroffen werden. Die Korrektheit der Einbettung aller axiomatisierter Operatoren konnte jedoch verifiziert werden, die Konsistenz des folgenden Theorems lässt sich problemlos nachweisen:

```
thf(freelogic_existence_type, type, ( eE: ( $i > $o ) ) ).

thf(freelogic_nonemptyE_axiom, axiom, ( ? [X: $i] : ( eE @ X ) ) ).

thf(freelogic_star_type, type, ( star: $i ) ).

thf(freelogic_star_axiom, axiom, ( ~ ( eE @ star ) ) ).

thf(the, type, ( the : ( $i > $o ) > $i ) ).

thf(the, axiom,
    ( ! [P: $i > $o, A: $i] :
        ( ( P @ A ) => ( ( ! [X: $i] :
                             ( ( P @ X ) => ( X = A ) ) )
                             => ( P @ ( the @ P ) ) ) ) )
).

thf(conj, conjecture,
    ( ( ( the @ ( ^ [X : $i] : ( X = star ) ) ) = star ) )
).
```

Ähnlich einfache Probleme für den axiomatisierten if-then-else-Operator und die Einbettung der Kennzeichnung konnten sowohl von Leo-II als auch von dem ebenfalls höherstufigen Beweiser Satallax (vgl. Brown 2012) als gültig erkannt werden. Zum aktuellen Zeitpunkt lässt sich nur vermuten, dass die native Unterstützung der Operatoren `$ite` und `@+` aufgrund des Verzichts auf Axiome eine bessere Erfüllbarkeitsprüfung im Allgemeinen garantieren wird, da kein höherstufiger Beweiser bekannt ist, der einen der beiden oder gar beide Operatoren

verarbeiten kann. Aus diesem Grund bleiben auch die Funktionstests für diese Einbettungsalternativen aus.

Nachdem die Einbettung auf das Notwendigste beschränkt wurde, konnte die überwiegende Anzahl der getesteten Probleme bewiesen werden und die Implementierung einem ersten Kompletttest unterzogen werden. Für den Test wurden 401 in Bezug auf ihre Schwierigkeit niedrig bewertete Theoreme aus der TPTP selektiert und als Probleme der freien Logik interpretiert[9], sodass sie nach deren Übersetzung an Satallax in der Version 2.7 weitergegeben werden konnten. Es wurde eine nicht-leere Domäne E_α vorausgesetzt und auf die Einbettung der Kennzeichnung wurde verzichtet. Von 401 getesteten Problemen konnten nur 158 als Theorem eingeordnet werden und 151 haben die interne Zeitbeschränkung von 200 Sekunden überschritten, sodass ihre Bearbeitung abgebrochen wurde. Für die restlichen Probleme konnte ein Gegenbeispiel gefunden werden.[10] Die Ursache der Zeitüberschreitung in den über 100 Fällen ist unbekannt. Die naive Adaption von THF zu HFF könnte die Schwierigkeit der Probleme erhoben haben oder die ausgewählten Theoreme sind schlicht nicht repräsentativ genug für Probleme der freien Logik, wodurch eine Auswertung erschwert wäre. Trotzdem können versuchsweise Optimierungen (vgl. Kapitel 8) angedacht werden.

Auch wenn die Performance aufgrund der Trivialität der Übersetzung in diesem Stadium

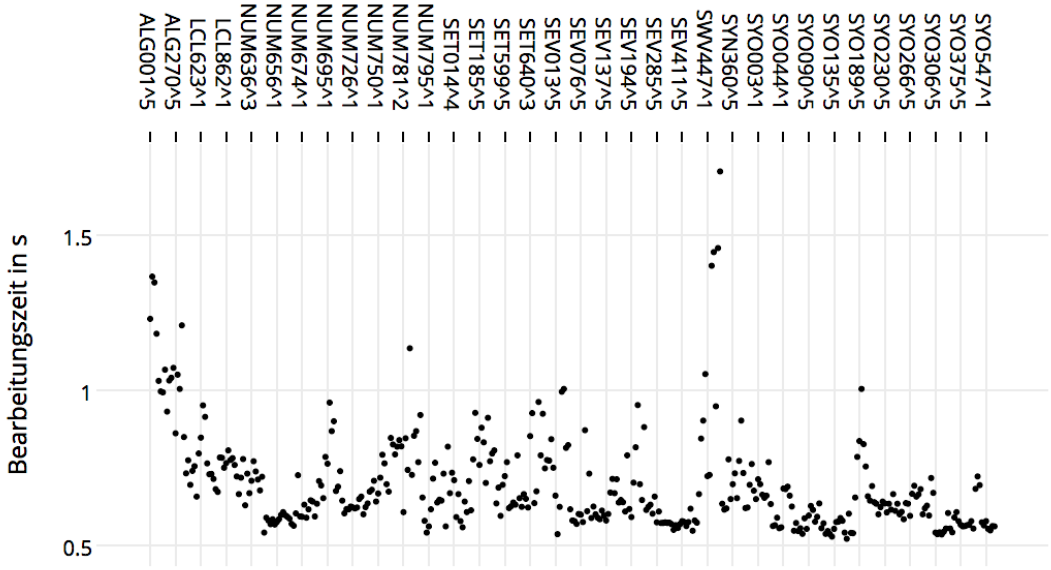

Abb. 6: Auswertung der Performance der Übersetzung

eine untergeordnete Rolle spielt, soll trotzdem eine kurze Evaluation erfolgen. Abbildung 6 gibt ein Gesamtbild der Performance aller 401 im Test übersetzter Probleme wieder. An der

[9] Die Theoreme werden als wahrheitserhaltend angenommen, mindestens aber für entscheidbar gehalten.

