Diplomarbeit

Implementierung einer Variante des Davis-Putnam (Un-)Erfüllbarkeitsalgorithmus für Aussagen, die um endliche Mengen-Prädikate erweitert sind.

Professur für
Künstliche Intelligenz und Softwaretechnologie

vorgelegt von:	Thomas Ilgner
Fachbereich:	Informatik und Mathematik
Matrikelnummer:	29 33 997
Erstgutachter:	Prof. Dr. M. Schmidt-Schauß

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Erklärung gemäß DPO §11 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Gründau, den 9. Juli 2012

Thomas Ilgner

Inhaltsverzeichnis

Abkürzungsverzeichnis

DPLL	Davis-Putnam-Logemann-Loveland Algorithmus
KNF	Konjunktive Normalform
DPFS	Davis-Putnam-With-Finite-Sets Algorithmus
MOMS	Maximum Occurrencies In Clauses of Minimum Size
DLCS	Dynamic Largest Combined Sum
DLIS	Dynamic Largest Individual Sum
Patricia-Tries	Practical Algorithm To Retrieve Information Coded In Alphanumeric tries
SAT	Entscheidbarkeit von aussagenlogischen Formeln
GHC	Glasgow Haskell Compiler

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Motivation und Zielsetzung

In der Aussagenlogik lassen sich einfache Probleme darstellen. Die Erfüllbarkeit einer aussagenlogischen Formel ist ein Entscheidbarkeitsproblem (SAT) und ist in polynomieller Zeit, in Abhängigkeit von der vorkommenden Anzahl der Variable entscheidbar.

In der Literatur gibt es verschiedene effiziente Verfahren zur Entscheidbarkeit von aussagenlogischen Formeln. Die Davis-Putnam Prozedur ist ein schnelles algorithmisches Verfahren, um eine aussagenlogische Formel auf ihre Unerfüllbarkeit zu testen. Viele Entscheidungsalgorithmen - [ZS00; Bac02] - basieren auf dieser ursprünglichen Prozedur.

Die Prozedur verwendet Deduktion und Heuristiken. Durch Deduktion wird versucht Schlussfolgerungen auf Variablenbelegungen zuschliessen. Hingegen werden Heuristiken benutzt, wenn keine Schlussfolgerungen mehr möglich sind, um eine noch nicht belegte Variable mit $Wahr$ oder $Falsch$ zu belegen. Des Weiteren lässt sich die Prozedur leicht modifizieren, sodass im Falle der Erfüllbarkeit auch ein Modell, d.h eine Interpretation von Variablenbelegungen, ausgegeben werden kann.

Jedoch gibt es Problemstellungen, dargestellt in einer aussagenlogischen Formel, welche eine sehr große Anzahl von Variablen benötigt. In Konsequenz ist solch eine Formel weniger performant und übersichtlich. Aus diesem Grund wäre es vorteilhaft, wenn die Anzahl der Variablen durch eine andere Kodierung des jeweiligen Problems reduziert werden könnte.

Ziel dieser Arbeit ist die Implementierung von zwei Varianten in der Programmiersprache Haskell, basierend auf dem Davis-Putnam Verfahren für Aussagen, die zum einen um endliche Mengen-Prädikate und zum anderen um Mengengleichheiten[1] erweitert sind. Dabei soll die Korrektheit und Vollständigkeit des Davis-Putnam Verfahrens erhalten bleiben.

[1] Kalküle

Es gibt Situationen, indem eine aussagenlogische Formel in ihrer Variablenanzahl mit den beiden Erweiterungen deutlich reduziert werden kann. Die Motivation liegt darin, signifikante Geschwindigkeitsvorteile in manchen Problemstellungen mit den beiden Erweiterungen zu erreichen.

```
Der Quelltext kann unter der folgenden Adresse runtergeladen werden:
    http://www.ki.informatik.uni-frankfurt.de/diplom/programme/ilgner
```

1.2 Überblick

Abschließend wird noch ein kurzer Überblick der weiteren Kapitel gegeben.

In *Kapitel* 2 werden die Grundlagen der Aussagenlogik, die Konzepte der Funktionalen Programmierung, sowie die zentralen Programmkonstrukte der Programmiersprache Haskell vorgstellt. Darüberhinaus wird noch das *containers*-Modul erläutert, dass in der Implementierung zur Repräsentation der endlichen Mengen-Prädikaten und den Mengengleichheiten dient.

In *Kapitel* 3 wird zurächst eine Beschreibung des David-Putnam Verfahrens vorgenommen und anschließend die beiden Erweiterungen mit ihren Regeln dargestellt. Dabei wird auch auf Unterschiede zwischen den Prozeduren eingegangen. Im Anschluss an die Beschreibungen einer Erweiterung wird die Ausdruckskraft anhand eines Beispiels exemplarisch gezeigt.

In *Kapitel* 4 werden die Implementierungdetails erläutert. Die Prozedur besteht in drei ineinander übergehenden Teilen: Parsen der Eingabe, Transformation in eine Konjunktive Normalform und der Test auf (Un-)Erfüllbarkeit.

In *Kapitel* 5 werden die beiden implementierten Erweiterungen, die in Kapitel 3 beschrieben werden, mit Beispielen getesten und diese experimentellen Ergebnisse analysiert.

In *Kapitel* 6 wird eine Zusammenfassung der Arbeit, sowie Anregungen meinerseits für zukünftige Verbesserungen dieser Arbeit gegeben.

Zum Abschluss werden in *Kapitel* 7 die Beispiele, welche in *Kapitel* 5 - Analyse - verwendet werden, kurz vorgestellt.

2 Grundlagen

2.1 Aussagenlogik

In diesem Teilabschnitt wird eine kurze Einleitung der Aussagenlogik, wie in [Lau12; SS11] nachzulesen ist, gegeben. Auf die Syntax wird im *Kapitel* 4 eingegangen, sodass an dieser Stelle nur informell die Aussagenlogik behandelt wird. In der Logik ist eine Aussage ist ein Satz, der entweder $Wahr$ oder $Falsch$ ist. Aussagen können miteinander logisch verknüpft werden. Die nachfolgende Auflistung zeigt die möglichen Verknüpfungen, wobei A und B zwei Aussagen. Verknüpfte Aussagen sind selbst wieder eine Aussage.

A $\wedge$ B	Konjunktion (Verundung)
A $\vee$ B	Disjunktion (Veroderung)
A $\Rightarrow$ B	Implikation
A $\Leftrightarrow$ B	Äquivalenz
A $\neg$ B	negierte Formel

2.2 Funktionale Programmierung

Der kommende Teilabschnitt soll einen kurzen Überblick über das Thema bzw. der Konzepte der funktionalen Programmierung geben, wie in zahlreichen Lehrbüchern nachzulesen ist. [Bir88b; Bir88a; PP06] Unter dem Begriff der Funktionalen Programmierung versteht man ein ganz bestimmtes Programmier-Paradigma.

Ein funktionales Programm besteht demnach aus einer Menge von Funktionsdefinitionen, wobei der Funktionsbegriff aus der modernen Mathematik hergeleitet wird.[2]

[2] Unter einer mathematischen Funktion versteht man eine Beziehung zwischen zwei Mengen, indem jeden Eingabelement x einer Funktion aus dem Definitionsbereich eindeutig ein Element y aus dem Wertebereich zugeordnet wird.

Die Definition einer Funktion hat die Form:

$$quadrat\ x = x * x \tag{2.1}$$

Dies entspricht einer Funktion mit Namen *quadrat*, mit einem Argument x und dem Ausdruck $x * x$, welcher als Funktionrumpf bezeichnet wird.
Zum fundamentalen Konzept der funktionalen Programmierung gehöhren die Funktionen höherer Ordung[3]. Funktionen höherer Ordung sind Funktionen, welche als Argumente ebenfalls Funktionen oder als Resultat Funktionen erlauben. In Konsequenz lassen sich Programme kompakter gestalten.
Die Programmausführung ist die Anwendung der Funktionen auf ihre Argumente. Im Folgenden wird dafür die Funktion in Gleichung 2.2 betrachtet. Die Argumente der Funktion werden in dem Funktionsrumpf ersetzt.

$$quadrat\ 5 \rightarrow 5 * 5 \rightarrow 25 \tag{2.2}$$

Im Abschnitt Auswertungsstrategien wird darauf nochmals Bezug genommen.

Nachfolgend wird kurz auf die Unterschiede zwischen imperativen und funktionalen Programmiersprachen eingegangen. Bei imperativen Programmiersprachen besteht ein Programm aus einer Abfolge von Anweisungen, welche den Zustand der Maschine sukzessive verändern. Das Resultat des Programms ist der veränderte Zustand nach der Abarbeitung alle Anweisungen, während es bei funktionalen Programmiersprachen keinen internen Zustand eines Programms gibt. Das Ergebnis eines Programms ist die Auswertung eines Ausdrucks. Innerhalb eines Programms gibt es keine Variablenzuweisungen. Variablen können nur als Argumente einer Funktion übergeben werden. Dies hat zur Folge, dass es zu keinen Seiteneffekten kommen kann.

Funktionale Programmiersprachen lassen sich anhand der verwendeten Konzepte und Paradigmas unterscheiden. Im Folgenden werden die Konzepte vorgestellt.

[3] engl. high-order functions

2.2.1 Pure und impure funktionale Programmiersprachen

Funktionale Programmiersprachen lassen sich klassifizieren in pure und impure Sprachen.[Sab93] Eine pure Funktionale Programmiersprachen basiert auf dem definierten Lambda-Kalkül. Eine kurze Erläuterung dazu, wird in [Jun04] gegeben.

Die beiden wichtigsten Eigenschaften sind zum einen die Referentiellen Transparenz[4] und zum anderen die Unabhängigkeit der Reihenfolge der Auswertung eines Ausdrucks. Referentiellen Transparenz bedeutet, wenn ein Ausdruck mehrfach angewendet mit ein und derselben Variablenbelegung als Argumente, dann wird dieser Ausdruck genau zum selben Wert/Ergebnis ausgewertet. Diese Eigenschaft hat die Konsequenz:

- Wird ein Wert eines Audrucks in einer puren Programmiersprache nicht verwendet, kann er ohne Weiteres gelöscht werden.
- Beliebige Teilausdrücke mit gleichem Wert können durch Ausdrücke mit gleichem Wert ersetzt werden.
- Sofern keine Abhängigkeit zwischen zwei Teilausdrücken, dann kann die Reihenfolge verändert werden oder die Auswertung in parallel erfolgen.

Impure Funktionale Programmiersprachen habe dagegen einen direkten Ansatz zur Verwendung von veränderbaren Zuständen.

2.2.2 Auswertungsstrategien

Applikative Reihenfolge (call-by-value)

Bei der applikativen Reihenfolge werden zuerst die Argumente ausgewertet, bevor diese in den Funktionsrumpf eingesetzt werden. Als Beispiel dient die Funktion *quadrat*, die bereits oben verwendet wurde.

$$
\begin{aligned}
& \text{quadrat}(\ 4 + 5\ ) \\
\rightarrow\ & \text{quadrat}(\ 9\ ) \\
\rightarrow\ & 9 * 9 \\
\rightarrow\ & 81
\end{aligned}
$$

[4] engl. referential transparency

Funktionale Programmiersprachen, die die applikative Reihenfolge verwenden, werden als *strikt* bezeichnet.

Normale Reihenfolge (call-by-name)

Bei der normalen Reihenfolge werden die Argumente sofort in den Funktionsrumpf eingesetzt, ohne diese vorher auszuwerten. Anschließend wird der Gesamtausdruck ausgewertet. Die Funktion *quadrat* wird wie folgt ausgewertet:

$$
\begin{aligned}
& \text{quadrat}(\ 4 + 5\ ) \\
\rightarrow\ & (\ 4 + 5\ ) * (\ 4 + 5) \\
\rightarrow\ & 9 * (\ 4 + 5\ ) \\
\rightarrow\ & 9 * 9 \\
\rightarrow\ & 81
\end{aligned}
$$

Der Nachteil dieser Strategie ist es, dass Ausdrücke doppelt ausgewertet werden. Im Beispiel oben wird der Ausdruck $(4 + 5)$ doppelt ausgewertet. Funktionale Programmiersprachen, die die normale Reihenfolge verwenden, werden als $(nicht - strikt)$ bezeichnet.

Verzögerte Reihenfolge (call-by-need)

Bei der verzögerten Reihenfolge wird eine optimale Anzahl der Reduktionen erreicht. Es werden ebenfalls die Argumente sofort in den Funktionsrumpf eingesetzt, ohne diese vorher auszuwerten. Jedoch werden unnötige Mehrfachauswertungen von Ausdrücken durch sogenanntes *sharing* vermieden.

$$
\begin{aligned}
& \text{quadrat}(\ 4 + 5\ ) \\
\rightarrow\ & (\ 4 + 5\ )^* * (\ 4 + 5)^* \\
\rightarrow\ & 9 * 9 \\
\rightarrow\ & 81
\end{aligned}
$$

Funktionale Programmiersprachen, die die verzögerte Reihenfolge verwenden, werden als *lazy* bezeichnet.

2.2.3 Haskell

Die Programmiersprache Haskell ist eine pure[5], nicht-strikte und statisch polymorph getypte[6] funktionale Programmiersprache.[SS09; Lip11] In diesem Teilabschnitt werden die grundlegenden Konzepte von Haskell, die auch in der später beschriebenen Implementierung im Rahmen dieser Arbeit verwendet wird, vorgestellt.

2.2.3.1 Datentypen

Mit dem Schlüsselwort *data* lassen sich in Haskell eigene Datentypen definieren. Zum Beispiel lässt sich der Datentyp eines Paares wie folgt darstellen:

$$data\ Pair\ a\ b\ =\ Pair\ a\ b$$

Die Definition eines Datentyps wird in Haskell mit dem *data*-Schlüesselwort eingeleitet. Das Wort *Pair* nachfolgend zu *data* bezeichnet den Namen des Datentyps. Der Datentyp besitzt zwei Typvariablen a und b. Da Haskell ein polymorphes Typsystem hat, können Typvariablen verwendet werden. Nach dem Gleichheitszeichen werden die Daten-Konstruktoren festgelegt. In diesem Fall gibt es nur einen Daten-Konstruktor. Es können auch mehrere Daten-Konstruktoren definiert werden, indem diese durch das pipe-Zeichen getrennt werden. Der Datentyp *Bool*, wie er innerhalb von Haskell benutzt wird, besitzt die beiden Daten-Konstruktoren $True$ und $False$.

$$data\ Bool\ =\ True\ |\ False$$

In Haskell gibt es zusätzlich die Möglichkeit Typsynonyme zu definieren. Zum Beispiel ist in Haskell der Typ *String* eine Liste von Typ *Char*. Zur besseren Lesbarkeit des Quelltextes ist bereits das Typsynonym dafür definiert.

$$type\ String\ =\ [Char]$$

[5] deut. reine
[6] Bei statisch getypten Sprachen findet der Typcheck schon während des Übersetzens statt.

2.2.3.2 Pattern Matching

Pattern Matching[7] ist ein mächtiges Merkmal von Haskell, mit dessen Hilfe es möglich ist, verschiedene Funktionsrumpfe für verschiedene Patterns, bzw. Muster zu definieren. Zum Beispiel die Fibonacci Funktion lässt sich mit Pattern Matching einfach beschreiben.

$$\begin{aligned} \text{fib } 0 &= 1 \\ \text{fib } 1 &= 1 \\ \text{fib } n &= \text{fib } (n\text{-}1) + \text{fib } (n\text{-}2) \end{aligned} \tag{2.3}$$

Pattern Matching kann ebenfalls dazu verwendet werden, um einen Datentyp nach den jeweiligen Daten-Konstruktoren zu trennen. Der Datentyp Bool aus dem letzten Teilabschnitt besitzt zwei Daten-Konstruktoren. Die nachfolgende Funktion f zeigt wie Pattern Matching in diesem Fall zum Trennen benutzt werden kann. Jeder Daten-Konstruktor entspricht einem Muster, welches abgelichen werden muss.

$$\begin{aligned} \text{f True} &= 0 \\ \text{f False} &= 1 \end{aligned} \tag{2.4}$$

2.2.3.3 Guards

Guards[8] ermöglicht eine Funktion in Abschnitte mit unterschiedlichen Funktionsrumpfen zu unterteilen. Im Grunde werden die übergebenen Argumente der Funktion nach besonderen Eigenschaften überprüft, sodass diese entweder $Wahr$ oder $Falsch$ sind. Guards werden angedeutet mit dem senkrechten Strich[9] | und einem nachfolgenden aussagenlogischen Ausdruck (Prädikat). Wird der Ausdruck zu $Falsch$ ausgewertet, wird der nächst folgende Guard getestet. Für den Fall, dass der Ausdruck zu $Wahr$ ausgewertet wird, führt dies zur Auswertung des Funktionsrumpfes des jeweiligen Guards. (Der Ausdruck nach dem Gleichheitszeichen)

[7] deutsch. Musterabgleich
[8] deut. Wächter
[9] engl. pipe character

Die untere Gleichung zeigt eine Funktion, die für zwei Argumente x und y prüft, ob diese ungleich sind. Dieser Sachverhalt kann mit Hilfe von Guards folgendermaßen definiert werden:

$$\begin{array}{lllll} \text{notEqual x y} & | & \text{x} > \text{y} & = & True \\ & | & \text{x} < \text{y} & = & True \\ & | & \text{otherwise} & = & False \end{array} \tag{2.5}$$

Der Ausdruck *otherwise* ist in Haskell vordefiniert vom Typ $True$, sodass dieser Guard immer zu $Wahr$ ausgewertet wird.

2.2.3.4 List Comprehension

In Haskell dienen List Comprehensions zum filtern, transformieren und vereinen von Listen. Sie sind besonders geeignet, um komplexe Listen mit besonderen Eigenschaften zu konstruieren. Die Syntax einer List Comprehension lässt sich wie folgt beschreiben:

$$[\, expr \mid q_1, \cdots, q_n \,]$$

Wobei $expr$ ist ein beliebiger Ausdruck und q_i ist entweder:

- Ein Generator der Form ($pattern \leftarrow e_i$), wobei e_i eine Liste vom beliebigen Typ, oder
- eine aussagenlogischer Ausdruck, welcher Elemente aus der Ergebnisliste entfernt oder
- eine lokale Definition.

2.2.4 Container Modul

Das Containers Paket beinhaltet effiziente Implementierungen von allgemeinen Datenstrukturen. Unter anderen werden die Module $IntMap$, Map und $IntSet$ bereitgestellt. Wobei es sich bei den ersten beiden Modulen um Zuordnungen von Schlüsseln zu Werten und beim Modul $IntSet$ um eine Mengenrepräsentation von Integer Typen handelt.

Die Implementierungen basiert auf der Verwendung von $big-endian$ Patricia-Bäumen. Diese Datenstruktur hat sich als besonders performant, vor allem auf den Operationen Mengenvereinigung und Mengenschnitt, gezeigt. Eine ausführliche Beschreibung ist in [OG98] nachzulesen.

3 Davis-Putnam Verfahren

Das Davis-Putnam Verfahren wurde ursprünglich entwickelt von Martin Davis und Hilary Putnam[MD60, DP 1960] und kurze Zeit später zusammen mit George Logemann und Donald Loveland weiterentwickelt.[MD62, *Davis − Putnam − Logemann − Loveland Algorithmus*(DPLL) 1962] Das Verfahren dient zur Entscheidung der Unerfüllbarkeit von aussagenlogischen Formeln[10] in Konjunktiver Normalform. Darüberhinaus verwendet das Verfahren das Resolutionsverfahren für aussagenlogische Formeln und Fallunterscheidungen.

Im Folgenden wird kurz auf Resolutionsverfahren für aussagenlogische Formeln eingegangen und im Anschluss das allgemeine Verfahren bzw. der Algorithmus des Davis-Putnam genauer betrachtet. Im letzten Teil des Kapitels werden die Erweiterungen des Davis-Putnam Verfahren, welche im Rahmen dieser Diplomarbeit entwickelt wurden, ausführlich vorgestellt.

3.1 Resolutionsverfahren für aussagenlogische Formeln

Die *Entscheidbarkeit von aussagenlogischen Formeln*(SAT)[11] ist ein zentrales Problem in der Theoretischen Informatik. SAT gehört zu der Klasse der NP-vollständigen Probleme, d.h aussagenlogische Formeln sind in polynomieller Zeit, in Abhängigkeit von der vorkommenden Anzahl der Variable/Literale entscheidbar. Als nächstes wird kurz auf zwei Methoden eingegangen, um eine aussagenlogische Formel auf Erfüllbarkeit zu testen. Die erste Methode ist die Lösung durch Aufstellung einer Wahrheitstabelle. Dabei wird jede aussagenlogische Variable mit allen möglichen Kombinationen der Wahrheitswerte $Wahr$ und $Falsch$ belegt.

[10] engl. propositional logic
[11] engl. satisfiability

A	**B**	**C**	**A** $\wedge$ **B**	(**A** $\wedge$ **B**) $\Rightarrow$ **C**	((**A** $\wedge$ **B**) $\Rightarrow$ **C**) $\Rightarrow$ **C**
1	1	1	1	1	**1**
1	1	0	1	0	**1**
1	0	1	0	1	**1**
1	0	0	0	1	**0**
0	1	1	0	1	**1**
0	1	0	0	1	**0**
0	0	1	0	1	**1**
0	0	0	0	1	**0**

Tabelle 3.1: Wahrheitstabelle zum Beispiel 3.1.1

Beispiel 3.1.1. $((A \wedge B) \Rightarrow C) \Rightarrow C)$

Beispiel 3.1.1 zeigt für die aussagenlogische Formel $((A \wedge B)$ => $C)$ => $C)$ die jeweilige Wahrheitstabelle. Diese Wahrheitstabelle gibt für alle möglichen Kombinationen der Variablen die Lösungen an. Diese triviale Methode ist allerdings sehr aufwändig. Für eine Formel mit n Variablen müssen 2^n-Kombinationen getestet werden.

Im Gegensatz dazu, ist das Resolutionsverfahren ein algorithmisches Verfahren zum Erkennen von Widersprüchen. Anstelle eine aussagenlogische Formel auf Erfüllbarkeit zu testen, wird sie auf ihre Unerfüllbarkeit bzw. Widerspruch getestet.

Das Verfahren besteht aus einer einzigen Regel. Dabei wird eine neue Klausel[12] aus zwei Klauseln einer Klauselmenge[13] hergeleitet, wobei in den beiden bestehenden Klauseln jeweils das Komplement eines gleichen Literals vorkommt. Diese neu hergeleitete Klausel wird als Resolvente bezeichnet und der bestehenden Klauselmenge hinzugefügt. In der unteren Formel sei x_i das Komplement von y_i und unter dem Strich ist die hergeleitete Klausel, die Resolvente.

Resolutionsregel:

$$\frac{\begin{array}{ll}(x_1 \vee \cdots \vee x_i \vee \cdots \vee x_n) & \text{Klausel 1} \\ (y_1 \vee \cdots \vee y_j \vee \cdots \vee y_m) & \text{Klausel 2}\end{array}}{(x_1 \vee \cdots \vee x_{i-1} \vee x_{i+1} \vee \cdots \vee x_n \vee y_1 \vee \cdots \vee y_{j-1} \vee y_{j+1} \vee \cdots \vee y_m)}$$

[12] Eine Klausel ist die Disjunktion von Literalen, wobei Literale logische Variablen entsprechen, welche in negierter und nich-negierter Form vorkommen

[13] Eine Klauselmenge ist die Konjunktion von Klauseln

Diese Regel wird nun solange wiederholt bis entweder eine leere Klausel als Resolvente der Klauselmenge hinzugefügt wird oder die Regel nicht mehr anwendbar ist. Wird eine leere Klausel als Resolvente hergeleitet, dann führt dies zum Widerspruch und die Formel bzw. die Klauselmenge ist nicht erfüllbar.

