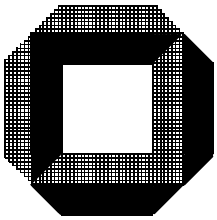

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme
Lehrstuhl: Prof.Dr. P.H.Schmitt
Betreuer: Joachim Posegga

Ein Deduktionssytem für Prädikatenlogik erster Stufe basierend auf Shannon-Graphen

Diplomarbeit

Bertram Ludäscher

30. Oktober 1992

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig und ohne unzulässige fremde Hilfe verfaßt und keine anderen als die angegebenen Quellen benutzt habe.

Bertram Ludäscher Karlsruhe, 30. Oktober 1992

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit Theorie und Implementierung von einigen auf *Shannon-Graphen* basierenden Beweisverfahren für Aussagenlogik und Prädikatenlogik erster Stufe.

Es wird gezeigt, daß Shannon-Graphen eine für das automatische Beweisen geeignete Darstellung von logischen Formeln sind. Der Shannon-Graph einer aussagenlogischen Formel F enthält sowohl Information über die Modelle von F als auch über die Modelle von $\neg F$. Dies hat zur Folge, daß die Beweisverfahren für Shannon-Graphen die aus dem Tableaukalkül bekannte *Lemma-Generierung* "automatisch" beinhalten. Die vorgestellten Verfahren können durch Verwendung einer bestimmten Übersetzungstechnik effizient implementiert werden. Die Grundidee dieser Übersetzung besteht darin, aus der Darstellung einer Formel F in Form eines Shannon-Graphen ein Programm zu generieren, das die Suche nach einer Widerlegung speziell für F vornimmt.

In dieser Arbeit werden die theoretischen Grundlagen von Shannon-Graphen und den darauf aufbauenden Beweisverfahren behandelt und gezeigt, daß diese Verfahren korrekt und vollständig sind. Auf die Realisierung der Verfahren wird ausführlich eingegangen und die Implementierung des im Rahmen dieser Arbeit entstandenen Beweisers beschrieben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Mit Hilfe von automatischen Theorembeweisern können Sätze der Prädikatenlogik verifiziert werden. Zum Aufbau von logischen Formeln werden in den meisten Verfahren des automatischen Beweisens die "klassischen" Junktoren $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, oder eine geeignete Auswahl derselben, verwendet.

Alternativ kann man logische Formeln auch mittels eines dreistelligen *if-then-else* Junktors darstellen. Einer der wenigen Ansätze im automatischen Beweisen, die auf dieser Darstellung beruhen, stellt die aussagenlogische Beweisprozedur von [Ehrenfeucht & Orłowska, 1967] dar. Diese wurde später auf eine entscheidbare Formelklasse der Prädikatenlogik erweitert [Orłowska, 1969a].

Die Repräsentation von boole'schen Funktionen in Form von sogenannten *B*inary *D*ecision *D*iagrams (BDDs) basiert ebenfalls auf dem genannten Junktor [Lee, 1959]. Erfahrungen in neuerer Zeit, unter anderem im Bereich der Hardware-Verifikation, haben gezeigt, daß mit BDDs eine effiziente Handhabung boole'scher Funktionen möglich ist [Bryant, 1986, Brace *et al.*, 1990]. Daher wurde der Versuch unternommen, dieses Prinzip in Richtung einer Beweisprozedur für die Prädikatenlogik erster Stufe zu erweitern.

Ein erster Ansatz hierzu wurde in [Posegga & Ludäscher, 1992] beschrieben. Bei diesem werden logische Formeln in Form von *Shannon-Graphen*, einer den BDDs verwandten Repräsentation, dargestellt. Dieser Ansatz wurde später zu einem leistungsfähigen Beweisverfahren für die Prädikatenlogik ausgebaut [Posegga, 1992]. Die Grundidee für die effiziente Implementierung des Verfahrens besteht darin, eine gegebene Formel F zunächst in eine Shannon-Graph-Darstellung zu überführen. Aus dieser wird in einem weiteren Schritt ein Programm erzeugt, das die Suche nach einer Widerlegung speziell für F vornimmt.

Die vorliegende Arbeit beschäftigt sich mit der Theorie und Implementierung dieser Verfahren und setzt Grundkenntnisse der symbolischen Logik voraus. Ausführliche Darstellungen der in dieser Arbeit verwendeten theoretischen Grundlagen finden sich in [Andrews, 1986], [Fitting, 1990] und [Menzel & Schmitt, 1991]. Die Formalisierung der Theorie der Shannon-Graphen orientiert sich an [Posegga, 1992].

Die Arbeit ist wie folgt gegliedert:

Im zweiten Kapitel wird auf die Theorie der aussagenlogischen Shannon-Graphen eingegangen und ein Beweisverfahren für diese vorgestellt. Es wird gezeigt, wie dieses Verfahren mittels einer bestimmten Übersetzungstechnik effizient implementiert werden

kann. Abschließend werden Ergebnisse der Implementierung eines aussagenlogischen Beweisers für eine Reihe von bekannten Testproblemen angegeben.

Das dritte Kapitel behandelt zunächst die Theorie der prädikatenlogischen Shannon-Graphen und stellt zwei darauf basierende Beweisverfahren vor. Insbesondere auf das zweite, verbesserte Verfahren der *selektiven Erweiterungen* wird dabei ausführlich eingegangen und gezeigt, daß dieses korrekt und vollständig ist. Im letzten Abschnitt des Kapitels wird beschrieben, wie sich das Verfahren der selektiven Erweiterungen realisieren läßt.

Die Implementierung des im Rahmen dieser Arbeit entstandenen prädikatenlogischen Beweisers $\mathcal{SHARE}$ wird im vierten Kapitel beschrieben. Außerdem sind in diesem Kapitel die Ergebnisse für einige bekannte prädikatenlogische Testprobleme zusammengestellt.

Der Anhang enthält eine kurze Anleitung zur Benutzung von $\mathcal{SHARE}$ und einen Beispiellauf für Pelletiers 24. Problem.

Kapitel 2

Aussagenlogische Shannon-Graphen

2.1 Einleitung

Am Anfang des vorgestellten Beweisverfahrens steht die Konvertierung einer logischen Formel in eine äquivalente Darstellung als Shannon-Graph. Shannon-Graphen stellen eine kompakte Repräsentation von logischen Formeln dar, in denen außer dem dreistelligen Junktor "sh" keine weiteren Junktoren auftreten. Im ersten Teil dieses Kapitels beschäftigen wir uns mit den grundlegenden Eigenschaften solcher Formeln und wie sich diese Darstellung aus einer beliebig gegebenen aussagenlogischen Formel gewinnen läßt. Wir geben dann ein auf Shannon-Graphen basierendes Beweisverfahren an und zeigen wie dieses effizient implementiert werden kann.

Mit Hilfe einer Erweiterung der Shannon-Graphen, den CUT-Shannon-Graphen, wird der Zusammenhang zwischen semantischen Tableaux und Shannon-Graphen beleuchtet. Schließlich betrachten wir verschiedene Konstruktionsvarianten für den Aufbau von Shannon-Graphen und geben die damit erzielten Ergebnisse für eine Reihe von bekannten Testproblemen an.

2.2 Theoretische Grundlagen

2.2.1 Notationen

Wir erweitern die übliche Definition der Sprache der Aussagenlogik zur Sprache $\mathcal{A}$, indem wir neben den Junktoren $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ den dreistelligen Shannon-Junktor sh zulassen. Außerdem nehmen wir die Atome $\mathbf{0}$ und $\mathbf{1}$ mit in die Signatur auf.

$\mathbf{For}_{At}$ bezeichnet die Menge der atomaren Formeln ohne $\{\mathbf{0}, \mathbf{1}\}$, $\mathbf{For}$ die Menge aller Formeln über $\mathbf{For}_{At}$ ohne Verwendung des Junktors sh und $\mathbf{For}_{\mathcal{SH}}$ steht schließlich für die Menge aller wohlgeformten Formeln von $\mathcal{A}$.

Definition 2.1 (Semantik von $\mathbf{0}, \mathbf{1}, sh$)
Seien $A, B, C \in \mathbf{For}_{\mathcal{SH}}$. Die Definition des Wahrheitswertes val : $\mathbf{For}_{\mathcal{SH}} \to \{\mathsf{F}, \mathsf{W}\}$ wird wie folgt auf Formeln aus $\mathbf{For}_{\mathcal{SH}}$ erweitert:

$$\begin{array}{lcl} val(\mathbf{0}) & := & \mathsf{F} \\ val(\mathbf{1}) & := & \mathsf{W} \\ val(sh(A,B,C)) & := & val(\neg A \wedge B \vee A \wedge C) \end{array}$$

val hängt von der gewählten Interpretation I, oder, im prädikatenlogischen Fall, von der Struktur $\mathcal{D}$ mit Variablenbelegung β ab.

A heißt die Bedingung, *B der* negative, *C der* positive Ausgang *von $sh(A,B,C)$. Für $val(A) = val(B)$ schreiben wir abkürzend $A \Leftrightarrow B$.*

Da die Disjunktion in $\neg A \wedge B \vee A \wedge C$ eine exklusive Lesart gestattet, können wir die Formel $sh(A,B,C)$ intuitiv lesen als

"Wenn A gilt, dann C, sonst B."

Der sh-Junktor ist also ein *if-then-else* Junktor, bei dem die Argumente 2 und 3 vertauscht sind.[1]

Wie sich noch zeigen wird, bildet der sh-Junktor zusammen mit $\mathbf{0}$ und $\mathbf{1}$ eine logische Basis,[2] d.h. wir können auf alle anderen Junktoren $\neg, \wedge, \vee, \ldots$ verzichten und nur noch Formeln über $\mathbf{0}$, $\mathbf{1}$ und sh betrachten. Dies gilt selbst dann noch, wenn wir für die Bedingung A nur atomare Formeln zulassen.[3] Die Menge $\mathcal{SH}$ der so aufgebauten Shannon-Formeln definieren wir formal:

Definition 2.2 (Shannon-Formeln, Shannon-Graphen)
Die Menge der Shannon-Formeln $\mathcal{SH} \subset \mathbf{For}_{\mathcal{SH}}$ ist definiert als die kleinste Menge, für die gilt

(1) $\mathbf{0}, \mathbf{1} \in \mathcal{SH}$

(2) Wenn $\mathcal{S}_0, \mathcal{S}_1 \in \mathcal{SH}$ und $A \in \mathbf{For}_{At}$, dann ist auch $sh(A, \mathcal{S}_0, \mathcal{S}_1) \in \mathcal{SH}$.

*Da wir Shannon-Formeln als Graphen repräsentieren (siehe Abschnitt 2.3.1), sprechen wir häufig synonym von Shannon-*Graphen. *Shannon-Formeln bezeichnen wir mit kalligraphischen Buchstaben.*

2.2.2 Darstellung von Shannon-Formeln in Form von Binärbäumen

Shannon-Formeln lassen sich in unmittelbar einsichtiger Weise als Binärbäume auffassen. Dabei bilden die atomaren Wahrheitswerte $\mathbf{0}$ und $\mathbf{1}$ die *Terminal-Knoten* oder *Blätter*, während ein Term $sh(A, \mathcal{B}, \mathcal{C})$ mit der Bedingung A und den Ausgängen $\mathcal{B}$ und $\mathcal{C}$ als *Nichtterminal-Knoten* mit dem Inhalt A und den Unterbäumen $\mathcal{B}$ und $\mathcal{C}$ dargestellt werden kann. Wenn wir gelegentlich von der Wurzel, Blättern und Nichtterminal-Knoten einer Shannon-Formel reden, so beziehen wir uns auf ihre Darstellung als Binärbaum.

[1] Bei der Darstellung von *BDDs* wird üblicherweise der negative Ausgang von A links neben dem positiven Ausgang dargestellt. Die beim sh-Junktor gewählte Anordnung trägt dem Rechnung.

[2] Dies zeigt z.B. [Church, 1956], wobei er anstelle des sh-Junktors die von ihm als *conditioned disjunction* bezeichnete Konstruktion $[A_1, B, A_0]$ mit der Semantik $[A_0, B, A_1] :\Leftrightarrow (B \to A_1) \wedge (\neg B \to A_0)$ verwendet.

[3] Siehe auch [Bauer & Wirsing, 1991, 82ff,99ff], dort heißen solche Formeln "*dyadische Fallunterscheidungen in Prämissen-Normalform*".

Beispiel 2.3
In Bild 2.1 ist die Shannon-Formel $sh(a, \mathbf{0}, sh(b, \mathbf{0}, \mathbf{1}))$ als Binärbaum $\mathcal{S}_1$ dargestellt. Jeder Nichtterminal-Knoten ist mit einer atomaren Formel (der Bedingung) versehen und hat eine negative und eine positive Kante zu den entsprechenden Unterbäumen.

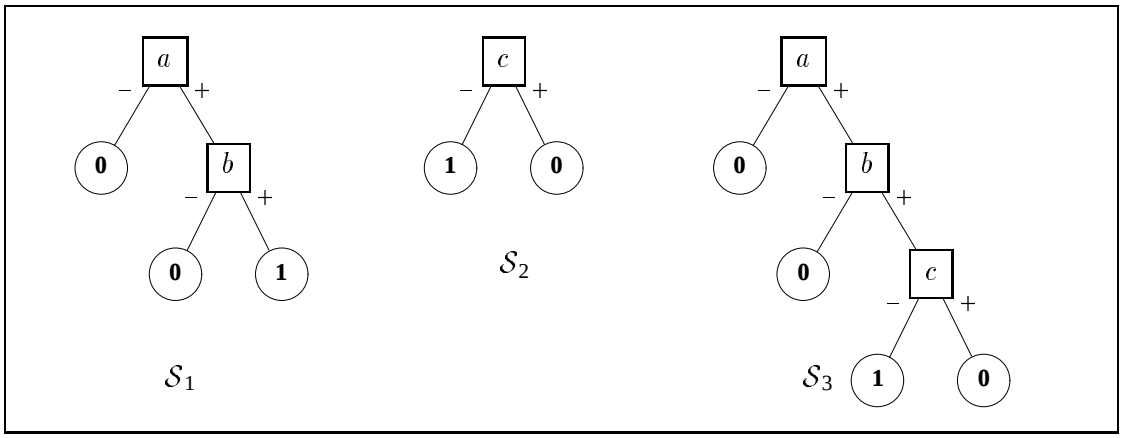

Abbildung 2.1: Shannon-Formeln für $a \wedge b$, $\neg c$, $a \wedge b \wedge \neg c$ in Binärbaum-Darstellung

2.2.3 Auswertung von Shannon-Formeln

Der Wahrheitswert einer Shannon-Formel unter einer gegebenen Interpretation I kann sehr einfach bestimmt werden:

Beginnend von der Wurzel verzweigen wir an jedem Nichtterminal-Knoten entsprechend dem angenommenen Wahrheitswert der Bedingung, bis wir schließlich ein Blatt erreichen, das den Wahrheitswert der Shannon-Formel unter I repräsentiert. In Abbildung 2.1 erhalten wir den Wahrheitswert W für $\mathcal{S}_1$ unter der Interpretation $I = \{a, b\}$. Jede andere Interpretation von a oder b macht $\mathcal{S}_1$ dagegen falsch; $\mathcal{S}_1$ ist demnach eine äquivalente Darstellung von $a \wedge b$.

2.2.4 Komposition von Shannon-Formeln

Um zwei Shannon-Formeln $\mathcal{A}$ und $\mathcal{B}$ konjunktiv zu verknüpfen, können wir anstelle der **1**-Blätter in $\mathcal{A}$ die Shannon-Formel $\mathcal{B}$ "einsetzen"; wir schreiben dann $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$: Eine Interpretation I macht $\mathcal{A}$ wahr, wenn wir beim Durchlaufen von $\mathcal{A}$ gemäß I in einem **1**-Blatt enden. Wenn dies auch für $\mathcal{B}$ gilt, dann und nur dann gilt $\mathcal{A} \wedge \mathcal{B}$.

Beispiel: Wir betrachten nochmals Abbildung 2.1. Durch Einsetzung von $\mathcal{S}_2$ für den **1**-Knoten in $\mathcal{S}_1$ ergibt sich $\mathcal{S}_3$, d.h. $\mathcal{S}_3 = \mathcal{S}_1[\frac{\mathbf{1}}{\mathcal{S}_2}]$. Es gilt $\mathcal{S}_3 \Leftrightarrow \mathcal{S}_1 \wedge \mathcal{S}_2$.

Nachfolgend formalisieren wir den Begriff der *Substitution von Blättern* und geben weitere Kompositionsregeln für Shannon-Formeln an.[4]

[4] In Kapitel 3 betrachten wir auch Shannon-Formeln, die in der Bedingung selbst wieder Shannon-Formeln enthalten. Die Definition der Blatt-Substitution stellt sicher, daß Ersetzungen nur innerhalb des zweiten und dritten Arguments einer Shannon-Formel vorgenommen werden. Die Schreibweise dieser Substitution ist an [Orłowska, 1969b] angelehnt.

Definition 2.4 (Substitution von Blättern)
Für $\mathcal{A}, \mathcal{S}_0, \mathcal{S}_1 \in \mathcal{SH}$ definieren wir

$$\mathcal{A}[\frac{\mathbf{0}}{\mathcal{S}_0}, \frac{\mathbf{1}}{\mathcal{S}_1}] = \begin{cases} \mathcal{S}_0 & \text{wenn } \mathcal{A} = \mathbf{0} \\ \mathcal{S}_1 & \text{wenn } \mathcal{A} = \mathbf{1} \\ sh(A, \mathcal{B}[\frac{\mathbf{0}}{\mathcal{S}_0}, \frac{\mathbf{1}}{\mathcal{S}_1}], \mathcal{C}[\frac{\mathbf{0}}{\mathcal{S}_0}, \frac{\mathbf{1}}{\mathcal{S}_1}]) & \text{wenn } \mathcal{A} = sh(A, \mathcal{B}, \mathcal{C}) \end{cases}$$

$\mathcal{A}[\frac{\mathbf{0}}{\mathcal{S}_0}]$ bzw. $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{S}_1}]$ stehen abkürzend für den Fall, daß nur **0** *bzw.* **1***-Blätter ersetzt werden sollen.*

Um schrittweise alle "herkömmlichen" Junktoren $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ einer Formel zu eliminieren und durch den sh-Junktor zu ersetzen, benötigen wir die nachfolgenden *Rechenregeln*, die sich unmittelbar aus Definition 2.1 ergeben.

Lemma 2.5 (Rechenregeln)
Seien $A, \ldots, D \in \mathbf{For}_{\mathcal{SH}}$, $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ ein zweistelliger Junktor, dann gilt

1. $\neg sh(A, B, C) \Leftrightarrow sh(A, \neg B, \neg C)$ *Negations-Elimination*
2. $sh(A, B, C) \circ D \Leftrightarrow sh(A, B \circ D, C \circ D)$ *Distributivität*

Durch eine einfache strukturelle Induktion und mit Hilfe der genannten Rechenregeln zeigt man die Gültigkeit der nachfolgenden *Kompositionsregeln*. Man beachte, daß zum Beweis von $(REP_\vee)$ und $(REP_\wedge)$ nur die jeweilige Distributivität benötigt wird, während alle anderen Kompositionsregeln von der Negations-Elimination Gebrauch machen.

Lemma 2.6 (Kompositionsregeln)
Seien $\mathcal{A}, \mathcal{B} \in \mathcal{SH}$, dann gilt

$$(REP_\vee) \quad \mathcal{A} \vee \mathcal{B} \Leftrightarrow \mathcal{A}[\frac{\mathbf{0}}{\mathcal{B}}] \qquad (REP_\wedge) \quad \mathcal{A} \wedge \mathcal{B} \Leftrightarrow \mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$$

$$(REP_\neg) \quad \neg \mathcal{A} \Leftrightarrow \mathcal{A}[\frac{\mathbf{0}}{\mathbf{1}}, \frac{\mathbf{1}}{\mathbf{0}}]$$

$$(REP_\rightarrow) \quad \mathcal{A} \rightarrow \mathcal{B} \Leftrightarrow \mathcal{A}[\frac{\mathbf{0}}{\mathbf{1}}, \frac{\mathbf{1}}{\mathcal{B}}] \qquad (REP_\leftrightarrow) \quad \mathcal{A} \leftrightarrow \mathcal{B} \Leftrightarrow \mathcal{A}[\frac{\mathbf{0}}{\neg\mathcal{B}}, \frac{\mathbf{1}}{\mathcal{B}}]$$

2.2.5 Eigenschaften von Shannon-Formeln

Um die logischen Eigenschaften von Shannon-Formeln untersuchen zu können, benötigen wir den Begriff des *Pfades*. Anschaulich verstehen wir unter einem Pfad einer Shannon-Formel $\mathcal{S}$ die Menge von Literalen, die man erhält, wenn man $\mathcal{S}$ von der Wurzel beginnend bis zu einem Blatt **0** oder **1** durchläuft und die besuchten atomaren Formeln vorzeichenrichtig aufsammelt. Das bedeutet, wenn wir den negativen Ausgang eines Nichtterminal-Knotens $sh(A, \mathcal{B}, \mathcal{C})$ wählen, nehmen wir $\neg A$ zum Pfad hinzu, andernfalls A.

Beispiel: $\mathcal{S}_1$ in Abbildung 2.1 enthält die Pfade $\{\neg a\}, \{a, \neg b\}$ zu **0**-Knoten und $\{a, b\}$ zum einzigen **1**-Knoten.

Im folgenden werden wir mit jeder Shannon-Formel $\mathcal{S}$ zwei disjunktive Normalformen $DNF_{\mathbf{1}}(\mathcal{S})$ und $DNF_{\mathbf{0}}(\mathcal{S})$ assoziieren, die Disjunktion aller Pfade zu **1**-Knoten, sowie die Disjunktion aller Pfade zu **0**-Knoten.

Um eine möglichst einfache Notation zu erhalten, die auch die leere Disjunktion und die leere Konjunktion umfaßt, schreiben wir eine disjunktive Normalform als Menge Π von Mengen $\pi_1, \ldots, \pi_n$. Die π_i deuten wir als Konjunktionen, während Π als Disjunktion betrachtet wird. Den Wahrheitswert der leeren Konjunktion definieren wir wie üblich als W, den der leeren Disjunktion als F.

Gelegentlich wollen wir zwei disjunktive Normalformen Π_1 und Π_2 konjunktiv verknüpfen; dies geschieht durch "Ausmultiplizieren". Hierzu definieren wir eine Operation $\circledwedge$, die dem kartesischen Produkt ähnlich ist:

Definition 2.7 (Produkt von Mengen)
Für die Mengen von Mengen Π_1, Π_2 *ist*

$$\Pi_1 \circledwedge \Pi_2 := \{\pi_1 \cup \pi_2 \mid \pi_1 \in \Pi_1, \pi_2 \in \Pi_2\}$$

Die Vereinigung $\pi_1 \cup \pi_2$ ergibt wieder eine Konjunktion, $\Pi_1 \circledwedge \Pi_2$ eine Disjunktion solcher Konjunktionen. Wenn alle Pfade π nur Literale enthalten, ist $\Pi_1 \circledwedge \Pi_2$ also eine disjunktive Normalform. Man beachte für die nachfolgende Definition, daß $\Pi \circledwedge \{\} = \{\} \circledwedge \Pi = \{\}$, während $\Pi \circledwedge \{\,\{\}\,\} = \{\,\{\}\,\} \circledwedge \Pi = \Pi$.

Wir können jetzt Pfade einer Shannon-Formel formal beschreiben und einige damit verbundene Sprechweisen einführen:[5]

Definition 2.8 (Pfade, $DNF_{\mathbf{1}}(\mathcal{S})$, $DNF_{\mathbf{0}}(\mathcal{S})$)
Zu $\mathcal{S} \in \mathcal{SH}$ *definieren wir die Menge aller Pfade zu* **1***-Knoten* $DNF_{\mathbf{1}}(\mathcal{S})$*, sowie die Menge aller Pfade zu* **0***-Knoten,* $DNF_{\mathbf{0}}(\mathcal{S})$ *als*

$$DNF_{\mathbf{1}}(\mathcal{S}) = \begin{cases} \{\} & \text{wenn } \mathcal{S} = \mathbf{0} \\ \{\,\{\}\,\} & \text{wenn } \mathcal{S} = \mathbf{1} \\ \{\{\neg A\}\} \circledwedge DNF_{\mathbf{1}}(\mathcal{B}) \;\cup\; \{\{A\}\} \circledwedge DNF_{\mathbf{1}}(\mathcal{C}) & \text{wenn } \mathcal{S} = sh(A, \mathcal{B}, \mathcal{C}) \end{cases}$$

$$DNF_{\mathbf{0}}(\mathcal{S}) = \begin{cases} \{\,\{\}\,\} & \text{wenn } \mathcal{S} = \mathbf{0} \\ \{\} & \text{wenn } \mathcal{S} = \mathbf{1} \\ \{\{\neg A\}\} \circledwedge DNF_{\mathbf{0}}(\mathcal{B}) \;\cup\; \{\{A\}\} \circledwedge DNF_{\mathbf{0}}(\mathcal{C}) & \text{wenn } \mathcal{S} = sh(A, \mathcal{B}, \mathcal{C}) \end{cases}$$

Die Elemente von $DNF_{\mathbf{1}}(\mathcal{S})$ *nennen wir* **1**-Pfade*, die von* $DNF_{\mathbf{0}}(\mathcal{S})$ **0**-Pfade.

Definition 2.9
Ein Pfad π *heißt* geschlossen *oder* inkonsistent*, wenn er komplementäre Literale enthält; offensichtlich ist* π *dann unerfüllbar.*

Definition 2.10
Ein aussagenlogischer Shannon-Graph heißt geschlossen*, wenn alle seine* **1***-Pfade geschlossen sind.*

[5] Diese Notation weicht von der in [Posegga, 1992] verwendeten ab. Die durch die Mengenschreibweise definierten Pfade entsprechen daher nicht mehr exakt dem üblichen Begriff. So kann es verschiedene "Wege" von der Wurzel zu Blättern geben, die durch die gleiche Menge beschrieben werden. Für die weiteren Betrachtungen spielt dies jedoch keine Rolle.

Wir haben die Mengen $DNF_{\mathbf{1}}(\mathcal{S})$ und $DNF_{\mathbf{0}}(\mathcal{S})$ so definiert, daß sie die Semantik des sh-Junktors korrekt widerspiegeln. Damit ergeben sich die folgenden fundamentalen Eigenschaften:

Satz 2.11 (Eigenschaften von Shannon-Formeln)
Sei $\mathcal{S} \in \mathcal{SH}$ eine Shannon-Formel, $\pi_1, \pi_2 \in DNF_{\mathbf{1}}(\mathcal{S}) \cup DNF_{\mathbf{0}}(\mathcal{S}), \pi_1 \neq \pi_2$ zwei verschiedene Pfade, dann gilt

(E1) $\mathcal{S} \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S})$ *DNF-Darstellung*

(E2) $\neg\mathcal{S} \Leftrightarrow DNF_{\mathbf{0}}(\mathcal{S})$ *Dualität*

(E3) π_1, π_2 haben kein gemeinsames Modell *Pfad-Eindeutigkeit*

(E1) besagt, daß eine Shannon-Formel logisch äquivalent zu einer disjunktiven Normalform, der Disjunktion aller **1**-Pfade ist. Diese Eigenschaft machen wir uns zunutze, um Modelle einer Shannon-Formel $\mathcal{S}$ zu bestimmen. Jeder nicht geschlossene **1**-Pfad π repräsentiert auf naheliegende Weise ein Modell I_π von $\mathcal{S}$: Für jedes Atom A setzen wir $I_\pi(A) = \mathsf{W}$, wenn $A \in \pi$, $I_\pi(A) = \mathsf{F}$, wenn $\neg A \in \pi$, beliebig sonst. Die Darstellung einer Shannon-Formel als disjunktive Normalform hat eine Analogie beim *Tableaukalkül*: Betrachtet man die Äste eines voll expandierten Tableaus $\mathcal{T}$ als Konjunktion der darauf liegenden Literale und die Äste als disjunktiv verknüpft, so erhält man ebenfalls eine disjunktive Normalform, die äquivalent zur Ausgangsformel des Tableaus ist. Offene Äste repräsentieren dann Modelle der Ausgangsformel. Die Unterschiede zwischen Ästen im Tableau und Pfaden einer Shannon-Formel werden durch die Eigenschaften *(E2)* und *(E3)* beleuchtet.

Die Dualität *(E2)* besagt, daß eine Shannon-Formel $\mathcal{S}$ auch ihre Gegenmodelle, d.h. Modelle von $\neg\mathcal{S}$ in Form der Pfade zu **0**-Knoten repräsentiert. Dies ist charakteristisch für Shannon-Formeln und hat keine Entsprechung beim Tableau-Verfahren.

Aus Eigenschaft *(E3)* folgt schließlich, daß es in einem Shannon-Graphen zu einer Interpretation I *genau einen* Pfad von der Wurzel zu einem Blatt-Knoten gibt, der unter I erfüllt ist. Damit kann man eine Shannon-Formel unter I bequem auswerten, ohne verschiedene Alternativen betrachten zu müssen. Dagegen gibt es bei einem voll expandierten Tableau im allgemeinen Interpretationen, die mehrere Äste zugleich erfüllen: Beim Tableau für $a \vee b$ ist $I = \{a, b\}$ eine solche.

An dieser Stelle wollen wir, exemplarisch für die anderen einfachen Induktionen, den Beweis durchführen.

BEWEIS (Satz 2.11)

(E1) **:** Wir zeigen dies mittels Induktion über den Aufbau von $\mathcal{S}$:

- $\underline{\mathcal{S} = \mathbf{0}:}$ Dann ist $DNF_{\mathbf{1}}(\mathcal{S}) = \{\}$ die leere Disjunktion, deren Wahrheitswert wir als F definiert haben.
- $\underline{\mathcal{S} = \mathbf{1}:}$ $DNF_{\mathbf{1}}(\mathcal{S}) = \{\{\}\}$ ist eine Disjunktion, die die leere Konjunktion enthält, also $DNF_{\mathbf{1}}(\mathcal{S}) \Leftrightarrow \mathbf{1}$.

$\underline{\mathcal{S} = sh(A,\mathcal{B},\mathcal{C})}$**:** Sei vorausgesetzt, daß die Behauptung bereits für $\mathcal{B}$ und $\mathcal{C}$ gezeigt wurde. Dann gilt

$$\begin{array}{lll} DNF_{\mathbf{1}}(\mathcal{S}) & \Leftrightarrow \begin{array}{c}\{\{\neg A\}\} \\ \cup \\ \{\{A\}\}\end{array} \begin{array}{c}\circledwedge DNF_{\mathbf{1}}(\mathcal{B}) \\ \circledwedge DNF_{\mathbf{1}}(\mathcal{C})\end{array} & \textit{Definition 2.8} \\ & \Leftrightarrow \neg A \wedge DNF_{\mathbf{1}}(\mathcal{B}) \vee A \wedge DNF_{\mathbf{1}}(\mathcal{C}) & \textit{“Vorziehen” von } \neg A \textit{ bzw. } A \\ & \Leftrightarrow \neg A \wedge \mathcal{B} \vee A \wedge \mathcal{C} & \textit{Induktionsvoraussetzung} \\ & \Leftrightarrow \mathcal{S} & \textit{Definition 2.1} \end{array}$$

(E2) **:**

$$\begin{array}{lll} DNF_{\mathbf{0}}(\mathcal{S}) & \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S}[\frac{\mathbf{0}}{\mathbf{1}},\frac{\mathbf{1}}{\mathbf{0}}]) & \textit{Definitionen 2.4 und 2.8} \\ & \Leftrightarrow \mathcal{S}[\frac{\mathbf{0}}{\mathbf{1}},\frac{\mathbf{1}}{\mathbf{0}}] & \textit{Wie in (E1) gezeigt} \\ & \Leftrightarrow \neg\mathcal{S} & \textit{Ersetzungsregel } (REP_{\neg}) \end{array}$$

(E3) **:** Seien I_1, I_2 Interpretationen mit $I_1 \models \pi_1$ und $I_2 \models \pi_2$ (falls mindestens ein Pfad unerfüllbar ist, so ist nichts weiter zu zeigen). Da $\pi_1 \neq \pi_2$, gibt es einen kleinsten (“obersten”) Nichtterminal-Knoten $sh(A,\mathcal{B},\mathcal{C})$, in dem sich π_1 und π_2 unterschiedlich verzweigen. Daher müssen sich die Interpretationen I_1 und I_2 in A unterscheiden.

■

Als Spezialfall von *(E1)* ergibt sich sofort das

Korollar 2.12
Eine aussagenlogische Shannon-Formel $\mathcal{S}$ ist unerfüllbar gdw. $\mathcal{S}$ geschlossen ist.

BEWEIS
Wenn $\mathcal{S}$ geschlossen ist, dann ist jeder **1**-Pfad $\pi_i \in DNF_{\mathbf{1}}(\mathcal{S})$ unerfüllbar, also auch $\mathcal{S}$. Andernfalls gibt es einen nicht geschlossenen **1**-Pfad π. Da π nur aus Literalen besteht, kann direkt eine Interpretation I_π angegeben werden, die π und somit $\mathcal{S}$ erfüllt. ■

2.2.6 Umwandlung von Formeln in Shannon-Formeln

Nachdem die grundlegenden Eigenschaften von Shannon-Formeln bekannt sind, geben wir eine Transformation an, die eine beliebige aussagenlogische Formel in eine logisch äquivalente Shannon-Formel verwandelt. Damit ist dann gezeigt, daß $\{sh, \mathbf{0}, \mathbf{1}\}$ eine logische Basis bildet. Diese Transformation leistet, durch sukzessive Anwendung der Kompositionsregeln $(REP_\wedge), (REP_\vee), \ldots$ aus Lemma 2.6, die nachfolgend definierte Funktion *conv* : **For**$\rightarrow\mathcal{SH}$:

Definition 2.13 (Convert)

$$conv(F) = \begin{cases} sh(F,\mathbf{0},\mathbf{1}) & \textit{wenn } F \in \mathbf{For}_{At} \\ conv(A)[\frac{\mathbf{0}}{\mathbf{1}},\frac{\mathbf{1}}{\mathbf{0}}] & \textit{wenn } F = \neg A \\ conv(A)[\frac{\mathbf{1}}{conv(B)}] & \textit{wenn } F = A \wedge B \\ conv(A)[\frac{\mathbf{0}}{conv(B)}] & \textit{wenn } F = A \vee B \\ conv(A)[\frac{\mathbf{0}}{\mathbf{1}},\frac{\mathbf{1}}{conv(B)}] & \textit{wenn } F = A \rightarrow B \\ conv(A)[\frac{\mathbf{0}}{conv(\neg B)},\frac{\mathbf{1}}{conv(B)}] & \textit{wenn } F = A \leftrightarrow B \end{cases}$$

Durch einfache strukturelle Induktion und mit Hilfe des Lemmas 2.6 zeigt man direkt das folgende Korollar:

Korollar 2.14
Sei $F \in$ **For**, dann gilt: $F \Leftrightarrow \mathit{conv}(F)$.

2.2.7 Ein Beweisverfahren für aussagenlogische Shannon-Formeln

In den vorangegangenen Abschnitten haben wir die notwendigen Grundlagen geschaffen, um eine auf Shannon-Formeln basierende Beweisprozedur für die Aussagenlogik anzugeben. Aufgabe einer solchen Prozedur ist es, für eine gegebene aussagenlogischen Formel alle Modelle zu ermitteln. Zuweilen genügt es auch festzustellen, ob eine Formel überhaupt ein Modell hat.

Mit Hilfe der Funktion *conv* sind wir in der Lage, eine beliebige aussagenlogische Formel in eine logisch äquivalente Shannon-Formel umzuwandeln. Nach Satz 2.11 ist eine Shannon-Formel $\mathcal{S}$ wiederum logisch äquivalent zu $\mathit{DNF}_{\mathbf{1}}(\mathcal{S})$, der Disjunktion aller Pfade zu **1**-Knoten.

Um die Modelle von $\mathcal{S}$ zu bestimmen, müssen geschlossene und somit unerfüllbare **1**-Pfade offensichtlich nicht berücksichtigt werden. Es genügt demnach die Menge $\mathit{DNF}_{\mathbf{1}}^{\mathrm{CONS}}(\mathcal{S})$ der *konsistenten* **1**-Pfade von $\mathcal{S}$ zu bestimmen. Da in Pfaden nur Literale vorkommen, repräsentiert jeder Pfad $\pi \in \mathit{DNF}_{\mathbf{1}}^{\mathrm{CONS}}(\mathcal{S})$ eine partielle Interpretation I_π, die $\mathcal{S}$ erfüllt. Die Eigenschaft der *Pfad-Eindeutigkeit* aus Satz 2.11 stellt zudem sicher, daß alle I_π verschieden sind.

Ein mögliches Vorgehen wäre, sukzessive alle Pfade von $\mathit{DNF}_{\mathbf{1}}(\mathcal{S})$ aufzuzählen und dabei inkonsistente Pfade zu eliminieren. Dieser Ansatz ist jedoch in der Praxis unbrauchbar[6] und man wird sinnvollerweise bereits inkonsistente *Teilpfade* verwerfen. Dadurch können in einem einzigen Schritt eine ganze Reihe inkonsistenter Pfade eliminiert werden.

Die Beweisprozedur stellt sich wie folgt dar: In einem *Vorverarbeitungsschritt* transformieren wir eine aussagenlogische Formel F in einen Shannon-Graphen $\mathcal{S}$. Um eine $\mathcal{S}$ erfüllende Interpretation I_π zu finden, durchlaufen wir $\mathcal{S}$ mittels Tiefensuche, wobei wir inkonsistente Teilpfade verwerfen; dies stellt die eigentliche *Beweissuche* dar. Man betrachte hierzu Algorithmus 1 in Abbildung 2.2.

Bei Aufruf der Funktion $\mathit{satisfy}(\mathcal{A}, \pi)$ repräsentiert der Pfad π eine partielle Interpretation I_π. Nachfolgend liefert $\mathit{satisfy}(\mathcal{A}, \pi)$ alle mit I_π verträglichen Modelle von $\mathcal{A}$:

Im ersten Basisfall wird die leere Disjunktion zurückgegeben, da Pfade, die in **0** enden, keine Modelle von $\mathcal{A}$ repräsentieren. Dagegen stellt, falls wir an einem **1**-Knoten angelangt sind, π ein Modell von $\mathcal{A}$ dar und wir liefern diesen konsistenten Pfad zurück. Im Rekursionsfall wird eine einmal getroffene Entscheidung für den Wahrheitswert einer atomaren Formel At beibehalten, d.h. entsprechend in den negativen oder positiven Nachfolger von $sh(At, \mathcal{B}, \mathcal{C})$ verzweigt: Wenn $\neg At \in \pi$, sind alle Pfade durch den positiven Ausgang $\mathcal{C}$ geschlossen, falls dagegen $At \in \pi$, so sind alle Pfade durch $\mathcal{B}$ geschlossen. Ist At jedoch noch nicht in π enthalten, d.h. noch nicht mit einem Wahrheitswert belegt, werden nacheinander beide möglichen Belegungen betrachtet und die gefundenen Lösungen vereinigt. Ist man höchstens an *einem* Modell interessiert, etwa um die Inkonsistenz von F zu zeigen, so kann man auf die Vereinigung verzichten und nur eine der möglichen Lösungen zurückliefern.

[6] Die Shannon-Formel für *Pigeon-7* (siehe Abschnitt 2.6.3, Tabelle 2.3) hat $\approx 2.4 \cdot 10^{43}$ Pfade zu **1**-Knoten!

Algorithmus 1

$\mathcal{S}$:= *conv*(F);
$DNF_{\mathbf{1}}^{\text{CONS}}(\mathcal{S})$:= *satisfy*$(\mathcal{S}, \{\})$.

```
function satisfy(A, π)
 case A of
  0: return {}
  1: return {π}
  sh(At, B, C): if ¬At ∈ π return satisfy(B, π)
                else if At ∈ π return satisfy(C, π)
                else return satisfy(B, {¬At} ∪ π) ∪ satisfy(C, {At} ∪ π)
```

Abbildung 2.2: Die aussagenlogische Beweisprozedur

2.3 Aspekte der Implementierung

In den folgenden Abschnitten beschreiben wir, wie die vorgestellte Beweisprozedur effizient realisiert werden kann. Eine wichtige Voraussetzung für das gewählte Verfahren ist die kompakte Darstellung von Shannon-Formeln in Form von Graphen. In Abschnitt 2.3.2 geben wir eine Prolog-Implementierung der Funktion *conv* an, die die gewünschte Graphdarstellung erzeugt. Abschnitt 2.3.3 behandelt die Realisierung der eigentlichen Beweissuche.