[10] Tests mit einer möglicherweise leeren Domäne E_α führten zu einem ähnlichen Ergebnis, von den 401 Problemen wurden 139 als Theoreme erkannt und 159 überschritten das Zeitlimit.

horizontalen Achse des Streudiagramms wurden beispielhaft einige der übersetzten Probleme aufgelistet, die vollständige Liste der Bearbeitungszeit aller 401 Probleme befindet sich im Projektordner der Implementierung. Die Übersetzung fügt im Durchschnitt 0,69 Sekunden zu der Gesamtbearbeitungszeit des Problems hinzu. Die Gesamtdauer der Übersetzung aller 401 Probleme betrug 278,59 Sekunden. Die längste Bearbeitungszeit hatte das Problem SWW478^3.p mit 1,7 Sekunden[11], die kürzeste Problem SYO149^5.p mit 0,52 Sekunden. Alle Tests wurden auf einem 13' Retina MacBook Pro aus dem Jahr 2015 und einer Broadwell-CPU (i5-5257U) mit einer Frequenz von 2,7GHz unter Oracles Java 8 (1.8.0) und EPFLs Scala 2.11.7 ausgeführt.

[11] Die relativ lange Bearbeitungszeit lässt sich durch die überdurchschnittlich hohe Zeilenanzahl von 8998 Zeilen in der Datei erklären.

8 Zusammenfassung und Ausblick

Beginnend mit der formalen Ausarbeitung einer höherstufigen freien Logik wurde deren Einbettung in Logik höherer Stufe wiedergegeben. Für die maschinenlesbare Kodierung beider Logiken wurde auf die bewährte Standardisierung des TPTP-Projekts zurückgegriffen. THF ist eine TPTP-Sprache, die sich stark am einfach typisierten λ-Kalkül orientiert und damit die Logik höherer Stufe wiederspiegelt, ein Standard, der im Rahmen dieser Arbeit in soweit ergänzt wurde, dass er für die freie Logik angewendet werden kann. Diese Neuinterpretation von THF, genannt HFF, richtet sich dabei an gängigen philosophischen Normen aus, um einen möglichst intuitiven Umgang damit zu gewährleisten. Zudem wurde eine Erweiterung der TPI-Sprache angeregt, um die freie Logik vollständig in das TPTP-Projekt integrieren zu können. Die kodierte Einbettung ist Grundlage für eine automatisierte, kommandozeilengesteuerte Übersetzung, bei der die festgelegten, freie Logik-spezifischen Sprachkonstrukte durch äquivalente THF-Prädikate ausgetauscht werden. Diese werden durch vorausgehende THF-Definitionen ergänzt, sodass im Gesamten eine auf ihre Konsistenz überprüfbare Datei, die komplett in der Logik höherer Stufe interpretiert werden kann, entsteht und von höherstufigen Theorembeweisern verarbeitet werden kann, was im Rahmen von diversen Tests auch realisiert wurde. Das Ziel, die Automatisierung von freier Logik unter Zuhilfenahme von höherstufigen Theorembeweisern zu erwirken, wurde mittels Implementierung dieser Übersetzung erreicht. Anhand einfacher Beispiele, von denen sich eines davon mit Ein- und Ausgabedatei im Anhang befindet, und kategorientheoretischer Formalisierungen konnte ihre Effektivität nachgewiesen werden: Die übersetzten Probleme der freien Logik konnten von verschiedenen Theorembeweisern validiert werden.

Die erweiterten Funktionstests zusammen mit den kategorientheoretischen Formalisierungen, evaluiert in Kapitel 7, zeigen, dass die Übersetzung in dem Umfang, wie sie aktuell vorgenommen wird, bereits nahezu uneingeschränkt zielführend ist. Nur ein paar der fortgeschrittenen kathegorientheoretischen Theoreme sowie einige TPTP-Probleme blieben unentscheidbar. Um letztendlich auch diese auswerten zu können, werden im Folgenden mögliche Lösungsansätzen für eine (noch) effektivere Übersetzung umrissen.

Verstärktes Setzen auf Polymorphismus von Seiten der Theorembeweiser stellt eine erste Möglichkeit dar. Kaliszyk, Sutcliffe und Rabe stellten 2016 das Format TH1 vor, eine Variante von THF, die polymorphe Typdefinitionen unterstützt. Eine solche Definition sieht wie folgt aus:

```
thf(freelogic_star_type, type, ( star: !> [A: $tType] : A > $o ) ).
```

`$star @ $i` könnte in diesem Fall direkt zu `star @ $i` und `$star @ $i > $o` zu `star @ $i > $o` übersetzt werden, was die Einbettung deutlich vereinfacht und verkürzt. Ob effizienteres Typhandling oder der Einsatz von `@+` und `$ite` als eingebaute Operatoren zu einer Verbesserung der Resultate der zuvor beschriebenen Tests führt, muss überprüft werden. Da in naher Zukunft die Umsetzung von polymorphen Typen für das Leo-III-Projekt geplant ist können diese Tests in Kürze nachgeliefert werden.

