Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

Implementierung eines Theorembeweisers für Dependence Logik

Andreas Bremer
Matrikelnummer 10004865

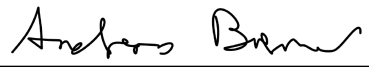
Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier
Betreuer: M. Sc. Timon Barlag

05.03.2021

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 4. März 2021



Andreas Bremer

Inhaltsverzeichnis

1 Einleitung

Die Dependence Logik ist eine Erweiterung der Prädikatenlogik erster Stufe, die es erlaubt, das Konzept der Abhängigkeit einer Variable von anderen Termen darzustellen. Um über solche Abhängigkeiten reden zu können, ist es nötig, eine Aussage über mehrere Belegungen zu machen. Dafür werden sogenannte "Teams", Mengen an Belegungen, verwendet. Diese können genutzt werden, um Aussagen darüber zu treffen, welche gemeinsamen Abhängigkeiten in diesen Teams auftreten. Zu diesem Zweck erweitert die Dependence Logik die Prädikatenlogik erster Stufe um ein neues Syntaxkonstrukt, das Dependence Atom. Durch dieses können gemeinsame Abhängigkeiten einer Variable in verschiedenen Belegungen ausgedrückt werden. Die Semantik dieser Dependence Logik wird in Kapitel 2 erklärt.

In [KV13] wurde ein Kalkül natürlichen Schließens für die Dependence Logik $\mathcal{D}$ aufgestellt, und gezeigt, dass für Formeln $T \cup \{\Psi\} \subset \mathcal{D}$ gilt, dass

$$T \vdash_{\mathcal{D}} \Psi \implies T \models \Psi.$$

Es ist theoretisch möglich, aus allen Sätzen der Dependence Logik ihre Folgerungen in der Prädikatenlogik erster Stufe mit diesem Kalkül herzuleiten. In dieser Arbeit wird untersucht, ob dies auch praktisch durchführbar ist. Es wird versucht, ein Programm zu entwickeln, welches mit begrenzter Rechenleistung und begrenztem Speicherplatz für ein solches gegebenes Problem automatisch einen Beweis herleiten kann. Dabei wird auf eine effiziente Implementierung Wert gelegt, die in endlicher Zeit eine passende Lösung erarbeiten soll.

Probierte und mögliche Ansätze werden hinsichtlich ihrer Merkmale und Voraussetzungen diskutiert. Die Umsetzung des gewählten Ansatzes wird beschrieben und es wird auf Schwierigkeiten bei dieser Herangehensweise sowie ihre Grenzen eingegangen.

Insbesondere ist dabei interessant, wie viele Informationen der Benutzer dem Theorembeweiser geben muss, um den gewünschten Beweis generieren zu können, sowie welche Formeln sich beweisen lassen, bis das verwendete Verfahren an seine Grenzen stößt.

2 Einführung in Dependence Logik

Die Dependence Logik ist eine Erweiterung der Prädikatenlogik erster Stufe. Belegungen werden zu Teams zusammengefügt, was es erlaubt, Beziehungen zwischen verschiedenen Belegungen darzustellen. Sie fügt sogenannte Dependence-Atome zur Syntax der Prädikatenlogik erster Stufe hinzu, durch die funktionale Abhängigkeiten einer Variable zu anderen Termen dargestellt werden können. Durch dependence logische Sätze kann dadurch mehr ausgedrückt werden, als durch die Prädikatenlogik erster Stufe möglich ist. Für eine ausführlichere Einführung in die Dependence Logik kann ein Standardwerk, wie [Abr+16], genutzt werden.

Die hier verwendeten Definitionen zur Dependence Logik basieren auf den Definitionen aus dem Paper von Juha Kontinen und Jouko Väänänen [KV13].

Definition 1

Sei $\mathfrak{A}$ ein Model mit Definitionsmenge A. Eine Belegung s von $\mathfrak{A}$ ist eine endliche Abbildung von Variablen nach A. Der Wert eines Terms t in s wird als $t^{\mathfrak{A}}\langle s\rangle$ geschrieben. Für ein Tupel $x = (x_1, ..., x_n)$ aus Variablen $x_1, ..., x_n$, ist $s(x) \coloneqq (s(x_1), ..., s(x_n))$.

Sei x eine Variable und $a \in A$, dann ist $s' = s(a/x)$ mit Definitionsmenge $Dom(s') = Dom(s) \cup \{x\}$ gleich wie s, nur dass $s'(x) = a$, also

$$s(a/x)(x_i) \coloneqq \begin{cases} a, & \text{falls } x_i = x \\ s(x_i), & \text{sonst.} \end{cases}$$

Sei $\{x_1, ..., x_k\}$ eine endliche Menge an Variablen, so ist ein Team X von A mit $Dom(X) = \{x_1, ..., x_k\}$ eine Menge an Belegungen von den Variablen $\{x_1, ..., x_k\}$ nach A.

2.1 Dependence Atome

Die Dependence Logik $\mathcal{D}$ erweitert die Prädikatenlogik erster Stufe **FO** syntaktisch durch Dependence Atome, dargestellt als

$$=(t_1, ..., t_n),$$

für Terme $t_1, ..., t_n$.

Hierbei ist $=(t_1, ..., t_n)$ so definiert, dass

$$\mathfrak{A} \models_X =(t_1, ..., t_n),$$

genau dann, wenn

$$\forall s, s' \in X : s(t_1, ..., t_{n-1}) = s'(t_1, ..., t_{n-1}) \implies {t_n}^{\mathfrak{A}}\langle s\rangle = {t_n}^{\mathfrak{A}}\langle s'\rangle.$$

Das heißt, in allen Belegungen mit gleichen $t_1, ..., t_{n-1}$ muss t_n den gleichen Wert haben.

Aus dieser Definition folgt für den Spezialfall mit $n = 0$, dass $=()$ immer wahr ist.

Eine Formel in $\mathcal{D}$ heißt first-order, wenn sie keine Dependence Atome enthält. Die Negation in $\mathcal{D}$ wird nur auf first-order Formeln angewandt, folglich gilt für $\neg\Phi \in \mathcal{D}$ ist $\Phi \in$ **FO**.

Beispiel 2

Seien

$$\mathfrak{A} = \{\alpha, \beta, \gamma\} \tag{1}$$

und

$$X = \begin{array}{|c|c|c|c|c|} \hline & x_0 & x_1 & x_2 & x_3 \\ \hline s_0 & \alpha & \beta & \gamma & \beta \\ s_1 & \alpha & \beta & \beta & \beta \\ s_2 & \beta & \beta & \gamma & \gamma \\ \hline \end{array} \tag{2}$$

Zur verbesserten Übersicht werden Teams als Tabellen dargestellt, wobei die Zeilen für jeweils eine Belegung und die Spalten für die Variablen stehen. Hier ist also $X = \{s_0, s_1, s_2\}$ und beispielsweise $s_1(x_0) = \alpha$.

Es gelten:

$$\mathfrak{A} \models_X =(x_1) \tag{3}$$

$$\mathfrak{A} \models_X =(x_0, x_1) \tag{4}$$

$$\mathfrak{A} \not\models_X =(x_1, x_0) \tag{5}$$

$$\mathfrak{A} \models_X =(x_0, x_3) \tag{6}$$

(3) folgt aus $\mathfrak{A} \models_X x_1 = \beta$, da $s_{1,2,3}(x_1)$ den gleichen Wert für alle Belegungen in X hat. Im Allgemeinen bedeutet ein Dependence Atom mit einem Term, dass der Wert dieses Terms konstant ist. Somit gilt auch (4).

(5) gilt, da $s_0(x_1) = s_2(x_1)$, aber $s_0(x_0) \neq s_2(x_0)$.

(6) fordert, dass Belegungen mit gleichen x_0 die gleichen Werte für x_3 zuweisen und ist daher erfüllt, da die durch $s_0(x_0) = s_1(x_0)$ notwendige Bedingung $s_0(x_3) = s_1(x_3)$ gilt.

Definition 3

Die Menge $Var(t)$ ist die Menge der Variaben, die in einem Term t vorkommen, definiert nach den Regeln der Prädikatenlogik erster Stufe.

Die Menge $Fr(\Phi)$ ist die Menge der freien Variablen einer Formel $\Phi \in \mathcal{D}$, definiert nach den Regeln der Prädikatenlogik erster Stufe, und wird für Dependence Atome erweitert als

$$Fr(=(t_1, ..., t_n)) = \bigcup_{1 \leq i \leq n} Var(t_i).$$

Falls $Fr(\Phi) = \emptyset$ heißt Φ ein Satz.

2.2 Konjunktion und Disjunktion

Während die Konjunktion in der Dependence Logik eine ähnliche Semantik zu der aus der Prädikatenlogik erster Stufe bekannten hat, unterscheidet sich die Bedeutung der Disjunktion in $\mathcal{D}$ von der aus **FO**. In Bezug auf Dependence Atome beschreibt die Disjunktion eine Aufteilung der Belegung in mehrere Teilmengen:

Definition 4

Seien $\mathfrak{A}$ ein Model mit Definitionsmenge A, X ein Team von A und $\Phi, \Psi \in \mathcal{D}$, dann sind

$$\mathfrak{A} \models_X \Phi \wedge \Psi \quad \textbf{gdw.} \quad \mathfrak{A} \models_X \Phi \quad \textbf{und} \quad \mathfrak{A} \models_X \Psi$$

$\mathfrak{A} \models_X \Phi \vee \Psi$ **gdw.** Es gibt Teams Y, Z mit $X = Y \cup Z$, **sodass** $\mathfrak{A} \models_Y \Phi$ **und** $\mathfrak{A} \models_Z \Psi$.