3.2 Allgemeines Verfahren

Dieser Abschnitt dient der tieferen Betrachtung des Davis-Putnam Verfahrens. Wie im einleitenden Text bereits beschrieben, eignet sich das Verfahren zur Entscheidung der Unerfüllbarkeit von aussagenlogischen Formeln in Konjunktiver Normalform.

Konjunktive Normalform (**KNF**)[14]
Eine aussagenlogische Formel ist in einer kunjunktiven Normalform, falls sie aus Konjunktionen[15] von Disjunktionstermen besteht. Wobei Disjunktionsterme aus Disjunktionen[16] von Literalen bestehen.

$$(X_1 \wedge Y_1) \vee (X_2 \wedge Y_2) \vee \cdots \vee (X_n \wedge Y_n) \tag{3.1}$$

Gleichung (3.1) zeigt eine Formel in kunjunktiver Normalform, wobei X_i und Y_i Literale darstellen. Wie leicht zu erkennen ist, ist eine Formel in konjunktiver Normalform nur erfüllbar, wenn jeder ihrer Disjunktionsterme erfüllbar ist.

Definition 3.2.1. Sofern ein Disjunktionsterm zu logisch $false$ evaluiert wird, dann ist die Formel unter dieser Interpretation nicht erfüllbar.

Zur Erinnerung - Eine Interpretation einer aussagenlogischen Formel, ist eine Zuordnung von Wahrheitswerten zu den in der Formel vorkommenden aussagenlogischen Variablen. Für jede aussagenlogische Formel lässt sich eine äquivalente kunjunktive Normalform finden. Die Transformation einer aussagenlogischen Formel lässt sich wie folgt zusammenfassen:

[14] auch als Klauselnormalform bezeichnet
[15] d.h logisch *und* verknüpfte Terme
[16] d.h logisch *oder* verknüpfte Literale

1. Schritt: Elimination von Äquivalenzen ($\Leftrightarrow$)

$$(X \Leftrightarrow Y) \rightarrow (X \Rightarrow Y) \wedge (Y \Rightarrow X)$$

2. Schritt: Elimination von Implikationen ($\Rightarrow$)

$$(X \Rightarrow Y) \rightarrow (\neg\, X \vee Y)$$

3. Schritt: Elimination von Negationen

4. Schritt: Anwendung der Assoziativität, Distributivität und Kommutativität.

Die oben beschriebene Transformation ist im schlechstesten Fall exponentiell, d.h die Anzahl der Literale in der konjunktiven Normalform wächst exponentiell mit der Größe der Ausgangsformel.[SS11, siehe Abschnitt Normalformen] Allerdings existieren für die Transformation Algorithmen, welche eine aussagenlogische Formel in polynomiellem Aufwand in eine konjunktive Normalform überführen, sodass die Erfüllbarkeit erhalten bleibt.[SS11; CC07]

Beispiel 3.2.1. Transformation der Formel $((A \wedge B) \Rightarrow C) \Rightarrow C)$ in eine äquivalente Konjunktive Normalform.

	$((A \wedge B) \Rightarrow C) \Rightarrow C)$	Ausgangsformel
$->$	$((\neg(A \wedge B) \vee C) \Rightarrow C)$	1. Implikation eliminiert
$->$	$(\neg(\neg(A \wedge B) \vee C) \vee C)$	2. Implikation eliminiert
$->$	$(((A \wedge B) \vee\ \neg C) \vee C)$	Negationen eliminiert
$->$	$((A \vee C) \wedge (B \vee C) \wedge (C \vee \neg C))$	Distributivgesetz
$->$	$((A \vee C) \wedge (B \vee C))$	Neutralitätsgesetze

3.2.1 Beschreibung Algorithmus

Der Algorithmus der das Verfahren beschreibt, besteht im Wesentlichen aus zwei ineinander übergehenden Teilen. Im ersten Teil wird die aussagenlogische Formel vorverarbeitet, indem sie in eine Konjunktive Normalform transformiert wird. D.h alle Äquivalenzen und Implikationen werden aufgelöst. Im zweiten Teil wird die in Konjunktiver Normalform vorliegende Formel, anhand

von Regeln auf Unerfüllbarkeit getestet. Der zweite Teil des Algorithmus verwendet im wesentlichen Rekursion, d.h die Regeln werden in einer bestimmten Reihenfolge immer wieder angewendet. Im weiteren Verlauf wird für die Formel die Bezeichnung Klauselmenge verwendet.

3.2.2 Regeln

1. Eliminierung von 1-Klauseln[17]
   (Klauseln die nur aus einem Literal bestehen)

   a) Gibt es in der Klauselmenge eine leere Klausel, dann führt dies zu einem Widerspruch ($\perp$) und die Klauselmenge ist unerfüllbar.

   b) Besteht die Klauselmenge unter anderem aus zwei 1-Klauseln, wobei in der einen Klausel das nicht-negierte Literal p vorkommt und der anderen das negierte Literal $\neg\ p$. Dann führt dies zu einem Widerspruch ($\perp$) und die Klauselmenge ist unerfüllbar.

   c) Existiert innerhalb der Klauselmenge eine 1-Klausel, d.h eine Klausel bestehend aus einem einzigen Literal der Form P, bzw $\neg\ P$, dann

      - Lösche alle Klauseln in denen das Literal vorkommt.
      - Lösche alle Vorkommen des Komplementes des jeweiligen Literals in allen anderen Klauseln.

2. Wahr-Falsch Regel[18]

   - Existiert in der Klauselmenge C ein Literal ausschließlich in positiver Form (p), bzw. ausschließlich in negierter Form ($\neg\ p$), dann können alle Klauseln, in dem dieses Literal vorkommt gelöscht werden. In der Literatur werden diese Literale auch als isolierte oder pure Literale bezeichnet.

     Begründung:
     Sei C die Klauselmenge und C besteht aus $\{A \wedge R\}$, wobei A die Klauseln beschreibt, indem das Literal p nur positiv vorkommt.

[17] engl. unit clauses
[18] engl. Affirmative-Negative Rule

Nun sei C unerfüllbar und $p = 1$, dann ist die Klausel A erfüllt und es gilt:

$$\{A \wedge R\} \quad \Leftrightarrow \quad \mathrm{R}$$

Somit ist die Klauselmenge C nur dann unerfüllbar, wenn R unerfüllbar ist. In Konsequenz kann A aus der Klauselmenge entfernt werden. Diese Begründung gilt auch für den Fall, dass p nur in negierter Form in der Klausel A vorkommt und es gilt $p = 0$ anstatt $p = 1$.

3. Splitting Regel

   - Ist keine der oberen Regeln mehr anwendbar, dann wird eine noch vorkommende aussagenlogische Variable P aus der Klauselmenge ausgewählt. P wird nun mit $true$ (bzw. $false$) belegt und als 1-Klausel der Klauselmenge hinzugeügt. Führt dies zum Widerpruch durch die oberen Regeln, wird die Klauselmenge zurückgesetzt[19] zu dem Punkt, bevor die 1-Klausel ihr hinzugefügt wurde. Anschließend wird P mit $false$ (bzw. $true$) belegt und ebenfalls als 1-Klausel der Klauselmenge hinzugefügt.
     Führt dies ebenfalls zum Widerpruch, dann ist die aussagenlogische Formel nicht erfüllbar.

3.2.3 Heuristiken für Variablenauswahl

Die Auswahl einer noch vorkommende, unbelegten Variable innerhalb der Splitting-Regel kann die Laufzeit der Prozedur entscheidend beeinflussen. Ziel der Auswahl des richtigen Literals ist es, durch die Regeln möglichst viele Klauseln zu löschen. Nachfolgend werden einige Ansätze für eine algorithmisch-optimale Auswahl beschrieben. Gegenwärtig gibt es viele verschiedene Ansätze[San11, Vgl. Kapitel 4], daher werden an dieser Stelle ausschließlich diese Ansätze betrachtet, die als Auswahlkriterium die Häufigkeit der Literale heranziehen.

1. *zufällige Auswahl:*[20] Eine Variable zufällig auszuwählen ist der wohl trivialste Ansatz, zwar lässt sich dieser leicht implementieren, aber führt

[19] engl. Backtracking
[20] engl. random

in der Praxis zu überwiegend deutlich längeren Laufzeiten. Die Laufzeitanalyse ist Teil des Kapitel 5.

2. *Maximum Occurrencies In Clauses of Minimum Size*(*MOMS*):[21] Diese Strategie wählt eine Variable aus, welche innerhalb der Klauselmenge häufiger im Vergleich zu allen anderen Variablen vertreten ist und dazu möglichst in kurzen Klauseln vorkommt. Der Gedanke hinter dieser Auswahlstrategie ist, wenn eine Variable umso häufiger in der Klauselmenge vorkommt, dass die Wahrscheinlichkeit höher ist Folgerungen daraus zu erzielen. (Anwendung der Regeln 1 und 2)

3. *Dynamic Largest Combined Sum*(*DLCS*)[22] / *Dynamic Largest Individual Sum* (*DLIS*):[23] Für eine Variable v sei V_p die absolute Häufigkeit der positiven Vorkommen (v) dieser Variable und analog V_n die absolute Häufigkeit der negativen Vorkommen ($\neg\, v$) dieser Variable.

   - Nun wird bei der Strategie DLCS eine Variable v ausgewählt, welche in der Summe, negativ wie positiv, am Häufigsten vorkommt. ($V_p + V_n$)
   - Bei der Strategie DLIS wird eine Variable v ausgewählt mit Maximum aus V_p oder V_n. ($max(V_p, V_n)$).

   Die Variable wird dann in beiden Varianten erst mir *true* belegt, wenn $V_p > V_n$, anderenfalls mit $false$.

Abbildung 3.1 zeigt für die Prozedur den rekursiven Algorithmus, der in der Literatur als DPLL[24] bezeichnet wird, wie in [MD60; MD62] nachzulesen ist. Die weiter vorne beschriebenen Regeln werden in dieser Reihenfolge angewendet. Der Aufruf der Prozedur DP(C,L), wobei C die Klauselmenge und L die Lösungsmenge beschreiben, terminiert mit $True$ oder $False$.
Wobei $True$ impliziert, dass die angebene Klauselmenge nicht erfüllbar ist, bzw. zu einem Widerpruch führt, anderenfalls $False$.

[21] engl. Maximum Occurrencies In Clauses Of Minimum Size
[22] engl. Dynamic Largest Combined Sum
[23] engl. Dynamic Largest Individual Sum
[24] benannt nach den Autoren Martin Davis, Hilary Putnam, Donald Loveland und George Logemann

3.2.4 Interpretation bei Erfüllbarkeit

Terminiert der obere Algorithmus mit $False$, dann ist die eingegebene Klauselmenge erfüllbar. Die allgemeine Davis-Putnam Prozedur kann leicht modifiziert werden, sodass im Falle der Erfüllbarkeit auch eine Interpretation bzw. ein Modell berechnet werden kann. Dafür werden die folgenden Annahmen getroffen:

- Während der Regel 1 gelöschte Literale in 1-Klauseln werden als $True$ belegt und der Lösungsmenge hinzugefügt
- Isolierte Literale, die während der Regel 2 gelöscht werden, werden als $True$ belegt und der Lösungsmenge hinzugefügt

Die Prozedur in Abbildung 3.1 ist bereits erweitert, sodass im Fall der Erfüllbarkeit der Klauselmenge C, die Lösungsmenge L ein Model darstellt. D.h eine Interpretation für die Klauselmenge, welche diese erfüllt. Findet der Algorithmus ein Model für die eingegebene Klauselmenge, dann entspricht L dem gefundenen Model.
Die Lösungsmenge L ist zu Beginn eine leere Menge, d.h. keiner der aussagenlogischen Variablen ist mit einem Wert belegt. Ab diesem Zeitpunkt wird versucht durch die Anwendung der Regeln 1[25] und 2[26] Wissen bzw. Folgerungen über die Variablenbelegungen abzuleiten. Ist die Anwendung dieser zwei Regeln nicht mehr möglich, dann muss mit der Splitting Regel eine Fallunterscheidung vorgenommen werden. Je nach Wahl der Heuristik, wird dann ein Literal erst mit $True$ (bzw. $False$) belegt.

Zurücksetzen[27] der Klauselmenge

Führt diese erste Belegungen während der Fallunterscheidung zum Widerpruch, muss die Klauselmenge zurückgesetzt werden. In Abbildung 3.1 wird dieses Zurücksetzen durch eine IF-Klausel beschrieben. Anschließend wird die Rekursion mit dem Komplement der vorherigen Belegung fortgesetzt. Tritt dann ebenfalls ein Widerspruch auf, ist die Klauselmenge nicht erfüllbar.

[25] Eliminierung von 1-Klauseln
[26] Wahr-Falsch Regel
[27] engl. Backtrack

Definition 3.2.2. Der Algorithmus ist korrekt und vollständig

- Terminiert der Algorithmus mit $True$, dann gibt es keine Interpretation/Variablenbelegungen für die eingegebene Klauselmenge. Die Klauselmenge ist somit nicht erfüllbar.
- Terminiert der Algorithmus mit $False$, dann existiert eine Interpretation/Variablenbelegungen für alle vorkommenden aussagenlogischen Variablen, sodass die eingegebene Klauselmenge mit dieser Variablenbelegungen erfüllt ist.

3.3 Verbesserungen

Die Davis-Putnam Prozedur ist eine schnelle Möglichkeit eine aussagenlogische Formel auf ihre Unerfüllbarkeit zu testen. Viele Entscheidungsalgorithmen, vgl. [ZS00; Bac02] basieren auf dieser ursprünglichen Prozedur. Ziel dieser Arbeit ist es, Erweiterungen der Davis-Putnam Prozedur zu entwickeln, welche in Abhängigkeit der Problemstellung signifikante Geschwindigkeitsvorteile erbringen. Genauer gibt es Situationen, indem eine aussagenlogische Formel in ihrer Variablenanzahl mit den beiden Erweiterungen deutlich reduziert werden kann. Die Formel wird dazu in eine neue Struktur überführt, sodass in Konsequenz, die Anwendung der Regeln der Davis-Putnam Prozedur vereinfacht wird. Nachfolgend werden zwei Erweiterungen im Detail genau beschrieben, welche im Rahmen dieser Arbeit von mir implementiert wurden. Anschließend werden die beiden Erweiterungen jeweils an Beispielen veranschaulicht.

3.3.1 Mengen Prädikate

In diesem Abschnitt wird die erste Erweiterung vorgestellt, die im Gegensatz zu aussagenlogischen Variablen (P oder $\neg P$) Atome der Form $x \in \mathcal{M}$ verwendet. Diese Atome sollen in bestimmten Situationen dazu dienen, Teilformeln zusammenzufassen oder gar zu vereinfachen. Im letzten Teil dieses Abschnitts werden diese Zusammenfassungen/Vereinfachungen, anhand eines Beispiels, genauer betrachtet. Die Menge $\mathcal{M}$ beschreibt eine vorher festgelegte endliche Menge. Dazu wurde der allgemeine Davis-Putnam Algorithmus durch spezialisierte Prozeduren erweitert.

Algorithm 1 $DP(C, L)$

Require: C ist eine Konjunktive Normalform und enthält keine Tautologien innerhalb einer Klausel.[28]

if C enthält eine leere Klausel **then**
  TRUE
else
  if C ist eine leere Klauselmenge **then**
    FALSE
  end if
end if

if C enthält eine 1-Klausel P oder $\neg P$ **then**
  - Lösche alle Klauseln in denen das Literal vorkommt
  - Lösche alle Komplemente des Literals in den anderen Klauseln
  $L \leftarrow L \cup P$ (bzw. $\neg P$)
  DP(C_{neu},L)
end if

if C enthält ein isoliertes Literal P oder $\neg P$ **then**
  - Lösche alle Klauseln in denen das Literal vorkommt
  $L \leftarrow L \cup P$ (bzw. $\neg P$)
  DP(C_{neu},L)
end if

Wähle ein noch vorkommendes Literal P aus der Klauselmenge (Splitting Regel)
if DP($\{C \cup P\}$,L) **then**
  DP($\{C \cup \neg P\}$,L)
else
  return TRUE
end if

Abbildung 3.1: Pseudocode des Davis-Putnam-Algorithmus

Definition 3.3.1. Sei $x \in \mathcal{M}$, wobei die endlichen Menge $\mathcal{M} = \{m_1, m_2, \cdots, m_n\}$. Dann entspricht dem Atom der Form $x \in \mathcal{M}$:

$$x = m_1 \vee x = m_2 \vee \ldots \vee x = m_n$$

Als zusätzliche Einschränkung dürfen keine negativen Literal innerhalb der Klauselmenge vorkommen. Eine solches negatives Literal kann durch die Komplementbildung ihrer Menge mit der vorher fixierten endlichen Menge $\mathcal{M}$ eliminiert werden.

Sei $\neg\ x$ *ein negatives Literal in dem Atom* $(\neg\ x) \in M_1$ *und* $\mathcal{M}$ *die fixierte endliche Menge und es gilt:* $M_1 \subset \mathcal{M}$.
Dann wird das negative Literal überführt, indem die Mengendifferenz zwischen der fixierten Menge $\mathcal{M}$ und der endlichen Menge des Literals durchgeführt wird.

$$\neg\ x \in M_1 \xrightarrow{\text{wird überführt}} x \in \{\mathcal{M} - M_1\}$$

3.3.1.1 Regeln

Im Vergleich zum Algorithmus des allgemeinen Davis-Putnam Verfahren, verwendet dieser spezialisierte Algorithmus zwei zusätzliche Regeln. Neben einer Regel zur Simplifikation/Vereinfachung, befassen sich zwei Regeln mit den Fällen, wenn gleiche Literale innerhalb einer Klausel (Regel 3), bzw. mehrerer 1-Klauseln mit gleichem Literal (Regel 2) existieren. Im Folgendem werden die Regeln, in der verwendeten Reihenfolge, im Algorithmus beschrieben.

Regel 1 *Simplifikationen*

Innerhalb der Klauselmenge können verschiedene Simplifikationen/Vereinfachungen vorgenommen werden. Tritt eines der unteren Fällen auf, dann wird die Klauselmenge entsprechend der Beschreibung bzw. der Pfeile simplifiziert.

- Besteht die Klauselmenge C zu Beginn aus einer leeren Klausel, dann führt dies zu einem Widerspruch ($\perp$) und die Klauselmenge ist unerfüllbar.
- Gibt es u.a innerhalb der Klauselmenge C eine leere Klausel, dann führt dies ebenfalls zu einem Widerspruch ($\perp$) und die Klauselmenge ist unerfüllbar.

- $x \in \emptyset \xrightarrow{\text{wird ersetzt durch}} False$
- $x \in \mathcal{M} \xrightarrow{\text{wird ersetzt durch}} True$, wobei $\mathcal{M}$ die festgelegte endliche Menge.

Zusätzlich zu den oberen Simplifikationen: Gibt es innerhalb einer Klausel ein $False$ (z.B. $False \vee R$), dann wird diese Klausel simplifiziert, indem das $False$ aus der Klausel entfernt wird. Im umgekehrten Fall, dass ein $True$ innerhalb der Klausel vorkommt, wird die komplette Klausel aus der Klauselmenge entfernt.

Regel 2 *Zusammenfassen von gleichen Literalen innerhalb einer Klausel*

Existieren Klauseln, in welcher ein Literal x mehrfach vorkommt, dann können die entsprechenden Atome ($x \in M_1$, $x \in M_2$, ..., $x \in M_n$) zu einem einzelnen Atom zusammengefasst werden. Gleichung (3.2) zeigt eine Klausel, wobei das Literal x in diesem Fall in zweifacher Ausführung auftritt und R alle anderen Atome dieser Klausel beschreibt.

$$(x \in M_1 \vee x \in M_2 \vee R) \tag{3.2}$$

Die beiden Vorkommen von x werden nun zusammengefasst, indem ihre beiden Menge M_1 und M_2 vereinigt werden. $\{M_1 \cup M_2\}$ Dies wird illustriert in Gleichung (3.3).

$$x \in M_1 \vee x \in M_2 \xrightarrow{\text{zusammengefasst}} x \in \{M_1 \cup M_2\} \tag{3.3}$$

Begründung:
Wie in Definition 3.3.1 beschrieben, entspricht dem Atom $x \in M$ genau die Veroderung der Wertzuweisungen des Literals mit den Elementen aus der Menge M.

Seien M_1, M_2 zwei endliche Menge, wobei $M_1 = \{m_1, m_2, \ldots, m_k\}$, $M_2 = \{m_{k+1}, m_{k+2}, \ldots, m_n\}$ und $\mathcal{M} = \{m_1, \ldots, m_n\}$.

$$\begin{aligned} x \in M_1 \vee x \in M_2 &\iff x = m_1 \vee \ldots \vee x = m_k \vee x = m_{k+1} \vee \ldots \vee x = m_n \\ &\iff x \in \{M_1 \cup M_2\} \end{aligned}$$

Regel 3 *Zusammenfassen von mehreren 1-Klauseln mit gleichem Literal*

Existiert für ein Literal x mehrere 1-Klauseln ($\{x \in M_1\} \wedge \{x \in M_2\} \wedge \ldots \wedge \{x \in M_n\}$), dann können diese 1-Klauseln durch eine einzelne 1-Klausel ersetzt werden. Wie in Gleichung (3.4) zu sehen, wird dazu ein Mengenschnitt unter allen der beteiligten Mengen M_i durchgeführt.

$$x \in M_1 \wedge x \in M_2 \wedge \cdots \wedge x \in M_n \xrightarrow{\text{zusammengefasst}} x \in \{M_1 \cap \cdots \cap M_n\} \quad (3.4)$$

Regel 4 *Eliminierung von 1-Klauseln*

- Besteht die Klauselmenge unter anderem aus zwei 1-Klauseln, mit den beiden Atomen $x \in M_1$ und $x \in M_2$ und es gilt: $\{M_1 \cap M_2\} = \emptyset$. Dann führt dies zu einem Widerspruch ($\perp$) und die Klauselmenge ist unerfüllbar.
- Existiert innerhalb der Klauselmenge eine 1-Klausel, d.h eine Klausel bestehend aus einem einzigen Atom der Form $x \in M^{'}$, dann
  - können alle Atome mit dem gleichem Literal $x \in M^{''}$ in den anderen Klauseln, durch $x \in \{M^{'} \cap M^{''}\}$ ersetzt werden und
  - diese 1-Klausel, bestehend aus dem Atom $x \in M^{'}$ wird aus der Klauselmenge C entfernt und der Lösungsmenge hinzugefügt.

  Zusätzlich können folgende Simplifikationen vorgenommen werden:
  - Gibt es Klauseln der Form $\{x \in M^{''} \vee R\}$[29], für welche die Beziehung $M^{'} \subseteq M^{''}$ gilt, dann können diese Klauseln gelöscht werden.

    Begründung:
    $M^{'}$ ist eine Teilmenge von $M^{''}$; Definitionsgemäß genau dann, wenn jedes Element von $M^{'}$ auch Element von $M^{''}$ ist. Da das Atom $x \in M^{'}$, in der 1-Klausel der Lösungsmenge hinzugefügt wird, kann aus dem Atom $x \in M^{''}$ nicht mehr weitere Folgerungen geschlossen werden. Die Klausel, welche diese Atom enthält, kann daher gelöscht werden.