Eine weitere Alternative wäre die Übersetzung mit einem Optimierungsschritt auszustatten. Die interaktive Beweisumgebung Isabelle/HOL übersetzt Probleme der Logik höherer

Stufe aus einer eigenen domänenspezifischen Sprache in THF und reicht diese so an automatische Theorembeweiser weiter. Bei dieser Übersetzung werden die Benutzereingaben stark optimiert, ein Schritt, der bei einer einfachen Übersetzung wie der in dieser Arbeit implementierten fehlt. Das formalisierte Lemma 1.13. aus Kapitel 6.1 wird von Isabelle/HOL wie folgt umgestellt und erweitert:

```
thf(lem1_13_1_type, type, X : $i ).

thf(lem1_13_1, conjecture,
    ( ( ( ( ~ ( ( eE_1 @ ( source @ ( source @ X ) ) ) ) )
          & ( ~ ( ( eE_1 @ ( source @ X ) ) ) ) )
        | ( ( eE_1 @ ( source @ ( source @ X ) ) )
          & ( ~ ( ( ( eE_1 @ ( source @ X ) )
                    => ( ~ ( ( ( source @ ( source @ X ) )
                               = ( source @ X ) ) ) ) ) ) ) ) )
).
```

Die Umsetzung einer solchen Übersetzung bringt in der Praxis aber einige Hindernisse mit sich, weswegen der Nutzen hier genau abgewogen werden muss. Optimierungen sind für sich nicht determiniert und unterscheiden sich je nach angewendetem Theorembeweiser. Ein Optimierungsschritt würde die Übersetzung im Wesentlichen verkomplizieren und eine umfassende Forschungsarbeit nach sich ziehen, sofern man sich nicht nur auf einen Theorembeweiser festlegen möchte.

Ein letzter Ansatz, der allerdings der konventionellen Idee einer Einbettung widerspricht, wäre, dass statt einer Einbettung der Quantifizierungsoperatoren im klassischen Sinn der Wächter direkt in die einzelnen Formeln gesetzt wird, sodass folgende Übersetzung angedacht werden könnte:

```
hff(lem, conjecture,
    ( ! [X: $i, Y: $i] : ( ( $e @ $i @ X ) & ( $e @ $i @ Y ) ) )
).
```

$\equiv$

```
thf(lem, conjecture,
    ( ! [X: $i, Y: $i] :
        ( ( eE_1 @ X )
          => ( ( eE_1 @ Y )
                => ( ( eE_1 @ X ) & ( eE_1 @ Y ) ) ) ) )
).
```

Solche Umformungen werden auch von der Isabelle/HOL-Umgebung im Rahmen ihrer internen Optimierungsmaßnahmen vorgenommen (siehe vorheriges Beispiel). Sollte die Einführung eigener Definitionen für grundlegende Operationen wie Quantifizierungen das

Problem der Nichtentscheidbarkeit begünstigen, dann wäre dies ein Ansatz, mit dem man diesem entgegenwirken und es näher betrachten könnte.

Abgesehen von erwägenswerten Verbesserungen an der Übersetzung per se gibt es noch weitere offene Fragestellungen, die das Gesamtprojekt betreffen. Die in dieser Arbeit vorgestellte Einbettung bildet nur die positive, eventuell inklusive freie Logik ab. Die Einbettung einer negativen und neutralen freien Logik steht noch aus. Insbesondere für die zweite der beiden Semantikvarianten fehlen entscheidende theoretische Grundlagen, sodass hier eine ausführliche Recherche vorausgehen muss. Letztendlich kann sich der Einbettung in Kombination mit mehrwertigen Logiken oder mittels einem $*_o$ angenähert werden. Zudem bildet, wie bereits in Kapitel 3.3.3 angedeutet, die freie Logik einen Grundbaustein für verschiedenste Logiken mit Kripke-Semantik, sodass deren Integration und Zusammenspiel mit der hier gezeigten Einbettung der freien Logik Beachtung finden sollte.

Für erste Tests ist eine kommandogesteuerte Übersetzung unabdingbar. Jedoch ist aus benutzerfreundlicher Sicht ein in sich geschlossener Prozess dem Erzeugen einer Zwischendatei vorzuziehen. Auf lange Sicht ist es wünschenswert, dass die Übersetzung auch intern in Leo-III angestoßen werden kann. In diesem Kontext sollte auch evaluiert werden, ob eine Erweiterung der der Übersetzung zugrunde liegenden Grammatik notwendig ist. Die bisher verwendete Grammatik unterstützt beispielsweise annotierte TPTP-Anweisungen nicht, aber eine in sich vollständige Syntaxanalyse im Übersetzungsschritt stellt einen zusätzlichen Rechenaufwand dar, der so nicht benötigt wird. Eine Lösung, die die Kombination einer Übersetzung mit der tatsächlichen internen Syntaxanalyse vorsieht, ist als weitaus effizienter anzusehen.

Zuletzt ist die Weiterentwicklung der Schnittstelle zur freien Logik stark abhängig von der zukünftigen offiziellen Unterstützung. Diverse übersetzte Probleme der freien Logik sollten Teil des Problemkatalogs, der obligatorisch für den CASC ist, werden, um einen Anreiz für höherstufige Theorembeweiser zu schaffen, diese lösen zu können, und die Unterstützung aller Einbettungsoperatoren voranzutreiben. Erst dann können vollständige Funktionstests durchgeführt werden und zuletzt auch hochkomplexe Probleme der freien Logik erfolgreich entschieden werden.