Beispiel 5

Für $\mathfrak{A}$, X aus Beispiel 2 gelten somit:

$$\mathfrak{A} \models_X =(x_1) \wedge =(x_0, x_1) \wedge =(x_0, x_3) \tag{7}$$

$$\mathfrak{A} \models_X =(x_1, x_0) \vee =(x_1, x_0) \tag{8}$$

Die Konjunktion ist genau dann erfüllt, wenn alle durch das $\wedge$ verknüpften Formeln erfüllt sind. (7) gilt, da die Konjunkte (3), (4) und (6) gelten.

Für die Disjunktion wird das Team X so geteilt, dass jede der durch das $\vee$ verknüpften Formeln auf einer der Teilmengen gilt. Dabei reicht es aus, wenn irgend eine solche Aufteilung existiert.

(8) gilt, da X in $Y \coloneqq \{s_0\}$, $Z \coloneqq \{s_1, s_2\}$ geteilt werden kann, mit $X = Y \cup Z$. Für Y, Z ist $\mathfrak{A} \models_{Y \text{resp.} Z} =(x_1, x_0)$ jeweils erfüllt. Hierbei ist besonders, dass $\mathfrak{A} \models_X =(x_1, x_0)$ nicht gilt (siehe (5)): Die Disjunktion ist in $\mathcal{D}$ nicht idempotent, weswegen die Disjunktionselimination nicht möglich ist; für sie gilt außerdem nicht das Distributivgesetz bezüglich der Konjunktion.

2.3 Quantoren

Für die Definition der Semantik der Quantoren werden zuerst das supplementierte und das duplizierte Team definiert:

Definition 6

Sei X ein Team von A und $F : X \to A$. Das supplementierte Team ist definiert als

$$X(F/x_n) \coloneqq \{s(F(s)/x_n) : s \in X\}$$

und das duplizierte Team

$$X(A/x_n) \coloneqq \{s(a/x_n) : s \in X, a \in A\}.$$

Wiederholte Anwendungen von Supplementierung und Duplikation können abgekürzt werden, z.B. $X(F_1/x_1)(F_2/x_2)(A/x_3)$ als $X(F_1F_2A/x_1x_2x_3)$.

Beispiel 7

Die Supplementierung fügt jeder Belegung einen Wert für eine Variable hinzu. Seien beispielsweise X und $\mathfrak{A}$ gegeben aus Beispiel 2, sowie

$$F(s) \coloneqq \begin{cases} \alpha, & \text{falls } s = s_0 \\ \beta, & \text{falls } s = s_1 \\ \gamma, & \text{falls } s = s_2, \end{cases}$$

so ist

$$X(F/x_4) = \begin{array}{|c|c|c|c|c|c|} \hline & x_0 & x_1 & x_2 & x_3 & x_4 \\ \hline s'_0 & \alpha & \beta & \gamma & \beta & \alpha \\ s'_1 & \alpha & \beta & \beta & \beta & \beta \\ s'_2 & \beta & \beta & \gamma & \gamma & \gamma \\ \hline \end{array} \tag{9}$$

Die Duplikation vervielfacht jede Belegung, sodass jeweils alle Werte aus A für die vervielfachte Variable hinzugefügt werden. Für X und $\mathfrak{A}$ ist

$$X[A/x_4] = \begin{array}{|c|c|c|c|c|c|} \hline & x_0 & x_1 & x_2 & x_3 & x_4 \\ \hline s_{0,0} & \alpha & \beta & \gamma & \beta & \alpha \\ s_{0,1} & \alpha & \beta & \gamma & \beta & \beta \\ s_{0,2} & \alpha & \beta & \gamma & \beta & \gamma \\ s_{1,0} & \alpha & \beta & \beta & \beta & \alpha \\ s_{1,1} & \alpha & \beta & \beta & \beta & \beta \\ s_{1,2} & \alpha & \beta & \beta & \beta & \gamma \\ s_{2,0} & \beta & \beta & \gamma & \gamma & \alpha \\ s_{2,1} & \beta & \beta & \gamma & \gamma & \beta \\ s_{2,2} & \beta & \beta & \gamma & \gamma & \gamma \\ \hline \end{array} \tag{10}$$

Definition 8

Seien $\mathfrak{A}$ ein Model mit Definitionsmenge A, X ein Team von A und $\Phi \in \mathcal{D}$, dann sind

$$\mathfrak{A} \models_X \exists x_n \Phi \quad \textbf{gdw.} \quad \mathfrak{A} \models_{X(F/x_n)} \Phi \quad \text{für irgend eine} \quad F : X \to A$$

$$\mathfrak{A} \models_X \forall x_n \Phi \quad \textbf{gdw.} \quad \mathfrak{A} \models_{X(A/x_n)} \Phi$$

Beispiel 9

Seien X und $\mathfrak{A}$ gegeben aus Beispiel 2. Dann gilt

$$\mathfrak{A} \models_X \exists x_4 {=}(x_4, x_3) \tag{11}$$

$$\mathfrak{A} \not\models_X \forall x_4 {=}(x_4, x_3) \tag{12}$$

(11) gilt, da für $X(F/x_4)$ unter anderem

$$F(s) \coloneqq \begin{cases} \alpha, & \text{falls } s = s_2 \\ \beta, & \text{sonst} \end{cases}$$

gewählt werden könnte.

(12) folgt aber daraus, dass durch die Duplikation Belegungen mit verschiedenen Werten für x_4 für gleiche x_3 in $X[A/x_4]$ existieren. Beispielsweise ist $s_0(\alpha/x_4)(x_4) = \alpha \neq \beta = s_0(\beta/x_4)(x_4)$, und die Abhängigkeit $=(x_4, x_3)$ nicht mehr gegeben.

3 Formeldarstellung

3.1 Hintergrund: De Bruijn Indizes

Durch de Bruijn Indizes werden im Lambda-Kalkül Terme geschrieben, ohne den Variablen Namen zu geben. Hierbei wird jedes Vorkommen einer Variable als eine natürliche Zahl dargestellt, die den Gültigkeitsbereich der Variablen beschreibt [BG72]. Beginnend mit 1 zählt dieser Index die Anzahl der Funktionen bis zur die Variable einleitenden Funktion.

Beispiel 10

$$\lambda x.(\lambda y.y) \tag{13}$$
$$\lambda.(\lambda.1)$$

$$\lambda x.(\lambda y.x) \tag{14}$$
$$\lambda.(\lambda.2)$$

$$\lambda y.(\lambda x.x) \tag{15}$$
$$\lambda.(\lambda.1)$$

In (13) bezieht sich y auf die innere Funktion und bekommt daher den de Bruijn Index 1. In (14) bezieht sich x auf die äußere Funktion, beziehungsweise die zweite Funktion von innen, und bekommt daher den de Bruijn Index 2. In (15) bezieht sich x auf die innere Funktion und bekommt daher den de Bruijn Index 1. In der de Bruijn Indizes verwendenden Darstellungsweise sind (13) und (15) gleich.

3.2 Darstellung gebundener Variablen

Die gleiche Methode kann auch zum Darstellen gebundener Variablen in aussagenlogischen Formeln genutzt werden. Hierbei wird der Gültigkeitsbereich der Variablen durch die umgebenden Quantoren definiert. Die gebundene Variable wird wieder durch einen Index, hier beginnend mit 0, repräsentiert. Um die Quantoren $\exists$ und $\forall$ voneinander unterscheiden zu können, werden alle Quantoren durch das jeweilige Symbol ausgedrückt. Auf diese Weise können sämtliche aussagenlogischen Formeln eindeutig dargestellt werden, wobei äquivalente Formeln mit unterschiedlich genannten gebundenen Variablen auf die gleiche Formel abgebildet werden:

Prädikatenlogik!

$$\forall x \exists y P(x, y) \tag{16}$$
$$\forall \exists P(1, 0)$$

$$\forall y \exists x P(y, x) \tag{17}$$
$$\forall \exists P(1, 0)$$

Hierbei ist der Gültigkeitsbereich eines Terms, in dem eine Variable gebunden vorkommt, immer relativ zu den umgebenden Quantoren. Der gleiche Quantor könnte also durch verschiedene Indizes referenziert werden, wenn die jeweiligen Terme einen unterschiedlichen Gültigkeitsbereich haben: In (18) wird die selbe Variable einmal durch den Index 0, als nächst äußerer Quantor, und durch 1, wenn $\exists y$ übersprungen wird, referenziert.

$$\forall x (P(x) \wedge \exists y Q(x)) \tag{18}$$
$$\forall (P(0) \wedge \exists Q(1))$$

Genauso kann der gleiche Index verschiedene Variablen referenzieren:

$$\forall x P(x) \vee \forall y Q(y) \tag{19}$$
$$\forall P(0) \vee \forall Q(0)$$

Außerdem werden Namenskollisionen vermieden. In der Formel $\forall x \forall x P(x)$ bezieht sich die Variable x in $P(x)$ auf das durch den inneren Quantor eingeführte x. Sie wird also dargestellt als $\forall \forall P(0)$. Es ist in der üblichen Darstellung nicht möglich, im Gültigkeitsbereich des inneren Quantors eine Einführung des selben Namens durch einen äußeren Quantor zu referenzieren. Um $\forall \forall P(1)$ auszudrücken, müssten beide Quantoren unterschiedliche Namen haben. Das Verändern des Gültigkeitsbereiches oder Einführen eines neuen Quantors könnte somit immer erfordern, dass Variablen umbenannt werden müssen, was durch die indexbasierte Variante vermieden wird.

