Fachbereich 12 Informatik und Mathematik
Institut für Informatik

Diplomarbeit

Implementierung von Varianten des Davis-Putnam-Logemann-Loveland Verfahrens in Haskell

Said Sadegh Nezhadian

12. Mai 2009

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie

Danksagung

Ich bedanke mich ganz herzlich bei all denen, die mich während der Entstehung dieser Arbeit begleitet und unterstützt haben.

Mein besonderer Dank gilt Herrn Dr. David Sabel und Herrn Prof. Dr. Manfred Schmidt-Schauß für Ihre ausgezeichnete Betreuung und Ihre unzähligen Anregungen.

Außerdem möchte ich mich bei meinen Eltern bedanken, die mir dieses Studium ermöglicht haben.

Said Sadegh Nezhadian

Erklärung gemäß DPO §11 Abs. 11

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Frankfurt am Main, den 12. Mai 2009

Said Sadegh Nezhadian

Inhaltsverzeichnis

Abkürzungsverzeichnis

CBJ	conflict-directed-backjumping
CNF	conjunctive normalform
DAG	directed acyclic graph
DPLL	Davis-Putnam-Logemann-Loveland
EDA	Electronic Design Automation
FPS	Funktionale Programmiersprache
gdw	genau dann wenn
GHC	Glasgow Haskell Compiler
KI	Künstliche Intelligenz
SAT-Problem	Satisfiability-Problem
SMT	SAT Modulo Theorien
UIP	unique implication point

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Das *Erfüllbarkeitsproblem der Aussagenlogik* (kurz *SAT-Problem*), d.h. das Problem, zu einer gegebenen Formel der Aussagenlogik zu entscheiden, ob sie erfüllbar ist oder nicht, liegt nicht nur im Kern der wichtigsten offenen Frage der Komplexitätstheorie ($\mathcal{P}$ vs. $\mathcal{NP}$), sondern sie hat ebenso eine herausragende Bedeutung für die Praxis. Das interessante am SAT-Problem ist, dass sich viele $\mathcal{NP}$-vollständige Probleme aus praktischen Anwendungen, wie auf dem Gebiet der Entwurfsautomatisierung und Verifikation in der Elektronik (EDA), oder bei Planungsproblemen in der Künstlichen Intelligenz (KI) [KS92], im Bereich des Model-Checkings [BCCZ99, McM02] und dem Operation Research - um nur einige zu nennen - nach SAT übersetzen und aufgrund neuester Fortschritte im Bereich von SAT-Algorithmen (auch *SAT-Solver* genannt) auf Erfüllbarkeit überprüfen lassen (siehe dazu Abbildung 1.1).

Obwohl das SAT-Problem 1971 von Thomas Cook als $\mathcal{NP}$-vollständig nachgewiesen wurde [Coo71], sind in den letzten Jahren SAT-Solver entwickelt worden, die Dank neuester Fortschritte Formeln mit vielen tausend und in einigen Fällen sogar millionen Variablen routinemäßig auf Erfüllbarkeit hin überprüfen können. Erst diese Algorithmen erlauben die Anwendung des SAT-Problems im industriellen Maßstab. Die meisten dieser SAT-Solver basieren im Kern auf Prinzipien, wie Resolution und Subsumption [DP60], und der Suche gemäß dem Davis-Putnam-Logemann-Loveland-Algorithmus (DPLL) [DLL62] aus den 60er Jahren. Ihre Leistungsfähigkeit verdanken die modernen SAT-Solver in erster Linie einem sogenannten *nicht-chronologischen-backtracking* Verfahren [MSS99] sowie zahlreichen *Entscheidungsheuristiken* [Ms99] und zum anderen *effiziente Datenstrukturen* [LMS05].

1.2 Zielsetzung

Die auf den modernsten Stand der Technik befindlichen SAT-Solver basieren auf einer modifizierten Variante des *klassischen DPLL* Verfahrens [DLL62]. Im Rahmen dieser Diplomarbeit werden die konzeptuellen Erweiterungen an dem klassischen DPLL Verfahren vorgestellt und in der Programmiersprache Haskell implementiert.

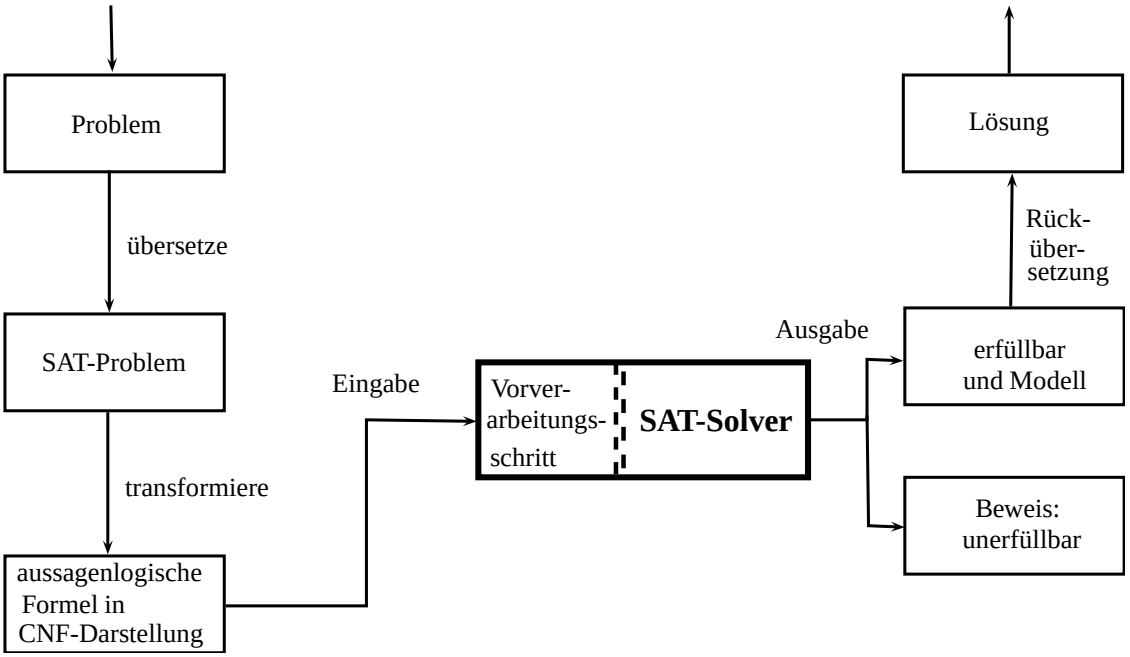

Abbildung 1.1: Das allgemeine Schema für die Verwendung von SAT-Solvern

Die Implementierung soll ihren Abschluss letztendlich in verschiedenen Varianten eines *modernen DPLL* Verfahrens finden, welche auf einem *abstrakten DPLL Framework* [NOT06] beruhen, das es ermöglicht, durch eine regelbasierte Formulierung moderne DPLL Algorithmen eindeutig auszudrücken und formal zu beweisen. Neben den wesentlichen Verbesserungen in der Performance, die durch die Verwendung von *backjumping* (eine Form des *nicht-chronologischen-backtrackings*) und *conflict-driven-learning* [MSS99] sowie *Entscheidungsheuristiken* erzielt werden sollen, trägt der deklarative Charakter der funktionalen Programmiersprache (FPS) wie Haskell dazu bei, dass der Algorithmus nicht wie in imperativen Sprachen aus einer Sequenz von Anweisungen besteht, sondern eher aus einer formalen Spezifikation des Ergebnisses. Dadurch wird es besser lesbar und ist schneller zu verstehen und somit auch leichter zu warten. Mit Haskell soll zudem die Erweiterbarkeit des Programms zusätzlich erleichtert werden. Ebenso lässt sich der Algorithmus in Haskell schneller und leichter schreiben, da man auf intuitive Weise den Algorithmus strukturieren und in Teilprobleme aufteilen kann, was bei den immer komplexer werdenden Erweiterungen an dem klassischen DPLL Algorithmus ein enormer Vorteil ist, zumal das Programm auch deutlich kürzer ausfallen wird als vergleichbare imperative Programme.

1.3 Überblick

Zunächst wird *Kapitel 2* auf die Merkmale von funktionalen Programmiersprachen und ihre Unterschiede zu imperativen Sprachen eingehen. Anschließend werden die Vorteile und auch die wenigen Nachteile, die sich aus der Verwendung einer funktionalen Sprache wie Haskell ergeben, zusammengefasst. Da die Implementierung in der Programmiersprache Haskell erfolgen soll, werden im Anschluss daran einige charakteristische Programmiertechniken von Haskell vorgestellt.

In *Kapitel 3* wird eine formale Definition der Aussagenlogik angegeben, nachdem das SAT-Problem anhand eines einfachen Beispiels eingeführt wurde, denn das Ziel dieses Programms ist es gerade das Erfüllbarkeitsproblem der Aussagenlogik (SAT-Probleme) zu lösen. Zunächst jedoch sollen einfache Verfahren angegeben werden, mit deren Hilfe sich SAT-Probleme auf intuitive Art und Weise auf Erfüllbarkeit prüfen lassen. Anschließend wird das SAT-Problem aus der Sicht der heutigen Praxis und Theorie beleuchtet.

In *Kapitel 4* wird eine Basis Terminologie eingeführt, auf der dann die weiteren Ausführungen basieren. Weiterhin wird der klassische DPLL Algorithmus als Grundgerüst für die nachfolgende Implementierung des modernen DPLL Algorithmus angegeben. Das Kapitel wird abgeschlossen durch eine formale Einführung in das *abstrakte DPLL Framework*.

Nachdem somit der klassische DPLL Algorithmus als Grundgerüst und seine Definition anhand des abstrakten DPLL Frameworks erarbeitet wurde, werden in *Kapitel 5* die erweiterten Komponenten im modernen DPLL Algorithmus zunächst ausführlich beschrieben und deren Erweiterungen anschließend mit Hilfe des vorgestellten Frameworks formal definiert. Das abstrakte DPLL Framework des modernen DPLL Algorithmus dient dann in Kapitel 6 als Basis für die Implementierung in Haskell.

Kapitel 6 widmet sich schließlich der Implementierung von Varianten des DPLL Algorithmus. Hier geht es darum die zugrundeliegende Datenstruktur zu erklären und den Aufbau des Programms mit dessen wichtigsten Besonderheiten auch bezüglich möglicher Veränderungen und Erweiterungen aufzuzeigen.

Alle implementierten Varianten des modernen DPLL Algorithmus werden dann in *Kapitel 7* anhand umfangreicher Benchmark-Tests überprüft und deren Ergebnisse tabellarisch ausgewertet.

Die Diplomarbeit schließt *Kapitel 8* mit einer kurzen Zusammenfassung und einem Ausblick auf zukünftige Aufgaben ab.

2 Grundlagen

2.1 Funktionale Programmiersprachen

Dieses Kapitel soll eine kurze Einführung in die wesentlichen Konzepte von funktionalen Programmiersprachen geben. Dabei wird auf deren grundlegende Merkmale - ähnlich wie in [Sch09] - eingegangen. Da im Rahmen dieser Diplomarbeit Haskell als funktionale Programmiersprache verwendet wird, werden im Anschluss daran einige Programmiertechniken von Haskell angegeben.

2.1.1 Das funktionale Konzept

Programmiersprachen lassen sich grob in *imperative* und *deklarative* Sprachen unterteilen. Imperative Programmiersprachen kennzeichnen sich durch die Abarbeitung einer Folge von Anweisungen zum Verändern von Werten. Hierbei geht es darum „WIE“ ein Problem gelöst werden soll, im Gegensatz zu deklarativen Programmiersprachen, bei der weitgehend vom WIE abstrahiert und stattdessen das gewünschte Ergebnis beschrieben wird, „WAS“ durchzuführen ist. Die Abstraktion erfolgt mit Hilfe einer *Funktion*, wie in Abbildung 2.1 als Blackbox bildlich dargestellt. Funktionale und logische Programmiersprachen zählen zu den deklarativen Programmiersprachen. Funktionale Programme bestehen aus *Funktionen* und Anwendungen von

Abbildung 2.1: Funktion als Blackbox

Funktionen auf *Argumente*, wobei eine Funktion ihr *Resultat* in Abhängigkeit ihrer Argumente liefert:

$$
\begin{array}{llll}
f_1 & x_{11} \cdots x_{sn} & = & e_1 \\
\vdots & & & \vdots \\
\underbrace{f_s} & \underbrace{x_{s1} \cdots x_{sn}} & = & \underbrace{e_s} \\
\text{Funktions-} & \text{Parameter-} & & \text{Rumpf-} \\
\text{bezeichner} & \text{variablen} & & \text{ausdruck} \\
 & & & \\
 & \text{„Argumente“} & & \text{„Resultat“}
\end{array}
$$

Die Ausführung eines Programms in funktionalen Programmiersprachen besteht im Auswerten von Ausdrücken durch *Reduktion*[1], wobei Funktionen als Ausdrücke verwendet werden, um bestimmte Probleme zu modellieren. Ein funktionales Programm geht dabei wie folgt vor:

- die Funktionsdefinitionen werden als Ersetzungsregeln definiert, außerdem gelten die Festlegungen für vordefinierte Operationen[2],
- eine Funktion wird durch den Rumpf der entsprechenden Definition ersetzt,
- es folgt die Substitution der formalen Parameter durch die aktuellen Parameter und
- für die Funktionsdefinition $f\ x_1 \cdots x_n = e$ erfolgt mit
  $f\ e_1 \cdots e_n \Rightarrow e[e_1/x_1, \cdots, e_n/x_n]$
  die Substitution der Variablen durch Argumentausdrücke.

Die wesentlichen Elemente von funktionalen Programmiersprachen sind Funktionsaufrufe, denen die Theorie des sogenannten λ-*Kalküls* von Alonzo Church und Stephen Kleene aus den 30er Jahren zugrunde liegt [MM97]. Wesentlich für eine FPS ist auch die Komposition von Funktionen in einzelne Module, d.h. die Zerlegung des eigentlichen Problems in kleinere Teile, und ein rekursiver Programmierstil, der Schleifen aus imperativen Programmiersprachen ersetzt.

2.1.2 Referentielle Transparenz

Eine weitere Besonderheit von *puren* funktionalen Programmiersprachen ist, dass diese ohne Variablenzuweisungen auskommen; das wiederum führt dazu, dass keine *Seiteneffekte* auftreten, was die Eigenschaft der *referentiellen Transparenz* zur Folge

[1] ersetze in einem Ausdruck f solange Teilausdrücke durch ihre Definition, oder berechne den Wert von Teilausdrücken, in denen nur elementare Funktionen vorkommen, bis keine Reduktion mehr möglich ist

[2] diese könnten z.B. elementare Hilfsfunktionen, wie die Addition oder Multiplikation sein

hat. Im Gegensatz dazu stehen *impure* funktionale Programmiersprachen, die Seiteneffekte eingeschränkt zulassen. Das Prinzip der referentiellen Transparenz kann analog zu [Sch09] wie folgt definiert werden: „die gleiche Funktion angewendet auf die gleichen Argumente, ergibt das gleiche Resultat“, d.h. nur die Argumente bestimmen den Wert, und die Funktion verändert sich nicht durch die Reihenfolge ihrer Verarbeitung, was eine Wiederverwendung von Funktionalitäten dadurch leicht möglich macht. Wir schauen uns an dieser Stelle einmal das Beispiel 2.1.1 an, in der die Funktion `square` das Quadrat einer Zahl berechnet. Weiterhin verwendet die Funktion `square` die Multiplikation `(*)` und die Addition `(+)` zweier Zahlen als vordefinierte elementare Hilfsfunktionen. Die Ersetzung der formalen Parametervariablen `x` durch ihre Definition `(3+4)` kann in beliebiger Reihenfolge (auch parallel) erfolgen.

Eine mögliche Auswertung von `square (3+4)` wäre, wie in a), zuerst die vordefinierte Operation `(+)` im Argument `(3+4)` von `square` auszuwerten, um anschließend die Funktion `square` darauf anzuwenden:

Beispiel 2.1.1. Sei `square x = x * x`.

a) `square (3+4)`	Def. `+`	`square 7`
	Def. `square`	`7 * 7`
	Def. `*`	`49`
b) `square (3+4)`	Def. `square`	`(3+4) * (3+4)`
	Def. `+`	`7 * (3+4)`
	Def. `+`	`7 * 7`
	Def. `*`	`49`

Würde man jedoch, wie in b), zuerst die Funktion `square` durch den Rumpf der Definition ersetzen, und im Anschluss das Argument der Funktion auswerten, erhält man das gleiche Resultat.

Das obige Beispiel verdeutlicht gleichzeitig das *Substitutionsprinzip* als eine weitere Eigenschaft, nämlich, dass Ausdrücke mit gleichem Wert im Programmtext ohne Konsequenz für den Wert des Gesamtausdrucks ausgetauscht werden können. Im konkreten Fall hieße das `(3+4)` mit `7` auszutauschen.

Darüber hinaus existieren einige Unterscheidungskriterien, die funktionale Programmiersprachen voneinander abgrenzen. Diese werden nun im Folgenden kurz erläutert.

2.1.3 Typsysteme

Die Aufgabe von *statischen* Typsystemen in Programmiersprachen ist es, ein Programm schon zur Compilezeit zu verifizieren. Ein Compiler bzw. Interpreter überprüft dabei die Korrektheit eines Programms nicht nur anhand von syntaktischen Regeln, wie sie etwa durch eine kontextfreie Grammatik vorgegeben ist, sondern auch anhand der Korrektheit der verwendeten Typen. Somit können Fehler, wie zum Beispiel die Vertauschung von zwei Parametern in einem Funktionsaufruf frühzeitig ermittelt werden.

Es kann jedoch je nach Typsystem sein, dass erst zur Laufzeit des Programms (*dynamisch*) die Korrektheit der verwendeten Typen verifiziert wird. Bei Typsystemen mit *dynamischer* Typüberprüfung, wie zum Beispiel in der Programmiersprache Java, lässt sich das Laufzeitverhalten zur Compilezeit nicht eindeutig voraussagen, da die Bindungen erst während der Laufzeit feststehen. Zur Laufzeit kann es so zu störenden Typfehlern kommen.

Bei *stark* statischen Typsystemen werden Typfehler vom Compiler nicht toleriert, und alle Ausdrücke sind getypt. Im Gegensatz zu *schwach* statischen Typsystemen, in der manche Ausdrücke ungetypt sind, und der Compiler Typfehler toleriert. Wenn Funktionen einen festen Typ haben, dann spricht man zusätzlich von einem *monomorphen* Typsystem.

Oftmals gibt es Funktionen, deren Algorithmus nicht von einem speziellen Datentyp abhängig ist, sondern mit jedem beliebigen Datentyp arbeiten kann. So ist zum Beispiel das Ermitteln der Länge einer Liste unabhängig davon, mit welchem konkreten Datentyp die Liste aufgebaut ist. Es ist nur nötig, die Anzahl der Listenelemente zu zählen. Ein solcher Datentyp ist dabei ein Stellvertreter für einen beliebigen, aber noch unbestimmten, konkreten Datentyp in einem sogenannten *polymorphen* Typsystem.

2.1.4 Status der Objekte

Funktionale Programmiersprachen lassen sich in Sprachen *erster Ordnung* und Sprachen *höherer Ordnung* unterteilen. In Sprachen erster Ordnung dürfen Argumente und Resultate nur Datenobjekte und keine Funktionen sein, d.h. solche Sprachen erlauben nur die Definition und den Aufruf von Funktionen, im Gegensatz zu Sprachen höherer Ordnung die zusätzlich Funktionen als Objekte zulassen, somit lassen sich für Argumente und Resultate wiederum Funktionen einsetzen.

2.1.5 Auswertungsstrategien

Ein weiteres Unterscheidungskriterium innerhalb funktionaler Programmiersprachen, ist ihre *Auswertungsstrategie*. Unter Auswertungsstrategie versteht man einen Algorithmus zur Auswahl des nächsten zu reduzierenden Teilausdrucks e (*Redex, reducable expression*).

Bei *strikter Auswertung* (engl. *strict* oder *eager evaluation*) wird stets der am weitesten links innen im Ausdruck vorkommende Redex ausgewertet. Das bedeutet, dass Argumente einer Funktion ausgewertet werden, bevor die Argumente in den Rumpf der Funktion eingesetzt werden (*call-by-value*).

Bei einer *nicht-strikten Auswertung* wird stets der am weitesten außen im Ausdruck vorkommende Redex gewählt. Die Argumente von Funktionen sind hier in der Regel unausgewertete Ausdrücke. Es werden nur diejenigen Ausdrücke ausgewertet, deren Wert zum Ergebnis beiträgt (*call-by-name*).
Eine verbesserte Variante der nicht-strikten Auswertung ist die *verzögerte Auswertung* (engl. *lazy evaluation*): Ausdrücke, die zum Ergebnis beitragen und mehrmals auftreten, werden nur einmal ausgewertet (*call-by-need*). Damit kombiniert diese Strategie die Vorteile von call-by-value und call-by-name.