Literatur

Backes, Julian (2010). „Tableaux for Higher-Order Logic with If-Then-Else, Description and Choice“. Masterarbeit. Universität des Saarlandes.

Backes, Julian und Brown, Chad E. (2011). Analytic Tableaux for Higher-Order Logic with Choice. In: *Journal of Automated Reasoning* 47.4, S. 451–479. ISSN: 1573-0670.

Béguet, Eric und Jonnalagedda, Manohar (2014). „Accelerating Parser Combinators with Macros“. In: *Proceedings of the Fifth Annual Scala Workshop*. SCALA '14. Uppsala, Sweden: ACM, S. 7–17. URL: http://doi.acm.org/10.1145/2637647.2637653.

Bencivenga, Ermanno (1986). „Free Logics“. In: *Handbook of Philosophical Logic: Volume III: Alternatives in Classical Logic*. Hrsg. von Dov M. Gabbay und Franz Guenthner. Dordrecht: Springer Netherlands, S. 373–426.

Benzmüller, Christoph (2013). „A Top-down Approach to Combining Logics“. In: *Proc. of the 5th International Conference on Agents and Artificial Intelligence (ICAART)*. Hrsg. von Joaquim Filipe und Ana Fred. Bd. 1. Barcelona, Spain: SCITEPRESS – Science und Technology Publications, Lda, S. 346–351. URL: http://christoph-benzmueller.de/papers/C35.pdf.

– (2015). „Higher-Order Automated Theorem Provers“. In: *All about Proofs, Proof for All*. Hrsg. von David Delahaye und Bruno Woltzenlogel Paleo. Mathematical Logic and Foundations. London: College Publications, S. 171–214. URL: http://christoph-benzmueller.de/papers/B14.pdf.

Benzmüller, Christoph, Brown, Chad E. und Kohlhase, Michael (2004). Higher-Order Semantics and Extensionality. In: *Journal of Symbolic Logic* 69.4, S. 1027–1088. URL: http://christoph-benzmueller.de/papers/J6.pdf.

Benzmüller, Christoph und Miller, Dale (2014). „Automation of Higher-Order Logic“. In: *Handbook of the History of Logic, Volume 9 — Computational Logic*. Hrsg. von Dov M. Gabbay, Jörg H. Siekmann und John Woods. North Holland, Elsevier, S. 215–254. URL: http://christoph-benzmueller.de/papers/B5.pdf.

Benzmüller, Christoph und Scott, Dana (2016a). „Automating Free Logic in Isabelle/HOL“. In: *Mathematical Software – ICMS 2016, 5th International Congress, Proceedings*. Hrsg. von Gert-Martin Greuel, Thorsten Koch, Peter Paule und Andrew Sommese. Bd. 9725. LNCS. Berlin: Springer, S. 43–50. URL: http://christoph-benzmueller.de/papers/C57.pdf.

– (2016b). *Axiomatizing Category Theory in Free Logic*. arXiv: 1609.01493v3 [cs.LO].

Benzmüller, Christoph, Theiss, Frank, Paulson, Lawrence und Fietzke, Arnaud (2008). „LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic (System Description)“. In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. Hrsg. von Alessandro Armando, Peter Baumgartner und Gilles Dowek. Bd. 5195. LNCS. Berlin: Springer, S. 162–170. URL: http://christoph-benzmueller.de/papers/C26.pdf.

Benzmüller, Christoph und Woltzenlogel Paleo, Bruno (2015a). „Experiments in Computational Metaphysics: Gödel's Proof of God's Existence“. In: *Science & Spiritual Quest, Proceedings of the 9th All India Students' Conference, 30th October – 1 November, 2015, IIT Kharagpur, India*. Hrsg. von Subhash C. Mishramand Ramgopal Uppaluri und Varun

Agarwal. Bhaktivedanta Institute, Kolkata, www.binstitute.org, S. 23–40. URL: http://christoph-benzmueller.de/papers/C52.pdf.

Benzmüller, Christoph und Woltzenlogel Paleo, Bruno (2015b). „Higher-Order Modal Logics: Automation and Applications". In: *Reasoning Web 2015*. Hrsg. von Adrian Paschke und Wolfgang Faber. LNCS 9203. Berlin, Germany: Springer, S. 32–74. URL: http://christoph-benzmueller.de/papers/C46.pdf.

– (2016). „The Inconsistency in Gödel's Ontological Argument: A Success Story for AI in Metaphysics". In: *IJCAI 2016*. Hrsg. von Subbarao Kambhampati. Bd. 1–3. AAAI Press, S. 936–942. ISBN: 978-1-57735-770-4. URL: http://www.ijcai.org/Proceedings/16/Papers/137.pdf.

Brown, Chad E. (2012). „Satallax: An Automatic Higher-Order Prover". In: *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. Hrsg. von Bernhard Gramlich, Dale Miller und Uli Sattler. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 111–117.

Burge, Tyler (1974). Truth and Singular Terms. In: *Noûs* 8.4, S. 309–325.

Church, Alonzo (1940). A Formulation of the Simple Theory of Types. In: *Journal of Symbolic Logic* 5.2, S. 56–68.