Definition 11

Die Substitution der Vorkommen der Variable x in einer Formel Φ, in denen sie als freie Variable auftritt, durch einen Term t, wird geschrieben als $\Phi(t/x)$. Hierbei darf keine in t frei vorkommende Variable gebunden werden.

Beispiel 12

Für $\Phi \underline{=} \forall x \exists z {=}(x, f(x, y), y) \vee {=}(x, f(x, y), y)$ und $t = z$ gilt

$$\Phi(t/x) \underline{=} \forall x \exists z {=}(x, f(x, y), y) \vee {=}(z, f(z, y), y).$$

Die durch das $\forall$ eingeführte gebundene Variable x wurde nicht ersetzt.

Die Substitution $\Phi(t/y)$ ist nicht erlaubt, da die in t frei vorkommende Variable z bei Ersetzen von y im zweiten Dependence Atom durch den Existenzquantor gebunden werden würde. Diese Einschränkung gilt nicht für die eingeführte Indexdarstellungsweise, da die Variablen durch ihren Gültigkeitsbereich differenziert werden.

4 Inferenzregeln

Für Formeln der Dependence Logik können nun Inferenzregeln für ein Kalkül natürlichen Schließens aufgestellt werden. Die aus der Prädikatenlogik erster Stufe bekannten Einführungs- und Eliminationsregeln gelten in $\mathcal{D}$ nur eingeschränkt. Der Grund für die zusätzlichen Bedingungen ist die für Dependence Atome erweiterte Semantik der Disjunktion, sowie die Einschränkung, dass $\neg$ nur auf first-order Formeln erlaubt ist. Auf Formeln ohne Dependence Atome gelten die aus der Prädikatenlogik erster Stufe geltenden Regeln ohne Einschränkungen.

Diese Einführungs- und Eliminationsregeln werden in Definition 13 aufgeführt. Des Weiteren gelten die in Definition 14 definierten Inferenzregeln.

In [KV13] wurde bewiesen, dass für Formeln $T \cup \{\Psi\} \subset \mathcal{D}$ gilt, dass

$$T \vdash_{\mathcal{D}} \Psi \implies T \models \Psi. \qquad \text{Umkehrung gilt, falls Psi eine PL-Formel ist.}$$

Mithilfe der hier beschriebenen Regeln ist es möglich, alle Folgerungen in der Prädikatenlogik erster Stufe aus Sätzen der Dependence Logik herzuleiten.

Definition 13

Operation	Einführung	Elimination
Konjunktion	$\dfrac{A \quad B}{A \wedge B}\,\wedge I$	$\dfrac{A \wedge B}{A}\,\wedge E \qquad \dfrac{A \wedge B}{B}\,\wedge E$
Disjunktion	$\dfrac{A}{A \vee B}\,\vee I \qquad \dfrac{B}{A \vee B}\,\vee I$	$\dfrac{A \vee B \qquad \begin{matrix}[A] \\ \vdots \\ C\end{matrix} \qquad \begin{matrix}[B] \\ \vdots \\ C\end{matrix}}{C}\,\vee E$ (1)
Negation	$\dfrac{\begin{matrix}[A] \\ \vdots \\ B \wedge \neg B\end{matrix}}{\neg A}\,\neg I$ (2)	$\dfrac{\neg\neg A}{A}\,\neg E$ (2)
Allquantor	$\dfrac{A}{\forall x_i A}\,\forall I$ (3)	$\dfrac{\forall x_i A}{A(t/x_i)}\,\forall E$
Existenzquantor	$\dfrac{A(t/x_i)}{\exists x_i A}\,\exists I$	$\dfrac{\exists x_i A \qquad \begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{B}\,\exists E$ (4)

Fonts

Bedingung (1): C ist first-order.
Bedingung (2): A und B sind first-order.
Bedingung (3): Die Variable x_i darf nicht frei in einer bestehenden Annahme, die für die Herleitung von A genutzt wurde, vorkommen.
Bedingung (4): Die Variable x_i darf nicht frei in B oder in einer bestehenden Annahme außer A, die für die Herleitung von B genutzt wurde, vorkommen.

Tabelle 1: Einführungs- und Eliminationsregeln

Definition 14

1. Disjunktionssubstitution:

$$\frac{A \vee B \qquad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{A \vee C}$$

2. Kommutation und Assoziativität der Disjunktion:

$$\frac{A \vee B}{B \vee A}$$

$$\frac{(A \vee B) \vee C}{A \vee (B \vee C)}$$

3. Gültigkeitsbereich-Erweiterung:

$$\frac{\forall x A \vee B}{\forall x (A \vee B)}$$

unter der Vorraussetzung $x \notin Fr(B)$

4. Gültigkeitsbereich-Erweiterung:

$$\frac{\exists x A \vee B}{\exists x (A \vee B)}$$

unter der Vorraussetzung $x \notin Fr(B)$

5. Unnesting:

$$\frac{=(t_1, ..., t_n)}{\exists z (=(t_1, ..., z, ..., t_n) \wedge z = t_i)}$$

wobei z eine neue Variable ist.

6. Dependence Verteilung:

Sei

$$A = \exists y_1 ... \exists y_n (\bigwedge_{i \leq j \leq n} =(\vec{z^j}, y_j) \wedge C)$$

$$B = \exists y_{n+1} ... \exists y_{n+m} (\bigwedge_{n+1 \leq j \leq n+m} =(\vec{z^j}, y_j) \wedge D)$$

wobei C und D quantorenfreie Formeln ohne Dependence Atome sind, und y_i, für $1 \leq i \leq n$, nicht in B vorkommt und y_i, für $n+1 \leq i \leq n+m$, nicht in A vorkommt. Dann gilt

$$\frac{A \vee B}{\exists y_1 ... \exists y_{n+m} (\bigwedge_{1 \leq j \leq n+m} =(\vec{z^j}, y_j) \wedge (C \vee D))}$$

7. Dependence-Einführung:

$$\frac{\exists x \forall y A}{\forall y \exists x (=(\vec{z}, x) \wedge A)}$$

wobei $\vec{z}$ die Variablen in $Fr(A) \setminus \{x, y\}$ auflistet.

8. Dependence-Elimination:

$$\frac{\forall \vec{x}_0 \exists \vec{y}_0 (\bigwedge_{1 \leq j \leq k} =(\vec{w}^{i_j}, y_{0,i_j}) \wedge B(\vec{x}_0, \vec{y}_0))}{\forall \vec{x}_0 \exists \vec{y}_0 (B(\vec{x}_0, \vec{y}_0) \wedge \forall \vec{x}_1 \exists \vec{y}_1 (B(\vec{x}_1, \vec{y}_1) \wedge \bigwedge_{=(\vec{w}_0^p, y_{0,p}) \in S} (\vec{w}_0^p = \vec{w}_1^p \rightarrow y_{0,p} = y_{1,p})))}$$

wobei $\vec{x}_l = (x_{l,1}, ..., x_{l,m})$ und $\vec{y}_l = (y_{l,1}, ..., y_{l,n})$ für $l \in \{0, 1\}$ (hierbei steht $\vec{w}^{i_j}$ für $\vec{w}_0^{i_j}$ und $\vec{w}_1^{i_j}$ bezeichnet das Tupel, das aus $\vec{w}^{i_j}$ durch Ersetzen von $x_{0,s}$ durch $x_{1,s}$, respektive $y_{0,s}$ durch $y_{1,s}$ entsteht), und die Variablen in $\vec{w}^{i_j}$ sind in der Menge $\{x_{0,1}, ..., x_{0,m}, y_{0,1}, ..., y_{0,i_j-1}\}$ enthalten.

Des Weiteren beinhaltet die Menge S die Konjunkte

$$\bigwedge_{1 \leq j \leq k} =(\vec{w}^{i_j}, y_{0,i_j})$$

und das Dependence Atom $=(x_{0,1}, ..., x_{0,m}, y_{0,p})$ für jede der Variablen $y_{0,p} (1 \leq p \leq n)$ mit $y_{0,p} \notin \{y_{0,i_1}, ..., y_{0,i_k}\}$.

9. Die Identitätsaxiome aus **FO**.

5 Theorembeweiser

Theorembeweiser sollen logische Aussagen durch das Anwenden definierter Regeln beweisen. Ein logisches Problem wird in Form von Annahmen und geltenden Axiomen sowie einer Vermutung gegeben und es ist herauszufinden, ob die Vermutung eine logische Folgerung aus der Annahme ist. Hierbei wird zwischen automatischen und interaktiven Theorembeweisern unterschieden.

Ein automatischer Theorembeweiser soll für ein gegebenes Problem selbstständig eine Lösung finden, indem er mithilfe der definierten Axiome einen Beweis für die logische Aussage erarbeit.

Dafür wird meistens ein Beweis durch Widerspruch verwendet. Um zu beweisen, dass für eine Annahme Φ und eine Vermutung Ψ

$\Phi \models \Psi$

gilt, wird gezeigt, dass $\Phi \wedge \neg\Psi$ zu einem Widerspruch führt. Ist dies der Fall, muss das ursprüngliche Theorem gelten.