2.1.6 Zusammenfassung

Eine passende Kategorisierung von funktionalen Programmiersprachen gemäß der genannten Kriterien, liefert die Tabelle aus [Rec06]:

		eager evaluation	lazy evaluation
dynamisch typisiert	erster Ordnung	Mathematica, Erlang	
	höherer Ordnung	Lisp, Scheme, APL, FP	SASL
statisch typisiert	erster Ordnung	SISAL	Id
	höherer Ordnung	ML, SML, Caml, Hope	Miranda, Haskell

Tabelle 2.1: Kategorisierung von funktionalen Programmiersprachen

Es sollen nun einige Gründe analog zu [Sch09] aufgezählt werden, warum Haskell als funktionale Programmiersprache ausgewählt wurde. Ebenso sollen auch die wenigen Nachteile nicht verschwiegen werden.

Vorteile:

- Nicht-strikte FPS erlauben die saubere Behandlung von Funktionen höherer Ordnung (Funktionen als Argumente, und als manipulierbare Objekte).
- Die Semantik erlaubt sehr leicht zahlreiche korrekte Programmtransformationen ohne sich um die Nebenbedingungen zu kümmern, ob Seiteneffekte auftreten, oder ob der Aufruf terminiert. Das bedeutet, dass Programmtransformationen einen einfachen Begriff von Korrektheit haben, der auch praktisch dazu führt, dass man Programme in korrekter Weise analysieren, optimieren und verändern kann, ohne Fehler einzuführen.
- Parallelisierbarkeit ist durch einfache Analysen zu erkennen.
- Das Typsystem, parametrisch polymorph und die Erweiterung zum Typklassensystem ist sehr gut durchdacht und sehr weitgehend, so dass der Programmierstil und das Typsystem dazu führt, dass zahlreiche Fehler bereits in der Programmierphase vermieden werden.
- Die aktive Forschergemeinde ist ziemlich groß, so dass in diesem Bereich noch wegweisende Entwicklungen zu erwarten sind.

Nachteile:

- Effizienz zur Laufzeit erfordert hohen Aufwand beim Schreiben eines Compilers.
- Die Schnittstellen zu GUIs, dem Betriebssystem und anderen IO-Medien sind unhandlich, da hier zwei verschiedene Prinzipien nicht ganz passen: Seiteneffekte, Sequentialität auf der einen, und Seiteneffektfreiheit sowie deklarative, nichtsequentielle Spezifikation auf der anderen Seite. Dies wird teilweise durch das monadische Programmierkonzept[3] wieder ausgeglichen.

2.2 Haskell

Haskell ist eine Programmiersprache mit einer verzögerten (nicht-strikten) Auswertung und einem stark statischen, polymorphen Typsystem, die Funktionen höherer Ordnung erlaubt und keine Seiteneffekte zulässt. Sie ist benannt nach dem Mathematiker und Logiker Haskell Brooks Curry (1900). Sie ist ursprünglich aus der ebenfalls nicht-strikten Programmiersprache Miranda (1985) hervorgegangen, als das Interesse an solchen Sprachen wuchs. Zu diesem Zeitpunkt existierten bereits dutzende nicht-strikte, pure funktionale Programmiersprachen, von denen Miranda die am

[3] siehe IO-Monade

weitesten verbreitete, jedoch nicht für jedermann zugängliche Sprache war.

Auf der Konferenz für funktionale Programmiersprachen und Computer Architektur (FPCA '87) in Portland, Oregon, kamen die Teilnehmenden zu der Übereinstimmung, einen öffentlichen Standard für diese Art von Sprachen zu definieren. Die Motivation lag darin, ein stabiles Fundament für die Entwicklung realer Anwendungen und ein Werkzeug zu schaffen, mit der andere motiviert würden funktionale Programmiersprachen zu verwenden. Die Anstrengungen des Komitees führte zu einer Serie von Sprachdefinitionen, die ihren Abschluss letztendlich in der *Haskell 98* Standard Sprache und seiner Veröffentlichung im Jahre 1999 in „The Haskell 98 Report" [PeHeA+99] fand. Einige charakteristische Programmiertechniken von Haskell wie *Modularisierung*, *Datentypen*, *Rekursion*, *Pattern matching*, *Guards* und *List comprehension* sowie das besondere feature der *unendlichen Listen* werden nun im weiteren Verlauf näher erläutert.

2.2.1 Modularisierung

In Haskell werden Programme aus Modulen gebildet. Gewöhnlich besteht ein umfangreiches Programm aus einer Vielzahl von Funktions- und Typdefinitionen. Es ist dabei nicht ratsam, alle diese Definitionen in einer einzigen Datei zu speichern. Haskell bietet daher, wie jede andere moderne Programmiersprache die Möglichkeit zur Strukturierung von Programmen, indem sie zulässt, dass zusammengehörende Definitionen in logische Einheiten zusammengefasst werden. In Haskell werden diese Einheiten *Module* genannt (siehe [PeHeA+99] Kapitel 5). Somit besteht ein Haskell-Programm aus einem oder mehreren Modulen. Das Hauptmodul muss den Namen `Main` tragen, welches den Wert (Funktion) von `main` exportiert, d.h. nach außen hin sichtbar macht. Datentypen, Werte, Klassen, etc. können mit dem Schlüsselwort `import` zusätzlich in den Gültigkeitsbereich eines Moduls mit aufgenommen werden. Ein Grundgerüst für das Hauptmodul hätte dann beispielsweise die Form:

```
1 module Main(main) where
2  import A
3  import B
4 main = ...
```

wobei die Module `A` und `B` mit unterschiedlichen Funktionalitäten (repräsentiert durch verschiedene Funktionen `f` und `g` innerhalb der Module) aus jeweils verschiedenen Dateien importiert werden können.

```
1 module A where
2   f = ...
```

und

```
module B where
  g = ...
```

2.2.2 Funktionen und Datentypen

Funktionen beschreiben Zuordnungen zwischen Eingabe- und Ausgabewerten. So ordnet beispielsweise die Funktion `square`, Werten vom Typ `Int` wiederum Werten vom Typ `Int` zu, wie in Abbildung 2.2 bildlich dargestellt:

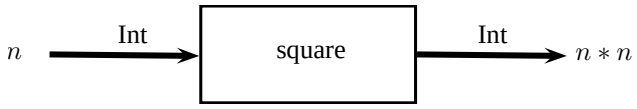

Abbildung 2.2: Funktion square

Eine vollständige Funktionsdefinition von `square` wird in Haskell dabei wie folgt definiert:

```
square :: Int → Int  -- Typsignatur
square n  = n * n    -- Funktionsgleichung
```

Typsignaturen dokumentieren die Funktionen für andere Programmierer und erleichtern es dem Haskell-System, Typfehler zu finden (Hinweis: mit zwei aufeinander folgenden Bindestrichen werden im Quelltext Kommentare eingeleitet).

Falls Funktionen nicht nur auf einen bestimmten Typ festgelegt werden sollen, wie es bei sogenannten *Überladenen Funktionen* der Fall ist, können *Typklassen* definiert werden (siehe [PeHeA+99] Abschnitt 4.3), die dann auf einer Menge von Typen operieren dürfen. So beispielsweise die Funktion (+), die sowohl auf dem Typ

```
(+)   :: Int → Int → Int
```

als auch auf dem Typ

```
(+)   :: Float → Float → Float
```

definiert ist. In der Tat sind arithmetische Operationen wie (+) und (*) nicht nur auf `Int` und `Float` beschränkt, sondern sie sollten für alle numerische Typen gelten. Daher lässt sich in Haskell z.B. die Funktion (+) für alle Typen der numerischen Werte (Klasse `Num`) wie folgt definieren:

```
(+)   :: Num a ⇒ a → a → a
```

Datentypen werden gemäß [PeHeA+99] Abschnitt 4.2 mit Hilfe des `data`-Konstrukts definiert. Wenn man etwa Boolesche Werte als neuen Datentyp definieren möchte, würde man dabei wie folgt vorgehen:

```
1 data Bool             =  False | True
```

Hierbei ist `Bool` der neue *Datentyp* sowie `False` und `True` die *Datenkonstruktoren*. Es handelt sich dabei insbesondere um einen *Aufzählungstyp*, der bereits in Haskell vordefiniert ist.

Des Weiteren sind auch *Tupeltypen* möglich, wie das nächste Beispiel zeigt:

```
1 data Point            =  Point Int Int
```

Hierbei ist jeder Datentyp zusätzlich mit einem *Konstruktor* versehen, wie etwa `Point` in diesem Beispiel. Man hat auch die Möglichkeit einen Konstruktor rekursiv zu definieren, wie das Beispiel ebenfalls zeigt.

Datentypen können parametrisiert werden, wie etwa durch das `a` im Beispiel des *Vereinigungstyps* `Maybe`:

```
1 data Maybe a          =  Nothing | Just a
```

Mit Hilfe des Schlüsselworts `type` lassen sich sogenannte *Typsynonyme* definieren. Sie werden verwendet um „große“ Typausdrücke abzukürzen und um die Lesbarkeit des Programms durch Verwendung aussagekräftiger Typnamen zu erhöhen, so z.B.:

```
1 type List = []
```

Bei der Namensgebung ist unschwer zu erkennen, dass es sich bei diesem Typ um eine Liste handeln soll. Beachte: im Gegensatz zu Datentypdefinitionen dürfen Typsynonyme *nicht* rekursiv definiert werden.

2.2.3 Rekursion

Wie bereits eingangs erwähnt, gibt es keine Programmschleifen in funktionalen Programmiersprachen, sie müssen stattdessen durch Rekursion ersetzt werden. Eine rekursive Funktion ist eine Funktion, die sich selbst wieder in ihrer eigenen Definition aufruft. Das nächste Beispiel zeigt eine typisch rekursive Funktionsdefinition:

```
1 factorial :: Int → Int
2 factorial 0 = 1                     -- Basis
3 factorial n = n * factorial (n - 1) -- Rekursionsschritt
```

Die Funktion `factorial` berechnet die Fakultät einer beliebigen Zahl `n`. Dabei ruft sich die Funktion im *Rekursionsschritt* selbst wieder im Funktionsrumpf auf. Die Abbruchbedingung wird in der *Basis* festgelegt. Es sei an dieser Stelle ebenfalls erwähnt, dass es für Haskell durchaus nicht unüblich ist, dass es Definitionen gibt, wie etwa `factorial`, die sehr ihren Definitionen aus der Mathematik ähneln. Manche mathematische Definitionen lassen sich sogar fast eins zu eins in Haskell übertragen.

2.2.4 Pattern matching

Eine weitere wichtige Programmiertechnik in Haskell ist das sogenannte *Pattern matching*[4]. Bevor sich ein Beispiel mit Listen diesem zuwidmet, werden Listen noch kurz eingeführt.
Listen ([PeHeA+99] Abschnitt 3.7) sind in funktionalen Programmen die zentrale Datenstruktur gemeinhin, sie stellen Folgen von Werten des gleichen Typs dar, so stellt z.B. `[Int]` eine Liste von Integern dar. Im Folgenden bezeichnet `[]`[5] die leere Liste (kennzeichnet das Listenende), und ein Doppelpunkt `(:)`[6] bezeichnet den Listenkonstruktor. Eine Liste entsteht durch das konkatenieren der Listenelemente mit Hilfe des (:) Operators:

$$[x_1, x_2, x_3, \cdots, x_n] = x_1 : (x_2 : (x_3 : \cdots : (x_n : [\,])))$$

Wegen des wiederholten Aufrufens von (:), wird bei der Bildung von Listen insbesondere die rekursive Struktur von Listen deutlich.

Es soll nun im folgenden eine Funktion definiert werden, die alle Elemente einer Liste addiert:

```
1 sum :: [Int] → Int
2 sum []     = 0
3 sum (x:xs) = x + sum xs
```

Die Definition bzw. das Ergebnis der Funktion `sum` hängt nun vom aufrufenden Parameter ab. Wenn `sum` mit der leeren Liste `[]` als Parameter aufgerufen wird, so wendet Haskell die erste Definition an, und wenn die Liste nicht leer ist, die zweite. Im letzteren Fall wird das erste Element der Liste an die Variable `x` gebunden und der Rest der Liste an die Variable `xs`. Pattern matching führt zu der Auswahl des „passenden“ Definitionsfalls für die auszuwertende Funktion, vorausgesetzt es handelt sich um den selben Funktionsnamen. Das Beispiel ist ein einfacher Fall, in der das Pattern matching in Haskell auftritt. Pattern matching ist jedoch weitaus

[4] engl. für Mustervergleich
[5] gesprochen als „nil“
[6] gesprochen als „cons“

mächtiger, da auch geschachtelte Pattern möglich sind. Sie sollen aber hier nicht weiter erläutert werden.

2.2.5 Guards

Eine weitere Möglichkeit Alternativen für Funktionsdefinitionen anzugeben, liefern uns die sogenannten *Guards*[7]. Ein einfaches Beispiel nach [Tho97] soll uns die Funktionsweise von Guards näher erläutern:

```
1 max :: Int → Int → Int
2 max x y
3  | x >= y   = x
4  | otherwise = y
```

Diese Funktion liefert das Maximum zweier Integer. Guards werden durch einen „pipe“[8] eingeleitet. Dabei prüft jeder Guard, ob seine Bedingung erfüllt ist, wenn ja, wird der Wert entsprechend der Definition gesetzt, wenn nein, wird die Bedingung des nächsten Guards geprüft, d.h. der erste Guard prüft zuerst und der letzte kommt zum Schluss. Es ist darauf zu achten, dass mit den Guards alle möglichen Bedingungen abgedeckt werden, da es sonst zu einer Fehlermeldung kommt. Ein Guard, der den Rest abfragt, wird durch das Schlüsselwort `otherwise` eingeleitet. `otherwise` ist dabei definiert als `otherwise` = true, d.h es ist nur ein Synonym für true und kein spezielles Konstrukt. Guards sind ähnlich zu `if` ... `then` ... `else` Konstrukten in imperativen Programmiersprachen. In Haskell ließe sich die obige Funktion auch wie folgt schreiben:

```
1 max :: Int → Int → Int
2 max x y = if x ≥ y then x
3           else y
```

2.2.6 List comprehension

Der Begriff *Comprehension* stammt aus der Mathematik (speziell aus der Mengenlehre). In Haskell dienen *List Comprehension* der Konstruktion komplexer Listen aus vorhandenen anderen Listen unter Verwendung von Prädikaten. Ein einfaches Beispiel einer Comprehension in der Mengenlehre wäre:

$$M = \{x^2 \mid x \in \{1..100\} \wedge x \bmod 2 = 0\}$$

und dasselbe Beispiel als List Comprehension geschrieben in Haskell:

[7] engl. für Wächter
[8] senkrechter Strich

```
m = [ x^2 | x ← [1..100], x `mod` 2 = 0]
```

Gemäß [PeHeA+99] Abschnitt 3.11 setzen sich List Comprehension zusammen aus:

- *Generatoren* als Mengen und
- *Selektoren* als Prädikate

Eine List Comprehension hat dabei die Form $[expr \mid q_1, ..., q_n]$, wobei jedes der q_i mit $1 \leq i \leq n$ entweder

- ein Generator der Form `pat <- listexp` ist, und `pat` dabei ein Muster mit dem Typ `a` und `listexp` eine Liste vom Typ `[a]`,
- eine Bedingung (Guard), d.h. ein beliebiger Ausdruck vom Typ `Bool` ist, oder
- eine lokale Bedingung, die in nachfolgenden Generatoren und Bedingungen benutzt werden kann.

2.2.7 Listen und Ströme als unendliche Listen

Unter anderem wurden *Listen* bereits kurz im Unterabschnitt 2.2.4 in Haskell erwähnt bzw. im Zusammenhang von *list Comprehension* implizit verwendet. Dieser Unterabschnitt soll jedoch näher auf Listen und insbesondere auf *unendliche Listen* als eines der features von Haskell eingehen. Des Weiteren sollen einige Listenoperationen definiert und erklärt werden, um Listen verarbeitende Funktionen, wie sie in Haskell sehr häufig definiert werden, besser zu verstehen.

In Haskell sind Listen sehr ausdrucksstarke Datentypen. Das Standard Prelude[9] von Haskell sowie die library module `List.hs` bieten zahlreiche vordefinierte Funktionen und Operationen für Listen an. Listen dienen zur Sammlung von mehreren Objekten des selben Typs, sodass es zu *jedem* Typ `t` in Haskell auch einen Typ „Liste von `t`“ gibt, der in Haskell mit `[t]` bezeichnet wird.

```
[1,2,3,4,1,4] :: [Int]
[True]        :: [Bool]
```

Dabei ist `[1,2,3,4,1,4]` eine Liste mit Elementen vom Typ `Int`, und `[True]` eine Liste mit Elementen vom Typ `Bool`. `String` gilt in Haskell als Synonym für `[Char]`, sodass die folgenden Listen das gleiche darstellen:

```
['a', 'a', 'b'] :: String
"aab"           :: String
```

[9] Prelude ist das Standardmodul in Haskell

Man kann auf allen möglichen Typen Listen definieren, so z.B. auch Listen von Funktionen oder Listen von Listen:

```
[factorial,fibonacci] :: [ Int → Int ]
[[1,2], [7,8,9], []]  :: [ [Int] ]
```

Im Folgenden sollen einige Listenoperationen, ähnlich wie in [OSG08], kurz vorgestellt werden. Es sei an dieser Stelle ebenfalls erwähnt, dass hiermit lediglich ein kleiner Eindruck vermittelt werden soll. Dazu wird der *Glasgow Haskell Compiler* (GHC) [JHH+93] verwendet, und mit `:type` immer zuerst der Typ der Funktion angegeben. Im Folgenden sehen wir jeweils die Eingabe in der GHCi Konsole und darunter die entsprechende Ausgabe.

Mit `length` kann man die Anzahl der Elemente einer Liste bestimmen:

```
ghci> :type length
length :: [a] -> Int

ghci> length []
0
ghci> length [1,2,3]
3
ghci> length „Strings sind auch Listen“
24
```

In Haskell definiert sich die `length` Funktion dabei folgendermaßen:

```
length []     = 0
length (_:t)  = 1 + (length t)
```

Wenn man etwa bestimmen möchte, ob eine Liste leer ist, kann man dazu die `null` Funktion verwenden:

```
ghci> :type null
null :: [a] -> Bool

ghci> null []
True
ghci> null „plugh“
False
```

Auf das erste Element einer Liste kann man mit `head` zugreifen:

```
ghci> :type head
head :: [a] -> a
```

```
ghci> head [1,2,3]
1
```

Bei manchen Listenoperationen, wie etwa dem `head`, sollte man jedoch aufpassen, dass man sie nicht auf leere Listen anwendet:

```
ghci> head []
*** Exception: Prelude.head: empty list
```

Wie man sieht, gibt der GHC ansonsten eine `Exception` aus.

Um zwei Listen aneinander zu fügen, gibt es in Haskell die `append` Funktion, die in der Infixschreibweise auch unter `++` bekannt ist:

```
ghci> :type (++)
(++) :: [a] -> [a] -> [a]

ghci> „Programmier“ ++ „sprache“
„Programmiersprache“
ghci> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
ghci> [] ++ [1,2,3]
[1,2,3]
```

Die Haskell-Definition von `append` lautet:

```
1 [] ++ ys      = ys
2 (x:xs) ++ ys  = x:(xs ++ ys)
```

Eine weitere sehr nützliche Funktion ist die `map` Operation. Sie wendet eine Funktion auf jedes Element einer Liste an und konstruiert als Ergebnis die Liste der Resultate. Um die `map` Operation an einem kleinen Beispiel zu demonstrieren, wird zunächst einmal die Funktion `square` definiert, die das Quadrat einer Zahl `x` berechnet.

```
ghci> let square x = x * x
ghci> :type square
square :: (Num a) => a -> a
```

Um nun das Quadrat von jedem Element in einer Liste zu bilden, wird anschließend die `map` Operation auf die `square` Funktion angewendet:

```
ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> map square [1..4]
[1,4,9,16]
```

Die Haskell-Definition von `map` ist:

```
map f []        = []
map f (x:xs)    = (f x) : (map f xs)
```

Darüber hinaus sind in Haskell noch weitere interessante und hilfreiche Operationen auf Listen definiert. Für ein vertieftes Studium mit Beispielen sei unter anderem auf [Tho97] und [OSG08] verwiesen.

Eine wichtige Konsequenz einer verzögerten Auswertung ist die Möglichkeit *unendliche Datenstrukturen* zu beschreiben. Ohne eine verzögerte Auswertung würde man „unendlich“ viel Zeit benötigen um eine unendliche Datenstruktur vollständig auszuwerten. Unter einer verzögerten Auswertung wird nämlich, statt des ganzen Objektes, nur der Teil der Datenstruktur ausgewertet, der von einer Funktion für die Berechnung eines Ergebnisses benötigt wird. Dieses Prinzip der unendlichen Datenstrukturen gleicht dem eines Stromes.