[29] R beschreibt die restlichen Atome innerhalb dieser Klausel

$$
\begin{aligned}
M^{'} \subseteq M^{''} \quad &\Longleftrightarrow \quad \forall \mathrm{x} \in M^{'} \ : \ x \in M^{''} \\
&\Longleftrightarrow \quad M^{'} \cup M^{''} = M^{''} \qquad \{Vereinigung\} \\
&\Longleftrightarrow \quad M^{'} \cap M^{''} = M^{'} \qquad \{Durchschnitt\} \\
&\Longleftrightarrow \quad M^{'} \setminus M^{''} = \emptyset \qquad \{Mengendifferenz\}
\end{aligned}
$$

– Gibt es Klauseln der Form $\{x \in M^{''} \vee R\}$ für welche die Beziehung $M^{''} \subset M^{'}$ und $M^{'} \neq M^{''}$ gilt, dann können aus dieser Beziehung keine unmittelbaren Folgerungen geschlossen werden.

Regel 5 *Wahr-Falsch Regel bzw. Regel für isolierte Literale*[30]

Zur Erinnerung: Existiert in der Klauselmenge C ein Literal ausschließlich in positiver Form (p), bzw. ausschließlich in negierter Form $(\neg\, p)$, dann können alle Klauseln, in dem dieses Literal vorkommt, gelöscht werden. Derartige Literale werden auch als isolierte Literale bezeichnet. Da innerhalb dieser Erweiterung Atome der Form $x \in \mathcal{M}$ benutzt werden, muss diese Regel entsprechend angepasst werden.
Für ein Literal x, sei $x \in M_1,\ x \in M_2, \ldots,\ x \in M_n$ alle vorkommenden Atome dieses Literals innerhalb der Klauselmenge. Wie in Gleichung (3.5) gezeigt, ist das Literal x isoliert, wenn der Mengenschnitt aller beteiligten Mengen M_i nicht der leeren Menge $(\emptyset)$ entspricht.

$$
M_1 \ \cap \ M_2 \ \cap \ \ldots \ \cap \ M_n \quad \neq \quad \emptyset \quad \{Mengenschnitt\} \tag{3.5}
$$

Als Begründung dient hier dieselbe Herangehensweise, wie bei Regel 2 im Abschnitt 3.2.2. In diesem Fall können alle Klauseln, welche dieses Literal enthalten, entfernt werden. Anschließend wird das Atom $x \in \{M_1 \cap M_2 \cap \ldots \cap M_n\}$ der Lösungsmenge hinzugefügt.

Regel 6 *Splitting Regel*

Innerhalb dieser Regel wird eine Fallunterscheidung durchgeführt. Dabei ist es erforderlich, dass innerhalb der Klauselmenge keine 1-Klauseln mehr bestehen. Dies wird sichergestellt, indem diese Regel zuletzt im Algorithmus Anwendung findet; D.h. nur wenn keine andere Regel mehr anwendbar ist.
Zunächst wird ein noch vorkommendes Atom $x \in M^{'}$ aus der Klauselmenge ausgewählt. Weiterhin ist $\mathcal{M}$ die als Eingabe für den Algorithmus festgelegte

[30] engl. Affirmative-Negative Rule

endliche Menge. Die Auswahl dieses Atoms erfolgt einer bestimmten Heuristik. Im nächsten Teilabschnitt wird auf diese Heuristik noch genauer eingegangen. Anschließend wird die Fallunterscheidung folgendermaßen durchgeführt:

Sei $x \in M'$ das ausgewählte Atom und $x \in \{\mathcal{M} \setminus M'\}$ dessen Komplement.

- 1. Fall: Das Atom $x \in M'$ wird mit $True$ belegt und als 1-Klausel der Klauselmenge hinzugefügt. $(C \cup \{x \in M'\})$
- 2. Fall: Das Atom $x \in M'$ wird mit $False$ belegt und als 1-Klausel der Klauselmenge hinzugefügt. Wie zu Beginn des Abschnitts bereits erwähnt, dürfen keine negativen Literale innerhalb der Klauselmenge existieren. Derartige Literale müssen - wie in Lemma 3.3.1.2 beschrieben - transformiert werden. $(C \cup \{x \in \{\mathcal{M} \setminus M'\}\})$

Bevor diese beiden Fälle betrachtet werden, wird die Klauselmenge gespeichert. Im Algorithmus finden diese beiden Fälle Anwendung, indem zunächst Fall 1. betrachtet wird. Führt die Belegung dieses Atoms, während der Anwendung der Regeln zum Widerspruch, muss die Klauselmenge zurückgesetzt werden. D.h. alle Veränderungen, welche durch die Regeln durchgeführt wurden, müssen zurückgenommen werden. Da zuvor die Klauselmenge gespeichert wurde, wird der zweite Fall eben mit dieser gespeicherten Klauselmenge betrachtet.

3.3.1.2 Heuristik der Atomauswahl

In Abschnitt 3.2.3 wurden bereits einige Heuristiken für eine mögliche Auswahl einer aussagenlogischen Variablen beschrieben. Der Algorithmus, wie er im Folgenden beschrieben wird, benutzt die MOMS[31]-Strategie. Ein Atom $x \in M'$ wird demnach ausgewählt, sodass:

- das Literal x am Häufigsten innerhalb der Klauselmenge vorkommt und
- unter allen vorkommenden Atomen dieses Literals, genau das gewählt wird, mit der kleinsten Menge M'.

[31] engl. Maximum Occurrencies In Clauses Of Minimum Size

3.3.1.3 Beschreibung Algorithmus

Nachdem im letzten Abschnitt die Vorgehensweise aller Regeln beschrieben wurde, steht jetzt die Anwendungsreihenfolge im Vordergrund. Abbildung 3.2 illustriert den Pseudocode des Algorithmus. Analog zum allgemeinen Davis-Putnam-Algorithmus, muss die aussagenlogische Formel in eine Konjunktive Normalform transformiert werden (vgl. Kapitel 3.2). Diese transformierte Klauselmenge, eine festgelegte endliche Menge $\mathcal{M}$ und die Lösungsmenge L sind die drei Eingabeargumente des Algorithmus. Die Lösungsmenge L ist beim ersten Aufruf des Algorithmus leer, d.h es gibt zu Beginn keine Belegungen der Literale. Sofern eine Interpretation, d.h eine Variablenbelegung für die eingegebene Klauselmenge existiert, sodass diese Klauselmenge erfüllt ist, dann terminiert der angebene Algorithmus genau mit dieser Interpretation. Andernfalls terminiert der Algorithmus mit der leeren Menge ($\emptyset$), bzw. mit dem Text, dass die eingegebene Klauselmenge nicht erfüllbar ist.
Der Algorithmus in Abbildung 3.2 ist eine rekursive Prozedur. Jedes Mal wenn, eine Regel die Klauselmenge verändert, wird die Prozedur rekursiv mit der veränderten Klauselmenge neu aufgerufen.

Algorithm 2 $DPFS(C, \mathcal{M}, L)$

Require: Klauselmenge C ist eine Konjunktive Normalform und enthält keine Tautologien innerhalb einer Klausel. Sei $\mathcal{M}$ die festgelegte endliche Menge und L die Lösungsmenge.

if C enthält eine leere Klausel **then**
  return "nicht erfüllbar"
else
  if C ist eine leere Klauselmenge **then**
    return L
  end if
end if

/*Simplifikationen*/
if C enthält Atome $x \in \emptyset$, $x \in \mathcal{M}$ oder $False$ und $True$ **then**
  Simplifiziere diese Atome nach Regel 1.
  Sei C_{neu} die veränderte Klauselmenge nach der Simplifikation.
  $DPFS(C_{neu}, \mathcal{M}, L)$
end if

if C enthält mehrere gleiche Literale innerhalb einer Klausel. **then**
  Zusammenfassung dieser Atome nach Regel 2
  Sei C_{neu} die veränderte Klauselmenge nach der Anwendung der Regel.
  $DPFS(C_{neu}, \mathcal{M}, L)$
end if

if C enthält mehrere 1-Klauseln mit gleichem Literal. **then**
  Zusammenfassung dieser Atome nach Regel 3
  Sei C_{neu} die veränderte Klauselmenge nach der Anwendung der Regel.
  $DPFS(C_{neu}, \mathcal{M}, L)$
end if

if C enthält eine 1-Klausel P oder $\neg P$ **then**
  - Lösche alle Klauseln in denen das Literal vorkommt
  - Lösche alle Komplemente des Literals in den anderen Klauseln
  $L \leftarrow L \cup P$ (bzw. $\neg P$)
  DP(C_{neu},L)
end if

if C enthält ein isoliertes Literal P oder $\neg P$ **then**
  - Lösche alle Klauseln in denen das Literal vorkommt
  $L \leftarrow L \cup P$ (bzw. $\neg P$)
  DP(C_{neu},L)
end if

Wähle ein Atom, nach der beschriebenen Heuristik aus (Splitting Regel), und
if $DPFS(\{C \cup x \in M\}, \mathcal{M}, L)$ **then**
  $DPFS(\{C \cup x \in \mathcal{M} \setminus M\}, \mathcal{M}, L)$
else
  return TRUE
end if

Abbildung 3.2: Pseudocode Erweiterung der DP-Prozedur um Mengen-Prädikate

Beispiel 3.3.1. n-Damen Problem

Zum Abschluss dieses Teilabschnitts der ersten Erweiterung, wird ergänzend als Beispiel das n-Damen Problem betrachtet. Ziel ist es n Damen auf einem $n \times n$-Schachbrett so zu platzieren, dass diese sich nicht schlagen können. Nicht schlagen bedeutet hierbei: Es dürfen nicht mehrere Damen in gleicher Zeile, Spalte oder Diagonalen sein. (siehe Abbildung 3.3) Weiter wird angenommen, dass in jeder Spalten und Zeile genau eine Dame vorkommt.

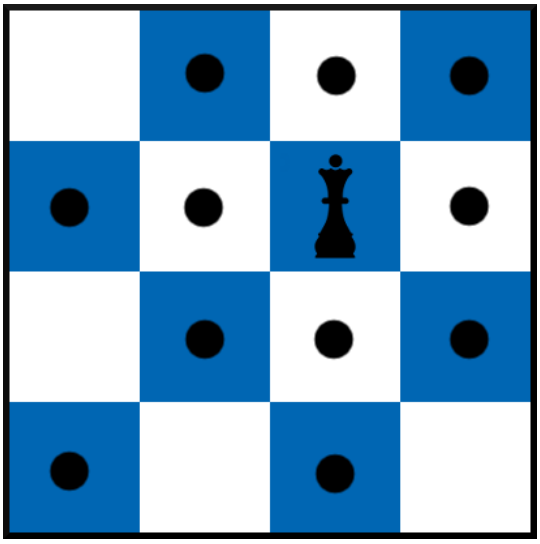


Abbildung 3.3: Eine Dame, wie im Bild platziert, kann potenzielle Damen auf dem Feldern mit dem Kreis markiert schlagen

Zunächst wird die Klauselmenge für die allgemeine Davis-Putnam Prozedur erläutert und im Anschluss folgt die Beschreibung der Klauselmenge für die erste Erweiterung um Mengen-Prädikate. Zur besseren Übersichtlichkeit betrachten wird im weiteren Verlauf das 4-Damen Problem.

Allgemeine Kodierung

Jedes Feld des 4×4-Schachbrett entspricht einer aussagenlogischen Variablen. (P bzw. $\neg P$) Bei $n = 4$ gibt es somit 16 Variablen.

Für jedes Feld bzw. für jede aussagenlogische Variable muss nun die oben angesprochenen Beschränkungen generiert werden. Es wird sich im Folgenden darauf beschränkt, dies exemplarisch am ersten Feld unten links zu zeigen.

Annahme:

Auf dem Feld mit der Nummer eins soll eine Dame platziert werden. Den

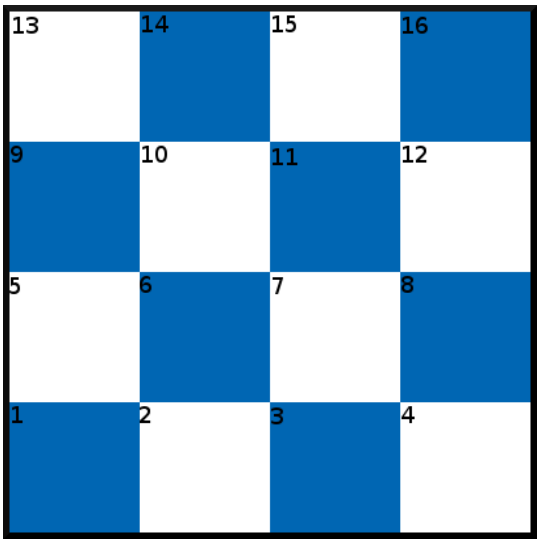


Abbildung 3.4: 4×4-Schachbrett allg. Kodierung der Klauselmenge

Nummern der Felder des Schachbretts wird zum besseren Verständnis, dass es sich um eine Variable handelt, das Präfix x vorangehängt.

- In jeder Zeile soll maximal eine Dame platziert werden. Diese Beschränkung wird durch die Veroderung ($\vee$) der aussagenlogischen Variablen in einer Zeile realisiert.

  $(x1 \vee x2 \vee x3 \vee x4)$
  $(x5 \vee x6 \vee x7 \vee x8)$
  $(x9 \vee x10 \vee x11 \vee x12)$
  $(x13 \vee x14 \vee x15 \vee x16)$

- In jeder Spalte soll maximal eine Dame platziert werden. Analog zur gerade beschriebenen Beschränkung wird dies durch die Veroderung ($\vee$) der aussagenlogischen Variablen in einer Spalte realisiert.

  $(x1 \vee x5 \vee x9 \vee x13)$
  $(x2 \vee x6 \vee x10 \vee x14)$
  $(x3 \vee x7 \vee x11 \vee x15)$
  $(x4 \vee x8 \vee x12 \vee x16)$

- Beschränkung gleiche Spalte:
  Wenn auf Feld 1 eine Dame steht, dann dürfen keine Damen auf den Feldern 5,9 und 13 platziert werden.

$$x1 \implies \neg x5$$
$$x1 \implies \neg x9$$
$$x1 \implies \neg x13$$

- Beschränkung gleiche Zeile:
  Wenn auf Feld 1 eine Dame steht, dann dürfen keine Damen auf den Feldern 2,3 und 4 platziert werden.

  $$x1 \implies \neg x2$$
  $$x1 \implies \neg x3$$
  $$x1 \implies \neg x4$$

- Beschränkung gleiche Diagonale:
  Wenn auf Feld 1 eine Dame steht, dann dürfen keine Damen auf der Hauptdiagonalen, d.h auf den Feldern 6,11 und 16 platziert werden.

  $$x1 \implies \neg x6$$
  $$x1 \implies \neg x11$$
  $$x1 \implies \neg x16$$

Diese Vorgehensweise muss nun für jedes der 16 Felder durchgeführt werden. Nach Optimierungen und Umformung in eine KNF ergeben sich insgesamt 84 Klauseln. In Abbildung 3.5 ist die generierte Klauselmenge in Kurzschreibweise zu sehen. Variablen in eckigen Klammer stellen die Klauseln dar. Die Variablen sind innerhalb der Klausel durch ein Komma getrennt, was der Veroderung entspricht. Die Klauseln wiederum sind ebenfalls durch ein Komma getrennt, was der Verundung entspricht. Ein Minus vor der Variable gibt an, dass diese Variable negiert ist.

Kodierung mit Mengen-Prädikaten

Die Klauseln der Klauselmenge bestehen aus Atomen der Form $x \in \mathcal{M}$. Im Gegensatz zur allgemeinen Kodierung entspricht nur jede Zeile des 4×4-Schachbrett einer aussagenlogischen Variablen. Bei $n = 4$ gibt es somit lediglich 4 Variablen.

Wie bereits eingangs erwähnt, modellieren die Literale die Zeilen und die Mengen-Prädikate die Spalten des Schachbretts. Abbildung zeigt dies für $n = 4$.

Bemerkung: $x1 \in \{1\}$ gibt an, dass in Zeile 1 und in Spalte 1 eine Dame steht.

```
[[x1,x2,x3,x4],[x5,x6,x7,x8],[x9,x10,x11,x12],[x13,x14,x15,x16],
[x1,x5,x9,x13],[x2,x6,x10,x14],[x3,x7,x11,x15],[x4,x8,x12,x16],
[-x1,-x5],[-x1,-x9],[-x1,-x13],[-x1,-x2],[-x1,-x6],[-x1,-x3],
[-x1,-x11],[-x1,-x4],[-x1,-x16],
[-x5,-x9],[-x5,-x13],[-x5,-x2], [-x5,-x6],[-x5,-x10],[-x5,-x7],[-x5,-x15],[-x5,-x8],
[-x9,-x13], [-x9,-x6],[-x9,-x10],[-x9,-x14],[-x9,-x3],[-x9,-x11],[-x9,-x12],
[-x13,-x10],[-x13,-x14],[-x13,-x7],[-x13,-x15],[-x13,-x4], [-x13,-x16],
[-x2,-x6],[-x2,-x10],[-x2,-x14],[-x2,-x3],[-x2,-x7], [-x2,-x4],[-x2,-x12],
[-x6,-x10],[-x6,-x14],[-x6,-x3],[-x6,-x7], [-x6,-x11],[-x6,-x8],[-x6,-x16],
[-x10,-x14],[-x10,-x7],[-x10,-x11], [-x10,-x15],[-x10,-x4],[-x10,-x12],
[-x14,-x11],[-x14,-x15],[-x14,-x8],[-x14,-x16],
[-x3,-x7],[-x3,-x11],[-x3,-x15],[-x3,-x4],[-x3,-x8],
[-x7,-x11],[-x7,-x15],[-x7,-x4],[-x7,-x8],[-x7,-x12],
[-x11,-x15],[-x11,-x8],[-x11,-x12],[-x11,-x16],
[-x15,-x12],[-x15,-x16],
[-x4,-x8],[-x4,-x12],[-x4,-x16],
[-x8,-x12],[-x8,-x16],
[-x12,-x16]]
```

Abbildung 3.5: Klauselmenge allg. Kodierung

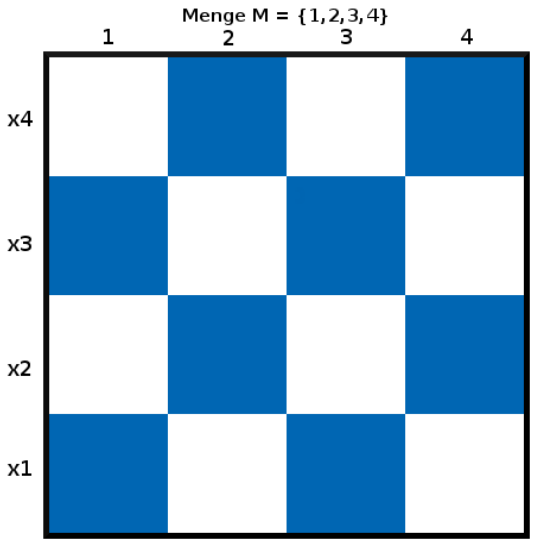


Abbildung 3.6: 4×4-Schachbrett mit Mengen-Prädikaten als Kodierung

Annahme: In Zeile 1 und Spalte 1 soll eine Dame platziert werden. Die jeweiligen Nummern der Zeilen des Schachbretts werden wieder zum besseren Verständnis, dass es sich um eine Variable handelt, das Präfix x vorangehängt.

- Beschränkung gleiche Spalte:
  Es dürfen keine Damen in der gleichen Spalte in den anderen Zeilen $2, 3, 4$ platziert werden.

  $$x1 \in \{1\} \implies x2 \in \{2,3,4\}$$
  $$x1 \in \{1\} \implies x3 \in \{2,3,4\}$$
  $$x1 \in \{1\} \implies x4 \in \{2,3,4\}$$

- Beschränkung gleiche Diagonale:
  In diesem Fall dürfen keine Damen auf der Hauptdiagonalen platziert werden.

  $$x1 \in \{1\} \implies x2 \in \{1,3,4\}$$
  $$x1 \in \{1\} \implies x3 \in \{1,2,4\}$$
  $$x1 \in \{1\} \implies x4 \in \{1,2,3\}$$

Für die Generierung muss diese Vorgehensweise für jede Zeile x_i und für jedes Element aus $\mathcal{M} = \{1, 2, 3, 4\}$ durchgeführt werden. Nach Optimierungen und Umformung in eine KNF ergeben sich insgesamt 52 Klauseln. Abbildung 3.7 zeigt die generierte Klauselmenge.

$[[x1 \in \{1,2,3,4\}]\ ,\ [x2 \in \{1,2,3,4\}]\ ,\ [x3 \in \{1,2,3,4\}]\ ,\ [x4 \in \{1,2,3,4\}],$
$[x1 \in \{2,3,4\}\ ,\ x2 \in \{3,4\}]\ ,\ [x1 \in \{2,3,4\}\ ,\ x3 \in \{2,4\}]\ ,\ [x1 \in \{2,3,4\}\ ,\ x4 \in \{2,3\}],$
$[x1 \in \{1,3,4\}\ ,\ x2 \in \{4\}]\ ,\ [x1 \in \{1,3,4\}\ ,\ x3 \in \{1,3\}]\ ,\ [x1 \in \{1,3,4\}\ ,\ x4 \in \{1,3,4\}],$
$[x1 \in \{1,2,4\}\ ,\ x2 \in \{1\}]\ ,\ [x1 \in \{1,2,4\}\ ,\ x3 \in \{2,4\}]\ ,\ [x1 \in \{1,2,4\}\ ,\ x4 \in \{1,2,4\}],$
$[x1 \in \{1,2,3\}\ ,\ x2 \in \{1,2\}]\ ,\ [x1 \in \{1,2,3\}\ ,\ x3 \in \{1,3\}]\ ,\ [x1 \in \{1,2,3\}\ ,\ x4 \in \{2,3\}],$
$[x2 \in \{2,3,4\}\ ,\ x1 \in \{3,4\}]\ ,\ [x2 \in \{2,3,4\}\ ,\ x3 \in \{3,4\}]\ ,\ [x2 \in \{2,3,4\}\ ,\ x4 \in \{2,4\}],$
$[x2 \in \{1,3,4\}\ ,\ x1 \in \{4\}]\ ,\ [x2 \in \{1,3,4\}\ ,\ x3 \in \{4\}]\ ,\ [x2 \in \{1,3,4\}\ ,\ x4 \in \{1,3\}],$
$[x2 \in \{1,2,4\}\ ,\ x1 \in \{1\}]\ ,\ [x2 \in \{1,2,4\}\ ,\ x3 \in \{1\}]\ ,\ [x2 \in \{1,2,4\}\ ,\ x4 \in \{2,4\}],$
$[x2 \in \{1,2,3\}\ ,\ x1 \in \{1,2\}]\ ,\ [x2 \in \{1,2,3\}\ ,\ x3 \in \{1,2\}]\ ,\ [x2 \in \{1,2,3\}\ ,\ x4 \in \{1,3\}],$
$[x3 \in \{2,3,4\}\ ,\ x1 \in \{2,4\}]\ ,\ [x3 \in \{2,3,4\}\ ,\ x2 \in \{3,4\}]\ ,\ [x3 \in \{2,3,4\}\ ,\ x4 \in \{3,4\}],$
$[x3 \in \{1,3,4\}\ ,\ x1 \in \{1,3\}]\ ,\ [x3 \in \{1,3,4\}\ ,\ x2 \in \{4\}]\ ,\ [x3 \in \{1,3,4\}\ ,\ x4 \in \{4\}],$
$[x3 \in \{1,2,4\}\ ,\ x1 \in \{2,4\}]\ ,\ [x3 \in \{1,2,4\}\ ,\ x2 \in \{1\}]\ ,\ [x3 \in \{1,2,4\}\ ,\ x4 \in \{1\}],$
$[x3 \in \{1,2,3\}\ ,\ 1x \in \{1,3\}]\ ,\ [x3 \in \{1,2,3\}\ ,\ x2 \in \{1,2\}]\ ,\ [x3 \in \{1,2,3\}\ ,\ x4 \in \{1,2\}],$
$[x4 \in \{2,3,4\}\ ,\ 1x \in \{2,3\}]\ ,\ [x4 \in \{2,3,4\}\ ,\ x2 \in \{2,4\}]\ ,\ [x4 \in \{2,3,4\}\ ,\ x3 \in \{3,4\}],$
$[x4 \in \{1,3,4\}\ ,\ 1x \in \{1,3,4\}]\ ,\ [x4 \in \{1,3,4\}\ ,\ x2 \in \{1,3\}]\ ,\ [x4 \in \{1,3,4\}\ ,\ x3 \in \{4\}],$
$[x4 \in \{1,2,4\}\ ,\ 1x \in \{1,2,4\}]\ ,\ [x4 \in \{1,2,4\}\ ,\ x2 \in \{2,4\}]\ ,\ [x4 \in \{1,2,4\}\ ,\ x3 \in \{1\}],$
$[x4 \in \{1,2,3\}\ ,\ 1x \in \{2,3\}]\ ,\ [x4 \in \{1,2,3\}\ ,\ x2 \in \{1,3\}]\ ,\ [x4 \in \{1,2,3\}\ ,\ x3 \in \{1,2\}]]$

Abbildung 3.7: Klauselmenge mit Mengen-Prädikaten

Schlussfolgerung

Für das 4-Damen Problem gibt es zwei Lösungen. (siehe Abbildung 3.8) Beide Kodierungen finden diese beiden Lösungen. Jedoch hat sich gezeigt, dass sich mit den Mengen-Prädikaten sowohl die Anzahl der Variablen, deren Vorkommen und letzlich die Anzahl der Klauseln drastisch reduzieren lassen. In Tabelle 3.2 werden die Unterschiede nochmal zusammengefasst.