Interaktive Theorembeweiser erfordern, dass der Benutzer die jeweils zu verwendende Regel angibt, und der Theorembeweiser das Ergebnis dieser Regelanwendung errechnet, oder falls dieses auch angegeben werden soll, überprüft, ob es dem Ergebnis der Inferenz entspricht.

In dieser Arbeit wird die Arbeitsweise eines automatischen Theorembeweisers beschrieben. Das Ziel des Theorembeweisers ist es, first-order Folgerungen aus dependence logischen Sätzen zu überprüfen: Dafür werden dem Theorembeweiser $\Phi_D \in \mathcal{D}$ als Axiom

und $\Psi \in$ **FO** als Vermutung gegeben. In dem Fall, dass $\Phi_D \models \Psi$ soll hierfür ein Beweis gefunden und ausgegeben werden. Der hier verfolgte Ansatz ist aus dem dependence logischen Satz einen Satz Φ der Prädikatenlogik erster Stufe herzuleiten, aus dem die gleichen Folgerungen bewiesen werden können.

Da für die Prädikatenlogik erster Stufe Algorithmen zum Beweisen von Theoremen existieren, kann dann mithilfe eines solchen Verfahrens bewiesen werden, ob $\Phi_D \models \Psi$ gilt.

5.1 Algorithmus

Die gegebene dependence logische Formel wird mithilfe der Inferenzregeln umgeformt, bis sie die für die Dependence-Elimination erforderte Form hat.

Die Aufteilung in die einzelnen Schritte des verwendeten Algorithmus basiert dabei auf dem "Completeness Theorem" aus [KV13]. Die Abarbeitung innerhalb der einzelnen Schritte basiert auf Umformungsfunktionen, die erlaubte Transformationen auf eine gegebene Formel anwenden. Dabei werden schrittweise einzelne der für die Dependence Logik definierten Inferenzregeln oder andere erlaubte Umformungen auf die gegebene dependence logische Formel oder jeweils eine ihrer Teilformeln angewandt, um eine Formel der nächsten geforderten Form zu erhalten. Die daraus entstehende dependence logische Formel ist äquivalent zum initialen Φ_D, da alle hierfür verwendeten Umformungsschritte stets in einer logisch äquivalenten dependence logischen Formel resultieren. Für diese wird dann eine Approximation in der Prädikatenlogik erster Stufe erstellt.

Die initiale dependence logische Formel wird zuerst in eine äquivalente Formel in Pränexform umgeformt. Hierfür wird die Formel bottom-up, rekursiv aufgebaut:

Für eine gegebene Formel werden zuerst ihre Teilformeln in Pränexform gebracht. Eine Negation wird dann auf sämtliche innere Quantoren angewandt. Die Umformungen $\neg\exists x_i\Theta$ zu $\forall x_i\neg\Theta$ sowie $\neg\forall x_i\Theta$ zu $\exists x_i\neg\Theta$ sind möglich, da die ursprüngliche Negation nur auf eine Formel in der Prädikatenlogik erster Stufe angewandt gewesen sein kann, und somit $\Theta \in$ **FO** gelten muss.

Zum Umformen der Konjunktion und Disjunktion werden die Inferenzregeln 3 und 4, Gültigkeitsbereich-Erweiterung, oder die analoge Regel zur Konjunktion, für Teilformeln mit jeweiligen Quantoren verwendet.

Aus dieser Formel in Pränexform wird dann die Form

$$Q^1x_1...Q^mx_m\exists z_1...\exists z_n\left(\bigwedge_{1\leq j\leq n} = (\vec{x}^j, z_j) \wedge \theta\right), \tag{20}$$

mit den Quantoren $Q^i \in \{\exists, \forall\}$ für $1 \leq i \leq m$ und quantorenfreiem $\theta \in$ **FO**, generiert.

Dafür wird auf alle nicht leeren Dependence Atome Regel 5 angewandt. Dadurch ist der letzte Term der jeweiligen Dependence Atome eine durch einen inneren Existenzquantor gebundene Variable. Diese Regel muss auch für Variablen, die an einen bereits in der vorherigen Formel existierenden Existenzquantor gebunden sind, angewandt werden, damit gewährleistet ist, dass der letzte Term aller Dependence Atome ungleich allen

anderen Termen in allen Dependence Atomen ist, was für die Auflösung der Dependence Atome in der Dependence-Elimination Regel notwendig ist.

Für das rekursive Kombinieren der Disjunktionen und Konjunktionen wird Regel 6, Dependence Verteilung, beziehungsweise die analoge Regel zur Konjunktion, respektive, verwendet. Die Regel für die Dependence Verteilung der Konjunktion ist

$$\frac{A \wedge B}{\exists y_1 ... \exists y_{n+m} (\bigwedge_{1 \leq j \leq n+m} =(\vec{z}^j, y_j) \wedge (C \wedge D))}$$

mit A, B, C und D, sowie y_i für $1 \leq i \leq n+m$, definiert wie in Regel 6.

Die resultierende Formel (20) wird danach in eine Form gebracht, die von der Dependence-Elimination Regel verwendet werden kann.

Dafür werden wiederholt alle $\forall$ innerhalb eines $\exists$ mit diesem getauscht, bis die Folge der Quantoren die Form $\forall x_1 ... \forall x_n \exists x_{n+1} ... \exists x_{n+m}$ hat. Dieser Quantorentausch verwendet Regel 7, gefolgt von wiederholtem Anwenden der Gültigkeitsbereicherweiterungsregeln, wobei diese vom Theorembeweiser nicht schrittweise angewandt werden, sondern einmal kombiniert gesammelt und innerhalb der Quantoren zusammengefasst werden, um die Anzahl der gebildeten Formeln und angewandten Umformungen zu reduzieren.

Diese Formel

$$\forall \vec{x}_0 \exists \vec{y}_0 (\bigwedge_{1 \leq j \leq k} =(\vec{w}^{i_j}, y_{0,i_j}) \wedge B(\vec{x}_0, \vec{y}_0))$$

ist logisch äquivalent zu Φ_D und hat die für die Dependence-Elimination benötigte Form.

Mithilfe der Dependence-Elimination Regel kann nun aus diesem Satz $\in \mathcal{D}$ eine infinitäre Formel in der Prädikatenlogik erster Stufe

$$\begin{aligned}
\Phi = \forall \vec{x_0} \exists \vec{y_0} \Bigg(& B(\vec{x_0}, \vec{y_0}) \wedge \\
\forall \vec{x_1} \exists \vec{y_1} \Bigg(& B(\vec{x_1}, \vec{y_1}) \wedge \bigwedge_{=(\vec{w}_0^p, y_{0,p}) \in S} (\vec{w}_0^p = \vec{w}_1^p \rightarrow y_{0,p} = y_{1,p}) \wedge \\
\forall \vec{x_2} \exists \vec{y_2} \Bigg(& B(\vec{x_2}, \vec{y_2}) \wedge \bigwedge_{=(\vec{w}_0^p, y_{0,p}) \in S} (\vec{w}_0^p = \vec{w}_2^p \rightarrow y_{0,p} = y_{2,p}) \wedge \\
& \bigwedge_{=(\vec{w}_0^p, y_{0,p}) \in S} (\vec{w}_1^p = \vec{w}_2^p \rightarrow y_{1,p} = y_{2,p}) \wedge \\
& \vdots \\
& ... \Bigg) \Bigg) \Bigg)
\end{aligned} \tag{21}$$

erstellt werden.

Diese unendliche Formel ist in endlicher Zeit und mit endlichem Speicherplatz nicht darstellbar. Es kann aber eine Approximation $\Phi^k \in \mathbf{FO}$ generiert werden, indem man die Tiefe der Konjunktion auf einen maximalen Generationsparameter k begrenzt.

Beispielsweise ist $\Phi^1 = \forall\vec{x_0}\exists\vec{y_0}(B(\vec{x_0}, \vec{y_0}))$ und Φ^2 das in Regel 8 angegebene Inferenzresultat.

Der Theorembeweiser kann für ein beliebiges $k \geq 1$ die Approximation Φ^k generieren, indem er die aus der Dependence-Elimination resultierende Formel rekursiv bis zur spezifizierten Tiefe aufbaut. Alle Sätze, die aus der Approximation folgen, folgen auch aus der dependence logischen Formel, da das Erfüllen von Φ^k als Teil der unendlichen Konjunktion Φ notwendig für das Erfüllen von Φ ist.

Es handelt sich dabei aber um eine Approximation, und ist möglich, dass nicht alle Folgerungen des dependence logischen Satzes aus einer bestimmten Approximation bewiesen werden können.

Die Approximation Φ^k kann für größere k schwieriger zu erfüllen sein, weshalb es möglich sein kann, dass mit ihr mehr Vermutungen bewiesen werden können.

Φ^k sowie die initiale Vermutung Ψ können dann mithilfe eines Theorembeweisverfahrens für die Prädikatenlogik erster Stufe auf $\Phi^k \models \Psi$ überprüft werden.

In dem Fall, dass für $\Phi^k \models \Psi$ ein Beweis gefunden wird, gilt $\Phi_D \models \Psi$. Falls $\Phi^k \models \Psi$ nicht gilt, wird der Theorembeweiser wahrscheinlich nicht terminieren, bis er sämtlichen verfügbaren Speicherplatz verbraucht. Daraus kann keine Schlussfolgerung bezüglich $\Phi^k \models \Psi$ oder $\Phi_D \models \Psi$ gezogen werden.