Es werden im Folgenden mittels unendlichen Listen einige Beispiele für Ströme veranschaulicht. Eine einfache unendliche Liste von der Konstanten 1 lässt sich wie folgt definieren:

```
eins = 1 : eins
```

Diese kann man als Strom von Einsen interpretieren.

Weiterhin lässt sich der Strom von natürlichen Zahlen auf folgende Weise darstellen:

```
ghci> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25, ...
```

Es kann hin und wieder passieren, dass man durch ungeeignetes Festhalten von Elementen eines Stroms zu viel Speicher blockiert. Das kann z.B. durch die feste Definition eines Stromes als Konstante im Programm auftreten, wobei die Länge davon abhängt, wie weit man die Liste im Programmverlauf angefordert hat. Beim Programmieren ist Vorsicht geboten, da man leicht nicht terminierende Anfragen stellen kann, wie z.B.:

```
ghci> length [1..]
Interrupted
```
[10]

Mit der vordefinierten Funktion `filter` lässt sich ein endlicher Strom aus einem unendlichen Strom herausfiltern:

```
filter ( \x → x > 100 && x < 1000) [1..]
```

```
[101,102,103,104,105,106,107,108,109,110,111,112,113,114,115, .....
......, 993,994,995,996,997,998,999]
```

[10] Berechnung wurde abgebrochen

Der entscheidende Vorteil von unendlichen Strukturen in einer Programmiersprache ist, dass die unendliche Version eines Programms sehr viel abstrakter sein kann, und sie dadurch einfacher zu implementieren ist. Betrachtet man etwa das Problem die $n$te Primzahl zu ermitteln:

```
1 factors :: Int → [Int]
2 factors n = [x | x ← [1..n], n `mod` x == 0]
```

```
1 primzahl :: Int → Bool
2 primzahl n = factors n == [1,n]
```

Wenn man in diesem Fall mit endlichen Listen arbeiten würde, dann müsste man im Voraus wissen wie groß die Liste sein muss, um die ersten n Primzahlen unterzubringen. Mit unendlichen Listen ist diese Information nicht nötig, denn es wird nur der Teil der Liste generiert, den man im Zuge der Berechnung benötigt.

3 Das SAT-Problem

3.1 Eine kurze Einführung in das SAT-Problem

Zur Veranschaulichung des *SAT-Problems* betrachten wir zunächst einmal folgendes diplomatische Problem von Brian Hayes [Hay79]:

> *"You are chief of protocol for the embassy ball. The crown prince instructs you either to invite Peru or to exclude Qatar. The queen asks you to invite either Qatar or Romania or both. The king, in a spiteful mood, wants to snub either Romania or Peru or both. Is there a guest list that will satisfy the whims of the entire royal family?"*

Um eine Lösung für sein verzwicktes Problem zu finden, könnte der Protokollchef sein Problem in der *Aussagenlogik*[1] (siehe Abbschnitt 3.2) wie folgt darstellen:

$$(Peru \vee \neg Qatar) \wedge (Qatar \vee Romania) \wedge (\neg Romania \vee \neg Peru)$$

Die Ländernamen stellen *Boolesche Variablen* dar, wobei das Zeichen $\vee$ für das logische OR und $\wedge$ für das logische AND steht. Eine *Negation* wird mit dem Zeichen $\neg$ dargestellt. Eine boolesche Variable tritt in einer *booleschen Formel* als *Literal* entweder negiert ($\neg$ Qatar) oder nicht negiert (Qatar) auf. In diesem Beispiel sind diejenigen Booleschen Variablen genau dann wahr, wenn wir respektive Peru, Quatar oder Rumänien einladen. Die Frage, die sich der Protokollchef stellen muss und die gleichzeitig die Definition des *Erfüllbarkeitsproblems der Aussagenlogik*[2] ist, lautet:

Gibt es eine Belegung der booleschen Variablen mit Wahrheitswerten wahr (*true*) und falsch (*false*), so dass die gesamte Formel den Wert wahr annimmt oder kann bewiesen werden, dass es keine solche Belegung gibt?

In diesem Beispiel gibt es zwei erfüllbare Belegungen (aus acht möglichen), also entweder Peru und Quatar jeweils den Wahrheitswert *true* zuweisen und Rumänien

[1] engl. propositional logic
[2] engl. *Boolean satisfiability problem* oder kurz: SAT-Problem

false, was bedeuten würde, dass der Protokollchef entweder beide, Peru und Quatar einladen würde, aber Rumänien ausschließen müsste, oder in der zweiten möglichen erfüllbaren Belegung nur Rumänien einladen würde, gleichzeitig jedoch Peru und Quatar ausschließen müsste.

1. erfüllende Belegung	2. erfüllende Belegung
$Peru = true$	$Romania = true$
$Quatar = true$	$Peru = false$
$Romania = false$	$Quatar = false$

3.2 Die Aussagenlogik (propositional calculus)

Im vorangegangenem Abschnitt wurden bereits Aussagen mittels logischer Verknüpfungen miteinander verbunden. Im Folgenden soll nun eine formale Definition der Aussagenlogik, ähnlich wie in [Sch06] angegeben werden. Grundbestandteile der Aussagenlogik sind Elementaraussagen:

- „Es regnet."
- „Newton ist ein Mensch."
- „Die Sonne ist heiß."
- „Auf dem Planeten Pluto gibt es Wasser."

3.2.1 Syntax

Dabei wird unterstellt, dass diese einfachen Aussagen durch die Wahrheitswerte *false* bzw. *true* beurteilbar sind. Formal lässt sich die Aussagenlogik durch ihre Syntax wie folgt definieren:

Definition 3.2.1. Die **Syntax** ist gegeben durch die Grammatik:

$$A \;:=\; X \,|(A \wedge A) \mid (A \vee A) \mid (\neg A) \mid (A \Rightarrow A) \mid (A \Leftrightarrow A) \mid 0 \mid 1$$

Hierbei ist X ein Nichtterminal für aussagenlogische Variablen und A ein Nichtterminal für Aussagen. Die Konstanten 0, 1 entsprechen *false* bzw. *true*. Üblicherweise werden bei der Notation von Aussagen Klammern weggelassen. Die Klammerung lässt sich durch die Prioritätsreihenfolge $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ wieder rekonstruieren.
Die Zeichen $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ nennt man *Junktoren*. Aussagen der Form 0, 1 oder

x nennen wir *Atome*. *Literale* sind entweder Atome oder Aussagen der Form $\neg A$, wobei A ein Atom ist.

A	*Atom, falls A eine Variable ist.*
A	*Literal, falls A ein Atom oder negiertes Atom ist.*
$\neg A:$	*negierte Formel.*
$A \wedge B:$	*Konjunktion (Verundung).*
$A \vee B:$	*Disjunktion (Veroderung).*
$A \Rightarrow B:$	*Implikation.*
$A \Leftrightarrow B:$	*Äquivalenz.*

3.2.2 Semantik

Es gilt nun den Formeln der Aussagenlogik eine Bedeutung zuzuordnen.

Definition 3.2.2. Semantik
Zunächst definiert man für jede Operation $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, Funktionen $\{0,1\} \to \{0,1\}$ bzw. $\{0,1\}^2 \to \{0,1\}$. Die Funktion $f_\neg$ ist definiert als $f_\neg(0) := 1$ und $f_\neg(1) := 0$. Ebenso definiert man $f_0 := 0$ und $f_1 := 1$. Die anderen Funktionen f_{op} sind definiert gemäß folgender Tabelle:

A	B	$\neg A$	$\wedge$	$\vee$	$\Rightarrow$	$\Leftarrow$	$\Leftrightarrow$
1	1	0	1	1	1	1	1
1	0	0	0	1	0	1	0
0	1	1	0	1	1	0	0
0	0	1	0	0	1	1	1

Jedem Atom wird durch Festsetzung ein Wahrheitswert zugeordnet. Dies geschieht durch eine Funktion im mathematischen Sinne, d.h. eine Abbildung von der Menge der Atome in die Menge der Wahrheitswerte. Formal definiert sich eine solche Zuordnung auch als *Belegung* oder *Interpretation*:

Definition 3.2.3. Eine **Interpretation** I ist eine Funktion
I : {aussagenlogische Variable} $\to \{0,1\}$.
Eine Interpretation I definiert für jede Aussage einen Wahrheitswert, wenn man sie auf Aussagen fortsetzt:

- $I(0) := 0$, $I(1) := 1$
- $I(\neg A) := f\neg(I(A))$
- $I(A\ op\ B) := f_{op}(I(A),\ I(B))$, wobei $op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

Wenn für eine Aussage F eine Interpretation I gilt: $I(F) = 1$, dann schreiben wir auch: $I \models F$ mit der Sprechweise I ist ein Modell für F, oder F gilt in I.

Weiterhin wollen wir folgende Erfüllbarkeitsbegriffe vereinbaren:

Definition 3.2.4. Sei A eine Aussage.

- A ist eine *Tautologie* (Satz, allgemeingültig) gdw. für alle Interpretationen I gilt: $I \models A$.
- A ist ein *Widerspruch* (unerfüllbar, widersprüchlich) gdw. für alle Interpretationen I gilt: $I(A) = 0$.
- A ist *erfüllbar* (konsistent) gdw. es eine Interpretationen I gibt mit: $I \models A$.
- ein *Modell* für eine Formel A ist eine Interpretation I mit $I(A) = 1$.

Beispiel 3.2.5.

- $X \vee \neg X$ ist eine Tautologie.
- $X \wedge \neg X$ ist ein Widerspruch.
- $X \vee Y$ ist erfüllbar.
- I mit $I(X) = 1$ sowie $I(Y) = 0$ ist ein Modell für $X \wedge \neg Y$.

3.2.3 Normalformen

In der Aussagenlogik gibt es verschiedene Normalformen. Unter anderem die *disjunktive Normalform (DNF)* und die *konjunktive Normalform (CNF)*. Die letzte nennt man auch *Klauselmenge*.

- *disjunktive Normalform (DNF)*:
  die Aussage ist eine Disjunktion von Konjunktionen von Literalen, d.h. von der Form
  $$(l_{1,1} \wedge \ldots \wedge l_{1,n_1}) \vee \ldots \vee (l_{m,1} \wedge \ldots \wedge l_{m,n_m})$$
  wobei $l_{i,j}$ Literale sind.
- *Konjunktive Normalform (CNF)*:
  die Aussage ist eine Konjunktion von Disjunktionen von Literalen, d.h. von der Form
  $$(l_{1,1} \vee \ldots \vee l_{1,n_1}) \wedge \ldots \wedge (l_{m,1} \vee \ldots \vee l_{m,n_m})$$
  wobei $l_{i,j}$ Literale sind.

Seien $l_1, \ldots, l_n$ Literale. Dann heißt $\omega_1 = (l_1 \vee \ \ldots \ \vee l_n)$ *Klausel*. Eine Klausel wird oft als Menge notiert: $\{l_1, \ldots, l_n\}$ und für eine Klauselmenge $\omega_1 \wedge \ldots \wedge \omega_m$ wird oft folgende Mengenschreibweise definiert: $\{\omega_1, \ldots, \omega_m\}$. Eine Klauselmenge ist daher gleichzusetzen mit einer Menge von Mengen, da sowohl $\wedge$ als auch $\vee$ assoziativ, kommutativ und idempotent sind, sodass Vertauschungen und ein Weglassen der runden Klammern erlaubt sind.

Beispiel 3.2.6.

$\omega = A \wedge (B \vee \neg C) \wedge (\neg A \vee C \vee D)$ wird notiert als

$\omega = \{\{A\}, \{B, \neg C\}, \{\neg A, C, D\}\}$

Es ist aus praktischer Sicht an dieser Stelle erwähnenswert, dass sich jede beliebige boolesche Formel in polynomieller Zeit in eine erfüllbarkeitsäquvalente[3] Klauselmenge transformieren lässt [Tse68].

3.3 Intuitive Verfahren zum Prüfen der Erfüllbarkeit

3.3.1 Wahrheitstafel

Um ein Modell für eine aussagenlogische Formel zu finden, gibt es grundsätzlich die Möglichkeit durch das Aufstellen einer Wahrheitstafel alle möglichen Belegungen mit $0 = \mathit{false}$ und $1 = \mathit{true}$ auszuprobieren. Abbildung 3.1 zeigt die Aufstellung der Wahrheitstafel für unser Beispiel aus Abschnitt 3.1, wobei die Variablen P, Q und R respektive für *Peru*, *Qatar* und *Romania* stehen.

Allerdings erweist sich das bei steigender Variablenanzahl als mühevoll, denn die Anzahl der Überprüfungen beträgt 2^n, wobei n die Anzahl der Variablen ist.

3.3.2 Resolution

Einer der einfachsten Verfahren zum Testen der Erfüllbarkeit basiert auf der *Resolution* [Rob65]. Sei A eine boolesche Variable in einer Klauselmenge, die die Klauseln C_i und C_j enthält, wobei $A \in C_i$ und $\neg A \in C_j$. Die Resolution erlaubt uns aus zwei solcher Klauseln C_i und C_j eine neue Klausel $C_i - A \vee C_j - \neg A$ herzuleiten und zur Klauselmenge hinzuzufügen. Man nennt die ersten beiden Klauseln auch

[3] nur unter Erhaltung der Erfüllbarkeit, aber nicht unter Erhaltung der Tautologie Eigenschaft

P	Q	R	$(P \vee \neg Q) \wedge (Q \vee R) \wedge (\neg R \vee \neg P)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

Abbildung 3.1: Wahrheitstafel zur Darstellung der möglichen Kombinationen von Variablenbelegungen

Elternklauseln und die neu hergeleitete Klausel *Resolvente*:
Resolution:

$$\frac{\begin{array}{ll} A & \vee C_i \\ \neg A & \vee C_j \end{array}}{C_i \vee C_j}$$

Außerdem nimmt man auch noch an, dass die Konjunktion der Klauseln eine Menge ist, d.h. nur neue Klauseln, die nicht bereits vorhanden sind, können hinzugefügt werden. Es werden solange Resolventen hergeleitet bis entweder die leere Klausel abgeleitet wurde oder keine neue Resolvente mehr herleitbar ist. Falls eine leere Klausel abgeleitet wird, so hat man ein Widerspruch, d.h. die Klauselmenge ist unerfüllbar.

Die Resolution ist ein *korrektes* aber *unvollständiges* Verfahren, da nicht garantiert ist, dass alle Klauseln, die man aus der Klauselmenge ableiten kann, auch durch Resolution hergeleitet werden. Jedoch ist das Verfahren *widerspruchs-vollständig*, da im Falle einer unerfüllbaren Formel immer garantiert ist, dass die leere Klausel abgeleitet wird. Daher dient das Resolutionsverfahren vielmehr zum Erkennen von Widersprüchen, wobei statt eines Tests auf Erfüllbarkeit einer Formel F die Formel $\neg F$ auf Unerfüllbarkeit getestet wird.

Betrachten wir als Beispiel nun folgenden Resolutionsverlauf:

1. $\{\neg P, R\}$
2. $\{\neg Q, R\}$
3. $\{\neg R\}$
4. $\{P, Q\}$

---

5. $\{\neg P\}$	1,3
6. $\{\neg Q\}$	2,3
7. $\{Q\}$	4,5
8. $\{\}$	6,7

Die Klauseln vor der Linie repräsentieren die anfänglichen Klauseln, während die hinter der Linie die Resolventen repräsentieren, zusammen mit der Nummer ihrer Elternklausel. Dieser Resolutionsverlauf zeigt, wie beispielsweise die leere Klausel aus den Original-Klauseln (1. - 4.) abgeleitet werden kann. Im übrigen hat man daher einen Widerspruch und somit die Unerfüllbarkeit der Original-Klauseln gezeigt.

Ein wichtiger Spezialfall der Resolution ist auch unter *Unit-Resolution* bekannt. In der Unit-Resolution besteht mindestens einer der Elternklauseln aus einer Klausel mit nur einem Literal. Aufgrund seiner Effizienz, gilt die Unit-Resolution als eine zentrale Technik, die in zahlreichen Algorithmen angewendet wird. Auf diese Resolutionstechnik wird im Rahmen des Kapitels 4 noch einmal zurückgegriffen.

3.3.3 Der Entscheidungsbaum

Eine bessere Möglichkeit wäre, dass man die Variablen nach und nach belegt und überprüft, ob sich für die anderen Variablen dadurch bereits Werte ergeben. Die Herangehensweise, die Variablen systematisch zu belegen, kann auch als *Entscheidungsbaum* dargestellt werden.

Jedes Belegen einer Variablen entspricht dabei einer Entscheidung, also einer Verzweigung im Baum. Bei dieser Vorgehensweise ist es ratsam, dass sich die boolesche Formel in CNF-Darstellung befindet. Betrachten wir z.B. die Klauselmenge $\{\{\neg P, Q\}, \{\neg Q, R\}\}$. Mit Hilfe des Entscheidungsbaumes lässt sich leicht erkennen, dass bei einer Belegung für $P = 0$ und $Q = 1$ die Formel nicht mehr erfüllt werden kann, unabhängig von weiteren Belegungen, da bereits die leere Klausel abgeleitet wurde. Sobald nämlich in einer Klauselmenge mindestens eine Klausel aufgrund einer Zuweisung zu *false* (leere Klausel) evaluiert, lässt sich somit einfacher feststellen, dass die gesamte Formel nicht mehr zu *true* evaluieren kann. Diese Vorgehensweise wird im Rahmen von Kapitel 4 noch einmal genauer erläutert.

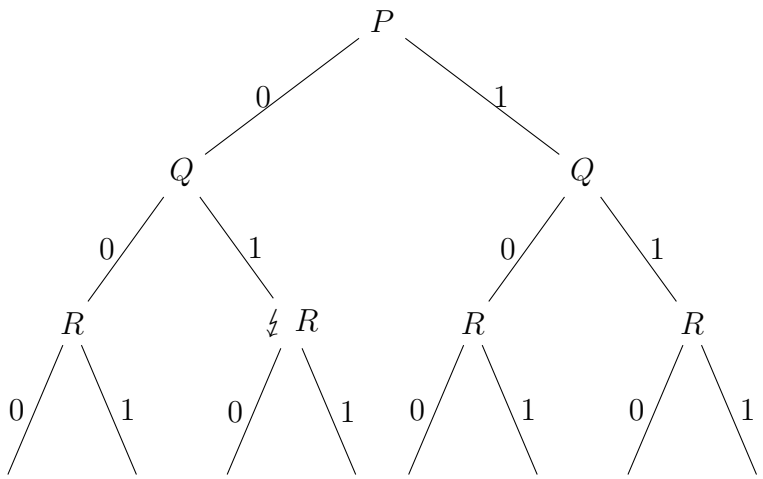

Abbildung 3.2: Entscheidungsbaum zur Darstellung der Suche nach Lösungsmöglichkeiten

Der Suchraum kann also dadurch reduziert werden, indem bestimmte Pfade im Entscheidungsbaum ignoriert werden, noch bevor man die gesamte Variablenbelegung kennt. Am Beispiel der Formel $\{\{\neg P, Q\}, \{\neg Q, R\}\}$ ist der obige Entscheidungsbaum an dieser Stelle mit einem $\lightning$ markiert.

3.4 Praxis und Theorie des SAT-Problems

Zum Lösen eines SAT-Problems sind automatische Verfahren in der Lage komplizierte boolesche Formeln auf Erfüllbarkeit zu prüfen. Solche sogenannten *SAT-Solver* werden bereits in weiten Spektren der EDA und Gebieten wie sie bereits im Abschnitt 1.1 erwähnt wurden, eingesetzt. Der erste SAT-Algorithmus ist auf Martin Davis und Hilary Putnam im Jahre 1960 [DP60] zurückzuführen (auch bekannt unter *DP-Algorithmus*). Danach erschienen zahlreiche Algorithmen und Techniken in der Literatur, mit dem Ziel die Effizienz von SAT-Solvern zu verbessern. Die meisten dieser Verfahren beruhen auf dem DPLL-Algorithmus aus dem Jahre 1962 [DLL62], einer Verfeinerung von George Logemann und Donald W. Loveland am früheren DP-Algorithmus.

Gleichzeitig stellt das SAT-Problem den Grundstein der Komplexitätstheorie dar, da es das erste Problem war, an der die $\mathcal{NP}$-*Vollständigkeit* nachgewiesen wurde [Coo71], auf das sich zahlreiche andere Probleme reduzieren lassen. Die Frage $\mathcal{P} = \mathcal{NP}$ - ob Probleme, die $\mathcal{NP}$-vollständig sind, im *worst-case* in polynomieller deterministischer Zeit gelöst werden können - ist eine der größten ungelösten Proble-

me der Komplexitätstheorie. Obwohl das SAT-Problem $\mathcal{NP}$-vollständig und damit theoretisch „schwer“ ist, lassen sich mit modernen SAT-Solvern in der Praxis mittlerweile sehr große Formeln (mit mehreren millionen Variablen) auf Erfüllbarkeit testen.