2.3.1 Repräsentation von Shannon-Formeln, Shannon-*Graphen*

Bisher haben wir synonym von Shannon-*Formeln* und Shannon-*Graphen* gesprochen. Erstere Bezeichnung hebt hervor, daß es sich um gewöhnliche logische Formeln handelt. Zuweilen kann jedoch die übliche Term-Darstellung recht unhandlich werden.[7] Deshalb stellen wir gemeinsame Unterstrukturen nur einmal dar (“*structure sharing*”), d.h. wir repräsentieren Shannon-Formeln als gerichtete, azyklische Graphen, im weiteren gelegentlich auch *DAGs*[8] genannt. Durch diese Darstellung erreichen wir insbesondere, daß in einem Shannon-Graphen nur zwei Terminal-Knoten **0** und **1** auftreten.

Beispiel 2.15
Abbildung 2.3 zeigt verschiedene Darstellungen der Shannon-Formel

$$sh(a, sh(c, \mathbf{0}, \mathbf{1}), sh(b, sh(c, \mathbf{0}, \mathbf{1}), \mathbf{1})) = \textit{conv}((a \wedge b) \vee c).$$

$\widehat{\mathcal{S}}$ ist die bereits bekannte Darstellung als Binärbaum. $\mathcal{S}$ ist die abstrakte Repräsentation von $\widehat{\mathcal{S}}$ in Form eines gerichteten, azyklischen Graphen. Diesen wiederum stellen wir

[7] Die bereits erwähnte Shannon-Formel für *Pigeon-7* hat als Binärbaum dargestellt über 10^{43} Knoten; die Graph-Darstellung kommt dagegen mit 294 Knoten aus.
[8] *D*irected *A*cylic *G*raphs

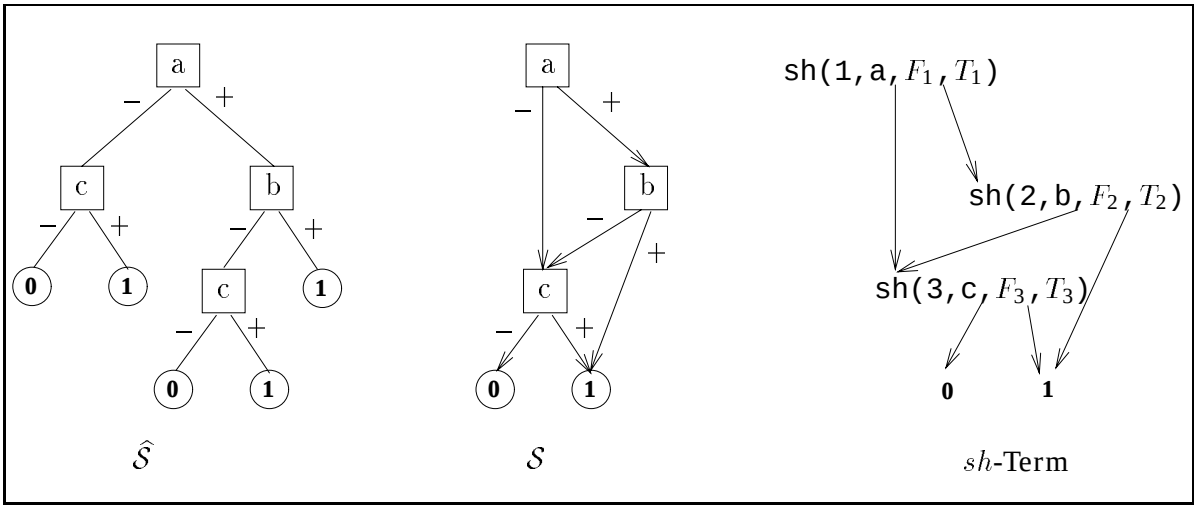

Abbildung 2.3: Verschiedene Repräsentationen der Shannon-Formel für $(a \wedge b) \vee c$

in der gewählten Implementierungssprache Prolog, in Form des rechts abgebildeten sh-Terms dar.

Wir repräsentieren demnach einen Nichtterminal-Knoten $sh(A, \mathcal{F}, \mathcal{T})$ eines Shannon-Graphen durch einen Term der Form

sh(*Id*,*A*,F,T)

Id ist eine natürliche Zahl und bezeichnet in eindeutiger Weise den jeweiligen Nichtterminal-Knoten. Diese *Knoten-Bezeichner* werden im weiteren Verlauf des Verfahrens benötigt. *A* ist die zu diesem Knoten gehörende atomare Formel. Die negative bzw. positive Kante zu den Untergraphen $\mathcal{F}$ und $\mathcal{T}$ stellen wir durch Prolog-Variablen F und T dar, die nach der Prolog-Unifikation $F = \mathcal{F}, T = \mathcal{T}$ die gewünschten Verweise enthalten.

Diese Darstellung als DAG läßt sich bei der Konvertierung einer Formel mittels *conv* sehr einfach gewinnen, wie wir im nächsten Abschnitt zeigen.

2.3.2 Realisierung von *conv*

Die Implementierung der Funktion *conv* aus Definition 2.13 soll zum einen effizient sein, zum anderen soll die erhaltene Repräsentation der Shannon-Formel nach Möglichkeit nicht mehr Speicherplatz benötigen als die ursprüngliche Formel. Diesen Anforderungen genügt die folgende Realisierung von *conv* durch das Prolog-Prädikat

conv(+*F*, -$\mathcal{S}$,-*False*,-*True*)

Ein- bzw. Ausgabeparameter sind, wie üblich, durch "+" und "-" gekennzeichnet. Der zur gegebenen Formel F äquivalente Shannon-Graph $\mathcal{S}$ wird, zusammen mit den Prolog-Variablen *False* und *True* zurückgegeben. In dem Term $\mathcal{S}$ treten anstelle der Blätter **0**

und **1** die Variablen *False* und *True* auf, die als Platzhalter für nachfolgende Einsetzungen dienen.[9] Durch Prolog-Unifikation dieser Variablen mit Termdarstellungen anderer Shannon-Graphen entsteht das gewünschte *structure sharing*. Um diesen Vorgang wiederholt anwenden zu können, müssen die Blätter einer Unterstruktur explizit als Ausgabeparameter zurückgeliefert werden.

Die Negation eines Shannon-Graphen ergibt sich nach $(REP_\neg)$ durch einfaches Vertauschen der **0** und **1**-Blätter:

```
conv(- A,ShA,False,True) :- !,[10]
     conv(A,ShA,True,False).
```

Um $conv(A \wedge B) := conv(A)[\frac{\mathbf{1}}{conv(B)}]$ zu bestimmen, berechnen wir zunächst rekursiv den Shannon-Graphen *ShA* von *A*, wobei wir anstelle der **1**-Blätter die noch ungebundene Prolog-Variable *ShB* einsetzen. Diese wird erst durch den nachfolgenden Aufruf von `conv(B,ShB,False,True)` gebunden. Dagegen bleiben *False* und *True* ungebunden, sie stehen für die Blatt-Knoten **0** und **1** des Ergebnisses:

```
conv(A & B,ShA,False,True) :- !,
     conv(A,ShA,False,ShB),
     conv(B,ShB,False,True).
```

Die anderen Fälle werden analog gehandhabt. Man beachte, daß der Aufwand für die Behandlung der Äquivalenz größer ist als für die anderen Operatoren:

```
conv(A v B,ShA,False,True) :- !,
     conv(A,ShA,ShB,True),
     conv(B,ShB,False,True).

conv(A => B,ShA,False,True) :- !,
     conv(A,ShA,True,ShB),
     conv(B,ShB,False,True).

conv(A <=> B,Sh,False,True) :- !,
     conv(A,Sh,ShNB,ShB),
     conv(B,ShB,False,True),
     conv(-B,ShNB,False,True).
```

War keiner der obigen Fälle zutreffend, so liegt eine atomare Formel A vor: Der Term `sh(Id,A,False,True)` mit dem neuen Bezeichner Id muß zurückgeliefert werden:

[9] Man bezeichnet $(\mathcal{S}, False, True)$ auch als *Differenz-Struktur* (siehe [Sterling & Shapiro, 1986], [O'Keefe, 1990]), das Prolog-spezifische probate Mittel, um unvollständige Datenstrukturen zu repräsentieren.

[10] Eine Bemerkung zur Verwendung des Prolog-Cuts "!": Bei den hier verwendeten Cuts handelt es sich um sogenannte *"green cuts"* ([Sterling & Shapiro, 1986]), d.h. man erhält die gleiche Menge von Antworten, wenn man die Cuts nicht verwendet. Allerdings ist im allgemeinen die Ausführungsgeschwindigkeit bei Verwendung derselben größer, da keine Rücksetzpunkte zur Bestimmung von alternativen Lösungen mittels Backtracking auf dem Aufruf-Keller abgespeichert werden müssen. Streicht man in der letzten Klausel von *conv* die Bedingung `atomic_formula(A)`, so werden die Cuts *"rot"*, und jede Formel, die nicht von der Bauart der zuvor abgehandelten Fälle ist, wird als atomar betrachtet.

```
conv(A, sh(Id,A,False,True),False,True) :-
     atomic_formula(A),!,
     ctr_inc(1,1,Id).
```

Wie bereits in Korollar 2.14 erwähnt, gilt $F \Leftrightarrow conv(F)$. Die vorliegende Realisierung von *conv* ist durch die Vewendung gemeinsamer Unterstrukturen zudem effizient, d.h. es gilt

Satz 2.16
Für jede aussagenlogische Formel $F \in$ **For**, die "$\leftrightarrow$" nicht enthält ist:

(1) $|conv(F)| = |F|$

Dabei bezeichnet $|F|$ die Länge der Eingabeformel (gemessen in Anzahl von atomaren Teilformeln) und $|conv(F)|$ die Anzahl der Nichtterminal-Knoten des Shannon-Graphen. Dies ist leicht einzusehen: Durch die Verwendung gemeinsamer Unterstrukturen ist gewährleistet, daß bei Komposition von $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$ gilt: $|\mathcal{A}[\frac{1}{\mathcal{B}}]| = |\mathcal{A}| + |\mathcal{B}|$. Ohne Verwendung von *structure sharing* hätten wir dagegen $|\mathcal{A}[\frac{1}{\mathcal{B}}]| = |\mathcal{A}| \cdot |\mathcal{B}|$.

Da für eine atomare Formel genau ein Nichtterminal-Knoten erzeugt wird, folgt die Behauptung für die Größe von $conv(F)$.

(2) $conv \in \mathcal{O}(|F|)$

Die zeitliche Komplexität von *conv* ist proportional zur Länge der Eingabeformel. Auch dies ist offenkundig, wenn man bedenkt, daß die Komposition zweier Shannon-Graphen durch das Prädikat `conv/4` auf die Prolog-Unifikation einer Variablen mit einem Term zurückgeführt wird, d.h. die Komposition ist eine Operation mit Aufwand $\mathcal{O}(1)$.

Die Aussagen (1) und (2) gelten – zumindest in der Aussagenlogik – auch für Formeln, die $\leftrightarrow$ enthalten, wenn wir Shannon-Graphen etwas anders definieren: Danach dürfen Kanten auch mit einem "Inverter"-Symbol "$\bullet$" versehen sein (siehe z.B. [Brace *et al.*, 1990] Abschnitt 5.1). Man setzt:

$$conv(A \leftrightarrow B) := conv(A)[\frac{\mathbf{0}}{\bullet conv(B)}, \frac{\mathbf{1}}{conv(B)}]$$

und erhält eine besonders kompakte Darstellung der Äquivalenz, da $conv(B)$ sowohl B als auch (via "$\bullet$") $\neg B$ darstellt. In dem von uns gewählten Beweisverfahren würde dadurch der Aufwand für die Vorverarbeitung reduziert, allerdings auf Kosten der nachfolgenden Beweissuche; denn während dieser muß die implizite Negation "$\bullet$" aufgelöst werden. Aus diesem Grunde wurde auf eine entsprechende Implementierung verzichtet.

2.3.3 Realisierung von *satisfy*, Kompilierung von Shannon-Graphen

Wir wollen nun daran gehen, die eigentliche Beweissuche des Verfahrens aus Abbildung 2.2 möglichst effizient zu implementieren. Um zu beweisen, daß eine aussagenlogische Formel F unerfüllbar ist, müssen wir zeigen, daß im zugehörigen Shannon-Graphen $\mathcal{S} = conv(F)$ alle **1**-Pfade geschlossen sind. Finden wir jedoch einen nicht geschlossenen **1**-Pfad, so ist F erfüllbar. Die Suche nach erfüllbaren Pfaden könnten wir gemäß der Funktion *satisfy* auf einer expliziten Datenstruktur für $\mathcal{S}$ durchführen. Wir gehen jedoch

einen anderen Weg, der zwar den Aufwand für die Vorverarbeitung erhöht, dafür aber eine deutliche Effizienzsteigerung bei der Beweissuche bewirkt.

Die zugrundeliegende Idee ist, einen Shannon-Graphen $\mathcal{S}$ in ein Prolog-Programm zu "kompilieren", d.h. wir generieren für $\mathcal{S}$ ein Prolog-Programm $P_{\mathcal{S}}$, das die Suche nach erfüllbaren Pfaden speziell für $\mathcal{S}$ löst. Auf die Anfrage `?-` *satisfy*(π) berechnet $P_{\mathcal{S}}$ sukzessive alle konsistenten **1**-Pfade π von $\mathcal{S}$. Gibt es keine solchen, so scheitert die Anfrage und $\mathcal{S}$ ist als unerfüllbar erkannt. Durch dieses Vorgehen wird die Suche nach konsistenten **1**-Pfaden direkt auf den Prolog-Inferenzmechanismus abgebildet. Das Vorgehen ist wie folgt:

In einem weiteren Vorverarbeitungsschritt, der *Kodegenerierung* erzeugen wir für jeden Nichtterminal-Knoten `sh(`*Id*`,`$A,\mathcal{B},\mathcal{C}$`)` von $\mathcal{S}$ mit Bezeichner *Id genau eine* Prolog-Klausel mit dem Kopf `node(`Id,π`)`. Wir erhalten so ein Prolog-Programm, dessen Größe proportional zur Länge der Eingabeformel ist.[11] π repräsentiert dabei einen teilweise instantiierten Pfad. `node(`Id,π`)` hat Erfolg, wenn es eine konsistente Erweiterung von π zu **1**-Knoten von $\mathcal{B}$ oder $\mathcal{C}$ gibt. Durch eine solche Erweiterung wird π im allgemeinen zusätzlich instantiiert.

Pfade π können wir in Prolog als Terme `path(`$A_1,\ldots,A_n$`)` fester Stelligkeit repräsentieren, denn nach der Vorverarbeitung mit *conv* ist uns die Anzahl n der Atome der Eingabeformel bekannt. Mit jeder Stelle i in `path/n` assoziieren wir ein bestimmtes Atom A_i. Ist die Stelle i noch nicht instantiiert, so gilt $A_i, \neg A_i \notin \pi$, d.h. wir haben A_i noch nicht interpretiert, andernfalls gibt uns die Instanz an, ob $A_i \in \pi$ oder $\neg A_i \in \pi$, d.h. ob wir A_i als `W` oder `F` annehmen.

Die Struktur der generierten Klausel läßt sich durch eine Abwandlung des Rekursionsfalls der Funktion *satisfy* herleiten. Als Beispiel betrachten wir dazu speziell den Knoten `sh(1,a,sh(3,...),sh(2,...))` aus Abbildung 2.3 (`2` und `3` sind die Knoten-Bezeichner der Nachfolgerknoten). Wenn wir das Atom `a` an der ersten Stelle von `path/3` darstellen, erhalten wir die folgende Prolog-Klausel:

```
node(1,π) :-   arg(1,π,-a), node(3,π)
             ; arg(1,π,+a), node(2,π).
```

Das Semikolon "`;`" stellt in Prolog eine Disjunktion dar. Die umgangssprachliche Bedeutung dieser Klausel ist die folgende:

"Ein konsistente Verlängerung des Teilpfades π durch den Knoten `1` *zu einem* **1***-Pfad enthält $\neg a$ und führt durch den Knoten* `3`*, oder enthält a, führt dann aber durch den Knoten* `2`*."*

In Abhängigkeit der Instantiierung von π ergibt sich konkret folgendes Verhalten:

$\pi =$ `path(-a,...,... )`

Dann gilt `arg(1,`π`,-a)`, d.h. $\neg a \in \pi$ und es wird einfach der negative Nachfolger `node(3,`π`)` aufgerufen. Das Ziel `arg(1,`π`,+a)` in der zweiten Zeile scheitert, der Pfad $\pi \cup a$ durch den positiven Ausgang des vorliegenden Knotens ist geschlossen.

$\pi =$ `path(+a,...,... )`

`arg(1,`π`,-a)` scheitert, d.h. der negative Ausgang ist wegen $a \in \pi$ geschlossen, und es wird der positive Nachfolger `node(2,`π`)` aufgerufen.

[11] Die bereits erwähnte Ausnahme bildet die Äquivalenz $\leftrightarrow$.

π = `path(X,...,... )`

Wenn an der ersten Stelle des Pfades eine Prolog-Variable steht, so heißt dies, daß weder a noch $\neg a$ in π sind. Durch `arg(1,`π`,-a)` wird dann a zunächst mit dem Wahrheitswert `F` belegt und versucht den so erweiterten Teilpfad $\pi \cup \neg a$ zu einem **1**-Pfad durch den negativen Ausgang zu vervollständigen. Sollte dies nicht erfolgreich sein, oder wird durch Backtracking eine zusätzliche Lösung gesucht, so wird die ursprüngliche Instantiierung der ersten Stelle von π zurückgenommen und statt dessen mit `+a` instantiiert. Durch Aufruf des positiven Nachfolgers wird schließlich eine konsistente Erweiterung von $\pi \cup a$ gesucht.

Bringt man am Ende der ersten Zeile einen Prolog-Cut "!" an, wird höchstens ein konsistenter **1**-Pfad geliefert.[12]

In der Klausel für *satisfy*(π) muß die Prolog-Klausel für den Wurzel-Knoten des Shannon-Graphen $\mathcal{S}$ mit einem leeren Pfad aufgerufen werden. Abbildung 2.4 zeigt das vollständige Programm für das Beispiel aus Abbildung 2.3. Es liefert auf die Anfrage `?-`*satisfy*(π) nacheinander π = `path`$(-a,*,+c)$; π = `path`$(+a,+b,*)$; π = `path`$(+a,-b,+c)$. Dabei steht $*$ für eine beliebige Belegung des entsprechenden Atoms.

```
satisfy(Path) :- Path = path(_,_,_), node(1,Path).

node(1,Path) :-  arg(1,Path,-a), node(3,Path)
                 ; arg(1,Path,+a), node(2,Path).
node(2,Path) :-  arg(2,Path,-b), node(3,Path)
                 ; arg(2,Path,+b).
node(3,Path) :-  arg(3,Path,+c).
```

Abbildung 2.4: Prolog-Programm für *conv*$((a \wedge b) \vee c)$

Wie in Tabelle 2.3 (Seite 35) dargestellt, ergeben sich sehr kurze Laufzeiten für den erzeugten Prolog-Kode.[13]

Die Verwendung von Prolog als Zielsprache für den beschriebenen Übersetzungsprozeß ist nicht obligatorisch. Da die Struktur des generierten Kodes einfach ist, kann man ohne größeren Aufwand andere Zielsprachen verwenden. Dies wurde am Beispiel von C und Assembler untersucht.

Generierung von C-Kode

Um einen Nichtterminal-Knoten `sh(`*Id*`,`$A,\mathcal{B},\mathcal{C}$`)` zu kompilieren, benötigen wir neben dem Bezeichner *Id* und dem Atom *A* lediglich noch Bezeichner der Nachfolger-Knoten,

[12] Vergl. die Schlußbemerkung in 2.2.7

[13] Auf einer *SUN-SPARC-2* unter *Quintus-Prolog 3.0* können damit ungefähr 100.000 Knoten pro Sekunde besucht werden.

d.h. der Wurzeln von $\mathcal{B}$ und $\mathcal{C}$. Aus diesen Informationen erzeugen wir eine der obigen Prolog-Klausel funktional entsprechende C-Funktion (Anhang A.1 enthält dazu ein Beispiel).

Die Belegungen der aussagenlogischen Atome speichern wir in globalen Variablen. Dies reicht aus, da "lokale" Belegungen indirekt über den Laufzeit-Keller zurückgewonnen werden. Funktionsaufrufe in C sind jedoch bekanntermaßen "teuer", da u.a. lokale Variablen auf den Keller gebracht werden müssen. In unserem Fall genügt es allerdings, nur die Rücksprungadresse auf dem Keller abzulegen, weshalb man einen hohen Durchsatz erwarten durfte. Voraussetzung dabei ist, daß der verwendete C-Compiler eine solche "*stack frame*-Optimierung" erlaubt.

Die experimentellen Ergebnisse konnten die Erwartungen nicht ganz erfüllen. Zum einen lagen die Zeiten für die Übersetzung des generierten C-Programms über denen für das entsprechende Prolog-Programm, zum anderen waren die Laufzeiten für die eigentliche Beweissuche fast gleich. Die Ergebnisse sind in Tabelle 2.3 dargestellt.

Generierung von Assembler-Kode

Begibt man sich dagegen durch Generierung von Assembler-Kode noch eine Stufe tiefer in Richtung Hardware, ergeben sich andere Resultate. Sehr kurze Übersetzungszeiten, die durch die direkte Erzeugung von Maschinen-Kode nochmals reduziert werden könnten, und eine Verkürzung der Laufzeit für die Beweissuche etwa um den Faktor 30, waren problemlos möglich.[14] Die Arbeitsweise des generierten Assembler-Programms entspricht, bis auf einige kleine Optimierungen, der des C-Programms. Im Anhang A.2 ist der generierte Assembler-Kode für das Beispiel aus Abbildung 2.4 dargestellt.

2.4 CUT-Shannon-Graphen

Eine charakteristische Eigenschaft von Shannon-Graphen, wie sie bisher definiert waren, ist, daß verschiedene Pfade π_1 und π_2 keine gemeinsamen Modelle haben (Satz 2.11, (E3)), also — sofern sie konsistent sind — sich ausschließende Interpretationen darstellen. Dies liegt daran, daß bei der Suche nach Modellen in einem Shannon-Graphen $\mathcal{S}$ wir uns an jedem Nichtterminal-Knoten *entweder* für die Gültigkeit des zugehörigen Atoms A, *oder* für die Gültigkeit von $\neg A$ entscheiden müssen. Wie kommt nun diese Struktur zustande?

Der Schlüssel zur Beantwortung dieser Frage liegt bei der Behandlung der Disjunktion. Eine Formel $A \vee B$ wird mittels *conv* in einen Shannon-Graphen konvertiert, indem zunächst $\mathcal{A} := \mathit{conv}(A)$ und $\mathcal{B} := \mathit{conv}(B)$ bestimmt werden und anschließend alle **0**-Knoten in $\mathcal{A}$ durch $\mathcal{B}$ ersetzt werden. Die potentiellen Modelle des so erhaltenen Shannon-Graphen $\mathcal{A}[\frac{\mathbf{0}}{\mathcal{B}}]$ sind durch dessen **1**-Pfade gegeben. Wir erreichen einen **1**-Knoten in $\mathcal{A}[\frac{\mathbf{0}}{\mathcal{B}}]$ entweder, indem wir einen **1**-Knoten von $\mathcal{A}$ erreichen, oder, indem wir über einen **0**-Knoten von $\mathcal{A}$ einen **1**-Knoten von $\mathcal{B}$ erreichen. Formal ausgedrückt gilt

$$DNF_{\mathbf{1}}(\mathcal{A}[\tfrac{\mathbf{0}}{\mathcal{B}}]) = DNF_{\mathbf{1}}(\mathcal{A}) \cup DNF_{\mathbf{0}}(\mathcal{A}) \otimes DNF_{\mathbf{1}}(\mathcal{B}).$$

[14] Allerdings wurden die Zeiten auf einer anderen Hardware, einem *Intel i486*, anstelle einer *SUN-SPARC-2*, gemessen. Die Größenordnung des Gewinns sollte jedoch in etwa gleich bleiben.

Die **0**-Pfade in $DNF_{\mathbf{0}}(\mathcal{A})$ stellen gerade die potentiellen Modelle von $\neg\mathcal{A}$ (Satz 2.11, (E2) Dualität), also auch von $\neg A$ dar. Somit zerfallen die bei einer Shannon-Graph-Traversierung betrachteten Modelle von $A \vee B$ in zwei disjunkte Klassen: Solche, in denen A gilt, und solche, in denen $\neg A \wedge B$ gilt.

Häufig ist diese Vorgehensweise günstig. In einem später noch näher präzisierten Sinn entspricht die Behandlung der Disjunktion der Lemma-Regel

$$\begin{array}{c} A \vee B \\ \hline A \mid \neg A \\ \quad\; \mid\; B \end{array}$$

beim Tableaukalkül.[15]

Andererseits gibt es Fälle, bei denen die Behandlung der Disjunktion in der oben genannten Art und Weise keine Vorteile bringt. Anhand eines einfachen Beispiels soll dies erläutert werden.[16]

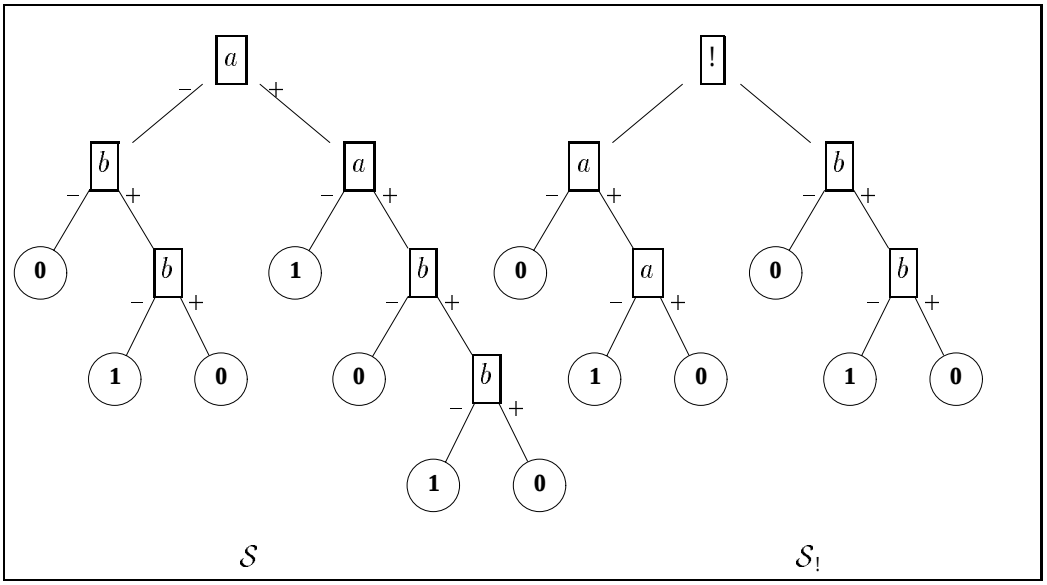

Abbildung 2.5: Shannon-Graph $\mathcal{S}$ *vs.* Shannon-Graph mit CUT $\mathcal{S}_!$

Beispiel 2.17
Abbildung 2.5 zeigt den Shannon-Graphen $\mathcal{S} = \mathit{conv}(a \wedge \neg a \vee b \wedge \neg b)$. Die Unerfüllbarkeit von $\mathcal{S}$ kann durch Schließen der drei **1**-Pfade nachgewiesen werden. Günstiger ist jedoch die mit $\mathcal{S}_!$ bezeichnete Darstellung, bei der die Disjunktion durch den mit "!" bezeichneten "CUT-Knoten" aufgelöst wird. Nun können die durch den CUT getrennten Fälle wie bei einem Tableau für $a \wedge \neg a \vee b \wedge \neg b$ unabhängig voneinander betrachtet werden, und es genügt, zwei **1**-Pfade abzuschließen.

[15] In [D'Agostino, 1992] werden die Vorteile der Lemma-Generierung aufgezeigt: D'Agostino gibt eine Klasse von Problemen an, für die die Länge der Beweise im Tableaukalkül ohne Lemma-Regel exponentiell ist verglichen mit der Länge von Beweisen im Tableaukalkül mit Lemma-Regel; siehe auch 2.6.2.

[16] Siehe [Posegga, 1992] für eine Begründung im Falle der Prädikatenlogik.

Ein anderes Beispiel, bei dem sich die "Abtrennung" zweier sich gegenseitig ausschließender Fälle als vorteilhaft erweist, ist die Äquivalenz $A \leftrightarrow B$. Analog zum Vorgehen beim Tableaukalkül können durch Einführung eines entsprechenden CUT-Knotens die Fälle $A \wedge B$ und $\neg A \wedge \neg B$ getrennt behandelt werden.

Die Erweiterung um CUT-Knoten läßt sich problemlos in die Theorie der Shannon-Graphen aufnehmen. Es wird sich zeigen, daß CUT-Shannon-Graphen als voll expandierte Tableaux aufgefaßt werden können, die man durch Anwendung einer bestimmten Tableau-Konstruktionsregel $\mathcal{R}$ erhält.

2.4.1 CUT-Shannon-Graphen *vs.* Tableaux

Definition 2.18 (Shannon-Formeln mit CUT, $\mathcal{SH}_!$)
Wir erweitern die Menge aller Shannon-Formeln $\mathcal{SH}$ zur Menge der Shannon-Formeln mit CUT *$\mathcal{SH}_!$, wie folgt:*

(1) $\mathcal{SH}_! \supset \mathcal{SH}$

(2) Wenn $\mathcal{S}_0, \mathcal{S}_1 \in \mathcal{SH}_!$, dann ist auch $sh(!, \mathcal{S}_0, \mathcal{S}_1) \in \mathcal{SH}_!$.

Man könnte anstelle des Ausdrucks $sh(!, \mathcal{A}, \mathcal{B})$ mit dem dreistelligen sh-Junktor und "!" auch die Disjunktion $\mathcal{A} \vee \mathcal{B}$ schreiben, oder einen zweistelligen Junktor $sh^!(\mathcal{A}, \mathcal{B})$ verwenden. Allerdings fügt sich die von uns gewählte "uniforme" Schreibweise besser ins Gesamtkonzept ein.[17]

Wir müssen nun noch die in $\mathcal{SH}_!$ neu hinzugekommenen Ausdrücke mit einer Semantik versehen und die Definition von $DNF_{\mathbf{1}}$ entsprechend erweitern, so daß wir auch Shannon-Formeln mit CUT als disjunktive Normalform auffassen können.

Definition 2.19 (Semantik von CUT-Shannon-Formeln, 1-Pfade in CUT-Shannon-Formeln)
Für $\mathcal{A}, \mathcal{B} \in \mathcal{SH}_!$ definieren wir

$$val(sh(!, \mathcal{A}, \mathcal{B})) \quad := \quad val(\mathcal{A} \vee \mathcal{B})$$

Wir erweitern die Definition der **1***-Pfade auf Shannon-Formeln mit* CUT*, wie folgt*

$$DNF_{\mathbf{1}}(sh(!, \mathcal{A}, \mathcal{B})) \quad := \quad DNF_{\mathbf{1}}(\mathcal{A}) \cup DNF_{\mathbf{1}}(\mathcal{B})$$

Wie man leicht nachprüft, gilt Eigenschaft *(E1)*: $\mathcal{S} \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S})$ auch für Shannon-Formeln mit CUT. Die Vereinigung der beiden disjunktiven Normalformen $DNF_{\mathbf{1}}(\mathcal{A})$ und $DNF_{\mathbf{1}}(\mathcal{B})$ entspricht gerade der Disjunktion aller **1**-Pfade aus $DNF_{\mathbf{1}}(\mathcal{A})$ und $DNF_{\mathbf{1}}(\mathcal{B})$.

Für CUT-Shannon-Formeln gelten jedoch einige andere Aussagen über Shannon-Formeln aus $\mathcal{SH}$ *nicht* mehr. Insbesondere die Rechenregel zur *Negations-Elimination*

$$\neg sh(A, B, C) \Leftrightarrow sh(A, \neg B, \neg C)$$

[17] Eine andere Deutung des CUT erhält man, wenn man diesen mit einem sonst in der Formel nirgends auftauchenden Atom identifiziert. Die entsprechende Konvertierungsfunktion vorausgesetzt, erhält man so zu einer gegebenen Formel F zwar eine *erfüllbarkeitsäquivalente*, jedoch keine *logisch äquivalente* Shannon-Formel $\mathcal{F}$ (siehe [Posegga, 1992]).

aus Lemma 2.5 ist nicht mehr gültig; denn

$$\neg sh(!, B, C) \Leftrightarrow \neg(B \vee C) \not\Leftrightarrow sh(!, \neg B, \neg C)$$

Als Folge hiervon gelten sowohl die Kompositionsregeln, die von der Negations-Elimination Gebrauch machen: $(REP_\neg)$, $(REP_\rightarrow)$, $(REP_\leftarrow)$, als auch die Eigenschaften *(E2) (Dualität)* und *(E3) (Pfad-Eindeutigkeit)*, für Shannon-Graphen mit CUT ebenfalls nicht mehr.[18]

Dagegen bleibt die Distributivität $sh(!, B, C) \circ D \Leftrightarrow sh(!, B \circ D, C \circ D)$ aus Lemma 2.5 für $\circ \in \{\wedge, \vee\}$ auch für CUT-Shannon-Graphen gültig. Daraus folgt, wie man durch Induktion leicht nachprüft, daß auch die Kompositionsregeln $(REP_\wedge)$, $(REP_\vee)$ weiterhin gelten.

Um eine beliebige Formel in eine logisch äquivalente Formel aus $\mathcal{SH}_!$ zu transformieren, müssen wir also die Negation auf andere Weise eliminieren. Dies erreichen wir durch Bildung einer Negations-Normalform mit Hilfe der bekannten Umformungsregeln wie etwa der Regeln von *de Morgan*. Diese Umformungen lassen sich wie nachfolgend gezeigt in die entsprechende Konvertierungsfunktion integrieren.

Eine Disjunktion kann nun auf zwei verschiedene Arten aufgelöst werden: Einerseits mit der Ersetzungsregel

$$(REP_\vee) : \mathcal{A} \vee \mathcal{B} \Leftrightarrow \mathcal{A}[\frac{\mathbf{0}}{\mathcal{B}}]$$

oder, analog zur Verzweigung der β-Regel im Tableaukalkül, durch den neu eingeführten CUT

$$\mathcal{A} \vee \mathcal{B} \Leftrightarrow sh(!, \mathcal{A}, \mathcal{B})$$

Wenn wir uns entscheiden, Disjunktionen *immer* in letzterer Form zu behandeln, dann erhalten wir Shannon-Graphen, die, wie sich zeigen wird, als Tableaux aufgefaßt werden können. Die Funktion $conv_{Tab} : \mathbf{For} \rightarrow \mathcal{SH}_!$ leistet das Gewünschte:[19]

Definition 2.20

$$conv_{Tab}(F) = \begin{cases} sh(A, \mathbf{0}, \mathbf{1}) & \text{wenn } F = A, A \in \mathbf{For}_{At} \\ sh(A, \mathbf{1}, \mathbf{0}) & \text{wenn } F = \neg A, A \in \mathbf{For}_{At} \\ conv_{Tab}(A) & \text{wenn } F = \neg\neg A \\ conv_{Tab}(\neg A \vee \neg B) & \text{wenn } F = \neg(A \wedge B) \\ conv_{Tab}(\neg A \wedge \neg B) & \text{wenn } F = \neg(A \vee B) \\ conv_{Tab}(A \leftrightarrow \neg B) & \text{wenn } F = \neg(A \leftrightarrow B) \\ conv_{Tab}(A)[\frac{\mathbf{1}}{conv_{Tab}(B)}] & \text{wenn } F = A \wedge B \\ sh(!, conv_{Tab}(A), conv_{Tab}(B)) & \text{wenn } F = A \vee B \\ sh(!, conv_{Tab}(A \wedge B), conv_{Tab}(\neg A \wedge \neg B)) & \text{wenn } F = A \leftrightarrow B \end{cases}$$

Man beachte, daß nunmehr **0**-Knoten während der Konvertierung nicht mehr ersetzt werden. Um die Beziehung zwischen $conv_{Tab}(F)$ und dem zugehörigen Tableau zu verdeutlichen, betrachten wir die Darstellung von $conv_{Tab}(a \wedge (b \vee c))$ als Binärbaum in Abbildung 2.6.

[18] Dies ist der Grund, warum wir die Definition von **0**-Pfaden nicht auf Shannon-Graphen mit CUT erweitert haben.

[19] Damit die Definition nicht zu unübersichtlich wird, wurden die Regeln für $\rightarrow$ weggelassen; sie können auf naheliegende Weise hinzugefügt werden.

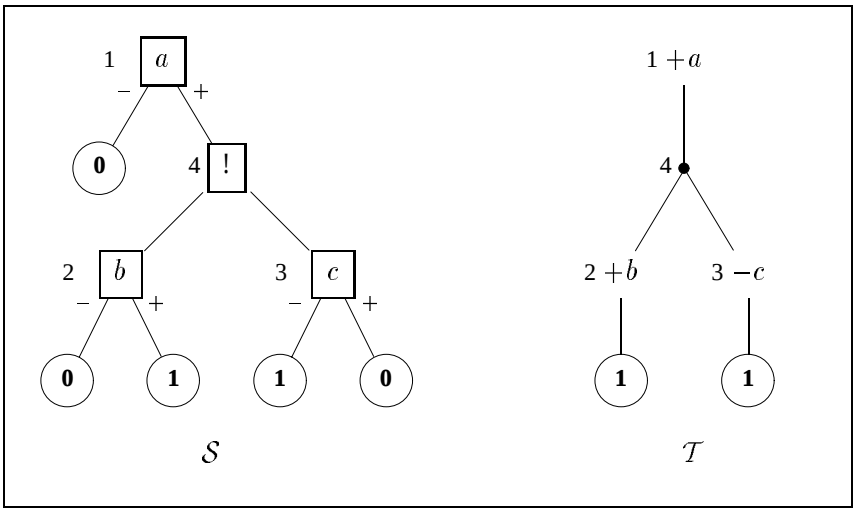

Abbildung 2.6: Shannon-Graph $\mathcal{S} = conv_{Tab}(a \wedge (b \vee \neg c))$ und Tableau $\mathcal{T}$

Aus der Definition von $conv_{Tab}$ folgt, daß jeder Literal-Knoten $sh(A, \ldots, \ldots)$ genau einen Nachfolger-Knoten **0** hat. Wir streichen diese **0** und erhalten so einen Knoten mit nur einem Nachfolger. Die atomare Formel dieses Knotens versehen wir mit dem Vorzeichen der verbleibenden Kante. Schreiben wir nun noch den CUT-Knoten als "gewöhnliche" Verzweigung, so ergibt sich der in Abbildung 2.6 dargestellte Shannon-Graph $\mathcal{T}$, der die Struktur eines voll expandierten Tableaus besitzt.[20] Die **1**-Knoten von $\mathcal{T}$ markieren dabei das Ende der Tableau-Äste.

Als Nachweis, daß $\mathcal{T}$ auch als Tableau mit Hilfe der üblichen Tableauregeln konstruiert werden kann, geben wir hierfür eine Tableau-Konstruktionsregel $\mathcal{R}$ an. Das Vorgehen gemäß $\mathcal{R}$ kann mit der Definition von $conv_{Tab}$ identifiziert werden. Daher sprechen wir auch von der Tableau-Konstruktionsregel $conv_{Tab}$ und nennen die damit konstruierten Shannon-Graphen einfach Tableaux.

Definition 2.21 (Tableau-Konstruktionsregel $\mathcal{R}$)
$Tab_{\mathcal{R}}(F)$ bezeichne das mit $\mathcal{R}$ konstruierte Tableau für die Formel F. Wir konstruieren $Tab_{\mathcal{R}}(F)$ wie folgt

- $F = \neg\neg F'$

  Wir ersetzen *F durch F', erweitern den Ast um $Tab_{\mathcal{R}}(F')$ und streichen anschließend F'.*

- $F = \beta$

  Wir ersetzen *zunächst β durch die Verzweigung $\beta_1|\beta_2$. Nach Konstruktion von $Tab_{\mathcal{R}}(\beta_1)$ unterhalb von β_1 streichen wir β_1. Anschließend verfahren wir ebenso mit β_2.*

- $F = \alpha$

  Wir ersetzen *α durch $\begin{smallmatrix}\alpha_1\\ \alpha_2\end{smallmatrix}$ und erweitern diesen Ast mit $Tab_{\mathcal{R}}(\alpha_1)$. α_1 wird gestrichen. An das Ende jeden Astes aus $Tab_{\mathcal{R}}(\alpha_1)$ schreiben wir $Tab_{\mathcal{R}}(\alpha_2)$ und streichen anschließend α_2.*

[20] Die übliche Tableau-Notation erhält man durch Weglassen des Vorzeichens "+" und Ersetzen von "–" durch "¬".

- $F = A$ *oder* $F = \neg A$*, wobei* A *atomar ist*
  Wir lassen F stehen.

Betrachtet man die Definition von $conv_{Tab}$, so kann man unschwer mit jedem darin auftauchenden Fall die zugehörige Tableau-Regel assoziieren.