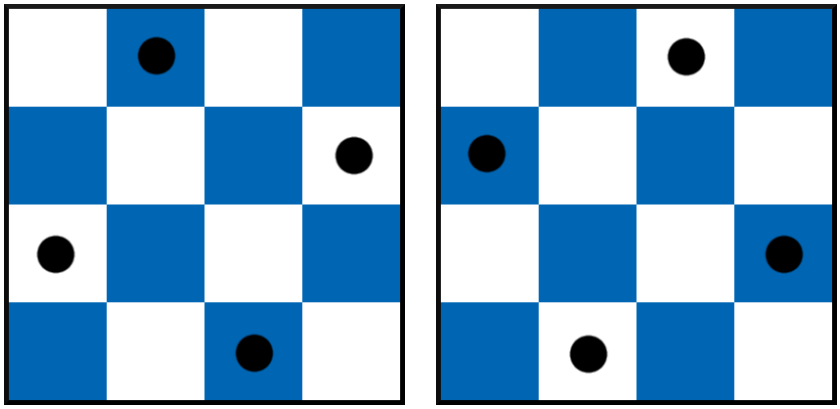


Abbildung 3.8: Lösungen des 4×4-Damen Problems

An dieser Stelle ist anzumerken, dass es diverse Verschärfungen des n-Damen Problems gibt, wie zum Beispiel das n-Superdamen Problem. Im Unterschied zum n-Damen Problem gibt es die zusätzliche Einschränkung, dass sich Damen auch mit einem Springer-Zug schlagen können. Ein Springer-Zug ist ein Zug über zwei Felder nach unten, oben, links oder rechts. Im Kapitel 5 - Analyse - wird darauf etwas detailierter, vor allem im Hinblick des Leistungsvergleiches, eingegangen.

	# Variablen	**# Vorkommen Variablen**	**# Klauseln**
allg. Kodierung	16	184	84
Mengen-Prädikate	4	100	52

Tabelle 3.2: Vergleich beider Kodierungen

3.3.2 Kalkülerweiterung

In diesem Abschnitt wird nun die zweite Erweiterung genau vorgestellt, welche neben den im letzten Teilabschnitt eingeführten Atomen der Form $x \in \mathcal{M}$ auch Kalküle der Form $(\{x_1, \ldots, x_n\} = \{1, \ldots, n\})$ verarbeiten kann. Ziel dieser Erweiterung ist ebenfalls in bestimmten Situationen Teilformeln zusammenzufassen, bzw. deren Ausdruckskraft zu verbessern. Zum besseren Verständnis wird, im Anschluss dieses Abschnitts, anhand des Logikrätsels Sudoku gezeigt, wie diese Kalküle zur Kompaktheit innerhalb einer Formel betragen können. Dabei werden die Informationen, dass jede Spalte, jede Zeile und jedes 3×3-Quadrat genau die Menge $\{1, \ldots, 9\}$ enthält, kompakt durch die Kalküle dargestellt. Die allgemeine Davis-Putnam Prozedur wurde für diese Kodierung entsprechend erweitert. Im weiteren Verlauf wird als Bezeichnung von Kalkülen auch der Begriff Mengengleichheit benutzt.

Definition 3.3.2. Sei $(\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\})$ ein Kalkül und alle Elemente der linken Menge Variablen und alle Elemente der rechten Menge Konstanten. Kalküle dieser Form sind ausschließlich 1-Klauseln und es gelten zusätzlich folgende Bedingungen:

- Die Menge rechts und links haben die gleiche Mächtigkeit.
- In der Menge links sind alle Variablen verschieden, d.h es dürfen keine Variablen doppelt vorkommen.
- In der Menge rechts sind alle Konstanten verschieden, d.h es dürfen keine Konstanten doppelt vorkommen.
- Alle Elemente beider Mengen kommen nur positiv vor.

Werden diese Bedingungen zu Beginn oder während der Prozedur verletzt, führt dies zum Abbruch und es wird eine leere Lösungsmenge zurückgegeben. Das Ergebnis entspricht der Unerfüllbarkeit der eingebenen Formel.

Definition 3.3.3. *Mengen-Prädikate $x \in \mathcal{M}$ und $|\mathcal{M}| > 1$*
Innerhalb der Klauselmenge sind Mengen-Prädikate, deren endliche Menge aus mehr als einem Element bestehen erlaubt. Diese endlichen Mengen der Mengen-Prädikate werden sukzessive durch die Anwendung der Regeln versucht zu verfeinern. D.h durch Deduktion werden Folgerungen dazu verwendet, um letztlich auf einen einzigen Wert dieser Variablen aus der mehrelementigen endlichen Menge zu schliessen.

3.3.2.1 Regeln

Als nächstes werden die Regeln, welche mit neuen Kodierung notwendig sind, erläutert. Diese unterscheiden sich grundlegend von den Regeln, die in den vorherigen Abschnitten (vgl. DPLL, Mengen-Prädikate) beschrieben wurden. Die Klauselmenge besteht nur noch aus 1-Klauseln. Diese 1-Klauseln sind entweder Atome $x \in \mathcal{M}$ oder Kalküle ($\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$). Die letztlich gleichbleibende Regel ist die Fallunterscheidung, welche, ähnlich zur allgemeinen Davis-Putnam Prozedur, dafür verwendet wird, eine unbelegte Variable, bzw. Literal mit einem Wert zu belegen. Diese Regel findet ausschließlich Anwendung, wenn keine Schlussfolgerungen durch die anderen Regeln mehr hergeleitet werden können. Die Prozedur verfolgt drei Regeln, welche im Folgenden beschrieben werden. Die Reihenfolge entspricht nicht der Anwendung im Algorithmus. Vielmehr ist sie gruppiert nach der jeweiligen Vorgehensweise. Im Kapitel 4 wird darauf nochmals Bezug genommen, indem die verwendeten Funktionen aus dem Quelltext beschrieben werden.

Regel 1 *Simplifikationen*

Auch für die Kalkülerweiterung können innerhalb der Klauselmenge Simplifikationen/Vereinfachungen durchgeführt werden. Tritt einer der unteren Fälle auf, dann wird die Klauselmenge entsprechend der Beschreibung simplifiziert.

- Besteht die Klauselmenge C zu Beginn aus einer leeren Klausel, dann führt dies zu einem Widerspruch ($\bot$) und die Klauselmenge ist unerfüllbar.
- Gibt es u.a innerhalb der Klauselmenge C eine leere Klausel, dann führt dies ebenfalls zu einem Widerspruch ($\bot$) und die Klauselmenge ist unerfüllbar.
- Sei $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ ein Kalkül und es gibt ein Mengen-Prädikat $x_1 \in c_i$, sodass c_i eine Konstante aus der Menge $\{c_1, \ldots, c_n\}$, dann kann das Kalkül dahingehend vereinfacht werden, indem die Variable x_1 und die Konstante aus der Mengengleichheit entfernt wird. Die

untere Gleichung illustriert diese Vereinfachung. Im Zähler wird die beschriebene Voraussetzung angedeutet und im Nenner die Konsequenz.

$$\frac{\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\} \ \wedge \ x_1 \in c_i}{\{x_2, \ldots, x_n\} = \{c_2, \ldots, c_n\} \setminus \{c_i\} \ \wedge \ x_1 \in c_i}$$

- Für den konträren Fall: Sei $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ ein Kalkül und es gibt ein Mengen-Prädikat $x_1 \in c$, sodass c eine Konstante, aber nicht in der rechten Menge des Kalküls enthalten ist. Dann werden diese beiden Klauseln zu $Falsch$ simplifiziert. In der unteren Gleichung gleicht der Zähler wieder der Voraussetzung und der Nenner der Konsequenz.

$$\frac{\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\} \ \wedge \ x_1 \in c}{False}$$

  Wird diese Regel zu irgendeinem Zeitpunkt angewendet, dann ist mit der aktuellen Belegung der Variablen die Unerfüllbarkeit nachgewiesen.

Die beiden zuletzt beschriebenen Vereinfachungen dürfen ausschließlich dann angewendet werden, wenn die endliche Menge desjeweiligen Mengen-Prädikat nur aus genau einer Konstante, bzw. einem Element besteht. Nach Definition 3.3.3 sind allerdings auch Atome der Form $x \in \mathcal{M}$ mit $|\mathcal{M}| > 1$ erlaubt. Wie in Teilabschnitt 3.3.1 dargelegt, entspricht einem Atom der Form $x \in \{m_1, \ldots, m_n\}$ genau dem Ausdruck in Gleichung (3.6).

$$x = m_1 \vee x = m_2 \vee \ldots \vee x = m_n \tag{3.6}$$

Auf Grund der Definition eines Kalküls und die damit verbundenen Beschränkungen (vgl. Definition 3.3.2) sind Vereinfachungen von Kalkülen nur in Verbindung mit Mengen-Prädikaten möglich, dessen endliche Menge nur eine Konstante beinhalten. In Konsequenz muss für ein vorkommendes Mengen-Prädikat $x \in \mathcal{M}$ mit $|\mathcal{M}| > 1$, die endliche Menge $\mathcal{M}$, wenn möglich, sukzessive verkleinert werden.

- Sei $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ ein Kalkül und $x_1 \in \mathcal{M}$ ein Mengen-Prädikat, mit $|\mathcal{M}| > 1$, dann wird die Menge $\mathcal{M}$ an die noch vorhandenen Möglichkeiten angepasst, indem ein Mengendurchschnitt der Menge $\mathcal{M}$ mit der rechten Seite eines Kalküls durchgeführt wird.

$$\frac{\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\} \ \wedge \ x_1 \in M}{\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\} \ \wedge \ x_1 \in (M \cap \{c_1, \ldots, c_n\})}$$

Diese Regel kann zu einem Widerspruch der Klauselmenge führen, wenn das Ergebnis dieses Mengendurchschnitt der leeren Menge ($\emptyset$) entspricht. D.h mit der aktuellen Belegung der Variablen ist die Unerfüllbarkeit nachgewiesen. Dazu wird im Abschnitt 4.3, bei der Beschreibung des verwendeten Algorithmus näher eingegangen.

Regel 2 *Mengensplitregel*

Die Regel prüft, ob es innerhalb der Klauselmenge Kalküle gibt, die in zwei oder mehrere Kalküle zerlegt werden können. Sei $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ ein Kalkül innerhalb der Klauselmenge. Zudem gibt es mehr als ein Mengen-Prädikat der Form $x_i \in M_i$, sodass eine Auswahl dieser Mengen-Prädikate $(x_1, \ldots, x_j)$ eine Teilmenge von der linken Seite der Mengengleichheit $\{x_1, \ldots, x_n\}$ ist. Ferner muss als zusätzliche Bedingung die Mächtigkeit der Mengenvereinigung aller Beteiligten M_i, genau mit der Mächtigkeit der Auswahl $x_1, \ldots, x_j$ übereinstimmen. Gleichung 3.7 fasst die Bedingungen nochmals zusammen.

$$\begin{aligned} &\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\} \ \wedge \ x_1 \in M_1 \ \wedge \ \ldots \ \wedge \ x_j \in M_j \\ &\{x_1, \ldots, x_j\} \ \subset \ \{c_1, \ldots, c_n\} \\ &M := M_1 \ \cup \ \ldots \cup \ M_j, \text{ mit } |M| = j \end{aligned} \tag{3.7}$$

Gibt es genau eine Auswahl von Mengen-Prädikaten zu einer linken Seite eines Kalkül und alle Bedingungen sind oben erfüllt, dann wir dieses Kalkül folgendermaßen in zwei Kalküle zerlegt:

$$\{x_1, \ldots, x_j\} = M \ \wedge \ (\{x_1, \ldots, x_n\} \setminus \{x_1, \ldots, x_j\} = \{c_2, \ldots, c_n\} \setminus M)$$

Bemerkung: Diese Regel wird in der Implementierung auch direkt auf alle entstandenen Kalküle angewendet, sofern die Mächtigkeit kleiner vier. (< 4) In Konsequenz wird ein Kalkül bestenfalls in mehrere Kalküle zerlegt. Des Weiteren wird ein Mengensplit der Simplifikation von Kalkülen in Regel 1 vorgezogen. Mehr dazu im Kapitel 4.3.

Abbruchkriterium: Diese Regel kann die Unerfüllbarkeit einer Klauselmenge nachweisen, indem eine Auswahl von Mengen-Prädikaten gefunden wird, sodass diese Auswahl eine Teilmenge der linken Seite eines Kalküls, aber eine für eine Zerlegung notwendige Bedingung verletzt ist. Gleichung 3.8 zeigt diesen Zusammenhang.

$$\begin{aligned} &\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\} \ \wedge \ y_1 \in M_1 \ \wedge \ \ldots \ \wedge \ y_j \in M_j \\ &\{y_1, \ldots, y_j\} \ \subset \ \{x_1, \ldots, x_n\} \\ &M := M_1 \ \cup \ \ldots \cup \ M_j, \text{ mit } |M| < j \end{aligned} \tag{3.8}$$

Regel 3 *Splitting Regel*

Wie eingangs schon beschrieben, ähnelt diese Regel denen aus dem vorhergehenden Teilabschnitten. Auch in diesem Fall darf die Splitting Regel nur angewendet werden, wenn keine andere Regel mehr angewendet werden kann. Innerhalb dieser Regel wird eine Fallunterscheidung durchgeführt. Als Erstes wird ein Kalkül aus der Klauselmenge ausgewählt. Die Auswahl dieses Kalküls erfolgt anhand einer bestimmten Heuristik, die im nächsten Teilabschnitt erläutert wird. Anschließend wird eine Fallunterscheidung folgendermaßen durchgeführt:

Sei $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ die ausgewählte Mengengleichheit. Die Anzahl der möglichen Fälle richtet sich nach der Anzahl der Konstanten in der rechten Menge der Mengengleichheit. Die Variable x_1 wird aufgezählt mit den Elementen aus der Menge $\{c_1, \ldots, c_n\}$.

$$\begin{array}{lcl} x_1 \in 1 & \wedge & \{x_2, \ldots, x_n\} = \{2, \ldots, n\} \\ x_1 \in 2 & \wedge & \{x_2, \ldots, x_n\} = (\{1, \ldots, n\} \setminus \{2\}) \\ \vdots & \vdots & \vdots \\ x_1 \in n & \wedge & \{x_2, \ldots, x_n\} = \{1, \ldots, (n-1)\} \end{array} \tag{3.9}$$

Vor der Anwendung der betreffenden Fälle wird die Klauselmenge gespeichert. Im Algorithmus werden die möglichen Fälle nacheinander betrachtet. Begin-

nend mit dem ersten Fall: Führt die Anwendung zu einem Widerspruch durch die oben beschriebenen Regeln, dann muss die Klauselmenge zurückgesetzt werden. D.h. alle Veränderungen, welche zwischenzeitlich durch die Regeln durchgeführt wurden, müssen zurückgesetzt werden. Anschließend wird der darauffolgende Fall betrachtet. Führen alle Fälle zu einem Widerpruch ist die Klauselmenge Unerfüllbar.

3.3.2.2 Heuristik der Atomauswahl

Unter Anderem wurden in Abschnitt 3.3.1.2 einige Heuristiken für eine mögliche Auswahl einer aussagenlogischen Variablen beschrieben. Für die Kalkülerweiterung wird allerdings eine einfachere Heuristik verwendet. Zuerst wird ein Kalkül $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ ausgewählt, sodass:

- die rechte Menge der Mengengleichheit die kleinste Mächtigkeit unter allen vorkommenden Mengengleichheiten aufweist.

Anschließend wird aus der linken Menge die erste Variable verwendet und wie in der Splitting-Regel beschrieben, als Mengen-Prädikat formal der Klauselmenge hinzugefügt.

3.3.2.3 Beschreibung Algorithmus

In Abbildung 3.9 ist der Ablauf des Algorithmus als Pseudocode dargestellt. Als Eingabeargumente wird eine Formel C, die festgelegte endliche Menge $\mathcal{M}$ und die Lösungsmenge L erwartet, wobei die Formel C in einer Konjunktiven Normalform (KNF) vorliegen muss. Analog zur ersten Erweiterung ist die Lösungsmenge L beim ersten Aufruf des Algorithmus leer, d.h es gibt zu Beginn keine Belegungen der Literale. Der Algorithmus ist eine rekursive Prozedur; Nachdem eine Regel angewendet wurde, wird der Algorithmus mit den veränderten Eingabeargumenten erneut rekursiv aufgerufen.
Existiert eine Interpretation, d.h eine Variablenbelegung für die eingegebene Klauselmenge, sodass diese Klauselmenge erfüllt ist, dann terminiert der angebene Algorithmus genau mit dieser Interpretation. Andernfalls terminiert der Algorithmus mit der leeren Menge ($\emptyset$), bzw. mit dem Text, dass die eingegebene Klauselmenge nicht erfüllbar ist.

Algorithm 3 $DPFS(C, \mathcal{M}, L)$

Require: Klauselmenge C ist eine Konjunktive Normalform und enthält keine Tautologien innerhalb einer Klausel. Sei $\mathcal{M}$ die festgelegte endliche Menge und L die Lösungsmenge.

if C enthält eine leere Klausel **then**
  TRUE
else
  if C ist eine leere Klauselmenge **then**
    TRUE
  end if
end if

/*Simplifikationen*/
if C enthält ein Kalkül $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ und ein Atom $x_1 \in \mathcal{M}$ mit $|\mathcal{M}| > 1$ **then**
  Simplifiziere dieses Atom nach Regel 1.
  Sei L_{neu} die veränderte Lösungsmenge nach der Simplifikation.
  $DPFS(C, \mathcal{M}, L_{neu})$
end if

if C enthält ein Kalkül $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ und ein Atom $x_1 \in \{c_1\}$ **then**
  Simplifiziere dieses Kalkül nach Regel 1.
  Sei C_{neu} die veränderte Klauselmenge nach der Simplifikation.
  $DPFS(C_{neu}, \mathcal{M}, L)$
end if

if C enthält ein Kalkül $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ und ein Atom $x_1 \in \{c\}$, $c \notin \{c_1, \ldots, c_n\}$ **then**
  Widerspruch gefunden, C nicht erfüllbar.
  TRUE
end if

/*Mengensplit*/
Sei $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ ein Kalkül und
$x_1 \in M_1 \wedge \ldots \wedge x_j \in M_j$ eine passende Auswahl, mit $M := M_1 \cup \ldots \cup M_j$
if $|M| < j$ **then**
  Widerspruch gefunden, C nicht erfüllbar.
  $L := \emptyset$
else
  if $|M| == j$ **then**
    Das Kalkül wird nach Regel 2 in ein oder mehrere Kalküle zerlegt.
    Sei C_{neu} die veränderte Klauselmenge nach dem Mengensplit.
    $DPFS(C_{neu}, \mathcal{M}, L)$
  end if
end if

/*Fallunterscheidung*/
Sei $\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\}$ der gewählte Mengenschnitt, nach der beschriebenen Heuristik und $x1$ wird als Variable angenommen

Abbildung 3.9: Pseudocode Erweiterung der DP-Prozedur um Mengengleichheiten

Beispiel 3.3.2. Sudoku Problem

Zum Abschluss dieses Kapitels der Beschreibung der Kalkülerweiterung wird zum besseren Verständnis ein Beispiel betrachtet. Als Beispiel dient das Sudoku-Problem, welches im weiteren Verlauf näher beschrieben wird.
Sudoku ist ein Logikrätsel und besteht in der ursprünglichen Variante aus einem 9×9-Quadrat, wie in Abbildung 3.10 zu sehen ist. Das 9×9-Quadrat ist wiederum in neun 3×3-Subquadrate[32] geteilt. Insgesamt gibt es 81 Felder. Ziel ist es, diese 81 Felder auszufüllen, sodass in jeder Reihe, in jeder Spalte und innerhalb jeder der neun Regionen die Zahlen 1 bis 9 jeweils nur einmal vorkommen.

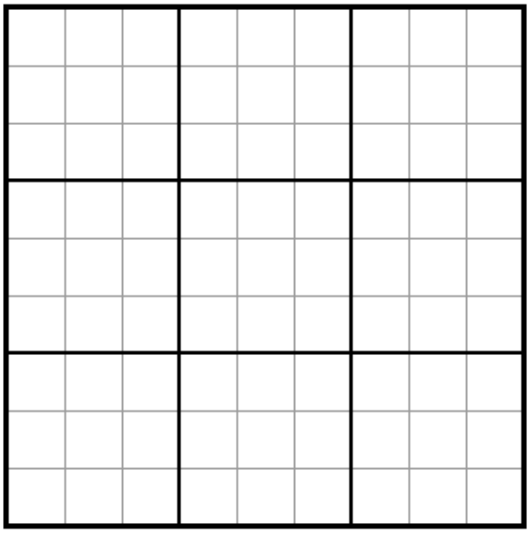


Abbildung 3.10: leeres Sudoku 9×9-Quadrat

Es gibt zusätzlich zum Standard-Sudoku diverse Varianten, wie zum Beispiel Hypersudoku, Nonominosudoku oder XSudoku. Diese drei Varianten werden unter anderem im Kapitel 5 - Analyse - behandelt, indem sie als entsprechende Formel dem Algorithmus übergeben werden.
Im nächsten Schritt wird die Kodierung für die allgemeine Davis-Putnam Prozedur und für die erste Erweiterung um Mengen-Prädikate beschrieben und die resultierende Klauselmenge im Anschluss der Klauselmenge mit der Kodierung mittels Kalkülen gegenübergestellt.