Auch in dem Fall, dass $\Phi^k \not\models \Psi$, kann keine Aussage über $\Phi_D \models \Psi$ getroffen werden, da es möglich wäre, für ein größeres $k' > k$ eine Approximation $\Phi^{k'}$ zu bilden, aus der $\Phi^{k'} \models \Psi$ bewiesen werden kann.

5.2 Programmaufbau

Hier wird ein oberflächlicher Überblick über die Struktur des Programmes gegeben.

Das Programm ist modular aufgebaut. Es gibt getrennte Module für einzelne Aufgaben, die die Daten jeweils an das nächste Modul weitergeben.

Das IO Modul ließt die Daten aus der Eingabedatei oder Standardeingabe, je nach Programmargument, ein, und setzt die spezifizierten Optionen. Aus den eingelesenen Formeln werden durch den Parser die internen Formel-Datenstrukturen erstellt. Der Parser basiert auf rekursivem Abstieg. Die Umformer für einzelne Formeln teilen die gleichen Datenstrukturen. Danach gibt es die Formatierung zum Erstellen der passenden Syntax, wie die lesbare Ausgabe in spezifizierter Syntax und die TPTP Darstellung für den Theorembeweiser der Prädikatenlogik erster Stufe. Das IO Modul schreibt die verwendeten Schritte und Regeln in die Standardausgabe und das Resultat in die, durch die entsprechende Option gesetzte, Ausgabedatei.

Der wesentliche Teil der Logik besteht aus den Umformungsregeln und Algorithmen:

Die Regeln sind unabhängig voneinander definiert und getrennt anwendbar.

Die Algorithmen zur Umformung wenden diese Regeln auf eine jeweilige Formel an und erzeugen dadurch eine neue Formel.

Die Formeln werden als Bäume dargestellt, wobei Konjunktion, Disjunktion, Negation und die Quantoren $\exists$ und $\forall$ die inneren Knoten sind, die ihre Teilformeln als Kinder beinhalten. Prädikat und Dependence Atom sind die Blätter und beinhalten ihre Terme. Terme sind auch als Bäume dargestellt, und sind jeweils entweder eine Funktion, die weitere Terme beinhalten kann, oder eine gebundene Variable.

Die gebundenen Variablen werden als De Bruijn Indizes dargestellt und speichern somit nur den Indexwert ihres Gültigkeitsbereiches. Da es sich nur um Sätze handelt, gibt es keine ungebundenen Variablen. Freie Variablen einer Teilformel sind alle Variablen mit größerem Index als der Anzahl äußerer Quantoren.

Gleiche Formeln werden nur einmal erstellt und gemeinsame Teile werden mehrfach referenziert. Wegen der Verwendung der De Bruijn Indizes lassen sich somit viele Formeln zusammenfassen, was die Anzahl an Speicherallokationen verringert. Das erlaubt es, dass gesamte Formelumformungssequenzen effizient abgespeichert werden und nach Durchführung der Algorithmen referenziert und ausgegeben werden können. Dadurch entsteht eine funktionale Arbeitsweise, bei der die einzelnen Formeln und Werte unveränderlich sind und Umformungen als pure Funktionen implementiert werden konnten.

5.3 Verwendung

5.3.1 Kompilieren

Die Module wurden in eine C++ Translation Unit ohne zusätzliche Header oder Dependencies kombiniert. Zum Kompilieren wird mindestens C++17 benötigt. Es wird nur Standard C++ verwendet und alle C++ Compiler kompilieren den Code erfolgreich (getestet mit gcc/g++, clang++, MSVC).

Als Theorembeweiser für die Prädikatenlogik erster Stufe habe ich die Standardinstallation von *E* verwendet. *E* ist ein Theorembeweiser für die Prädikatenlogik erster Stufe mit Gleichheit [Sch98]. Er bietet mehrere Optionen und Erweiterungen, wie beispielsweise die Logik höherer Stufe. Für die von diesem Programm erzeugten Ausgabedateien müssen keine Optionen angegeben oder Erweiterungen verwendet werden. Als Speicherlimit wurde bei meinen Tests immer der maximal verfügbare Speicher genutzt.

Alternativ könnten auch andere Theorembeweiser benutzt werden, da das Programm die Ausgabe in TPTP Syntax erlaubt. Die TPTP Sprache ist weit verbreitet und erlaubt es, logische Formeln darzustellen [Sut17].

Der Theorembeweiser für die Prädikatenlogik erster Stufe kann direkt auf der erzeugten .p Datei aufgerufen werden.

5.3.2 Ausführen

Die Eingabe ist über die Kommandozeile oder über eine Textdatei möglich. Der Theorembeweiser sucht standardmäßig nach einer Datei *input.txt* im aktuellen Pfad. Um eine andere Datei einzulesen, kann der Dateiname beim Programmaufruf an den Theorembeweiser übergeben werden. Alternativ kann eine 0 übergeben werden, um von der Kommandozeile (stdin 0) einzulesen.

Die Eingabe erfolgt in mehreren Zeilen: Die erste Zeile ist die Prämisse, dargestellt als eine beliebige dependence logische Formel. Die zweite Zeile ist die zu beweisende Vermutung in der Prädikatenlogik erster Stufe, also ohne Dependence Atome. Ab der optionalen dritten Zeile können die Programmoptionen durch den jeweiligen Optionsdeskriptor, gefolgt vom entsprechenden Wert, festgelegt werden. Die Approximationstiefe k der zu generierenden Approximation Φ^k in der Prädikatenlogik erster Stufe, mit $1 \leq k \in \mathbb{N}$, wird mit dem Zeichen k, gefolgt von dem Wert angegeben. Die Ausgabedatei wird mit f, gefolgt vom relativen oder absoluten Pfad, angegeben.

Die Eingabe wird beendet, sobald das Ende der Eingabedatei erreicht oder eine Zeile ohne definierten Optionsdeskriptor eingelesen wird.

Für die Eingabe können die Zeichen der Formeln als die entsprechenden UTF-8 codierten Unicodezeichen angegeben werden.

Um die Eingabe über die Kommandozeile oder einen einfachen Texteditor zu erleichtern, wurde zu jedem Symbol ein entsprechendes Computersymbol gewählt, welches durch pure ASCII Zeichen darstellbar ist. Alternativ erkennt der Parser auch das entsprechende LaTeX Kommando.

Die vewendete Eingaberepräsentation wird automatisch erkannt und kann innerhalb der Formeln gemischt verwendet werden.

Die gewählten Symbole werden in der folgenden Tabelle aufgeführt:

Symbol	ASCII-Zeichen	LaTeX	Unicode Code Point
$\wedge$	&	`\land{}`	U+2227
$\vee$	|	`\lor{}`	U+2228
$\neg$	$\sim$	`\lnot{}`	U+00AC
$\rightarrow$	->	`\rightarrow{}`	U+2192
$\leftrightarrow$	<->	`\leftrightarrow{}`	U+2194
$\exists$	?	`\exists{}`	U+2203
$\forall$	!	`\forall{}`	U+2200

Dependence Atome werden als $= (t_1, ..., t_n)$, mit einer kommaseparierten Sequenz an Termen $t_1, ..., t_n$ für beliebige $n \geq 1$ geschrieben.

Ein Prädikat P, das die Terme $t_1, ..., t_n$ für $n \geq 0$ als Argumente hat, wird als $P(t_1, ..., t_n)$ geschrieben. Gleichheit kann als das binäre Prädikat EQ geschrieben werden: $x = y$ wird als $EQ(x, y)$ eingegeben, $x \neq y$ als $\sim EQ(x, y)$.

Ein Quantor wird von dem Namen der durch ihn gebundenen Variable gefolgt. Eine gebundene Variable wird durch ihren Namen dargestellt. Eine Funktion mit Name f und Argumenten $t_1, ..., t_n$ wird als $f(t_1, ..., t_n)$ geschrieben.

Alle Symbole haben die aus der Prädikatenlogik erster Stufe bekannte Präzedenz und sind linksassoziativ. Es können Klammern zum Anpassen der Präzedenz verwendet werden. Ein Punkt kann verwendet werden, um Klammern um die auf ihn folgende Formel zu ersetzen.

Die Ausgabesyntax für die Standardausgabe wird über den Optionsdeskriptor s gefolgt von dem Namen der gewünschten Sytax spezifiziert. Die verwendeten Namen sind *ascii*, *latex* und *unicode*, wobei *unicode* eine UTF-8 kodierte Zeichenausgabe verwendet.

5.3.3 Beispiel

Der Satz

$$\Phi_D = \exists z \forall x \exists y \, (=(y, x) \wedge \neg y = z)$$

ist wahr in einem Model $\mathfrak{A}$ genau dann, wenn seine Domäne A unendlich ist.

Die Formel erfordert, dass es auf der Menge A eine Funktion gibt, die durch $=(y, x)$ das jeweilige y auf genau ein x abbildet. Sie muss daher eine injektive Abbildung von y auf x darstellen. Da aber für alle x für das selbe z auch $\neg y = z$ gelten muss, könnte diese Abbildung auf endlichem A nicht surjektiv sein. $\forall x \exists y(...)$ wäre nicht gegeben, da es für mindestens ein x kein solches y gäbe.

A muss somit unendlich sein.

$\Psi_{\geq 3} := \exists a \exists b \exists c (a \neq b \wedge b \neq c \wedge a \neq c)$ ist wahr genau dann, wenn es 3 oder mehr Elemente in A gibt. Da $\Psi_{\geq 3}$ logisch aus Φ_D folgt und $\Psi_{\geq 3} \in \mathbf{FO}$ sollte der Theorembeweiser $\Phi_D \models \Psi_{\geq 3}$ beweisen können.