Insbesondere entspricht der Konjunktionsauflösung $conv_{Tab}(A)[\frac{\mathbf{1}}{conv_{Tab}(B)}]$ die Erweiterung der Äste von $Tab_{\mathcal{R}}(A)$ mit $Tab_{\mathcal{R}}(B)$ und der Disjunktionsauflösung mit CUT-Knoten die Verzweigung gemäß der β-Regel im Tableaukalkül.

2.4.2 Shannon-Graphen ohne CUT *vs.* Tableaux mit Lemmata

Im vorigen Abschnitt haben wir gezeigt, daß mittels $conv_{Tab}$ konstruierte CUT-Shannon-Graphen strukturell voll expandierten Tableaux entsprechen. Wie hängen nun aber Shannon-Graphen aus $\mathcal{SH}$, d.h. ohne CUT, die wir mit *conv* aus Abschnitt 2.2.6 konstruiert haben, und Tableaux zusammen? Insbesondere die *Dualitätseigenschaft (E2)* aus Satz 2.11, die besagt, daß ein Shannon-Graph $\mathcal{F}$ auch Gegenmodelle einer Formel F in Form der konsistenten Pfade zu **0**-Knoten darstellt, ist bei Tableaux, allein schon wegen der fehlenden **0**-Knoten, nicht gegeben.

Wie sich zeigen wird, entsprechen den **1**-Pfaden eines Shannon-Graphen ohne CUT die Äste eines voll expandierten Tableaus mit Lemmata, d.h. während der Beweissuche werden in beiden Fällen dieselben partiellen Modelle betrachtet.[21] Die Rolle der **0**-Pfade in einem Shannon-Graphen übernehmen im Tableau gerade die Lemmata.

Um diesen Zusammenhang zu untersuchen, benötigen wir eine einfache Notation für voll expandierte Tableaux. Da CUT-Shannon-Graphen als Tableaux gedeutet werden können, liegt es nahe diesen Formalismus beizubehalten und Tableaux mit Lemmata als CUT-Shannon-Graphen darzustellen.

Nachfolgend definierte Funktion $conv_{TL}$: **For**$\rightarrow \mathcal{SH}_!$ (*convert für* T*ableaux mit* L*emmata*) liefert einen CUT-Shannon-Graphen, der strukturell einem voll expandierten Tableau $\mathcal{T}$ entspricht. Dabei kann $\mathcal{T}$ unter ausschließlicher Verwendung der üblichen Tableau-Regeln und, anstelle der β-Regel für $A \vee B$, mit Hilfe der Lemma-Regel

$$\frac{A \vee B}{\neg A \wedge B \mid A}$$

erzeugt werden.[22] Die zugehörige Tableau-Konstruktionsregel erhält man durch eine naheliegende Modifikation der Definition 2.21 von $\mathcal{R}$. Wir sprechen deshalb auch vom Tableau (mit Lemmata) $conv_{TL}(F)$.

Definition 2.22
*$conv_{TL}$: **For**$\rightarrow \mathcal{SH}_!$ ist definiert wie $conv_{Tab}$ bis auf die Auflösung der Disjunktion, die analog zur Lemma-Regel des Tableaukalküls definiert wird:*

$$conv_{TL}(A \vee B) := sh(!, conv_{TL}(\neg A \wedge B), conv_{TL}(A))$$

[21] In [Posegga, 1992] wurde bereits gezeigt, daß die Anzahl der **1**-Pfade eines Shannon-Graphen mit der Anzahl der Äste eines voll expandierten Tableaus mit Lemmata übereinstimmt.

[22] Um den nachfolgenden Vergleich zu Shannon-Graphen zu erleichtern, ist die Anordnung der Äste $\neg A \wedge B$ und A, gegenüber der sonst bei Tableaux üblichen vertauscht, d.h. wir betrachten zunächst das Lemma $\neg A$.

Das mit $conv_{TL}$ konstruierte Tableau mit Lemmata enthält die gleiche Menge von **1**-Pfaden wie der mit *conv* gebildete Shannon-Graph *ohne* CUT. Dies ist die Aussage des folgenden Satzes. Allerdings unterscheiden sich beide, wie wir noch sehen werden, in ihrer *Struktur*.

Satz 2.23 (Shannon-Graphen *vs.* **Tableaux mit Lemmata)**
Sei F eine aussagenlogische Formel über der Basis $\{\neg, \wedge, \vee\}$, $\mathcal{S} = conv(F)$ ein Shannon-Graph *ohne* CUT, $\mathcal{T} = conv_{TL}(F)$ ein Tableau mit Lemmata, dann gilt

$$DNF_{\mathbf{1}}(\mathcal{S}) = DNF_{\mathbf{1}}(\mathcal{T}).$$

Beispiel 2.24
Wir wollen zunächst die Konstruktion mittels $conv_{TL}$ für die Formel $F := (a \wedge b \wedge c) \vee d$ skizzieren. Die Schreibweise

$$\frac{conv_{TL}(\beta)}{conv_{TL}(\beta_1) \mid conv_{TL}(\beta_2)}$$

bedeutet, daß wir, um das Tableau für β zu bestimmen, β durch eine Verzweigung und die Tableaux für β_1 und β_2 *ersetzen*. Zur Vereinfachung betrachten wir nur die Auflösung der Disjunktionen:

$$\frac{conv_{TL}((a \wedge b \wedge c) \vee d)}{\begin{array}{l|l} conv_{TL}(\neg a \vee \neg b \vee \neg c) & conv_{TL}(a \wedge b \wedge c) \\ conv_{TL}(d) & \end{array}}$$

mit Hilfe der Lemma-Regel und dem Lemma $\neg a \vee \neg b \vee \neg c$.

$$\frac{conv_{TL}(\neg a \vee \neg b \vee \neg c)}{\begin{array}{l|l} conv_{TL}(a) & conv_{TL}(\neg a) \\ conv_{TL}(\neg b \vee \neg c) & \end{array}}$$

mit Lemma a; schließlich noch:

$$\frac{conv_{TL}(\neg b \vee \neg c)}{\begin{array}{l|l} conv_{TL}(b) & conv_{TL}(\neg b) \\ conv_{TL}(\neg c) & \end{array}}$$

Abbildung 2.7 zeigt den Shannon-Graphen $\mathcal{S} := conv((a \wedge b \wedge c) \vee d)$ und das mit $conv_{TL}$ konstruierte Tableau mit Lemmata $\mathcal{T} := conv_{TL}((a \wedge b \wedge c) \vee d)$. Zwar sind die Mengen der **1**-Pfade für beide Darstellungen gleich, jedoch erweist sich die Shannon-Graph-Struktur als kompakter:

Zum einen kann $\mathcal{S}$ mit nur vier Nichtterminal-Knoten als DAG dargestellt werden, was bei $\mathcal{T}$ nicht möglich ist; zum anderen können wir $\mathcal{S}$ disjunktiv mit einem weiteren Shannon-Graphen $\mathcal{S}'$ verknüpfen $\mathcal{S}[\frac{\mathbf{0}}{\mathcal{S}'}]$, wobei wir das "Lemma" $\neg\mathcal{S}$ in Form der **0**-Pfade bereits zur Verfügung haben.

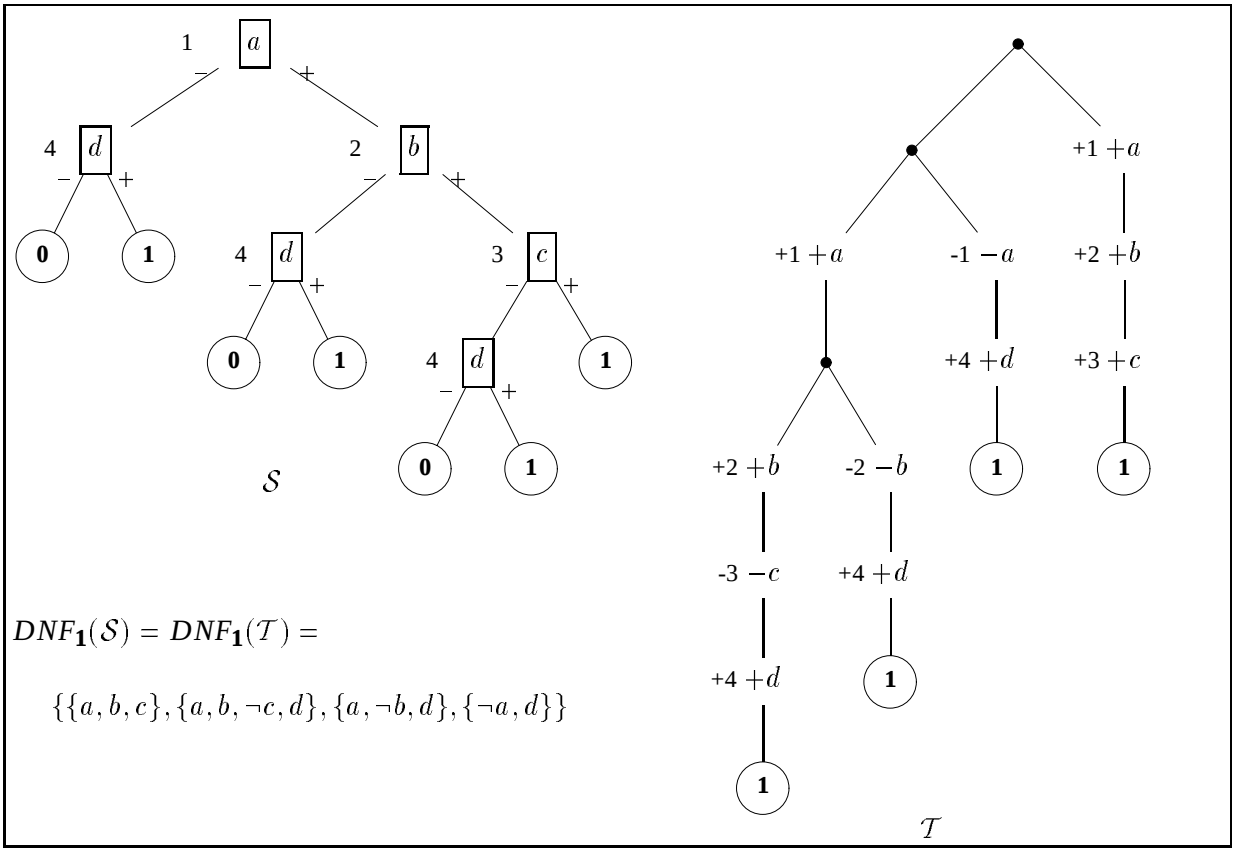

Abbildung 2.7: Shannon-Graph $\mathcal{S}$ *vs.* Tableau mit Lemmata $\mathcal{T}$

BEWEIS (Satz 2.23)
Um den Beweis möglichst einfach zu gestalten, verwenden wir eine alternative Formulierung für *conv* aus Definition 2.13, bei der wir die Negation nicht mittels $(REP_\neg)$ auflösen, sondern wie im Falle von $conv_{Tab}$ "nach innen ziehen". Die so definierte Funktion *conv′* liefert für Formeln über $\neg, \wedge, \vee$ exakt die gleichen Shannon-Formeln wie die ursprünglichen Definition von *conv*.[23]

Die genannte Formulierung vorausgesetzt, erhalten wir Definitionen von *conv* und $conv_{Tab}$, die sich strukturell nur in der Behandlung der Disjunktion unterscheiden. Wir zeigen nun die Behauptung durch Induktion über den Formelaufbau von F, wobei wir annehmen, daß diese bereits für die direkten Unterformeln von F und deren Negationen gezeigt wurde:

$F \in$ **For**$_{At}$**:** Die Definitionen von *conv* und $conv_{TL}$ sind für atomare Formeln identisch.

F **ist zusammengesetzt:** Da sich die Definitionen von *conv* und $conv_{TL}$ nur für $F = A \vee B$ unterscheiden, ist für alle anderen Fälle die Induktion bereits abgeschlossen.

Sei nun also $F = A \vee B$ und die Behauptung für A und B bereits gezeigt. Es folgt

$$DNF_{\mathbf{1}}(conv(A \vee B)) \tag{2.1}$$
$$= DNF_{\mathbf{1}}(conv(A)[\tfrac{\mathbf{0}}{conv(B)}]) \tag{2.2}$$

[23] Wir nehmen an, daß $\rightarrow$ und $\leftrightarrow$ bereits nach den üblichen Regeln aufgelöst wurden.

$$
\begin{aligned}
&= DNF_{\mathbf{1}}(conv(A)) \cup DNF_{\mathbf{0}}(conv(A)) \circledwedge DNF_{\mathbf{1}}(conv(B)) && (2.3)\\
&= DNF_{\mathbf{1}}(conv(A)) \cup DNF_{\mathbf{1}}(conv(\neg A)) \circledwedge DNF_{\mathbf{1}}(conv(B)) && (2.4)\\
&= DNF_{\mathbf{1}}(conv_{TL}(A)) \cup DNF_{\mathbf{1}}(conv_{TL}(\neg A)) \circledwedge DNF_{\mathbf{1}}(conv_{TL}(B)) && (2.5)\\
&= DNF_{\mathbf{1}}(conv_{TL}(A)) \cup DNF_{\mathbf{1}}(conv_{TL}(\neg A \wedge B)) && (2.6)\\
&= DNF_{\mathbf{1}}(sh(!, conv_{TL}(\neg A \wedge B), conv_{TL}(A))) && (2.7)\\
&= DNF_{\mathbf{1}}(conv_{TL}(A \vee B)) && (2.8)
\end{aligned}
$$

(2.2) und (2.3) ergeben sich direkt aus den Definitionen von *conv*, $DNF_{\mathbf{1}}$ bzw. $DNF_{\mathbf{0}}$ und der Definition der Blatt-Substitution $\mathcal{A}[\frac{\mathbf{0}}{\mathcal{B}}]$.

(2.4) gilt wegen der leicht zu zeigenden Eigenschaft von *conv*: $conv(\neg A) = conv(A)[\frac{\mathbf{0}}{\mathbf{1}}, \frac{\mathbf{1}}{\mathbf{0}}]$.

In (2.5) geht die Induktionshypothese ein, während (2.6) bis (2.8) wieder unmittelbar auf die jeweiligen Definitionen zurückzuführen sind.

Da wir in der Induktionsannahme vorausgesetzt hatten, daß die Behauptung für alle direkten Unterformeln und deren Negation bereits bewiesen wurde, müssen wir nun noch zeigen, daß die Behauptung auch für die Negation $\neg(A \vee B)$ von F gilt. Da sowohl *conv* als auch $conv_{TL}$ die Definition von $\neg(A \vee B)$ auf $\neg A \wedge \neg B$ zurückführen, "trägt" hier wieder die Induktionsvoraussetzung, und wir sind fertig. ■

2.4.3 Implementierung von CUT-Shannon-Graphen

Die Beweisprozedur aus Abbildung 2.2 für Shannon-Graphen ohne CUT muß um die Behandlung von CUT-Knoten erweitert werden. Aus Definition 2.19 ($DNF_{\mathbf{1}}(sh(!, \mathcal{B}, \mathcal{C}))$) erkennt man unmittelbar, was zu tun ist: An einem CUT-Knoten erweitern wir den aktuellen Pfad π *nicht*, sondern geben einfach die Disjunktion der **1**-Pfade von $\mathcal{B}$ und $\mathcal{C}$ zurück.

Abbildung 2.8 zeigt die so erweiterte Beweisprozedur. Dabei ist *conv*$_!$ eine Variante von *conv*, die CUT-Shannon-Graphen konstruiert. Im nächsten Abschnitt werden wir verschieden Varianten untersuchen.

Die Kodegenerierung für CUT-Shannon-Graphen gestaltet sich ebenfalls recht einfach. Bei der Übersetzung eines CUT-Knotens

```
sh(Id,!,sh(Id0,...),sh(Id1,...))
```

generieren wir eine Prolog-Klausel

```
node(Id,π) :- node(Id0,π)
            ; node(Id1,π).
```

die ihre Nachfolger-Klauseln mit unverändertem Pfad π aufruft. Bis auf diesen zusätzlichen Fall ändert sich die Implementierung des Beweisverfahrens nicht. Da wir mittels $conv_{Tab}$ "gewöhnliche" Tableaux erzeugen können, haben wir damit "ganz nebenbei" eine Implementierung eines Tableau-Beweisers geschaffen, die auf der beschriebenen Übersetzungstechnik basiert.

Algorithmus 2

$\mathcal{S} := \mathit{conv}_!(F)$;
$DNF_{\mathbf{1}}^{\text{CONS}}(\mathcal{S}) := \mathit{satisfy}_!(\mathcal{S}, \{\})$.

```
function satisfy_!(A, π)
 case A of
  0: return {}
  1: return {π}
  sh(At, B, C): if ¬At ∈ π return satisfy_!(B, π)
                else if At ∈ π return satisfy_!(C, π)
                else return satisfy_!(B, {¬At} ∪ π) ∪ satisfy_!(C, {At} ∪ π)
  sh(!, B, C):  return satisfy_!(B, π) ∪ satisfy_!(C, π)
```

Abbildung 2.8: Die aussagenlogische Beweisprozedur für CUT-Shannon-Graphen

2.5 Varianten beim Graphaufbau mit *conv*

Im folgenden untersuchen wir verschiedene Varianten der Funktionen *conv* und $conv_{Tab}$. Bisher haben wir nicht von der Tatsache Gebrauch gemacht, daß eine Konjunktion von Shannon-Graphen auf zweierlei Arten aufgelöst werden kann:

$$\mathcal{A} \wedge \mathcal{B} \Leftrightarrow \begin{cases} \mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}] \\ \mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}] \end{cases}$$

Bei den Definitionen von *conv* und $conv_{Tab}$ hatten wir uns willkürlich auf $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$ festgelegt. Entsprechendes galt für die Disjunktion bei *conv*.

Nach Satz 2.16 ist die Größe beider Repräsentationen in Graph-Darstellung zwar gleich, d.h. $|\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]| = |\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}]|$, dies gilt jedoch nicht für die Größe der durch sie repräsentierten Binärbäume.

Wir bezeichnen im folgenden mit $\#_{\mathbf{1}}\mathcal{S}$ die Anzahl aller **1**-Knoten einer Darstellung von $\mathcal{S}$ als Binärbaum, entsprechend ist $\#_{\mathbf{0}}\mathcal{S}$ erklärt. Offensichtlich gilt[24]

$$|DNF_{\mathbf{1}}(\mathcal{S})| \leq \#_{\mathbf{1}}\mathcal{S}$$
$$|DNF_{\mathbf{0}}(\mathcal{S})| \leq \#_{\mathbf{0}}\mathcal{S}$$

Da $DNF_{\mathbf{1}}(\mathcal{S})$ die maximale Anzahl der zu untersuchenden Modelle darstellt, scheint es sinnvoll, $\#_{\mathbf{1}}\mathcal{S}$ möglichst klein zu halten. Für die Auflösung der Konjunktion $\mathcal{A}\wedge\mathcal{B}$ mittels $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$ gilt

[24] Wenn man Pfade als Folgen und nicht, wie in unserem Fall, als Mengen definiert, steht hier die Gleichheit. So aber können verschiedene inkonsistente "Wege" (Pfade im üblichen Sinne) die gleiche Mengendarstellung haben und in $DNF_{\mathbf{1}}$ bzw. $DNF_{\mathbf{0}}$ "zusammenfallen".

$$\begin{aligned} \#_{\mathbf{1}}\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}] &= \#_{\mathbf{1}}\mathcal{A} \cdot \#_{\mathbf{1}}\mathcal{B} \\ \#_{\mathbf{0}}\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}] &= \#_{\mathbf{0}}\mathcal{A} + \#_{\mathbf{1}}\mathcal{A} \cdot \#_{\mathbf{0}}\mathcal{B} \end{aligned}$$

Ersetzt man $\mathcal{A} \wedge \mathcal{B}$ jedoch durch $\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}]$, so erhält man eine andere Anzahl von **0**-Knoten, während die Anzahl der **1**-Knoten gleich bleibt. Werden in einem weiteren Schritt, etwa bei der Auflösung der Disjunktion, diese **0**-Knoten ersetzt, so hat dies Einfluß auf die Zahl der sich letztendlich ergebenden **1**-Pfade.

Wir können also eine Heuristik einsetzen, die die "bessere" (sprich: kleinere) der Alternativen $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$ und $\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}]$ auswählt. Dabei wirkt diese jedoch nur *lokal*, d.h. es ist nicht garantiert, daß der so erhaltene Baum minimal in der Anzahl der **1**-Blätter ist.

Durch das unterschiedliche "Zusammenhängen" der Shannon-Graphen verändern wir aber nicht nur die Größe des Beweissuchraumes, sondern auch die Anordnung der Literale auf den Pfaden, was sich sowohl günstig als auch ungünstig auswirken kann. Tatsächlich hat die Reihenfolge, in der man die verschiedenen Literale untersucht, wie dies auch aus anderen Beweisverfahren bekannt ist,[25] einen erheblichen Einfluß auf die Beweislänge.

Wir sollten also nicht zuviel von solchen Heuristiken erwarten. Andererseits müssen wir uns, ob wir wollen oder nicht, für eine der jeweils möglichen Kompositionsvarianten entscheiden.

Wir betrachten der Einfachheit halber nur den Fall, daß wir die Konjunktion $\mathcal{A} \wedge \mathcal{B}$ auflösen müssen. Im einzelnen wurden folgende Varianten untersucht:

- *conv*

  Dies ist die in Definition 2.13 beschriebene Variante. Wir wählen immer $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$.

- $conv_{min1st}$

  Wie *conv*, außer wenn

  $$\#_{\mathbf{0}}\mathcal{B} + \#_{\mathbf{1}}\mathcal{B} \leq \#_{\mathbf{0}}\mathcal{A} + \#_{\mathbf{1}}\mathcal{A}$$

  dann wählen wir $\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}]$. Wir untersuchen also die "einfachere" Formel zuerst.

- $conv_{min}$

  Wie *conv*, außer wenn

  $$\#_{\mathbf{0}}\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}] + \#_{\mathbf{1}}\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}] \leq \#_{\mathbf{0}}\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}] + \#_{\mathbf{1}}\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$$

  dann wählen wir $\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}]$. Wir wählen damit diejenige Komposition, die (lokal) den kleineren Ergebnisbaum erzeugt.

- $conv_{Tab}$

  Konstruiert wird ein Tableau, wir wählen immer $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$.

[25] Siehe [Harche *et al.*, 1991]

- $conv_{\alpha\_min}$

  Wie $conv_{Tab}$, außer wenn

  $$\#_{\mathbf{1}}\mathcal{B} \leq \#_{\mathbf{1}}\mathcal{A}$$

  dann konstruieren wir $\mathcal{B}[\frac{\mathbf{1}}{\mathcal{A}}]$. Wie bei $conv_{min1st}$ wird dadurch zuerst der "einfachere Fall" betrachtet.

- $conv_{L1}$

  Die zugrunde liegende Heuristik ist hier etwas komplizierter: Wir versuchen wieder einen möglichst kleinen Shannon-Graphen zu konstruieren, berücksichtigen aber jetzt die nachfolgende Operation, für die dieser Shannon-Graph benötigt wird. Wenn wir etwa eine Disjunktion $conv_{L1}(A) \vee conv_{L1}(B)$ auflösen wollen, so erscheint es günstig, vorab die Anzahl der **0**-Knoten in $\mathcal{A}$ und $\mathcal{B}$ zu minimieren, während im Falle der Konjunktion das Minimierungskriterium durch die Anzahl der **1**-Knoten gegeben ist. Wir betreiben also eine Art *1-Vorausschau* und geben den Aufrufen von $conv_{L1}(A)$ und $conv_{L1}(B)$ das entsprechende Minimierungkriterium vor. Außerdem wird die Disjunktion entweder durch Ersetzung der **0**-Blätter, oder durch Einführung eines CUT-Knotens aufgelöst, je nachdem, welche Option günstiger erscheint.

Obwohl die vorgeschlagenen Heuristiken kein optimales Ergebnis garantieren, hat sich doch gezeigt, daß damit häufig Verbesserungen möglich sind. Der Aufwand zur Berechnung der jeweils benötigten Information ist minimal und spielt für die Laufzeit praktisch keine Rolle. Die Ergebnisse für die verschiedenen Varianten sind im nächsten Abschnitt aufgeführt.

2.6 Testergebnisse des aussagenlogischen Beweisers

2.6.1 Pelletiers Testprobleme

Tabelle 2.1 zeigt Ergebnisse der Beweisläufe für die aussagenlogischen Probleme *pel1-pel17* aus [Pelletier, 1986]. Für diese einfachen Probleme machten die Übersetzungszeiten von Quintus-Prolog, die zwischen 200-780 *ms* lagen, den Hauptanteil der Gesamtlaufzeit aus. Die Dauer der eigentlichen Beweissuche konnte dagegen nicht gemessen werden, da sie unterhalb der Meßgenauigkeit von Quintus-Prolog ($\approx$ 17 *ms*) lag.

Was die Anzahl der geschlossenen Pfade anbetrifft, so hat für *diese Probleme* keine der Varianten $conv_{min}, conv_{min1st}, conv_{L1}$ schlechter abgeschnitten als die "Urversion" *conv*. Nur bei den Problemen *pel5, pel8, pel14* und *pel17* wirkten sich die Heuristiken überhaupt, dann aber günstig, auf die Größe des potentiellen Suchraumes $\#_1\mathcal{S}$ aus. Entsprechend war die Anzahl der geschlossenen Pfade geringer. Auffällig dabei ist, daß $conv_{L1}$ z.B. bei *pel14* zwar die wenigsten geschlossenen Pfade hat, hierzu jedoch mehr Knoten besuchen muß als *conv*, $conv_{min}$, $conv_{min1st}$, d.h. die untersuchten Modelle sind im Durchschnitt größer. Hierfür sind die von der Heuristik zusätzlich "eingestreuten" CUT-Knoten verantwortlich.

Ebenfalls bewährt hat sich $conv_{\alpha\_min}$. Bei dieser Tableau-Variante ist die Anzahl der **1**-Pfade $\#_1\mathcal{S}$ zwar grundsätzlich gleich der Anzahl der **1**-Pfade von $conv_{Tab}$ (es handelt sich bei beiden um voll expandierte Tableaux), jedoch ist die Anzahl der *geschlossenen* Pfade gelegentlich und die Anzahl der besuchten Knoten häufig geringer.

Problem	$\|\mathcal{S}\|$	$\#_1\mathcal{S}$						Geschlossene Pfade						Besuchte Knoten					
		s1	s2	s3	s4	t1	t2	s1	s2	s3	s4	t1	t2	s1	s2	s3	s4	t1	t2
pel1	6	4	4	4	4	4	4	4	4	4	4	4	4	7	7	7	8	9	8
pel2	3	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	4	4	4
pel3	4	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3
pel4	6	4	4	4	4	4	4	4	4	4	4	4	4	7	7	7	8	9	8
pel5	7	5	3	3	3	3	3	5	3	3	3	3	3	11	6	6	7	10	7
pel6	2	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2
pel7	2	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2
pel8	4	3	2	2	2	2	2	3	2	2	2	2	2	6	4	4	3	5	3
pel9	8	16	16	16	16	16	16	7	6	7	7	9	8	10	9	10	10	16	14
pel10	11	24	24	24	24	24	24	10	8	8	8	11	8	17	13	13	16	23	18
pel11	3	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	4	4	4
pel12	19	32	32	32	32	32	32	24	24	24	24	24	24	31	31	31	60	64	60
pel13	11	14	14	14	12	12	12	10	10	10	9	9	9	15	16	16	18	18	18
pel14	11	14	14	14	12	12	12	10	10	10	9	10	10	14	14	14	19	22	22
pel15	6	4	4	4	4	4	4	4	4	4	4	4	4	7	7	7	8	9	8
pel16	4	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3
pel17	16	34	30	30	24	24	24	20	17	17	14	14	14	30	27	26	26	30	26

$|\mathcal{S}| =$ Anzahl der Nichtterminal-Knoten des Shannon-Graphen
s1,. . . , $s4 =$ Varianten *conv*, $conv_{min}$, $conv_{min1st}$, $conv_{L1}$
t1,t2 = Varianten $conv_{Tab}$, $conv_{\alpha\_min}$

Tabelle 2.1: Vergleich verschiedener *conv*-Varianten

2.6.2 Probleme nach D'Agostino

D'Agostino beschreibt in [D'Agostino, 1990] eine Klasse von Problemen, die für Tableau-Beweiser ohne Lemma-Generierung sehr schwierig zu beweisen sind. Er zeigt, daß bei Verwendung der Standard-Tableauregeln die Beweise exponentiell in der Länge der Ausgangsformel sind.

D'Agostino hat einen Tableaukalkül **KE** vorgeschlagen ([D'Agostino, 1992]), der auf dem sogenannten *Principle of Bivalence* beruht und eine Lemma-Generierung "automatisch" beinhaltet. Mit diesem Kalkül sind, wie auch beim Tableau mit Lemmata, die Beweise dieser Probleme wesentlich kürzer. Wie in Abschnitt 2.4.2 gezeigt, entsprechen die von Shannon-Graphen (ohne CUT) repräsentierten partiellen Modelle denen eines Tableaus mit Lemmata.

Die in Tabelle 2.6.2 aufgeführten Ergebnisse bestätigen diesen Zusammenhang: *conv* ist dem Tableau ohne Lemmata $conv_{Tab}$ deutlich überlegen. $conv_{min}$, $conv_{min1st}$, $conv_{L1}$ brachten keine weiteren Verbesserungen gegenüber *conv*, die Beweise waren teilweise sogar geringfügig länger. $conv_{\alpha\_min}$ benötigte exakt die gleiche Anzahl von Inferenzen (besuchte Knoten, geschlossene Pfade) wie $conv_{Tab}$.

2.6.3 Laufzeitvergleich für verschiedene Zielsprachen

In Tabelle 2.3 sind die Ergebnisse für die als schwierig bekannten *Pigeon-Hole* Formeln für die Zielsprachen Prolog, C und 8086-Assembler vergleichend dargestellt. Im Gegensatz zu den Problemen von D'Agostino, bei denen die Vorverarbeitungszeit dominierend war, gibt hier die Beweissuche den entscheidenden Ausschlag.

Problem	$\|\mathcal{S}\|$	$\#_1\mathcal{S}$	*conv*				$conv_{Tab}$			
			GP	BK	VV	BS	GP	BK	VV	BS
ago2	8	16	6	9	316	†0	9	16	316	0
ago3	24	6561	24	31	616	0	341	510	766	0
ago4	64	$4,3 \cdot 10^9$	80	95	1.416	0	594.853	793.136	1.750	6.967
ago5	160	$2,3 \cdot 10^{22}$	240	271	3.517	0	?	?	4.346	?
ago6	384	$6,3 \cdot 10^{49}$	672	735	8.133	0	?	?	10.051	?

$|\mathcal{S}| =$ Anzahl der Nichtterminal-Knoten des Shannon-Graphen
GP = Anzahl der geschlossenen Pfade
BK = Anzahl der besuchten Knoten
VV = Laufzeit in *ms* für die Vorverarbeitung (inkl. Kompilierung durch Quintus-Prolog)
BS = Laufzeit in *ms* für die Beweissuche
" ? " = Beweisversuch nach 30 *min* abgebrochen
† Zeit lag unterhalb der Meßgenauigkeit von Quintus-Prolog (17 ms)

Tabelle 2.2: D'Agostinos Probleme

Die Formel *Pigeon-n* kann als folgende unerfüllbare Aussage interpretiert werden :

> "*n* Tauben können so auf *n-1* Löcher verteilt werden, daß in jedem Loch genau eine Taube sitzt."

Zur Formulierung dieser Aussage werden $n \cdot (n-1)$ aussagenlogische Atome benötigt. Die Länge der Formel, gemessen in Anzahl der Auftreten von Atomen, ist gleich der Anzahl $|\mathcal{S}|$ der Nichtterminal-Knoten der Shannon-Graph-Darstellung.

Die Varianten $conv_{Tab}$ und $conv_{\alpha\_min}$ verhielten sich gleich und benötigten wesentlich längere Beweise (1.038.336 geschlossene Pfade für *pig4*) als die abgebildete Variante $conv_{min1st}$.

C scheint als Zielsprache für die Kompilierung von Shannon-Graphen weniger geeignet. Der gegenüber Prolog höhere Aufwand zum Übersetzen des Programms wiegt bei vielen Beispielen schwerer als der Vorteil, der sich durch die geringfügig kürzere Laufzeit der Beweissuche ergibt. Beste Ergebnisse werden natürlich mit Assembler als Zielsprache erreicht. Die angegebenen Zeiten für die Vorverarbeitung, die zum größten Teil in Prolog stattfindet, können durch die direkte Erzeugung von Maschinen-Kode weiter drastisch verringert werden.

2.6.4 Fazit

Die vorgestellten Heuristiken haben sich im großen und ganzen bewährt. Bei den eigentlichen Shannon-Graph Varianten erwiesen sich $conv_{min1st}$ und $conv_{L1}$ im Durchschnitt als die beste Wahl. Die mit $conv_{\alpha\_min}$ konstruierten Tableaux wiederum waren häufig geeigneter als die von $conv_{Tab}$ erzeugten. Allerdings berücksichtigt keine der vorgestellten Heuristiken die Reihenfolge der besuchten Literale, wodurch ihre Leistungsfähigkeit eingeschränkt wird. Mögliche Verbesserungen des Verfahrens müssen an dieser Stelle ansetzen, etwa durch die Vorgabe einer Ordnung auf den Literalen, wie sie auch bei *BDDs* verwendet wird.

Problem	$\|\mathcal{S}\|$	$\#_1\mathcal{S}$	Geschl. Pfade	Besuchte Knoten	*Prolog*†		*C*‡		*Assembler*⋆	
					VV	BS	VV	BS	VV	BS
pig2	4	2	2	4	100	*0	347	0	117	0
pig3	18	512	37	61	400	0	463	0	563	0
pig4	48	$2,1 \cdot 10^7$	644	883	916	0	1.190	0	947	0
pig5	100	$1,1 \cdot 10^{15}$	7.351	9.678	1.883	83	3.190	60	1.700	0
pig6	180	$5,9 \cdot 10^{26}$	124.265	156.757	3.349	1.717	5.593	1.170	2.787	110
pig7	294	$2,4 \cdot 10^{43}$	2.414.873	2.961.696	5.433	28.733	9.953	22.250	4.377	1.040

$|\mathcal{S}| =$ Anzahl der Nichtterminal-Knoten des Shannon-Graphen
VV = Laufzeit in *ms* für die Vorverarbeitung (inkl. Kompilierung bzw. Assemblierung)
BS = Laufzeit in *ms* für die Beweissuche
† *Quintus-Prolog 3.0* auf *SUN-SPARC-2*
‡ *Unix cc* auf *SUN-SPARC-2*
⋆ *8086-Assembler-Kode* auf *Intel i486* 40 MHz
* Zeit lag unterhalb der Meßgenauigkeit (17 ms)

Tabelle 2.3: Vergleich von Laufzeiten für *Pigeon-Hole* Formeln

Durch die Verwendung der beschriebenen Übersetzungsstechnik erzielen wir einen erheblichen Geschwindigkeitsgewinn bei der eigentlichen Beweissuche. Dafür müssen wir, insbesondere bei "leichten" Problemen, vergleichsweise lange Gesamtlaufzeiten in Kauf nehmen. Beispielsweise ist der Beweis für die Formel *ago6* aus D'Agostinos Problemklasse sehr kurz im Vergleich zur Länge der Formel (672 geschlossene Pfade gegenüber 384 Literalen der Ausgangsformel). Die Kompilierung des generierten Programms ist in diesem Fall aber mit hohem Zeitaufwand verbunden. Dies ist jedoch kein prinzipieller Nachteil des Verfahrens: Während die Konvertierung einer Formel in einen Shannon-Graphen bereits mit Prolog extrem effizient realisiert werden kann,[26] ist auch der Aufwand für die Kodegenerierung und eine sich eventuell anschließende Kompilierung linear in der Länge der Eingabeformel. Hier können durch Verwendung einer spezialisierten Zielsprache oder einer maschinennahen Sprache die relativ hohen Kosten für die Kompilierung von Prolog- bzw. C-Programmkode eingespart werden.

Durch die Wahl einer maschinennahen Zielsprache erreichen wir zusätzlich eine deutliche, wenn auch lineare, Verkürzung der Laufzeiten. Der Implementierungsaufwand Programmkode für verschiedene Sprachen zu erzeugen ist gering, da hierzu nur ein Modul der bestehenden Implementierung angepaßt werden muß.

[26] Auf einer *SUN-SPARC-2* mit *Quintus-Prolog 3.0* benötigt die Konstruktion des Shannon-Graphen für eine Formel mit 10.000 Literalen etwa 400 ms.

Kapitel 3

Prädikatenlogische Shannon-Graphen

3.1 Einleitung

Im Rahmen des automatischen Beweisens liegt die Problemstellung häufig in folgender Form vor: Man möchte wissen, ob ein gegebener Satz[1] Th (Theorem) aus einer endlichen Mengen von Sätzen $Ax_1, \ldots, Ax_n$ (Axiome) logisch folgt, das heißt, ob $\{Ax_1, \ldots, Ax_n\} \models Th$ gilt. Dieses Problem läßt sich auf den Nachweis der Unerfüllbarkeit des Satzes $S = Ax_1 \wedge \ldots \wedge Ax_n \wedge \neg Th$ reduzieren. Deshalb können wir im folgenden unser Bemühen darauf beschränken, einen gegebenen Satz S zu widerlegen.

Während im aussagenlogischen Fall algorithmisch entschieden werden kann, ob eine Formel unerfüllbar ist, müssen wir uns in der Prädikatenlogik mit einem "Semi-Entscheidungsverfahren" begnügen: Die Menge der unerfüllbaren Formeln der Prädikatenlogik ist (wie auch die Menge der allgemeingültigen Formeln) rekursiv aufzählbar, aber nicht entscheidbar. Ein vollständiges Beweisverfahren vorausgesetzt, können wir also zumindest den Nachweis für die Unerfüllbarkeit einer Formel erbringen.

Zunächst beschäftigen wir uns in Abschnitt 3.3 mit der Vorverarbeitung von prädikatenlogischen Sätzen. Wir richten unser Augenmerk dabei besonders auf die im Rahmen der Vorverarbeitung notwendige Skolemisierung und zeigen exemplarisch, daß die in der Literatur häufig verwendete, hier $\diamond$ genannte Skolemisierung für unsere Zwecke weniger geeignet ist als die Skolemisierung $\star$ nach [Andrews, 1986].

Abschnitt 3.4 beschreibt einen einfachen Ansatz, wie man die aussagenlogische Beweisprozedur aus Kapitel 2 auf die Prädikatenlogik erweitern kann.[2] Dazu wird in einem ersten Schritt die Ausgangsformel in eine erfüllbarkeitsäquivalente, quantorenfreie Formel überführt, die dann ihrerseits mittels der aussagenlogischen *conv*-Funktion in einen Shannon-Graphen konvertiert werden kann. Gelingt es, die freien Variablen des Shannon-Graphen so zu instantiieren, daß alle **1**-Pfade geschlossen sind, so ist die Formel widerlegt. Anderfalls muß man sich durch eine sogenannte *maximale Erweiterung* weitere

[1] Eine Formel, die keine freien Variablen enthält, bezeichnet man üblicherweise als *geschlossene Formel*. Um Verwechslungen mit dem Begriff des geschlossenen, d.h. inkonsistenten Shannon-Graphen vorzubeugen, nennen wir eine Formel, in der keine freien Variablen vorkommen, einen *Satz*.

[2] Dies ist der in [Posegga & Ludäscher, 1992] beschriebene Ansatz.

Instanzen des initialen Graphen mit neuen freien Variablen verschaffen und versuchen, den erweiterten Graphen abzuschließen. Dieser Ansatz liefert für nicht allzu schwierige Probleme bereits brauchbare Resultate. Allerdings geht bei der Umformung in eine quantorenfreie Formel wichtige Information über den Skopus von $\forall$-quantifizierten Formeln verloren, was im allgemeinen den Suchaufwand unnötig vergrößert.

Die in Abschnitt 3.5 vorgestellte Beweisprozedur der *selektiven Erweiterungen* behebt diese Schwäche des ersten Ansatzes und stellt, was die Leistungsfähigkeit angeht, eine erhebliche Verbesserung gegenüber diesem dar. Anstatt jeden nicht geschlossenen **1**-Pfad π mit dem gesamten initialen Graphen zu erweitern, wählen wir einen in π auftretenden sogenannten γ-*Untergraphen* $\mathcal{G}$ und erweitern diesen **1**-Pfad selektiv mit $\mathcal{G}$. Die γ-Untergraphen übernehmen dabei eine ähnliche Funktion wie die γ-Formeln im *Tableaukalkül mit freien Variablen*. Um den Nachweis der Korrektheit und Vollständigkeit des Verfahrens der selektiven Erweiterungen zu erbringen, benutzen wir Beweistechniken, die ebenfalls aus dem Tableaukalkül mit freien Variablen bekannt sind.