Allgemeine Kodierung

[32] auch als Regionen bezeichnet

In der Aussagenlogik kann jede Variable mit den Werten 1 ($Wahr$) oder 0 ($Falsch$) belegt werden. In Sudoku besitzt jedes Feld einen Wert zwischen 1 und 9; Aus diesem Grund werden aussagenlogische Variablen als 3-Tupel modelliert. Ein 3-Tupel repräsentiert ein Feld, bestehend aus der Reihe, der Spalte und dem Wert. (r, s, w) Eine Variable wird mit 1 ($Wahr$) belegt, genau dann wenn in der jeweiligen Zeile und Spalte der Wert w steht. Demnach gibt es für jedes der 81 Felder 9 Variablen, sodass insgesamt 729 aussagenlogische Variablen benötigt werden. Infolge dessen wird an dieser Stelle nur informell auf die Modellierung der Beschränkungen eingegangen und auf die detailiertere Beschreibung in [KJ06] verwiesen.

Beschränkungen 1: Felder

Die folgenden Beschränkungen werden exemplarisch am ersten Feld $(1, 1, w)$ gezeigt und sind für alle 81 Felder zu wiederholen:

- Jedes Feld muss genau einen Wert zwischen 1 und 9 besitzen. Diese Beschränkung wird durch die Veroderung ($\vee$) der aussagenlogischen Variablen (r, s, w_i) realisiert, wobei $i \in \{1..9\}$.

  $(1,1,1) \vee (1,1,2) \vee (1,1,3) \vee (1,1,4) \vee (1,1,5) \vee (1,1,6) \vee (1,1,7) \vee (1,1,8) \vee (1,1,9)$

- Jedes Feld darf nur einen einzigen Wert besitzen. Dazu muss für jeden möglichen Wert eines Feldes die anderen Werte durch Implikationen ausgeschlossen werden.

  $$\begin{array}{lcl} (1,1,1) & \Longrightarrow & \neg(1,1,2) \\ (1,1,1) & \Longrightarrow & \neg(1,1,3) \\ \vdots & \vdots & \vdots \\ (1,1,1) & \Longrightarrow & \neg(1,1,9) \end{array}$$

Beschränkungen 2: Zeilen

Die folgenden Beschränkungen werden exemplarisch an der ersten Zeile und mit dem Wert 1 gezeigt und sind für alle anderen Werte von 2 bis 9 zu wiederholen:

- In jeder Zeile müssen die Werte 1 bis 9 vorkommen. Diese Beschränkung wird durch die Veroderung ($\vee$) der betreffenden aussagenlogischen Variablen realisiert.

$$(1,1,1) \lor (1,2,1) \lor (1,3,1) \lor (1,4,1) \lor (1,5,1) \lor (1,6,1) \lor (1,7,1) \lor (1,8,1) \lor (1,9,1)$$

- In jeder Zeile kommen die Werte 1 bis 9 genau einmal vor.

$$\begin{array}{lll} (1,1,1) & \Longrightarrow & \neg(1,2,1) \\ (1,1,1) & \Longrightarrow & \neg(1,3,1) \\ \vdots & \vdots & \vdots \\ (1,1,1) & \Longrightarrow & \neg(1,9,1) \end{array}$$

Beschränkungen 3: Spalten

Analog zur Zeilenbeschränkung werden diese für die Spalten examplarisch an der ersten Spalte mit dem Wert 1 gezeigt und sind ebenfalls für alle anderen Werte von 2 bis 9 zu wiederholen:

- In jeder Zeile müssen die Werte 1 bis 9 vorkommen. Diese Beschränkung wird durch die Veroderung ($\lor$) der betreffenden aussagenlogischen Variablen realisiert.

$$(1,1,1) \lor (2,1,1) \lor (3,1,1) \lor (4,1,1) \lor (5,1,1) \lor (6,1,1) \lor (7,1,1) \lor (8,1,1) \lor (9,1,1)$$

- In jeder Spalte kommen die Werte 1 bis 9 genau einmal vor.

$$\begin{array}{lll} (1,1,1) & \Longrightarrow & \neg(2,1,1) \\ (1,1,1) & \Longrightarrow & \neg(3,1,1) \\ \vdots & \vdots & \vdots \\ (1,1,1) & \Longrightarrow & \neg(9,1,1) \end{array}$$

Beschränkungen 4: Regionen

Für die Regionen werden die oberen Beschränkungen analog definiert, sodass augenblicklich darauf verzichtet wird.

Werden die Beschränkungen anhand der Beschreibungen für alle Fälle durchgeführt, führt dies zu einer Klauselmenge, ohne Vorbelegungen von etwa 12000 Klauseln. Der überwiegende Teil der Klauseln enthalten zwei aussagenlogische Variablen.

Kodierung mit Mengen-Prädikaten

In diesem Fall bestehen die Klauseln aus Atomen der Form $x \in \mathcal{M}$ und im Gegensatz zur allgemeinen Kodierung entspricht jedem Feld eine aussagenlogische

Variable. Bei einem 9×9-Quadrat gibt es somit 81 Variablen. Für ein Atom $x_1 \in \mathcal{M}$, sei x_1 die Variable, welches dem untersten linken Feld entspricht und $\mathcal{M}$ die Menge der möglichen Werte für dieses Feld.

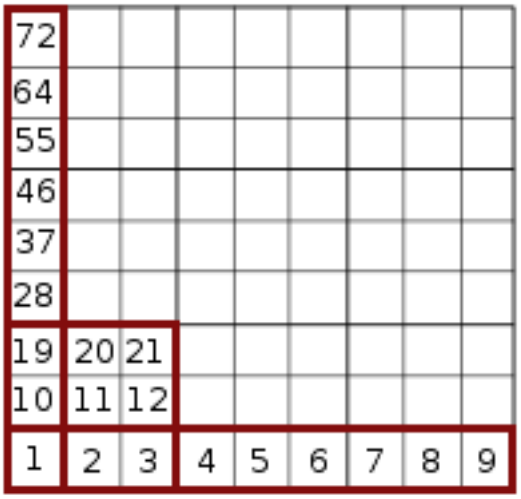


Abbildung 3.11: Sudoku 9×9-Quadrat Modellierung

Annahme: Wie in Abbildung 3.11 zu sehen, werden die Beschränkungen für die erste Zeile, die erste Spalte, sowie für die erste Region erläutert. Zum besseren Verständnis, dass es sich bei den Nummern der Felder um Variablen handelt, wird das Präfix „x“ vorangehängt.

- Beschränkung Zeile:
  In jeder Zeile dürfen die Zahlen 1 bis 9 nur einmal vorkommen. Wenn zum Beispiel das Feld x_1 den Wert 1 annimmt, dann führt dies zu folgenden Klauseln:

  $$\begin{aligned} x_1 \in \{1\} &\implies x_2 \in \{2..9\} \\ x_1 \in \{1\} &\implies x_3 \in \{2..9\} \\ \vdots \quad &\quad \vdots \quad \vdots \\ x_1 \in \{1\} &\implies x_9 \in \{2..9\} \end{aligned}$$

  Diese Vorgehensweise muss für alle Felder x_i in der Zeile und für alle möglichen Werte wiederholt werden.

- Beschränkung Spalten:
  In jeder Spalte dürfen die Zahlen 1 bis 9 nur einmal vorkommen. Wenn zum Beispiel das Feld $x1$ den Wert 1 annimmt, dann führt dies zu folgenden Klauseln:

$$
\begin{array}{lcl}
x_1 \in \{1\} & \Longrightarrow & x_10 \in \{2..9\} \\
x_1 \in \{1\} & \Longrightarrow & x_19 \in \{2..9\} \\
\vdots & \vdots & \vdots \\
x_1 \in \{1\} & \Longrightarrow & x_72 \in \{2..9\}
\end{array}
$$

Diese Vorgehensweise muss für alle Felder x_i in der Spalte und für alle möglichen Werte wiederholt werden.

- Beschränkung Regionen:
  Für die Regionen werden die oberen Beschränkungen analog definiert, sodass auch in diesem Fall darauf verzichtet wird.

Für die Generierung müssen die eben beschriebenen Beschränkungen für jeden Fall durchgeführt werden. Nach Optimierungen und Umformungen in eine Konjunktive Normalform ergeben sich in etwa 13446 Klauseln.

Kodierung mit Kalkülen

An dieser Stelle wird nochmals darauf hingewiesen, dass die Klauselmenge nur aus 1-Klauseln bestehen darf, welche wiederum aus Mengen-Prädikaten oder Kalkülen besteht. Aufbauend auf der Kodierung mit Mengen-Prädikaten wird jedem Feld des 9×9-Quadrates, fortlaufend von links nach rechts, eine aussagenlogische Variable zugeordnet, insgesamt 81 Variablen.
Aber im Gegensatz zu den zuvor betrachteten Kodierungen werden für das Sudoku Problem die Beschränkungen der Spalten, Zeilen und der Regionen jeweils durch Mengengleichheiten beschrieben. Eventuell zu Beginn bereits ausgefüllte Felder werden der Klauselmenge als Atome der Form $x \in M$ hinzugefügt.
Die linke Seite einer Mengengleichheit modelliert die aussagenlogischen Variablen, die den Feldern entsprechen und die rechte Seite den möglichen Werten die für die Felder angenommen werden können. Durch die Beschränkungen in Definition 3.3.2 und den definierten Regeln ist sichergestellt, dass keine zwei aussagenlogische Variablen auf der linken Seite der Mengengleichheit denselben Wert annehmen können.

Annahme: Wie in Abbildung 3.11 zu sehen, werden die Beschränkungen für die erste Zeile und Spalte, sowie für die Region 1 erläutert. Zum besseren Verständnis, dass es sich bei den Nummern der Felder um Variablen handelt, wird das Präfix „x“ vorangehängt.

- Beschränkung Zeile:
  In jeder Zeile dürfen die Zahlen 1 bis 9 nur einmal vorkommen.

$$\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

- Beschränkung Spalten:
  In jeder Spalte dürfen die Zahlen 1 bis 9 nur einmal vorkommen.

$$\{x_1, x_{10}, x_{19}, x_{28}, x_{37}, x_{46}, x_{55}, x_{64}, x_{72}\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

- Beschränkung Regionen:
  In jeder 3×3-Region dürfen die Zahlen 1 bis 9 nur einmal vorkommen.

$$\{x_1, x_2, x_3, x_{10}, x_{11}, x_{12}, x_{19}, x_{20}, x_{21}\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Für die Generierung der kompletten Klauselmenge muss diese Vorgehensweise für jede Zeile, jede Spalte und für alle neun Regionen durchgeführt werden. Schließlich besteht die Klauselmenge aus 27 Klauseln, bzw. Kalkülen. Abbildung 3.12 zeigt die generierte Klauselmenge.
Für eventuelle Vorbelegungen der Felder mit Werten, müssen diese Felder als Mengen-Prädikate der generierten Klauselmenge angehängt werden. Soll zum Beispiel auf Feld 1 eine 3 als Vorbelegung stehen, dann muss der generierten Klauselmenge als 1-Klausel ein Mengen-Prädikat $[x3 \in \{3\}]$ angehängt werden.

Schlussfolgerung

Das Sudoku Problem lässt sich mit den drei Kodierungen modellieren und lösen. Bei der Erweiterung um Mengen-Prädikaten lässt sich die Kodierung der Information, dass eine Zeile, Spalte und die neun Regionen genau die Menge $\{1, .., 9\}$ enthält, nur sehr umständlich modellieren. Auch führte die Kodierung in diesem Fall zu keiner Verringerung der Klauselanzahl im Vergleich zur allgemeinen aussagenlogischen Kodierung.
Die abschließend vorgestellte Kodierung mit Kalkülen sticht nicht nur durch die kompakte und aussagekräftige Darstellung der Information heraus, auch ist insgesamt die Klauselanzahl sehr gering. Im Kapitel 5 - Analyse - werden auch

$$
\begin{aligned}
&[[\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{19}, x_{20}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{27}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{28}, x_{29}, x_{30}, x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{37}, x_{38}, x_{39}, x_{40}, x_{41}, x_{42}, x_{43}, x_{44}, x_{45}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{46}, x_{47}, x_{48}, x_{49}, x_{50}, x_{51}, x_{52}, x_{53}, x_{54}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{55}, x_{56}, x_{57}, x_{58}, x_{59}, x_{60}, x_{61}, x_{62}, x_{63}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{64}, x_{65}, x_{66}, x_{67}, x_{68}, x_{69}, x_{70}, x_{71}, x_{72}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{73}, x_{74}, x_{75}, x_{76}, x_{77}, x_{78}, x_{79}, x_{80}, x_{81}\} = \{1,2,3,4,5,6,7,8,9\}],
\end{aligned}
$$

$$
\begin{aligned}
&[\{x_1, x_{10}, x_{19}, x_{28}, x_{37}, x_{46}, x_{55}, x_{64}, x_{72}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{19}, x_{20}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{27}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{28}, x_{29}, x_{30}, x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{37}, x_{38}, x_{39}, x_{40}, x_{41}, x_{42}, x_{43}, x_{44}, x_{45}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{46}, x_{47}, x_{48}, x_{49}, x_{50}, x_{51}, x_{52}, x_{53}, x_{54}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{55}, x_{56}, x_{57}, x_{58}, x_{59}, x_{60}, x_{61}, x_{62}, x_{63}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{64}, x_{65}, x_{66}, x_{67}, x_{68}, x_{69}, x_{70}, x_{71}, x_{72}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{73}, x_{74}, x_{75}, x_{76}, x_{77}, x_{78}, x_{79}, x_{80}, x_{81}\} = \{1,2,3,4,5,6,7,8,9\}],
\end{aligned}
$$

$$
\begin{aligned}
&[\{x_1, x_2, x_3, x_{10}, x_{11}, x_{12}, x_{19}, x_{20}, x_{21}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_4, x_5, x_6, x_{13}, x_{14}, x_{15}, x_{22}, x_{23}, x_{24}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_7, x_8, x_9, x_{16}, x_{17}, x_{18}, x_{25}, x_{26}, x_{27}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{28}, x_{29}, x_{30}, x_{37}, x_{38}, x_{39}, x_{46}, x_{47}, x_{48}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{31}, x_{32}, x_{33}, x_{40}, x_{41}, x_{42}, x_{49}, x_{50}, x_{51}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{34}, x_{35}, x_{36}, x_{43}, x_{44}, x_{45}, x_{52}, x_{53}, x_{54}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{55}, x_{56}, x_{57}, x_{64}, x_{65}, x_{66}, x_{73}, x_{74}, x_{75}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{58}, x_{59}, x_{60}, x_{67}, x_{68}, x_{69}, x_{76}, x_{77}, x_{78}\} = \{1,2,3,4,5,6,7,8,9\}],\\
&[\{x_{61}, x_{62}, x_{63}, x_{70}, x_{71}, x_{72}, x_{79}, x_{80}, x_{81}\} = \{1,2,3,4,5,6,7,8,9\}]]
\end{aligned}
$$

Abbildung 3.12: Klauselmenge mit der Kalkülkodierung

noch andere Varianten von Sudoku vor allem hinsichtlich der Geschwindigkeit der Lösungssuche betrachtet.

4 Implementierung

Im diesem Abschnitt wird die Implementierung des Software-Programms beschrieben, sodass die, im vorherigen Kapitel vorgestellten, Erweiterungen des Davis-Putnam Prozedur verarbeitet werden können. Es wird sich im Folgenden auf die besonders interessanten Programmteile beschränkt und ferner weitgehend auf die Implementierungsdetails verzichtet. Anderenfalls wird auf dem dokumentierten Quellcode verwiesen. Als Programmiersprache wird die im Kapitel 2 (vgl. 2.2.3) vorgestellte Programmiersprache Haskell verwendet.

```
Der Quelltext kann unter der folgenden Adresse runtergeladen werden:
    http://www.ki.informatik.uni-frankfurt.de/diplom/programme/ilgner
```

Das Programm aus insgesamt aus vier Modulen:

- Dpfs.DavisPutnamFiniteSets
- Dpfs.Parser
- Dpfs.SimpCnf
- Dpfs.Simp

Wie bereits im letzten Abschnitt angesprochen wird im Wesentlichen für beide Erweiterungen das gleiche Programm verwendet. Die Entscheidung, welche Erweiterung letztlich gewählt werden soll, geschieht über einen Parameter beim Aufruf des Programms.

Programmaufruf
Die Funktion *dpfs* ist die zentrale Funktion innerhalb der Implementierung. Sie ruft alle notwendigen Programmteile auf, welche im weiterem Verlauf beschrieben werden.

```
dpfs :: Int → Int → [Char] → [Char]
dpfs sel fset expr =
```

Die Funktion dpfs hat drei Argumente: *sel*, $fset$ und *expr*. Das erste Argument *sel* ist der angesprochene Selektor für die Auswahl der Erweiterung. Der Wertebereich ist $sel = \{0, 1\}$, wobei mit 0 die erste Erweiterung und mit 1 die zweite Erweiterung ausgewählt wird. Hingegen ist das zweite Argument die festgelegte endliche Menge $\mathcal{M}$; die Funktion erwartet hierfür einen Int-Wert. Dieser Int-Wert entspricht dem maximalem Element der Menge. D.h für einen Wert n wird die Menge $\{1, \ldots, n\}$ angenommen. Das letzte Argument *expr* entspricht der aussagenlogischen Eingabeformel.

Das Programm besteht im Wesentlichen aus drei ineinander übergehenden Teilen:

1. Lexikalische Analyse und Parsen der Eingabeformel
2. Umwandlung der Formel in eine Konjunktive Normalform (KNF)
3. Test auf (Un-)Erfüllbarkeit

Diese drei Programmteile beziehen sich im Ganzen lediglich auf die erste Erweiterung um Mengen-Prädikate.[vgl. 3.3.1] Für die Kalkülerweiterung muss eine Eingabeformel direkt als Klauselmenge übergeben werden. Auf die Syntax einer Klauselmenge wird im weiterem Verlauf genauer eingegangen.
Die ersten beiden Programmteile transformieren die Eingabeformel in eine Klauselmenge. Zur Erinnerung: Eine Klauselmenge ist die Konjunktion von Klauseln und Klauseln entsprechen einer Disjunktion von Atomen der Form $x \in \mathcal{M}$. In der Implementierung ist eine Klauselmenge eine Liste von Listen, welche den Klauseln entsprechen. Im weiteren Verlauf wird darauf noch detailierter eingegangen. Bevor die drei Teile nacheinander beschrieben werden, wird kurz auf die verwendete Syntax der Eingabeformel eingegangen.

Syntax
Als Eingabeformel soll eine aussagenlogische Formel übergeben werden. Es werden alle erlaubten Schlüsselworte, Operatoren und Konstanten verwendet. (vgl. Kapitel 2.1 Aussagenlogik)

Überblick:

Schlüsselworte: $\langle \in \rangle$

Operatoren: $\langle \neg,\ \wedge,\ \vee,\ \Rightarrow,\ \Leftrightarrow,\ ),\ (\rangle$

Konstanten: 0, 1

Beschreibung der Syntax:
Die erste Erweiterung um Mengen-Prädikate macht es notwendig die Syntax entsprechend zu erweitern. Sei $x \in M$ ein Atom, wobei $M = \{m_1, \ldots, m_n\}$.

- Aussagenlogische Variablen sind entweder Buchstaben oder Worte
- Das Elementsymbol innerhalb eines Atoms wird dargestellt als $ELEM$
- Die Elemente der Menge M werden in geschweiften Klammern, durch Kommata getrennt, dargestellt. $\{m_1, \ldots, m_n\}$
- Die Negation (logisch Nicht) wird dargestellt als einfaches Minuszeichen $(-)$
- Die Konjunktion (logisch Und) wird dargestellt als /\\
- Die Disjunktion (logisch Oder) wird dargestellt als \\/
- Die Implikation $(\Rightarrow)$ wird dargestellt als =>
- Die Äquivalenz $(\Leftrightarrow)$ wird dargestellt als <=>
- Die Konstanten 0 und 1 werden für Falsch (False) bzw. Wahr (True) verwendet
- Ausdücke können mit runden Klammern gruppiert werden

Beispiel 4.0.3. Die Formel $x1 \in \{1, 2, 3\} \Rightarrow x2 \in \{3, 4\}$ muss nun folgendermaßen eingegeben werden:

$$x1\ ELEM\ \{1, 2, 3\}\ =>\ x2\ ELEM\ \{3, 4\}$$

Bemerkung:
Die oben genannten Programmteile können auch unabhängig voneinander aufgerufen werden. D.h. wenn eine Eingabeformel bereits in einer Klauselmenge vorliegt, können die ersten beiden Programmteile übersprungen werden. In diesem Fall muss diese Eingabeformel direkt als Klauselmenge[33] dem letzten Programmteil übergeben werden. Die im Folgendem beschriebene Syntax der Klauselmenge wird nur formal angegeben, d.h unabhängig vom verwendeten Datentyp für die Atome. An später̈er Stelle wird auf dem verwendeten Datentyp eingegangen und somit wird die Syntax vervollständigt.

[33] Eine Klauselmenge repräsentiert eine Konjunktive Normalform.

Syntax Klauselmenge

- Atome der Form $x \in \mathcal{M}$ sind innerhalb der Klauseln durch ein Komma getrennt und mit eckigen Klammern umschlossen
- Klauseln werden durch Kommas getrennt dargestellt und ebenfalls mit eckigen Klammern umschlossen
- Literale dürfen nur aus Zahlen bestehen
- Negative Literale sind nicht erlaubt. Diese müssen vorher umgeformt werden. (siehe Lemma 3.3.1)

Der erste Punkt bezieht sich ausschließlich für die erste Erweiterung um endliche Mengen-Prädikate. Bei der zweiten Erweiterung darf es dagegen nur 1-Klauseln, d.h Klauseln die nur aus einem Atom oder Kalkül bestehen, geben.

4.1 Lexikalische Analyse und Parser

Die als Eingabe übergebene Formel muss eine syntaktisch korrekte aussagenlogische Formel sein. Auf Grund dessen muss die Eingabeformel durch einen Parser auf Korrektheit überprüft werden. Der Parser wird mit dem Parsergenerator Happy[34] generiert. Dazu ist es erforderlich, dass die Eingabeformel durch eine lexikalische Analyse in sogenannte Tokens zerlegt und im Anschluss anhand einer Kontextfreien Grammatik (KfG) auf syntaktische Korrektheit überprüft wird. Token stellen in diesem Zusammenhang Schlüsselworte, Operatoren und Konstanten dar. Listing (4.1) zeigt den benutzten Datentyp, sowie alle möglichen Tokens.

```
data LexTok = TokVar String
            | TokParL
            | TokParR
            | TokColon
            | TokComma
            | TokNot
            | TokTru
            | TokFal
```

[34] mehr Informationen auf http://www.haskell.org/happy

```
            | TokAnd
            | TokOr
            | TokImpl
            | TokEqiv
            | TokElem
            | TokBraceL
            | TokBraceR
            | TokSElem String
            deriving Eq
```

Listing 4.1: Datentyp aller möglichen Tokens

Die Funktion *lexProp* zerlegt die Eingabeformel, anhand von erkannten Schlüsselworten, in Tokens. Die Datenstruktur (Datentyp *LexTok*) ist letztlich ein Strom/Liste von Tokens, wobei jedes Token mit einer Spalte und Zeile markiert ist. Diese Markierung entspricht genau der Position in der ursprünglichen Eingabeformel und dient zur exakten Fehlerangabe eines eventuell unbekannt eingelesenen Symbols. Listing (4.2) zeigt die Lexer Funktion *lexProp* in verkürzter Form.

```
1  type Col   = Int
2  type Row   = Int
3  type Token = (LexTok, Row, Col)
4
5  lexProp :: Row → Col → [Char] → [Token]
6  lexProp row col "" = []
7  lexProp row col ('1' : cs)
8    = (TokTru, row, col) : (lexProp row (col + 1) cs)
9  ..
10 ..
11 lexProp row col ('E' : 'L'  : 'E' : 'M' : cs)
12   = (TokElem,   row, col) : (lexProp row (col + 4) cs)
```

Listing 4.2: Programmauszug Lexer

Die Eingabeformel wird Zeichen für Zeichen eingelesen; Wird zum Beispiel eine 1 als nächstes gelesen, dann wir die Funktion rekursiv mit der restlichen Eingabeformel (ohne das aktuell gelesene Zeichen), sowie mit angepassten Zeilen- und Spaltenangaben wieder aufgerufen. Das erkannte Token, in diesem Fall

TokTru, wird dem Tokenstrom hinzugefügt. Zur detaillierteren Darstellung wird in diesem Fall auf dem dokumentierten Quellcode verwiesen.