4 Klassischer DPLL Algorithmus mit chronologischem-backtracking

Das Benutzen von Maschinen zum Lösen logischer Probleme kann bis in das Jahr 1869 auf die erste mechanische Maschine von William Stanley Jevon zurück datiert werden. Seit Beginn der Erfindung von digitalen Rechnern wurden große Anstrengungen dahingehend gemacht Algorithmen zu entwickeln, um mit Hilfe von modernen Computern Probleme aus der Logik zu lösen. Diese Anstrengungen wurden erstmals durch die KI mit der Absicht instanziiert, Programme zu entwickeln mit der Fähigkeit „zu denken“. Weitere Entwicklungen werden unlängst durch die EDA weiter vorangetrieben.
Der erste Algorithmus zum Lösen von SAT-Problemen ist auf den resolutionsbasierten Algorithmus von Davis und Putnam im Jahre 1960 [DP60] zurückzuführen. Dieser Algorithmus leidet jedoch unter einem rasanten Speicheranstieg. Um dieses Problem zu bewältigen, veröffentlichten Davis zusammen mit Loveland und Logemann im Jahre 1962 eine modifizierte Version, die statt der Resolution auf einer Suche aufbaute. Dieser Algorithmus ist in der Literatur auch nach den Initialen der Autoren bekannt unter dem Namen DLL- oder DPLL-Algorithmus [DLL62].

Neben diesen beiden Algorithmen gibt es in der Literatur noch andere nennenswerte SAT-Algorithmen, die auf unterschiedliche Prinzipien basieren, zu ihnen zählen unter anderem Binary Decision Diagrams (BDDs) [Bry86], Stalmarck's Algorithmus [Stå89, MS98] und randomisierte lokale Suchalgorithmen [SLM92]. Einen entscheidenden Vorteil des DPLL-Algorithmus gegenüber randomisierte lokale Suchalgorithmen, ist die Vollständigkeit, d.h. für ein gegebenes SAT-Problem, wird entweder eine Lösung gefunden oder festgestellt, dass keine Lösung existiert. [GPFW96] bietet einen guten Überblick über viele bislang entwickelte SAT-Algorithmen.
Die DPLL-Implementierungen sind aktuell die schnellsten implementierten Algorithmen um eine aussagenlogische Formel auf Erfüllbarkeit zu testen, aus diesem Grund werden wir im weiteren Verlauf dieser Arbeit andere SAT-Algorithmen außenvorstellen und uns nur auf die DPLL-Implementierungen konzentrieren.

Dazu wird im nächsten Abschnitt eine Basis Terminologie eingeführt, auf die dann weitere Diskussion beruhen werden. Im Anschluss daran wird detailliert auf die DPLL-Implementierung gemäß [DLL62] eingegangen, welches das Grundgerüst für

die darauffolgenden Verbesserungen darstellt. Ferner bezeichnen wir dieses Grundgerüst als den *klassischen DPLL Algorithmus* und deren Verbesserungen als *modernen DPLL Algorithmus*.

4.1 Basis Terminologie

In diesem Abschnitt wird die Basis Terminologie gemäß [NOT06] eingeführt, auf die sich dann die weiteren Ausführungen beziehen werden.

Sei P eine feste Menge von aussagenlogischen Symbolen. Wenn $p \in P$, dann ist p eine *Variable* und p und $\neg p$ sind *Literale* von P. Die *Negation* des Literals l, geschrieben als $\neg l$, bezeichnet $\neg p$, wenn $l = p$, und p, wenn $l = \neg p$ ist. Eine *Klausel* ist eine Disjunktion von Literalen $l_1 \vee \cdots \vee l_n$. Eine *Unit-Klausel* ist eine Klausel, die nur aus einem einzigen Literal besteht. Es gibt verschiedene Darstellungsmöglichkeiten von Booleschen Formeln, wir werden uns jedoch nur auf die CNF-Darstellung, der Definition der Klauselmenge (Abschnitt 3.2) konzentrieren, da einzig diese für unsere Diskussion relevant ist. Des Weiteren wird im Folgenden davon ausgegangen, falls von einer *Formel* F gesprochen wird, dann ist damit die Klauselmenge, nämlich eine Konjunktion von einem oder mehreren Klauseln $C_1 \wedge \cdots \wedge C_n$ gemeint, und wir schreiben die Formel als Menge $\{C_1, \cdots, C_n\}$, in der wir die $\wedge$ Verknüpfung einfach durch Kommata ersetzen (siehe dazu auch das Beispiel 3.2.6).

Eine (Wahrheits-)*Belegung* M ist eine Menge von Literalen, sodass für kein p gilt, dass $\{p, \neg p\} \subseteq M$. Ein Literal l in M ist *true*, wenn $l \in M$, ein Literal l ist *false* in M, wenn $\neg l \in M$, ansonsten bezeichnen wir das Literal als *undefiniert* in M. Ein Literal ist *definiert* in M, wenn es entweder *true* oder *false* ist in M. Die Belegung M ist *total* über P, wenn kein Literal in M undefiniert ist. Eine Belegung M ist *partial* über P und wird als *Teilbelegung* M bezeichnet, wenn nicht alle Literale in M definiert sind. Die Teilbelegung entspricht also einer Wahrheitsbelegung für eine Teilmenge von Variablen aus F. Dabei bezeichnet $F|_M$ die *vereinfachte* Formel, die man erhält, wenn man alle in M definierten Variablen in der Formel mit ihren zugewiesenen Wahrheitswerten ersetzt. Das bedeutet für die Formel F konkret: wenn ein Literal in M als l (bzw. $\neg l$) definiert ist, dann werden alle Klauseln in F, in der dieses Literal mindestens einmal als l (bzw. $\neg l$) vorkommt, entfernt (*Unit-Resolution*) und aus den übrigen Klauseln in F das Literal, welches als $\neg l$ (bzw. l) vorkommt, gelöscht (*Unit-Subsumption*). Das Ergebnis ist dann die durch M vereinfachte Formel $F|_M$. Im Übrigen wird die Anwendung der beiden Regeln Unit-Resolution und Unit-Subsumption auch als *Unit-Propagation* zusammengefasst.

Eine Klausel C ist *true* in M, wenn mindestens einer seiner Literale in M *true* ist, und eine Klausel ist *false*, wenn alle ihre Literale in M *false* sind, ansonsten ist

eine Klausel undefiniert in M.
Eine Belegung M falsifiziert eine Formel F genau dann, wenn $M \models \neg F$ gilt. Eine Formel F ist *true* bzw. *SAT*[1] in M, oder *erfüllbar* durch M, bezeichnet durch $M \models F$, wenn alle ihre Klauseln *true* sind. In diesem Fall ist M ein *Modell* von F. Wenn F keine Modelle besitzt, dann ist F *unerfüllbar* bzw. *UNSAT*[2]. Seien F und F' Formeln, dann schreiben wir $F \models F'$, wenn F' in allen Modellen von F *true* ist. Wir sagen dann: F' ist eine *logische Konsequenz* von F.

Im Folgenden bezeichnen (indizierte) Kleinbuchstaben immer Literale, C und D bezeichnen Klauseln, F und G bezeichnen Formeln und M und N bezeichnen Belegungen. Wenn C eine Klausel $l_1 \vee \cdots \vee l_n$ ist, dann schreiben wir auch manchmal $\neg C$ statt $\neg l_1 \wedge \cdots \wedge \neg l_n$.

4.2 Davis-Putnam-Logemann-Loveland Algorithmus

4.2.1 Erfüllbarkeit einer Formel in CNF-Darstellung

Damit eine Formel in CNF-Darstellung insgesamt durch ein Modell M erfüllbar wird, muss jede ihrer Klauseln durch M erfüllbar sein. Sobald eine Klausel existiert, in der alle ihre Literale den Wahrheitswert *false* haben, dann ist die Klausel *false* bzw. UNSAT und die Formel F kann in der gegenwärtigen Variablenbelegung M nicht mehr erfüllt werden. Eine solche Klausel, deren Literale alle *false* sind wird auch als *Konflikt-Klausel* oder engl. *conflicting clause* bezeichnet. Wenn etwa bei einer aktuellen Belegung M alle Literale in einer Klausel F bis auf eines zu *false* evaluiert sind, muss das übrig gebliebene Literal dieser Klausel zu *true* evaluieren, andernfalls würde die Klausel zu *false* evalieren. Um dieses festzustellen, wird das im Abschnitt 4.1 bereits definierte Verfahren der Unit-Propagation angewendet.

4.2.2 Suche im Entscheidungsbaum

Bevor der klassische DPLL Algorithmus anhand eines Pseudocodes angegeben wird, soll seine Vorgehensweise zunächst einmal anhand eines Entscheidungsbaumes erklärt werden. Die Herangehensweise des DPLL Algorithmus die Variablen systematisch zu belegen, entspricht nämlich einer rekursiven Suche in einem Entscheidungsbaum (wie in Abbildung 4.1 dargestellt). Wenn der DPLL Algorithmus die Suche mit dem ersten Knoten (auch Wurzelknoten genannt) beginnt, enthält dieser zunächst

[1] SAT steht für engl. satisfiable = erfüllbar
[2] UNSAT steht für engl. unsatisfiable = unerfüllbar

einmal eine Original-Formel F in CNF-Darstellung und eine leere Menge M (in M sind noch keine Variablenbelegungen definiert). Im Entscheidungsbaum gehen von jedem Knoten bis auf die Blätter zwei Kanten aus: die linke Kante repräsentiert die Belegung der Variable x mit *true* und die rechte Kante repräsentiert die Belegung der Variable x mit *false*.

Welche Variablenbelegung auch immer für x in M definiert wird, so wirkt sich die Belegung auf die aktuell betrachtete Formel F aus. Die Berechnung dieser Auswirkung auf F besteht im Grunde aus der Anwendung der Unit-Propagation auf F (siehe Abschnitt 4.1). Wenn wir uns für $x = true$ entscheiden, was im Folgenden kurz mit $F|_x$ bezeichnet wird, dann wenden wir eine Unit-Propagation mit x auf F an, und falls wir uns für $x = false$ entscheiden, dann schreiben wir $F|_{\neg x}$ und wenden entsprechend eine Unit-Propagation mit $\neg x$ auf F an. Die Entscheidung, ob wir eine Unit-Propagation mit x oder $\neg x$ auf F anwenden, wird dabei in der Belegung M gesichert, was durch eine Erweiterung der aktuellen Belegung M durch M' ausgedrückt wird.

Es sind drei Fälle in F' nach einer Anwendung von M' auf F möglich:

- F' hat keine Klauseln mehr, d.h. F' ist die leere Formel.
- F' ist nicht die leere Formel und enthält auch keine leere Klausel.
- F' enthält die leere Klausel.

Sobald die resultierende Formel F' keine Klauseln mehr enthält, ist mit M' ein Modell gefunden. Der Grund ist nämlich, dass eine Klausel aus einer Formel entfernt wird, wenn sie erfüllbar ist. Wenn also alle Klauseln entfernt wurden, dann waren alle Klauseln erfüllbar, und somit ist auch die Formel als Ganzes erfüllbar. In diesem Fall ist die Suche beendet, und M' kann als Modell von F ausgegeben werden.

Im dem Fall, dass die leere Formel noch nicht erreicht ist, und die Formel noch keine leere Klausel enthält, wird die Suche fortgesetzt, indem die nächste Variable aus F' ausgewählt, ihr einen Wahrheitswert zugewiesen und die Unit-Propagation angewendet wird.

Wenn die resultierende Formel F' die leere Klausel enthält, dann ist mit M' eine Belegung erreicht, die F' falsifiziert. Das passiert, wenn alle Literale in einer Klausel unter den Belegungen in M' zu *false* evaluiert sind. Ein Literal, welches zu *false* evaluiert ist, wird nämlich aus der Klausel, in der sie vorkommt, gelöscht. Um eine Lösung zu finden, muss nun eine vorher getroffene Entscheidung revidiert werden. An dieser Stellen wird die letzte Entscheidung in M' wieder zurückgenommen, d.h. falls x_i vorher mit *true* belegt wurde, so wird x_i jetzt mit *false* belegt, oder falls x_i

vorher mit *false* belegt wurde, dann wird x_i jetzt mit *true* belegt. Diese neue Belegung in M'' führt dann zu einer neuen resultierenden Formel F''. Jetzt könnte F'' wieder die leere Klausel enthalten. Dies würde bedeuten, dass beide Belegungen von x_i (*false* oder *true*) nicht dazu geführt haben, dass F erfüllbar wird. In diesem Fall müsste dann die nächst höhere Belegung in M'' revidiert werden. Das wird so lange fortgesetzt, bis wir entweder eine erfüllbare Belegung gefunden haben, die zu einer leeren Formel führt, oder alle Möglichkeiten Belegungen zu revidieren, ausgeschöpft sind, und die Formel somit insgesamt nicht mehr erfüllbar ist. An dieser Stelle wäre man wieder an der Wurzel des Entscheidungsbaumes angelangt.

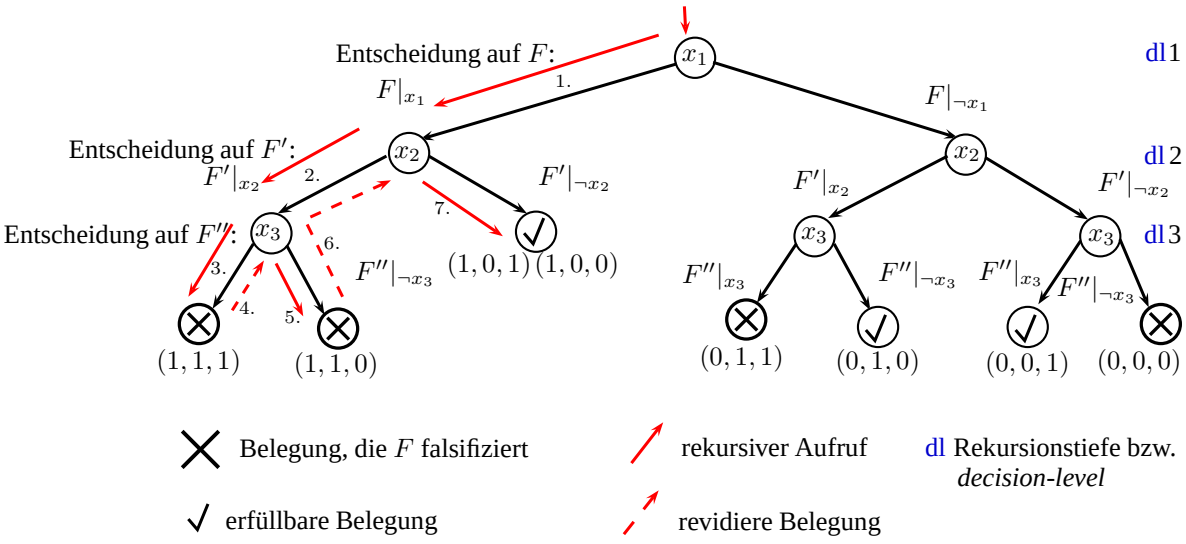

Abbildung 4.1: Rekursive Suche des DPLL Algorithmus in einem Entscheidungsbaum

Um das Prinzip der DPLL Suche zu verdeutlichen, soll die Suche anhand der folgenden Formel F und der Belegung M mit Hilfe des Entscheidungsbaumes in Abbildung 4.1 visualisiert werden:

$$F = \{\{\neg x_1, \neg x_2, \neg x_3\}, \{x_1, \neg x_2, \neg x_3\}, \{x_1, x_2, x_3\}, \{\neg x_1, \neg x_2, x_3\}\}$$
$$M = \{\ \}.$$

Um die Suche durchzuführen, nehmen wir an, dass wir zuerst die Variable x_1 belegen. Somit ist x_1 der erste Suchknoten (Wurzelknoten). Wir entscheiden uns ihn zuerst mit dem Wahrheitwert *true* zu belegen, sodass wir den ersten Suchknoten mit einer ausgehenden Kante $F|_{x_1}$ verlassen und in einem neuen Suchknoten landen. Die resultierende Formel und die dazugehörige Belegung M sind dann:

$$F' = \{\{\neg x_2, \neg x_3\}, \{\neg x_2, x_3\}\}$$
$$M' = \{x_1\}.$$

Wir erhalten F', indem wir $F|_x$ anwenden. Zuerst entfernen wir $\neg x_1$ aus der ersten und letzten Klausel in F, da unter der Belegung $x_1 = \mathit{true}$ das Literal $\neg x_1$ in der ersten und letzten Klausel zu *false* evaluiert. Ferner können wir die Klauseln entfernen, in denen x_1 vorkommt, da die entsprechenden Klauseln durch die Belegung $x_1 = \mathit{true}$ erfüllt sind.

Wie wir sehen können, ist F' weder die leere Formel, noch enthält sie eine leere Klausel, sodass wir unsere Suche fortsetzen, dadurch dass wir eine in F' noch nicht zugewiesene Variable mit einem Wahrheitswert belegen. Wir wollen nun x_2 als nächste Variable aus F' auswählen und ihr den Wahrheitswert $x_2 = \mathit{true}$ zuweisen. Das Ergebnis ist die Formel F'' und die Belegung M'':

$$\begin{aligned} F'' = &\quad \{\{\neg x_3\}, \{x_3\}\} \\ M'' = &\quad \{x_1, x_2\}. \end{aligned}$$

Auch in diesem Fall enthält die Formel weder eine leere Klausel noch ist die Formel leer. Wir setzen also die Suche fort, indem wir die letzte Variable $x_3 = \mathit{true}$ setzen. Dadurch können wir die zweite Klausel aus F'' entfernen, da sie erfüllt wird, und gleichzeitig kann das Literal $\neg x_3$ in der ersten Klausel entfernt werden, da das Literal in dieser Belegung zu *false* evaluiert. Die resultierende Formel enthält jetzt die leere Klausel:

$$\begin{aligned} F''' = &\quad \{\{\ \}\ \} \\ M''' = &\quad \{x_1, x_2, x_3\}. \end{aligned}$$

An diesem Punkt stellen wir fest, dass wir mit M''' eine Belegung erreicht haben, die F falsifiziert, da die resultierende Formel F''' eine leere Klausel enthält. Jetzt revidieren wir die letzte Belegung, indem wir zu dem Knoten x_3 zurückgehen und die Belegung $x_3 = \mathit{false}$ auf F'' anwenden (haben wir noch nicht betrachtet). Dies führt wiederum zu einer leeren Klausel, sodass wir die nächst höhere Variable x_2 revidieren müssen, da x_3 in beiden Belegungen zu keiner Lösung geführt hat. Wir wenden also $x_2 = \mathit{false}$ auf F' an. Das Ergebnis ist die leere Formel, da alle Klauseln durch die Belegung $x_1 = \mathit{true}$ und $x_2 = \mathit{false}$ erfüllbar sind. Somit haben wir ein Modell für F gefunden. Der Wahrheitswert für x_3 ist unter einer solchen Belegung nicht mehr relevant und kann beliebig gesetzt werden.

4.2.3 Der DPLL Algorithmus

Wir wollen uns nun einmal den klassischen DPLL Algorithmus anhand des Pseudocodes in Abbildung 4.2 anschauen. Als Eingabe erhält der Algorithmus eine Formel F und liefert, falls sie erfüllbar ist, die Ausgabe SAT und das entsprechende Modell M, oder falls die Formel unerfüllbar ist, UNSAT.

Während mit einer leeren Belegung M begonnen wird, travesiert ein Backtrack-Suchalgorithmus implizit den Suchraum der Wahrheitsbelegungen und organisiert dabei die Suche für eine erfüllbare Belegung M, indem es einen entsprechenden Entscheidungsbaum aufbaut. Die Suche ist dabei mit der einer Tiefensuche vergleichbar. Jeder Knoten im Entscheidungsbaum entspricht dabei der Belegung einer Variablen.

Der Suchprozess durchläuft folgende Schritte iterativ:

1. *decision process*: Erweitern der aktuellen Belegung, indem ein Literal eine Wahrheitsbelegung zugewiesen wird (Literal wird in M definiert). Diese Entscheidungsprozedur leitet den Algorithmus in eine neue Region des Suchraums ein. Die Suche endet erfolgreich, wenn alle Klauseln erfüllbar sind; es endet nicht erfolgreich, wenn mindestens eine Klausel *false* wird und alle möglichen Belegungen erschöpft sind.