Die Eingabe für den Theorembeweiser erfolgt in der beschriebenen Syntax; als Approximationstiefe wird für dieses Beispiel 2 gewählt. Die Approximationstiefe 2 ist der Standardwert und müsste nicht explizit gesetzt werden. Der Punkt fasst die restliche Formel zu einer Konjunktion zusammen, statt Klammern verwenden zu müssen. Ohne ihn würde der Quantor nur auf das erste Prädikat gebunden werden.

```
?z!x?y. =(y,x) & ~EQ(y,z)
?a?b?c. ~EQ(a,b) & ~EQ(b,c) & ~EQ(a,c)
k 2
```

Zuerst wird nun mithilfe des beschriebenen Algorithmus Φ_D umgeformt, bis eine äquivalente Formel erreicht wird, aus der eine Approximation in der Prädikatenlogik erster Stufe generiert werden kann.

Die Formel ist bereits in Pränexform und wird für den ersten Schritt daher nicht umgeformt.

Durch Anwenden der Regel 5 wird eine neue Variable u eingeführt. Der letzte Term des Dependence Atoms ist nun, wie benötigt, durch den innersten Quantor gebunden:

$$\Phi_D \equiv \exists z \forall x \exists y \exists u (=(y, u) \wedge (\neg y = z \wedge u = x))$$

Für das Kombinieren der quantorenfreien Teilformeln sind keine weiteren Umformungen nötig, da keine Konjunktionen oder Disjunktionen dependence logischer Teilformeln vorhanden sind.

Danach werden die äußeren Quantoren getauscht, um den Allquantor von x nach außen zu setzen. Dafür wird Regel 7 durchgeführt. Weil

$$Fr(\exists y \exists u (=(y, u) \wedge (\neg y = z \wedge u = x))) \setminus \{x, z\} = \emptyset,$$

wird das Dependence Atom $=(z)$ eingeführt. Dieses wird durch die Gültigkeitsbereicherweiterung in den dependence logischen Teil der inneren Konjunktion gebracht:

$$\Phi_D \equiv \forall x \exists z \exists y \exists u (=(z) \wedge =(y, u) \wedge (\neg y = z \wedge u = x)) \tag{22}$$

Beim Anwenden der Dependence-Elimination sind

$$x_0 = (x),$$

$$y_0 = (z, y, u),$$

$$i = (1, 3),$$

$$w = (\emptyset, \{y\}).$$

S beinhaltet die beiden Dependence Atome aus (22), sowie das Dependence Atom $=(x, y)$, da $y = y_{0,2} \notin \{y_{0,i_1}, ..., y_{0,i_k}\}$.

Somit ist $S \coloneqq \{=(z), =(y, u), =(x, y)\}$.

Für die Approximationstiefe 2 wird dadurch die Approximation in der Prädikatenlogik erster Stufe Φ^2 generiert:

Liefert das Programm diese Ableitung? „Trace"?

$$\begin{aligned}\Phi^2 =& \forall x \exists z \exists y \exists u \Big(\neg y = z \wedge u = x \wedge \\ & \forall x_2 \exists z_2 \exists y_2 \exists u_2 \Big(\neg y_2 = z_2 \wedge u_2 = x_2 \wedge \\ & (z = z_2 \wedge (\neg y = y_2 \vee u = u_2) \wedge (\neg x = x_2 \vee y = y_2)) \\ & \Big) \Big)\end{aligned} \tag{23}$$

Dann wird $\Phi^2 \models \Psi_{\geq 3}$ als Problem der Prädikatenlogik erster Stufe an einen first-order Theorembeweiser gegeben.

Für diesen Fall lässt sich ein Beweis finden. Mit $k = 1$ kann noch kein Beweis für $\Phi^1 \models \Psi_{\geq 3}$ gefunden werden.

Um $\Psi_{\geq 3}$ zu beweisen, wird also $k \geq 2$ benötigt.

Aus größeren k lässt sich $\Psi_{\geq i}$ für größere i beweisen: $\Psi_{\geq 2}$ folgt bereits aus $k \geq 1$, da für $\Psi_{\geq 2} \coloneqq \exists a \exists b (a \neq b)$ bereits das erste durch die Dependence-Elimination generierte $y \neq z$ ausreicht.

Es konnte mit diesem Φ_D aber nicht $\Phi_D \models \Psi_{\geq 4}$ bewiesen werden. Bei allen ausprobierten Approximationstiefen läuft zuvor der Speicherplatz aus. Daher wurde versucht, den gleichen Beweis mit einer anderen Formel als Prämisse durchzuführen:

Sei

$$\Phi_{D,2} \coloneqq \forall x \exists y \exists z (=(y, z) \wedge (x = z \wedge y \neq c)), \tag{24}$$

wobei c eine Konstante ist. Es gilt $\mathfrak{A} \models \Phi_{D,2}$ genau dann, wenn die Definitionsmenge A von $\mathfrak{A}$ unendlich ist.

Für die Approximationen Φ_2^n dieses Satzes gilt $\mathfrak{A} \models \Phi_2^k$ genau dann, wenn $|A| \geq k + 1$. Die Konstante wird im Theorembeweiser als eine nulläre Funktion dargestellt. Mit $\Phi_{D,2}$ ließen sich die gleichen Sätze wie mit Φ_D beweisen. Es ließ sich aber auch $\Phi_{D,2} \models \Psi_{\geq 4}$ beweisen, wobei der benötigte Speicherplatz und Zeitaufwand bereits deutlich höher als für den Beweis von $\Psi_{\geq 3}$ waren.

Das liegt vermutlich an der ineffizienten Darstellung der Formeln Φ_2^i und $\Psi_{\geq i+1}$, die quadratisch zu i wachsen. Insbesondere die Länge der Approximationen in der Prädikatenlogik erster Stufe, Φ_2^i beziehungsweise Φ^i, wird bereits für kleine i sehr groß. Das liegt an der aus der Dependence Elimination folgenden Form, bei der die verschachtelten Konjunktionen an Vergleichen jedes neuen i mit allen äußeren wiederholt werden müssen, und kann bei diesem Ansatz nicht vermieden werden.

$\Phi_{D,2}$ ist für das Beweisen von $\Psi_{\geq i}$ besser geeignet als Φ_D. Bereits $\Phi_{D,2} \models \Psi_{\geq 5}$ lässt sich aber auch mit $\Phi_{D,2}$ nicht mehr beweisen.

Für komplexere Sätze, insbesondere Sätze mit mehr Quantoren, werden die erzeugten Approximationen größer und der Beweis in der Prädikatenlogik erster Stufe gelingt schon für einfachere Folgerungen nicht mehr.

Sei

$$\begin{aligned}\Phi_{\equiv 0 \bmod 2} \coloneqq \forall x_0 \exists x_1 \exists x_2 \exists x_3 (&\neg x_0 = x_1 \\ &\wedge =(x_2, x_3)) \\ &\wedge (x_0 = x_2 \to x_1 = x_3) \\ &\wedge (x_1 = x_2 \to x_3 = x_0)).\end{aligned}$$

$\Phi_{\equiv 0 \bmod 2}$ ist auf $\mathfrak{A}$ mit einer endlichen Definitionsmenge A genau dann wahr, wenn die Anzahl der Elemente in A gerade ist [Vää07]. Des Weiteren seien $\Psi_{\neq 1}$ beziehungsweise $\Psi_{\neq 3}$ wahr genau dann, wenn die Anzahl der Elemente ungleich zu 1, beziehungsweise 3, respektive, ist.

Offensichtlich gelten

$$\Phi_{\equiv 0 \bmod 2} \models \Psi_{\neq 1} \tag{25}$$

sowie

$$\Phi_{\equiv 0 \bmod 2} \models \Psi_{\neq 3}. \tag{26}$$

Für (25) konnte der Theorembeweiser einen Beweis finden; für (26) war das bereits nicht mehr möglich.

5.4 Komplexität des Algorithmus

Hier werden der verwendete Algorithmus und seine Implementierung, sowie das Aufbauen der Approximation in der Prädikatenlogik erster Stufe, bezüglich ihrer Laufzeitkomplexität und des verwendeten Speicherplatzes untersucht.

Das verwendete Maschinenmodell entspricht einem üblichen Computer mit konstanter Wortgröße und random access memory, die den Zugriff auf beliebigen Speicher über einen Pointer in konstanter Zeit erlaubt. Das Instruction Set erlaubt Arithmetik, Vergleichen, Laden und Speichern mit Werten einer Wortgröße in konstanter Zeit.

Als atomare Operationen werden das Erstellen einer neuen Formel aus gegebenen Teilformeln, das Vergleichen zweier Formeln auf Gleichheit, und der Zugriff auf eine direkte Teilformel einer Formel definiert:

Aufgrund der einmaligen Speicherung gleicher Formeln ist der Vergleich zweier beliebig großer Formeln als ein einfacher Gleichheitsvergleich ihrer Pointer implementiert. Diese Operation kann als $\mathcal{O}(1)$ angenommen werden und ist in allen Computern eine der schnellsten existierenden Instruktionen.