Beschreibung der Happy Grammatik-Datei

Im weiterem Verlauf wird die Grammatik-Datei beschrieben, welche die kontextfreie Grammatik angibt und schließlich den Parser zusammen mit dem Lexer *lexProp* generiert. Für weitere Erläuterung wird auf die ausführliche Dokumentation des Parsergenerators *happy* hingewiesen.[MG01]

Zunächst muss der Name der Parserfunktion und den als Eingabe verwendete Datentyp deklariert werden. In diesem Fall ist der Name der Funktion *prop* und der Datentyp *Token*.

```
%name prop Expr
%tokentype { Token }
%error { happyError }

%token
    selem  { (TokSElem $$, _, _)}
    all    { (TokAll,    _, _) }
    exists { (TokExists, _, _) }
    lpar   { (TokParL,   _, _) }
    rpar   { (TokParR,   _, _) }
    ':'    { (TokColon,  _, _) }
    ','    { (TokComma,  _, _) }
    not    { (TokNot,    _, _) }
    and    { (TokAnd,    _, _) }
    or     { (TokOr,     _, _) }
    impl   { (TokImpl,   _, _) }
    eqiv   { (TokEqiv,   _, _) }
    elem   { (TokElem,    _, _) }
    lbrace { (TokBraceL,  _, _) }
    rbrace { (TokBraceR,  _, _) }
    true   { (TokTru,    _, _) }
    false  { (TokFal,    _, _) }
    var    { (TokVar $$, _, _) }
```

In der letzten Zeile wird mit dem *%error* Deklarativ die Funktion zum Abfangen von Fehlern deklariert. Anschließend wird mit dem nachfolgenden Deklarativ *%token* alle möglichen Tokens deklariert: In der linken Spalte entsprechen die Symbole den Terminalen in der Kontextfreien Grammatik und in der rechten Spalte genau dem Haskell Pattern Matchings. (genau den Datentyp $(LexTok, Row, Col)$) Da in diesen Zusammenhang die Zeilen- und Spaltenangaben keine Rolle spielen, sind diese mit einem Unterstrich angegeben. Der Unterstrich ist in der Programmiersprache *Haskell* eine Möglichkeit anzudeuten, dass bestimmte Teile nicht von Bedeutung sind.[35]
Sofern ein Token selbst nicht einem Wert entspricht, ist das doppelte $$-Zeichen ein Platzhalter für den Wert desjeweiligen Tokens. Zum Beispiel für das Token, welches eine Variable darstellt: In diesem Fall wird der Wert einer zusätzlichen Typvariable gegeordnet.

Im nächsten Schritt werden die Produktionen der kontextfreien Grammatik (KfG) betrachtet. Formal ist eine kontextfreie Grammatik eine Grammatik, die ausschließlich Produktionen bzw. Ersetzungsregeln enthält, welche auf der linken Seite Nichtterminalsymbole[36] besitzen und rechts eine beliebige Folge von Nichtterminalsymbolen und/oder Terminalsymbolen[37].

```
Expr
 : not Expr            { Pnot $2 }
 | Expr and  Expr      { Pand [$1, $3] }
 | Expr or   Expr      { Por  [$1, $3] }
 | Expr impl Expr      { Pimpl $1 $3 }
 | Expr eqiv Expr      { Pequiv $1 $3 }
 | var elem Expr1      { PvarElem (Pvar $1, $3) }
 | lpar Expr rpar      { $2 }
 | true                { Ptrue }
 | false               { Pfalse }

Expr1
 : lbrace Expr1 rbrace { $2 }
 | selem ',' Expr1     { PSelem [Pselem $1, $3] }
 | selem         { Pselem $1 }
```

[35] Diese Teile werden mit einem generischen Typ versehen, dem Unterstrich.
[36] Ein Nichtterminalsymbol können durch durch Produktionen weiter ersetzt werden
[37] Ein Terminalsymbol ist ein Symbole, welches nicht durch eine Produktion ersetzt werden kann

Die dargestellte kontextfreie Grammatik enthält die Nichtterminalsymbole *Expr*, *Expr*1 und als Terminalsymbole alle möglichen Tokens, die unter dem Deklarativ *%token* deklariert wurden. Jede Produktion ist mit einem zugehörigem Haskellfragment verbunden, dargestellt in geschweiften Klammern am Ende jeder Produktion. Das Startsymbol ist *Expr*.

Der generierte Parser vergleicht nun jedes Token mit den rechten Seiten aller definierten Produktionen. Findet der Parser eine passende Sequenz von Nichtterminal- und/oder Terminalsymbolen, konstruiert er das Nichtterminalsymbol auf der linken Seite dieser Regel und ersetzt den Wert mit dem Haskellfragment. Diese Vorgehensweise führt der Parser solange durch bis nur noch ein Symbol übrig bleibt, das Startsymbol. Im Anschluss gibt der Parser den geparsten Syntaxbaum als Haskell-Liste zurück, andernfalls wird ein Fehler mit genauer Positionsangabe ausgegeben. Der Datentyp des geparsten Ausdrucks ist Listing 4.3 in dargestellt.
Letztlich wurden der Lexer der die Eingabeformel in Tokens zerlegt, die nötige Grammatik und alle Datentypen deklariert. Somit kann der Parser mit dem Parsergenerator happy generiert werden. Das ausführlich dokumentierte Modul hat dem Namen Parser.hs und liegt im Quellcode bereit.

```
data Pexpr name = Ptrue
                | Pfalse
                | Pvar name
                | Pselem name
                | PSelem [(Pexpr name)]
                | PvarElem ((Pexpr name),(Pexpr name))
                | Pnot  (Pexpr name)
                | Pand [(Pexpr name)]
                | Por  [(Pexpr name)]
                | Pimpl  (Pexpr name) (Pexpr name)
                | Pequiv (Pexpr name) (Pexpr name)
```

Listing 4.3: Datentyp der geparsten Formel

Beispiel 4.1.1. Funktionsweise Parser
Die Eingabeformel $(x \in \{1\}) => (y \in \{2,3\})$ soll durch den oben beschriebenen Parser verarbeitet werden. Zunächst wird im ersten Schritt die Formel

in Tokens zerlegt: Das Ergebnis ist eine Haskell-Liste von erkannten Token. Zur Erinnerung: Ein Token ist deklariert als drei-Tupel, bestehend aus seiner Bezeichnung/Wert, die Zeile und die Spalte in der Eingabeformel. (vgl. Zeile 3 in Listing 4.2) Im zweiten Schritt wird die Liste von Tokens auf syntaktische Korrektheit hin überprüft. Abbildung 4.1 zeigt den generierten Syntaxbaum zur Eingabeformel.

Ausgabe der Funktionen:

Schritt 1: Lexikalische Analyse

```
% lexProp 1 1 "(x ELEM {1})=>(y ELEM {2,3})"
[((,1,1),(VARx,1,2),(EPS,1,3),({,1,8),(SElem1,1,9),(},1,9),(),1,10),
(=>,1,11),((,1,13),(VARy,1,14),(EPS,1,15),({,1,20),(SElem2,1,21),
(,,1,21),(SElem3,1,22),(},1,22),(),1,23)]
```

Schritt 2: Syntaktische Analyse

```
% prop $ lexProp 1 1 "(x ELEM {1})=>(y ELEM {2,3})"
Pimpl (PvarElem (Pvar "x",Pselem "1"))
  (PvarElem (Pvar "y",PSelem [Pselem "2",Pselem "3"]))
```

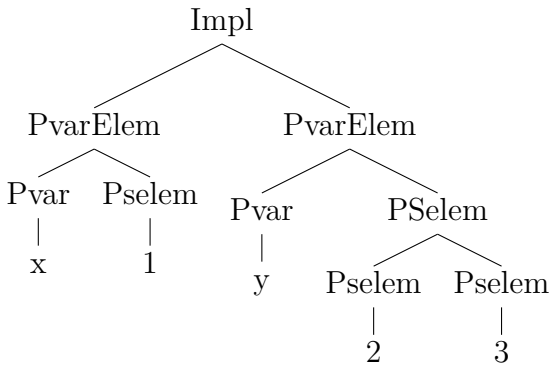


Abbildung 4.1: Syntaxbaum der Formel aus Beispiel 4.1.1

4.2 Transformation in eine Konjunktive Normalform

Nachdem die Eingabeformel im vorherigen Abschnitt auf ihre syntaktische Korrektheit überprüft wurde, liegt diese möglicherweise nicht in einer Konjunktiven Normalform vor. (vgl. Beispiel 4.1.1) Für die zentrale Prozedur bzw.

der Test auf (Un-)Erfüllbarkeit muss eine Eingabeformel in eine Konjunktive Normalform transformiert werden.
Die Transformation umfasst folgende Schritte:

- Variablen innerhalb der Formel werden durch disjunkte Zahlen ersetzen
- Äquivalenzen auflösen
- Implikationen auflösen
- logisch Nicht[38] auflösen
- logisch Wahr/Falsch eliminieren
- Tautologien erkennen und ggf. löschen

Zum besseren Umgang mit den Variablen werden diese durch eine Zahlendarstellung vom Typ Int ersetzt. Die Funktion $numPropSubs$ ersetzt alle Variablen durch Zahlen, wobei identische Variablen immer mit denselben numerischen Wert ersetzt werden. Als Argumente erwartet die Funktion $numPropSubs$ zum einem die festgelegte endliche Menge $\mathcal{M}$ und zum anderem die geparste Formel vom Typ $Pexpr$. (vgl. Listing 4.3) Die Funktion überprüft nun für jede gefundene Variable, ob es bereits eine Zuordnung dieser Variablen mit einem numerischen Wert gibt. Falls ja, dann wird diese Variable anhand der Zuordnung ersetzt. Gibt es noch keine Zuordnung wird dieser Variablen durch eine unbenutzte Zahl ersetzt. Die Funktion liefert als Rückgabewert ein Tupel aus der Liste von Zuordnungen der Variablen zu den jeweiligen Zahlen und die geänderte Formel.

Ebenfalls zur besseren Verarbeitung wird in der oberen Funktion der Datentyp $PvarElem$ zu einem $Pvar$ (x, y) vereinfacht, währen x die Zahlendarstellung der Variable und y die endliche Menge als Listendarstellung entspricht.

An dieser Stelle wird nur kurz auf die weiteren Transformationschritte eingegangen und nachfolgend auf den dokumentierten Quellcode verwiesen. Zum Thema Konjunktive Normalform (kurz KNF) wurde bereits in Kapitel 3.2 eine umfassende Einführung gegeben. Die Eingabeformel wird sukzessive auf Äquivalenzen, Implikationen und logisch Nicht überprüft und gegebenenfalls aufgelöst - anhand der in Kapitel 3.2 beschriebenen Transformationregeln.

[38] engl. not

Beispiel 4.2.1. Transformation
Die Formel in Beispiel 4.1.1, überführt in eine Konjunktive Normalform, ist in Listing 4.2 zu sehen. Der Daten-Konstruktor *Pselem* wird während der Ersetzung der Variablen durch Zahlen in eine Listendarstellung überführt..

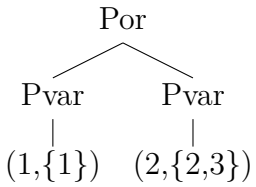


Abbildung 4.2: Formel aus Beispiel 4.1.1 in KNF transformiert

4.3 Test auf (Un)-Erfüllbarkeit

Die Funktion $dpfsSat$ ist die strenggenommene Kernfunktion der Implementierung. Diese Funktion überprüft eine Eingabeformel auf (Un)-Erfüllbarkeit, wobei die Eingabeformel einer Klauselmenge entsprechen muss. Bevor zu den einzelnen Regeln die Implementierung erläutert wird, wird die Auswahl des Datentypes der Klauselmenge genauer beschrieben.
Sowohl die erste Erweiterung um Mengen-Prädikate, als auch die zweite Erweiterung um Kalküle operieren mit endlichen Mengen:

- 1. Erweiterung: $x \in \mathcal{M}$, wobei $\mathcal{M}$ einer endlichen Menge entspricht,
- 2. Erweiterung: $(\{x_1, \ldots, x_n\} = \{1, \ldots, n\})$, wobei die Darstellungen links und rechts des Gleichheitszeichen endlichen Menge entsprechen.

Die in Kapitel 3.3 vorgestellten Regeln der beiden Erweiterungen führen hauptsächlich folgende Operationen auf diesen Mengen durch: Suchen von Elementen, die Vereinigung von zwei endlichen Mengen, sowie der Durchschnitt von zwei endlichen Mengen. Vor allem in Hinblick der Effizienz der beiden zuletzt genannten Operationen wird der Datentyp $IntSet$ aus dem Modul $Data.IntSet$[39] als Repräsentation der endlichen Mengen verwendet.

[39] Nähere Informationen unter http://http://hackage.haskell.org/package/containers-0.5.0.0

Die Implementierung dieses Datentyps basiert auf der Verwendung von Patricia-Bäumen[40], welche besonders performant in den oben beschriebenen häufig benutzten Operationen sind.[OG98]
In Konsequenz besteht ein Patricia-Baum mit n-Blättern aus $n-1$-Knoten. Ergänzend werden die Besonderheiten von Patricia-Bäumen kurz zusammengefasst, gleichwohl wird an dieser Stelle aber auf die detailiertere Erläuterung im Grundlagen Kapitel verwiesen.

- Kompaktere Darstellung, indem jeder innere Knoten, welcher nur ein Kind besitzt zusammengefasst wird.
- Im Vergleich zu anderen Implementierungen haben die inneren Knoten Markierungen, die die Position der Bitfolge darstellen.
- Speicherung der Bits in „Big endian“ Reihenfolge. (Ein Wort wird mit dem höchstwertigen Bit zuerst gespeichert)

Nachdem der verwendete Datentyp für die endlichen Mengen festgelegt ist, wird im Folgenden der Datentyp für die Atome und der Klauselmenge selbst betrachtet. Ein Atom ist entweder ein Mengen-Prädikat oder ein Kalkül. Listing 4.4 zeigt die Definition des verwendeten Datentyps. Das Wort Atom nachfolgend zum *data* Schlüsselwort bezeichnet den Typ des definierten Datentyps. Rechts vom Gleichheitszeichen gibt es zwei Daten-Konstruktoren: Xin und $Calc$.

```
data Atom = Xin  Int IntSet.IntSet
    | Calc IntSet.IntSet IntSet.IntSet
    deriving (Eq,Show)

type Klausel = [Atom]
type KlauselMenge = [Klausel]
type AtomTupel = (Int, IntSet.IntSet)
type FiniteSet = Int
type CalculusSwitch = Int
type SolutionSet = IntMap.IntMap IntSet.IntSet
```

Listing 4.4: Datentyp KlauselMenge

[40] engl. „Practical Algorithm To Retrieve Information Coded In Alphanumeric tries“ , kurz patricia tries

Der erste Daten-Konstruktor „Xin“ hat zwei Typen und repräsentiert Mengen-Prädikate der Form $x \in \mathcal{M}$. Wobei der erste Typ einer Variablen x als Int und der zweite Typ einer endliche Menge $\mathcal{M}$ als $IntSet$ entspricht.
Analog hat der zweite Daten-Konstruktor $Calc$ ebenfalls zwei Typen und repräsentiert Kalküle der Form $(\{x_1, \ldots, x_n\} = \{1, \ldots, n\})$. Der erste Typ entspricht der linken Seite und der zweite Typ der rechten Seite der Mengengleichheit bzw. dem Kalkül.
Nachfolgend zur Definition des Datentyps $Atom$ sind zur besseren Lesbarkeit und Dokumentierung des Quelltextes noch Typ-Synonyme definiert. Eine Klausel besteht demnach aus einer Liste von Elementen vom Datentyp $Atom$ und eine Klauselmenge aus einer Liste von Klauseln.

Nachdem der verwendete Datentyp beschrieben wurde, steht im Folgenden die bereits zu Beginn angesprochene Funktion $dpfsSat$ und die im weiteren Verlauf benutzten Funktionen im Vordergrund. Die Typ-Signatur der Funktion $dpfsSat$ ist im Folgendem dargestellt.

```
type FiniteSet = Int
type CalculusSwitch = Int
type SolutionSet = IntMap.IntMap IntSet.IntSet

dpfsSat :: CalculusSwitch → FiniteSet → KlauselMenge → SolutionSet
dpfsSat 0 fset klauselMenge = dpfsSat' 0 fset klauselMenge IntMap.
    empty
dpfsSat 1 fset klauselMenge = dpfsSat' 1 fset klauselMenge IntMap.
    empty
```

Gleichbedeutend zur zu Beginn des Kapitels beschriebenen Funktion $dpfs$ besitzt die Funktion $dpfsSat$ auch drei Argumente. Das erste Argument entspricht ebenfalls einen Selektor, womit die jeweilige Erweiterung gewählt wird. Mit der Zahl 0 wird die Erweiterung um Mengen-Prädikaten genutzt und mit der Zahl 1 die Erweiterung um Kalküle.
Das zweite Argument ist die festgelegte endliche Menge $\mathcal{M}$; die Funktion erwartet hierfür einen Int-Wert. Dieser Int-Wert entspricht dem maximalem Element der Menge. D.h für einen Wert n wird die Menge $\{1, \ldots, n\}$ angenommen.
Schlussendlich ist das letzte Argument die zu überprüfende Formel als Klauselmenge. (siehe definierten Typ in Listing 4.4)

Das Resultat der Funktion, gleichbedeutend mit der Lösungsmenge L, ist eine $IntMap$ (Typsynonym $SolutionSet$), d.h eine Sammlung von Zuordnungen von Schlüsseln zu Werten. In Kapitel 3.3 wurden die Regeln für die beiden Erweiterungen erläutert. Je nach Anwendung der Regeln, werden Atome der Form $x \in \mathcal{M}$ zu der Lösungsmenge hinzugefügt. Dieses Hinzufügen entspricht einen neuen Eintrag eines Schlüssels in der $IntMap$. Wobei als Schlüssel der Int-Wert der Variable und als Wert die jeweilige Menge M angenommen wird. Als Implementierung der Lösungsmenge wird eine $IntMap$ verwendet um doppelte Einträge innerhalb der Lösungsmenge zu vermeiden. D.h gibt es zu einem Zeitpunkt in der Lösungsmenge bereits eine Zuordnung einer Variablen mit einem Wert, und im Folgendem soll ein weiteres Atom ($x \in M_1$) mit derselben Variablen hinzugefügt werden. Dann wird die betreffende Zuordung dahingend verändert, indem als neuer Wert der Mengendurchschnitt des alten Wertes und der Menge M_1 des zu hinzugefügendem Atom angenommen wird.

Im weiteren Verlauf wird nun die Funktion $dpfsSat'$ abhängig vom gewählten Selektor beschrieben. Beide Fälle entsprechen einer Rekursion, wobei die Vorgehensweise sich im Einzelnen unterscheiden.

Implementierung: Mengen-Prädikate

Einleitend mit der Erweiterung um Mengen-Prädikate, wird der Auszug aus dem Quelltext in Listing 4.5 dargestellt. Dem ersten Argument, dem Selektor, muss hierfür eine 0 übergeben werden. Wie anhand der Signatur in Zeile 5 zu erkennen, hat die Funktion $dpfsSat'$ ein zusätzliches Argument. Dieses Argument spiegelt in jedem Rekursionschritt die aktuelle Lösungsmenge wieder. Beim ersten Aufruf der Funktion ist die Lösungsmenge leer, d.h die $IntMap$ ist eine leere Zuordnung.
Es ist leicht zu erkennen, dass die Funktion $dpfsSat'$ eine End-Rekursive[41] Funktion ist. Der letzte Funktionsaufruf innerhalb der Rekursion ist die Funktion $dpfsSat'$ selbst.
Zur besseren Übersichtlichkeit und um die Anwendbarkeit der Regeln zu testen, wird innerhalb der Funktion Guards als Kontrollstruktur verwendet.(vgl. Abschnitt 2.2.3.3) Guards werden angedeutet mit dem senkrechten Strich[42] und einem nachfolgenden aussagenlogischen Ausdruck. Wird der Ausdruck zu

[41] engl. tail recursiv
[42] engl. pipe

Falsch[43] ausgewertet, wird der nächst folgende Guard getestet. Für den Fall, dass der Ausdruck zu *Wahr*[44] ausgewertet wird, führt dies zur Auswertung des Funktionsrumpfes des jeweiligen Guards. (Der Ausdruck nach dem Gleichheitszeichen)
Im Rahmen der Implementierung handelt es sich bei den Ausdrücken überwiegend um Funktionen, die als Rückgabewert den Typ *Maybe* besitzen. In Haskell besitzt ein Wert von Typ Maybe entweder einen Wert vom Typ *a* (angedeutet mit Just *a*), oder der Wert ist leer[45] (angedeutet mit Nothing). Mit den beiden Funktionen *isJust* und *fromJust* lassen sich Ausdrücke vom Typ *Maybe* als aussagenlogischen Typ darstellen und sind somit als Ausdruck innerhalb eines Guards verwendbar.

Innerhalb der Funktion gibt es sechs dieser Guards, welche den in Abschnitt 3.3.1.1 beschriebenen Regeln entsprechen. Im Folgendem wird Bezug auf diese Regeln genommen, jedoch wird für eine genauere Vorgehensweise auf den gerade referenzierten Abschnitt hingewiesen. Die beiden Guards in Zeile 8 bis 10 erfüllen die Regel 1. Die nachfolgenden drei Guards in Zeile 11 bis 18 entsprechen den Regeln 3,4 und 5.

Erklärung des Quelltextes:

- Zeile 8: Der erste Guard dient als Abbruchkriterium; Existiert innerhalb der Klauselmenge eine leere Klausel, dann führt dies zum Widerspruch und die Klauselmenge ist nicht erfüllbar. Der Rückgabewert der Funktion ist eine leere *IntMap*.
- Zeile 9: Die Funktion *findTrueFalse* bekommt als Argumente die festgelegt endliche Menge und die Klauselmenge übergeben. Der Rückgabewert ist vom Typ *Maybe*; Existiert innerhalb der Klauselmenge ein Atom *u*, sodass die Simplifikationen, beschrieben in 3.3.1.1 angewendet werden können, wird als Typ **„Just u“** zurückgegeben. Dieses gefundene Atom *u* wird dann durch die Funktion *resolveTrueFalse* simplifiziert. Für den Fall, dass kein passendes Atom gefunden wird, wird der Typ **„Nothing“** zurückgegeben.