Schließlich beschreiben wir in Abschnitt 3.6 das prinzipielle Vorgehen, nach dem das Verfahren der selektiven Erweiterungen implementiert wurde.

Bevor wir zur Tat schreiten, erläutern wir kurz die verwendete Notation.

3.2 Vereinbarungen und Notationen

Wir erweitern die übliche Definition der Sprache der Prädikatenlogik durch Hinzunahme des dreistelligen Shannon-Junktors sh und der beiden Atome **0** und **1** zur Sprache $\mathcal{P}$. Die Definition der Semantik von sh, bzw. **0** und **1** aus Kapitel 2 kann direkt für $\mathcal{P}$ übernommen werden.

$\mathcal{P}$ umfaßt neben der Menge der Prädikatensymbole auch eine abzählbar unendliche Menge von Funktionssymbolen und die Menge der Variablen $\mathbf{Var} = \{x_1, x_2, \ldots\}$. Mit **Term** bezeichnen wir die Menge aller Terme.

Für ein Tupel $\langle x_1, \ldots, x_n \rangle$ von Variablen schreiben wir kurz $\bar{x}$, analog $\bar{t}$ für ein Tupel von Termen. Freie Variablen schreiben wir gelegentlich mit Großbuchstaben, um sie von gebundenen Variablen abzuheben.

$\mathbf{For}_{At}$, **For** und $\mathbf{For}_{\mathcal{SH}}$ stehen jetzt für die Menge der prädikatenlogischen Atome ohne $\{\mathbf{0}, \mathbf{1}\}$, die Menge der wohlgeformten Formeln von $\mathcal{P}$ ohne Verwendung von sh, **0**, **1**, bzw. die Menge aller wohlgeformten Formeln von $\mathcal{P}$.

$\mathcal{SH}$ und $\mathcal{SH}_!$, die Menge der Shannon-Formeln bzw. die Menge der CUT-Shannon-Formeln, sind wie in Kapitel 2, nun aber über prädikatenlogischen Atomen definiert. Aus dem Zusammenhang ist eindeutig zu entnehmen, ob es sich bei den bezeichneten Formelmengen um aussagenlogische oder prädikatenlogische handelt, so daß keine Verwechslungsgefahr bestehen dürfte.

Im übrigen bleiben die Rechenregeln aus Lemma 2.5, die Kompositionsregeln aus Lemma 2.6 und insbesondere die *DNF*-Darstellung $\mathcal{S} \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S})$ aus Satz 2.11 auch bei prädikatenlogischer Lesart, d.h. bei vorgegebener Struktur $\mathcal{D}$ und Variablenbelegung β korrekt.

3.3 Vorverarbeitung von prädikatenlogischen Sätzen

Quantorenfreie prädikatenlogische Formeln können wir mit Hilfe der Funktion *conv* aus Kapitel 2 in logisch äquivalente Shannon-Formeln transformieren. Um beliebige prädikatenlogische Formeln betrachten zu können, müssen wir uns daher in geeigneter Weise der Quantoren entledigen.

Ein erster Ansatz besteht darin, einen gegebenen Satz S der Prädikatenlogik in eine quantorenfreie Formel $F(\bar{x})$ mit den freien Variablen $\bar{x}$ umzuwandeln. Zwar können wir im allgemeinen nicht erreichen, daß S und $F(\bar{x})$ logisch äquivalent sind, jedoch bleibt die *Erfüllbarkeit* bei dieser Transformation erhalten, d.h. S ist unerfüllbar gdw. der Allabschluß $\forall\bar{x}F(\bar{x})$ von $F(\bar{x})$ unerfüllbar ist, oder kurz:

$$S \models \square \text{ gdw. } \forall\bar{x}F(\bar{x}) \models \square$$

In der Literatur findet man häufig die folgende Vorgehensweise, um $F(\bar{x})$ aus S zu konstruieren:[3]

1. S wird mit Hilfe der üblichen Umformungsregeln (wie etwa der *de Morgan*-Regeln) zunächst in eine Formel S' in *Negations-Normalform* umgewandelt.
2. Durch gebundene Umbenennung erhält man die *bereinigte Form* S'', in der alle durch Quantoren gebundene Variablen alphabetisch verschieden bezeichnet sind.
3. Nun werden die Regeln[4]

$$\begin{array}{rcl}(\mathbf{Q}x\,A) \circ B & \Leftrightarrow & \mathbf{Q}x(A \circ B) \\ B \circ (\mathbf{Q}x\,A) & \Leftrightarrow & \mathbf{Q}x(B \circ A)\end{array}$$

   in Richtung von links nach rechts angewendet, um alle Quantoren $\mathbf{Q}$ "nach außen zu ziehen", wodurch sich eine *pränexe Normalform* $\mathbf{Q}_1, \ldots, \mathbf{Q}_k S'''$ ergibt.
4. $\mathbf{Q}_1, \ldots, \mathbf{Q}_k S'''$ wird schließlich durch eine hier $\diamond$ genannte *Skolemisierung* in die gewünschte Form $\forall\bar{x}F(\bar{x})$ gebracht.

In der Skolemisierung $\diamond$ nutzt man nun die Anordnung der Quantoren im Präfix $\mathbf{Q}_1, \ldots, \mathbf{Q}_k$ der Formel aus:

Definition 3.1 (Skolemisierung $\diamond$)
Die Skolemisierung $S^\diamond$ eines Satzes S ist die letzte Formel der Folge von Formeln S_i, die durch die nachfolgende Iteration gegeben sind:

1. *S_0 ist eine pränexe Normalform von S.*
2. *Ist S_i von der Form $\forall x_1 \ldots \forall x_n \exists y\ S'$ für ein $n \geq 0$, dann ist*

$$S_{i+1} = \forall x_1 \ldots \forall x_n S'\{y/f_{i+1}(x_1, \ldots, x_n)\},$$

   d.h. die Auftreten von y in S' werden durch ein neues n-stelliges Skolemfunktionssymbol f_{i+1} mit den Argumenten $x_1, \ldots, x_n$ ersetzt.

[3] Siehe z.B. [Chang & Lee, 1973],[Schöning, 1987].
[4] "$\circ$" steht für $\wedge$ bzw. $\vee$.

3. *Falls S_i keinen $\exists$-Quantor mehr enthält, setzen wir $S^\diamond := S_i$ und erklären das Verfahren für beendet.*

Eine $\exists$-quantifizierte Variable y hängt nach dieser Skolemisierung also von allen $\forall$-quantifizierten Variablen $x_1, \ldots, x_n$ ab, die im Quantor-Präfix vor y stehen. Durch dieses Vorgehen entstehen jedoch *zuviele Abhängigkeiten,* wie das nachfolgende Beispiel belegt:

Beispiel 3.2
Eine pränexe Normalform von $S = \forall x(\exists y(p(y)) \wedge \exists z(q(x,z)))$ ist

$$\forall x \exists y \exists z(p(y) \wedge q(x,z))$$

damit erhalten wir

$$S^\diamond = \forall x(p(f_1(x)) \wedge q(x, f_2(x))).$$

Diese Formel enthält die Abhängigkeit der Skolemfunktion f_1 von x, die in der Ausgangsformel S jedoch nicht vorhanden ist. Mittels der Regel $\forall x(A \circ B) \Leftrightarrow A \circ \forall x B$ (falls x nicht frei in A), können wir S zunächst umformen zu $\exists y(p(y)) \wedge \forall x \exists z(q(x,z))$. Eine mögliche pränexe Normalform ist nun

$$S' = \exists y \forall x \exists z(p(y) \wedge q(x,z))$$

mit der Skolemisierung

$$S'^\diamond = \forall x(p(f_1) \wedge q(x, f_2(x)))$$

und der von x unabhängigen Skolem*konstanten* f_1.

Für einen prädikatenlogischen Theorembeweiser, der Interpretationen über dem Herbrand-Universum $\mathcal{U}_F$ einer Formel F betrachten muß, ist die Form $S'^\diamond$ der Form $S^\diamond$ vorzuziehen, denn $\mathcal{U}_{S'^\diamond} = \{f_1, f_2(f_1), f_2(f_2(f_1)), \ldots\}$ ist deutlich "einfacher" als $\mathcal{U}_{S^\diamond} = \{c, f_1(c), f_2(c), f_1(f_2(c)), f_2(f_1(c)), \ldots\}$.[5]

Eine Möglichkeit, diese ungewollten Abhängigkeiten zu vermeiden, könnte also darin bestehen, mit Hilfe der Regeln

$$\begin{array}{llll} \forall x(A \circ B) & \Leftrightarrow & \forall x(A) \circ B & \forall x(B \circ A) \Leftrightarrow B \circ \forall x(A) \\ \exists x(A \circ B) & \Leftrightarrow & \exists x(A) \circ B & \exists x(B \circ A) \Leftrightarrow B \circ \exists x(A) \end{array} \quad \text{(falls } x \text{ nicht frei in } B\text{)}$$

in einem ersten Schritt Quantoren möglichst weit "nach innen zu ziehen", um sie anschließend wieder so nach "außen zu ziehen", daß vor einem $\exists$-Quantor möglichst wenige $\forall$-Quantoren stehen.

Dies ist jedoch aus mehreren Gründen nicht angeraten: Einerseits wird die Implementierung des Verfahrens durch den Indeterminismus bei der Regelanwendung erschwert, zum anderen gibt es einfache Beispiele, bei denen durch eine Linearisierung der Abhängigkeiten im Quantor-Präfix der Formel immer ungewollte "Scheinabhängigkeiten" entstehen, die man durch eine geschicktere Art der Skolemisierung vermeiden kann.

[5] Definitionsgemäß enthält das Herbrand-Universum mindestens eine – hier mit c bezeichnete – Konstante.

Beispiel 3.3
In der Formel

$$\forall w \exists x (p(w,x) \wedge \forall y (q(w,x,y) \vee \exists z (r(y,z))))$$

können wir nach obigen Regeln keinen Quantor nach innen ziehen. Obwohl das Auftreten der $\exists$-quantifizierten Variablen z nur von y und nicht von w abhängt, wird in einer pränexen Normalform stets w vor z stehen, wie auch immer wir obige Regeln anwenden.[6]

Um die genannten Probleme zu umgehen, verwenden wir die folgende Skolemisierung $\star$, die der in [Andrews, 1986] beschrieben entspricht, wobei wir gegenüber dem dortigen Ausgangspunkt zusätzlich annehmen, daß die zu skolemisierende Formel S bereits in Negations-Normalform (aber nicht in pränexer Normalform) vorliegt.

Definition 3.4 (Skolemisierung $\star$)
Die Skolemisierung $S^\star$ eines Satzes S ist die letzte Formel der Folge von Formeln S_i, die durch die nachfolgende Iteration gegeben sind:

1. *S_0 ist die bereinigte Negations-Normalform von S.*
2. *Sei $\exists y\ F$ der "am weitesten links stehende" $\exists$-Quantor von S_i mit Skopus F, $x_1, \ldots, x_n$ die in F freien Variablen.*

   Dann ergibt sich S_{i+1} aus S_i durch Ersetzen von $\exists y\ F$ durch $F\{y / f_{i+1}(x_1, \ldots, x_n)\}$,

   d.h. die Auftreten von y in der Teilformel F von S_i werden durch ein neues n-stelliges Skolemfunktionssymbol f_{i+1} mit den Argumenten $x_1, \ldots, x_n$ ersetzt.
3. *Falls S_i keinen $\exists$-Quantor mehr enthält, setzen wir $S^\star := S_i$ und erklären das Verfahren für beendet.*

Die Skolemisierung $\star$ erzeugt "bessere" Formeln als die Skolemisierung $\diamond$, d.h. Formeln, bei denen die Skolemfunktionen von weniger Variablen abhängen, als bei Verwendung von $\diamond$. Zudem ist die Skolemisierung $\star$ zur Implementierung besonders geeignet, da die Ersetzungen rekursiv für Teilformeln angewendet werden können.

Für das Beispiel 3.3 erhalten wir nun das gewünschte Ergebnis

$$\forall w (p(w, f_1(w)) \wedge \forall y (q(w, f_1(w), y) \vee (r(y, f_2(y))))).$$

Definition 3.5
Wir nennen die Form $S^\star$, die man nach der Skolemisierung $\star$ von S erhält, die skolemisierte Normalform *SNF* von S.*

Nach Definition von $\star$ ist ein Satz in *SNF** demnach in Negations-Normalform und enthält nur noch $\forall$-Quantoren. Für die beschriebene Skolemisierung gilt ([Andrews, 1986]):

Satz 3.6
Ein Satz S und dessen Skolemisierung $S^\star$ sind erfüllbarkeitsäquivalent, d.h.

$$S \models \square \quad \text{gdw.} \quad S^\star \models \square$$

[6] Man versuche dies selbst.

In den beiden nächsten Abschnitten gehen wir stets von einer Formel $S^\star$ in *SNF** aus. Zusätzlich fordern wir für das folgende Verfahren mit *maximalen Erweiterungen*, daß $S^\star$ noch in eine pränexe Normalform gebracht wird, wodurch sich schließlich die gewünschte Form $\forall \bar{x} F(\bar{x})$ mit der quantorenfreien Formel $F(\bar{x})$ und den freien Variablen $\bar{x}$ ergibt.

3.4 Shannon-Graphen mit maximalen Erweiterungen

Sei also eine quantorenfreie Formel $F(\bar{x})$ mit den freien Variablen $\bar{x}$ gegeben. Wie zeigen wir nun, daß der Allabschluß $\forall \bar{x} F(\bar{x})$ unerfüllbar ist? Wir benutzen die folgenden wohlbekannten Eigenschaften der Prädikatenlogik:

$\forall \bar{x} F(\bar{x}) \models \square$

gdw. $\{F(\bar{t}) \mid \bar{t} \in \mathcal{U}_F\} \models \square$ *Satz von Gödel-Herbrand-Skolem*

gdw. $\{F(\bar{t}_1), \ldots, F(\bar{t}_n)\} \models \square$ *Endlichkeitssatz*

gdw. $(F(\bar{x}_1) \wedge \ldots \wedge F(\bar{x}_n))\sigma \models \square$ *Substitutionslemma*

Dabei sind $\bar{t}_1, \ldots, \bar{t}_n$ Tupel von Termen aus dem Herbrand-Universum $\mathcal{U}_F$ von F. Alle Variablen, die in $\bar{x}_1, \ldots, \bar{x}_n$ auftreten, sind paarweise verschieden, und σ ist eine Grundsubstitution dieser Variablen nach $\mathcal{U}_F$.

Zusammenfassend gilt also der folgende Satz, der die theoretische Grundlage des Verfahrens der maximalen Erweiterungen darstellt:

Satz 3.7
Für jede quantorenfreie prädikatenlogische Formel $F(\bar{x})$ mit den freien Variablen $\bar{x}$ gilt

$$\forall \bar{x} F(\bar{x}) \models \square$$

gdw. es ein $n \in \mathbb{N}$ und eine Substitution σ : **Var**$\rightarrow \mathcal{U}_F$ gibt, so daß

$$(F(\bar{x}_1) \wedge \ldots \wedge F(\bar{x}_n))\sigma \models \square$$

Die Idee für das Beweisverfahren ist jetzt recht einfach: Da $F(\bar{x})$ keine Quantoren mehr enthält, können wir mittels der Funktion *conv* aus Definition 2.13 einen logisch äquivalenten Shannon-Graphen $\mathcal{F}_0(\bar{x}_0)$ mit den freien Variablen $\bar{x}_0 = \bar{x}$ konstruieren, den sogenannten *initialen Graphen* von $F(\bar{x})$.

Wir versuchen nun, $\mathcal{F}_0(\bar{x}_0)$ durch geeignete Instantiierung der Variablen $\bar{x}_0$ zu schließen. Hierzu definieren wir:

Definition 3.8 (Geschlossene Shannon-Graphen)
Ein Shannon-Graph $\mathcal{F}(\bar{x})$ mit den freien Variablen $\bar{x}$ heißt geschlossen, *wenn es eine Substitution σ gibt, so daß alle* **1**-*Pfade in $\mathcal{F}(\bar{x})\sigma$ geschlossen sind. σ heißt* schließende Substitution *für $\mathcal{F}(\bar{x})$.*

Zur Erinnerung: Ein **1**-*Pfad heißt* geschlossen, *wenn er komplementäre Literale enthält.*

Können wir $\mathcal{F}_0(\bar{x}_0)$ jedoch nicht abschließen, so betrachten wir die Formel $F(\bar{x}_0) \wedge F(\bar{x}_1)$. Diese ist logisch äquivalent zum Shannon-Graphen $\mathcal{F}_1(\bar{x}_0, \bar{x}_1) = \mathcal{F}_0(\bar{x}_0)[\frac{\mathbf{1}}{\mathcal{F}_0(\bar{x}_1)}]$. Wir versuchen nun erneut, $\mathcal{F}_1(\bar{x}_0, \bar{x}_1)$ zu schließen. War $\forall x F(\bar{x})$ unerfüllbar, so garantiert Satz 3.7, daß dieses Vorgehen, wenn man es wiederholt anwendet, schließlich für ein $n \in I\!N$ mit einem geschlossenen Shannon-Graphen $\mathcal{F}_n(\bar{x}_0, \ldots, \bar{x}_n)$ zu Ende kommt. Dem Problem, wie wir die $\mathcal{F}_n$ schließende Substitutionen tatsächlich finden können, wenden wir uns in Abschnitt 3.6.1 zu.

Beispiel 3.9
Wir wollen zeigen, daß $\exists y(p(y) \wedge \neg p(ffy)) \wedge \forall x(p(x) \rightarrow p(fx))$ unerfüllbar ist. Dabei ist f ein einstelliges Funktionssymbol, bei dem wir, der besseren Lesbarkeit wegen, Klammern weglassen. Nach Skolemisierung erhalten wir die Formel $F(X_0) = p(a) \wedge \neg p(ffa)) \wedge (\neg p(X_0) \vee p(fX_0))$ mit der Skolemkonstanten a und der freien Variablen X_0.

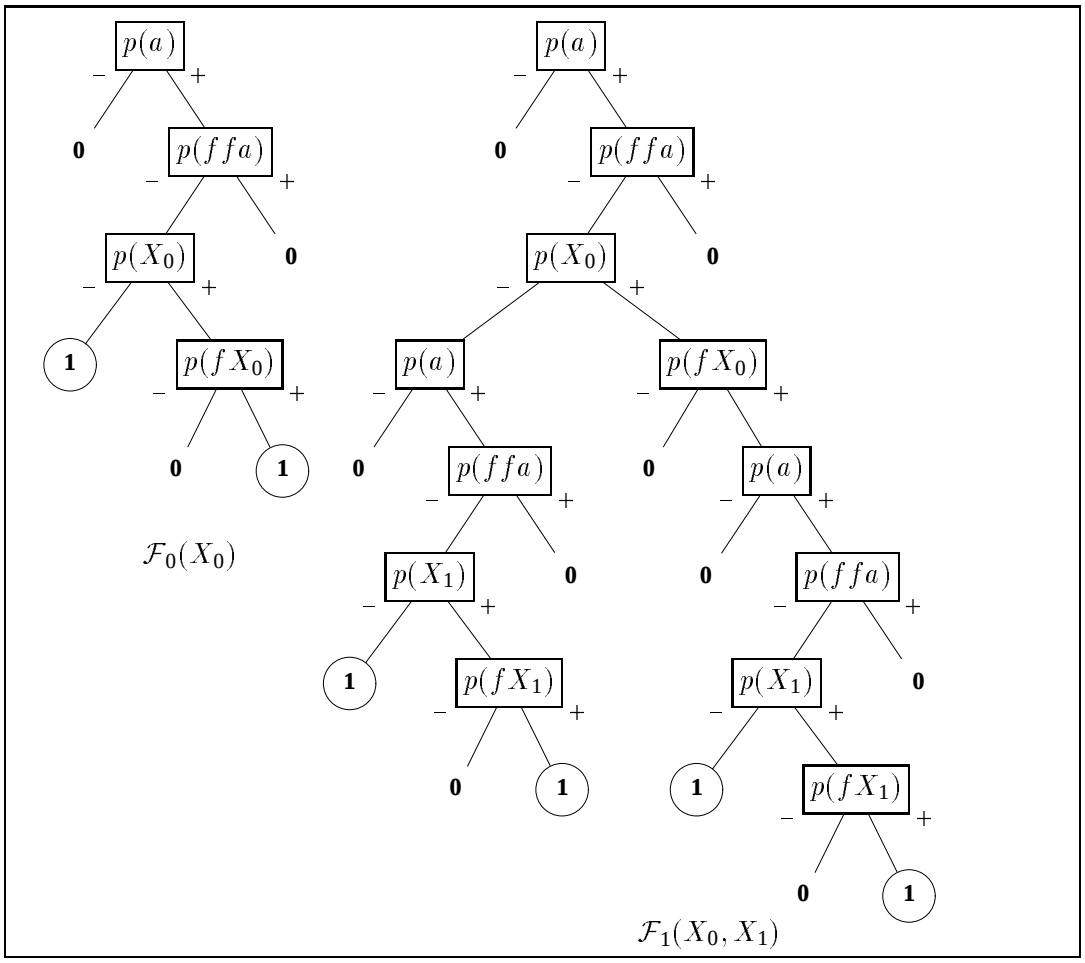

Abbildung 3.1: Initialer Shannon-Graph $\mathcal{F}_0(X_0)$ und erste Erweiterung $\mathcal{F}_1(X_0, X_1)$

Abbildung 3.1 zeigt den zu $F(X_0)$ äquivalenten initialen Graph $\mathcal{F}_0(X_0) = \mathit{conv}(F(X_0))$.[7]

[7] Da **1**-Knoten im weiteren im Gegensatz zu **0**-Knoten eine besondere Rolle spielen, haben wir, um die

Wählen wir $\sigma = \{X_0/a\}$, so können wir den **1**-Pfad $\{p(a), \neg p(ffa), \neg p(X_0)\}$ von $\mathcal{F}_0$ schließen, nicht jedoch den verbleibenden **1**-Pfad. Mit $\sigma = \{X_0/fa\}$ verhält es sich gerade umgekehrt. Wir benötigen also eine zweite Instanz $F(X_1)$ mit der neuen freien Variablen X_1. $\mathcal{F}_1(X_0, X_1) = \mathcal{F}_0(X_0)[\frac{\mathbf{1}}{\mathcal{F}_0(X_1)}]$ ist der zu $F(X_0) \wedge F(X_1)$ logisch äquivalente Shannon-Graph, der nun mit der Substitution $\sigma = \{X_0/a, X_1/f(a)\}$ geschlossen werden kann.

Es ergibt sich demnach folgendes Verfahren der maximalen Erweiterungen:

Definition 3.10 (Maximale Erweiterungen)
Sei $S^\star$ die skolemisierte Normalform eines Satzes S der Prädikatenlogik, $\forall x F(\bar{x})$ die pränexe Normalform von $S^\star$. Zu $F(\bar{x}_0)$ mit den freien Variablen $\bar{x}_0$ konstruieren wir eine Folge $\mathcal{F}_i$ von Shannon-Graphen:

$$\begin{aligned} \mathcal{F}_0(\bar{x}_0) &:= \mathit{conv}(F(\bar{x}_0)) \\ \mathcal{F}_{i+1}(\bar{x}_0, \ldots, \bar{x}_{i+1}) &:= \mathcal{F}_i(\bar{x}_0, \ldots, \bar{x}_i)[\frac{\mathbf{1}}{\mathcal{F}_0(\bar{x}_{i+1})}] \end{aligned}$$

Ist ein $\mathcal{F}_i$ für eine Grundsubstitution σ geschlossen, so ist das Verfahren beendet, andernfalls konstruieren wir in der beschriebenen Weise $\mathcal{F}_{i+1}$.

$\mathcal{F}_0$ heißt der initiale Graph *von $F(\bar{x})$, $\mathcal{F}_i$ die* i-te maximale Erweiterung.

Diese Beweisprozedur geht der Idee nach auf ein in [Orłowska, 1969a] vorgestelltes Verfahren zurück. Die dort beschriebene Beweisprozedur stellt für die Menge aller Sätze ohne Funktionssymbole und in pränexer Normalform, wobei die Quantor-Präfixe von der Form $\forall \cdots \forall \exists \cdots \exists$ (kurz: $\forall^* \exists^*$) sein müssen, ein Entscheidungsverfahren auf Allgemeingültigkeit dar.[8] Der Nachweis der Allgemeingültigkeit eines solchen Satzes $\forall^* \exists^* S$ ist äquivalent zum Nachweis der Unerfüllbarkeit des Satzes $\exists^* \forall^* \neg S$. Dieses Problem ist entscheidbar, da das Herbrand-Universum $\mathcal{U}_{S^\star}$ der Skolemisierung $S^\star$ von $\exists^* \forall^* \neg S$ endlich ist.

Das Verfahren der maximalen Erweiterungen kann natürlich kein Entscheidungsverfahren für die Unerfüllbarkeit eines beliebigen prädikatenlogischen Satzes S sein. Es ist aber korrekt und vollständig, wie der folgende Satz belegt.

Satz 3.11 (Korrektheit und Vollständigkeit der maximalen Erweiterungen)
Sei S ein Satz der Prädikatenlogik, $\mathcal{F}_n$ die n-te maximale Erweiterung, dann gilt:

$S \models \Box$ gdw. $\mathcal{F}_n(\bar{x}_0, \ldots, \bar{x}_n)$ geschlossen für ein gewisses $n \in I\!N$.

BEWEIS
S und die zugehörige skolemisierte Formel in pränexer Normalform $\forall \bar{x} F(\bar{x})$ sind erfüllbarkeitsäquivalent, d.h. $S \models \Box$ gdw. $\forall \bar{x} F(\bar{x}) \models \Box$. Nach Satz 3.7 gilt $\forall \bar{x} F(\bar{x}) \models \Box$

Wichtigkeit der ersteren hervorzuheben, letztere *ohne Kreis* dargestellt.

[8] Am Ende von [Orłowska, 1969b] wird angedeutet, wie das Verfahren auf die volle Prädikatenlogik (ohne Gleichheit) erweitert werden kann, wobei die Entscheidbarkeit selbstverständlich verloren geht.

gdw. $(F(\bar{x}_1)\wedge\ldots\wedge F(\bar{x}_n))\sigma \models \Box$ für ein $n \in I\!N$ und eine Grundsubstitution σ. Mittels Induktion und unter Zuhilfenahme der Eigenschaft $F(\bar{x}) \Leftrightarrow \mathit{conv}(F(\bar{x}))$ von *conv* (Satz 2.11) und der Kompositionsregel $(REP_\wedge)$ zeigt man sofort, daß für alle i gilt

$$\mathcal{F}_i(\bar{x}_0,\ldots,\bar{x}_i) \Leftrightarrow F(\bar{x}_1)\wedge\ldots\wedge F(\bar{x}_i).$$

Also gilt $S \models \Box$ gdw. $(\mathcal{F}_n(\bar{x}_0,\ldots,\bar{x}_n))\sigma \models \Box$ für ein $n \in I\!N$ und eine Grundsubstitution σ. Letzteres ist aber ein aussagenlogischer Shannon-Graph, der nach Korollar 2.12 genau dann unerfüllbar ist, wenn er geschlossen ist. ■

3.4.1 Eine einfache Verbesserung der maximalen Erweiterungen

In der vorgestellten Form werden bei der Konstruktion einer maximalen Erweiterung $\mathcal{F}_{i+1}$ von $\mathcal{F}_i$ alle **1**-Knoten von $\mathcal{F}_i$ durch dieselbe Shannon-Formel $\mathcal{F}_0(\bar{x}_{i+1})$ mit den freien Variablen $\bar{x}_{i+1}$ ersetzt. Dies hat zur Folge, daß auf jedem "Erweiterungslevel" j dieselben Variablen $\bar{x}_j$ verwendet werden müssen. Tatsächlich ist es jedoch möglich, an jedem **1**-Knoten von $\mathcal{F}_i$ neue, voneinander unabhängige Instanzen von $\mathcal{F}_0$ mit neuen freien Variablen zu verwenden. Dies erleichtert zum einen die Implementierung des Verfahrens (Variablenbindungen können dann vom "Erweiterungslevel" unabhängig repräsentiert werden), zum anderen kann durch den größeren Vorrat an freien Variablen ein Abschluß früher, d.h. mit weniger Erweiterungen, gefunden werden. Das folgende Beispiel belegt dies. Die Korrektheit dieser Modifikation ergibt sich aus der Tatsache, daß diese (wie auch das ursprüngliche Verfahren der maximalen Erweiterungen) Spezialfälle des in Abschnitt 3.5 beschriebenen Verfahrens der *selektiven Erweiterungen* sind, dessen Korrektheit wir noch zeigen werden.

Beispiel 3.12
Es ist die Unerfüllbarkeit der Formel $\forall x_0(p(a)\wedge\neg p(fffa)\wedge(\neg p(x_0)\vee p(fx_0)))$ nachzuweisen. Der zugehörige initiale Graph entspricht dem in Beispiel 3.9 gezeigten $\mathcal{F}_0$, bis auf den Knoten für $p(ffa)$, der hier durch $p(fffa)$ ersetzt wird.

Um den Shannon-Graphen abzuschließen, sind *zwei* maximale Erweiterungen

$$\mathcal{F}_1(X_0,X_1) := \mathcal{F}_0(X_0)[\frac{\mathbf{1}}{\mathcal{F}_0(X_1)}] \text{ und } \mathcal{F}_2(X_0,X_1,X_2) := \mathcal{F}_1(X_0,X_1)[\frac{\mathbf{1}}{\mathcal{F}_0(X_2)}]$$

nötig. Erweitert man an den **1**-Knoten von $\mathcal{F}_0$ jedoch mit voneinander unabhängigen Instanzen $\mathcal{F}'_0(X_1)$ und $\mathcal{F}''_0(X_2)$, so erhält man einen um ein "Erweiterungslevel" kleineren Shannon-Graphen $\mathcal{F}_1(X_0,X_1,X_2)$.

Abbildung 3.2 zeigt diesen, der für $\sigma = \{X_0/fa, X_1/a, X_2/ffa\}$ geschlossen ist. Anhand der Nummern neben den Nichtterminal-Knoten lassen sich die verschiedenen Instanzen des initialen Graphen $\mathcal{F}_0(X_0)$, $\mathcal{F}'_0(X_1)$ und $\mathcal{F}''_0(X_2)$ unterscheiden.

An diesem Beispiel fällt auf, daß trotz der genannten Verbesserung die Struktur von $\mathcal{F}_1(X_0,X_1,X_2)$ einige Redundanz aufweist: Durch die maximale Erweiterung der **1**-Knoten von $\mathcal{F}_0(X_0)$ werden auch Atome der Ausgangsformel neu in die Baumstruktur mit aufgenommen (1',2',1",2"), obwohl diese in allen **1**-Pfaden von $\mathcal{F}_0(X_0)$ auftauchen und somit in ihrer Deutung bei "Betreten" der Erweiterung bereits festgelegt sind. Für größere Probleme stellt dies einen gravierenden Nachteil dar, der jedoch durch das im nächsten Abschnitt beschriebene Verfahren behoben werden kann.

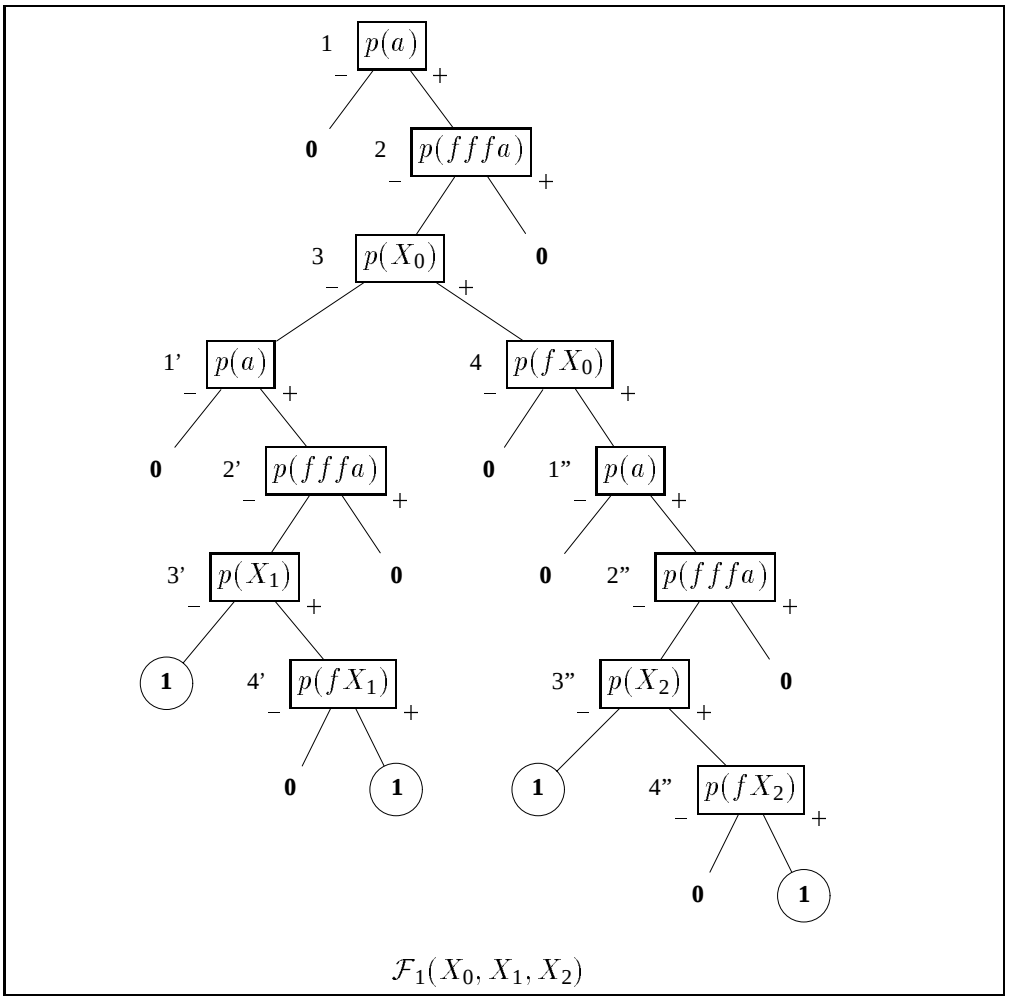

Abbildung 3.2: Erweiterung mit unabhängigen Instanzen $\mathcal{F}'_0(X_1)$ und $\mathcal{F}''_0(X_2)$

3.5 Shannon-Graphen mit selektiven Erweiterungen

Mit dem Verfahren der maximalen Erweiterungen kann eine neue Instanz einer $\forall$-quantifizierten Variablen der Ausgangsformel $\forall\bar{x}F(\bar{x})$ nur erhalten werden, indem man an einem **1**-Knoten mit dem gesamten initialen Graphen $\mathcal{F}_0(\bar{X})$ erweitert.

Dies liegt daran, daß durch die Bildung einer pränexen Normalform $\forall\bar{x}F(\bar{x})$ aus der skolemisierten Formel $S^\star$, in der noch $\forall$-quantifizierte Unterformeln auftauchen, eine "große" $\forall$-quantifizierte Formel $\forall\bar{x}F(\bar{x})$ entsteht. Die quantorenfreie Formel $F(\bar{x})$ wird dann beim Verfahren der maximalen Erweiterungen mittels der aussagenlogischen *conv*-Funktion in einen logisch äquivalenten Shannon-Graphen umgewandelt.

Besser ist es jedoch, die $\forall$-Quantoren in $S^\star$ "vor Ort" stehen zu lassen, um so die vorhandene Skopus-Information auszunutzen. Allerdings müssen wir dann das Verfahren um die Behandlung von $\forall$-quantifizierte Unterformeln in geeigneter Weise erweitern. Dies geschieht, indem wir die Einschränkung abschwächen, daß die Bedingung einer Shannon-Formel nur atomare Formeln (bzw. das CUT-Symbol "!") enthalten darf. Wir

lassen nun auch sogenannte γ-*Shannon-Formeln* als Bedingung in einer Shannon-Formel zu.[9]

Definition 3.13 (γ-Shannon-Formeln, γ-Graphen)
Die Menge der γ-Shannon-Formeln $\mathcal{SH}_\gamma$ ist definiert als die kleinste Menge, für die gilt[10]

(1) $\mathcal{SH}_\gamma \supset \mathcal{SH}_!$

(2) Wenn $\mathcal{G}, \mathcal{S}_0, \mathcal{S}_1 \in \mathcal{SH}_\gamma$, dann ist $sh(\forall \bar{x}\, \mathcal{G}, \mathcal{S}_0, \mathcal{S}_1) \in \mathcal{SH}_\gamma$, wobei $\bar{x}$ freie Variablen aus $\mathcal{G}$ sind.

Häufig nennen wir γ-Shannon-Formeln auch γ-Shannon-Graphen oder kurz γ-Graphen.

Offensichtlich gelten auch für Formeln aus $\mathcal{SH}_\gamma$ die Kompositionsregeln[11]

$$(REP_\vee): \quad \mathcal{A} \vee \mathcal{B} \Leftrightarrow \mathcal{A}[\frac{\mathbf{0}}{\mathcal{B}}] \quad \text{und} \quad (REP_\wedge): \quad \mathcal{A} \wedge \mathcal{B} \Leftrightarrow \mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}].$$

Wir erweitern die Definition $DNF_{\mathbf{1}}$ von **1**-Pfaden in naheliegender Weise auf Formeln aus $\mathcal{SH}_\gamma$, so daß in **1**-Pfaden neben Literalen nun auch Formeln $\forall \bar{x} \mathcal{G}(\bar{x})$ und $\neg \forall \bar{x} \mathcal{G}(\bar{x})$ vorkommen. Man zeigt ganz analog zum aussagenlogischen Fall, daß die *DNF-Darstellung* weiterhin für γ-Shannon-Formeln gilt, d.h. für alle $\mathcal{S} \in \mathcal{SH}_\gamma$ ist

$$\mathcal{S} \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S})$$

Hierbei ist zu beachten, daß $DNF_{\mathbf{1}}(\mathcal{S})$ nach wie vor eine Disjunktion der **1**-Pfade ist. Da in diesen nun aber zusammengesetzte (γ-Shannon-)Formeln auftreten, ist $DNF_{\mathbf{1}}(\mathcal{S})$ keine disjunktive *Normalform* mehr.

Sei nun also ein Satz S in $SNF^\star$ gegeben. Dann ist S in Negations-Normalform und enthält nur noch $\forall$-Quantoren. Mit der nachfolgenden Funktion $conv_\gamma$ können wir den *initialen Graphen* $\mathcal{S}_0 = conv_\gamma(S)$ konstruieren. Dieser enthält keine freien Variablen, ist also selbst ein Satz.

Definition 3.14 (Convert_γ)
Für S in $SNF^\star$ ist

$$conv_\gamma(S) = \begin{cases} sh(A, \mathbf{0}, \mathbf{1}) & \text{wenn } S = A, A \in \mathbf{For}_{At} \\ sh(A, \mathbf{1}, \mathbf{0}) & \text{wenn } S = \neg A, A \in \mathbf{For}_{At} \\ conv_\gamma(A)[\frac{\mathbf{1}}{conv_\gamma(B)}] & \text{wenn } S = A \wedge B \\ conv_\gamma(A)[\frac{\mathbf{0}}{conv_\gamma(B)}] & \text{wenn } S = A \vee B \\ sh(\forall \bar{x}\ conv_\gamma(A), \mathbf{0}, \mathbf{1}) & \text{wenn } S = \forall \bar{x} A \end{cases}$$

[9] Die Bezeichnung ist eine Anlehnung an die γ-*Formeln* des Tableaukalk üls.

[10] Um nicht allzuviele verschiedene Formelmengen betrachten zu m üssen, schließen wir in dieser Definition CUT-Shannon-Formeln mit ein.

[11] Hier tritt im übrigen erstmals der Fall auf, der die Blatt-Substitution $\mathcal{A}[\frac{\mathbf{0}}{\mathcal{B}}]$ bzw. $\mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$ von der "gewöhnlichen" Substitution unterscheidet: Die Ersetzung wird nur im zweiten und dritten Argument einer Shannon-Formel durchgeführt.

Mit der Gültigkeit von $(REP_\vee)$ und $(REP_\wedge)$ folgt per struktureller Induktion, daß $conv_\gamma$ einen Satz S in *SNF** in eine logisch äquivalente γ-Shannon-Formel umwandelt:

Korollar 3.15
Für alle S in *SNF** ist $S \Leftrightarrow conv_\gamma(S)$.