Das Erzeugen einer neuen Formel aus gegebenen Teilformeln verwendet maximal zwei Teilformeln, nämlich für Konjunktion und Disjunktion. Eine Darstellung dieser als mehr elementige Mengen aus ihren Konjunkten beziehungsweise Disjunkten wäre auch vorstellbar. Die Hashwerte der Teilformelpointer werden kombiniert und damit wird diese neue Formel in einem Hashset gespeichert, oder, falls bereits vorhanden, referenziert. Dabei wird maximal ein einziger neuer Knoten mit konstanter Größe erzeugt. Der Zugriff auf eine direkte Teilformel oder einen direkten Teilterm dereferenziert den entsprechenden Pointer.

Da Terme eine gleiche Datenstruktur wie Formeln, als gerichteten azyklischen Graph, dessen einzelne Knoten jeweils unterschiedliche Werte darstellen, verwenden, gelten die gleichen Annahmen auch für diese.

Einen Sonderfall bilden Prädikate und Funktionen, deren Knoten möglicherweise mehr Kinder haben können. Die Anzahl der gespeicherten Terme ist bei ihnen durch die Anzahl ihrer Terme in der Eingabe gegeben. Diese werden aber nur beim Tauschen zweier Quantoren oder Hinzufügen eines Inneren Quantors erstellt, und können für die Beurteilung des restlichen Algorithmus ignoriert werden. Der Zugriff auf einen bestimmten Term und das Vergleichen auf Gleichheit sind, unabhängig von der Anzahl dieser Terme, wie beschrieben $\mathcal{O}(1)$.

5.4.1 Algorithmus

Für den ersten Schritt des Algorithmus, die Umwandlung in die Pränexform, ergibt sich eine Laufzeit in $\mathcal{O}(n^2)$ für eine Formel mit n Teilformeln, darunter $n_{\exists,\forall}$ Quantoren und $n_{\wedge,\vee,\neg}$ logischen Operationen. Jeder Quantor kann maximal einmal pro äußerer Negation, Konjunktion oder Disjunktion verschoben werden. Dabei werden immer höchstens zwei neue Knoten, der nun äußere Quantor und die nach innen verschobene logische Operation, erstellt. Es ergibt sich eine Laufzeit in $\mathcal{O}(n_{\exists,\forall} \cdot n_{\wedge,\vee,\neg})$, und für eine Formel mit $n_{\exists,\forall} \sim n_{\wedge,\vee,\neg} \sim n$ somit $\mathcal{O}(n^2)$. In dem häufigen Fall, dass die gegebene Formel bereits in Pränexform war, wird diese Tatsache in $\Theta(n_{\exists,\forall})$ herausgefunden, da alle Formeln abspeichern, ob sie Quantoren beinhalten, und das Erreichen der äußersten quantorenfreien Formel durch sukzessives Dereferenzieren der Quantoren somit diesen Schritt terminiert.

Die Umformung zur kombinierten Konjunktion der Dependence Atome verwendet Regeln 5, 6 und das Regel 6 Analog zur Konjunktion. Die Durchführung der Regel 5 kann ein Umschreiben der inneren Terme nach sich ziehen, um den veränderten Index der gebundenen Variablen anzupassen. Eine einmalige Anwendung dieser Regel muss daher als $\mathcal{O}(n)$ angenommen werden. Sie wird einmal auf jedes Dependence Atom angewandt. Die Regel 6, für Konjunktionen und Disjunktionen, kombiniert die Konjunktionen der Dependence Atome und ihre quantorenfreien first-oder Teilformeln. Das ist in $\mathcal{O}(1)$ möglich, da die neue Formel nur auf diese bereits existierenden Teilformeln zeigt. Das Erreichen des quantorenfreien inneren Teils der zu kombinierenden Formeln erfordert $n_{\exists,\forall}$ Pointer-Dereferenzierungen und das Kopieren der $n_{\exists,\forall}$ Quantoren wurde zu einem Schritt zusammengefasst, wobei auch hier die Indizes der auf die nun äußeren Quantoren verweisenden gebundenen Variablen angepasst werden müssen. Insgesamt ergibt sich für diesen Schritt die obere Grenze $\mathcal{O}(n \cdot n_{=(t_1,\ldots,t_n)} + n_{\exists,\forall} + n)$, mit $n_{=(t_1,\ldots,t_n)}$ als Anzahl der Dependence

Atome, also $\mathcal{O}(n^2)$ falls die Anzahl der Dependence Atome in Φ_D entsprechend groß ist.

Das Anwenden der Regel 7 läuft in $\mathcal{O}(n)$ ab, da auch hier die inneren Terme angepasst werden müssen, und die freien Variablen der inneren Formel gesammelt werden müssen. Es wird für alle $n_\forall$ Allquantoren auf alle äußeren der $n_\exists$ Existenzquantoren angewandt, und ist dadurch durch $\mathcal{O}(n_\forall \cdot n_\exists \cdot n)$ begrenzt.

Die Umformung zur Pränexform und das Kombinieren der Konjunktion der Dependence Atome erzeugen jeweils eine Formel, deren Größe proportional zu der der Ausgangsformel ist. Beim wiederholten Quantorentausch durch Regel 7 können aber, in dem Fall, dass alle Existenzquantoren außerhalb aller Allquantoren sind, bis zu $n_\forall \cdot n_\exists$ neue Dependence Atome hinzugefügt werden.

Die gesamte Laufzeit des Algorithmus liegt in

neu?

$$\mathcal{O}(n_{\exists,\forall} \cdot n_{\wedge,\vee,\neg} + (n \cdot n_{=(t_1,\ldots,t_n)} + n_{\exists,\forall} + n) + n_\forall \cdot n_\exists \cdot n) \subseteq \mathcal{O}(n^3),$$

für n proportional zur Größe der ursprünglichen Φ_D.

5.4.2 Approximationsaufbau

Ausschlaggebend für die Größe der erzeugten Approximation und die Laufzeit ihrer Generierung ist die gewählte Approximationstiefe k.

Vor der Anwendung der Dependence-Elimination Regel muss das Programm erst die Elemente der Menge S herausfinden. Das Dereferenzieren der Quantoren ist wieder in $\mathcal{O}(n_{\exists,\forall})$ durchführbar. Die Konjunkte

$$\bigwedge_{1 \leq j \leq k} =(\vec{w}^{i_j}, y_{0,i_j})$$

lassen sich dabei direkt aus der gegebenen Formel übernehmen, und das Dependence Atom $=(x_{0,1}, ..., x_{0,m}, y_{0,p})$ für jede der Variablen $y_{0,p} (1 \leq p \leq n)$ mit $y_{0,p} \notin \{y_{0,i_1}, ..., y_{0,i_k}\}$ wird ermittelt, indem die aus i fehlenden p der $y_{0,p}$ aller $\vec{y}_0$ der Existenzquantoren gesammelt werden. Dafür werden diese Quantoren und Dependence Atome einmal linear durchlaufen. Somit ergibt sich dafür insgesamt eine Laufzeit, sowie der Speicheraufwand, um S zu speichern, in $\mathcal{O}(n_{\exists,\forall} + n_{=(t_1,\ldots,t_n)})$.

Die Dependence Elimination erzeugt oft eine sehr große Formel, aber da gemeinsame Teile der Formel nicht neu allokiert werden, kann ein Großteil des sonst notwendigen Speicherverbrauchs gespart werden:

Alle in der Prädikatenlogik erster Stufe gegebenen Teilformeln, in der dependence logischen Formel als $B(\vec{x}_0, \vec{y}_0)$ bezeichnet, werden als $B(\vec{x}_i, \vec{y}_i)$ für $0 \leq i < k$ generiert, aber beziehen sich immer nur auf die davor liegenden $n_{\exists,\forall}$ Quantoren und sind somit in der verwendeten De Bruijn Darstellungsweise gleich. Sie referenzieren die ursprüngliche Formel ohne neuen Speicherplatz zu verwenden.

Die Vergleiche der $\vec{w}_i^p$ und $y_{i,p}$ in der verschachtelten Konjunktion der Tiefe k beziehen sich immer auf Quantoren der Tiefen $k, k-1, ..., 0$. Daher muss für ihren verwendeten Speicherplatz nur die Konjunktion der innersten Approximationstiefe betrachtet werden, alle anderen sind darin enthalten und referenzieren Teile in ihr. Somit ist die Anzahl der

gespeicherten Vergleiche proportional, statt quadratisch, zu k. Die gesamte Zahl der Konjunkte der Implikationen ist immer noch quadratisch zu k, diese Konjunkte verbrauchen aber wegen des Teilens ihrer Teilformeln als einzelner Knoten jeweils wenig Speicherplatz.

Die benötigte Zeit ist auch quadratisch, da jede dieser Implikationen emittiert werden muss. Somit liegen Laufzeit- und Speicherkomplexität beide in $\mathcal{O}(k^2 \cdot (n'_{\exists} + n'_{=(t_1,\ldots,t_n)}))$, wobei sich $n'_{\exists}$ und $n'_{=(t_1,\ldots,t_n)}$ hier auf die jeweilige Anzahl nach Durchführung des Algorithmus beziehen und größer als in Φ_D sein können:

Die Anzahl der durch den Algorithmus erzeugten Dependence Atome war höchstens $n_{\forall} \cdot n_{\exists}$ durch wiederholtes Anwenden der Regel 7, die Anzahl der Quantoren konnte bis dahin, durch Regel 5, nur um die Anzahl der ursprünglichen Dependence Atome in Φ_D steigen. Somit ist die Größe der Approximation immer in $\mathcal{O}(k^2 \cdot n^2)$, mit n als Größe der ursprünglichen Φ_D, darstellbar.