2. *deduction process*: Erweitern der aktuellen Belegung, indem die logischen Konsequenzen der zugewiesenen Wahrheitsbelegungen aus 1. durch Unit-Propagation abgeleitet werden. Die zusätzlichen Belegungen, die im deduction process abgeleitet werden, nennt man auch *implizierte Belegungen* oder einfach *Implikationen*. Der deduction process kann seinerseits auch zur Identifizierung von einer oder mehreren unerfüllbaren Klauseln führen, woraus wiederum folgt, dass die aktuelle Belegung die zugrundeliegende Formel falsifiziert. Eine solche Situation wird dann als *Konflikt* bezeichnet, und die dazugehörige Belegung nennt man auch *Konfliktbelegung*.

3. *backtracking process*: Revidieren der letzten Belegung, falls sie eine Konfliktbelegung ist, sodass ihre komplementäre Belegung ausprobiert werden kann. Dieser Mechanismus zieht den Algorithmus aus derjenigen Region des Suchraumes zurück, die nicht zu einer erfüllbaren Belegung geführt hat.

Die Idee des Algorithmus 4.2, ist es iterativ ein noch in M undefiniertes Literal l aus F auszuwählen, und rekursiv eine Suche nach einer erfüllbaren Belegung für $F|_l$ bzw. $F|_{\neg l}$ durchzuführen. Der Schritt, wobei ein solches l aus F ausgewählt wird, bezeichnet man als *branching*[3]. Wenn l dabei mit dem Wahrheitswert *true* (l wird zu M hinzugefügt) oder mit *false* ($\neg l$ wird zu M hinzugefügt) belegt wird, nennt man dies auch *decision* und die dazugehörige Rekursionstiefe - zu welchem Zeitpunkt das *branching* durchgeführt wurde - als *decision-level*. Das entsprechende Literal wird dann als *decision-literal* bezeichnet.

Die Teilbelegung M wird beibehalten, solange F unter dieser Belegung nicht falsifiziert wird. Sobald $F|_M$ eine leere Klausel enthält gilt $M \models \neg F$. Trifft der Algorithmus nun auf eine Belegung M mit $M \models \neg F$, bzw. auf einen daraus resultierenden

[3] engl. für Fallunterscheidung

Input : zu Beginn ist die Belegung $M = \emptyset$, und F ist eine Formel in CNF-Darstellung.
Output: UNSAT oder SAT und ein Modell, d.h eine Belegung M, die F erfüllt.

```
DPLL-recursive(M, F)
begin
    if F enthält die leere Klausel then
        return UNSAT
    end
    if F ist die leere Formel then
        Output M
        return SAT
    end

    UnitPropagate(M, F);                                  // unit-propagation
    begin
        while F enthält eine Unit-Klausel und keine leere Klausel do
            F' ← F|_v
            M' ← M ∪ {v}
        end
        return DPLL-recursive(M', F')
    end

    PureLiteral(M, F);                                    // pure literal rule
    begin
        while F enthält ein pures Literal p (bzw. ¬p) do
            F' ← lösche in F alle Klauseln, die p (bzw. ¬p) enthalten
            M' ← M ∪ {p} (bzw. M ← M ∪ {¬p})
        end
        return DPLL-recursive(M', F')
    end

    Decide(M, F);                                                 // brachning
    begin
        l ← wähle ein undefiniertes Literal aus M
        if DPLL-recursive(M ∪ {l}, F|_l) = SAT then
            return SAT
        else
            return DPLL-recursive(M ∪ {¬l}, F|_¬l)
        end
    end
end
```

Abbildung 4.2: Algorithmus DPLL-recursive(M, F)

Konflikt für die Formel F, so kommt es zu einem sogenannten *Backtracking*. Das bedeutet, dass die Rekursion an dieser Stelle abgebrochen und alle Variablenbelegungen bis einschließlich dem letzten decision-literal, welches zuletzt durch ein branching ausgewählt wurde, aus M gelöscht und das decision-literal in seiner Belegung umgekehrt wird. Wenn wir ein decision-literal in seiner Belegung umkehren, dann meinen wir, falls es vorher mit l in M definiert war, dass es anschließend mit $\neg l$ in M definiert wird, und umgekehrt. Das Backtracking ist hierbei nur auf *das Umkehren der zuletzt getroffenen Entscheidung* (des letzten decision-literals) beschränkt, was man auch als *chronologisches-backtracking* bezeichnet.

Um die Effizienz zu steigern werden die Literale in Unit-Klauseln immer sofort so in M definiert, dass ihre Klauseln *true* werden, wie aus der Prozedur `UnitPropagate` ersichtlich wird. Weiterhin lässt sich mit Hilfe der Prozedur `PureLiteral` die Formel zusätzlich vereinfachen. Die Aufgabe der *Pure Literal Regel* ist es, isolierte Literale[4] zu finden, und diejenigen Klauseln, in der sie vorkommen zu löschen. Auch hier werden isolierte Literale immer so in M definiert, dass ihre Klauseln, in der sie vorkommen *true* werden. Diese Regel wird jedoch wegen des hohen Rechenaufwandes aufgrund der Suche von isolierten Literalen, häufig nicht angewendet. Unter Umständen kann sie den Gesamtprozess sogar verlangsamen. Aus Effizienzgründen wird die Pure Literal Regel daher in modernen DPLL Algorithmen normalerweise nur während eines Vorverarbeitungschrittes[5] herangezogen.

4.2.4 Korrektheit und Vollständigkeit des DPLL Algorithmus

Der DPLL Algorithmus ist ein vollständiges und korrektes Entscheidungsverfahren für die (Un)Erfüllbarkeit von aussagenlogischen Formeln. Der Beweis beruht auf dem folgenden Satz aus [DP60]:

Satz 4.2.1. Eine aussagenlogische Formel in CNF-Darstellung ist unerfüllbar, wenn es durch Resolution möglich ist eine leere Klausel aus den original Klauseln zu generieren (siehe Bsp. [DP60]).

Beweis: Klauseln, die durch Resolution generiert werden sind redundant und können zu der Original-Formel hinzugefügt werden. Wenn eine Formel in CNF-Darstellung eine leere Klausel enthält, dann ist sie unerfüllbar. Daher gilt Satz 4.2.1. □

[4] ein Literal p ist isoliert, wenn $\neg p$ nicht mehr in F vorkommt, entsprechend ist $\neg p$ isoliert, wenn p nicht mehr in F vorkommt

[5] bei einem Vorverarbeitungsschritt wird die Formel F soweit wie möglich vereinfacht bevor mit der eigentlichen Suche begonnen wird, für eine Zusammenfassung möglicher Vorverarbeitungsschritte siehe [LMS01]

4.2.5 Beispiele zum DPLL Algorithmus

Wir schauen uns als nächstes einige kleine Beispiele an.

Beispiel 4.2.2.

(a) $F = \{\{\neg x\}, \{x, y\}, \{x, \neg y, z\}, \{\neg y, \neg z\}\}$
`UnitPropagate` mit $v = \neg x$:
Klausel $\{\neg x\}$ wird gestrichen, x wird aus allen Klauseln gestrichen
$\rightsquigarrow F = \{\{y\}, \{\neg y, z\}, \{\neg y, \neg z\}\}$
`UnitPropagate` mit $v = y$:
$\rightsquigarrow F = \{\{z\}, \{\neg z\}\}$
`UnitPropagate` mit $v = z$:
$\rightsquigarrow F = \{\{\ \}\}$
F enthält die leere Klausel $\Rightarrow F$ ist unerfüllbar.

(b) $F = \{\{x, y, z\}, \{\neg x, y, z\}, \{\neg x\}, \{z, \neg y\}\}$
`UnitPropagate` mit $v = \neg x$:
$\rightsquigarrow F = \{\{y, z\}, \{z, \neg y\}\}$
`Decide`:

- Fall 1: $F|_y$ y ist nun **decision-literal**
  lösche Klausel, in der y vorkommt und lösche Literal $\neg y$ aus allen Klauseln
  $\rightsquigarrow F = \{\{z\}\}$
  `UnitPropagate` mit $v = z$:
  $\rightsquigarrow F = \{\ \}$
  F ist die leere Formel $\Rightarrow F$ ist erfüllbar und das Modell $M = \neg x, y, z$
- Fall 2: $F|_{\neg y}$
  wird nicht mehr untersucht[6].

(c) $F = \{\{w, x\}, \{\neg y, z\}, \{y, \neg z\}, \{w, \neg z\}\}$
`pure literal rule` mit w als isoliertes Literal:
lösche alle Klauseln, in der w vorkommt
$\rightsquigarrow F = \{\{\neg y, z\}, \{y, \neg z\}\}$
`Decide`:

- Fall 1: $F|_{\neg y}$: $\neg y$ ist nun **decision-literal**
  $\rightsquigarrow F = \{\{\neg z\}\}$
  `UnitPropagate` mit $v = \neg z$:
  $\rightsquigarrow F = \{\ \}$
  F ist die leere Formel $\Rightarrow F$ ist erfüllbar und das Modell $M = w, \neg y, \neg z$

[6] falls die Formel mehr als ein Modell besitzt, kann dieser Fall auch untersucht werden

- Fall 2: $F|_y$
  wird nicht mehr untersucht.

(d) $F = \{\{\neg w, \neg x\}, \{x, y\}, \{\neg w, \neg y, z\}, \{x, \neg y, \neg z\}\}$
`Decide`:

- Fall 1: $F|_w$: w ist nun **decision-literal**
  $\rightsquigarrow F = \{\{\neg x\}, \{x, y\}, \{\neg y, z\}, \{x, \neg y, \neg z\}\}$
  `UnitPropagate` mit $v = \neg x$:
  $\rightsquigarrow F = \{\{y\}, \{\neg y, z\}, \{\neg y, \neg z\}\}$
  `UnitPropagate` mit $v = y$:
  $\rightsquigarrow F = \{\{z\}, \{\neg z\}\}$
  `UnitPropagate` mit $v = z$:
  $\rightsquigarrow F = \{\ \{\ \}\}$
  F enthält die leere Klausel, dem Algorithmus signalisiert das bei einem vorhandenem decision-literal ein **Konflikt**. Der Algorithmus läuft jetzt beim branching in den Else-Zweig. An dieser Stelle erfolgt ein Backtrack, d.h. alles bis zum letzten decision-literal w wird revidiert
  $\rightsquigarrow F = \{\{\neg w, \neg x\}, \{x, y\}, \{\neg w, \neg y, z\}, \{x, \neg y, \neg z\}\}$
  das letzte decision-literal w wird zu $\neg w$ umgekehrt (siehe unten Fall 2):
- Fall 2: $F|_{\neg w}$
  $\rightsquigarrow F = \{\{x, y\}, \{x, \neg y, \neg z\}\}$
  `Decide`:
  - Fall 2a: $F|_x$:
    $\rightsquigarrow F = \{\ \}$
    F ist die leere Formel $\Rightarrow F$ ist erfüllbar und unter dem Modell $M = \neg w, x$ können y und z beliebig gesetzt werden, da sie die Erfüllbarkeit der gesamten Formel nicht beeinflussen.
  - Fall 2b: $F|_{\neg x}$
    wird nicht mehr untersucht.

4.3 Das Abstrakte DPLL Framework

Dieser Abschnitt beginnt mit einer formalen Einführung in das *Abstrakte DPLL Framework* und dessen Übergangsystem. Nachdem der klassische DPLL Algorithmus mit Hilfe dieses Frameworks beschrieben wurde, wird im nächsten Kapitel der moderne DPLL Algorithmus zunächst ausführlich erklärt, und anschließend ebenfalls anhand des in diesem Abschnitt vorgestellten Frameworks definiert. Das abstrakte DPLL Framework des modernen DPLL Algorithmus dient dann als Basis für die Implementierung in Haskell.

4.3.1 Zustände und Übergangssysteme

Das DPLL-Verfahren kann abstrakt dadurch beschrieben werden, dass jeder Anwendungsschritt im DPLL einen Übergang von einem Zustand in einen anderen darstellt.
Ein Zustand hat dabei die Form eines Paares von $M \parallel F$, wobei F eine Formel in CNF-Darstellung ist und M eine (Teil-)Belegung. Präziser ausgedrückt stellt M die aktuelle Belegung von aussagenlogischen Variablen durch eine Folge von Literalen dar. Diese Folge darf die selbe Variable nicht gleichzeitig als *true* und *false* enthalten. Hierbei werden die Literale, für die eine Fallunterscheidung gemacht werden muss, mit d für *decision-literal* gekennzeichnet. Zu Beginn, also wenn noch keine Belegung stattgefunden hat, ist M leer, und wir bezeichnen die leere Folge mit $\emptyset$. Eine Klausel C ist eine *Konflikt-Klausel* in einem Zustand $M \parallel F,\ C$, wenn gilt $M \models \neg C$.
Wir modellieren jeden DPLL Schritt im Sinne einer Menge von Zuständen mit einer binären Relation $\Longrightarrow$ auf diesen Zuständen und bezeichnen die Relation als *Übergangsrelation*. Wenn $S \Longrightarrow S'$, dann sagen wir, dass es einen *Übergang* von S nach S' gibt. Weiterhin ist jede Sequenz von Übergängen der Form $S_0 \Longrightarrow S_1 \Longrightarrow \cdots$ eine *Ableitung* und jede Untersequenz eine *Unterableitung*.
Im Folgenden werden Übergangsrelationen durch bedingte *Übergangsregeln* definiert, die für einen gegebenen Zustand S genau definieren, ob es einen Übergang von S gibt, und wenn ja, zu welchem Zustand S'. Ein solcher Übergang bezeichnet einen *Anwendungsschritt* einer entsprechenden Regel.
Ein *Übergangssystem* ist eine Menge von Übergangsregeln, die auf einer gegebenen Menge von Zuständen definiert ist.

4.3.2 Klassische DPLL Prozeduren

Im Folgenden wird ein Übergangssystem für den *klassischen DPLL-Algorithmus* aus Abschnitt 4.2 anhand von fünf Übergangsregeln analog zu [NOT06] definiert.

Definition 4.3.1. Der *klassische DPLL-Algorithmus* ist ein Übergangssystem Cl bestehend aus den folgenden fünf Übergangsregeln. In diesem System werden alle Literale, die durch die Regel `Decide` zu M hinzugefügt werden als *decision-literale* bezeichnet, und diejenigen Literale, die durch alle anderen Regeln zu M hinzugefügt werden als *implied-literale* bezeichnet.

Die DPLL Prozedur besitzt fünf Regeln:

`UnitPropagate`:

$$M \mid\mid F,\ C \vee l \quad \Longrightarrow M\ l \mid\mid F,\ C \vee l \quad \textbf{wenn} \begin{cases} M \models \neg C \\ l \text{ ist undefiniert in } M \end{cases}$$

`PureLiteral`:

$$M \mid\mid F \quad \Longrightarrow M\ l \mid\mid F \quad \textbf{wenn} \begin{cases} l \text{ kommt in einer Klausel} \\ \text{von } F \text{ vor,} \\ \neg l \text{ kommt in } F \text{ nicht vor,} \\ l \text{ ist undefiniert in } M \end{cases}$$

`Decide`:

$$M \mid\mid F \quad \Longrightarrow M\ l^d \mid\mid F \quad \textbf{wenn} \begin{cases} l \text{ oder } \neg l \text{ kommt in einer} \\ \text{Klausel von } F \text{ vor,} \\ l \text{ ist undefiniert in } M \end{cases}$$

`Fail`:

$$M \mid\mid F,\ C \quad \Longrightarrow \mathit{FailState} \quad \textbf{wenn} \begin{cases} M \models \neg C \\ M \text{ enthält keine } \textit{decision-literale} \end{cases}$$

`Backtrack`:

$$M\ l^d\ N \mid\mid F,\ C \quad \Longrightarrow M\ \neg l \mid\mid F,\ C \quad \textbf{wenn} \begin{cases} M\ l^d N \models \neg C \\ N \text{ enthält keine } \textit{decision-literale} \end{cases}$$

Für die Erfüllbarkeit einer aussagenlogischen Formel F, kann das oben definierte Übergangsystem Cl herangezogen werden, indem eine beliebige Ableitung $\emptyset \mid\mid F \Longrightarrow_{Cl} \cdots \Longrightarrow_{Cl} S_n$ generiert wird, wobei S_n ein Endzustande bezüglich Cl darstellt. Die Anwendbarkeit jeder dieser fünf Regeln ist einfach zu überprüfen, und ihre Anwendung führt immer zu einer endlichen Ableitung. An dieser Stelle wird auf die folgende Literatur [NOT06] verwiesen. Weiterhin gilt für jede Ableitung, die in einen Endzustand S_n landet

(i) F ist unerfüllbar $\Longleftrightarrow$ S_n ist ein *Failstate* (unerfüllbarer Zustand) und,

(ii) wenn S_n ist von der Form $M \mid\mid F \Longrightarrow M$ ist ein Modell von F

Erwähnenswert ist, dass bei diesem klassischen DPLL System in der zweiten Komponente eines Zustands keine Klauseln hinzukommen, eine Eigenschaft, die im nächsten noch vorzustellenden Übergangssystem des modernen DPLL Algorithmus nicht

mehr gilt.
Wir geben nun kurz einige Bemerkungen zu den einzelnen DPLL Regeln an. Wenn M im Folgenden eine Folge der Form $M_0 l_1 M_1 \cdots l_k M_k$ ist, wobei die l_i *decision-literale* in M darstellen, dann sagen wir, dass sich der Zustand $M \mid\mid F$ im *decision-level* k befindet und alle Literale $l_i M_i$ *zu dem decision-level* i *gehören.*

—`UnitPropagate`: Wenn es eine Klausel $C \vee l$ gibt, sodass $M \models \neg C$, d.h. C ist *false*, dann muss l *true* sein, mit anderen Worten muss gelten: $F \models \neg C$, damit $C \vee l$ eine Unit-Klausel ist. Also kann man l zu M (ohne Alternative) hinzunehmen.

—`PureLiteral`: Wenn l in F vorkommt, $\neg l$ aber nicht, d.h. l ist ein isoliertes Literal, dann kann man l als *true* annehmen. Wenn l also in M undefiniert ist, dann kann es um l als *true* erweitert werden. Im Fall der Erfüllbarkeit gibt es auch ein Modell, in der l als *True* definiert ist.

—`Decide`: Diese Regel repräsentiert eine Fallunterscheidung, sie wird angewendet, wenn keine andere Regel anwendbar ist. Ein undefiniertes Literal wird aus F ausgewählt und zu M hinzugefügt. Dieses Literal wird als *decision-literal* bezeichnet, damit wird gekennzeichnet, dass noch eine Alternative $M\ \neg l$ betrachtet werden kann, falls $M\ l$ nicht zu einem Modell führt. Dieses geschieht im Rahmen der `Backtrack` Regel.

—`Fail`: Falls M keine desision-literale enthält, entdeckt diese Regel eine Konflikt-Klausel C und landet in den *FailState* Zustand, d.h. F ist unerfüllbar.

—`Backtrack`: Die `Backtrack` Regel wird angewendet, falls eine Konflikt-Klausel C mit $M \models \neg C$ entstanden ist, und die Regel `Fail` nicht anwendbar ist. Alle Literale in N sind Folgerungen aus $M\ l^d$, also kann man bis zum letzten *decision-literal* zurückgehen und sich dort anders entscheiden, d. h. das letzte *decision-literal* l^d wird mit $\neg l$ ersetzt und alle Belegungen N, die nach dieser Entscheidung getroffen wurden, werden gelöscht. Beachte, dass $\neg l$ jetzt kein *decision-literal* mehr sein darf, da die mögliche Alternative l bereits betrachtet wurde.

Im Folgenden zeigen wir ein Beispiel einer Ableitung im klassischen DPLL System, in der jeder Übergang durch die entsprechende Regel gekennzeichnet ist. Es sei erwähnt, dass in dieser Darstellung die Formel F unverändert bleibt, was aber in der späteren Implementierung nicht der Fall sein wird, da dann durch Streichen von Klauseln und Literalen in F eine leere Klausel oder die leere Formel festgestellt wird. Weiterhin werden, um die Lesbarkeit zu verbessern, natürliche Zahlen für jedes Atom verwendet, und für das zu l komplementäre Literal schreiben wir $\bar{l}$, d.h. $\bar{l} = \neg l$.

Beispiel 4.3.2.

$\emptyset$	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	$\Longrightarrow$ `Decide`
1^d	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	$\Longrightarrow$ `UnitPropagate`
$1^d\ 2$	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	$\Longrightarrow$ `Decide`
$1^d\ 2\ 3^d$	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	$\Longrightarrow$ `UnitPropagate`
$1^d\ 2\ 3^d\ 4$	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	$\Longrightarrow$ `Decide`
$1^d\ 2\ 3^d\ 4\ 5^d$	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	$\Longrightarrow$ `UnitPropagate`
$1^d\ 2\ 3^d\ 4\ 5^d\ \overline{6}$	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	$\Longrightarrow$ `Backtrack`
$1^d\ 2\ 3^d\ 4\ \overline{5}$	$\|\|$	$\overline{1} \vee 2$,	$\overline{3} \vee 4$,	$\overline{5} \vee \overline{6}$,	$6 \vee \overline{5} \vee \overline{2}$	

Vor der `Backtrack` Regel kommt es in der Klausel $6 \vee \overline{5} \vee \overline{2}$ zu einem Konflikt. Die Regel `Fail` ist nicht anwendbar, da die Belegung M noch *decision-literale* enthält. Jetzt findet die Backtrack Regel ihre Anwendung, das bedeutet, dass wir bis zum letzten *decision-literal* 5^d zurückgehen und die Belegung $\overline{6}$ löschen, die nach dieser Entscheidung getroffen wurde und ersetzen 5^d mit $\overline{5}$. Der letzte Zustand dieser Ableitung ist zwar partial, weil die Variable 6 noch undefiniert ist, jedoch spielt das für die Erfüllbarkeit der Formel F als ganzes keine Rolle mehr. Die Variable 6 kann daher entweder mit *true* oder *false* belegt werden.