Doenitz, Mathias und Myltsev, Alexander (2013). *Parboiled2: A macro-based PEG Parser Generator for Scala 2.10.3+*. URL: https://github.com/sirthias/parboiled2 (besucht am 28. 10. 2016).

Farmer, William M. (2008). The Seven Virtues of Simple Type Theory. In: *Journal of Applied Logic* 6.3, S. 267–286. URL: http://dblp.uni-trier.de/db/journals/japll/japll6.html#Farmer08.

Freyd, Peter J. und Scedrov, Andre (1990). *Categories, Allegories*. Bd. 39. Mathematical Library. North Holland.

Gabbay, Dov M. (1996). *Labelled Deductive Systems*. Bd. 1. Oxford Logic Guides. Oxford: Clarendon Press. URL: http://opac.inria.fr/record=b1092542.

Garson, James W. (1991). „Applications of Free Logic to Quantified Intensional Logic". In: *Philosophical Application of Free Logic*. Hrsg. von Karel Lambert. Oxford University Press, S. 111–142.

Henkin, Leon (1950). Completeness in the Theory of Types. In: *J. Symbolic Logic* 15.2, S. 81–91.

Kaliszyk, Cezary, Sutcliffe, Geoff und Rabe, Florian (2016). „TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism". In: *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR 2016), Coimbra, Portugal, July 2nd, 2016*. Hrsg. von Pascal Fontaine, Stephan Schulz und Josef Urban. Bd. 1635. CEUR Workshop Proceedings. CEUR-WS.org, S. 41–55. URL: http://ceur-ws.org/Vol-1635/paper-05.pdf.

Kripke, Saul A. (1963). Semantical Considerations on Modal Logic. In: *Acta Phil. Fennica* 16, S. 83–94.

Kurš, Jan, Vraný, Jan, Ghafari, Mohammad, Lungu, Mircea und Nierstrasz, Oscar (2016). „Optimizing Parser Combinators". In: *Proceedings of International Workshop on Smalltalk Technologies (IWST 2016)*. Im Erscheinen. URL: http://scg.unibe.ch/archive/papers/Kurs16a-Compiler.pdf.

Lambert, Karel (1960). The Definition of E! in Free Logic. In: *Abstracts: The International Congress for Logic, Methodology and Philosophy of Science*.
– Hrsg. (1991). *Philosophical Application of Free Logic*. Oxford University Press.
– (2001). „Free Logics". In: *The Blackwell Guide to Philosophical Logic*. Hrsg. von Lou Goble. Blackwell Publishers, S. 258–279.
Lehmann, Scott (2001). „‚No Input, No Output' Logic". In: *New Essays in Free Logic: In Honour of Karel Lambert*. Hrsg. von Edgar Morscher und Alexander Hieke. Dordrecht: Springer Netherlands, S. 147–155.
Meinong, Alexius (1904). *Über Gegenstandstheorie*. Leipzig: Barth, S. 1–50.
Morscher, Edgar und Simons, Peter (2001). „Free Logic: A Fifty-Year Past and an Open Future". In: *New Essays in Free Logic: In Honour of Karel Lambert*. Hrsg. von Edgar Morscher und Alexander Hieke. Dordrecht: Springer Netherlands, S. 1–34.
Nipkow, Tobias, Paulson, Lawrence C. und Wenzel, Markus (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Bd. 2283. LNCS. Springer.
Nolt, John (2002). „Free Logics". In: *Philosophy of Logic*. Hrsg. von Dale Jacquette. Handbook of the Philosophy of Science. North-Holland, S. 1023–1060.
– (2014). „Free Logic". In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Winter 2014.
Paśniczek, Jacek (2001). „Can Meinongian Logic be Free?" In: *New Essays in Free Logic: In Honour of Karel Lambert*. Hrsg. von Edgar Morscher und Alexander Hieke. Dordrecht: Springer Netherlands, S. 227–236.
Quine, Willard V. (1954). Quantification and the Empty Domain. In: *Journal of Symbolic Logic* 19.3, S. 177–179.
Riazanov, Alexandre und Andrei, Voronkov (1999). „Vampire". In: *Automated Deduction — CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 292–296.
Russell, Bertrand (1920). *Introduction to Mathematical Philosophy*. Georg Allen und Unwin.
Schweizer, Paul (2015). „Negative Existentials and Non-denoting Terms". In: *Logic and Its Applications: 6th Indian Conference, ICLA 2015, Mumbai, India, January 8-10, 2015. Proceedings*. Hrsg. von Mohua Banerjee und Shankara Narayanan Krishna. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 183–194.
Scott, Dana (1979). „Identity and Existence in Intuitionistic Logic". In: *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*. Hrsg. von Michael Fourman, Christopher Mulvey und Dana Scott. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 660–696.
– (1991). „Existence and Description in Formal Logic". In: *Philosophical Application of Free Logic*. Hrsg. von Karel Lambert. Oxford University Press, S. 28–48.
Sutcliffe, Geoff (2009). The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. In: *Journal of Automated Reasoning* 43.4, S. 337–362.
Sutcliffe, Geoff und Benzmüller, Christoph (2010). Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. In: *Journal of Formalized Reasoning* 3.1, S. 1–27. ISSN: 1972-5787. URL: http://christoph-benzmueller.de/papers/J22.pdf.
van Fraassen, Bas C. (1966). „Singular Terms, Truth-Value Gaps, and Free Logic". In: Bd. 63. 17. Journal of Philosophy Inc, S. 481–495.