[43] engl. false
[44] engl. true
[45] engl. empty

```
type FiniteSet = Int
type CalculusSwitch = Int
type SolutionSet = IntMap.IntMap IntSet.IntSet

dpfsSat' :: CalculusSwitch → FiniteSet → KlauselMenge →
    SolutionSet → SolutionSet
dpfsSat' 0 _ [] loesung = loesung
dpfsSat' 0 f km loesung
 | [] `elem` km = IntMap.empty -- Abruchkriterium, Widerspruch
    gefunden
 | isJust (fTF) =
    dpfsSat' 0 f (resolveTrueFalse f (fromJust fTF) km) loesung
 | isJust (fSU) =
    dpfsSat' 0 f ((substituteSimilarUnits (fromJust fSU) km)) loesung
 | isJust (fU) = let (u1,u2) = fromJust fU
                 in
                  dpfsSat' 0 f ((resolveUnit (u1,u2) km )) (insAtom
                     u1 u2)
 | isJust (fP) = let (u1,u2) = fromJust fP
                 in
                  dpfsSat' 0 f ((resolvePureLiteral (u1,u2) km)) (
                     insAtom u1 u2)
 | otherwise = let (m1,m2) = findBestLiteral km
                   (k1,k2) = (m1,(Set.difference (Set.fromList [1..f
                      ]) m2))
                   newMap  = (insAtom m1 m2)
                   res = let k' = (resolveUnit (m1,m2) km)
                         in
                          (dpfsSat' 0 f k' newMap)
               in
                {- Fuehrt die aktuelle Belegung zum Widerspruch, muss
                    die
                Klauselmenge zurueckgesetzt werden (Backtrack) -}
                if IntMap.null res
                then
                  dpfsSat' 0 f (resolveUnit (k1,k2) km) (insAtom k1
                     k2)
                else
                  res
  where
    fTF = findTrueFalse f km
    fSU = findSimilarAtomUnits f km
    fU  = findUnit km
    fP  = findPureLiteral km
    insAtom v1 v2 = IntMap.insertWith (Set.intersection) v1 v2
        loesung
```

Listing 4.5: Algorithmus Mengen-Prädikate

- Zeile 11: Die Funktion $findSimilarAtomUnits$ bekommt als Argumente die festgelegt endliche Menge und die Klauselmenge übergeben und liefert als Rückgabewert ein Tupel aus einem gefundenen Literal $x1$, sodass $x1$ in mehreren 1-Klauseln vorkommt und der aktualisierten Menge M. (Mengenschnitt aller beteiligenten Menge M_i) Die $substituteSimilarUnits$ fasst die entsprechenden 1-Klauseln zu einer 1-Klausel zusammen.
  Wird kein passendes Literal $x1$ gefunden, wird als Typ **Nothing** zurückgegeben.
- Zeile 13: Die Funktion $findUnit$ findet 1-Klauseln innerhalb der Klauselmenge und gibt die zuerst gefundene Klausel, bzw. deren Atom zurück. Anschließend löst die im Funktionsrumpf sich befindene Funktion $resolveUnit$ diese 1-Klausel nach Regel 4 auf.
- Zeile 16: Die Funktion $findPureLiteral$ findet isolierte Literale innerhalb der Klauselmenge. Sofern ein isoliertes Literal vorkommt, gibt die Funktion dieses Literal x und deren Menge vom Typ **(Just** x, M**)** zurück. In diesem Fall wird das Literal der Funktion $resolvePureLiteral$ übergeben und aufgelöst. Anderenfalls gibt die Funktion den Typ **Nothing** zurück.

Werden alle Ausdrücke der Guards zu $Falsch$ ausgewertet, kann anhand der beschriebenen Regeln keine logischen Schlüsse/Folgerungen mehr hergeleitet werden. In diesem Fall wird die Splitting Regel, bzw. der Funktionsrumpf des letzten Guards „*otherwise*“ angewendet. Innerhalb dieses Funktionsrumpfes wird eine Fallunterscheidung durchgeführt und für die Auswahl eines Mengen-Prädikats wird als Heuristik die MOMS-Strategie verwendet. (vgl. Abschnitt 3.3.1.2) Der anschließende Quelltext zeigt die Funktion zur Auswahl eines Atom unter der Berücksichtigung der beschriebenen Heuristik.

```
1 findBestLiteral :: KlauselMenge → (IntMap.Key, Set.IntSet)
2 findBestLiteral k =
3   let k' = concat k
4       k'' = foldl' (λmap1 (Xin k1 _) → IntMap.insertWith (+) k1 1
            map1) IntMap.empty k'
5       min = fst $ IntMap.foldlWithKey' (λ(x1,x2) y z → if x1 == 0
6                                                          then (y,z)
7                                                        else
8                                                          if x2 > z
```

```
9                                                          then (y,z)
10                                                        else (x1,x2
                                                             ) )
                                                             (0,0) k
                                                             ''
11        (v1,v2) = findSmallestSet min k'
12    in
13     v1 `strictPair` v2
```

Zunächst wird die als Argument der Funktion $findBestLiteral$ übergebene Klauselmenge konkateniert, d.h alle Klauseln werden miteinander verknüpft. Im zweiten Schritt werden die Häufigkeiten alle Literale bestimmt. Dazu werden nach und nach alle Literale in einer $IntMap$ gespeichert. Da innerhalb der Klauselmenge Literale mehrfach vorkommen dürfen und in einer $IntMap$ einen Schlüssel nur einmal geben darf, kann es zu Konflikten kommen. In diesen Fällen wird der Wert dieser Zuordung immer um eins inkrementiert. Anschließend kann man sich das Literal x, welches am Häufigsten vorkommt ausgeben lassen. Mit dieser Information wird nun eine zweite $IntMap$ befüllt. Alle Atome $x \in M$, wobei x des am Häufigsten vorkommenden Literals darstellt, werden nach und nach betrachtet. Der Wert der Zuordnung in der $IntMap$ wird nur dann angepasst, falls die Menge M des gerade betrachteten Atoms kleiner ist, als der bisherige Wert der Zuordnung.
Abschließend wird das Ergebnis als Tupel aus dem Literal in Zahlendarstellung und der kleinsten Menge dieses Literals zurückgegeben.
Das von der Funktion $findBestLiteral$ zurückgegebene Mengen-Prädikat wird nun durch eine Fallunterscheidung verwendet und es gilt die folgende Vorgehensweise:

- Zunächst wird das Mengen-Prädikat $x \in M$, welches die Funktion $findBestLiteral$ zurück gibt, als 1-Klausel der Klauselmenge und ebenfalls der Lösungsmenge hinzugefügt. Führt diese Belegung im weiterem Verlauf zu einem Widerspruch durch die Anwendung der Regeln, bzw. Funktionen, dann muss die Klauselmenge zurückgesetzt werden, d.h. zwischenzeitliche Veränderungen müssen revidiert werden. Anschließend wird für das Mengen-Prädikat das Komplement der Menge M gebildet und das Mengen-Prädikat mit dem Komplement M' anstatt mit der Menge M als 1-Klausel der revidierten Klauselmenge hinzugefügt. In

der Implementierung wird dies durch eine if-Schleife realisiert. Die if-Schleife reicht in diesem Zusammenhang, da es nur zwei Fälle innerhalb der Fallunterscheidung zu betrachten gilt.

Implementierung Kalkülerweiterung

Als nächstes wird der Funktionsrumpf für die Kalkülerweiterung der Funktion $dpfsSat'$ erläutert. Der entsprechende Quelltext ist in Listing 4.6 zu sehen. Dem ersten Argument, dem Selektor muss in diesem Fall eine 1 übergeben werden. Die Argumente zwei und drei sind zum einen die festgelegte endliche Menge und zum anderem die zu überprüfende Formel in Konjunktiver Normalform (KNF). Wie im vorherigen Teilabschnitt bereits angesprochen, besitzt die Funktion $dpfsSat'$ ein Argument, um jedem Rekursionsaufruf die aktuelle Lösungsmenge übergeben zu können. Analog zu der Implementierungsbeschreibung der ersten Erweiterung um Mengen-Prädikate wird für die Lösungsmenge eine $IntMap$ verwendet. Ebenso wird der Funktion beim ersten Aufruf eine leer $IntMap$ als Lösungsmenge übergeben, d.h eine leere Zuordnung.
Des Weiteren wird bei der Implementierung der Kalkülerweiterung zur besseren Übersichtlichkeit ebenfalls Guards als Kontrollstruktur verwendet. Insgesamt werden für die Kontrollstruktur sieben Guards verwendet um die Anwendbarkeit, der in Abschnitt 3.3.2.1 beschriebenen Regeln, zu prüfen. Ergänzend wird auf die den Regeln entsprechenden Programmstellen bezug genommen, allerdings für eine detailiertere Beschreibung auf den gerade referenzierten Abschnitt verwiesen.

Bemerkung: Zu Beginn der Funktion in Listing 4.6 werden alle Atome der Form $x \in \mathcal{M}$, d.h vor Anwendung der Teilfunktionen in die Lösungsmenge eingefügt. Die Klauselmenge besteht ab diesem Zeitpunkt nur aus Mengengleichheiten, bzw. Kalkülen. Es gibt allerdings Regeln, wie zum Beispiel der Mengensplit, indem ein Kalkül anhand einer Auswahl von gefunden Atomen der Form $x \in M$ in zwei oder mehrere Kalküle zerlegt werden können. In diesem Fall wird für die Suche einer möglichen Auswahl der Inhalt der aktuellen Lösungsmenge verwendet.
Konsequenterweise gilt für die während der Fallunterscheidung ausgewählten Atome $x \in M$ ebenfalls: Das Mengen-Prädikat wird ausschließlich direkt der Lösungsmenge hinzugefügt.

Erklärung des Quelltextes:

```haskell
type FiniteSet = Int
type CalculusSwitch = Int
type SolutionSet = IntMap.IntMap IntSet.IntSet

dpfsSat' :: CalculusSwitch → FiniteSet → KlauselMenge →
    SolutionSet → SolutionSet
dpfsSat' 1 _ [] l = l
dpfsSat' 1 f k l
  | [] `elem` k = IntMap.empty
  | isJust fU  = dpfsSat' 1 f ( resolveUnitCalculus v1 k ) (insAtom (
     fst v1) (snd v1))
  | isJust fSR = let l' = resolveSimpRule3 v4 k l in dpfsSat' 1 f k l
     '
  | isJust fF  = dpfsSat' 1 f ( resolveFalse_Calculus k ) l
  | isJust fSU = if IntMap.null (snd v3)
                 then dpfsSat' 1 f ( resolveCalculus1 (fst v3) k l)
                     l
                else dpfsSat' 1 f ( resolveCalculus1 (fst v3) k (snd
                     v3)) (snd v3)
  | isJust fSe = dpfsSat' 1 f (setSplit2 v2 k l) l
  | otherwise  = fSplit minUnit
  where
    fU  = findUnit k
    fF  = findFalse_Calculus k
    fUM = findUnitMultiple $ lToList2 l
    fSU = findSimpUnitCalculus k l
    fSR = findSimpRule3 k l
    fSe = findSelection2 k l
    v1  = expectJust fU
    v2  = expectJust fSe
    v3  = expectJust fSU
    v4  = expectJust fSR
    minUnit = enumerate_BestLiteral (snd $ findBestLiteral_Calculus k
       )
    fSplit [] = IntMap.empty
    fSplit (x:xs)
      | IntMap.null expr = fSplit xs
      | otherwise = expr
      where expr = (dpfsSat' 1 f k (insAtom (fst x) (snd x)))
    insAtom v1 v2 = IntMap.insertWith (Set.intersection) v1 v2 l
```

Listing 4.6: Algorithmus Mengengleichheiten (Kalküle)

- Zeile 8: Der erste Guard dient als Abbruchkriterium; Existiert innerhalb der Klauselmenge eine leere Klausel, dann führt dies zum Widerspruch und die Klauselmenge ist nicht erfüllbar. Der Rückgabewert der Funktion ist eine leere $IntMap$.
- Zeile 9: Die Funktion $finUnit$ durchsucht die Klauselmenge nach 1-Klauseln deren Element ein Atom der Form $x \in M$ ist. Existiert solch ein Atom in der Klauselmenge, dann enfernt die Funktion $resolveUnitCalculus$ dieses Atom aus der Klauselmenge und fügt es der Lösungsmenge hinzu. D.h es wird ein neuer Eintrag der $IntMap$ hinzugfügt, sodass der Schlüssel die Variable x als Int und der Wert die Menge M entspricht.
- Zeile 10: Die Funktion $findSimpRule3$ bekommt als Argumente die Klauselmenge und die aktuelle Lösungsmenge übergeben und liefert einen Rückgabewert vom Typ $Maybe$. Existiert in der Lösungsmenge ein Atom $x \in M$ mit $|M| > 1$ und innerhalb der Klauselmenge ein Kalkül c, sodass die Regel in [3.3.2.1] zu einer Verkleinerung der Menge M führt, dann wird ein Tupel bestehend aus dem Mengen-Prädikat und den Kalkül c zurückgegeben. Das Mengen-Prädikat wird anschließend durch die Funktion $resolveSimpRule3$ simplifiziert. Für den Fall, dass für ein Atom $x \in M$, mit $|M| > 1$ kein Kalkül gefunden wird, wird der Typ **„Nothing“** zurückgegeben.
- Zeile 11: Die Funktion $findFalseCalculus$ erhält als Argument die Klauselmenge und hat einen Rückgabewert vom Typ **„Maybe“** ; Existiert innerhalb der Klauselmenge ein Kalkül, sodass die linke und/oder die rechte Menge dieses Kalküls entweder einer leeren Menge oder ungleicher Mächtigkeit entsprechen, dann wird dieses Kalkül c als Typ **„Just** c**“** zurückgegeben. Existiert kein passendes Kalkül, wird der Typ **„Nothing“** zurückgegeben.
- Zeile 12: Die Funktion $findSimpUnitCalculus$ durchsucht die Klauselmenge nach Simplifikationen der Regel 1. Sei $x \in M$ ein Atom und gibt es eine Mengengleichheit, sodass x in der linken Menge vorkommt. Dann wird diese Mengengleichheit durch die Funktion $resolveCalculus1$ simplifiziert. (Siehe Regel 1 - Abschnitt 3.3.2.1)
- Zeile 15: Die Funktion $findSelection2$ überprüft jede Mengengleichheit der Klauselmenge, ob es eine passende Auswahl von Atomen der Form

$(x_1 \in M_1), \ldots, (x_n \in M_n)$ in der Lösungsmenge gibt. Existiert solch eine Auswahl, dann wird die entsprechende Mengengleichheit c vom Typ **„Maybe“** als **„Just c“** zurückgegeben. Andernfalls liefert die Funktion den Typ **„Nothing“** , d.h. es wurde keine Auswahl gefunden für alle Mengengleichheiten.

Die Funktion *setSplit*2 bekommt schließlich als Argument, die von der Funktion $findSelection2$ zurückgegebene Mengengleichheit und zelegt es in zwei Mengengleichheiten. Die beiden enstandenen Mengengleichheiten werden anschließend, sofern die Mächtigkeit der Mengen rechts und links des Gleichheitszeichens zwischen zwei und vier besteht, nochmals versucht zu zerlegen.

Werden alle Ausdrücke der Guards zu $Falsch$ ausgewertet, kann anhand der beschriebenen Regeln keine logischen Schlüsse/Folgerungen mehr hergeleitet werden. In diesem Fall wird die Splitting-Regel angewendet, indem die Funktion $fSplit$ ausgeführt wird. $fSplit$ erhält als Argument eine Liste von Mengen-Prädikaten.[46] $((x \in M_1), \cdots, (x \in M_n))$ Diese Liste ist das Ergebnis der Auswahl einer Variable für die Fallunterscheidung. Der folgende Quelltext zeigt die Funktion zur Auswahl einer Variablen unter der Berücksichtigung der verwendeten Heuristik.(vgl. Abschnitt 3.3.2.2) Der nachfolgende Auszug aus dem Quelltext zeigt die beiden Funktionen, welche die Auswahl eines Atom der Form $x \in M$ treffen.

```
1 findBestLiteralCalculus :: KlauselMenge → (Int,(Set.IntSet, Set.
      IntSet))
2 findBestLiteralCalculus klauselMenge = let cl = (concat klauselMenge)
3                                        in
4                                         IntMap.findMin $ IntMap.
                                              fromList $ map (λ(Calc a
                                              b) → ((Set.size b),(a,b)
                                              )) cl
5
6 enumerateBestLiteral :: (Set.IntSet, Set.IntSet) → [(Int,Set.IntSet)
      ]
7 enumerateBestLiteral (x,y)
8   | Set.null x = []
```

[46] Alle Atome innerhalb der Liste entsprechen der gleichen Variable, aber mit unterschiedlichen Mengen

```
9   | otherwise = let (i,_) = Set.deleteFindMin x
10                    b     = Set.toList y
11                in
12                 (foldr (λz l → (i,(Set.singleton z)) : l) [] b)
```

Die verwendete Heuristik wählt zunächst ein Kalkül c, sodass die Menge rechts, die kleinste Mächtigkeit von allen anderen Kalkülen aufweist. Anhand diesem gewählten Kalkül wird die Zusammensetzung der oben beschriebenen Liste vorgenommen. Die Funktion *findBestLiteralCalculus* ist die Implementierung der Auswahl eines Kalkül mit minimaler Länge der rechten Menge und die Funktion *enumerateBestLiteral* füllt eine Liste von Mengen-Prädikaten mit jeweils gleicher Variable.
Die Funktion *findBestLiteralCalculus* bekommt als Argument eine Klauselmenge übergeben und liefert als Rückgabewert ein Tupel aus der Anzahl der Elemente der Mengen eines Kalküls und dem Kalkül selbst zurück. Ähnlich zur verwendeten Implementierung der ersten Erweiterung werden die Klauseln der als Argument übergebene Klauselmenge zunächst miteinander konkateniert. Anschließend wird jedes vorkommende Kalkül nach und nach in eine *IntMap* eingefügt, wobei der Schlüssel die Mächtigkeit der jeweiligen Mengen entspricht und der Wert ein Tupel aus der linken und der rechten Seite dieses Kalküls. Derjenige Eintrag mit dem kleinsten Schlüssel ist der Rückgabewert.
Die Funktion *enumerateBestLiteral* bekommt als Argument genau ein Tupel, welches von der vorherigen Funktion berechnet wird und liefert als Rückgabewert die Liste, die wie folgt aufgebaut:

Sei $(\{x_1, x2, \ldots, x_n\}, \{c_1, c_2, \ldots, c_n\})$ das als Argument übergebene Kalkül.

- die erste Variable x_1 wird als Variable angenommen, und
- entsprechend der Anzahl der Konstanten in der rechten Menge, wird die Variable x_1 folgendermaßen aufgezählt:

$$[(x_1 \in c_1), (x_1 \in c_2), \ldots, (x_1 \in c_n)]$$

Die Liste von Mengen-Prädikaten wird nun durch die Funktion $fSplit$ für eine Fallunterscheidung verwendet. Insgesamt gibt es genau so viele Fälle wie Elemente innerhalb dieser Liste. Beim ersten Aufruf von $fSplit$ wird das erste Element der Lösungsmenge hinzugefügt. Führt diese Belegung im weiterem

Verlauf zu einem Widerspruch, mössen alle Veränderungen der Klauselmenge bis dahin zurückgesetzt werden. Die Unerfüllbarkeit ist somit für die aktuelle Belegung nachgewiesen. In diesem Fall wird die Funktion *fSplit* rekursiv mit dem nächsten Element in der Liste aufgerufen. Diese Vorgehensweise kann solange wiederholt werden, bis alle Elemente der Liste, bzw. alle Fälle der Fallunterscheidung ausprobiert wurden. Führen alle Fälle zu einem Widerspruch, ist die Klauselmenge nicht erfüllbar.

5 Analyse

Die Laufzeit von Entscheidungsalgorithmen, insbesondere die auf dem *Davis -Putnam - Logemann - Loveland Algorithmus* (DPLL) basieren, sind sehr stark abhängig von der Größe der Formel. Die Formel muss in einer Konjunktiven Normalform (KNF) vorliegen und somit ist die Anzahl und die Größe der Disjunktionsterme (Klauseln) entscheidend. Zur Erinnerung: Eine Formel in Konjunktiver Normalform gilt als erfüllt, wenn alle Disjunktionsterme innerhalb der Formel zu $Wahr$ evaluiert werden. Anderenfalls wie in Definition 3.2.1 beschrieben, ist die Formel mit der aktuellen Interpretation nicht erfüllbar.

Ziel dieser Arbeit war es, eine aussagenlogische Formel durch zwei geeignete Kodierungen zu vereinfachen. Ich habe dazu zwei Varianten auf Grundlage der Davis-Putnam Prozedur implementiert, welche die Formel kompakter darstellen sollen als in der aussagenlogischen Form mit Variablen, die entweder mit 1 ($Wahr$) oder 0 ($Falsch$) belegt werden können.

Um die Ausdruckskraft der beiden Erweiterungen zu illustrieren, wurden in Kapitel 3.3 Beispiele erläutert, die besonders von den neuen Kodierungen profitieren. In diesem Kapitel werden nun die Laufzeiten des Algorithmus betrachtet. Dazu werden die angesprochenen Beispiele und noch ein paar Weitere als Eingabeformel dem Algorithmus übergeben. Im Quelltext gibt es ein Modul mit dem Namen Test, welche die Funktionen für die Generierung der Formeln enthalten.
Die im weiterem Verlauf gezeigten Experimente wurden auf einem 2.93Ghz Intel(R) Core(TM) i3 Prozessor mit 4GB Arbeitsspeicher unter dem Betriebssystem Linux[47] durchgeführt. Als Compiler wurde die Version 7.4.1 des *Glasgow Haskell Compiler*(GHC)[48] und zusätzlich dazu die Version 0.5.0.0 des Containers Paket, welches zur Repräsentation der Mengen benutzt wird, verwendet. Des Weiteren wurde bei der Implementierung darauf geachtet, dass neben möglichst effizienten Datenstrukturen auch möglichst End-Rekursion in Hinblick auf den Speicherverbauch verwendet wird.

[47] Kernelversion 2.6.42
[48] kompiliert mit den Compilerflags: -O2 -fforce-recomp -funbox-strict-fields

Erste Erweiterung - Mengen-Prädikate

Zusätzlich zum allgemeinen N-Damen Problem werden noch zwei Verschärfungen betrachtet. Zum Einem das N-Superdamen und zum Anderen das N-Torusdamen Problem. Für eine genauere Erklärung der Problemstellung wird an dieser Stelle auf das Kapitel 7 verwiesen.
In den beiden Tabellen (5.1) und (5.2) sind die experimentellen Resultate für den verwendeten Algorithmus zu sehen. Insgesamt werden die Beispiele auf neun verschiedenen Schachbrettgrößen ausprobiert, wie in Spalte 1 zu sehen ist. Es wird bei dem N-Damen Problem auf dem Torus andere Schachbrettgrößen verwendet, da es nur vollständige Lösungen gibt, d.h. in jeder Zeile und in jeder Spalte kommt eine Dame vor, wenn N teilerfremd zu der Zahl sechs. In der nächsten Spalte ist die Anzahl der Klauseln, welche mit der Kodierung um Mengen-Prädikaten notwendig sind, dargestellt. Dem gegenübergestellt wird die Anzahl der Klauseln in der aussagenlogischen Form. Für jede Variante des N-Damen Problems wurden zwei Werte gemessen: Die Anzahl der Zurückgesetzten Literale und die Gesamtlaufzeit bis eine Lösung gefunden wurde.