Beispiel 3.16
Zu der Formel S in *SNF**

$$(\forall x\ p(x) \vee \forall y\ \neg q(y)) \wedge \neg p(a) \wedge q(b)$$

ist die mit $conv_\gamma$ konstruierte γ-Shannon-Formel $conv_\gamma(S) =$

$$\begin{array}{l} sh(\forall x\ sh(p(x), \mathbf{0}, \mathbf{1}), \\ \qquad sh(\forall y\ sh(q(y), \mathbf{1}, \mathbf{0}), \\ \qquad\qquad \mathbf{0}, \\ \qquad\qquad sh(p(a), sh(q(b), \mathbf{0}, \mathbf{1}), \mathbf{0})), \\ \qquad sh(p(a), sh(q(b), \mathbf{0}, \mathbf{1}), \mathbf{0})) \end{array}$$

3.5.1 Visualisierung von γ-Graphen

Wesentlich übersichtlicher als die Formelschreibweise ist die Darstellung von γ-Graphen in Form von Binärbäumen. Neben den Blättern **0** und **1**, den bereits bekannten Typen von Nichtterminal-Knoten $sh(A, \mathcal{B}, \mathcal{C})$, wobei $A \in \mathbf{For}_{At}$ *(Literal-Knoten)* und $sh(!, \mathcal{B}, \mathcal{C})$ *(CUT-Knoten)*, treten nun auch sogenannte γ-*Knoten* $sh(\forall \bar{x}\ \mathcal{G}, \mathcal{B}, \mathcal{C})$ auf. Die innerhalb eines γ-Knotens auftretenden "γ-Untergraphen" $\mathcal{G}$ stellen wir dabei als eigenständige Binärbäume dar.

Abbildung 3.3 zeigt diese Darstellung für den γ-Graphen $\mathcal{S} = conv_\gamma(S)$ aus Beispiel 3.16. Obwohl wir γ-Graphen hier und im folgenden in Form von Binärbäumen veranschaulichen, werden diese nach wie vor als gerichtete, azyklische Graphen repräsentiert. Mit Hilfe der Nummern neben den Nichtterminal-Knoten soll die Identifikation dieser Graphstruktur erleichtert werden. Man erkennt, daß zur Repräsentation von $\mathcal{S}$ neben den Blättern **0** und **1** vier Literal-Knoten nötig sind, die mit den entsprechenden Literalen der Ausgangsformel S identifiziert werden können. Außerdem werden noch zwei γ-Knoten benötigt, die in Abbildung 3.3 als doppelt umrahmte Rechtecke dargestellt sind.

Wir zeigen nun anhand eines Beispiels, wie wir aus $\mathcal{S}$ einen geschlossenen γ-Graphen erhalten können. Zuvor definieren wir, völlig analog zum vorigen Abschnitt 3.4:

Definition 3.17 (Geschlossene γ-Graphen)
Ein γ-Graph $\mathcal{S}(\bar{x})$ mit den freien Variablen $\bar{x}$[12] *heißt* geschlossen, *wenn es eine Substitution σ gibt, so daß alle* **1***-Pfade in $\mathcal{S}(\bar{x})\sigma$ geschlossen sind.*

Ein **1***-Pfad heißt* geschlossen, *wenn er komplementäre Literale enthält.*

[12] $\bar{x}$ kann auch das "leere Tupel" sein, dann ist $\mathcal{S}(\bar{x})$ ein Satz.

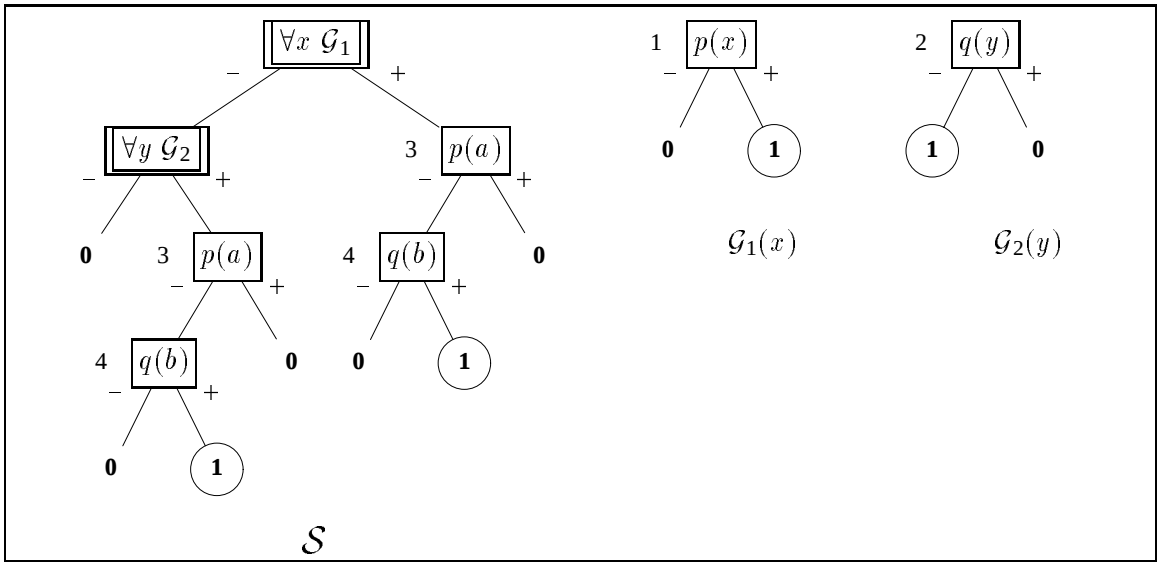

Abbildung 3.3: Initialer Shannon-Graph $\mathcal{S}$ mit γ-Untergraphen $\mathcal{G}_1(x), \mathcal{G}_2(y)$

Wir kehren zu obigem Beispiel zurück: Ist ein **1**-Pfad $\pi \in DNF_{\mathbf{1}}(\mathcal{S})$ von $\mathcal{S}$ nicht geschlossen und gilt $\forall x\ \mathcal{G}_i(x) \in \pi$, so können wir π um eine Instanz $\mathcal{G}_i(X)$ des γ-Untergraphen $\mathcal{G}_i$ mit der neuen und damit freien Variablen X erweitern und versuchen, die Menge der neu entstandenen Pfade abzuschließen.

In Abbildung 3.3 ist der Pfad zum linken **1**-Blatt von $\mathcal{S}$:

$$\{\neg\forall x\ \mathcal{G}_1(x), \forall y\ \mathcal{G}_2(y), \neg p(a), q(b)\}$$

Diesen **1**-Pfad π erweitern wir, indem wir anstelle des **1**-Blattes den γ-Untergraphen $\mathcal{G}_2(Y)$ mit der freien Variablen Y einsetzen. Wir erhalten so den erweiterten **1**-Pfad

$$\{\neg\forall x\ \mathcal{G}_1(x), \forall y\ \mathcal{G}_2(y), \neg p(a), q(b), \neg q(Y)\}$$

der mit der Substitution $\sigma = \{Y/b\}$ geschlossen werden kann. Analog verfahren wir mit dem zweiten **1**-Pfad, dessen **1**-Blatt wir durch eine Instanz $\mathcal{G}_1(X)$ von $\mathcal{G}_1$ ersetzen. Der erweiterte Pfad ist dann

$$\{\forall x\ \mathcal{G}_1(x), \neg p(a), q(b), p(X)\}$$

Abbildung 3.4 zeigt den sich ergebenden, für die Substitution $\sigma = \{X/a, Y/b\}$ geschlossenen Shannon-Graphen $\mathcal{S}'$.

An dieser Stelle ist nochmals eine Bemerkung zur verwendeten Graphstruktur angebracht: Der γ-Graph $\mathcal{S}$ aus Abbildung 3.3 besitzt die beiden erwähnten **1**-Pfade, die in der *Graph*-Darstellung im selben **1**-Knoten enden. Es ist nun sowohl anschaulicher als auch in der Sprechweise wesentlich einfacher, wenn wir uns $\mathcal{S}$ als Binärbaum vorstellen. Dann können wir sagen, wir hätten *das linke* bzw. *das rechte* **1**-Blatt durch die entsprechenden γ-Untergraphen ersetzt. Auf die zughörige Repräsentation in Graphform übertragen

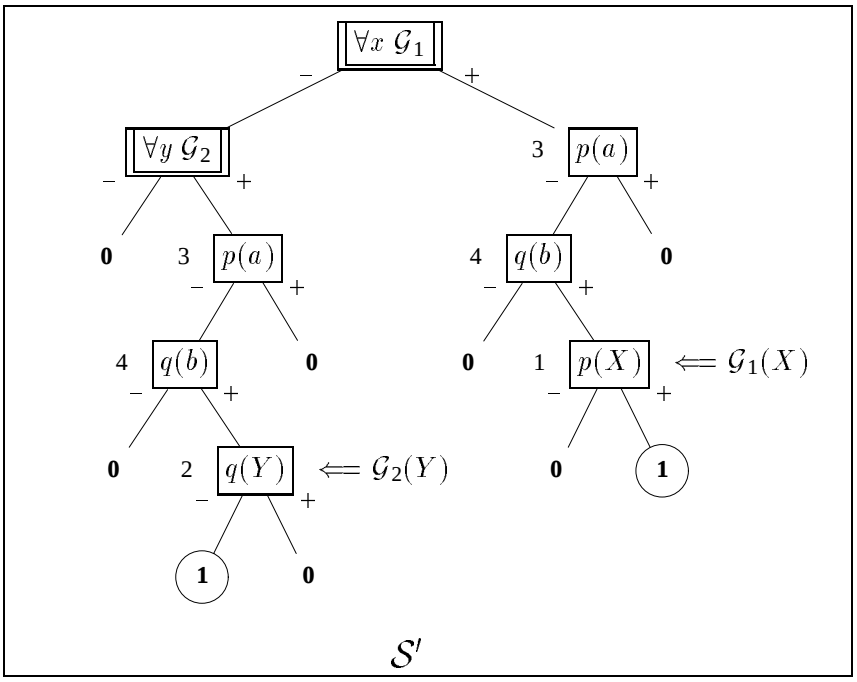

Abbildung 3.4: Mit selektiven Erweiterungen geschlossener Shannon-Graph $\mathcal{S}'$

heißt dies, daß die beiden Kanten auf den eindeutigen **1**-Knoten ersetzt werden durch Kanten auf die entsprechenden γ-Untergraphen.

Im folgenden wollen wir die vereinfachte Sprechweise bevorzugen und uns γ-Graphen als Binärbäume vorstellen. Die tatsächliche Realisierung des Verfahrens arbeitet dagegen unverändert auf einer kompakten Graphstruktur, woran – sollten wir Gefahr laufen, dies zu vergessen – noch immer die Bezeichnung γ-*Graph* erinnert.

Nachdem wir informell gezeigt haben, wie die Erweiterung eines **1**-Pfades mit einer neuen Instanz eines γ-Untergraphen vorgenommen wird, geben wir nachfolgend eine formale Definition:

Definition 3.18 (γ-Extension)
Sei $\mathcal{S}(\bar{x}) \in \mathcal{SH}_\gamma$ ein γ-Shannon-Graph mit den freien Variablen $\bar{x}$, $\pi \in DNF_{\mathbf{1}}(\mathcal{S}(\bar{x}))$ der zu einem **1***-Knoten von $\mathcal{S}(\bar{x})$ gehörende* **1***-Pfad. Weiter sei $\forall \bar{y}\ \mathcal{G}(\bar{y}) \in \pi$. Dann erhalten wir die* γ-Extension *$\mathcal{S}'(\bar{x}, \bar{Y})$ von $\mathcal{S}(\bar{x})$ durch Ersetzen dieses* **1***-Knotens durch den γ-Untergraphen $\mathcal{G}(\bar{Y})$ mit den neuen, freien Variablen $\bar{Y}$:*

$$Ext_\gamma^\pi : \mathcal{S}(\bar{x}) \mapsto \mathcal{S}'(\bar{x}, \bar{Y}) = \mathcal{S}(\bar{x})\lfloor \tfrac{\mathbf{1}}{\mathcal{G}(\bar{Y})} \rfloor^\pi$$

Durch die Schreibweise $\lfloor \frac{\mathbf{1}}{\mathcal{G}(\bar{Y})} \rfloor^\pi$ deuten wir an, daß nur ein spezieller **1***-Knoten (mit zugehörigem Pfad π) ersetzt wird und nicht simultan alle* **1***-Knoten, wie bei $\mathcal{S}(\bar{x})[\frac{\mathbf{1}}{\mathcal{G}(\bar{Y})}]$.*

Wir nennen die γ-Extension $\mathcal{S}' = Ext_\gamma^\pi(\mathcal{S})$ auch *selektive Erweiterung* von $\mathcal{S}$: Zu einem **1**-Knoten mit Pfad π können wir einen γ-Untergraphen $\mathcal{G} \in \pi$ *auswählen*, mit dem erweitert

werden soll und müssen nicht, wie im Falle der maximalen Erweiterungen, mit dem gesamten initialen Graphen erweitern.

Wie sieht nun die Menge der **1**-Pfade einer γ-Extension, etwa für $\mathcal{S}' = \mathcal{S}\lfloor\frac{\mathbf{1}}{\mathcal{G}}\rfloor^{\pi_j}$ aus? Sei $\pi_j \in DNF_{\mathbf{1}}(\mathcal{S}) = \{\pi_1, \ldots, \pi_n\}$. Alle **1**-Pfade $\pi_i \neq \pi_j$ sind natürlich auch in $DNF_{\mathbf{1}}(\mathcal{S}')$; anstelle von π_j treten in $DNF_{\mathbf{1}}(\mathcal{S}')$ nun aber die **1**-Pfade der Menge $\{\pi_j\}\otimes DNF_{\mathbf{1}}(\mathcal{G})$ auf.

Beispiel:

Für $\pi_j = \{\forall x\, \mathcal{G}_1(x), \neg p(a), q(b)\}$ aus Abbildung 3.4 und $DNF_{\mathbf{1}}(\mathcal{G}_1(x)) = \{\{p(X)\}\}$ enthält $\{\pi_j\}\otimes DNF_{\mathbf{1}}(\mathcal{G}_1(x))$ nur den einen, erweiterten **1**-Pfad $\{\forall x\, \mathcal{G}_1(x), \neg p(a), q(b), p(X)\}$.

Nach Definition der **1**-Pfade gilt allgemein:

$$DNF_{\mathbf{1}}(\mathcal{S}\lfloor\frac{\mathbf{1}}{\mathcal{G}}\rfloor^{\pi_j}) \quad = \quad DNF_{\mathbf{1}}(\mathcal{S})\backslash\{\pi_j\} \cup \{\pi_j\}\otimes DNF_{\mathbf{1}}(\mathcal{G}) \tag{3.1}$$

In einem **1**-Pfad eines γ-Graphen kann auch die Negation $\neg\forall x\mathcal{G}$ eines γ-Untergraphen $\forall x\mathcal{G}$ auftreten. Hierbei handelt es sich eigentlich um eine $\exists$-quantifizierte Formel. Wie sich noch zeigen wird, können diese negativen Auftreten von γ-Untergraphen in einem Pfad ignoriert werden. Wenn also ein γ-Graph überhaupt geschlossen werden kann, dann reichen hierzu die γ-Extensionen aus.

Bevor wir einen γ-Graphen $\mathcal{S}$ mittels γ-Extension erweitern, können wir zunächst versuchen, einen oder mehrere **1**-Pfade von $\mathcal{S}$ mit einer Substitution σ abzuschließen. Hierbei müssen wir darauf achten, daß σ *frei* für $\mathcal{S}$ ist, d.h. anschaulich, daß keine freie Variable durch Anwendung von σ durch einen Quantor aus $\mathcal{S}$ gebunden wird. Formal ausgedrückt:

Definition 3.19 (Freie Substitutionen)
Ein Term t heißt frei *für die Variable x in der Formel F, wenn keine freie Variable aus t bei Einsetzung für x in F von einem Quantor aus F gebunden wird.*

Eine Substitution σ heißt frei *für eine Formel F mit den freien Variablen $\bar{x}$, wenn für jede Variable x_i aus $\bar{x}$ der Term $x_i\sigma$ frei für x_i in F ist.*

Demnach sind also insbesondere alle Grundsubstitutionen frei. Wir können nun das Gesamtverfahren der selektiven Erweiterungen angeben.

Definition 3.20 (Selektive Erweiterungen)
Um die Unerfüllbarkeit eines Satzes S der Prädikatenlogik nachzuweisen, gehe man wie folgt vor:

1. Vorverarbeitung:[13]
   (a) *Konstruiere zu S die Formel $S^\star$ in SNF*.*
   (b) *Konstruiere den initialen γ-Graphen $\mathcal{S}_0 = conv_\gamma(S^\star)$.*
2. Beweissuche: *Solange $\mathcal{S}_i$ mit den freien Variablen $\bar{x}$ nicht geschlossen und einer der folgenden Schritte anwendbar ist:*

[13] Der Schritt 1.(a) kann problemlos in die Funktion $conv_\gamma$ integriert werden (siehe Abschnitt 4.2.3).

(a) Schließe einen **1***-Pfad durch eine Substitution* σ*, die frei für* $\mathcal{S}_i$ *ist:*

$$\mathcal{S}_{i+1} = \mathcal{S}_i\sigma$$

(b) oder erweitere $\mathcal{S}_i$ *durch* γ*-Extension*

$$\mathcal{S}_{i+1} = Ext^{\pi}_{\gamma}(\mathcal{S}_i)$$

Man stellt fest, daß sich das Verfahren der maximalen Erweiterungen als Spezialfall der selektiven Erweiterungen ergibt, wenn in Schritt 1.(a) $S^{\star}$ noch pränex gemacht wird. Wir zeigen im folgenden, daß das Verfahren der selektiven Erweiterungen korrekt ist, woraus dann insbesondere auch die Korrektheit der in Abschnitt 3.4.1 beschriebenen Modifikation der maximalen Erweiterungen folgt.

3.5.2 Korrektheit der selektiven Erweiterungen

Der Nachweis der Korrektheit der selektiven Erweiterungen ist recht einfach und eher technischer Natur. Das Vorgehen ist ähnlich wie beim Nachweis der Korrektheit des Tableaukalküls mit freien Variablen. Im wesentlichen haben wir zu zeigen, daß ein *erfüllbarer* Shannon-Graph auch nach Anwendung einer freien Substitution bzw. einer γ-Extension erfüllbar bleibt; hierzu definieren wir

Definition 3.21
Ein γ*-Shannon-Graph* $\mathcal{S}(\bar{x})$ *mit den freien Variablen* $\bar{x}$ *heißt* erfüllbar, *wenn er ein* Modell $\mathcal{D}$ *hat, d.h. wenn für eine Struktur* $\mathcal{D}$ *und* alle *Variablenbelegungen* β *gilt:*

$$val_{\mathcal{D},\beta}(\mathcal{S}(\bar{x})) = \mathsf{W}$$

Zum Nachweis, daß die Anwendung einer Substitution erfüllbarkeitserhaltend ist, stützen wir uns auf das folgende bekannte *Substitutionslemma.* Ein Beweis für dessen Gültigkeit findet sich z.B. in [Andrews, 1986, Seite 79ff], allerdings in einer gänzlich anderen Notation. Die Aussage läßt sich offensichtlich problemlos von Formeln aus **For** auf Formeln aus $\mathbf{For}_{\mathcal{SH}}$ erweitern.[14]

Lemma 3.22 (Substitutionslemma)
Sei $A \in \mathbf{For}_{\mathcal{SH}}$, $\sigma = \{\bar{y}/\bar{t}\}$ eine Substitution **Var**→**Term**, die frei für A ist. Zu gegebener Struktur $\mathcal{D} = \langle \mathbf{D}, \mathbf{I}\rangle$ und Variablenbelegung β sei $\beta[\bar{y}/\beta_{\mathbf{I}}(\bar{t})]$ die Modifikation von β so, daß die Variablen $\bar{y}$ als die Termauswertung $\beta_{\mathbf{I}}(\bar{t})$ interpretiert werden. Dann gilt

$$val_{\mathcal{D},\beta}(A\{\bar{y}/\bar{t}\}) = val_{\mathcal{D},\beta[\bar{y}/\beta_{\mathbf{I}}(\bar{t})]}(A)$$

Nachdem die technischen Vorbedingungen geschaffen sind, können wir sofort das folgende Lemma beweisen.

Lemma 3.23
Wenn $\mathcal{S}(\bar{x})$ erfüllbar und σ eine für $\mathcal{S}(\bar{x})$ freie Substitution ist, dann ist $\mathcal{S}(\bar{x})\sigma$ erfüllbar.

[14] Man betrachte $sh(A,B,C)$ einfach als Abkürzung für $\neg A \wedge B \vee A \wedge C$.

BEWEIS
Sei $\sigma = \{\bar{y}/\bar{t}\}$ eine beliebige Substitution, die frei für $\mathcal{S}(\bar{x})$ ist. Nach dem obigem Substitutionslemma gilt für alle Strukturen $\mathcal{D}$ und Variablenbelegungen β:

$$val_{\mathcal{D},\beta}(\mathcal{S}(\bar{x})\sigma) = val_{\mathcal{D},\beta[\bar{y}/\beta_{\mathbf{I}}(\bar{t})]}(\mathcal{S}(\bar{x}))$$

$\mathcal{S}(\bar{x})$ hat nach Voraussetzung ein Modell $\mathcal{D}$. Also gilt $\mathcal{S}(\bar{x})$ für alle Variablenbelegungen unter $\mathcal{D}$, insbesondere für die Modifikation $\beta[\bar{y}/\beta_{\mathbf{I}}(\bar{t})]$ von β. Damit gilt $\mathcal{S}(\bar{x})\sigma$ unter β und, da β beliebig war, ist $\mathcal{D}$ auch ein Modell von $\mathcal{S}(\bar{x})\sigma$. ■

Korollar 3.24
Wenn $\mathcal{S}(\bar{x})$ geschlossen ist, dann ist $\mathcal{S}(\bar{x})$ unerfüllbar.

BEWEIS
Wenn $\mathcal{S}(\bar{x})$ geschlossen, dann gibt es eine Substitution σ so, daß alle **1**-Pfade der Disjunktion $DNF_{\mathbf{1}}(\mathcal{S}(\bar{x})\sigma)$ geschlossen also auch unerfüllbar sind.[15] Dann ist wegen $\mathcal{S}(\bar{x})\sigma \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S}(\bar{x})\sigma)$ aber auch $\mathcal{S}(\bar{x})\sigma$ unerfüllbar und nach Lemma 3.23 schließlich auch $\mathcal{S}(\bar{x})$. ■

Lemma 3.25 (Korrektheit der γ-Extension)
Sei $\mathcal{S}(\bar{x})$ erfüllbar. Dann ist die γ-Extension von $\mathcal{S}(\bar{x})$ ebenfalls erfüllbar.

BEWEIS
Sei $DNF_{\mathbf{1}}(\mathcal{S}(\bar{x})) = \{\pi_1, \ldots, \pi_n\}$ und $\mathcal{S}'(\bar{x}, \bar{Y}) = \mathcal{S}(\bar{x})\lfloor\frac{\mathbf{1}}{\mathcal{G}(\bar{Y})}\rfloor^{\pi_j}$ die γ-Extension von $\mathcal{S}(\bar{x})$, wobei $\forall\bar{y}\mathcal{G}(\bar{y}) \in \pi_j$. Da $\mathcal{S}(\bar{x})$ nach Voraussetzung erfüllbar ist, gibt es eine Struktur $\mathcal{D}$, die Modell von $\mathcal{S}(\bar{x})$ ist. Wir betrachten eine beliebige Variablenbelegung β.

Es gilt $val_{\mathcal{D},\beta}(\mathcal{S}(\bar{x})) = \mathsf{W}$, also auch $val_{\mathcal{D},\beta}(\pi_1 \vee \ldots \vee \pi_n) = \mathsf{W}$. Demnach gibt es einen Pfad $\pi_i \in DNF_{\mathbf{1}}(\mathcal{S}(\bar{x}))$ mit $val_{\mathcal{D},\beta}(\pi_i) = \mathsf{W}$.

Ist $\pi_i \neq \pi_j$, so ist π_i auch ein **1**-Pfad von $\mathcal{S}'(\bar{x}, \bar{Y})$ und damit $val_{\mathcal{D},\beta}(\mathcal{S}'(\bar{x}, \bar{Y})) = \mathsf{W}$. Falls aber $\pi_i = \pi_j$, gilt wegen $\forall\bar{y}\mathcal{G}(\bar{y}) \in \pi_j$ auch $val_{\mathcal{D},\beta}(\forall\bar{y}\mathcal{G}(\bar{y})) = \mathsf{W}$ und so insbesondere $val_{\mathcal{D},\beta}(\mathcal{G}(\bar{Y})) = \mathsf{W}$. Da $\mathcal{G}(\bar{Y}) \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{G}(\bar{Y}))$, gibt es einen **1**-Pfad μ in $\mathcal{G}(\bar{Y})$ mit $val_{\mathcal{D},\beta}(\mu) = \mathsf{W}$. $\pi_j \cup \mu$ ist also ein **1**-Pfad von $\mathcal{S}'(\bar{x}, \bar{Y})$,[16] also gilt wiederum $val_{\mathcal{D},\beta}(\mathcal{S}'(\bar{x}, \bar{Y})) = \mathsf{W}$.

■

Satz 3.26 (Korrektheit des Beweisverfahrens mit selektiven Erweiterungen)
Sei S ein Satz der Prädikatenlogik, $\mathcal{S}_n$ eine nach Definition 3.20 konstruierte selektive Erweiterung von S. Dann gilt:

Wenn $\mathcal{S}_n$ geschlossen ist, dann ist S unerfüllbar.

BEWEIS
Angenommen $\mathcal{S}_n$ ist geschlossen und S erfüllbar. Dann ist auch $S^\star$, und wegen $S^\star \Leftrightarrow conv_\gamma(S^\star) = \mathcal{S}_0$, auch der initiale Graph $\mathcal{S}_0$ erfüllbar. $\mathcal{S}_n$ ist durch endlich häufige Anwendung von (freien) Substitutionen und γ-Extensionen aus $\mathcal{S}_0$ entstanden. Nach Lemma 3.23 und Lemma 3.25 ist schließlich auch $\mathcal{S}_n$ erfüllbar. Andererseits kann $\mathcal{S}_n$ nach Korollar 3.24 nicht erfüllbar sein; Widerspruch. ■

[15] Zur Erinnerung: Ein Pfad heißt geschlossen, wenn er komplementäre Literale L und $\neg L$ enthält. Da Pfade konjunktiv interpretiert werden, ist ein geschlossener Pfad immer unerfüllbar.

[16] Siehe (3.1) auf Seite 50. NB: $\pi_j \cup \mu$ ist als Vereinigung zweier Pfade eine Konjunktion.

3.5.3 Vollständigkeit der selektiven Erweiterungen

Überblick

Wir möchten zeigen, daß ein unerfüllbarer γ-Graph mit Hilfe der selektiven Erweiterungen aus Definition 3.20 immer geschlossen werden kann. In diesem Verfahren verwenden wir die negativen Auftreten $\neg\forall x\mathcal{G}(x)$ eines γ-Untergraphen in einem Pfad π weder zum Abschluß, noch zur Erweiterung von π.

Im ersten Teil dieses Abschnitts zeigen wir nun, daß wir in Schritt *1.(b)* aus Definition 3.20 $conv_\gamma$ durch eine Variante $conv_\gamma^+$ ersetzen dürfen. Die mit $conv_\gamma^+$ gebildeten γ-Graphen haben dabei die gleichen **1**-Pfade wie die mit $conv_\gamma$ konstruierten, nur treten darin keine negierten γ-Untergraphen $\neg\forall x\mathcal{G}(x)$ mehr auf. Der Korrektheitsbeweis kann für das so modifizierte Verfahren der selektiven Erweiterungen wörtlich übernommen werden. Praktisch heißt dies aber nichts anderes, als daß wir weiterhin die ursprüngliche Version mit $conv_\gamma$ verwenden dürfen, wobei wir die negierten Auftreten von γ-Untergraphen für die Beweissuche ignorieren.

Im zweiten Teil beweisen wir schließlich die Vollständigkeit des Verfahrens der selektiven Erweiterungen. Hierbei machen wir von dem Umstand Gebrauch, daß in **1**-Pfaden nur noch positive γ-Untergraphen auftreten.

Elimination von negierten Auftreten von γ-Graphen

Die Definition von $DNF_{\mathbf{1}}$ spiegelt für beliebige Shannon-Formeln deren Semantik exakt wider und gestattet es, jede Shannon-Formel als disjunktive Form (im aussagenlogischen Fall sogar als disjunktive Normalform) zu betrachten. Insbesondere wird nach dieser Definition für eine Formel $sh(\mathcal{G},\mathcal{A},\mathcal{B})$ die Negation $\neg\mathcal{G}$ in alle **1**-Pfade, die durch den negativen Ausgang $\mathcal{A}$ "laufen", mit aufgenommen. Wenn $\mathcal{G}$ ein Atom ist, so soll dies auch weiterhin gelten, während wir im Falle eines γ-Untergraphen $\mathcal{G}$ dieses im folgenden gerade ausschließen wollen.

Zu diesem Zweck definieren wir hilfsweise einen neuen dreistelligen Junktor sh^+, dessen Semantik genau beschreibt, was wir beabsichtigen, und der uns in die Lage versetzt, auf syntaktischer Ebene zu erkennen, an welchen Knoten wir *keine* Erweiterung der **1**-Pfade mit der negierten Bedingung wünschen.

Definition 3.27 (Der Junktor sh^+)
Wir erweitern die Menge der γ-Graphen $\mathcal{SH}_\gamma$ zur Menge $\mathcal{SH}^+$, indem wir anstelle des Shannon-Junktors zusätzlich den Junktor sh^+ zum Formelaufbau zulassen. Dessen Wahrheitswert ist definiert als

$$val(sh^+(A,B,C)) \;:=\; val(B \vee A\wedge C)$$

Weiterhin müssen wir die Definition von **1***-Pfaden auf Shannon-Formeln mit sh^+ erweitern:*

$$DNF_{\mathbf{1}}(sh^+(A,\mathcal{B},\mathcal{C})) \;:=\; DNF_{\mathbf{1}}(\mathcal{B}) \cup \{\{A\}\} \otimes DNF_{\mathbf{1}}(\mathcal{C})$$

Mittels struktureller Induktion zeigt man leicht, daß diese Definition von $DNF_{\mathbf{1}}$ die Semantik von sh^+ korrekt widerspiegelt, d.h. es gilt für alle $\mathcal{S} \in \mathcal{SH}^+$

$$\mathcal{S} \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S})$$

Um nun sicherzustellen, daß in **1**-Pfaden keine negierten γ-Graphen vorkommen, ersetzen wir in der Definition von $conv_\gamma$ im Falle einer $\forall$-quantifizierten Formel den Junktor sh durch sh^+:

Definition 3.28 (Convert$^+_\gamma$)
Für S in SNF* ist

$$conv^+_\gamma(S) = \begin{cases} sh(A, \mathbf{0}, \mathbf{1}) & \text{wenn } S = A, A \in \mathbf{For}_{At} \\ sh(A, \mathbf{1}, \mathbf{0}) & \text{wenn } S = \neg A, A \in \mathbf{For}_{At} \\ conv^+_\gamma(A)[\frac{\mathbf{1}}{conv^+_\gamma(B)}] & \text{wenn } S = A \wedge B \\ conv^+_\gamma(A)[\frac{\mathbf{0}}{conv^+_\gamma(B)}] & \text{wenn } S = A \vee B \\ sh^+(\forall \bar{x}\ conv^+_\gamma(A), \mathbf{0}, \mathbf{1}) & \text{wenn } S = \forall \bar{x} A \end{cases}$$

In welchem semantischen Zusammenhang stehen nun $sh(\mathcal{A}, \mathcal{B}, \mathcal{C})$ und $sh^+(\mathcal{A}, \mathcal{B}, \mathcal{C})$? Zwar gilt für alle $\mathcal{A}, \mathcal{B}, \mathcal{C} \in \mathcal{SH}^+$

$$sh(\mathcal{A}, \mathcal{B}, \mathcal{C}) \Rightarrow sh^+(\mathcal{A}, \mathcal{B}, \mathcal{C})$$

aber *nicht* notwendigerweise:[17]

$$sh(\mathcal{A}, \mathcal{B}, \mathcal{C}) \Leftarrow sh^+(\mathcal{A}, \mathcal{B}, \mathcal{C})$$

Die Rückrichtung gilt genau dann nicht, wenn $\mathcal{A} \wedge \mathcal{B} \wedge \neg \mathcal{C}$ gilt (man zeigt dies leicht z.B. mit Hilfe einer Wahrheitstabelle). Wenn wir allerdings voraussetzen, daß $\mathcal{B} \Rightarrow \mathcal{C}$ gilt, dann kann $\mathcal{A} \wedge \mathcal{B} \wedge \neg \mathcal{C}$ nicht gelten. In diesem Falle dürfen wir sh durch sh^+ ersetzen, denn beide Schreibweisen sind logisch äquivalent. Nachfolgend zeigen wir, daß diese Ersetzung insbesondere in der durch $conv^+_\gamma$ vorgenommenen Weise zulässig ist.

Lemma 3.29
Sei $sh(\mathcal{G}, \mathcal{B}, \mathcal{C}) \in \mathcal{SH}_\gamma$ und $\mathcal{B} \Rightarrow \mathcal{C}$. Weiter sei

$$sh(\mathcal{G}, \mathcal{B}', \mathcal{C}') = \begin{cases} sh(\mathcal{G}, \mathcal{B}, \mathcal{C})[\frac{\mathbf{0}}{\mathcal{D}}] \\ \text{oder} \\ sh(\mathcal{G}, \mathcal{B}, \mathcal{C})[\frac{\mathbf{1}}{\mathcal{D}}] \end{cases}$$

Dann gilt $\mathcal{B}' \Rightarrow \mathcal{C}'$ und $sh(\mathcal{G}, \mathcal{B}', \mathcal{C}') \Leftrightarrow sh^+(\mathcal{G}, \mathcal{B}', \mathcal{C}')$.

[17] $A \Rightarrow B$ steht für: $val_{\mathfrak{D},\beta}(A) = \mathsf{W}$ impliziert $val_{\mathfrak{D},\beta}(B) = \mathsf{W}$.

BEWEIS
Im Falle der Ersetzung eines **0**-Blattes gilt $\mathcal{B}' \Leftrightarrow \mathcal{B} \vee \mathcal{D}$ und $\mathcal{C}' \Leftrightarrow \mathcal{C} \vee \mathcal{D}$ und somit $\mathcal{B}' \Rightarrow \mathcal{C}'$. Entsprechendes gilt bei Ersetzung eines **1**-Blattes. Dann gilt auch $sh(\mathcal{G}, \mathcal{B}', \mathcal{C}') \Leftrightarrow sh^+(\mathcal{G}, \mathcal{B}', \mathcal{C}')$. ■

Wendet man diese Aussage induktiv an, so haben wir folgendes gezeigt: Ausgehend von einer Shannon-Formel $sh(\mathcal{G}, \mathcal{B}, \mathcal{C})$, für die $\mathcal{B} \Rightarrow \mathcal{C}$ also auch $sh(\mathcal{G}, \mathcal{B}, \mathcal{C}) \Leftrightarrow sh^+(\mathcal{G}, \mathcal{B}, \mathcal{C})$ gilt, erhalten wir durch beliebig häufige Anwendung von Blatt-Substitutionen schließlich eine Shannon-Formel $sh(\mathcal{G}, \mathcal{B}', \mathcal{C}')$, für die ebenfalls $sh(\mathcal{G}, \mathcal{B}', \mathcal{C}') \Leftrightarrow sh^+(\mathcal{G}, \mathcal{B}', \mathcal{C}')$ gilt. Wichtig ist in diesem Fall die Induktionsverankerung $\mathcal{B} \Rightarrow \mathcal{C}$. Betrachten wir die Definition von $conv_\gamma$, so ist offensichtlich, daß wir genau im Falle von unnegierten atomaren Formeln und im Falle von $\forall$-quantifizierten Formeln den Junktor sh durch sh^+ ersetzen dürfen. Insbesondere gilt damit, daß $conv_\gamma^+(S)$ eine zu $conv_\gamma(S)$ logisch äquivalente Formel konstruiert.

Korollar 3.30 (Negative Auftreten von γ-Graphen können eliminiert werden)
Für alle S in $SNF^\star$ gilt:

(1) $conv_\gamma^+(S) \Leftrightarrow conv_\gamma(S)$

(2) $DNF_{\mathbf{1}}(conv_\gamma^+(S)) \Leftrightarrow DNF_{\mathbf{1}}(conv_\gamma(S))$

BEWEIS
(1) folgt per Induktion aus Lemma 3.29. Eine andere Möglichkeit, dies zu zeigen, ist der Nachweis, daß für alle S in $SNF^\star$ gilt: $S \Leftrightarrow conv_\gamma^+(S)$. Dies ist ebenfalls einfach: Wie beim Beweis für $conv_\gamma$ zeigt man zunächst, daß die Distributivität $sh^+(\mathcal{A}, \mathcal{B}, \mathcal{C}) \circ \mathcal{D} \Leftrightarrow sh^+(\mathcal{A}, \mathcal{B} \circ \mathcal{D}, \mathcal{C} \circ \mathcal{D})$ für $\circ \in \{\wedge, \vee\}$ gilt. Mit struktureller Induktion beweist man damit leicht die Korrektheit der Kompositionsregeln $(REP_\vee)$ und $(REP_\wedge)$ für alle Formeln aus $\mathcal{SH}^+$, woraus wieder per Induktion $S \Leftrightarrow conv_\gamma^+(S)$ folgt.

(2) ist eine triviale Folge von *(1)*, denn für alle $\mathcal{S} \in \mathcal{SH}^+$ gilt $\mathcal{S} \Leftrightarrow DNF_{\mathbf{1}}(\mathcal{S})$. ■

Wir dürfen also in Schritt *1.(b)* aus Definition 3.20 $conv_\gamma$ durch $conv_\gamma^+$ ersetzen, oder was gleichbedeutend ist, weiterhin $conv_\gamma$ verwenden und negative Auftreten von γ-Graphen in **1**-Pfaden für die Beweissuche ignorieren.

Vollständigkeit des Beweisverfahrens mit selektiven Erweiterungen

Wir müssen zeigen, daß für jeden unerfüllbaren Satz S mit dem Verfahren der selektiven Erweiterungen ein γ-Graph $\mathcal{S}_n$ konstruiert werden kann, der für eine gewisse Substitution σ geschlossen ist. Dazu verwenden wir eine *faire Erweiterungsstrategie*, die gewährleistet, daß im Laufe des Verfahrens mit allen möglichen γ-Untergraphen auf allen Pfaden beliebig oft erweitert wird.

Eine solche Erweiterungsstrategie vorausgesetzt, können wir dann zeigen, daß es einen geschlossenen γ-Graph $\mathcal{S}_n$ gibt, sofern S unerfüllbar ist.

Definition 3.31 (Erweiterungsstrategie)
Eine Erweiterungsstrategie **ES** *ist eine Vorschrift, die zu einem γ-Graphen $\mathcal{S}_i$, der mittels γ-Extension erweitert werden kann, eindeutig einen Folgegraphen, die selektive Erweiterung $\mathcal{S}_{i+1} = \mathcal{S}_i\lfloor\frac{\mathbf{1}}{\mathcal{G}(\bar{X})}\rfloor^{\pi}$ von $\mathcal{S}_i$ angibt. Dazu muß durch* **ES** *ein* **1***-Knoten von $\mathcal{S}_i$ mit Pfad π und ein γ-Untergraph $\forall\bar{x}\ \mathcal{G}(\bar{x}) \in \pi$ festgelegt werden. Falls $\mathcal{S}_i$ keine γ-Untergraphen enthält, so liefert* **ES** *das Ergebnis "* keine Erweiterung möglich*" und bricht die Folge ab.*

Damit nun tatsächlich alle γ-Untergraphen eines γ-Graphen "zum Zuge kommen" benötigen wir eine Erweiterungsstrategie, die *fair* ist:

Definition 3.32 (Faire Erweiterungsstrategie)
Eine Erweiterungsstrategie **ES** *heißt* fair*, wenn zu jedem Satz S für die mit* **ES** *konstruierte Folge $conv_\gamma(S) = \mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \ldots$ gilt:*

Mit jedem γ-Untergraphen $\forall\bar{x}\ \mathcal{G}(x)$, der in einem **1***-Pfad von $\mathcal{S}_i$ auftritt, wird im weiteren Verlauf der Folge $\mathcal{S}_{i+1}, \mathcal{S}_{i+2}, \ldots$ beliebig häufig erweitert und zwar auf allen* **1***-Pfaden, auf denen $\forall\bar{x}\ \mathcal{G}(\bar{x})$ auftritt.*

In den **1**-Pfaden eines mit $conv_\gamma$ bzw. $conv_\gamma^+$ konstruierten γ-Graphen können selbst wieder γ-Untergraphen auftreten. γ-Graphen besitzen also eine "hierarchische Verschachtelungsstruktur". Im nachfolgenden Beweis benötigen wir diese an einer Stelle als Ordnung für einen Induktionsbeweis. Zu diesem Zweck definieren wir

Definition 3.33 (γ-Grad)
*Der γ-*Grad $\deg(\mathcal{S})$ *eines γ-Shannon-Graphen $\mathcal{S}$ ist wie folgt definiert:*

$$\deg(\mathcal{S}) = \begin{cases} 0 & \text{wenn } \mathcal{S} \in \mathbf{For}_{At} \cup \{\mathbf{0}, \mathbf{1}\} \\ \max(\deg(\mathcal{A}), \deg(\mathcal{B}), \deg(\mathcal{C})) & \text{wenn } \mathcal{S} = sh(\mathcal{A}, \mathcal{B}, \mathcal{C}) \\ \deg(\mathcal{G}(x)) + 1 & \text{wenn } \mathcal{S} = \forall x\ \mathcal{G}(x) \end{cases}$$

Nach dieser Definition hat jedes Element eines **1**-Pfades von $\mathcal{G}(x)$ einen kleineren γ-Grad als $\forall x\ \mathcal{G}(x)$.[18] Nun sind wir gerüstet, den Vollständigkeitssatz anzugehen:

Satz 3.34 (Vollständigkeit des Beweisverfahrens mit selektiven Erweiterungen)
Sei **ES** eine faire Erweiterungsstrategie und $\mathcal{S}_0$ der initiale Graph eines Satzes S. Dann gilt:

> Wenn S unerfüllbar ist, dann gibt es in der Folge der mit **ES** konstruierten selektiven Erweiterungen einen Graphen $\mathcal{S}_n$, der geschlossen ist.