Als letzter, ineffizienter Bearbeitungsschritt muss diese Formel ausgegeben werden. Diese Formel ist zwar intern effizient dargestellt, um an den Theorembeweiser der Prädikatenlogik erster Stufe gegeben zu werden, muss sie aber durch jeweiliges Ausschreiben der geteilten Teilformeln pro besitzenden Knoten komplett ausgegeben werden. Der Zeitaufwand des Ausschreibens liegt folglich in $\mathcal{O}(k^2 \cdot n^2)$ und der Theorembeweiser für die Prädikatenlogik erster Stufe enthält eine Formel, die um bis zu $k^2 \cdot n^2$ der ursprünglichen Φ_D gewachsen sein kann.

6 Möglicher Ansatz mit Heuristik

Der Theorembeweiser unterstützt Umformungen in beliebiger Reihenfolge anhand der definierten Regeln. Diese Regeln können auf jede Formel angewandt werden, die die von der Regel geforderte Form hat.

Beim derzeitigen Ansatz werden diese Regeln gezielt verwendet, um den beschriebenen Algorithmus abzuarbeiten und letztendlich eine Approximation in der Prädikatenlogik erster Stufe zu erzeugen.

Die von der Dependence-Elimination geforderte Form wird dabei immer erreicht und die Umformungen laufen auch sehr zeit- und ressourceneffizient. Es muss aber festgelegt werden, bis zu welcher Approximationstiefe diese Approximation generiert werden soll, da eine unendliche Formel nicht darstellbar ist. Dadurch ist ein Beweis einer korrekten Aussage nicht garantiert.

Das Programm ermöglicht es abzufragen, welche Regeln auf eine Formel bestimmter Form angewandt werden können. Diese Funktion wird teilweise im Algorithmus verwendet um den nächsten durchzuführenden Umformungsschritt herauszufinden.

Mithilfe dieser Abfragefunktionen und der entsprechenden Umformungsregeln wurde auch ein nicht zielgesteuerter Ansatz ausprobiert, bei dem durch willkürliches Anwenden der Regeln auf der ursprünglichen Formel Φ_D sowie den dadurch entstandenen Formeln eine Menge an logisch aus Φ_D folgenden Formeln aufgebaut wird.

Dieser brute force Ansatz müsste mit beliebigem Speicherplatz und Zeitaufwand immer einen Beweis finden. Da aber bestimmte Regeln, wie die Einführung eines Quantors oder die Konjunktion zweier bestehender Regeln, immer angewandt werden können, wurde

die Formelmenge mit immer größeren Formeln gefüllt und das Programm wurde durch den begrenzten Speicherplatz unterbrochen.

Auch ein Anwenden der Regeln mit Präferenz derer, deren Umformungen eine bestimmte strukturelle Vorbedingung hatten und die deshalb zu generell kleineren Formeln führten, war nicht effizient genug und kann nicht zum Herleiten der Vermutung oder einer guten Approximation genutzt werden. Es wurden immer noch zu viele Formeln generiert, die für den Beweis nicht sinnvoll waren.

Es wäre aber möglich die Regeln gezielter anzuwenden und durch das Anwenden dieser Regeln einen Beweis durchzuführen, indem eine passende Heuristik genutzt wird, um die jeweils nächsten sinnvollen Regeln auszuwählen.

Mithilfe dieser Heuristik sollten auf als nützlich klassifizierten Formeln bestimmte Regeln angewandt werden, um jeweils gezielte, zielführende Umformungen durchzuführen. Hierbei könnten Umformungen bevorzugt werden, deren Resultate die Struktur der zu beweisenden Vermutung Ψ haben. Das könnte es ermöglichen, einen Beweis durchzuführen, ohne dass ein separater Beweis durch einen Theorembeweiser der Prädikatenlogik erster Stufe erfordert wird, indem Ψ direkt, nur durch das Verwenden der Umformungsregeln, aus Φ_D hergeleitet wird.

In diesem Fall müsste keine Approximationstiefe vorher spezifiziert werden.

Es ist unklar ob eine solche Heuristik implementiert und mit ausreichend geringem Zeit- und Speicheraufwand evaluiert werden könnte.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Dependence Logik erklärt und ein automatischer Theorembeweiser für diese entwickelt. Der Theorembeweiser kann Folgerungen in der Prädikatenlogik erster Stufe aus dependence logischen Sätzen beweisen. Das verwendete Verfahren ist, die gegebene dependence logische Prämisse in einen äquivalenten dependence logischen Satz umzuwandeln, aus dem eine Approximation in der Prädikatenlogik erster Stufe erstellt wird.

Der Theorembeweiser wurde getestet, indem verschiedene dependence logische Sätze als Prämisse gegeben und umgeformt wurden.

Die Inferenzregeln aus dem $\mathcal{D}$ Kalkül wurden dabei korrekt angewandt und es kann für jeden gegebenen dependence logischen Satz die Approximation mit der spezifizierten Approximationstiefe generiert werden. Damit konnten Beweise für bestimmte, kleinere Folgerungen in der Prädikatenlogik erster Stufe berechnet werden. Für größere Formeln war es aber nicht mehr möglich, mit dieser Approximation den Beweis in der Prädikatenlogik erster Stufe durchzuführen.

Das Programm ist bedingt konfigurierbar, da es möglich ist, die gewünschte Approximationstiefe k anzugeben, statt nur die in der Inferenzregel angegebene Approximationstiefe $k = 2$ zu verwenden. Das Angeben dieser Approximationstiefe ist zwar vor der Durchführung des Beweises nötig, es ist aber oft nicht intuitiv absehbar, welches k für das Beweisen eines bestimmten Satzes in der Prädikatenlogik erster Stufe ausreicht.

Der zur Umformung genutzte Algorithmus ist davon unabhängig und erreicht für alle getesteten Prämissen schnell den äquivalenten umgeformten Satz.

Das Programm ist modular und erweiterbar geschrieben und es ist möglich, andere Ansätze zu implementieren:

In Kapitel 6 wurde ein möglicher Ansatz diskutiert, bei dem die Regelanwendungen auf Basis einer gegebenen Heuristik durchgeführt werden würden. Dieser Ansatz lässt sich dank der Struktur des Programms gut einbauen, und eine ähnliche Herangehensweise wurde bereits ausprobiert. Anfänglich wurden die Regelanwendungen ohne eine gute Heuristik ausprobiert, was nicht effizient genug war und durch den derzeitigen, gezielten Ansatz mit Algorithmus ersetzt wurde. Das Finden einer passenden Heuristik wäre eine spannende Weiterführung dieser Arbeit.

Ein weiterer möglicher Ansatz ist die Implementierung eines interaktiven Theorembeweisers. Der interaktive Theorembeweiser müsste die jeweils zu verwendende Regel einlesen und auf eine gegebene Formel oder eine ihrer Teilformeln anwenden. Dieser sollte durch die definierten Regelfunktionen, sowie die dazu definierten Abfragefunktionen zum Herausfinden, welche Regeln anwendbar sind, gut umsetzbar sein. Mit diesem könnte man manuell oder extern berechnete Beweise, die auf dem $\mathcal{D}$ Kalkül basieren, überprüfen. Für diesen Ansatz wäre vorheriges Wissen darüber, welche Regeln wann angewandt werden müssen, nötig. Es wäre auch denkbar, den gewünschten Grad der Automatisierung festzulegen. Der Theorembeweiser würde teilweise selbstständig Umformungen durchführen, und bei bestimmten Schritten, je nach gewünschtem Grad der Automatisierung, durch Eingaben gelenkt werden.

Bei dem derzeitigen Ansatz ist der wesentliche limitierende Faktor die Größe der generierten Formeln. Die generierten Approximationen sind oft sehr groß, und ihre Größe steigt quadratisch zu k an. Wird diese Approximationstiefe aber zu klein gewählt, lässt sich der Beweis nicht durchführen, da die generierte Approximation nicht stark genug ist. Es wäre interessant herauszufinden, welche Approximationstiefe für einen zu beweisenden Satz der Prädikatenlogik erster Stufe benötigt wird.

Literatur

[BG72] de Bruijn und Nicolaas Govert. “Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem”. In: (1972).

[Sch98] Stephan Schulz. *The E Theorem Prover*. `www.eprover.org`. [Online; letzter Zugriff Februar 2021], Version release 2.5. 1998.

[Vää07] Jouko A. Väänänen. *Dependence Logic - A New Approach to Independence Friendly Logic*. Bd. 70. London Mathematical Society student texts. Cambridge University Press, 2007. ISBN: 978-0-521-70015-3. URL: `http://www.cambridge.org/de/knowledge/isbn/item1164246/?site%5C_locale=de%5C_DE`.

[KV13] Juha Kontinen und Jouko A. Väänänen. “Axiomatizing first-order consequences in dependence logic”. In: *Ann. Pure Appl. Log.* 164.11 (2013), S. 1101–1117. DOI: `10.1016/j.apal.2013.05.006`. URL: `https://doi.org/10.1016/j.apal.2013.05.006`.

[Abr+16] Samson Abramsky u. a., Hrsg. *Dependence Logic, Theory and Applications*. Springer, 2016. ISBN: 978-3-319-31801-1. DOI: `10.1007/978-3-319-31803-5`. URL: `https://doi.org/10.1007/978-3-319-31803-5`.

[Sut17] G. Sutcliffe. “The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0”. In: *Journal of Automated Reasoning* 59.4 (2017), S. 483–502.