5 Moderner DPLL Algorithmus mit nicht-chronologischem-backtracking

Wie bereits erwähnt basieren die Mehrzahl von Algorithmen auf dem originalen Backtrack-Suchalgorithmus von Davis, Logemann und Loveland [DLL62]. Die meisten SAT-Solver dieser Art bestehen konzeptionell aus den drei Hauptprozessen: dem *decision process*, dem *deduction process* und dem *backtrack process* (siehe dazu auch 4.2.3). Der decision process wählt während eines Verzweigungsschrittes eine Variable aus und weist ihr einen Wahrheitswert zu (siehe `Decide` Prozedur aus dem klassischen DPLL System). Der deduction process erkennt logische Konsequenzen, d.h. implizierte Belegungen, die aus jeder ausgewählten Variablenzuweisung des decision process resultieren (siehe `UnitPropagate` Prozedur aus dem klassischen DPLL System). Und schließlich implementiert der backtrack process den Backtrack Schritt des Algorithmus, der immer dann erfolgt, wenn der decision process oder der deduction process einen Konflikt verursacht haben (siehe `Fail` und `Backtrack` Prozedur aus dem klassischen DPLL System).
Der wesentlich Unterschied zwischen dem klassischen DPLL und dem modernen DPLL System liegt dabei im backtrack process. Während der klassische DPLL Algorithmus im backtrack process nur ein chronologisches-backtracking ausführt, verwenden moderne DPLL Algorithmen ein *nicht-chronologisches-backtracking* Verfahren. Im nächsten Abschnitt erklären wir zunächst unformal, was man darunter versteht.

5.1 Nicht-chronologisches-backtracking

Wir betrachten im Folgenden denjenigen Hauptprozess in einem DPLL Algorithmus, in dem ein Konflikt (Konflikt-Klausel) entsteht. Der Solver wird in dieser Phase ausgeführte Entscheidungen revidieren und die Suche in eine andere Richtung leiten. Dazu bedienen sich moderne DPLL Algorithmen einer sogenannten *Konflikt-Analyse* Prozedur mit der Aufgabe den Grund für einen Konflikt zu finden und ihn aufzulösen. Konflikt-Analyse teilt dabei dem SAT-Solver mit, dass es in einer bestimmten Verzweigung im Entscheidungsbaum keine Lösung für das Problem gibt und leitet

den Algorithmus in einen neuen Suchraum ein, in der die Suche dann fortgeführt wird.

Der klassische DPLL Algorithmus merkt sich für jedes decision-literal anhand eines Flags, ob ein decision-literal schon in beiden Belegungen im Modell definiert wurde oder nicht (ob ein decision-literal in seiner Belegung umgekehrt wurde oder nicht). Wenn nun ein Konflikt entsteht, schaut der backtrack process nach der zuletzt gemachten Entscheidung (letztes decision-literal), kehrt das Literal in seiner Belegung um, welche noch nicht umgekehrt wurde, markiert diese als „umgekehrt“ und löscht alle Belegungen aus dem Modell, die nach diesem decision-level definiert wurden. Danach untersucht es die komplementäre Belegung des Literals. Diese Methode bezeichnet man auch als *chronologisches-backtracking*, weil immer die zuletzt getroffene Entscheidung bzw. das letzte decision-literal , welches noch nicht umgekehrt wurde, revidiert wird. Chronologisches-backtracking funktioniert gut für zufällig generierte Formeln. Für strukturierte Probleme jedoch (wie sie normalerweise für Probleme aus der Praxis generiert werden) ist das chronologische-backtracking nicht sehr effizient, was die Einschränkung des Suchraumes betrifft.

Moderne DPLL Algorithmen erweitern daher den backtrack process um eine Konflikt-Analyse Prozedur, die die Ursachen für einen Konflikt untersucht. Diese Methode ist zusätzlich in der Lage mehr als nur die letzte Entscheidung zu revidieren. Daher heißt dieses Verfahren auch *nicht-chronologisches-backtracking*. Des Weiteren wird während der Konflikt-Analyse in modernen DPLL Algorithmen die Ursache des aktuellen Konflikts in Form von Klauseln festgehalten und zur Original-Formel hinzugefügt. Die hinzugefügten Klauseln ändern dabei die (Un)Erfüllbarkeit der Formel nicht, sie dienen vielmehr dazu den Suchraum effizienter einzuschränken. Dieser Mechanismus wird als *conflict-driven-learning* bezeichnet. Solche zur Formel hinzugefügten oder gelernten Klauseln bezeichnet man auch als *conflict-induced-clauses* oder *Backjump-Klauseln*. Die Konflikt-Analyse zusammen mit dem Lernen von Backjump-Klauseln wird auch als *Konflikt-Analyse und Lernen* zusammengefasst, auf die wir dann im Abschnitt 5.2 genauer eingehen werden.

Nicht-chronologisches-backtracking wurde erstmals im Zusammenhang mit dem *Constraint Satisfaction Problem* (CSP) eingeführt [Pro93]. Die Integration dieses Mechanismus in einem SAT-Solver gelang J. P. Marques-Silva und K. A. Sakallah [MSS99] mit GRASP[1] sowie R. Bayardo und R. Schrag mit dem REL_SAT Solver [JS97]. Diese Technik des nicht-chronologischen-backtrackings trug wesentlich zur Effizienzsteigerung in modernen SAT-Solvern bei, sodass sich kurz nach GRASP veröffentliche Solver wie SATO [Zha97] und CHAFF [MMZ+01] auf ähnliche Konflikt-Analyse Prozeduren beruhten. Der Vollständigkeit halber sei an dieser Stelle erwähnt, dass SATO und CHAFF neben der im GRASP beschriebenen Konflikt-Analyse, unter anderem

[1] SAT-Solver steht für Generic seaRch Algorithm for the Satisfiability Problem

zusätzlich effizientere Datenstrukturen verwenden mit dem Ziel die Rechenzeit der Suche pro Knoten im Entscheidungsbaum zu reduzieren. Beispiele für solche effiziente Datenstrukturen sind die *head/tail Listen* in SATO oder etwa die *two watched literals* in CHAFF [MMZ+01].

Nicht-chronologisches-backtracking ist in modernen SAT-Solvern eng verknüpft mit dem Ableiten von einer oder mehreren sogenannten Backjump-Klauseln. Das Identifizieren einer Backjump-Klausel geschieht dabei durch die Analyse eines sogenannten *Implikationsgraphen*. Dieser hält die durch den decision process ausgewählten Variablenbelegungen und die durch den deduction process implizierten Literale in Form eines gerichteten azyklischen Graphen (DAG)[2] fest. Der Implikationsgraph liefert der Konflikt-Analyse zusätzlich neben der Information, welche Entscheidung zu welcher notwendigen Implikation geführt hat, auch die Information zu welchem decision-level dies geschehen ist. Im Rahmen der Konflikt-Analyse ist der Implikationsgraph daher eine wichtige Datenstruktur, um die Ursachen für ein Konflikt zu finden. Wir werden den Implikationsgraphen noch im weiteren Verlauf im Abschnitt 5.2.1 formal definieren.

5.2 Konflikt-Analyse und Lernen

Der Algorithmus in Abbildung 5.1 stellt die Idee eines DPLL-basierten SAT-Solvers mit Konflikt-Analyse und Lernen auf höchster Abstraktionsebene dar. Die Prozedur `Decide` wählt zunächst eine Variable für das Belegen mit einem Wahrheitswert aus. Die Prozedur `UnitPropagate` wendet die Unit-Propagation an, gleichzeitig bemerkt sie das Entstehen einer leeren Klausel. Wenn alle Klauseln erfüllt sind, deklariert sie die Formel als erfüllt. Die Prozedur `AnalyzeConflict` analysiert die Struktur der Implikationen anhand des Implikationsgraphen und berechnet daraus eine zu lernende Backjump-Klausel. Sie berechnet außerdem einen *backjump-level* blevel, in der die Suche anschließend fortgesetzt wird. Es sei an dieser Stelle nochmals daran erinnert, dass beim nicht-chronologischen-backtracking die Möglichkeit besteht, die Suche gegebenenfalls in einem höheren level als nur im vorletzten level des Entscheidungsbaums fortzusetzen.

Wiederholend halten wir fest, dass Variablen, die durch `Decide` zugewiesen werden, als *decision-literale* und diejenigen, die durch `UnitPropagate` zugewiesen werden als *implied-literale* bezeichnet werden. Bei jedem `Decide` wird der decision-level des Literals um eins inkrementiert, was damit gleichzusetzen ist, dass die Suche mit jeder neuen Verzweigung im Entscheidungsbaum in einer Ebene tiefer fortgesetzt wird. Daher ist der *decision-level eines decision-literals* x *bzw.* $\neg x$ um eins größer als die

[2] directed acyclic graph

Anzahl des aktuellen decision-literals zum Zeitpunkt der Verzweigung von x. Der *decision-level eines implied-literals* y ist das Maximum der decision-level aus den decision-literalen, durch die y impliziert wurde. Er wird in Abschnitt 5.2.1 in (5.1) formal definiert. Falls y ohne ein decision-literal impliziert wurde, dann hat y den decision-level 0. Der decision-level jedes beliebigen Schrittes in dem zugrunde liegenden DPLL Algorithmus entspricht dem Maximum der decision-level aller aktuellen decision-literale, und der decision-level ist 0, falls noch kein decision-literal existiert. Falls beispielsweise der Algorithmus mit einer ersten Verzweigung bei x beginnt, dann hat x den decision-level 1, und der Algorithmus befindet sich an dieser Stelle auf decision-level 1. Die genannten Definitionen zur Unterscheidung eines decision- und implied-literals werden uns später bei der Implementierung nützlich sein, was die Datenstruktur des Modells M betrifft. Denn wir müssen dann nicht nur die Wahrheitsbelegung jeder definierten Variable kennen, sondern uns auch deren zugehörige level merken, und, ob sie durch `Decide` zugewiesen oder durch `UnitPropagate` impliziert wurden.

Input : Belegung $M = \emptyset$ und F in CNF-Darstellung.
Output: UNSAT, oder SAT und ein Modell, welches F erfüllt.

```
begin
    while TRUE do
        Decide
        while TRUE do
            status ← UnitPropagate
            if status = KONFLIKT then
                blevel ← AnalyzeConflict
                if blevel = 0 then return UNSAT
                Backjump(blevel)
            else if status = SAT then
                Output M
                return SAT
            else break
        end
    end
end
```

Abbildung 5.1: Algorithmus DPLL-AnalyzeConflict(M, F)

Der Algorithmus in 5.1 stoppt und deklariert eine Formel als UNSAT, sobald die Unit-Propagation im decision-level 0 auf einen Konflikt trifft, z.B. dann, wenn noch keine Variablenzuweisung durch `Decide` stattgefunden hat. Diese Bedingung wird manchmal auch als *Konflikt im decision-level 0* bezeichnet.
Im Allgemeinen soll der Lernprozess, der sich hinter `AnalyzeConflict` verbirgt, sicherstellen, dass identische Variablenbelegungen, die zu einem Konflikt geführt ha-

ben, nicht noch einmal berechnet werden. Es existieren unterschiedliche Variationen Backjump-Klauseln anhand des Implikationsgraphen auszuwählen (siehe Abschnitt 5.2.5 verschiedene Lernschemata). Mitunter ist es auch möglich mehrere Backjump-Klauseln auf der Grundlage eines einzelnen Konflikts zu lernen. Wir geben im nächsten Unterabschnitt eine formale Definition des Implikationsgraphen an.

5.2.1 Der Implikationsgraph

Sei $M \,||\, F$ ein Zustand des modernen DPLL Algorithmus. Betrachte eine Formel F und die Klausel $\omega = (x \vee \neg y)$ mit der Annahme, dass $y = \mathit{true}$ in M. Damit nun die Klausel $\omega = (x \vee \neg y)$ erfüllbar wird, ist erforderlich, dass $x = \mathit{true}$ ist in ω (x muss in M mit x definiert werden), wir sagen, $y = \mathit{true}$ impliziert $x = \mathit{true}$ in ω. Allgemein gilt, falls eine Unit-Klausel $(l_1 \vee \cdots \vee l_k)$ in F mit dem undefinierten Literal l_j existiert, dann muss l_j notwendigerweise zu *true* evaluieren, d.h. wenn $l_j = x$ in ω, dann muss die Variable als x in M definiert werden. Falls $l_j = \neg x$ in ω, dann muss die Variable x in M als $\neg x$ definiert werden. Solche zwingenden Belegungen nennen wir im Folgenden *Implikationen*.
Sei $x \in M$. Dann ist $A^{\omega}(x)$, die vorangehende Belegung von x, bezüglich einer Klausel ω. Es definiert die übrigen Variablenbelegungen für die Literale in ω ohne die Variablenbelegung von x. Intuitiv bezeichnet $A^{\omega}(x)$ diejenigen Variablenbelegungen, die für das Implizieren der Variable x in ω verantwortlich sind.
Betrachten wir ein Beispiel: die vorangehende Belegung von z, d.h. $A^{\omega}(z)$ bzgl. der Klausel $\omega = (x \vee y \vee \neg z)$ ist $A^{\omega}(z) = \{\neg x, \neg y\}$. Beachte, dass die Menge der vorangehenden Belegung eines decision-literals immer leer ist.

Definition 5.2.1. Ein *Implikationsgraph* I zu einem beliebigen Zustand $M \,||\, F$ im modernen DPLL Algorithmus ist ein *gerichteter azyklischer Graph*, wobei

1. für jede Belegung $x \in M$ gilt: jeder Knoten in I entspricht einer Variablenbelegung x, d.h. $x = \mathit{true}$, bzw. $\neg x$, d.h. $x = \mathit{false}$ (siehe Abbildung 5.2). Jede Variablenbelegung in einem Knoten ist zusätzlich mit seinem decision-level in Klammern gekennzeichnet.

2. die Vorgängerknoten von x in I sind die vorangehenden Belegungen $A^{\omega}(x)$, die aus der Unit-Klausel ω stammen und zu der Implikation von x geführt haben. Die gerichteten Kanten vom Knoten $A^{\omega}(x)$ nach x werden mit ω gekennzeichnet. Knoten, die keine Vorgänger haben sind decision-literale oder implizierte Literale aus originalen Unit-Klauseln (haben immer einen decision-level von 0).

3. zusätzlich wird ein spezieller Konfliktknoten κ zu I hinzugefügt, der auf einen Konflikt hindeuten soll. Die Vorgänger eines Konfliktknotens κ entsprechen

der *Konfliktbelegung* (Variablenbelegungen, die dazu geführt haben, dass die Klausel ω falsifiziert wurde). Sie werden mit $A^{\omega}(\kappa)$ bezeichnet. Die gerichteten Kanten von Knoten aus $A^{\omega}(x)$ nach κ sind ebenfalls alle mit ω gekennzeichnet.

Mit $A(x)$ bezeichnen wir die Menge der direkten Vorgänger eines Knotens x im Implikationsgraphen. In einem Implikationsgraph ohne Konflikt gibt es zu jeder Variable einen Knoten. Ein Konflikt entsteht, wenn im Implikationsgraph versucht wird, einen Knoten für eine Variable sowohl in positiver als auch in negativer Belegung einzufügen, bzw. umgekehrt.

Seien die folgenden Variablenbelegungen in M ein Ausschnitt aus einer Sequenz von Implikationen, wobei jede Variable x im Knoten mit seinem decision-level in Klammern gekennzeichnet wird.

aktuelle Variablenbelegung in M:	$\{\neg x_9\ (1), \neg x_{10}\ (3), \neg x_{11}\ (3), x_{12}\ (2), x_{13}\ (2), \ldots\}$
aktuelles decision-literal:	$\{x_1\ (6)\}$

Weiterhin seien folgende Klauseln ein Ausschnitt aus der Formel F

$\omega_1 = (\neg x_1 \vee x_2)$	$\omega_6 = (\neg x_5 \vee \neg x_6)$
$\omega_2 = (\neg x_1 \vee x_3 \vee x_9)$	$\omega_7 = (\neg x_1 \vee x_7 \vee \neg x_1 2)$
$\omega_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$	$\omega_8 = (\neg x_1 \vee x_8)$
$\omega_4 = (\neg x_4 \vee x_5 \vee x_{10})$	$\omega_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
$\omega_5 = (\neg x_4 \vee x_6 \vee x_{11})$	$\cdots$

Der dazugehörige Ausschnitt des Implikationsgraphen sieht dann wie folgt aus:

Dabei wird der decision-level eines implizierten Literals x mit seinen vorangehenden Belegungen $A^{\omega}(x)$ wie folgt in Beziehung gesetzt:

$$\delta(x) = max\{\delta(y) \mid y \in A^{\omega}(x)\} \tag{5.1}$$

5.2.2 Berechnen der Backjump-Klausel

Wenn ein Konflikt entsteht wird die Sequenz von Implikationen, die auf ein Konfliktknoten κ zusammenlaufen, analysiert um diejenigen Variablenbelegungen oder Konfliktbelegungen zu bestimmen, die F falsifizieren, d.h. die unmittelbar für den Konflikt verantwortlich sind. Dabei stellt die Disjunktion solcher Konfliktbelegungen eine hinreichende Bedingung für das Entstehen des Konfliktes dar. Somit liefert die Disjunktion der Negation der Konfliktbelegungen eine Klausel, die in der Formel F

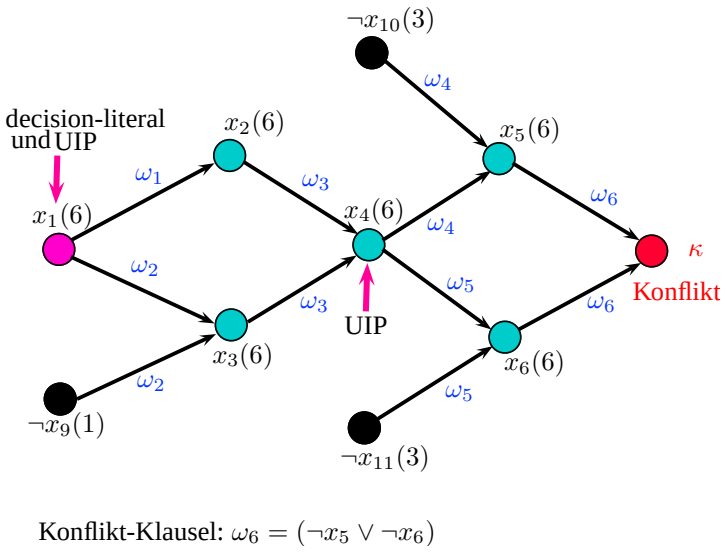

Abbildung 5.2: Aussschnitt aus einem Implikationsgraph

nicht existiert. Wir bezeichnen die Konfliktbelegung des zugehörigen Konfliktknotens κ mit $A^{\omega c}(\kappa)$ und die zugehörige Klausel $\omega c(\kappa)$ im Folgenden als Backjump-Klausel.
Die Konfliktbelegung wird beginnend vom Konfliktknoten durch ein rückwärts Travesieren des Implikationsgraphen ermittelt. Um nun die Konfliktbelegungen $A^{\omega c}(\kappa)$ zu bestimmen, partitionieren wir die vorangehenden Belegungen von κ und die Variablenbelegungen des aktuellen levels in zwei Mengen. Sei x entweder κ oder eine Variable, die zum aktuellen decision-level belegt worden ist. Die Partition von $A(x)$ ist dann gegeben durch:

$$\begin{aligned} \Lambda(x) &= \{y \in A(x) \mid \delta(y) < \delta(x)\} \\ \Sigma(x) &= \{y \in A(x) \mid \delta(y) = \delta(x)\}. \end{aligned} \tag{5.2}$$

Bezogen auf den Implikationsgraph in Abbildung 5.2, ist beispielsweise $\Lambda(x_6) = \{\neg x_{11}\ (3)\}$ und $\Sigma(x_6) = \{x_4\ (6)\}$. Die Berechnung der Konfliktbelegung $A^{\omega c}(\kappa)$ kann jetzt ausgedrückt werden durch:

$$A^{\omega c}(\kappa) = causes\_of(\kappa)$$

wobei sich $causes\_of(.)$ wie folgt definiert:

$$causes\_of(\kappa) = \begin{cases} x & \text{wenn } A(x) = \emptyset \\ \Lambda(x) \cup \left[\bigcup\limits_{y \in \Sigma(x)} causes\_of(y) \right] & \text{sonst.} \end{cases} \tag{5.3}$$

Neben Belegungen mit kleineren decision-level als die des letzten decision-levels, wird auch die Belegung des letzten decision-literals in $A^{\omega c}(\kappa)$ mit einbezogen. Dieses ist dadurch gerechtfertigt, dass die Belegung des letzten decision-literals unmittelbar für das Implizieren der Variablen auf demselben level verantwortlich ist. Daher ist die Belegung des letzten decision-literals neben den Belegungen aus früheren leveln eine hinreichende Bedingung für den Konflikt. Die entsprechende Backjump-Klausel zu $A^{\omega c}(\kappa)$ kann jetzt wie folgt bestimmt werden:

$$\omega c(\kappa) = \bigvee_{x \in A^{\omega c}(\kappa)} x, \tag{5.4}$$

wobei, für die binäre Variablenbelegung x gilt:$\neg x \equiv x$, und $x \equiv \neg x$, d.h. die Backjump-Klausel $\omega c(\kappa)$ besteht aus einer Disjunktion der Negation der berechneten Konfliktbelegungen $A^{\omega c}(\kappa)$. Wenn man (1.2)-(1.4) auf den Konflikt in Abbildung 5.2 anwendet, so erhält man in decision-level 6 die folgende Konfliktbelegung $A^{\omega c}(\kappa)$ mit der dazugehörigen Backjump-Klausel $\omega c(\kappa)$:

$$\begin{array}{ll} A^{\omega c}(\kappa) & = \{x_1\ (6), \neg x_9\ (1), \neg x_{10}\ (3), \neg x_{11}\ (3)\} \\ \omega c(\kappa) & = (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11}). \end{array} \tag{5.5}$$

Das Berechnen von Backjump-Klauseln $\omega c(\kappa)$ ermöglicht es (durch das Ableiten weiterer Variablenbelegungen, die sich ohne $\omega c(\kappa)$ nicht unmittelbar abgeleitet hätten) die Suche im Entscheidungsbaum zu reduzieren. Insbesondere soll durch das Hinzufügen von $\omega c(\kappa)$ zur Formel F verhindert werden, dass eine Konfliktbelegung, die zum gegenwärtigen Konflikt geführt hat, ein weiteres Mal abgeleitet wird.