Der Beweis verläuft im wesentlichen analog zum Beweis der Vollständigkeit des Tableau-Verfahrens mit freien Variablen in [Fitting, 1990, Seite 179ff].

Im nachfolgenden Beweis spielen **0**-Pfade keine Rolle. Daher sind, wenn wir von Pfaden reden, immer **1**-Pfade gemeint. Um von unendlichen Pfaden reden zu können, müßten

[18] O.B.d.A. gehen wir im nachfolgenden Beweis davon aus, daß in jedem γ-Untergraphen $\forall x\ \mathcal{G}(x)$ nur eine Variable x gebunden ist.

wir strenggenommen die Definition der **1**-Pfade durch $DNF_{\mathbf{1}}$ abändern. Da es offensichtlich ist, wie diese unendlichen Pfade gebildet werden, verzichten wir auf eine formale Definition.

BEWEIS (Satz 3.34)
Wir zeigen: Ist kein $\mathcal{S}_n$ der mit **ES** konstruierten Folge $\mathcal{S}_0, \mathcal{S}_1, \ldots$ geschlossen, so ist $\mathcal{S}_0$ erfüllbar und, weil $\mathcal{S}_0$ und S erfüllbarkeitsäquivalent sind, damit auch S erfüllbar.

Enthält $\mathcal{S}_0$ mindestens einen γ-Untergraphen $\mathcal{G}$, so ist die Folge $\mathcal{S}_0, \mathcal{S}_1, \ldots$ unendlich. Andernfalls liegt ein aussagenlogischer Shannon-Graph $\mathcal{S}_0$ vor. Ist dieser nicht geschlossen, so folgt mit Korollar 2.12, daß $\mathcal{S}_0$ erfüllbar ist und wir sind fertig.

Sei nun also die Folge $\mathcal{S}_0, \mathcal{S}_1, \ldots$ unendlich und kein $\mathcal{S}_i$ geschlossen. Da **ES** exakt vorschreibt, wie $\mathcal{S}_{i+1}$ aus $\mathcal{S}_i$ entsteht, kann man diese Folge als Approximation an einen unendlichen Shannon-Graphen $\widetilde{\mathcal{S}}$ auffassen.

Angenommen $\widetilde{\mathcal{S}}$ wäre für eine Substitution σ geschlossen. Dann kann jeder unendliche Pfad von $\widetilde{\mathcal{S}}$ unterhalb der jeweiligen Abschlußliterale "abgeschnitten" werden. Man erhält so einen mit σ geschlossenen Shannon-Graphen $\widetilde{\mathcal{S}}^*$, in dem jeder Pfad endlich ist. Da $\widetilde{\mathcal{S}}^*$ einen endlichen Verzweigungsgrad hat, ist $\widetilde{\mathcal{S}}^*$ nach Königs Lemma[19] selbst endlich und damit ein Teilbaum eines Shannon-Graphen $\mathcal{S}_n$ der mit **ES** konstruierten Folge. Dann ist aber auch $\mathcal{S}_n$ mit σ geschlossen, im Widerspruch zur Annahme, daß kein γ-Graph der Folge geschlossen ist. Also ist $\widetilde{\mathcal{S}}$ für keine Substitution σ geschlossen.

Im folgenden sei $\tilde{\pi}_1, \tilde{\pi}_2, \ldots$ eine Aufzählung der Pfade von $\widetilde{\mathcal{S}}$ und $t_1, t_2, \ldots$ eine Aufzählung der Terme des Herbrand-Universums $\mathcal{U}_{\mathcal{S}_0}$. Schließlich sei $X_{i,j,k}$ diejenige freie Variable, die bei der i-ten γ-Extension mit dem γ-Untergraphen $\mathcal{G}_j$ von $\mathcal{S}_0$ auf dem Pfad $\tilde{\pi}_k$ neu eingesetzt wurde. Wir betrachten nun die spezielle Substitution $\sigma : X_{i,j,k} \mapsto t_i$.

Da $\widetilde{\mathcal{S}}\sigma$ nicht geschlossen ist, gibt es einen Pfad $\tilde{\pi}_m\sigma$ von $\widetilde{\mathcal{S}}\sigma$, der nicht geschlossen ist. Wir zeigen nachfolgend, daß $\tilde{\pi}_m\sigma$ ein Modell $\mathcal{M}$ hat. Damit ist dann der Beweis erbracht, daß $\mathcal{S}_0$ erfüllbar ist; denn $\tilde{\pi}_m\sigma$ enthält als "Anfangsstück" einen **1**-Pfad $\pi_0\sigma$ von $\mathcal{S}_0\sigma$. Da π_0 keine freien Variablen enthält, gilt dann auch $\mathcal{M} \models \pi_0$ und somit $\mathcal{M} \models \mathcal{S}_0$ und die Erfüllbarkeit von $\mathcal{S}_0$ ist bewiesen.

Wir konstruieren das Modell $\mathcal{M} = \langle \mathcal{U}_{\mathcal{S}_0}, \mathbf{I}_H \rangle$ als Herbrand-Modell über dem Universum $\mathcal{U}_{\mathcal{S}_0}$ von $\mathcal{S}_0$ auf folgende Weise:

$\mathbf{I}_H$ ist eine Herbrand-Interpretation, d.h. $\mathbf{I}_H(t) = t$ für alle $t \in \mathcal{U}_{\mathcal{S}_0}$.

Weiter setzen wir für alle Prädikatensymbole P und alle Tupel von Termen $\bar{t}$ aus $\mathcal{U}_{\mathcal{S}_0}$

$$\mathbf{I}_H(P(\bar{t})) := \begin{cases} \mathsf{W} & \text{gdw. } P(\bar{t}) \in \tilde{\pi}_m\sigma \\ \mathsf{F} & \text{sonst} \end{cases}$$

Wir zeigen nun durch Induktion über den γ-Grad, daß $\mathcal{M}$ Modell aller Formeln $F \in \tilde{\pi}_m\sigma$ ist und somit von $\tilde{\pi}_m\sigma$.

$\underline{\deg(F) = 0:}$

Dann ist F ein Literal, und die Behauptung gilt nach Definition von $\mathcal{M}$.

[19] Siehe z.B. [Fitting, 1990, Seite 23ff] für einen Beweis.

deg$(F) = n + 1$:

Wir nehmen an, daß für alle Formeln $F' \in \tilde{\pi}_m\sigma$ mit deg$(F') \leq n$ die Behauptung bereits gezeigt wurde.

Nach Korollar 3.30 enthält $\tilde{\pi}_m\sigma$ neben Literalen nur positive Auftreten von γ-Untergraphen. Damit ist also $F = \forall x\ \mathcal{G}_r(x)$ einer der γ-Untergraphen von $\mathcal{S}_0$. $\tilde{\mathcal{S}}$ wurde mit der fairen Erweiterungsstrategie **ES** konstruiert, also wurde auf $\tilde{\pi}_m$ für alle $i \in I\!N$ mit $\mathcal{G}_r(X_{i,r,m})$ erweitert. Dann enthält $\tilde{\pi}_m$ aber auch für jede dieser Erweiterungen $\mathcal{G}_r(X_{i,r,m})$ einen der **1**-Pfade von $\mathcal{G}_r(X_{i,r,m})$. Sei $\pi_i^r(X_{i,r,m})$ dieser **1**-Pfad von $\mathcal{G}_r(X_{i,r,m})$.

Es ist also $(\pi_i^r(X_{i,r,m}))\sigma \in \tilde{\pi}_m\sigma$. Jedes Element von $(\pi_i^r(X_{i,r,m}))\sigma$ ist entweder ein Literal oder ein weiterer γ-Untergraph, jedenfalls eine Formel vom γ-Grad $\leq n$. Also gilt nach Induktionsvoraussetzung: $\mathcal{M} \models (\pi_i^r(X_{i,r,m}))\sigma$. Da $\pi_i^r(X_{i,r,m})$ ein **1**-Pfad von $\mathcal{G}_r(X_{i,r,m})$ ist, gilt $\mathcal{M} \models (\mathcal{G}_r(X_{i,r,m}))\sigma$ und wegen $(X_{i,r,m})\sigma = t_i$ schließlich auch $\mathcal{M} \models \mathcal{G}_r(t_i)$ für alle $i \in I\!N$. Damit haben wir gezeigt, daß für alle Terme t von $\mathcal{U}_{\mathcal{S}_0}$ gilt: $\mathcal{M} \models \mathcal{G}_r(t)$ und so auch $\mathcal{M} \models \forall x\ \mathcal{G}_r(x)$.

■

3.6 Realisierung des Verfahrens der selektiven Erweiterungen

Wir haben im vorangegangenen Abschnitt gezeigt, daß zu jedem unerfüllbaren Satz S mit Hilfe einer fairen Erweiterungsstrategie ein Shannon-Graph $\mathcal{S}_n$ gefunden werden kann, der für eine Substitution σ geschlossen ist.

Im folgenden erläutern wir, wie in einer realen Implementierung der geschlossene Shannon-Graph tatsächlich bestimmt werden kann. Die vorgestellte Beweisprozedur bildet die Grundlage des im Rahmen dieser Arbeit entstandenen Beweisers $\mathcal{SHARE}$, dessen konkrete Implementierung im sich anschließenden Kapitel 4 beschrieben wird.

Das Verfahren der maximalen Erweiterungen stellt einen Spezialfall der selektiven Erweiterungen dar, bei dem der initiale Graph der einzige γ-Graph ist. Aus diesem Grunde gelten die nachfolgenden Erläuterungen in entsprechend vereinfachter Form auch für das Verfahren der maximalen Erweiterungen.

3.6.1 Die Beweisprozedur für selektive Erweiterungen

Abbildung 3.5 zeigt die Grobstruktur der Beweisprozedur für das Verfahren der selektiven Erweiterungen (Definition 3.20) in Prolog-ähnlicher Notation: Die während der *Vorverarbeitung* notwendigen Schritte *1.(a)* und *1.(b)*, d.h. das Bilden der Negations-Normalform, die Skolemisierung $\star$ und die Konvertierung in den initialen Shannon-Graphen $\mathcal{S}_0$ lassen sich zusammenfassen und sind mittels des in Abschnitt 4.2.3 beschriebenen Prolog-Prädikats `convert` realisiert.

Das Prädikat `closed(`$\mathcal{S}$`,`π`)` hat Erfolg, wenn alle Fortsetzungen des Teilpfades π mit **1**-Pfaden aus $\mathcal{S}$ geschlossen werden können, d.h. wenn $\pi \wedge \mathcal{S}$ unerfüllbar ist. π repräsentiert eine partielle Interpretation, die alle in π auftretenden Formeln wahr macht, wodurch sich "Einschränkungen" für die möglichen Modelle von $\mathcal{S}$ ergeben.

Algorithmus 3

```
unsatisfiable(S) :-
                    convert(S, S_0),
                    closed(S_0, { }).

closed(0, π).
closed(1, π) :-
                    selected-γ-graph(π, G_sel),
                    closed(G_sel, π).
closed(sh(At, B, C), π) :-
                    (closed_path(π, ¬At) ; closed(B, π ∪ ¬At)),
                    (closed_path(π, At) ; closed(C, π ∪ At)).
closed(sh(!, B, C), π) :-
                    closed(B, π),
                    closed(C, π).
closed(sh(∀xG(x), B, C), π) :-
                    closed(B, π),
                    closed(C, π ∪ ∀xG(x)).
```

Abbildung 3.5: Die Beweisprozedur für selektive Erweiterungen

Zu Anfang wird `closed` mit dem initialen Graphen $\mathcal{S}_0$ und leerem Pfad $\{\,\}$ aufgerufen.

Für die eigentliche *Beweissuche* durchläuft `closed` den Graphen $\mathcal{S}_0$ mit Tiefensuche, wobei der Graph an **1**-Knoten durch γ-Extensionen erweitert werden kann. Wir haben folgende Fälle für `closed(`$\mathcal{S}, \pi$`)` zu unterscheiden:

1. $\mathcal{S} = \mathbf{0}$

   Pfade, die in **0** enden, repräsentieren keine Modelle der Ausgangsformel, sind also "automatisch" geschlossen.

2. $\mathcal{S} = \mathbf{1}$

   Sind wir an einem **1**-Blatt angekommen, so konnte der Pfad π bisher nicht geschlossen werden. Es gibt zwei Fälle zu unterscheiden:

   2.1 π enthält keine γ-Untergraphen.

   Dann scheitert `selected-`γ`-graph` und somit auch der Versuch, die Unerfüllbarkeit von S zu beweisen. In diesem Fall ist π ein Modell der Ausgangsformel, denn π besteht aus einer konsistenten Menge von variablenfreien Literalen. π ist variablenfrei, da der initiale Graph $\mathcal{S}_0$ ein Satz ist, also selbst keine freien Variablen enthält und diese nur durch Anwendung einer γ-Extension auf einen Pfad gebracht werden können.

   2.2 π enthält mindestens einen γ-Untergraphen.

   Mit `selected-`γ`-graph` wird ein γ-Untergraph $\mathcal{G}_{sel}$ von π ausgewählt und anschließend durch Aufruf von `closed(`$\mathcal{G}_{sel}, \pi$`)` versucht diesen zusammen mit π abzuschließen. Dies entspricht der γ-Extension des Pfades π mit $\mathcal{G}_{sel}$.

3. $\mathcal{S} = sh(At, \mathcal{B}, \mathcal{C})$

   An einem Literal-Knoten, der die atomare Formel At enthält, gehen wir folgendermaßen vor:

   3.1 Zunächst wird durch den Aufruf von `closed_path(`$\pi, \neg At$`)` versucht, den negativen Ausgang zum Untergraphen $\mathcal{B}$ abzuschließen. Es gibt wiederum zwei Fälle:

   3.1.1 Enthält π ein mit At unifizierbares Literal, so wird die durch die Unifikation gegebene Substitution σ auf den gesamten Graphen angewendet und damit der negative Ausgang zu $\mathcal{B}$ geschlossen.

   3.1.2 Andernfalls wird durch Aufruf von `closed(`$\mathcal{B}, \pi \cup \neg At$`)` versucht, den Untergraphen $\mathcal{B}$ mit dem um $\neg At$ erweiterten Pfad abzuschließen.

   3.2 Konnte der negative Ausgang von $sh(At, \mathcal{B}, \mathcal{C})$ geschlossen werden, so wird auf analoge Weise versucht, den positiven Ausgang zu $\mathcal{C}$ abzuschließen.

4. $\mathcal{S} = sh(!, \mathcal{B}, \mathcal{C})$

   Der CUT-Shannon-Graph $sh(!, \mathcal{B}, \mathcal{C})$ ist logisch äquivalent zur Disjunktion $\mathcal{B} \vee \mathcal{C}$, ist also geschlossen, wenn sowohl $\mathcal{B}$ als auch $\mathcal{C}$ geschlossen sind. π wird hierbei nicht erweitert.

5. $sh(\forall x \mathcal{G}(x), \mathcal{B}, \mathcal{C})$

Dieser Graph ist geschlossen, wenn sowohl der negative Ausgang $\mathcal{B}$ mit π als auch der positive Ausgang $\mathcal{C}$ mit dem um $\forall x \mathcal{G}(x)$ erweiterten Pfad geschlossen werden kann. Wie im theoretischen Teil gezeigt, können wir darauf verzichten, den negierten γ-Graphen $\neg\forall x \mathcal{G}(x)$ in den Pfad durch den negativen Ausgang $\mathcal{B}$ mit aufzunehmen. Gelangen wir dagegen nachfolgend an ein **1**-Blatt von $\mathcal{C}$, so ist an diesem nun eine γ-Extension mit $\mathcal{G}$ möglich.

Um die Vollständigkeit des Verfahrens sicherzustellen, müssen wir zum einen gewährleisten, daß mit jedem γ-Untergraphen auf jedem Pfad beliebig oft erweitert werden kann und zum anderen, daß mit den Substitutionen, die einzelne Pfade abschließen, der gesamte Graph geschlossen werden kann.

Recht einfach ist es, die faire Auswahl eines γ-Untergraphen mit `selected-`γ`-graph` in Schritt 2.2 zu realisieren. Hierzu verwenden wir sogenannte "γ-Zähler", in denen für jeden γ-Untergraphen festgehalten ist, wie oft mit diesem auf dem aktuellen Pfad bereits erweitert wurde. Eine faire Auswahl durch `selected-`γ`-graph` ist sichergestellt, wenn immer mit dem γ-Untergraphen erweitert wird, der den kleinsten Zählerstand hat.

Schwerwiegender ist dagegen das Problem, daß wir im allgemeinen nicht wissen, welche der in Schritt 3.1.1 möglichen Substitutionen zum Abschluß des Pfades π die für den Abschluß des Gesamtgraphen "richtige" ist. Diesem Problem sind wir im Vollständigkeitsbeweis in Abschnitt 3.5.3 aus dem Wege gegangen, indem wir solange keine Substitution angewendet haben, bis ein γ-Graph $\mathcal{S}_n$ mit der dort verwendeten Substitution $\sigma : X_{i,j,k} \mapsto t_i$ geschlossen war. In der beschriebenen Beweisprozedur müssen wir uns in Schritt 3.1.1 für eine schließende Substitution entscheiden. Im ungünstigsten Fall verhindern wir durch Anwendung der "falschen" Substitution, daß ein anderer Pfad π' geschlossen werden kann. Die Erweiterung von π' mittels γ-Extensionen führt so unweigerlich in eine "unendliche Sackgasse".

In der vorliegenden Implementierung wurde dieses Problem mittels "iterierter, beschränkter Tiefensuche" *(iterative deepening)* gelöst. D.h. wir geben eine Konstante K, das sogenannte "γ-Limit", vor und erlauben für jeden Pfad π und jeden darin vorkommenden γ-Untergraphen $\mathcal{G}$ eine höchstens $K-$malige γ-Extension von π mit $\mathcal{G}$. Wenn in Schritt 2.2 auf dem nicht geschlossenen Pfad π das γ-Limit für jeden γ-Untergraphen von π erreicht ist, so scheitert `selected-`γ`-graph`, und es tritt Backtracking auf. Als Folge hiervon werden die in Schritt 3.1.1 getroffenen Entscheidungen zurückgenommen und andere Substitutionen, die den Pfad schließen, ausgewählt.

Verwendet man iterativ immer größere γ-Limits, so wird schließlich, sofern die Ausgangsformel unerfüllbar war und die Zeit- und Speicherressourcen dies erlauben, ein geschlossener Graph gefunden.

3.6.2 Kompilierung von γ-Shannon-Graphen

Die vorgestellte Beweisprozedur kann nun realisiert werden, indem man auf einer expliziten Datenstruktur des initialen Graphen $\mathcal{S}_0$ arbeitet.

Allerdings ist es wie schon im aussagenlogischen Fall vorteilhaft, die kompakte Struktur eines Shannon-Graphen auszunutzen, um die Beweissuche direkt in eine Programmiersprache zu "kompilieren". Durch die Verwendung einer solchen Übersetzungstechnik

lassen sich deutliche Verkürzungen der Laufzeit für die Beweissuche erzielen. Der Übersetzungsvorgang eines γ-Graphen verläuft ähnlich wie im aussagenlogischen Fall:

Für jeden Nichtterminal-Knoten des initialen Graphen, d.h. für jeden Literal-, CUT- und γ-Knoten erzeugen wir in einem weiteren Vorverarbeitungsschritt, der *Kodegenerierung*, genau eine Prolog-Klausel. Die Kodegenerierung kann auf folgende Art und Weise geschehen:

Zunächst wird der initiale Graph $\mathcal{S}_0$ durchlaufen und für jeden Literal-, CUT- und γ-Knoten die entsprechende Prolog-Klausel generiert. Die in einem γ-Knoten auftauchenden γ-Untergraphen werden während dieses Durchlaufs in einer Liste aufgesammelt und anschließend selbst der Kodegenerierung zugeführt. Beginnend mit dem initialen Graphen, der den größten γ-Grad hat, bis zu den γ-Graphen mit γ-Grad 0, die selbst keine weiteren γ-Untergraphen mehr enthalten, werden damit sukzessive alle γ-Untergraphen besucht.

Die Anzahl der Klauseln des so generierten Prolog-Programms ist proportional zur Länge der Ausgangsformel $S^\star$ in (skolemisierter) Negations-Normalform:

Für jede atomare Teilformel von $S^\star$ enthält $\mathcal{S}_0$ einen Literal-Knoten, zusätzlich für jede $\forall$-quantifizierte Unterformel von $S^\star$ einen γ-Knoten und schließlich für die mittels CUT aufgelösten Disjunktionen (siehe Abschnitt 2.4) einen CUT-Knoten.

Die Funktionalität einer generierten Prolog-Klausel entspricht dem Verhalten einer der letzten drei Klauseln des `closed`-Prädikats aus Abbildung 3.5. Dies stellt eine gewisse Umkehrung des Verhaltens gegenüber den Klauseln dar, die wir für aussagenlogische Shannon-Graphen erzeugt haben: Im aussagenlogischen Fall hatte eine generierte Klausel Erfolg, wenn es eine konsistente Erweiterung des aktuellen Teilpfades zu einem **1**-Blatt gab. Nun sind wir jedoch an inkonsistenten, d.h. geschlossenen Pfaden interessiert. Also hat eine Klausel für einen Nichtterminal-Knoten Erfolg, wenn sowohl der Untergraph am negativen Ausgang, als auch der am positiven Ausgang geschlossen werden kann, wobei wir zum Abschluß die Literale des aktuellen Teilpfades benutzen dürfen.

Der Kopf einer generierten Klausel hat prinzipiell folgenden Aufbau:

`node(`*Id, Path,* γ*-Count, VarBinding*`)`

Id ist wie im aussagenlogischen Fall der Knoten-Bezeichner des zugehörigen Nichtterminal-Knotens. Jeder Teilpfad kann Literale und γ-Untergraphen enthalten. Erstere benutzen wir zum Abschluß von Pfaden, während letztere lediglich festhalten, welche γ-Extensionen auf dem aktuellen Pfad möglich sind. Aus technischer Sicht ist es daher sinnvoll, diese beiden Teilmengen eines Pfades getrennt zu repräsentieren:

Die Literale eines Pfades stellen wir durch eine Datenstruktur *Path* dar, die nun keine feste maximale Länge mehr hat, sondern durch γ-Extensionen beliebig erweitert werden kann. In γ-*Count* speichern wir für jeden γ-Untergraphen des initialen Graphen, ob dieser auf dem aktuellen Teilpfad auftritt und wenn ja, wie oft mit diesem bereits erweitert wurde.

Schließlich müssen die aktuellen Bindungen der im Shannon-Graph auftretenden freien Variablen an die Prolog-Klausel übergeben werden. Hierzu benutzen wir eine Datenstruktur *VarBinding*.

Nachdem erläutert wurde, wie sich das Verfahren der selektiven Erweiterungen allgemein implementieren läßt, wird im nächsten Kapitel die konkrete Implementierung des im Rahmen dieser Arbeit entstandenen Beweissystems $\mathcal{SHARE}$ beschrieben.

Kapitel 4

Die Implementierung des Beweisers $\mathcal{SHARE}$

Die Implementierung des prädikatenlogischen Beweisers $\mathcal{SHARE}$[1] erfolgte in Quintus-Prolog[2] und besteht aus einer Anzahl von Modulen, deren wichtigste in Abbildung 4.1 jeweils zusammen mit den zentralen exportierten Prädikaten dargestellt sind. Die Modulstruktur wurde so gewählt, daß gegenwärtige und zukünftige Varianten und Erweiterungen des Beweisverfahrens möglichst einfach zu integrieren sind. Nach einem Überblick über das System folgen Erläuterungen zu den wichtigsten verwendeten Datenstrukturen, sowie Einzelbeschreibungen der Module.

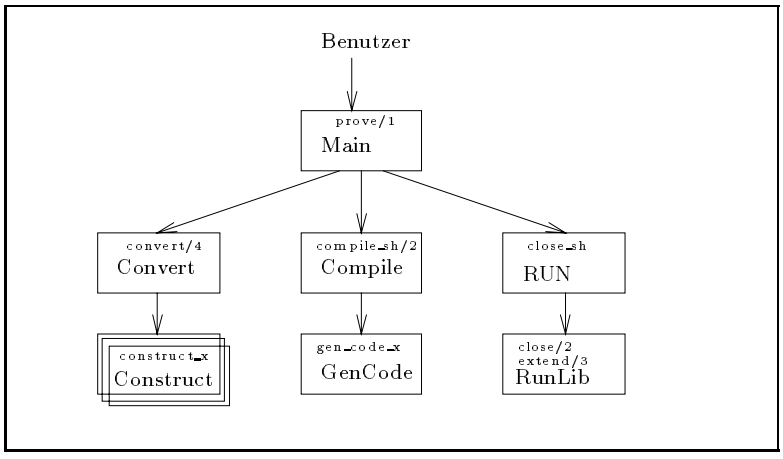

Abbildung 4.1: Aufruf-Abhängigkeiten der wichtigsten Module von $\mathcal{SHARE}$

Abbildung 4.1 zeigt die Aufruf-Abhängigkeiten für die wichtigsten Module von $\mathcal{SHARE}$. Das Hauptmodul *Main* benutzt die Module *Convert*, *Compile* und *Run*, um die einzelnen Schritte des Beweisverfahrens zu realisieren. Gleichzeitig stellt *Main* die Schnittstelle des Systems zum Benutzer in Form einer erweiterten Prolog-Shell dar. Innerhalb dieser gibt der Benutzer durch Anfragen der Art `?-<command>.` Kommandos an das System.

[1] $\mathcal{SH}$Annon-Graph $\mathcal{R}$efutation Syst$\mathcal{E}$m
[2] [Quintus Computer Systems, Inc., 1990]

Die Aufgabe von *Convert* ist es, eine gegebene Formel in einen Shannon-Graphen umzuwandeln. Dazu bedient sich *Convert* eines der *Construct*-Module, die für die verschiedenen Konstruktionsvarianten existieren. *Compile* bedient sich seinerseits *Gen_Code*, um aus dem erhaltenen Shannon-Graphen ein Prolog-Programm zu erzeugen. Dieses läuft innerhalb des Moduls *Run* ab. Während der Beweissuche benötigt *Run* zusätzlich Funktionen zum Abschluß und zur Erweiterung von Pfaden, die von *RunLib* zur Verfügung gestellt werden.

Um verschiedene Strategien zum Abschluß von Pfaden zu realisieren, müssen lediglich die von *RunLib* exportierten Prädikate neu geschrieben werden. Soll dagegen eine andere Zielsprache als Prolog für die Übersetzung von Shannon-Graphen Verwendung finden, so genügt im wesentlichen die Änderung des Moduls *Gen_Code*.

4.1 Verwendete Datenstrukturen

4.1.1 Repräsentation von Shannon-Graphen

Terminal-Knoten 0 und 1

werden als Prolog-Konstanten `false` und `true` dargestellt.

Literal-Knoten $sh(A, \mathcal{B}, \mathcal{C})$

wobei A eine atomare Formel ist, stellen wir durch einen Term der Form

- `sh(`Id, A, ShB, ShC`)`

dar. Dabei ist Id eine natürliche Zahl, die diesen Knoten in eindeutiger Weise bezeichnet, ShB und ShC sind die entsprechenden Prolog-Darstellungen für $\mathcal{B}$ und $\mathcal{C}$.

CUT**-Knoten** $sh(!, \mathcal{B}, \mathcal{C})$

werden in naheliegender Weise als

- `sh(`$Id, !, ShB, ShC$`)` repräsentiert.

γ**-Knoten** $sh(\forall \bar{x}\mathcal{G}(\bar{x}), \mathcal{B}, \mathcal{C})$

schließlich haben die Form

- `sh(ga(`$Id, Vars$`),`$GaSh, ShB, ShC$`)`.

Der Konstruktor `ga/2` zeichnet den Knoten als γ-Knoten mit dem Bezeichner Id und der Liste von $\forall$-quantifizierten Variablen $Vars$ aus. Anstelle der atomaren Formel steht nun die Repräsentation $GaSh$ des γ-Untergraphen $\mathcal{G}$.

4.1.2 Repräsentation von Pfaden

In den Pfaden eines γ-Graphen treten zum einen Literale und zum anderen γ-Untergraphen auf. Es ist zweckmäßig diese beiden Teilmengen eines Pfades getrennt zu repräsentieren.

Die Information, ob ein γ-Untergraph auf einem Pfad auftritt und, wenn dies der Fall ist, wie oft mit ihm erweitert wurde, speichern wir in einer gesonderten Datenstruktur, dem

γ-Zähler. Als *Pfad* bezeichnen wir dann im folgenden speziell die *Teilmenge der Literale* des Gesamtpfades.

Beim aussagenlogischen Beweiser konnten wir Pfade als Prolog-Term $\texttt{path}/n$ fester Stelligkeit n darstellen, wobei jeder aussagenlogischen Variablen eine bestimmte Stelle in $\texttt{path}/n$ zugeordnet war. Da im Falle der Prädikatenlogik Pfade durch die Anwendung von γ-Extensionen beliebig lang werden können, müssen wir die Darstellung nun dahingehend erweitern.

Eine sehr häufige Operation auf Pfaden ist der Versuch des Abschlusses mit einem Literal L, d.h. es wird in einem Pfad π ein zu L komplementäres Literal K gesucht, das mit L unifizierbar ist. Hier kann durch eine geeignete Darstellung von π einiger Suchaufwand von vornherein vermieden werden. Dazu stellen wir π in folgender Form dar:

$$\pi = \texttt{path}(L_1^-/L_1^+, L_2^-/L_2^+, \cdots, L_n^-/L_n^+)$$

Unter Berücksichtigung der Stelligkeit wird jedem Prädikatensymbol P eine bestimmte Position i in π zugeordnet. Diese Zuordnung wird während der Konstruktion eines Shannon-Graphen mit *conv* durch das Prädikat $\texttt{pred\_pos}/1$ geschaffen und ist für die nachfolgende Kodegenerierung verfügbar.[3] Die *negativen* Auftreten von Literalen mit dem Prädikatensymbol P speichern wir in der Liste L_i^-, die *positiven* in der Liste L_i^+.

Soll während der Beweissuche der Pfad π mit dem Literal L abgeschlossen werden, und ist P_i das zu L gehörende Prädikatensymbol, dann kommen nur noch die Literale aus L_i^- bzw. L_i^+ (in Abhängigkeit vom Vorzeichen von L) als Abschlußkandidaten in Frage. Sowohl i als auch das Vorzeichen von L sind bereits zum Zeitpunkt der Kodegenerierung verfügbar.

4.1.3 Repräsentation von Variablen

Die Anzahl k der $\forall$-quantifizierten Variablen eines Shannon-Graphen ist uns ebenfalls nach der Vorverarbeitung durch *conv* bekannt.[4] Damit können wir mit jeder dieser Variablen eine Position im sogenannten *Variablen-Vektor* assoziieren. Dieser wird in Form des k-stelligen Prolog-Terms

$$\texttt{vars}(V_1, V_2, \cdots, V_k)$$

dargestellt.

Bei "Eintritt" in einen γ-Untergraphen $\forall x \mathcal{G}(x)$ müssen wir eine neue freie Variable X an der entsprechenden Stelle im Variablen-Vektor erzeugen. Die "alte Bindung" einer früheren γ-Extension mit $\forall x \mathcal{G}(x)$ bleibt innerhalb der Literale des aktuellen Pfades $\texttt{path}(\ldots)$ erhalten. Sei x mit der Position i im Variablen-Vektor assoziiert. Dann wird eine neue freie Variable für x durch Einsetzen einer neuen und damit ungebundenen Prolog-Variablen an der i-ten Stelle erzeugt. Bindungen können nachfolgend durch Instantiierung dieser Prolog-Variablen erzeugt werden. Durch die Weitergabe des Variablen-Vektors im Kopf der generierten Prolog-Klauseln werden die Variablenbindungen der aktuellen Extensionen über verschiedene solcher Klauseln hinweg wirksam (anhand des Beispiels 4.1

[3] $\texttt{pred\_pos}/1$ speichert diese Zuordnung mit Hilfe eines dynamischen Prädikats $\texttt{pred\_pos}(Pred\_symbol, Arity, Position)$.

[4] Siehe in Abschnitt 4.2.3 die Behandlung einer $\forall$-quantifizierten Unterformel.

auf Seite 76 wird eine genauere Erklärung für die Behandlung von Variablenbindungen gegeben).

Wenn sich im Laufe der Beweissuche Instantiierungen als "falsch" herausstellen, werden diese durch das Prolog-Backtracking automatisch rückgängig gemacht.

4.1.4 γ-Zähler

Schließlich ist auch die Anzahl der γ-Untergraphen eines Shannon-Graphen nach abgeschlossener Vorverarbeitung eine bekannte Größe. Zur Laufzeit werden für den gerade untersuchten Pfad eine Reihe von γ-Zählern in Form eines γ-*Vektors* mitgeführt, in dem die Anzahl der Aufrufe von γ-Untergraphen festgehalten wird. Diese Information ist für die *faire Auswahl* bei einer γ-Extension an einem **1**-Knoten notwendig. Die Darstellung des γ-Vektors bei m vorhandenen γ-Untergraphen ist

$$\texttt{ga}(G_1, G_2, \cdots, G_m)$$

$G_i - 1$ gibt an, wie oft mit dem γ-Untergraphen $\mathcal{G}_i$ auf dem aktuellen Pfad bereits erweitert wurde. Ist $G_i = 0$, so taucht $\mathcal{G}_i$ auf dem aktuellen Pfad überhaupt nicht auf, eine Extension mit $\mathcal{G}_i$ ist dann nicht erlaubt.

4.2 Beschreibung der wichtigsten Module

4.2.1 Main

Dieses Modul stellt die Schnittstelle von $\mathcal{SHARE}$ zum Benutzer dar. Die wichtigsten Prädikate sind:

- `prove(`+*KB_File*`)`
- `prove_inc(`+*KB_File*`)`
- `prove_file(`+*KB_Collection_File*`)`

KB_File ist die sogenannte *Wissensbasis*, die eine Menge von Axiomen $\{Ax_1, \ldots, Ax_n\}$ und eventuell ein Theorem Th enthält. Durch die Anfrage `?-prove(`*KB_File*`)` wird versucht, die Gültigkeit von

$$\{Ax_1, \ldots, Ax_n\} \models Th$$

durch Nachweis der Inkonsistenz von

$$Ax_1 \wedge \ldots \wedge Ax_n \wedge \neg Th$$

zu beweisen. Die Syntax der Wissensbasis entspricht im wesentlichen der des Tableaux-Beweisers ${}_3T^AP$ und ist z.B. in [Hähnle *et al.*, 1992] beschrieben. Da die bestehende Implementierung von $\mathcal{SHARE}$ keine *Sorten* verwendet, werden die entsprechenden Deklarationen ignoriert. Zusätzlich kann eine Wissensbasis Anweisungen der Form

`:-` *KB_Goals*`.`

enthalten. Die Prolog-Ziele *KB_Goals* werden während des Ladens der Wissenbasis ausgeführt.

Durch die Anfrage `?-prove_inc(`+*KB_File*`)` wird zunächst das γ-Limit auf 1 gesetzt und dann versucht, das durch die Wissenbasis +*KB_File* gegebene Problem zu beweisen. Nach jedem gescheiterten Versuch wird das γ-Limit um 1 erhöht und anschließend erneut ein Beweisversuch unternommen.

Häufig möchte man eine ganze Reihe von Problemen nacheinander beweisen, etwa um die Auswirkungen verschiedener Schalterstellungen, Heuristiken usw. zu untersuchen. Dazu erzeugt man eine Datei *KB_Collection_File*, die durch Aufruf des Kommandos `?-prove_file(`*KB_Collection_File*`)` abgearbeitet wird. Dabei können die Ergebnisse der Beweisläufe automatisch in einer Tabelle zusammengestellt werden.

Folgende Anweisungen sind zulässig:

`:-` *KB_Collection_Goals*`.`
: Führt die Prolog-Ziele *KB_Collection_Goals* aus. Insbesondere können so Schalter für die nachfolgenden Beweise gesetzt werden.

`prove(`*KB_File*`).`
: Löst einen Aufruf des gleichnamigen Prädikats und so den Beweis der Wissensbasis *KB_File* aus.

4.2.2 Input

Stellt dem Hauptmodul *Main* die Funktion

- `load_axioms(`+*KB_File*, -*Formula*`)`

zur Verfügung. *Formula* ist die zu widerlegende Formel, die aus der Wissensbasis *KB_File* durch Konjunktion der Axiome und, falls vorhanden, des negierten Theorems gebildet wird.

4.2.3 Convert

Dieses Modul implementiert, zusammen mit einem der *Construct*-Module, die Funktion $conv$: **For**$\rightarrow \mathcal{SH}_\gamma$. Es werden folgende Prädikate exportiert :

- `convert(`+*Formel*`,`-*Sh*`,`-*Stat*`,`-*SkolFormel*`)`
- `conv(`+*Formel*`,`-*Sh*`,`-*F*`,`-*T*`,` *Stat*`,` *SkolFormel*`)`

Die Hauptarbeit erledigt `conv/6`: Die gegebene *Formel* wird in einen Shannon-Graphen *Sh* konvertiert, wobei anstelle der **0** und **1**-Blätter die Prolog-Variablen *F* und *T* zurückgeliefert werden, um nachfolgende Einsetzungen zu ermöglichen. Wie schon im aussagenlogischen Fall beschrieben[5], bilden *Sh/F/T* eine Differenzstruktur. *Stat* liefert Information

[5] Siehe Abschnitt 2.3.2

über die Größe des zu *Sh* gehörenden Binärbaumes $\widehat{Sh}$. Diese kann von Heuristiken verwendet werden, um den potentiellen Suchraum (lokal) zu minimieren. Schließlich liefert *SkolFormel* das Ergebnis der Skolemisierung der Ausgangsformel. Diese wird dem Benutzer nur zur Information zurückgegeben und spielt für die Durchführung des Beweises keine unmittelbare Rolle. Für den Benutzer ist diese Formel jedoch interessant, da sie logisch äquivalent zu *Sh* ist.