Wisniewski, Max, Steen, Alexander und Benzmüller, Christoph (2014). „The Leo-III Project". In: *Joint Automated Reasoning Workshop and Deduktionstreffen*. Hrsg. von Alexander Bolotov und Manfred Kerber, S. 38. URL: http://christoph-benzmueller.de/papers/W53.pdf.

– (2015). „LeoPARD - A Generic Platform for the Implementation of Higher-Order Reasoners". In: *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*. Hrsg. von Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe und Volker Sorge. Bd. 9150, S. 325–330. URL: http://christoph-benzmueller.de/papers/C45.pdf.

– (2016). „TPTP and Beyond: Representation of Quantified Non-Classical Logics". In: *ARQNL 2016. Automated Reasoning in Quantified Non-Classical Logics*. Hrsg. von Christoph Benzmüller und Jens Otten. Im Erscheinen. Coimbra, Portugal.

Abbildungsverzeichnis

A Einbettung von freier Logik in THF

A.1 Einbettung mit choice-Operator (@+) und if-then-else Operator ($ite)

```
%----Declaration of the existence predicate

thf(freelogic_existence_type, type, ( eE: ( $i > $o ) ) ).

%----Axiom for a non-empty domain E

thf(freelogic_nonemptyE_axiom, axiom, ( ? [X: $i] : ( eE @ X ) ) ).

%----Declaration of the star

thf(freelogic_star_type, type, ( star: $i ) ).

thf(freelogic_star_axiom, axiom, ( ~ ( eE @ star ) ) ).

%----Definition of free logic universal quantification

thf(freelogic_forall_type, type, ( fforall: ( ( $i > $o ) > $o ) ) ).

thf(freelogic_forall, definition,
    ( fforall =
      ( ^ [Phi: $i > $o] : ! [X: $i] : ( ( eE @ X ) => ( Phi @ X ) ) ) )
).

%----Definition of free logic existential quantification

thf(freelogic_exists_type, type, ( fexists: ( ( $i > $o) > $o ) ) ).

thf(freelogic_exists, definition,
    ( fexists =
      ( ^ [Phi: $i > $o] : ( ~ ( fforall @ ^ [X: $i] : ( ~ ( Phi @ X ) ) ) ) ) )
).

%----Definition of free logic description

thf(ffthat_type, type, ( i: ( $i > $o ) > $i ) ).

thf(ffthat, definition,
    ( i =
      ( ^ [Phi: $i > $o] :
          ( $ite
            @ ( ? [X: $i] :
```

```
                ( ( eE @ X )
                  & ( Phi @ X )
                  & ( ! [Y: $i] :
                        ( ( ( eE @ Y ) & ( Phi @ Y ) )
                          => ( Y = X ) ) ) ) )
            @ ( @+ [X: $i] :
                  ( ( eE @ X ) & ( Phi @ X ) ) )
            @ star ) ) )
).
```

A.2 Einbettung mit Axiomen für choice- und if-then-else-Operatoren

```
%----Declaration of the existence predicate

thf(freelogic_existence_type, type, ( eE: ( $i > $o ) ) ).

%----Axiom for a non-empty domain E

thf(freelogic_nonemptyE_axiom, axiom, ( ? [X: $i] : ( eE @ X ) ) ).

%----Declaration of the star

thf(freelogic_star_type, type, ( star: $i ) ).

thf(freelogic_star_axiom, axiom, ( ~ ( eE @ star ) ) ).

%----Definition of free logic universal quantification

thf(freelogic_forall_type, type, ( fforall: ( ( $i > $o ) > $o ) ) ).

thf(freelogic_forall, definition,
    ( fforall =
      ( ^ [Phi: $i > $o] : ! [X: $i] : ( ( eE @ X ) => ( Phi @ X ) ) ) )
).

%----Definition of free logic existential quantification

thf(freelogic_exists_type, type, ( fexists: ( ( $i > $o) > $o ) ) ).

thf(freelogic_exists, definition,
    ( fexists =
      ( ^ [Phi: $i > $o] : ( ~ ( fforall @ ^ [X: $i] : ( ~ ( Phi @ X ) ) ) ) ) )
).

%----Definition of free logic description

thf(the_type,type,( the: ( $i > $o ) > $i ) ).

thf(the, axiom,
    ( ! [P: $i > $o, A: $i] :
         ( ( P @ A ) => ( ( ! [X: $i] :
                              ( ( P @ X ) => ( X = A ) ) )
                          => ( P @ ( the @ P ) ) ) ) )
).
```

```
thf(if_type, type, ( if: $o > $i > $i > $i ) ).

thf(if, axiom,
    ( if = ( ^ [P: $o] :
      ( ^ [X: $i] :
          ( ^ [Y: $i] :
              ( the @ ( ^ [Z: $i] :
                          ( ( ( ( P ) => ( Z = X ) ) )
                            & ( ( ~ ( P ) ) => ( Z = Y ) ) ) ) ) ) ) ) ) )
).

thf(if_ax1, axiom, ( ! [P: $o] : ( ( P = $true ) | ( P = $false ) ) ) ).

thf(if_ax2_1, axiom, ( ! [X: $i, Y: $i] : ( ( if @ $false @ X @ Y ) = Y ) ) ).

thf(if_ax2_2, axiom, ( ! [X: $i, Y: $i] : ( ( if @ $true @ X @ Y ) = X ) ) ).

thf(freelogic_fthat_type, type, ( i: ( $i > $o ) > $i ) ).

thf(freelogic_fthat, definition,
    ( i =
      ( ^ [Phi: $i > $o] :
          ( if
            @ ( ? [X: $i] :
                  ( ( eE @ X )
                    & ( Phi @ X )
                    & ( ! [Y: $i] :
                          ( ( ( eE @ Y ) & ( Phi @ Y ) )
                            => ( Y = X ) ) ) ) )
            @ ( the @ ( ^ [X: $i] :
                          ( ( eE @ X ) & ( Phi @ X ) ) ) )
            @ star ) ) )
).
```