5.2.3 Berechnen des backjump-levels

Wenn $\omega c(\kappa)$ nur Literale enthält, die einen decision-level d oder kleiner d enthalten, wobei d der aktuelle decision-level ist, dann kann man gemäß dem *conflict-directed-backjumping* (CBJ) [Pro93][3] direkt nach decision-level d springen. In einem solchen Fall kann in einem level nach dem höchsten decision-level der Literale in $\omega c(\kappa)$ und der gleichen Variablenbelegung in M die Formel F nicht mehr erfüllt werden.

Daher kann der backjump-level β gemäß [MSS99] wie folgt berechnet werden:

$$\beta = max\{\delta(x) \mid x \in A^{\omega c}(\kappa)\} \tag{5.6}$$

Wenn $\beta = d - 1$, dann führt die Suche ein *chronologisches*-backtracking zum direkt vorangehenden decision-level aus, und wenn $\beta < d - 1$, dann macht die Suche ein *nicht-chronologisches*-backtracking, indem es im Entscheidungsbaum über

[3] erstmals im Zusammenhang mit dem Constraint Satisfaction Problem erwähnt

mehrere level hinweg springt. Beachte, dass alle Variablenbelegungen, die (vor einem backtracking) in M nach decision-level β definiert wurde, eine kurz zuvor ermittelte Backjump-Klausel $\omega c(\kappa)$ falsifizieren würden. Daher werden nach einem nicht-chronologischen-backtracking alle Variablenbelegungen, die nach dem decision-level β stattgefunden haben, aus der Belegung M gelöscht. Eine Suche die nur ein chronologisches-backtracking ausführt, verschwendet im Allgemeinen eine Menge Zeit damit, nutzlose Regionen im Suchraum zu durchsuchen.

5.2.4 Unique Implication Points

Bei einer genaueren Analyse des Implikationsgraphen I lässt sich eine weitere Verbesserung in der Konflikt-Analyse Prozedur erreichen, indem man kürzere Backjump-Klauseln ermittelt. Wir definieren zunächst einmal den Begriff der *Dominatoren* [Tar74]. In einem Implikationsgraph I dominiert ein Knoten a einen Knoten b, genau dann, wenn es einen Pfad von dem aktuellen decision-literal von a nach b durch a gibt. Angenommen ein Konflikt entsteht, sei $U = \{u_1, \cdots, u_k\}$ die Menge der Dominatoren des Konfliktknotens κ bezüglich des aktuellen decision-literals im decision-level d. Jedes u_i entspricht dabei einem sogenannten *unique implication point* kurz UIP. Ein UIP liegt also auf dem Pfad des aktuellen decision-literals d zum Konfliktknoten κ, d.h. ein UIP ist ein Knoten mit dem aktuellen decision-level, der den Konfliktknoten κ dominiert. Um die Anwendung der UIPs zu verdeutlichen, schauen wir uns noch einmal den Implikationsgraphen in Abbildung 5.2 an: die Menge der Dominatoren des Konfliktknotens κ bezüglich des aktuellen decision-literals x_1 ist dann $\{x_1, x_4\}$, da x_1 und x_4 beide den Konfliktknoten κ dominieren. Intuitiv ist ein UIP eine *Einzel*-Ursache, die den Konflikt zum aktuellen decision-level impliziert hat. Das decision-literal ist immer ein UIP. Zusammen mit den vorangehenden Belegungen $\neg x_{10}$ und $\neg x_{11}$ stellt der UIP x_4 daher eine hinreichende Bedingung dar, um den selben Konflikt in Abbildung 5.2 zu implizieren. Daher ist die Klausel $(\neg x_4 \vee x_{10} \vee x_{11})$ ebenfalls eine mögliche Backjump-Klausel, welche noch nicht in der Formel existiert. Diese Klausel ist in der Tat kürzer als die in (5.5) berechnete Backjump-Klausel.
Das Verfahren zum Berechnen von kürzeren Backjump-Klauseln kann für jedes UIP aus $U = \{u_1, \cdots, u_k\}$ und einem Konflikt κ durch das Modifizieren von (5.3) wie folgt definiert werden:

$$causes\_of(x, u_i) = \begin{cases} u_i & \text{wenn } x = u_i \text{ oder } A(x) = \emptyset \\ \Lambda(x) \cup \left[\bigcup_{y \in \Sigma(x)} causes\_of(y, u_i) \right] & \text{sonst,} \end{cases} \tag{5.7}$$

wobei $u_i \in U$ und $causes\_of(x, u_i)$ jetzt als die Menge der vorangehenden Belegungen von x ist, aufgrund von u_i und den übrigen Belegungen, die x aus früheren decision-leveln implizieren. Die Backjump-Klausel kann jetzt für jedes adjazente Paar von UIPs (u_{i-1}, u_i) mit $i = 2, \cdots, k$ wie folgt ermittelt werden:

$$\omega c(u_{i-1}, u_i) = \left[\bigvee_{x \in causes\_of(u_{i-1}, u_i)} x \right] \vee u_i \tag{5.8}$$

Des Weiteren wird die Backjump-Klausel für den vom Konfliktknoten ausgehend ersten UIP u_k und dem Konfliktknoten κ wie folgt definiert:

$$\omega c(\kappa, u_k) = \bigvee_{x \in causes\_of(\kappa, u_k)} x, \tag{5.9}$$

wobei hier in beiden Fällen (5.8) und (5.9) analog zu (5.4) für die binäre Variablenbelegung x gilt: $\neg x \equiv x$, und $x \equiv \neg x$ genauso wie für die UIPs $\neg u_i \equiv u_i$, und $u_i \equiv \neg u_i$.
Das Lernen der Backjump-Klausel wie sie in (5.9) berechnet wird, ist auch unter dem Namen *first UIP Lernschema* bekannt. Im nächsten Abschnitt werden wir verschiedene Lernschemata erläutern und in diesem Zusammenhang das first UIP Lernschema nochmals aufgreifen.

5.2.5 Verschiedene Lernschemata

Bei genauerer Betrachtung ergibt sich eine Backjump-Klausel durch eine Aufteilung des Implikationsgraphen in zwei Teile. Dabei ist der Implikationsgraph so aufgeteilt, dass sich alle decision-literale auf der einen Seite (*reason-side*) und die Konfliktbelegung auf der anderen Seite (*conflict-side*) befinden. Alle Knoten auf der reason-side mit mindestens einer Kante zur conflict-side stellen eine Ursache für das Entstehen des Konfliktes dar. Die Negation der Literale aus eben solchen Knoten, bilden die Backjump-Klausel. Wir bezeichnen die Zweiteilung des Implikationsgraphen im Folgenden als *cut*. Unterschiedliche cuts entsprechen verschiedenen Lernschemata. In Abbildung 5.3 kann die Klausel $(\neg x_1 \vee \neg x_3 \vee x_5 \vee x_{17} \vee \neg x_{19})$ entsprechend dem cut 1 als Backjump-Klausel herangezogen werden. Genauso wie cut 2, welche in der Abbildung 5.3 der Backjump-Klausel $(x_2 \vee \neg x_4 \vee \neg x_8 \vee x_{17} \vee \neg x_{19})$ entspricht. Gegenstand dieses Unterabschnittes ist es, verschiedene Lernschemata aus bestehenden SAT-Solvern vorzustellen und letzendlich auf der Basis von experimentellen Ergebnissen aus [ZM01] ein robustes und effizientes Lernschema aufzuzeigen, welche auch im Rahmen dieser Diplomarbeit implementiert und getestet wird.

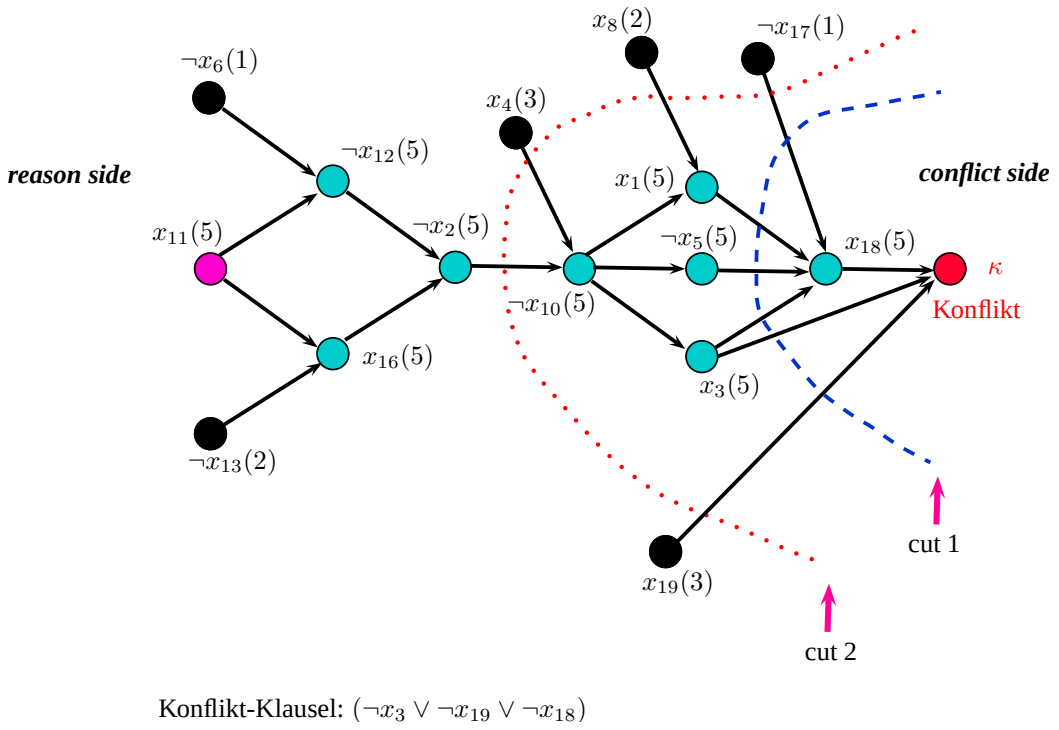

Abbildung 5.3: Aufteilung des Implikationsgraphen durch verschiedene cuts

GRASP [MSS99] ist neben REL_SAT [JS97] einer der ersten SAT-Solver, in der das nicht-chronologische-backtracking implementiert wurde. GRASP versucht soviel wie möglich aus einem Konflikt zu lernen, indem es mehr als nur eine Backjump-Klausel lernt. Das hat den Vorteil, dass es dadurch weniger Verzweigungsschritte macht, aber dafür unter Umständen beim Lernen und der Unit-Propagation an Rechenzeit verliert (siehe experimentelle Ergebnisse in [ZM01]). In [ZM01] wird das GRASP *Lernschema* dabei wie folgt beschrieben:

Bei dem Eintreten eines Konfliktes werden zwei Fälle unterschieden. Wenn das aktuelle decision-literal ein *echtes* decision-literal ist (wird später erläutert), dann wird zu jedem Paar von adjazenten UIPs (u_{i-1}, u_i) mit $i = 2, \cdots, k$ (siehe Definition in (5.8) des Unterabschnitts 5.2.4) des aktuellen decision-levels eine Backjump-Klausel hinzugefügt. Auf den Graphen in Abbildung 5.4 bezogen, würde GRASP also einmal die Backjump-Klausel $(x_2 \vee x_6 \vee \neg x_{11} \vee x_{13})$ lernen, wenn x_{11} ein echtes decision-literal (z.B. ohne Vorgängerknoten) zum Zeitpunkt des Konfliktes wäre. Darüber hinaus lernt GRASP auch eine Backjump-Klausel, die der Aufteilung des Implikationsgraphen entspricht, in der sich alle Variablenbelegungen nach dem first UIP mit dem aktuellen decision-level auf der conflict-side und alle restlichen Knoten auf der reason-side befinden. Bezogen auf die Abbildung 5.4, wäre das die Backjump-Klausel

$(x_{10} \vee \neg x_8 \vee x_{17} \vee \neg x_{19})$ entsprechend dem first UIP cut. Nach einem backtrack wird die gelernte Backjump-Klausel dann zu einer Unit-Klausel, und da gerade das aktuelle decision-literal die Unit-Klausel bildet, wird sie dann gezwungenermaßen in seiner Belegung umgekehrt.

Wir machen an dieser Stelle eine kleine Nebenbemerkung, bevor wir mit dem zweiten zu unterscheidenden Fall im GRASP Lernschema fortfahren: eine solche Backjump-Klausel, die dazu führt, dass eines seiner Literale nach einem backtrack von selbst umgekehrt wird, ist in der Literatur auch unter *asserting-clause* bekannt. Es sei erwähnt, dass diese nützliche Eigenschaft durchaus für Backjump-Klauseln erwünscht ist, d.h. es wird angestrebt, eine Backjump-Klausel zu finden, die eine asserting-clause ist. Das wird insbesondere dadurch erreicht, dass man einen cut macht, der ein UIP des aktuellen decision-levels auf die reason-side und alle anderen Knoten mit demselben level, wie die des UIPs auf die conflict-side setzt.

Kehren wir nun wieder zum GRASP Lernschema zurück. Nach einem backtrack wird die Backjump-Klausel entsprechend dem first UIP cut nun zu einer asserting-clause, was dazu führt, dass x_{10} umgekehrt wird. Beachte, dass x_{10} vorher kein decision-literal war. In GRASP ist ein solches Literal ein *gefälschtes* decision-literal. Des Weiteren bleibt der decision-level nach einem backtrack unverändert. Dies bedeutet, dass wir mit der Analyse des aktuellen decision-levels noch nicht fertig sind. Falls nun ein weiterer Konflikt auftritt, obwohl das (gefälschte) decision-literal umgekehrt wurde, wird neben den Klauseln, die im ersten Fall gelernt wurden, im zweiten Fall eine weitere Klausel gelernt. Diese Klausel geht nun aus einem cut hervor, welche alle Knoten aus dem aktuellen decision-level (inklusive des gefälschten decision-literals) auf der conflict-side von allen restlichen Knoten trennt. Wenn wir in unserem Beispiel in Abbildung 5.4 annehmen würden, dass x_{11} ein gefälschtes decision-literal wäre, das aufgrund einer angenommenen asserting-clause $(x_{21} \vee x_{20} \vee \neg x_{11})$ umgekehrt wurde, dann würde GRASP neben den vorher gelernten Klauseln zusätzlich noch die Klausel $(x_{21} \vee x_{20} \vee x_6 \vee x_{13} \vee \neg x_4 \vee \neg x_8 \vee x_{17} \vee \neg x_{19})$ lernen. Diese Klausel beinhaltet nur Literale aus früheren decision-leveln als die des aktuellen levels. Beachte, dass dann auf der Basis einer solchen Klausel immer ein nicht-chronologisches-backtracking stattfindet.

Kommen wir nun zum Lernschema in REL_SAT. Es berechnet seine Backjump-Klausel, indem es rekursiv die Konflikt-Klausel mit deren vorangehenden Belegungen auflöst, bis die aufgelöste Klausel nur noch decision-literale des aktuellen levels und Literale mit niedrigeren leveln enthält. In der Darstellung des Implikationsgraphen, wäre dies mit einem cut gleichzusetzen, der alle Literale mit dem aktuellem decision-level außer die des aktuellen decision-literals auf die conflict-side setzt, und alle anderen Literale mit niedrigeren leveln auf die reason-side. Bezogen auf Abbildung 5.4 wäre das also die Backjump-Klausel $(\neg x_{11} \vee x_6 \vee x_{13} \vee \neg x_4 \vee \neg x_8 \vee x_{17} \vee \neg x_{19})$. Wir bezeichnen dieses Schema gemäß [ZM01] als REL_SAT *Lernschema*. Da das letz-

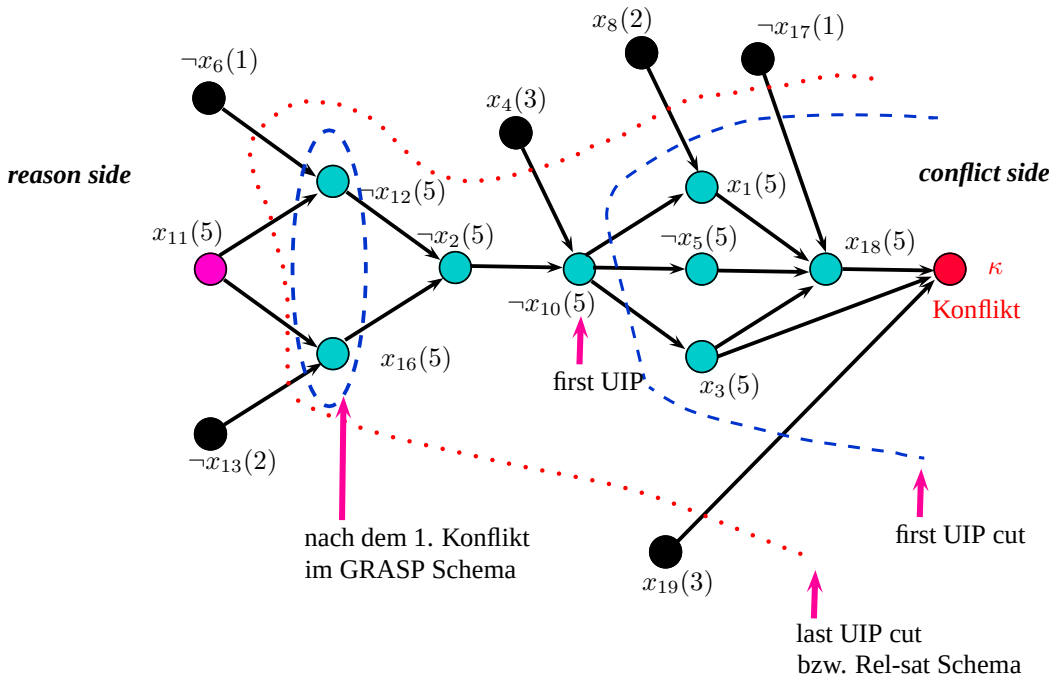

Abbildung 5.4: Ausschnitt eines Implikationsgraphen mit verschiedenen Lernschemata

te decision-literal immer das letzte UIP im Abstand zum Konflikt bezüglich eines bestimmten levels ist, wäre dieses Schema auch mit dem sogenannten *last UIP cut* bzw. *last UIP Lernschema* gleichzusetzen.