`convert/4` bildet lediglich die Schnittstelle zum Hauptmodul *Main* und ruft nach einigen Initialisierungen `conv/6` auf. Nachdem die Konstruktion des Shannon-Graphen abgeschlossen ist, können für die Blätter *F* und *T* Konstanten `false` und `true` eingesetzt werden:

```
convert(Formula,Sh,Stat,SkolFormula) :-
     convert_init,
     conv(Formula,Sh,false,true,Stat,SkolFormula).
```

`conv/6` arbeitet folgendermaßen: Wie im theoretischen Teil beschrieben, müssen wir zunächst eine Negations-Normalform berechnen. Dies wird auf naheliegende Weise erreicht, indem wir Negationen "nach innen ziehen":

```
conv(-(A & B),ShAB,F,T,Stat,Skol) :-
        conv(-A v -B,ShAB,F,T,Stat,Skol).

conv(-(A v B),ShAB,F,T,Stat,Skol) :-
        conv(-A & -B,ShAB,F,T,Stat,Skol).

conv(- ( A ! B ) ,ShAB,F,T,Stat,Skol) :-
        conv(-A & -B,ShAB,F,T,Stat,Skol).

conv(-(A => B),ShAB,F,T,Stat,Skol) :-
        conv(A & -B,ShAB,F,T,Stat,Skol).

conv(-(A <=> B),ShAB,F,T,Stat,Skol) :-
        conv(A <=> -B,ShAB,F,T,Stat,Skol).

conv(-(forall(Vars) in For),Sh,F,T,Stat,Skol) :-
        conv(exists(Vars) in -For,Sh,F,T,Stat,Skol).

conv(-(exists(Vars) in For),Sh,F,T,Stat,Skol) :-
        conv(forall(Vars) in -For,Sh,F,T,Stat,Skol).

conv(- (- A),Sh,F,T,Stat,Skol) :-
        conv(A,Sh,F,T,Stat,Skol).
```

Im Falle einer Konjunktion bzw. einer Disjunktion werden die entsprechenden Prädikate des Moduls *Construct* aufgerufen:

```
conv(A & B,ShAB,F,T,StatAB,SkolA & SkolB) :-
        conv(A,ShA,FA,TA,StatA,SkolA),
        conv(B,ShB,FB,TB,StatB,SkolB),
        construct_conj(ShA,FA,TA,StatA,
                ShB,FB,TB,StatB,
                ShAB,F,T,StatAB).

conv(A v B,ShAB,F,T,StatAB,SkolA v SkolB) :-
        conv(A,ShA,FA,TA,StatA,SkolA),
        conv(B,ShB,FB,TB,StatB,SkolB),
        construct_disj(ShA,FA,TA,StatA,
                ShB,FB,TB,StatB,
                ShAB,F,T,StatAB).

conv(A => B,ShAB,F,T,Stat,Skol) :-
         conv(- A v B,ShAB,F,T,Stat,Skol).
```

Für eine Formel mit CUT-Operator (`A ! B`) wird ein CUT-Knoten mit dem neuen Bezeichner `Id` und dem CUT-Symbol "`!`" (anstelle der Bedingung) erzeugt. Ein solcher CUT-Knoten wird während der Kodegenerierung gesondert behandelt. Die Größe des resultierenden Shannon-Graphen in Baumdarstellung, d.h. die Anzahl der **0** und **1**-Knoten ergibt sich als Summe der Größen der Teilbäume für `A` und `B`.

```
conv((A ! B),sh(Id,!,ShA,ShB),F,T,s(AB0s,AB1s),(SkolA ! SkolB)):-
        ctr_inc(1,1,Id),        % new CUT-node
        conv(A,ShA,F,T,s(A0s,A1s),SkolA),
        conv(B,ShB,F,T,s(B0s,B1s),SkolB),
                AB0s is A0s + B0s,
                AB1s is A1s + B1s.
```

Die Konstruktion für die Äquivalenz wird, da hier verschiedene Varianten möglich sind, im Modul *Construct* vorgenommen.

```
conv(A <=> B,ShAB,F,T,Stat,Skol) :-
        construct_eqv(A,B,ShAB,F,T,Stat,Skol).
```

Eine $\forall$-quantifizierte Formel `forall(Vars) in For` wird in einen Term der Form `sh(ga(GId,Vs),Sh,F,T)` umgewandelt. Dabei repräsentiert `Sh` den eigentlichen γ-Untergraphen von `For`, in dem die Blätter **0** und **1** bereits mit `false` und `true` instantiiert wurden. Die "Einbettung" von `Sh` in den gesamten Graphen ergibt sich durch die Verweise auf den negativen bzw. positiven Untergraphen, die nachfolgend für `F` bzw. `T` eingesetzt werden können.

```
conv(forall(Vars) in For,sh(ga(GId,Vs),Sh,F,T),F,T,Stat,Skol) :-
        insert_variables(Vars,For,For1,Vs0),
        conv(For1,Sh1,false,true,Stat,Skol),
        (Sh1 = sh(ga(GId,Vs1),GaSh,false,true)
           ->    append(Vs0,Vs1,Vs),
                 Sh = GaSh
           ;     ctr_inc(7,1,GId),
                 Vs = Vs0,
                 Sh = Sh1
        ).
```

Zunächst werden alle Auftreten von Variablen aus `Vars` in `For` durch spezielle Prolog-Terme der Form `'$VAR'(I)` ersetzt, dies erledigt `insert_variables/4`. Die so erhaltene Formel `For1` wird mittels `conv/6` in den γ-Untergraphen `Sh1` konvertiert. War `For` selbst eine $\forall$-quantifizierte Formel, so ist `Sh1` ebenfalls von der Form `sh(ga(GId,Vs1),GaSh,false,true)`, und die beiden Untergraphen können zusammengefaßt werden. Diese Zusammenfassung entspricht der syntaktischen Vereinfachung von $\forall x \forall y(F)$ zu $\forall x, y(F)$.

Für die Kodegenerierung ist wichtig, welche Variablen in einem γ-Untergraphen quantifiziert sind; `Vs` enthält diese Variablen in Form der zuvor eingesetzten Terme `'$VAR'(I)`.

Bei einer $\exists$-quantifizierten Formel entledigen wir uns des Existenzquantors mit Hilfe der in Abschnitt 3.3 beschriebenen Skolemisierung $\star$, d.h. wir ersetzen alle $\exists$-quantifizierten Variablen `Vars` in `For` durch Skolemfunktionen und rufen `conv/6` für die so erhaltene Formel `For1` auf. Dabei ist zu beachten, daß `For` eventuell Terme `'$VAR'(I)` enthält, die die $\forall$-quantifizierten Variablen innerhalb von `For` repräsentieren; diese sind gerade die Argumente der Skolemfunktionen.

```
conv(exists(Vars) in For,Sh,F,T,Stat,Skol) :-
        skolemize(Vars,For,For1),
        conv(For1,Sh,F,T,Stat,Skol).
```

Im Fall einer atomaren Formel A konstruieren wir direkt den entsprechenden Term `sh(Id,A,F,T)`. Dies ist die einzige Stelle, an der wir die "Löcher" `F` und `T` für nachfolgende Einsetzungen explizit zurückgeben. `s(N0s,N1s)` bezeichnet die Anzahl der **0** bzw. der **1**-Knoten, wenn wir den Shannon-Graphen als Binärbaum deuten. Da `N0s` und `N1s` sehr groß werden können,[6] müssen wir diese als Fließkommazahlen darstellen. `pred_pos/1` ordnet dem Prädikatensymbol von `A` in eindeutiger Weise eine Position in der Datenstruktur für Pfade zu. Diese Zuordnung wird während der Kodegenerierung[7] benötigt.

```
conv(- A,sh(Id,A,T,F),F,T,s(1.0,1.0),- A) :-
        atomic_formula(A),
        pred_pos(A),
        ctr_inc(1,1,Id).
```

[6] Siehe 2.6.3

[7] Siehe 4.2.6, Seite 75

```
conv(A, sh(Id,A,F,T),F,T,s(1.0,1.0),A) :-
        atomic_formula(A),
        pred_pos(A),
        ctr_inc(1,1,Id).
```

4.2.4 Construct

Wie schon im aussagenlogischen Fall wurden auch für die Prädikatenlogik verschiedene Varianten der Funktion *conv* implementiert.

Um die Implementierung übersichtlich zu halten, wurden die verschiedenen Konstruktions-Varianten aus dem Modul *Convert* ausgelagert, und es entstand eine Menge von sogenannten *Construct*-Modulen. Der Benutzer von $\mathcal{SHARE}$ kann zur Laufzeit eines dieser Module auswählen, um so die verschiedenen Varianten vergleichen zu können. Derzeit gibt es folgende *Construct*-Module:

- *consSh* : Erzeugt Shannon-Graphen ohne CUT. Die Äquivalenz wird gemäß

  $$conv(A \leftrightarrow B) := conv((A \rightarrow B) \wedge (B \rightarrow A))$$

  aufgelöst.

- *consShEq* : Die Äquivalenz wird durch Einführung eines CUT-Knotens aufgelöst:

  $$conv(A \leftrightarrow B) := sh(!, conv(A \wedge B), conv(\neg A \wedge \neg B)).$$

  Dies ist in der Regel günstiger als die erste Variante, da die sich gegenseitig ausschließenden Fälle $A \wedge B$ und $\neg A \wedge \neg B$ getrennt behandelt werden.

- *consShOpt* : Wie *consShEq*, außer daß die Heuristik $conv_{min1st}$ aus Abschnitt 2.5 zum Einsatz kommt.

- *consTab* : Erzeugt CUT-Shannon-Graphen, die als Tableaux aufgefaßt werden können und entspricht $conv_{Tab}$ aus Kapitel 2.

- *consTabAlphaOpt*: Wie *consTab*, jedoch mit der $conv_{\alpha\_min}$ entsprechenden Heuristik.

Die Schnittstelle nach außen bilden folgende Prädikate, die von jedem der *Construct*-Modul exportiert werden:

- `construct_conj(`*+ShA, +FA,+TA,+StatA,*
  +ShB, +FB,+TB,+StatB,
  -ShAB,-F, -T, -Stat `)`

  konstruiert die Konjunktion *ShAB* der Shannon-Graphen *ShA* und *ShB*. *FA* und *TA* stehen für die "Löcher" in *ShA*, entsprechendes gilt für *FB,TB* und *F,T*.

- `construct_disj(`*+ShA, +FA,+TA,+StatA,*
  +ShB, +FB,+TB,+StatB,
  -ShAB,-F, -T, -Stat `)`

  Konstruiert in analoger Weise die Disjunktion von *ShA* und *ShB*.

- `construct_eqv(`+*FormelA*`,`+*FormelB*`,`
  -*ShAB*`,`-*F*`,`-*T*`,`-*Stat*`,`-*Skol* `)`

  Konstruiert den Shannon-Graphen *ShAB* zur Formel $FormelA \leftrightarrow FormelB$. Die Parameter unterscheiden sich von den beiden obigen Fällen, da bei der Äquivalenz zusätzliche Freiheitsgrade bei der Konstruktion bestehen. So könnte man die (meist ungünstige) Umformung nach $(FormelA \rightarrow FormelB) \wedge (FormelB \rightarrow FormelA)$ durchführen oder aber einen CUT-Knoten einführen und die Fälle $FormelA \wedge FormelB$ sowie $\neg FormelA \wedge \neg FormelB$ getrennt behandeln.

4.2.5 Compile

Aufgabe dieses Moduls ist die Übersetzung eines Shannon-Graphen in ein Programm, das die Beweissuche im Shannon-Graphen durchführt. Die eigentliche Kodegenerierung wurde in das Modul *Gen_Code* ausgelagert. Dadurch ist das Modul *Compile* unabhängig von der gewählten Zielsprache. Die Schnittstelle nach außen bildet das Prädikat

- `compile_sh(`+*Sh*`,`+*RunFile*`)`

das aus dem Shannon-Graphen *Sh* eine Datei *RunFile* erzeugt, die das entsprechende Programm zur Beweissuche enthält. Während des Übersetzungsvorgangs wird für jeden Nichtterminal-Knoten von *Sh* genau eine Prolog-Klausel generiert. Da *Sh* im allgemeinen selbst γ-Untergraphen enthält, ist die Übersetzung mehrstufig:

Wenn *Sh* einen γ-Knoten der Form `sh(ga(`$GId, Vars$`),`ShG, ShB, ShC`)` enthält, so wird in einem ersten Durchlauf Kode erzeugt, der lediglich festhält, wo in *Sh* der γ-Untergraph *ShG* eingesetzt zu denken ist, d.h. es werden die Kanten zu *ShB* und *ShC* im Programmkode festgehalten. Es wird jedoch noch kein Kode für *ShG* erzeugt; stattdessen wird *ShG* zusammen mit allen anderen Untergraphen in *Sh* in einer Liste aufgesammelt und in einem nachfolgenden Durchlauf übersetzt. Jeder dieser so aufgesammelten Untergraphen kann seinerseits wieder Untergraphen enthalten, die analog bearbeitet werden. Die Verschachtelungstiefe entspricht genau derjenigen der $\forall$-quantifizierten Formeln in der Negations-Normalform der Ausgangsformel. Diese wiederholte Übersetzung leistet das Prädikat `compile_iterate/2`, das die Hauptarbeit an `comp/2` weiterdelegiert.

```
compile_iterate([]).
compile_iterate([GaSh|More]):-
        comp(GaSh,InnerGammas/[]),
        compile_iterate(More),
        compile_iterate(InnerGammas).
```

`comp(`+*Sh*`,`-*InnerGammas*`)` übersetzt *Sh* und sammelt dabei die γ-Untergraphen in der Liste *InnerGammas* auf.[8] In Abhängigkeit der Struktur von *Sh* sind folgende Fälle zu unterscheiden

[8] *InnerGammas* wird aus Effizienzgründen als Differenz-Liste dargestellt.

1. *Sh* ist ein "gewöhnlicher" sh-Term, d.h. Darstellung eines γ-Shannon-Graphen $\mathcal{S}$. Es gibt wiederum mehrere Fälle

   (a) Ist $\mathcal{S}$ ein Terminal-Knoten, so ist nichts weiter zu tun; denn **0**-Knoten erzeugen grundsätzlich keinen Kode, während der Kode für einen **1**-Knoten bereits vom entsprechenden Vorgänger-Knoten generiert wird.

   ```
   comp(false,Gs/Gs).
   comp(true,Gs/Gs).
   ```

   (b) Ist die Wurzel von $\mathcal{S}$ ein Knoten für den bereits Kode generiert wurde, so ist ebenfalls nichts weiter zu tun. Dies abzufragen sichert, daß nicht der gesamte Binärbaum traversiert, sondern für jeden Nichtterminal-Knoten im Graphen $\mathcal{S}$ genau einmal Kode erzeugt wird.

   ```
   comp(sh(Id,_,_,_),Gs/Gs):-
           already_compiled(Id),
           !.
   ```

   (c) Ist die Wurzel von $\mathcal{S}$ ein γ-Knoten `sh(ga(`*GId,Vs*`),`*GaSh,Sh0,Sh1*`)`, so wird für diesen Kode generiert, und es werden rekursiv *Sh0* und *Sh1* übersetzt. Wie bereits erwähnt wird für *GaSh* zunächst noch kein Kode erzeugt. Statt dessen wird *GaSh* in die Liste der noch zu bearbeitenden γ-Untergraphen aufgenommen.

   ```
   comp(sh(ga(Gid,Vs),GaSh,S0,S1),
        [sub_graph(Gid,GaSh,Vs)|More]/End) :-
           gen_code_gamma(Gid,[],S0,S1),
           comp(S0,More/More2),
           comp(S1,More2/End).
   ```

   (d) CUT-Knoten und Literal-Knoten erzeugen über die entsprechenden Aufrufe des Moduls *Gen_Code* den gewünschten Programmkode. Die negativen und positiven Nachfolger *S0,S1* werden durch rekursive Aufrufe von `comp/2` abgearbeitet.

   ```
   comp(sh(Id,!,S0,S1),More/End) :-
           gen_code_cut(Id,[],S0,S1),
           comp(S0,More/More2),
           comp(S1,More2/End).
   comp(sh(Id,Atom,S0,S1),More/End) :-
           gen_code_atomic(Id,Atom,[],S0,S1),
           comp(S0,More/More2),
           comp(S1,More2/End).
   ```

2. *Sh* ist ein γ-Untergraph, der in einem vorangegangenen Durchlauf aufgesammelt wurde, d.h. ein Term der Form `sub_graph(`*Gid,GaSh,Vars*`)`. Die Kodegenerierung unterscheidet sich in einem kleinen, aber wichtigen Detail von den obigen Fällen: Im Programmkode für den *Wurzel-Knoten* von *GaSh* müssen die in diesem

γ-Untergraphen $\forall$-quantifizierten Variablen als neue freie Variablen ausgezeichnet werden. Dies geschieht durch Übergabe der Liste *Vars* an die entsprechenden Prädikate `gen_code_gamma/4`, `gen_code_cut/4` und `gen_code_atomic/5` des Moduls *Gen_Code* (siehe dazu Abschnitt 4.2.6). In den obigen Fällen wurde anstelle von *Vars* dagegen die leere Liste `[]` übergeben, da an einem Nicht-Wurzel-Knoten keine neuen Variablen erzeugt werden dürfen. Die Zuordnung zwischen dem γ-Untergraphen Gid und dessen Wurzel-Knoten $NodeId$ wird durch Aufruf von `gen_code_gamma_entry(Gid,NodeId)` im generierten Programm festgehalten.

```
comp(sub_graph(Gid,sh(ga(SubGid,SubVars),GaSh,S0,S1),Vars),
     [sub_graph(SubGid,GaSh,SubVars)|More]/End) :-
        gen_code_gamma(SubGid,Vars,S0,S1),
        gen_code_gamma_entry(Gid,ga(SubGid)),
        comp(S0,More/More2),
        comp(S1,More2/End).

comp(sub_graph(Gid,sh(Id,!,S0,S1),Vars),More/End) :-
        gen_code_cut(Id,Vars,S0,S1),
        gen_code_gamma_entry(Gid,Id),
        comp(S0,More/More2),
        comp(S1,More2/End).

comp(sub_graph(Gid,sh(Id,Atom,S0,S1),Vars),More/End) :-
        gen_code_atomic(Id,Atom,Vars,S0,S1),
        gen_code_gamma_entry(Gid,Id),
        comp(S0,More/More2),
        comp(S1,More2/End).
```

4.2.6 Gen_Code

Dieses Modul wird von *Compile* aufgerufen und muß für die folgenden Typen von Knoten Prolog-Kode erzeugen können:

- Literal-Knoten $sh(A, \mathcal{B}, \mathcal{C})$
- CUT-Knoten $sh(!, \mathcal{B}, \mathcal{C})$
- γ-Knoten $sh(\forall \bar{x} \mathcal{G}(\bar{x}), \mathcal{B}, \mathcal{C})$

Zu diesem Zweck werden die Prädikate

- `gen_code_atomic/5`
- `gen_code_cut/4`
- `gen_code_gamma/4`
- `gen_code_gamma_entry/2`

- `gen_code_top_clause/1`

exportiert.

Für jeden Nichtterminal-Knoten des zu übersetzenden Shannon-Graphen wird genau eine Prolog-Klausel mit dem Kopf

$$\texttt{node}(Id, Path, VarBinding, GaCount)$$

erzeugt. Id ist die Kennzeichnung des Knotens, $Path$ die Teilmenge der Literale des aktuellen Pfads. Variablenbindungen sind durch den Variablen-Vektor $VarBinding$ repräsentiert. Die Häufigkeit der Anwendung von γ-Extensionen ist im γ-Vektor $GaCount$ festgehalten.

`node`$(Id, Path, VarBinding, GaCount)$ hat Erfolg gdw. der Untergraph mit Wurzel Id, zusammen mit den Einschränkungen, die durch den Pfad $Path$, die Variablenbindungen $VarBinding$ und den γ-Vektor $GaCount$ gegeben sind, geschlossen werden kann. Anhang B.2 enthält ein Beispiel für den Prolog-Kode, der von den nachfolgend erläuterten Prädikaten generiert wird.

- `gen_code_atomic`$(+Id, +Atom, +NewVars, +\mathcal{S}_0, +\mathcal{S}_1)$

  erzeugt Prolog-Kode für einen Literal-Knoten mit dem Bezeichner Id, der atomaren Formel $Atom$ und den Nachfolger-Knoten $\mathcal{S}_0, \mathcal{S}_1$. $NewVars$ ist die Liste der Variablen, die als *neue freie Variablen* zu behandeln sind. Ist der vorliegende Literal-Knoten Wurzel eines γ-Untergraphen, so besteht $NewVars$ gerade aus den in diesem Untergraphen gebundenen Variablen (siehe Abschnitt 4.2.5), anderfalls ist $NewVars$ die leere Liste `[]`.

  Wir nehmen an, daß dem Prädikatensymbol der atomaren Formel $Atom$ die Stelle i im Pfad zugeordnet ist,[9] die Anzahl der $\forall$-quantifizierten Variablen des betrachteten Shannon-Graphen sei m. Weiterhin seien Id_0 und Id_1 die Knoten-Bezeichner der Nachfolger-Knoten $\mathcal{S}_0$ und $\mathcal{S}_1$. Dann sieht die von `gen_code_atomic`$/5$ generierte Prolog-Klausel wie folgt aus:

  `node(`Id`, path(`$P_1, \ldots, L_i^-/L_i^+, \ldots, P_k$`), vars(`$V_1, \ldots, V_m$`), Ga) :-`
  `(close(-`$Id : Atom, L_i^+$`)`
  `; add_to_path(-`$Id : Atom, L_i^-, L_i^{neu-}$`),`
  `node(`Id_0`, path(`$P_1, \ldots, L_i^{neu-}/L_i^+, \ldots, P_k$`), vars(`$V_1, \ldots, V_m$`), Ga)`
  `),`
  `(close(+`$Id : Atom, L_i^-$`)`
  `; add_to_path(+`$Id : Atom, L_i^+, L_i^{neu+}$`),`
  `node(`Id_1`, path(`$P_1, \ldots, L_i^-/L_i^{neu+}, \ldots, P_k$`), vars(`$V_1, \ldots, V_m$`), Ga)`
  `).`

  Zur Laufzeit enthält der Variablen-Vektor `vars(`$V_1, \ldots, V_m$`)` die aktuellen Variablenbindungen. Mit jeder Position i im Variablen-Vektor ist eine bestimmte Variable eines γ-Untergraphen assoziiert. V_i repräsentiert die Bindung der aktuellen ("neuesten") Instanz dieser Variable.

[9] Siehe Abschnitt 4.1.2

Ist der vorliegende Literal-Knoten Wurzel eines γ-Untergraphen $\forall\bar{x}\mathcal{G}(\bar{x})$, so entspricht der Aufruf von `node(`$Id,\ldots$`)` einer γ-Extension mit $\mathcal{G}(\bar{X})$, bei dem die neuen freien Variablen $\bar{X}$ erzeugt werden müssen. $NewVars$ ist dann die Liste dieser Variablen. Für jede der Variablen in $NewVars$ wird an der entsprechenden Stelle im Variablen-Vektor `vars(`$V_1,\ldots,V_m$`)` eine neue und damit ungebundene Prolog-Variable eingesetzt. Die "alte Bindung" bleibt in Form der Literale in `path(`$\ldots$`)` erhalten.

Beispiel 4.1
Wir betrachten den γ-Untergraphen $\mathcal{G}_1$ in Abbildung B.1 auf Seite 96. Der generierte Prolog-Kode für die Wurzel `sh(1,s(A),`$\ldots$`,`$\ldots$`)` dieses γ-Untergraphen ist:

```
node(1,path(Neg/Pos,Pr2,Pr3,Pr4),vars(_,B,C,D),Ga):-
    (close(-1:s(A),Pos)
       ; add_to_path(-1:s(A),Neg,Neg1),
         extend(path(Neg1/Pos,Pr2,Pr3,Pr4),vars(A,B,C,D),Ga)
    ),
    (close(1:s(A),Neg)
       ; add_to_path(1:s(A),Pos,Pos1),
         node(2,path(Neg/Pos1,Pr2,Pr3,Pr4),vars(A,B,C,D),Ga)
    ).
```

Die von diesem γ-Untergraphen gebundene Variable ist mit der ersten Stelle im Variablen-Vektor assoziiert. Eine neue Instanz dieser Variablen wird in Form der zunächst ungebundenen Prolog-Variablen `A` erzeugt. Die "alte Bindung" aus einer früheren γ-Extension wird an dieser Stelle nicht mehr benötigt. Deshalb dürfen wir im Variablen-Vektor im Kopf der Klausel an der ersten Position einen Unterstrich "_" einsetzen. Die alte Bindung geht jedoch nicht verloren, sondern bleibt innerhalb der Literale der Datenstruktur `path(`$\ldots$`)` vorhanden.

Zurück zum allgemeinen Fall: Durch Aufruf von `close(-`$Id:Atom,L_i^+$`)` wird versucht, den aktuellen Pfad abzuschließen, indem das negierte $Atom$ mit einem der positiven Auftreten des gleichen Prädikats der Liste L_i^+ unifiziert wird. Dabei können Variablen aus $Atom$ und aus L_i^+ instantiiert werden. Diese Bindungen werden mit Hilfe des Variablen-Vektors an die Nachfolger-Knoten weitergegeben. Das Prädikat `close` muß vom Modul *RunLib* zur Verfügung gestellt werden.

Gelingt der Abschluß jedoch nicht, oder wird mittels Backtracking eine zuvor gefundene Abschlußmöglichkeit verworfen, so wird der aktuelle Pfad durch Aufruf des Prädikats `add_to_path(-`$Id:Atom,L_i^-,L_i^{neu-}$`)` (das vom Modul *RunLib* bereit gestellt wird) erweitert und der negative Nachfolger $\mathcal{S}_0$ mit dem neuen Pfad aufgerufen. Man beachte, daß der neue Pfad sich vom ursprünglichen nur an der Stelle i unterscheidet, alle anderen Literal-Listen werden unverändert weitergereicht. Der Aufruf `node(`$Id_0,\ldots$`)` des Nachfolgers ist genau dann erfolgreich, wenn $\mathcal{S}_0$ zusammen mit dem neuen Pfad geschlossen werden kann.

Wenn es so möglich war, den Untergraphen am negativen Ausgang zu schließen, wird analog für den positiven Nachfolger verfahren. Der γ-Vektor *Ga* wird unverändert an die Nachfolger-Knoten weitergegeben.

- `gen_code_cut`$(+ Id, + NewVars, + \mathcal{S}_0, + \mathcal{S}_1)$

  erzeugt Prolog-Kode für den CUT-Knoten mit dem Bezeichner Id und den Nachfolger-Knoten $\mathcal{S}_0, \mathcal{S}_1$. Bei CUT-Knoten wird weder der Pfad verändert, weshalb auch keine neuen Abschlußmöglichkeiten mit `close` betrachtet werden müssen, noch wird der γ-Vektor verändert. Die generierte Klausel ist daher sehr einfach:

  ```
  node(Id, Path, V', Ga) :-
      node(Id_0, Path, V, Ga),
      node(Id_1, Path, V, Ga).
  ```

  Die Klausel hat Erfolg genau dann, wenn beide Nachfolger $\mathcal{S}_0$ und $\mathcal{S}_1$ geschlossen werden können. Die Variablen-Vektoren V' und V sind verschieden, wenn die Liste der freien Variablen $NewVars$ nicht leer ist, das heißt, wenn der vorliegende CUT-Knoten Wurzel eines γ-Untergraphen ist.

- `gen_code_gamma`$(+ Id, + NewVars, + \mathcal{S}_0, + \mathcal{S}_1)$

  erzeugt Prolog-Kode für den γ-Knoten mit dem Bezeichner Id und den Nachfolger-Knoten $\mathcal{S}_0$ und $\mathcal{S}_1$. Auch hier unterscheiden sich die Variablen-Vektoren V' und V nur, wenn der vorliegende Knoten die Wurzel eines γ-Untergraphen ist. Die Aufgabe dieses Knotens ist, im γ-Vektor des *positiven* Nachfolgers $\mathcal{S}_1$ den γ-Zähler Ga_{Id} um Eins hochzuzählen, um so nachfolgende γ-Extension mit dem entsprechenden γ-Untergraphen zu ermöglichen.

  ```
  node(gaId, Path, V',ga(Ga_1, ..., Ga_Id, ..., Ga_n) :-
      Ga_Id^neu is Ga_Id+1,
      node(Id_0, Path, V,ga(Ga_1, ..., Ga_Id, ..., Ga_n),
      node(Id_1, Path, V,ga(Ga_1, ..., Ga_Id^neu, ..., Ga_n).
  ```

- `gen_code_gamma_entry`$(+ GaId, + NodeId)$

  Hat man bei einem Extensionschritt einen γ-Untergraphen mit dem Bezeichner $GaId$ selektiert, so muß die Klausel des zugehörigen Wurzel-Knotens mit dem Bezeichner $NodeId$ aufgerufen werden. Bei der Kompilierung des Wurzel-Knotens speichern wir diese Zuordnung zwischen dem Bezeichner eines γ-Untergraphen und seiner Wurzel in einer globalen Datenstruktur.[10] Zu diesem Zweck erzeugt `gen_code_gamma_entry`$(+GaId, +NodeId)$ Fakten `gamma_entry`$(GaId, NodeId)$, die dem generierten Programm hinzugefügt werden.

- `gen_code_top_clause`$(+ Id)$

  erzeugt eine Klausel, die zur Ausführungszeit zunächst einige Initialisierungen vornimmt, um dann die Wurzel Id des initialen Shannon-Graphen mit leerem Pfad `path`$([]/[], \ldots, []/[])$, ungebundenen Variablen und auf 0 gesetzten γ-Zählern aufzurufen:

[10] Alternativ kann man diese Information auch in Form von weiteren Parametern den Klauseln `node`$(\ldots)$ hinzufügen.

```
closed_graph :-
    Initialisierungen,
    node(Id, path([]/[], ..., []/[]), vars(_, ..., _), ga(0, ..., 0)).
```

Bisher sind wir implizit davon ausgegangen, daß alle Nachfolger-Knoten $\mathcal{S}_0$ und $\mathcal{S}_1$ des zu übersetzenden Knotens Nichtterminal-Knoten sind. Ist einer der Nachfolger ein **0**-Knoten, die per Definition bereits geschlossen sind, so entfällt der Aufruf mit `node(...)`. Der Kode für die generierte Prolog-Klausel vereinfacht sich entsprechend. Liegt jedoch ein **1**-Knoten vor, so wird der Aufruf `node(...)` ersetzt durch den Aufruf des Prädikats `extend(`$Path, V, Ga$`)` aus dem Modul *RunLib,* wobei der aktuelle Pfad $Path$, der Variablen-Vektor V und der γ-Vektor Ga übergeben werden.

4.2.7 RunLib

Dieses Modul enthält die Prädikate, die das generierte Prolog-Programm zur Laufzeit benötigt. Die wichtigsten sind `extend/3` zur selektiven Erweiterung eines Pfades an einem **1**-Knoten und `close/2` zum Abschluß von Pfaden:

- `extend(`$+Path, +Bindings, +GaCount$`)`

  wird aufgerufen, wenn wir bei der Traversierung des Shannon-Graphen an einem **1**-Knoten angekommen sind. Der γ-Vektor $GaCount$ enthält für jeden γ-Untergraphen $\mathcal{G}_i$ die um Eins erhöhte Anzahl G_i der Anwendungen der entsprechenden γ-Extension. Erlaubt sind nur Extensionen für solche $\mathcal{G}_j$, für die $G_j \geq 1$ gilt. $G_j = 0$ bedeutet, daß $\mathcal{G}_j$ auf dem aktuellen Pfad nicht vorkommt. Um eine *faire Auswahl* der γ-Untergraphen sicherzustellen, wird der Untergraph mit dem kleinsten γ-Zähler selektiert und der zugehörige Wurzel-Knoten aufgerufen.

- `close(`$+Id : Atom, AtomList$`)`

  hat Erfolg, wenn $Atom$ mit einem der Atome aus $AtomList$ unifiziert werden kann. Die Auswahl des "Abschlußpartners" von $Atom$ muß entweder *fair* sein, so daß eine einmal getroffene Entscheidung nicht mehr zurückgenommen werden muß, oder `close` muß via Backtracking alle möglichen Lösungen aufzählen können. Die aktuelle Version des Moduls *RunLib* verwendet Backtracking, um die Vollständigkeit des Verfahrens zu gewährleisten.

  Die Reihenfolge, in der man die möglichen Lösungen von `close` aufzählt, hat einen erheblichen Einfluß auf die Beweislänge. Id ist der Bezeichner des Knotens, aus dem die atomare Formel $Atom$ stammt. Die Atome in $AtomList$ sind ebenfalls mit entsprechenden Knoten-Bezeichnern versehen. Diese Information kann von verschiedenen Auswahlstrategien verwendet werden, um die Häufigkeit bestimmter Abschlußpaare zu zählen. Man kann dann diejenigen Abschlußpaare vorziehen, die am seltensten verwendet wurden. Diese Information alleine ist allerdings noch nicht ausreichend, um eine faire Auswahl zu garantieren, bei der auf Backtracking verzichtet werden kann.

4.3 Testergebnisse des prädikatenlogischen Beweisers $\mathcal{SHARE}$

In den Tabellen 4.1 und 4.2 sind Statistiken für die mit $\mathcal{SHARE}$ bewiesenen prädikatenlogischen Probleme *pel18-pel46* aus [Pelletier, 1986] angegeben. Eine auf dem Verfahren der maximalen Erweiterungen basierende frühe Version von $\mathcal{SHARE}$ konnte einige dieser Probleme noch nicht beweisen.

Tabelle 4.1 zeigt die Ergebnisse für die Konstruktionsvariante "*consShEq*".[11] Beim Aufbau des initialen Graphen wird in dieser Variante die Äquivalenz durch Einführung eines CUT-Knotens aufgelöst:

$$conv_\gamma(A \leftrightarrow B) := sh(!, conv_\gamma(A \wedge B), conv_\gamma(\neg A \wedge \neg B))$$

Dadurch werden die sich gegenseitig ausschließenden Modelle von $A \wedge B$ und $\neg A \wedge \neg B$ getrennt betrachtet. In der älteren Version "*consSh*" wird die Äquivalenz gemäß

$$conv_\gamma(A \leftrightarrow B) := conv_\gamma((A \rightarrow B) \wedge (B \rightarrow A))$$

aufgelöst. In den so konstruierten Shannon-Graphen treten redundante Pfade auf, die der Suche nach Modellen von $A \wedge B \wedge \neg B$ bzw. $\neg A \wedge B \wedge A$ entsprechen. Daher benötigt die Version mit "*consSh*" für Formeln, in denen eine Äquivalenz auftritt, mehr Abschlüsse als die abgebildete Version "*consShEq*".

In Tabelle 4.2 sind für dieselben Testbeispiele die Ergebnisse unter Verwendung der Konstruktionsvariante "*consTab*" aufgeführt. Die mit dieser Variante konstruierten Shannon-Graphen können als Tableaux aufgefaßt werden (siehe Abschnitt 2.4.1).

Die Zeit für die Vorverarbeitung eines Problems ist im wesentlichen unabhängig von der gewählten Variante[12] und verteilt sich im Durchschnitt wie folgt auf die einzelnen Schritte:

convert benötigt nur 1–2% der Vorverarbeitungszeit, auf die Kodegenerierung entfallen knapp 19%, während der Hauptanteil von etwa 80% von Quintus-Prolog für die Kompilierung des generierten Prolog-Programmes benötigt wird. Auch hier gilt wie schon im Falle der Aussagenlogik, daß dieser vergleichsweise hohe Aufwand durch Verwendung einer spezialisierten Zielsprache verringert werden könnte. Allerdings fällt bei Problemen, die schwieriger als die hier vorgestellten sind, die Vorverarbeitungszeit gegenüber der Laufzeit für die eigentliche Beweissuche weniger ins Gewicht.

Durch die verwendete Übersetzungstechnik sind die Laufzeiten für die Beweissuche bei beiden Varianten sehr gering; in den meisten Fällen lagen sie unterhalb der Meßgenauigkeit von Quintus-Prolog. Alle Testbeispiele, mit Ausnahme von *pel38* und *pel43*, wurden mit einem γ-Limit von 2 durchgeführt, d.h. mit jedem γ-Untergraphen konnte auf einem Pfad höchstens zweimal erweitert werden. Bei *pel38* und *pel43* wurde das γ-Limit auf 1 gesetzt. Gerade für diese beiden Probleme verhalten sich die beiden Varianten sehr unterschiedlich:

Für die 1. Variante tritt beim Beweis von *pel38* wesentlich häufiger Backtracking auf bis die "richtige" Substitution gefunden ist als für die 2. Variante. Dies wirkt sich entsprechend auf die Anzahl der geschlossenen Pfade GP und somit schließlich auch auf die

[11] Siehe auch Abschnitt 4.2.4.

[12] Die mit *consTab* gebildeten initialen Graphen sind im allgemeinen geringfügig größer als die mit *consShEq* konstruierten, da sie zusätzliche CUT-Knoten enthalten.

Beweiszeit aus: Bedingt durch den unterschiedlichen Aufbau der initialen Graphen werden von der verwendeten Abschlußstrategie die möglichen schließenden Substitutionen in unterschiedlicher Reihenfolge aufgezählt, womit sich das stark differierende Verhalten erklären läßt. Hier besteht demnach noch viel Spielraum für Verbesserungen der Strategie.

In GP sind auch diejenigen Pfadabschlüsse mitgezählt, die durch Backtracking wieder aufgehoben werden. Würde immer gleich die "richtige" schließende Substitution angewendet, müßten demnach lediglich $(GP - BT)$ Pfade geschlossen werden, um einen Beweis zu finden, d.h. 52 für "*consShEq*" und 46 für "*consTab*". Beim Problem *pel43* schneidet dagegen die Variante "*consShEq*" wesentlich besser ab, da ein Beweis direkt, also ohne Backtracking, gefunden wird.

Problem	$\lvert\mathcal{S}_0\rvert$	γ	GP	BK	BT	Conv	Gen	Comp	BS
pel18	2	1	1	3	0	†0	67	283	0
pel19	4	1	7	25	3	16	84	433	0
pel20	7	3	3	6	0	17	150	700	0
pel21	9	1	7	15	0	0	150	567	0
pel22	10	2	4	10	0	0	150	567	0
pel23	9	2	4	8	0	0	133	583	0
pel24	12	4	14	20	0	0	267	1066	0
pel25	13	4	7	14	0	17	283	1083	0
pel26	21	5	23	44	0	17	400	1367	0
pel27	13	4	36	59	19	17	283	1084	17
pel28	12	4	10	35	0	33	267	1000	17
pel29	19	3	32	69	8	16	333	1267	17
pel30	8	2	5	11	0	17	167	700	0
pel31	9	3	4	9	0	16	200	767	17
pel32	12	3	7	13	0	16	300	1000	0
pel33	21	2	27	42	3	16	283	1217	0
pel34	85	24	77	259	0	50	1416	7284	150
pel35	2	1	1	2	0	17	83	283	0
pel36	8	5	7	14	0	17	200	883	0
pel37	10	5	5	13	0	50	233	1034	16
pel38	37	7	474	1115	422	83	617	2933	500
pel39	5	1	2	4	0	17	100	350	0
pel40	10	2	4	10	0	0	150	633	0
pel41	8	2	3	7	0	0	133	583	0
pel42	7	2	5	10	0	17	133	517	0
pel43	14	3	25	36	0	33	283	1200	0
pel44	10	3	7	16	0	17	200	833	0
pel45	18	6	13	27	0	50	433	1717	0
pel46	18	4	17	26	0	17	400	1500	17

$\lvert\mathcal{S}_0\rvert$ = Anzahl der Nichtterminal-Knoten des initialen Graphen $\mathcal{S}_0$
γ = Anzahl der γ-Untergraphen von $\mathcal{S}_0$
GP = Anzahl der geschlossenen Pfade
BK = Anzahl der besuchten Knoten
BT = Häufigkeit des Auftretens von Backtracking
Conv = Laufzeit in *ms* für *convert*
Gen = Laufzeit in *ms* für die Kodegenerierung
Comp = Laufzeit in *ms* für die Kompilierung des generierten Programms
BS = Laufzeit in *ms* für die Beweissuche
† Zeit lag unterhalb der Meßgenauigkeit von Quintus-Prolog (17 ms)

Tabelle 4.1: Testergebnisse der Konstruktionsvariante *consShEq*

Problem	$\|\mathcal{S}_0\|$	γ	GP	BK	BT	Conv	Gen	Comp	BS
pel18	2	1	1	3	0	†0	67	283	0
pel19	5	1	6	22	2	0	83	383	0
pel20	9	3	3	6	0	16	150	700	0
pel21	11	1	8	19	0	17	134	533	0
pel22	10	2	4	10	0	0	134	583	0
pel23	11	2	4	8	0	17	150	567	0
pel24	18	4	21	37	0	16	250	1050	0
pel25	18	4	7	14	0	17	300	1033	0
pel26	25	5	26	58	0	17	350	1366	0
pel27	19	4	36	59	19	33	300	1050	17
pel28	16	4	7	24	0	16	250	967	0
pel29	25	3	26	55	7	16	300	1184	16
pel30	10	2	3	7	0	0	133	600	0
pel31	12	3	4	9	0	0	200	700	0
pel32	17	3	6	12	0	17	267	983	0
pel33	29	2	12	23	0	0	283	1084	16
pel34	85	24	77	259	0	50	1383	7300	150
pel35	2	1	1	2	0	17	67	300	0
pel36	10	5	3	8	0	17	200	800	17
pel37	14	5	7	18	0	50	250	1034	16
pel38	53	7	138	225	92	84	616	2817	100
pel39	5	1	2	4	0	17	100	367	0
pel40	10	2	4	10	0	17	133	634	0
pel41	9	2	3	7	0	33	134	583	0
pel42	8	2	5	10	0	17	100	517	0
pel43	20	3	118	222	100	50	233	1050	117
pel44	13	3	7	16	0	17	200	784	0
pel45	25	6	11	24	0	33	417	1583	0
pel46	27	4	16	26	0	17	383	1400	0

$|\mathcal{S}_0|$ = Anzahl der Nichtterminal-Knoten des initialen Graphen $\mathcal{S}_0$
γ = Anzahl der γ-Untergraphen von $\mathcal{S}_0$
GP = Anzahl der geschlossenen Pfade
BK = Anzahl der besuchten Knoten
BT = Häufigkeit des Auftretens von Backtracking
Conv = Laufzeit in *ms* für *convert*
Gen = Laufzeit in *ms* für die Kodegenerierung
Comp = Laufzeit in *ms* für die Kompilierung des generierten Programms
BS = Laufzeit in *ms* für die Beweissuche
† Zeit lag unterhalb der Meßgenauigkeit von Quintus-Prolog (17 ms)

Tabelle 4.2: Testergebnisse der Konstruktionsvariante *consTab*

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Theorie und Implementierung von einem auf Shannon-Graphen basierenden Beweisverfahren für die Aussagenlogik und die Prädikatenlogik erster Stufe behandelt.