B Exemplarische Ein- und erwartete Ausgabedatei

Eingabedatei

```
hff(r_const, type, ( r: ( $i > $i > $o ) ) ).

hff(lem, conjecture,
    ( ( ( ( !+ [X: $i] :
                ( ( r @ X @ X ) => ( r @ X @ X ) ) )
          & ( ? [Y: $i] : ( Y = Y ) ) )
        => ( ? [Y: $i] : ( ( r @ Y @ Y ) => ( r @ Y @ Y ) ) ) ) )
)).
```

Ausgabedatei

```
%------------------------------------------------------------------------------
% Semantic embedding of higher-order free logic into higher-order logic
%------------------------------------------------------------------------------

%----Declaration of the existence predicate for type $i

thf(freelogic_existence_type_1, type, ( eE_1: ( $i > $o ) ) ).

%----Axiom for a non-empty domain E for type $i

thf(freelogic_nonemptyE_axiom_1, axiom, ( ? [X: $i] : ( eE_1 @ X ) ) ).

%----Declaration of the star for type $i

thf(freelogic_star_type_1, type, ( star_1: $i ) ).

thf(freelogic_star_axiom_1, axiom, ( ~ ( eE_1 @ star_1 ) ) ).

%----Definition of free logic universal quantification for type $i

thf(freelogic_forall_type_1, type, ( fforall_1: ( ( $i > $o ) > $o ) ) ).

thf(freelogic_forall_1, definition,
    ( fforall_1 =
      ( ^ [Phi: $i > $o] :
          ! [X: $i] : ( ( eE_1 @ X ) => ( Phi @ X ) ) ) )
).
```

```
%----Definition of free logic existential quantification for type $i

thf(freelogic_exists_type_1, type, ( fexists_1: ( ( $i > $o) > $o ) ) ).

thf(freelogic_exists_1, definition,
    ( fexists_1 =
      ( ^ [Phi: $i > $o] :
          ( ~ ( fforall_1 @ ^ [X: $i] : ( ~ ( Phi @ X ) ) ) ) ) )
).

%----------------------------------------------------------------------------

thf(r_const, type,( r: ( $i > $i > $o ) ) ).

thf(lem, conjecture,
    ( ( ( ( ! [X: $i] :
                ( ( r @ X @ X ) => ( r @ X @ X ) ) )
          & ( ( fexists_1 @ ^ [Y: $i] :
                              ( ( Y = Y ) ) ) ) )
        => ( ( fexists_1 @ ^ [Y: $i] :
                            ( ( ( r @ Y @ Y ) => ( r @ Y @ Y ) ) ) ) ) ) )
).
```