Hinweis: Wird für einen betrachteten Fall kein Zurücksetzen eines Literals vorgenommen oder gibt es insgesamt keine Lösung, wird an dieser Stelle ein Minuszeichen eingefügt. Des Weiteren gibt ein Stern neben der Zeit an, dass der Arbeitspeicher nicht mehr ausgereicht hat und es zum *swapping* gekommen ist. Die jeweilige Zeit ist damit nicht mehr repräsentativ.

			allgemein N-Damen		N-Superdamen	
$n \times n$	# Klauseln	# Klauseln (AL)	# ZS	Zeit (Sek.)	# ZS	Zeit (Sek.)
4×4	52	84	-	0.005	-	-
8×8	456	744	-	0.011	-	-
16×16	3856	6352	1	0.122	1	0.122
24×24	13272	21944	1	1.256	1	1.040
32×32	31776	52640	-	9.561	1	7.050
40×40	62440	103560	1	36.000	1	28.640
48×48	168336	179824	1	104.350	-	79.100
56×56	172536	286552	-	243.900	-	194.740
64×64	258112	428864	1	1578.730*	1	636.290

Tabelle 5.1: Test Resultate der Varianten allgemeines N-Damen Problem und dem N-Superdamen Problem

			N-Damen Problem Torus	
$n \times n$	# Klauseln	# Klauseln (AL)	# ZS	Zeit (Min.)
13×13	2041	4084	1	0.059
19×19	6517	13036	1	0.335
25×25	15025	30052	1	5.659
31×31	28861	57724	1	13.644
37×37	49321	98644	1	94.710

Tabelle 5.2: Test Resultate des N-Damen Problem auf dem Torus

Für alle drei Varianten wird die gleiche Anzahl von Klauseln generiert. Der Unterschied liegt aber in der Größe der jeweiligen Mengen der Atome. Diese geben die möglichen Felder der Zeilen an, die innerhalb dieser Zeile besetzt werden können.
Die Laufzeit hängt auch sehr stark davon ab, wie groß diese Mengen sind, d.h wie viele Elemente sie besitzen. Jede dieser Mengen entspricht in der Implementierung einem Patricia-Baum (Patricia-Tries), sodass je größer die Menge, die Operationen auf diesen Mengen/Bäumen umso länger dauern. Aus dieser Tatsache lässt sich der Laufzeitunterschied ableiten, dass das allgemeine N-Damen Problem zum Teil eine längere Laufzeit besitzt als die anderen beiden Varianten, insbesondere bei immer größeren Mengen.
Des Weiteren kann man erkennen, dass die verwendete Heuristik zur Auswahl eines noch vorkommenden Atoms aus der Klauselmenge innerhalb der Fallunterscheidung effizient ist, da nicht mehr als ein Zurücksetzen der Klauselmenge vorgenommen wird.

Zweite Erweiterung - Mengengleichheiten (Kalküle)

Für die zweite Erweiterung werden ebenfalls mehrere Beispiele betrachtet. Neben dem Standard-Sudoku werden zusätzlich X-Soduku, Hyper-Sudoku, Nomino-Sudoku und zuletzt auch das aus fünf zum Teil überlappenden Standard-Sudokus bestehende Samurai-Sudoku betrachtet. An dieser Stelle wird keine detailierte Beschreibung der eben genannten Sudoku-Varianten vorgestellt. Vielmehr, wie bereits zu Beginn beschrieben, wird in Kapitel 7 eine genauere Beschreibung erläutert.
Für die Varianten Standard-Sudoku, X-Sudoku und Hyper-Sudoku werden die Gittergrößen von 9×9 bis 36×36 getestet für jeweils die Falle, dass das Gitter

unbelegt ist. Diese experimentellen Ergebnise des verwendeten Algorithmus sind in folgenden Tabellen dargestellt.

Standard-Sudoku			
Variante	Klauseln	Zurücksetzen	Zeit (Sek.)
9×9	27	5805	2.274
16×16	48	10609	3.687
25×25	75	16415	5.833
36×36	108	23110	8.506

X-Sudoku			
Variante	Klauseln	Zurücksetzen	Zeit (Sek.)
9×9	29	247	0.317
16×16	50	492	0.462
25×25	77	701	0.554
36×36	110	977	0.819

Abschließend werden noch die experimentellen Ergebniss für die beiden Varianten Nonomino und Samurai gezeigt.

weitere Sudoku Varianten				
Variante	$n \times n$	Klauseln	Zurücksetzen	Zeit (Sek.)
Nonomino	9×9	27	110	0.177
Hyper	9×9	31	5805	2.747
Samurai	-	51	426	1.633

Die verschiedenen Varianten von Sudoku lassen sich mit Kalkülen ausdrucksstark und kompakt darstellen. Insgesamt spielt im diesem Zusammenhang die Größe der jeweiligen endlichen Mengen eine untergeordnete Rolle. Es fällt auf, dass die Zahl des Zurücksetzens der Klauselmenge sehr hoch ist, insbesondere im Standard-Sudoku. Da die Beispiele ohne Vorbelegungen getestet werden, ist diese Tatsache aber nicht ungewöhnlich. Jedoch ist trotz der zum Teil großen Anzahl von Zurücksetzens die Implementierung sehr performant.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Ziel dieser Arbeit war es, eine Entscheidungsprozedur, basierend auf der Davis-Putnam Prozedur (DPLL), zum Erkennen der (Un-)Erfüllbarkeit von logischen Formel zu entwickeln, die im Gegensatz zu den aussagenlogischen Variablen Atome der Form $(x \in \mathcal{M})$ und $(\{x_1, \ldots, x_n\} = \{c_1, \ldots, c_n\})$, wobei $\mathcal{M}$ eine vorher festgelegte endliche Menge ist, verarbeitet. In Konsequenz soll ein Effizienzsteigerung hinsichtlich der Kompaktheit und Ausdruckskraft erreicht werden. Hiefür wurde die allgemeine Davis-Putnam Prozedur dahingehend modifiziert und letztlich in der Programmiersprache Haskell implementiert. Die Prozedur trägt die Bezeichnung DPFS[49].

In *Kapitel* 2 wurde zu Beginn eine Einführung in die Aussagenlogik und die Erfüllbarkeit von aussagenlogischen Formeln gegeben. Anschließend wurden die Konzepte der Funktionalen Programmierung erläutert, sowie eine Übersicht der Programmiersprache Haskell einschließlich einigen wichtigen Programmkonstrukten gegeben. Abgeschlossen wurde das Kapitel mit der Vorstellung des *containers*-Modul, welches für den Datentyp, bzw. für die Repräsentation der Mengen in den Atomen verwendet wird.

Da die Grundlage der zu entwickelten Entscheidungsprozedur das Davis-Putnam Verfahren war, musste in *Kapitel* 3 zunächst eine Beschreibung der Davis-Putnam Prozedur erfolgen. Dazu wurden unter Anderem die entsprechenden Regeln für das logische Schließen von Variablenbelegungen und für die Fallunterscheidung, der Algorithmus, sowie die Korrektheit und Vollständigkeit des Verfahrens gezeigt. Im Anschluss sind die beiden Erweiterungen und deren Kodierung gezeigt worden. Dabei wurde auf den Unterschied in den Regeln und der Fallunterscheidung im Vergleich zur allgemeinen Davis-Putnam Prozedur eingegangen. Des Weiteren wurde die Ausdruckskraft der Kodierung mit Atomen anhand von Beispielen gezeigt.

Nachdem die spezialisierte Prozedur erläutert wurde, erfolgten in *Kapitel* 4 die Implementierungsdetails. Die Prozedur umfasst drei Programmteile: Zuerst

[49] Die Bezeichnung steht für: *Davis – Putnam – With – Finite – Sets Algorithmus*

wurde die Syntax für die Eingabe der Formel, sowie der Parser dazu beschrieben. Im Anschluss sind die Funktionen zur Transformation einer geparsten Formel in eine Konjunktive Normalform gezeigt worden und zuletzt die Funktionen zum Test auf (Un-)Erfüllbarkeit dieser transformierten Formel. Dabei entsprechen die verwendeten Funktionen den in *Kapitel* 3 beschriebenen Regeln.

Im *Kapitel* 5 stand die Leistungsanalyse im Vordergrund. Der in *Kapitel* 4 beschriebene Algorithmus wurde mit mehreren Beispielen und Ausprägungen getestet. Dabei stellte sich unter Anderem heraus, dass die Laufzeit im Einzelnen ebenfalls sehr stark mit der verwendenten Anzahl der Elemente innerhalb der Mengen korreliert.

Zusammengefasst führen die beiden Erweiterungen zur einer deutlichen Steigerung der Ausdruckskraft und zu einer Verminderung der Klauselanzahl im Vergleich zur allgemeinen Davis-Putnam Prozedur. Die Korrektheit und Vollständigkeit bleibt bei der spezialisierten Prozedur erhalten und auch die Performance führt zu einem zufriedenstellenden Ergebnis, wobei eine Steigerung hinsichtlich der Operationen auf großen Menge und die Parallelisierung einzelner Programmteile sinnvoll und wünschenswert wären.

6.2 Verbesserungsansätze

Zum Schluss werden noch zwei Ansätze zur Verbesserung erläutert, die zukünftig noch umgesetzt werden könnten.

6.2.1 Effizienzsteigerung

Im *Kapitel* 5 - Analyse - wurde bereits angemerkt, dass die Performance stark nachlässt, umso größer die jeweilen endlichen Mengen der Atome werden. Dies ist zum Teil daran geschuldet, dass die in der Implementierung häufig benutzten Operationen auf den endlichen Mengen (Mengenvereinigung, Mengenschnitt etc.) sehr aufwendig sind.

Es könnte sich lohnen durch Optimierungen die Anzahl der Mengen insgesamt, oder die Elemente innerhalb einer Menge zu reduzieren. Zum einen könnte dies erreicht werden, indem für relativ große Mengen das Komplement gespeichert

wird. Sinnvoll wäre das, wenn zum Beispiel $x \in M$ ein Atom ist und die Menge M besitzt beinahe so viele Elemente wie die beim Aufruf des Algorithmus festgelegte endlichen Menge $\mathcal{M}$. Zum anderen lässt sich die Formel durch Vorarbeiten eventuell noch weiter Optimieren, sodass Variablen mit unterschiedlichen Mengen M_i noch mehr zusammenfassen lassen.

7 Beschreibung Beispiele

Ziel dieses Kapitel ist es, eine Beschreibung der in *Kapitel* 5 - Analyse - benutzten Beispiele zu geben. Zu jedem Beispiel wird eine kurze Erläuterung vorgenommen, ohne zu tief ins Detail zu gehen. Ferner wurde versucht, durch Abbildungen ein Verständnis der jeweiligen Problemfälle zu geben.

7.1 Damenproblem

Bei dem ursprünglichen Problem handelt sich es um das 8-Damen Problem. Es gilt acht Damen auf einem 8×8 großen Schachbrett dahingehend zu verteilen, sodass diese sich nicht nach den Schachregeln schlagen können. Es dürfen nicht mehrere Damen in gleicher Zeile, Spalte oder Diagonalen sein. Darüberhinaus wird angenommen, dass in jeder Spalten und Zeile genau eine Dame vorkommt. Das Problem lässt sich auf den allgemeinen Fall $(n \times n)$ erweitern. Die Abbildung 7.1 illustriert die verbotenen Felder exemplarisch am Feld in der sechsten Zeile und sechsten Spalte.

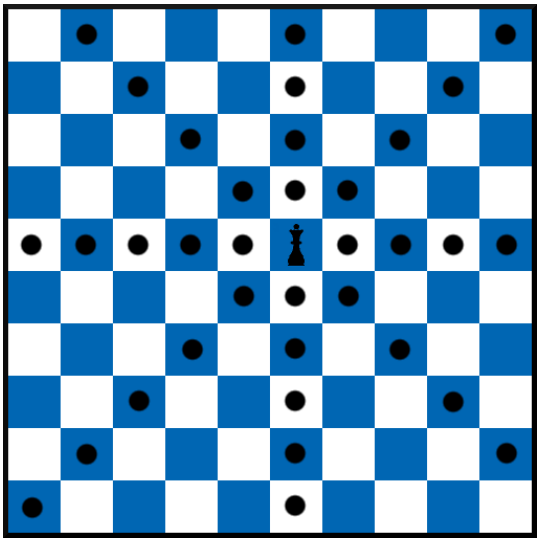


Abbildung 7.1: Allg. Damen Problem - 10×10-Schachbrett - Verbotene Felder

7.1.1 N-Super-Damen Problem

Das N-Super-Damen Problem ist eine Verschärfung des allgemeinen N-Damen Problem. Eine Super-Dame, platziert auf einem Feld auf dem Schachbrett, kann nicht nur andere Super-Damen platziert in der gleichen Reihe, Spalte und Diagonalen schlagen, sondern zusätzlich auch alle Super-Damen innerhalb eines Springer-Zug. Ein Springer-Zug ist Bewegung von zwei Feldern entweder nach Unter, Oben, Links oder Rechts mit einer anschließenden Bewegung nach links oder rechts. Wie in Abbildung 7.2 zu sehen ist, werden im Vergleich zum allg. Damenproblem, die möglichen Felder für eine Platzierung der anderen Super-Damen veringert.

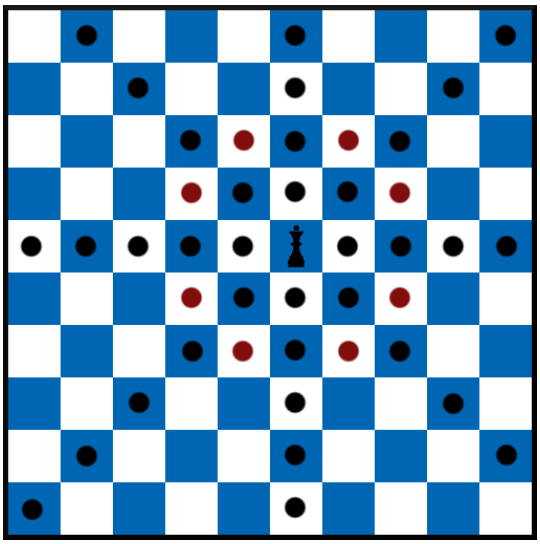


Abbildung 7.2: Superdamen Problem - 10×10-Schachbrett - Verbotene Felder

7.1.2 N-Torus-Damen Problem

Auch das N-Torus-Damen Problem ist eine Verschärfung des allgemeinen N-Damen Problem. Die Bewegungsmöglichkeiten einer Dame ist nicht durch die Schachbretträder begrenzt. Am einfachsten kann man sich diesen Sachverhalt vorstellen, indem die jeweils gegenüberliegenden Ränder zu einen Schachtorus verklebt werden. Somit wird der Wirkungsbereich einer Dame vergrössert. Abbildung 7.3 zeigt diese grenzüberschreitenden Wirkungsbereiche eine Dame. Für eine detailierte Beschreibung wird auf [uES] verwiesen.

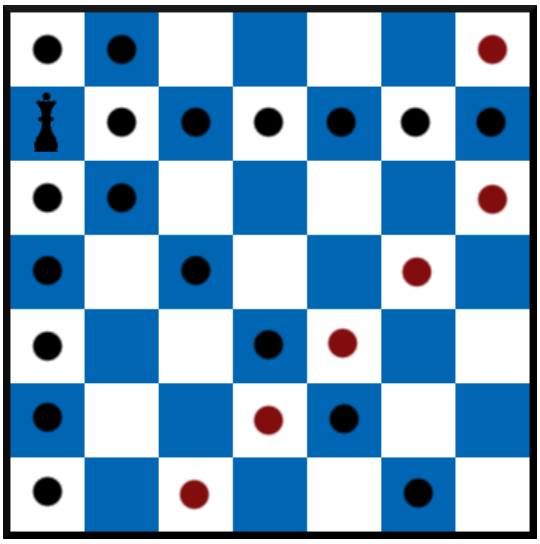


Abbildung 7.3: Damen Problem Torus - 7 × 7-Schachbrett - Verbotene Felder

7.2 Sudoku

Das Standard-Sudoku besteht aus 81 Felder, bzw. Quadraten und 9 Regionen. In jeder Zeile, in jeder Spalte und innerhalb jeder Region dürfen die Zahlen 1 bis 9 jeweils nur einmal vorkommen. Zu Beginn, je nach Schwierigkeitsgrad, sind bereits einige Felder mit Zahlen ausgefüllt. Zusätzlich zur dieser Standard Variante wurden im Kapitel 5 noch andere Varianten verwendet. Im Folgenden werden diese anderen Varianten kurz vorgestellt.

7.2.1 X-Sudoku

X-Sudoku besitzt im Vergleich zur Standard-Variante die zusätzliche Beschränkung, dass auf den beiden Hauptdiagonalen ebenfalls die Zahlen 1 bis 9 nur einmal vorkommen dürfen. Abbildung 7.4 zeigt diesen Zusammenhang. Die betreffenden Felder sind mit einem grauen Hintergrund versehen.

7.2.2 Hypersudoku

Hypersudoku verhält sich wie das Standard-Sudoku, besitzt aber zusätzlich zu den neun Regionen noch vier weitere, für die ebenfalls die gleichen Beschränkungen gelten. In Abbildung 7.5 sind die zusätzlichen Regionen mit einer an-

1	2	3	7	8	9	4	5	6
4	5	6	1	2	3	7	8	9
7	8	9	4	5	6	1	2	3
2	3	1	8	9	7	5	6	4
5	6	4	2	3	1	8	9	7
8	9	7	5	6	4	2	3	1
3	1	2	9	7	8	6	4	5
6	4	5	3	1	2	9	7	8
9	7	8	6	4	5	3	1	2

Quelle: [Sud]

Abbildung 7.4: X-Sudoku Variante

deren Hintergrundfarbe abgesetzt.

9	4	6	8	3	2	7	1	5
1	5	2	6	9	7	8	3	4
7	3	8	4	5	1	2	9	6
8	1	9	7	2	6	5	4	3
4	7	5	3	1	9	6	8	2
2	6	3	5	4	8	1	7	9
3	2	7	9	8	5	4	6	1
5	8	4	1	6	3	9	2	7
6	9	1	2	7	4	3	5	8

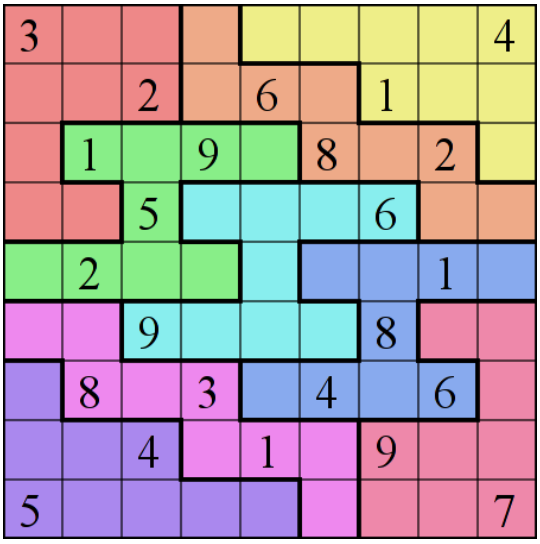
Quelle: [Sud]

Abbildung 7.5: Hyper-Sudoku Variante

7.2.3 Nonomino-Sudoku

Nonomino ist eine Sudoku Variante, welche ebenfalls in 81 Felder aufgeteilt ist und in jeder Zeile und Spalte dürfen die Zahlen 1 bis 9 jeweils nur einmal vorkommen. Jedoch bestehen die insgesamt neun Regionen aus unregelmäßig geformten Nonominos. Ein Nonomino ist eine Fläche von neun zusammenhän-

genden Quadraten. Abbildung 7.6 zeigt dies exemplarisch.



Quelle: [Sud]

Abbildung 7.6: Nonomino-Sudoku Variante

7.2.4 Samurai-Sudoku

Samurai-Sudoku ist eine Sudoku Variante, die aus mehreren Standard-Sudoku Gittern besteht die sich zum Teil überlappen. Im Kapitel 5 wurde als Eingabe ein Samurai-Sudoku bestehend aus fünf Standard-Sudokus verwendet. Abbildung 7.7 zeigt dies exemplarisch.

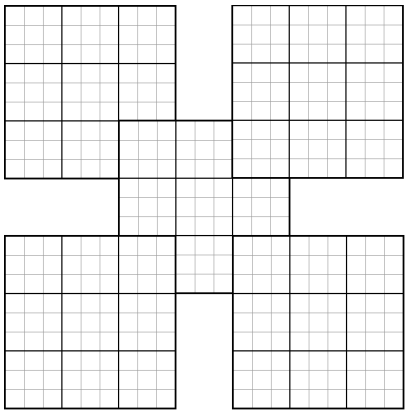


Abbildung 7.7: Samurai-Sudoku Variante

Literaturverzeichnis

Bac02 Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *AAAI/IAAI*, pages 613–619, 2002. 1.1, 3.3

Bir88a Richard Bird. *Introduction to functional programming.* Prentice Hall, 1988. 2.2

Bir88b Richard Bird. *Introduction to functional Programming using Haskell.* Prentice Hall, 1988. 2.2

CC07 Priyank Kalla Christopher Condrat. A gröbner basis approach to cnf-formulae preprocessing. *TACAS'07 Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, 2007. 3.2

Jun04 Achim Jung. A short introduction to the lambda calculus. 2004. 2.2.1

KJ06 Gihwon Kwon and Himanshu Jain. Optimized cnf encoding for sudoku puzzles. In *In 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 2006. 3.3.2.3

Lau12 Thomas Lauszus. *Einführung in die Aussagenlogik.* Grin Verlag, 2012. 2.1

Lip11 Miran Lipovaca. *Learn You a Haskell for Great Good!* No Starch Press, 2011. 2.2.3

MD60 Hilary Putnam Martin Davis. A computing procedure for quantification theory. *The Journal of Symbolic Logic*, 7:201–215, 1960. 3, 3.2.3

MD62 Donald Loveland Martin Davis, George Logemann. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962. 3, 3.2.3

MG01 Simon Marlow and Andy Gill. Dokumentation "happy user guide". online ressource, April 2001. http://www.haskell.org/happy/doc/html/index.html. 4.1

OG98 Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *In Workshop on ML*, pages 77–86, 1998. 2.2.4, 4.3

PP06 Petra Hofstedt Peter Pepper. *Funktionale Programmierung*. Springer Verlag Berlin Heidelberg, 2006. 2.2

Sab93 Amr Sabry. What is a purely functional language? 1993. 2.2.1

San11 Tian Sang. *Complete Search Algorithms for Model Counting, Inference, and Optimization Problems*. Proquest, Umi Dissertation Publishing, 2011. 3.2.3

SS09 Prof. Dr. Manfred Schmidt-Schauß. Einführung in die funktionale programmierung, Wintersemester 2009. 2.2.3

SS11 Prof. Dr. Manfred Schmidt-Schauß. Einführung in die methoden der künstlichen intelligenz kaptitel aussagenlogik, Sommersemester 2011. 2.1, 3.2

Sud Sudoku. Online Artikel: http://de.wikipedia.org/wiki/Sudoku, http://en.wikipedia.org/wiki/Sudoku. 7.2.1, 7.2.2, 7.2.3

uES Konrad Schlude und Ernst Specker. Zum problem der damen auf dem torus. In *ETH Zürich*. 7.1.2

ZS00 Hantao Zhang and Mark E. Stickel. Implementing the davis-putnam method. *Journal of Automated Reasoning*, 24:277–296, 2000. 1.1, 3.3