Die Darstellung von logischen Formeln in Form von Shannon-Graphen hat sich dabei als besonders geeignet für das automatische Beweisen erwiesen: Aussagenlogische Shannon-Graphen repräsentieren Modelle und Gegenmodelle[1] einer Formel in kompakter Form. Es konnte ein enger Zusammenhang zwischen aussagenlogischen Shannon-Graphen und Tableaux mit Lemmata aufgezeigt werden. In diesem Sinne ist eine Art Lemma-Generierung, die häufig die Beweisfindung erleichtert, in Shannon-Graphen auf natürliche Weise "automatisch eingebaut". Durch die Einführung von sogenannten CUT-Knoten kann, falls dies erwünscht ist, diese automatische Lemma-Generierung teilweise oder ganz aufgehoben werden.

Das aussagenlogische Beweisverfahren wurde zunächst in Form des Verfahrens der *maximalen Erweiterungen* auf die Prädikatenlogik erster Stufe erweitert. Schwächen dieses Ansatzes konnten durch das verbesserte Verfahren der *selektiven Erweiterungen* behoben werden. Die Korrektheit und Vollständigkeit dieser Verfahren wurde bewiesen.

Ein besonderer Vorteil, der für die Verwendung von Shannon-Graphen im automatischen Beweisen spricht, liegt darin, daß sich die darauf basierenden Verfahren effizient implementieren lassen: Shannon-Graphen können in eine Programmiersprache "kompiliert" werden, d.h. anstatt die Beweissuche auf einer expliziten Datenstruktur eines Shannon-Graphen $\mathcal{S}$ vorzunehmen, kann man ein Programm $P_{\mathcal{S}}$ generieren, das die Beweissuche speziell für $\mathcal{S}$ vornimmt. Die Verwendung dieser Übersetzungstechnik resultiert in einer beträchtlichen Beschleunigung der Beweissuche.

Im Rahmen dieser Arbeit entstand das auf der beschriebenen Übersetzungstechnik basierende, in Quintus-Prolog implementierte Beweissystem $\mathcal{SHARE}$. Durch dessen modularen Aufbau kann $\mathcal{SHARE}$ leicht an verschiedene Modifikationen des Beweisverfahrens angepaßt werden. Hier sind unter anderem die Verwendung von verschiedenen Strategien zum Abschluß von Pfaden und die Wahl von anderen Zielsprachen zu nennen. Insbesondere durch Verwendung von verbesserten Strategien zum Abschluß von Pfaden kann die Leistungsfähigkeit des bestehenden Systems weiter erhöht werden. Die

[1] D.h. Modelle der negierten Formel.

Bestrebungen sollten dabei in Richtung einer fairen Strategie gehen, die möglichst ohne Backtracking auskommt.

Ein weiterer erfolgversprechender Ansatz, die bestehende Implementierung zu verbessern, liegt in der Verwendung sogenannter *universeller Formeln*, die aus dem Tableaukalkül mit freien Variablen bekannt sind [Beckert & Hähnle, 1992]. Anhand eines einfachen Beispiels kann man bereits erkennen, welche Verbesserungen damit möglich sind: Derzeit werden zum Beweis des Problems *pel19* (Abschnitt 4.3) 7 Abschlüsse von Pfaden, von denen durch Backtracking 3 wieder verworfen werden, benötigt. Nutzt man in diesem Beispiel dagegen das Vorhandensein von universellen Formeln und der zugehörigen universellen Variablen aus, so tritt zum einen kein Backtracking mehr auf, und es genügt zum anderen weniger, nämlich 2 statt 4 Pfade abzuschließen.

Schließlich erscheint es für die Zukunft vielversprechend, die prädikatenlogische Deduktion mit Shannon-Graphen noch näher in Richtung der bei BDDs verwendeten Techniken weiterzuentwickeln. Bei BDDs wird durch Vorgabe einer Ordnungsrelation auf den Atomen eine weitgehend redundanzfreie Darstellung von logischen Funktionen erreicht. Zwar ist der Aufwand für das Erstellen einer solchen reduzierten Form wesentlich größer als der Aufwand zur Konstruktion eines Shannon-Graphen, dafür ist das *aussagenlogische* Beweisverfahren mit der Erstellung dieser Normalform aber bereits beendet: Eine unerfüllbare aussagenlogische Formel wird in reduzierter Form immer als **0** dargestellt. Wie dieses Prinzip im Rahmen eines prädikatenlogischen Beweisverfahrens eingesetzt werden kann, müssen weitere Untersuchungen zeigen.

Anhang A

Alternative Zielsprachen für die Aussagenlogik

A.1 C-Kode

Abbildung A.1 zeigt den generierten C-Kode für das Beispiel 2.15 aus Kapitel 2. Das Prolog-Programm für dieses Beispiel ist in Abbildung 2.4 dargestellt. Der Klausel `node`$(i,\ldots)$ entspricht die C-Funktion `f`i. Die globalen Variablen `a,b,c` repräsentieren die jeweils aktuellen Wahrheitswerte der gleichnamigen aussagenlogischen Variablen, wobei die Wahrheitswerte `F` und `W` als `0` bzw. `1` dargestellt werden. Ein Wert von `2` bedeutet, daß das zugehörige Atom nicht auf dem aktuellen Pfad auftritt.

Da die Funktionen `f`i weder lokale Variablen noch Parameter besitzen, muß beim Funktionsaufruf nur die Rücksprungadresse auf dem Laufzeitkeller abgelegt werden. Ein "guter Compiler" sollte daher in der Lage sein, effizienten Kode für die Funktionsaufrufe zu erzeugen. Die Ergebnisse aus Abschnitt 2.6.3 zeigen, daß der vom getesteten C-Compiler generierte Kode kaum besser ist als der kompilierte Prolog-Kode.[1]

A.2 Assembler-Kode

Ebenfalls für das Beispiel 2.15 zeigt Abbildung A.2 den generierten Assembler-Kode. Die mit dem Label `f`i beginnenden Unterprogramme haben dieselbe Funktionalität und Arbeitsweise, wie die gleichnamigen C-Funktionen. Für die Abarbeitung dieser Unterprogramme, d.h. für das "Besuchen" des zughörigen Nichtterminal-Knotens im Shannon-Graphen werden nur noch wenige Maschinenzyklen benötigt. Auf einem Rechner mit *40 MHz i486-CPU* lassen sich so $\approx$ 3.000.000 Knoten pro Sekunde besuchen.

Hervorgehoben werden muß, daß es recht einfach ist, die "Übersetzungsschablonen" des Moduls *Gen_Code* für andere (Assembler)-Sprachen anzupassen.

[1] Experimente mit einem anderen C-Compiler (*Turbo-C 2.0* auf einem PC) lieferten bessere Resultate. Insbesondere konnte der Compiler durch eine Option veranlaßt werden, keine Standard-Prozedurschachteln (*standard stack frames*) für die Funktionsaufrufe zu verwenden; diese werden nur für Funktionen mit lokalen Variablen bzw. Parametern benötigt. Allein dadurch wurde die Laufzeit um 1/3 verkürzt.

```
int a,b,c;

char i2c( i )                              /* integer -> char */
int i;
{ if (i == 2) return '*';
  else if (i == 0) return '-';
  else return '+';
}
pm(){                                       /* *print_model*  */
      printf("%ca ",i2c(a));
      printf("%cb ",i2c(b));
      printf("%cc ",i2c(c));
      printf("\n");
}

f1(){
      if (a == 0) {f3(); return;}      /* linker Nachfolger */
      else if (a == 1) {f2(); return;} /* rechter Nachfolger*/
      else{ a=0;f3();                  /* beide Nachfolger  */
            a=1;f2();
            a=2;
      }
}
f2() {
      if (b == 0) {f3(); return;}
      else  if (b == 1) {pm(); return;}
      else{ b=1;pm();
            b=0;f3();
            b=2;
      }
}
f3() {
      if (c == 0) return;
      else  if (c == 1) {pm(); return;}
      else{ c=1;pm();
            c=2;
      }
}

main() {
      a = b = c = 2;
      f1(); exit(0);
}
```

Abbildung A.1: Generierter C-Kode für $(a \wedge b) \vee c$

```
_a:     dw 2              ; 2 => Variable noch nicht belegt
_b:     dw 2              ; ...1 => Variable mit 'W' belegt
_c:     dw 2              ; ...0 => Variable mit 'F' belegt
;-------------------------------------------------------------
f1:     cmp         _a,1
        jg          b1     ; _a > 1 ?, dann beide Nachfolger
        je          e1     ; _a = 1 ?, dann rechter Nachfolger
        jmp         f3     ; sonst linker Nachfolger
e1:     jmp         f2
b1:     mov         _a,0
        call        f3
        mov         _a,1
        call        f2
        mov         _a,2
        ret
;---------------------
f2:     cmp         _b,1
        jl          w2     ; _b < 1 ?, dann linker Nachfolger
        je          e2     ; _b = 1 ?, dann *print_model*
        mov         _b,1   ; sonst _b := TRUE...
        call        pm     ; ... *print_model*
        mov         _b,0   ; ... _b := FALSE
        call        f3     ; ... rechter Nachfolger
        mov         _b,2   ; ...  _b := UNDEF
        ret
e2:     jmp          pm
w2:     jmp          f3
;---------------------
f3:     cmp         _c,1
        jl          e3      ; _c < 1 ?, dann Pfad geschlossen
        je          w3      ; _c = 1 ?, dann *print_model*
        mov         _c,1    ; sonst _c := TRUE...
        call        pm      ; ... *print_model*
        mov         _c,2    ; ...  _c := UNDEF
e3:     ret
w3:     jmp        pm
;-------------------------------------------------------------

_main:  call        f1
        push        0
        call        _exit
        inc         sp
        inc         sp
        ret
```

Abbildung A.2: Generierter 8086-Kode für $(a \wedge b) \vee c$

Anhang B

Die Benutzung von $\mathcal{SHARE}$

B.1 Kurzbeschreibung der Kommandos

Der folgende Abschnitt enthält eine knappe Anleitung zur Benutzung des automatischen Beweisers $\mathcal{SHARE}$. Die Implementierung von $\mathcal{SHARE}$ erfolgte unter Quintus-Prolog 3.0. Funktionen zur automatischen Darstellung des initialen Graphen und von Ergebnistabellen werden angeboten, setzen aber zusätzlich die Benutzeroberfläche *X-Windows* und die entsprechenden Programme TreeTEX, LATEXund `xdvi` voraus.

Nach Starten von Quintus-Prolog wird das System durch die Anweisung

```
| ?- [share].
```

geladen. Ist der Ladevorgang abgeschlossen, so erscheinen einige Informationen über die aktuellen Einstellungen des Beweisers. Der Benutzer kann nun von einer erweiterten Prolog-Shell aus mit $\mathcal{SHARE}$ arbeiten. Es stehen die folgenden Kommandos zu Verfügung (Abkürzungen für die Kommandos sind in eckigen Klammern angegeben):

On-Line Hilfestellung

- `helpme [hm]`

  Durch Aufruf dieses Kommandos wird eine kurze Hilfestellung zu den Kommandos und gegenbenenfalls Information zur aktuellen Version von $\mathcal{SHARE}$ gegeben.

Kommandos zum Starten von Beweisen

- `prove(`*KB_File*`) [pr]`

  Lädt die Wissenbasis *KB_File* und versucht das darin beschriebene Problem zu beweisen.

- `proveinc(`*KB_File*`) [pi]`

  Dieses Prädikat muß vom Modul `runlib` exportiert werden und implementiert die “iterierte, beschränkte Tiefensuche” *(iterative deepening)*. Eine mögliche Realisierung benutzt Backtracking über das γ-Limit:

  Zunächst wird das γ-Limit auf 1 gesetzt und dann mit prove(*KB_File*) ein Beweisversuch unternommen. Nach jedem gescheiterten Versuch wird das γ-Limit um 1 erhöht und anschließend der Beweis erneut versucht.

- `provefile(`*KB_Collection_File*`)`

  Die Datei *KB_Collection_File* kann Anweisungen der Form

  `prove(`*KB_File*$_1$`).`
  $\vdots$
  `prove(`*KB_File*$_n$`).`

  enthalten, die durch das Kommando `provefile` nacheinander abgearbeitet werden. Wenn der entsprechende Schalter gesetzt ist (`textab(on)`), wird automatisch eine Tabelle mit statistischen Informationen über die einzelnen Beweisläufe in Form eines LaTeX-Dokumentes erstellt und mit Hilfe des Programms `xdvi` auf dem Bildschirm angezeigt.

- `redo [rd]`

  Wiederholt das letzte Kommando der Form `prove(`*KB_File*`)`.

Kommandos zum Abfragen/Setzen von Schaltern

- `switches [sw]`

  Durch Aufruf dieses Kommandos werden alle aktuellen Schalterstellungen angezeigt.

Werden die nachfolgenden Kommandos mit einer Prolog-Variablen als Argument aufgerufen, so wird die entsprechende Schalterstellung zurückgeliefert. Andernfalls wird der Schalter gemäß dem übergebenen Argument gesetzt.

- `convert(`*Cons_Variant*`) [cv]`

  Mit diesem Kommando wird die Konstruktionsvariante zum Aufbau des initialen Graphen festgelegt. Derzeit sind für *Cons_Variant* folgende Werte möglich:

  - `consSh`

    Es wird ein initialer γ-Graph ohne Verwendung von CUT-Knoten konstruiert. Die Äquivalenz wird gemäß

    $$conv(A \leftrightarrow B) := conv((A \rightarrow B) \wedge (B \rightarrow A))$$

    aufgelöst.

  - `consShEq`

    Die Äquivalenz wird durch Einführung eines CUT-Knotens aufgelöst :

    $$conv(A \leftrightarrow B) := sh(!, conv(A \wedge B), conv(\neg A \wedge \neg B)).$$

    Dies ist in der Regel günstiger als die erste Variante und ist deshalb die Voreinstellung.

  - `consShOpt`

    Wie `consShEq`, außer daß die Heuristik $conv_{min1st}$ aus Abschnitt 2.5 zum Einsatz kommt.

  - `consTab`

    Erzeugt CUT-Shannon-Graphen, die als Tableaux aufgefaßt werden können und entspricht $conv_{Tab}$ aus Kapitel 2.

  - `consTabAlphaOpt`

    Wie *consTab*, jedoch mit der $conv_{\alpha\_min}$ entsprechenden Heuristik.

- `runlib(`*RunLib_File*`)` `[rl]`

  Legt fest welches Modul (mit dem Dateinamen *RunLib_File*) die Funktionen zur Erweiterung und zum Abschluß von Pfaden zur Verfügung stellt.

- `gammalimit(`*N*`)` `[gl]`

  Setzt das γ-Limit auf *N*, d.h. mit jedem γ-Untergraphen kann auf einem Pfad höchstens *N*-mal erweitert werden.[1]

- `info(`*Switch*`)` `[in]`

  Ist *Switch* = `on`, so werden statistische Daten über den Beweislauf angezeigt. Diese umfassen u.a. die Anzahl der geschlossenen Pfade, die Häufigkeit des Auftretens von Backtracking und die Dauer des Beweises.

- `protocol(`*Switch*`)` `[pc]`

  Wenn *Switch* = `on` ist, wird ein einfaches Protokoll des Beweislaufs ausgegeben.

- `tree(`*Switch*`)` `[tr]`

  Ist dieser Schalter auf `on`, so wird für nachfolgende Beweisläufe jeweils ein TreeTEX-Dokument erstellt, in dem der initiale Graph und dessen γ-Untergraphen in Form von Binärbäumen dargestellt sind. Die Darstellung auf dem Bildschirm erfolgt automatisch mit Hilfe des Programms `xdvi`.

- `prooftree(`*Switch*`)` `[pt]`

  Wie `tree` nur daß der instantiierte Beweisbaum dargestellt wird.

- `treestyle(`*Style*`)` `[ts]`

  Es gibt die folgenden Schalterstellungen für *Style*, die festlegen, welche Form der mit TreeTEX dargestellte Baum haben soll:

  - `treeTab`

    Erzeugt "Tableau-artige" Bäume, d.h. es werden keine **0**-Blätter dargestellt und Nichtterminal-Knoten, die einen **0**-Nachfolger haben, als unäre Knoten behandelt. Der mit $\mathcal{T}$ bezeichnete Shannon-Graph auf Seite 25 wurde mit dieser Option gezeichnet.

  - `tree01`

    Erzeugt die bekannte Darstellung von Shannon-Graphen in Form von Binärbäumen, d.h. alle Nichtterminal-Knoten haben zwei Ausgänge und es werden sowohl **0** als auch **1**-Knoten dargestellt.

  - `tree1`

    Wie `tree01` nur daß **0**-Knoten nicht dargestellt werden.

[1] Für weitere Schalter im Zusammenhang mit `proveinc` siehe Modul `runlib` bzw. `helpme`.

- `textab(`*Switch*`)  [tt]`

  Nur wenn *Switch* = `on` ist wird für die mit `prove_file` abgearbeiteten Beweisläufe eine Tabelle erzeugt. Diese liegt dann in Form einer LaTeX-Datei vor.

B.2 Ein Beispiellauf für Pelletiers 24. Problem

Wir zeigen im folgenden am Beispiel des 24. Problems aus [Pelletier, 1986], wie ein Beweislauf mit dem Beweiser $\mathcal{SHARE}$ aussehen kann.[2]

Nach Eingabe von

```
| ?- [share].
```

innerhalb der Shell von Quintus-Prolog wird $\mathcal{SHARE}$ geladen und meldet sich anschließend, wie folgt:

```
*** SHARE loaded.
*** Type 'helpme.' to get help on available commands.
*** The current settings are:

convert ..... consShEq
runlib ...... gammarunlib
gammalimit... 2
info ........ on
protocol .... off
tree ........ on
tree_style .. treeTab
tex_tab...... off
```

Die *Wissenbasis* des zu beweisenden Problems steht in einer Datei, die folgendes Aussehen hat:

```
axiom pel24_1; -(exists [x]:top in (s(x) & q(x))).
axiom pel24_2; (forall [x]:top in (p(x) => (q(x) v r(x)))).
axiom pel24_3; (-(exists [x]:top in (p(x))) =>
                  (exists [y]:top in (q(y)))).
axiom pel24_4; (forall [x]:top in ((q(x) v r(x)) => s(x))).

theorem pel24; (exists [x]:top in (p(x) & r(x))).
```

Es ist nun zu zeigen, daß aus den Axiomen $pel24_1, \ldots, pel24_4$ das Theorem $pel24$ folgt, d.h., daß

$$\{pel24_1, \ldots, pel24_4\} \models pel24$$

gilt. Wir nehmen an, daß die Wissenbasis in der Datei `pel24` abgespeichert ist. Dann wird durch Aufruf von

[2] Die Ausgaben stammen von der ersten Version von $\mathcal{SHARE}$ und können in nachfolgenden Versionen davon abweichen.

```
| ?- prove('pel24')
```

der Beweislauf gestartet. Intern wird dabei zunächst eine Formel konstruiert, die aus einer Konjunktion der Axiome und des negierten Theorems besteht:

$$
\begin{array}{lll}
 & \neg(\exists[x](s(x) \wedge q(x))) & \textit{Axiom Pel24}_1 \\
\wedge & (\forall[x](p(x) \rightarrow (q(x) \vee r(x)))) & \textit{Axiom Pel24}_2 \\
\wedge & ((\neg(\exists[x]p(x))) \rightarrow (\exists[y]q(y))) & \textit{Axiom Pel24}_3 \\
\wedge & (\forall[x]((q(x) \vee r(x)) \rightarrow s(x))) & \textit{Axiom Pel24}_4 \\
\wedge & \neg(\exists[x](p(x) \wedge r(x))) & \textit{Theorem Pel24 (negiert)}
\end{array}
$$

Durch Bilden der Negations-Normalform und Skolemisierung entsteht die folgende Formel:

$$
\begin{array}{lll}
 & (\neg s(A) \vee \neg q(A)) & \textit{Untergraph } \mathcal{G}_1(A) \\
\wedge & (\neg p(B) \vee q(B) \vee r(B)) & \textit{Untergraph } \mathcal{G}_2(B) \\
\wedge & (p(sk0) \vee q(sk1)) & \textit{Knoten 6 und 7 in } \mathcal{S}_0 \\
\wedge & ((\neg q(C) \wedge \neg r(C)) \vee s(C)) & \textit{Untergraph } \mathcal{G}_3(C) \\
\wedge & (\neg p(D) \vee \neg r(D)) & \textit{Untergraph } \mathcal{G}_4(D)
\end{array}
$$

Diese wird intern in Form des initialen γ-Graphen $\mathcal{S}_0$ mit den γ-Untergraphen $\mathcal{G}_1, \ldots, \mathcal{G}_4$ repräsentiert. Da der Schalter `tree(on)` gesetzt ist, erscheinen diese Graphen wie in Abbildung B.1 dargestellt auf dem Bildschirm.

Nach kurzer Zeit wird das Ergebnis des Beweislaufs angezeigt:

```
Loading File pel24
conv                    16 ms
Graph-Size :
total nodes             12
literal nodes           12
<=> ! nodes             0
 !    nodes             0
free variables          4
gamma subgraphs         4
Tree-Size :
Leaves 0/1              69/72
% compiling file /home/emmy/ludaesch/dipl/shtmp.501.pl
% shtmp.501.pl compiled in module run, 1.033 sec 6,116 bytes
visited Lit-nodes  20
visited Cut-nodes  0
backtracking       0
closed paths       14
reopened paths     0
tried extensions   9
succ. extensions   9
proof search       0 ms
proof              yes
```

Der instantiierte Beweisbaum in Abbildung B.2 zeigt die vorgenommenen γ-Extensionen.

Abschließend stellen wir noch das im Laufe des Beweises generierte Prolog-Programm dar. Die Arbeitsweise der erzeugten Klauseln ist in Abschnitt 4.2.6 beschrieben.

```
:-module(run,[closed_graph/0]).
:-use_module(library(ctr),[ctr_set/2,ctr_inc/1]).
:-use_module(sh_home(output),[prot_format/2]).
:-use_module(sh_home(gammarunlib),[add2path/3,close/2,extend/3]).

closed_graph :-
        node(ga1,path([]/[],[]/[],[]/[],[]/[]),_,ga(0,0,0,0)).

node(ga1,Pa,V,ga(Ga1,Ga2,Ga3,Ga4)):-
    Gn1 is Ga1+1,
    node(ga2,Pa,V,ga(Gn1,Ga2,Ga3,Ga4)).

node(ga2,Pa,V,ga(Ga1,Ga2,Ga3,Ga4)):-
    Gn2 is Ga2+1,
    node(6,Pa,V,ga(Ga1,Gn2,Ga3,Ga4)).

node(6,path(Pr1,Pr2,Neg/Pos,Pr4),V,Ga):-
    (close(-6:path(sk0),Pos)
    ; add2path(-6:path(sk0),Neg,Neg1),
      node(7,path(Pr1,Pr2,Neg1/Pos,Pr4),V,Ga)),
    (close(6:path(sk0),Neg)
    ; add2path(6:path(sk0),Pos,Pos1),
      node(ga3,path(Pr1,Pr2,Neg/Pos1,Pr4),V,Ga)).

node(7,path(Pr1,Neg/Pos,Pr3,Pr4),V,Ga):-
    (close(7:q(sk1),Neg)
    ; add2path(7:q(sk1),Pos,Pos1),
      node(ga3,path(Pr1,Neg/Pos1,Pr3,Pr4),V,Ga)).

node(ga3,Pa,V,ga(Ga1,Ga2,Ga3,Ga4)):-
    Gn3 is Ga3+1,
    node(ga4,Pa,V,ga(Ga1,Ga2,Gn3,Ga4)).

node(ga4,Pa,V,ga(Ga1,Ga2,Ga3,Ga4)):-
    Gn4 is Ga4+1,
    extend(Pa,V,ga(Ga1,Ga2,Ga3,Gn4)).

node(1,path(Neg/Pos,Pr2,Pr3,Pr4),vars(_,B,C,D),Ga):-
    (close(-1:s(A),Pos)
    ; add2path(-1:s(A),Neg,Neg1),
      extend(path(Neg1/Pos,Pr2,Pr3,Pr4),vars(A,B,C,D),Ga)),
    (close(1:s(A),Neg)
    ; add2path(1:s(A),Pos,Pos1),
      node(2,path(Neg/Pos1,Pr2,Pr3,Pr4),vars(A,B,C,D),Ga)).
gamma_entry(1,1).

node(2,path(Pr1,Neg/Pos,Pr3,Pr4),vars(A,B,C,D),Ga):-
    (close(-2:q(A),Pos)
    ; add2path(-2:q(A),Neg,Neg1),
      extend(path(Pr1,Neg1/Pos,Pr3,Pr4),vars(A,B,C,D),Ga)).

node(3,path(Pr1,Pr2,Neg/Pos,Pr4),vars(A,_,C,D),Ga):-
    (close(-3:path(B),Pos)
    ; add2path(-3:path(B),Neg,Neg1),
```

```
      extend(path(Pr1,Pr2,Neg1/Pos,Pr4),vars(A,B,C,D),Ga)),
    (close(3:path(B),Neg)
    ; add2path(3:path(B),Pos,Pos1),
      node(4,path(Pr1,Pr2,Neg/Pos1,Pr4),vars(A,B,C,D),Ga)).
gamma_entry(2,3).

node(4,path(Pr1,Neg/Pos,Pr3,Pr4),vars(A,B,C,D),Ga):-
    (close(-4:q(B),Pos)
    ; add2path(-4:q(B),Neg,Neg1),
      node(5,path(Pr1,Neg1/Pos,Pr3,Pr4),vars(A,B,C,D),Ga)),
    (close(4:q(B),Neg)
    ; add2path(4:q(B),Pos,Pos1),
      extend(path(Pr1,Neg/Pos1,Pr3,Pr4),vars(A,B,C,D),Ga)).

node(5,path(Pr1,Pr2,Pr3,Neg/Pos),vars(A,B,C,D),Ga):-
    (close(5:r(B),Neg)
    ; add2path(5:r(B),Pos,Pos1),
      extend(path(Pr1,Pr2,Pr3,Neg/Pos1),vars(A,B,C,D),Ga)).

node(8,path(Pr1,Neg/Pos,Pr3,Pr4),vars(A,B,_,D),Ga):-
    (close(-8:q(C),Pos)
    ; add2path(-8:q(C),Neg,Neg1),
      node(9,path(Pr1,Neg1/Pos,Pr3,Pr4),vars(A,B,C,D),Ga)),
    (close(8:q(C),Neg)
    ; add2path(8:q(C),Pos,Pos1),
      node(10,path(Pr1,Neg/Pos1,Pr3,Pr4),vars(A,B,C,D),Ga)).
gamma_entry(3,8).

node(9,path(Pr1,Pr2,Pr3,Neg/Pos),vars(A,B,C,D),Ga):-
    (close(-9:r(C),Pos)
    ; add2path(-9:r(C),Neg,Neg1),
      extend(path(Pr1,Pr2,Pr3,Neg1/Pos),vars(A,B,C,D),Ga)),
    (close(9:r(C),Neg)
    ; add2path(9:r(C),Pos,Pos1),
      node(10,path(Pr1,Pr2,Pr3,Neg/Pos1),vars(A,B,C,D),Ga)).

node(10,path(Neg/Pos,Pr2,Pr3,Pr4),vars(A,B,C,D),Ga):-
    (close(10:s(C),Neg)
    ; add2path(10:s(C),Pos,Pos1),
      extend(path(Neg/Pos1,Pr2,Pr3,Pr4),vars(A,B,C,D),Ga)).

node(11,path(Pr1,Pr2,Neg/Pos,Pr4),vars(A,B,C,_),Ga):-
    (close(-11:path(D),Pos)
    ; add2path(-11:path(D),Neg,Neg1),
      extend(path(Pr1,Pr2,Neg1/Pos,Pr4),vars(A,B,C,D),Ga)),
    (close(11:path(D),Neg)
    ; add2path(11:path(D),Pos,Pos1),
      node(12,path(Pr1,Pr2,Neg/Pos1,Pr4),vars(A,B,C,D),Ga)).
gamma_entry(4,11).

node(12,path(Pr1,Pr2,Pr3,Neg/Pos),vars(A,B,C,D),Ga):-
    (close(-12:r(D),Pos)
    ; add2path(-12:r(D),Neg,Neg1),
      extend(path(Pr1,Pr2,Pr3,Neg1/Pos),vars(A,B,C,D),Ga)).
```

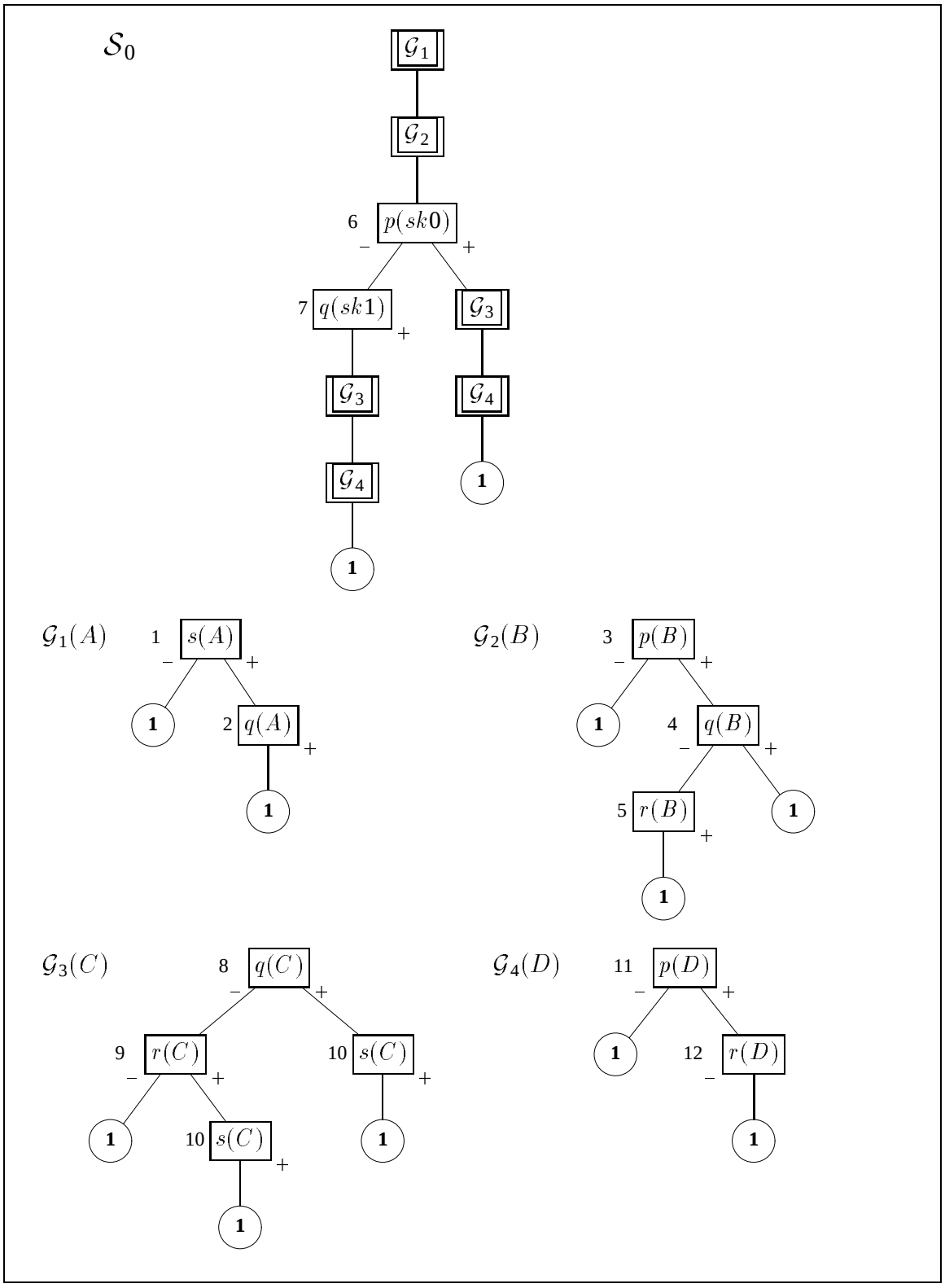

Abbildung B.1: Initialer Shannon-Graph $\mathcal{S}_0$ mit γ-Untergraphen für *pel24*

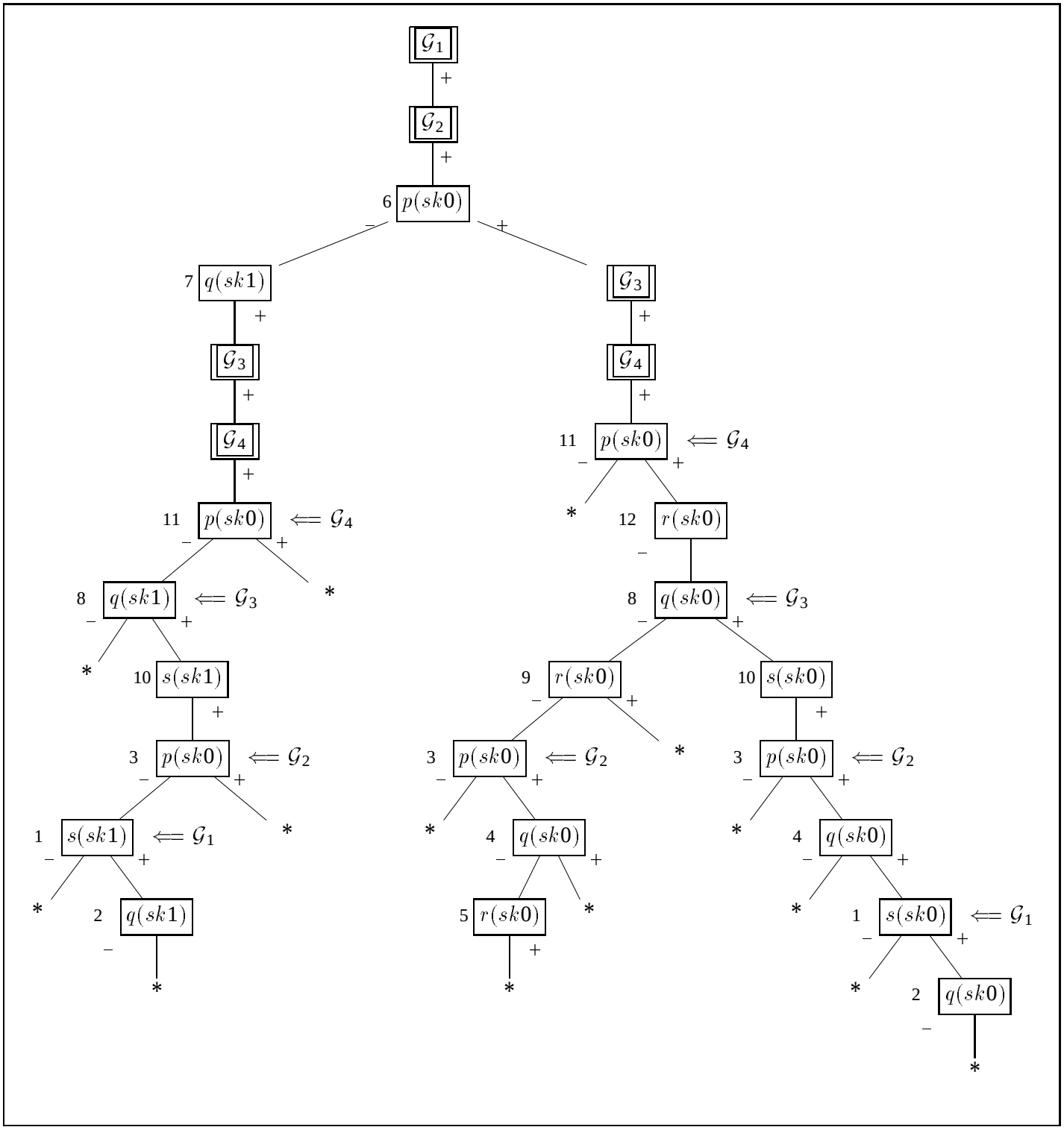

Abbildung B.2: Instantiierter Beweisbaum für *pel24*

Literaturverzeichnis

[Andrews, 1986] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Inc., Orlando, Florida, 1986.

[Bauer & Wirsing, 1991] Friedrich L. Bauer & Martin Wirsing. *Elementare Aussagenlogik*. Springer–Verlag, Berlin, Heidelberg, 1991.

[Beckert & Hähnle, 1992] Bernhard Beckert & Reiner Hähnle. An improved method for adding equality to free variable semantic tableau. In D. Kapur, editor, *Proc. 11^{th} Conference on Automated Deduction CADE, Albany/NY*, volume 607 of *LNCS*, pages 507 – 521. Springer Verlag, 1992.

[Brace *et al.*, 1990] Karl S. Brace, Richard L. Rudell, & Randal E. Bryant. Efficient implementation of a BDD package. In *Proc. 27^{th} ACM/IEEE Design Automation Conference*, pages 40 – 45. IEEE Press, 1990.

[Bryant, 1986] Randal Y. Bryant. Graph–based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C–35:677 – 691, 1986.

[Chang & Lee, 1973] Chin-Liang Chang & Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, London, 1973.

[Church, 1956] Alonzo Church. *Introduction to Mathematical Logic*. Princeton University Press, Princeton, New Jersey, 1956.

[D'Agostino, 1990] Marcello D'Agostino. *Investigations into the Complexity of some Propositional Calculi*. PhD thesis, Oxford University, Computing Laboratory, November 1990.

[D'Agostino, 1992] Marcello D'Agostino. Are tableaux an improvement on truth–tables? Cut–free proofs and bivalence. *To appear in Journal of Logic, Language and Information*, 1992.

[Ehrenfeucht & Orłowska, 1967] A. Ehrenfeucht & E. Orłowska. Mechanical proof procedure for propositional calculus. *Bull. de L'Acad. Pol. des Sci., Série des sci. math., astr. et phys.*, XV(1):25–30, 1967.

[Fitting, 1990] Melvin C. Fitting. *First–Order Logic and Automated Theorem Proving*. Springer, New York, 1990.

[Hähnle *et al.*, 1992] Reiner Hähnle, Bernhard Beckert, Stefan Gerberding, & Werner Kernig. The Many–Valued Tableau–Based Theorem Prover $_3T^AP$. IWBS Report 227, Wissenschaftliches Zentrum Heidelberg, IWBS, IBM Deutschland, Juli 1992.

[Harche *et al.*, 1991] F. Harche, J.N. Hooker, & G.L. Thompson. A computational study of satisfiability algorithms for propositional logic. 1991.

[Lee, 1959] C. Lee. Representation of switching circuits by binary decision diagrams. *Bell System Technical Journal*, 38:985–999, 1959.

[O'Keefe, 1990] Richard O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, Massachusetts, 1990.

[Menzel & Schmitt, 1991] Wolfram Menzel, Peter H. Schmitt. Skriptum zur Vorlesung *Formale Systeme*. Universität Karlsruhe, 1991.

[Orłowska, 1969a] Ewa Orłowska. Automatic theorem proving in a certain class of formulae of predicate calculus. *Bull. de L'Acad. Pol. des Sci., Série des sci. math., astr. et phys.*, XVII(3):117 – 119, 1969.

[Orłowska, 1969b] Ewa Orłowska. Mechanical theorem proving in a certain class of formulae of the predicate calculus. *Studia Logica*, XXV:17 – 27, 1969.

[Pelletier, 1986] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191 – 216, 1986.

[Posegga & Ludäscher, 1992] Joachim Posegga & Bertram Ludäscher. Towards first-order deduction based on shannon graphs. In *Proceedings 16th German Workshop on Artificial Intelligence (GWAI), Bonn*. Springer Lecture Notes in AI, 1992.

[Posegga, 1992] Joachim Posegga. *First–order Deduction with Shannon Graphs*. PhD thesis, Universität Karlsruhe, (Vorabversion).

[Quintus Computer Systems, Inc., 1990] Quintus-Prolog 3.0 User's Guide Quintus Computer Systems, Inc. Mountain View, California

[Schöning, 1987] Uwe Schöning. *Logik für Informatiker*. BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 1987.

[Shannon, 1938] C. E. Shannon. A symbolic analysis of relay and switching circuits. *AIEE Transactions*, 67:713 – 723, 1938.

[Sterling & Shapiro, 1986] Leon Sterling & Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1986.