C Eingabedatei für die kategorientheoretische Annahme 1.18.

```
%----------------------------------------------------------------------------
% Formalization (Categories, Allegories by Freyd and Scedrov)
%----------------------------------------------------------------------------

hff(a_type, type, ( a: $i ) ).

hff(b_type, type, ( b: $i ) ).

%----Definition of a weak, non-reflexive identity on the set of existing objects

hff(eq1_a_type, type, ( eq1_a: a > a > $o ) ).

hff(eq1_a, definition,
    ( eq1_a =
      ( ^ [X: a, Y: a] : ( ( $e @ a @ X ) & ( $e @ a @ Y ) & ( X = Y ) ) ) )
).

hff(eq1_b_type, type, ( eq1_b: b > b > $o ) ).

hff(eq1_b, definition,
    ( eq1_b =
      ( ^ [X: b, Y: b] : ( ( $e @ b @ X ) & ( $e @ b @ Y ) & ( X = Y ) ) ) )
).

%----Definition of Kleene equality

hff(eq2_a_type, type, ( eq2_a: a > a > $o ) ).

hff(eq2_a, definition,
    ( eq2_a =
      ( ^ [X: a, Y: a] :
          ( ( ( $e @ a @ X ) | ( $e @ a @ Y )  ) => ( eq1_a @ X @ Y ) ) ) )
).

hff(eq2_b_type, type, ( eq2_b: b > b > $o ) ).

hff(eq2_b, definition,
    ( eq2_b =
      ( ^ [X: b, Y: b] :
          ( ( ( $e @ b @ X ) | ( $e @ b @ Y )  ) => ( eq1_b @ X @ Y ) ) ) )
).
```

```
%----Definition of directed Kleene equality

hff(eq3_a_type, type, ( eq3_a: a > a > $o ) ).

hff(eq3_a, definition,
    ( eq3_a = ( ^ [X: a, Y: a] : ( ( $e @ a @ X ) => ( eq1_a @ X @ Y ) ) ) )
).

hff(eq3_b_type, type, ( eq3_b: b > b > $o ) ).

hff(eq3_b, definition,
    ( eq3_b = ( ^ [X: b, Y: b] : ( ( $e @ b @ X ) => ( eq1_b @ X @ Y ) ) ) )
).

%----Definitions of basic notions for category theory

hff(source_a_type, type, ( source_a: a > a ) ).

hff(target_a_type, type, ( target_a: a > a ) ).

hff(composition_a_type, type, ( comp_a: a > a > a ) ).

hff(source_b_type, type, ( source_b: b > b ) ).

hff(target_b_type, type, ( target_b: b > b ) ).

hff(composition_b_type, type, ( comp_b: b > b > b ) ).

%----Scott's axiom system

hff(scott_s1_axiom_a, axiom,
    ( !+ [X: a] : ( ( $e @ a @ ( source_a @ X ) ) => ( $e @ a @ X ) ) )
).

hff(scott_s2_axiom_a, axiom,
    ( !+ [X: a] : ( ( $e @ a @ ( target_a @ X ) ) => ( $e @ a @ X ) ) )
).

hff(scott_s3_axiom_a, axiom,
    ( !+ [X: a] :
         ( !+ [Y: a] :
              ( ( $e @ a @ ( comp_a @ X @ Y ) )
                <=> ( eq1_a @ ( source_a @ X ) @ ( target_a @ Y ) ) ) ) )
).
```

```
hff(scott_s4_axiom_a, axiom,
    ( !+ [X: a] :
         ( !+ [Y: a] :
              ( !+ [Z: a] :
                   ( eq2_a
                     @ ( comp_a @ X @ ( comp_a @ Y @ Z ) )
                     @ ( comp_a @ ( comp_a @ X @ Y ) @ Z ) ) ) ) )
).

hff(scott_s5_axiom_a, axiom,
    ( !+ [X: a] : ( eq2_a @ ( comp_a @ ( source_a @ X ) @ X ) @ X ) )
).

hff(scott_s6_axiom_a, axiom,
    ( !+ [X: a] : ( eq2_a @ ( comp_a @ X @ ( target_a @ X ) ) @ X ) )
).

hff(scott_s1_axiom_b, axiom,
    ( !+ [X: b] : ( ( $e @ b @ ( source_b @ X ) ) => ( $e @ b @ X ) ) )
).

hff(scott_s2_axiom_b, axiom,
    ( !+ [X: b] : ( ( $e @ b @ ( target_b @ X ) ) => ( $e @ b @ X ) ) )
).

hff(scott_s3_axiom_b, axiom,
    ( !+ [X: b] :
         ( !+ [Y: b] :
              ( ( $e @ b @ ( comp_b @ X @ Y ) )
                <=> ( eq1_b @ ( source_b @ X ) @ ( target_b @ Y ) ) ) ) )
).

hff(scott_s4_axiom_b, axiom,
    ( !+ [X: b] :
         ( !+ [Y: b] :
              ( !+ [Z: b] :
                   ( eq2_b
                     @ ( comp_b @ X @ ( comp_b @ Y @ Z ) )
                     @ ( comp_b @ ( comp_b @ X @ Y ) @ Z ) ) ) ) )
).

hff(scott_s5_axiom_b, axiom,
    ( !+ [X: b] : ( eq2_b @ ( comp_b @ ( source_b @ X ) @ X ) @ X ) )
).
```

```
hff(scott_s6_axiom_b, axiom,
    ( !+ [X: b] : ( eq2_b @ ( comp_b @ X @ ( target_b @ X ) ) @ X ) )
).

hff(functor_type, type, ( functor: ( a > b ) > $o ) ).

hff(functor, definition,
    ( functor =
      ( ^ [F: a > b] :
          ( ( ! [A: a] :
                ( ! [B: a] :
                    ( eq2_a @ ( source_a @ A ) @ B )
                    => ( eq2_b @ ( source_b @ ( F @ A ) ) @ ( F @ B ) ) ) )
            & ( ! [A: a] :
                  ( ! [B: a] :
                      ( eq2_a @ ( target_a @ A ) @ B )
                      => ( eq2_b @ ( target_b @ ( F @ A ) ) @ ( F @ B ) ) ) )
            & ( ! [A: a] :
                  ( ! [B: a] :
                      ( ! [C: a] :
                          ( eq2_a @ ( comp_a @ A @ B ) @ C )
                          => ( eq2_b
                               @ ( comp_b @ ( F @ A ) @ ( F @ B ) )
                               @ ( F @ C ) ) ) ) ) ) ) )
).

hff(functor_equivalence, conjecture,
    ( ! [F: a > b] :
        ( ( functor @ F )
          => ( ( ! [X: a] :
                   ( eq2_b
                     @ ( F @ ( source_a @ X ) )
                     @ ( source_b @ ( F @ X ) ) ) )
               & ( ! [X: a] :
                     ( eq2_b
                       @ ( F @ ( target_a @ X ) )
                       @ ( target_b @ ( F @ X ) ) ) )
               & ( ! [X: a] :
                     ( ! [Y: a] :
                         ( eq3_b
                           @ ( F @ ( comp_a @ X @ Y ) )
                           @ ( comp_b @ ( F @ X ) @ ( F @ Y ) ) ) ) ) ) ) )
).
```