Neben den zwei genannten Lernschemata sind noch einige andere Optionen möglich. Eine davon wäre z.B. ein cut, in der alle decision-literale auf der reason-side liegen (*Decision Lernschema*). In diesem Fall wären alle decion-literale in der Backjump-Klausel enthalten, was eigentlich nutzlos wäre, da eine solche Kombination von Belegungen der decision Variablen nicht wieder auftreten kann, außer bei einem *restart*[4] (siehe Abschnitt 5.5). Daher wird eine solche gelernte Klausel kaum dazu beitragen den Suchraum zu reduzieren.

Wir haben bereits erwähnt, dass es wünschenswert ist, eine Backjump-Klausel zu finden, die eine asserting-clause ist. Diese schöne Eigenschaft einer asserting-clause, dass sich nach einem backtrack eines ihrer Literale aus dem aktuellen level automatisch umkehrt, trifft im Allgemeinen immer bei UIPs zu. REL_SAT benutzt dabei augenscheinlich immer den last UIP, indem es das aktuelle decision-literal als ein UIP auswählt, wobei GRASP und zCHAFF [MMZ+01] den first UIP nehmen, und

[4] dabei müsste man jedoch gelernte Klauseln beibehalten

das aus dem bestimmten Grund, dass man mit dem first UIP ferner einen cut im Implikationsgraphen erhält, der dem Konfliktknoten κ am nächsten steht. Zusätzlich sei der Vollständigkeit halber auch erwähnt, dass das Prinzip der UIPs nicht unbedingt auf dem aktuellen decision-level beschränkt sein muss. Man kann es daher mit dem *All UIP Schema* auf alle decision-level folgendermaßen verallgemeinern: sei ein *1 UIP Schema* ein cut, der nur den first UIP des aktuellen levels involviert, und ein *2 UIP Schema* ein cut, welcher den first UIP des aktuellen levels und den first UIP des vorherigen levels involviert, ähnlich können wir ein *3 UIP Schema* erhalten, und so weiter. Das All UIP Schema hat dann mehrere cuts und involviert dann die first UIPs aller decision-level, wobei der letzte entstandene cut bis zu decision-level 1 zurückreicht.

Alle genannten Lernschemata können durch das Travesieren des Implikationsgraphen in einer Zeitkomplexität von $O(V + E)$ implementiert werden, wobei V und E respektive die Anzahl der Knoten und Kanten im Implikationsgraphen sind.

Es stellt sich nun die Frage, was ein „gutes“ Lernschema ausmacht. Es soll in erster Linie die Anzahl der Entscheidungen, die für das Lösen eines Problems benötigt werden, so weit wie möglich reduzieren. Dabei ist es a priori sehr schwierig abzuschätzen, wie effektiv ein bestimmtes Lernschema sein wird, da das Lösen eines SAT-Problems ein dynamischer Prozess ist. Es herrscht ein komplexes Zusammenspiel zwischen dem Lernen auf der einen, und der Suche auf der anderen Seite. Wie effektiv letzendlich ein Lernschema ist, kann also gegenwärtig nur durch empirische Daten bestimmt werden. Man könnte auch von der Annahme ausgehen, je kürzer eine gelernte Klausel, desto effektiver ist das Lernschema, vor dem Hintergrund, dass kürzere Klauseln mehr Informationen enthalten als längere. In den experimentellen Ergebnissen aus [ZM01] wird diese Annahme jedoch nicht in allen zugrundeliegenden Testfällen bestätigt, wobei auf der Grundlage dieser Experimente im Ergebnis gesagt werden kann, dass sich das first UIP Lernschema vergleichweise als ein äußerst robustes und effizientes Lernschema erwiesen hat. Es wird daher in den meisten modernen DPLL Algorithmen implementiert, was uns den Ausschlag dafür gibt, es ebenfalls im Rahmen der Haskell Implementierung zu testen.

5.3 Entscheidungsheuristiken

Die Aufgabe einer Entscheidungsheuristik in einem Modernen DPLL Algorithmus ist es im Rahmen der `Decide` Prozedur zu entscheiden, welche Variable als nächstes mit welchem Wahrheitswert belegt wird. [Ms99] untersucht die Auswirkung von verschiedenen Entscheidungsheuristiken auf die Laufzeit von Modernen SAT-Solvern. Wir wollen in diesem Absschnitt ähnlich wie in [Ms99] eine kurze Zusammenfassung

über die entwickelten Entscheidungsstrategien geben, denn die Entscheidungsheuristik kann unter Umständen die Größe des Suchraums (und damit die Laufzeit des DPLL Algorithmus) stark beeinflussen. Da jedoch die Nützlichkeit einer bestimmten Entscheidungsheuristik auch problemabhängig ist, gibt es allgemein keine „beste“ Strategie. Die Grundidee, die im Zusammenhang mit einer Entscheidungsheuristik steht, ist

- eine maximale Vereinfachung der Formel F zu erreichen, d.h. die Anzahl der subsumierten (zu löschenden) Klauseln zu maximieren
- handhabbare Teilklassen[5] von SAT zu erreichen
- im Rahmen der Konflikt-Analyse, Literale aus kürzlich gelernten Klauseln zu präferieren

Dabei ist die einfachste Heuristik die nächste unbelegte Variable aus F per Zufall auszuwählen, und sie zufällig mit einem Wahrheitswert zu belegen. Diese Entscheidungsheuristik wird in der Literatur mit RAND abgekürzt. Im Folgenden werden einige Heuristiken in der zeitlichen Reihenfolge ihres Auftretens in SAT-Solvern beschrieben.

5.3.1 BOHM's Heuristik

Bohm's Heuristik wird in [IBB92] beschrieben und wählt eine Variable x, für das sein Vektor $(H_1(x), H_2(x), \ldots, H_n(x))$ in lexikographischer Reihenfolge maximal ist, dabei wird jedes $H_i(x)$ wie folgt berechnet:

$$H_i = \alpha\ max(h_i(x), h_i(\neg x))\ +\ \beta\ min(h_i(x), h_i(\neg x)),$$

wobei $h_i(x)$ die Anzahl der verbleibenden Klauseln ist, die das Literal x i-mal enthalten. Es wird also diejenige Variable vorzugsweise ausgewählt, die möglichst kleine Klauseln erfüllt (wenn die Variable auf *true* gesetzt wird), oder, die die Größe von kleinen Klauseln weiter reduziert (wenn die Variable auf *false* gesetzt wird). Die Werte für α und β werden heuristisch bestimmt, dabei wird in [IBB92] für $\alpha = 1$ und für $\beta = 2$ vorgeschlagen.

5.3.2 MOM's Heuristik

MOM ist die Abkürzung für „*Maximum Occurences on clauses of Minimum size*“ (maximales Auftreten in Klauseln minimaler Länge). Sei $f*(x)$ die Häufigkeit

[5] z.B. 2-SAT, Horn-SAT, nur positive Klauseln

des Auftretens eines Literals in den kleinsten unerfüllten Klauseln, dann wählt die MOM's Heuristik eine Variable x, die

$$[f*(x) \ + \ f*(\neg x)] * 2^k \ + \ f*(x) \ * \ f*(\neg x)$$

maximiert. Intuitiv wird also diejenige Variable mit einem hohen Auftreten in Klauseln mit x oder $\neg x$ (angenommen für ausreichend großes k) und zugleich mit einem hohen Auftreten in Klauseln mit x *sowie* $\neg x$ bevorzugt. Für eine detaillierte Beschreibung der MOM's Heuristik sei auf [Fre95] verwiesen.

5.3.3 Jeroslow-Wang Heuristik

Sei für ein gegebenes Literal l

$$J(l) = \sum_{l \in \omega \wedge \omega \in F} 2^{-|\omega|}$$

Von Jeroslow und Wang werden in [JW90] bezüglich $J(l)$ zwei Entscheidungsheuristiken vorgeschlagen:

- JW-OS (one-sided): wählt das Literal aus, für das $J(l)$ maximal ist.
- JW-TS (two-sided): wählt die Variable x aus, für die die Summe $J(x) + J(\neg x)$ maximal ist, und falls $J(x) \geq J(\neg x)$, dann wird x mit *true* belegt, andernfalls wird x mit *false* belegt.

5.3.4 Literal Count Heuristik

In diesem Unterabschnitt fassen wir drei einfache Entscheidungsheuristiken zusammen, die sich dadurch kennzeichnen, dass sie das Auftreten von Literalen einer gegebenen Variable in unerfüllten Klauseln zählen.
Dabei bezeichnen Literal Count Heuristiken die Anzahl der Vorkommnisse einer gegebenen Variable x in unerfüllten Klausel als positives Literal mit C_P und als negatives Literal als C_N. Diese beiden Werte können entweder gemeinsam oder einzeln betrachtet werden. Wenn man sie gemeinsam betrachtet, z.B. durch $C_P + C_N$, wählt man diejenige Variable mit der höchsten Summe $C_P + C_N$ aus, und belegt sie mit *true*, falls $C_P \geq C_N$, oder falls $C_P < C_N$ mit *false*. Da C_P und C_N während der Suche immer wieder neu berechnet werden, nennt man diese Heuristik auch *dynamic largest combined sum*, oder kurz DLCS.

Wenn die Werte C_P und C_N einzeln betrachtet werden, dann wählt man diejenige Variable mit dem höchsten Wert in C_P oder C_N aus, und belegt diese dann mit *true*, falls $C_P \geq C_N$ ist, oder mit *false*, falls $C_P < C_N$ ist. Diese Heuristik nennt man

dann *dynamic largest individual sum*, oder kurz DLIS.

Weitere Möglichkeiten wie das RDLIS bzw. RDLCS bieten sich ebenfalls an, wobei das „R“ jeweils für eine randomisierte Auswahl des Wahrheitswertes steht, mit der dann eine ausgewählte Variable belegt wird.

5.3.5 VSIDS Heuristik

Im Rahmen des implementierten SAT-Solvers in [MMZ+01], werden unter anderem alle bisher genannten Entscheidungsheuristiken untersucht. Aus [Ms99] geht weiterhin hervor, dass die DLIS eine „solide allzweck Strategie“ ist. Auch wir werden die DLIS Strategie implementieren und testen.

Darüberhinaus entwickelt [MMZ+01] auch eine neue Strategie VSIDS, welche für *Variable Statement Independant Decaying Sum* steht. Diese wird wie folgt beschrieben:

- jedes Literal besitzt einen Zähler, der anfangs auf 0 steht
- wenn Literale durch Lernen zur Formel F hinzugefügt werden, dann erhöhe den Zähler aller Literale, die in der gelernten Klausel vorkommen um eins
- wähle dann dasjenige noch undefinierte Literal aus F, dessen Zähler am höchsten ist
- teile periodisch alle Zähler durch eine konstante

Insgesamt kann man sagen, dass diese Strategie die Absicht hat Konflikt-Klauseln, speziell *kürzlich* aufgetretene Konflikt-Klauseln zu erfüllen, indem beim Auswählen eines Literals die Häufigkeit seines Auftretens in Konflikt-Klauseln berücksichtigt wird.

5.4 Klausel entfernen

Wir haben bereits besprochen, wie das Lernen von Backjump-Klauseln dazu führt, dass der Suchraum effizient reduziert wird. Doch kann das Hinzufügen von Backjump-Klauseln auch beträchtlich die Größe der Original-Formel erhöhen und unter Umständen ein Speicherproblem verursachen. Des Weiteren kann man sagen, je mehr Klauseln eine Formel hat, desto länger dauert die Unit-Propagation. In der Praxis ist es nicht ungewöhnlich, dass die Anzahl der gelernten Klauseln sogar die Anzahl der Klauseln in der Original-Formel übersteigt.
Dieser Tatsache kann man auf mindestens zwei Arten entgegen wirken. Erstens

könnten gelernte Klausel wiederum vorher gelernte Klauseln subsumieren und man müsste sie garnicht erst hinzugefügen. Zweitens könnte man sich dafür entscheiden, gelernte Klauseln im Falle von Effizienz- oder Speicherproblemen zu löschen. In einem solchen Fall wird z.B. nach der Größe oder der Aktivität einer gelernten Klausel abgewogen und diejenigen gelernten Klauseln für das Löschen bevorzugt, die entweder sehr viele Klauseln (mit etwa 100-200 Literalen) haben, oder die am wenigsten aktiv[6] sind.

5.5 Restarts

Die Laufzeit, die für das Lösen von zwei ähnlichen SAT-Problemen benötigt wird, kann oft sehr stark variieren, allein dadurch, dass man die Variablenreihenfolge von zwei exakt gleichen Problemen verändert. Diese Tatsache führt zu einer weiteren wichtigen Technik, die in vielen modernen DPLL Algorithmen angewendet wird, das Neustarten. Wir bezeichnen es im Folgenden als *restart* [GSK98].
Wenn ein SAT-Solver ein restart ausführt, dann verwirft er alle bisher gemachten Variablenbelegungen in M und startet die Suche erneut an der Wurzel des Entscheidungsbaums, während er die gelernten Klauseln jedoch beibehält und sie dann als Original-Klauseln behandelt. Ein restart soll verhindern, dass sich der Solver über eine lange Zeitspanne in bestimmten Teilen des Suchraumes festfährt, in der es keine Lösung gibt.

In modernen SAT-Solvern ist das Entfernen von gelernten Klausel aus den bereits im vorigen Abschnitt 5.4 genannten Gründen unausweichlich, d.h. also nicht alle gelernten Klauseln können beibehalten werden. Der restart zusammen mit dem Entfernen von Klauseln kann aber dazu führen, dass der Solver in seiner Vollständigkeit beeinträchtigt wird, da man dann unter Umständen nicht definitiv verhindern kann, dass die selbe Suche gegebenenfalls wiederholt wird. Um die Vollständigkeit zu gewährleisten, werden daher die Intervalle zwischen jedem restart gradweise erhöht, um dem Solver im Endeffekt die Möglichkeit zu geben eine Suche auch ohne einen restart bis zu einem Ende[7] durchzuführen.
Die ausführliche Untersuchung von verschiedenen restart Strategien in [Hua07] zeigt, dass es vorteilhaft ist sehr häufig einen restart auszuführen. Diejenige Strategie, die sich in [Hua07] als empirisch beste erwiesen hat, wird nach dem neuesten Stand der Technik auch als Standard eingesetzt.

[6] z.B. wie oft aus einer gelernten Klausel eine Unit-Klausel oder eine Konflikt-Klausel entsteht
[7] bis zu einem Blattknoten im Entscheidungsbaum

5.6 Erweiterung am klassischen DPLL System

Moderne DPLL Algorithmen implementieren die `PureLiteral` Regel von dem bereits vorgestellten klassischen DPLL System aus Effizienzgründen nicht, es wird daher nur im einem Vorverarbeitungsschritt verwendet. Die bahnbrechende Effizienz steigernde Änderung in modernen DPLL Algorithmen gegenüber dem klassischen DPLL Algorithmus ist das im Rahmen des Abschnitts 5.1 ausführlich vorgestellte nicht-chronologische-backtracking, welche wir im Folgenden kurz als *backjumping* bezeichnen wollen. Wie wir bereits wissen ist das Backjumping im Gegensatz zum *chronologischen-backtracking* ein allgemeinerer und mächtigerer `Backtrack` Mechanismus. An dieser Stelle möchten wir noch einmal das Backjumping rekapitulieren und uns noch einmal das Beispiel 4.3.2 vor dem Hintergrund des abstrakten DPLL Frameworks anschauen.

5.6.1 Backjumping

Wenn man sich das Beispiel 4.3.2 noch einmal vor Augen hält,

$$
\begin{array}{lllllll}
\emptyset & || & \overline{1} \vee 2, & \overline{3} \vee 4, & \overline{5} \vee \overline{6}, & 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \quad \texttt{Decide} \\
\vdots & & & & & & \\
1^d\ 2\ 3^d\ 4\ 5^d\ \overline{6} & || & \overline{1} \vee 2, & \overline{3} \vee 4, & \overline{5} \vee \overline{6}, & 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \quad \texttt{Backtrack} \\
1^d\ 2\ 3^d\ 4\ \overline{5} & || & \overline{1} \vee 2, & \overline{3} \vee 4, & \overline{5} \vee \overline{6}, & 6 \vee \overline{5} \vee \overline{2} &
\end{array}
$$

und den Konflikt ausgehend von der Konflikt-Klausel $6 \vee \overline{5} \vee \overline{2}$ und der Belegung $1^d\ 2\ 3^d\ 4\ 5^d\ \overline{6}$ genauer analysiert, dann stellt man fest, dass die Entscheidung 1^d und die daraus resultierende Belegung 2 zusammen mit der Entscheidung 5^d und die hieraus resultierende Belegung $\overline{6}$, die Ursache für den Konflikt gewesen ist. D.h. im Klartext: die Belegungen 1^d und 5^d sind unvereinbar. Eine weitere wichtige Feststellung ist, dass die Entscheidung 3^d nicht zur Entstehung des Konflikts beigetragen hat. Insgesamt kann man also die Klausel $\overline{1} \vee \overline{5}$, herleiten, und würde somit die Inkompatibilität zwischen 1^d und 5^d vermeiden, wenn diese Klausel nämlich in der Formel vorhanden gewesen wäre, dann hätte ihre Präsenz bei der ersten Entscheidung 1^d schon dazu geführt, dass `UnitPropagate` $\overline{5}$ abgeleitet, und dadurch die Inkompatibilität zwischen 1^d und 5^d vermieden hätte. Da man die Klausel $\overline{1} \vee \overline{5}$, wie wir in Abschnitt 5.2.3 gesehen haben, auch algorithmisch ermitteln kann, definieren wir eine neue Regel, die statt einem `Backtrack` einen `Backjump` ausführt und somit die Entscheidung 3^d überspringt und $\overline{5}$ durch `UnitPropagate` impliziert und zu M hinzufügt.

$$
\begin{array}{lllllll}
1^d\ 2\ 3^d\ 4\ 5^d\ \overline{6} & || & \overline{1} \vee 2, & \overline{3} \vee 4, & \overline{5} \vee \overline{6}, & 6 \vee \overline{5} \vee \overline{2} & \Longrightarrow \quad \texttt{Backjump} \\
1^d\ 2\ \overline{5} & || & \overline{1} \vee 2, & \overline{3} \vee 4, & \overline{5} \vee \overline{6}, & 6 \vee \overline{5} \vee \overline{2} &
\end{array}
$$

Die DPLL Prozedur besitzt ohne die `PureLiteral` Regel nun insgesamt vier Regeln, wobei die Backtrack Regel durch die `Backjump` Regel folgendermaßen ersetzt wird:

Definition 5.6.1.

`Backjump`:

$$M\ l^d\ N||\ F,\ C \quad \Longrightarrow M\ l'\ ||\ F,\ C \quad \textbf{wenn} \begin{cases} M\ l^d\ N \models \neg C \text{ und es gibt} \\ \text{eine Klausel } C' \vee l' \text{ mit} \\ F,\ C \models C' \vee l' \text{ und } M \models \neg C', \\ l' \text{ ist undefiniert in } M, \\ l' \text{ oder } \neg l' \text{ kommen in } F \\ \text{oder in } M\ l^d\ N \text{ vor} \end{cases}$$

Hierbei ist C die Konflikt-Klausel und $C' \vee l'$ eine (algorithmisch) ermittelte Backjump-Klausel. Im obigen Beispiel ist $C = 6 \vee \overline{5} \vee \overline{2}$, $C' = \overline{1}$ und $l' = \overline{5}$.

Es kann noch einmal festgehalten werden, dass beim chronologischen-backtracking (im klassischen DPLL System durch `Backtrack` modelliert) immer nur die letzte Entscheidung l revidiert wird, indem man zum vorherigen level zurückgeht und die Belegung $\neg l$ zu M hinzufügt. Beim nicht-chronologischen-backtracking jedoch (jetzt durch `Backjump` modelliert), ist man auch in der Lage mehr als nur ein chronologisches-backtracking auszuführen, indem man die Gründe analysiert, die eine Konflikt-Klausel verursacht haben. `Backjump` kann häufig *mehrere* getroffene Entscheidungen gleichzeitig revidieren, indem es zu einer früheren Entscheidung als nur die letzte springt und einige neue Literale auf diesem level zu der Belegung M hinzufügt. Es springt über diejenigen level, die für die Entstehung des Konflikts quasi irrelevant sind hinweg. Im vorherigen Beispiel haben wir bei genauerer Betrachtung feststellen können, dass die Entscheidung 3^d und deren Konsequenz 4 gänzlich nicht zum Konflikt in der Klausel $6 \vee \overline{5} \vee \overline{2}$ beigetragen haben. Daher war der Zustand $1^d\ 2\ \overline{5}$ nach der `Backjump` Regel viel „intelligenter“ als der Zustand $1^d\ 2\ 3^d\ 4\ \overline{5}$ nach einem `Backtrack`.

Es gibt im Falle eines Konfliktes immer eine Backjump-Klausel, falls die `Fail` Regel nicht anwendbar ist. Weiterhin gibt es die Möglichkeit einfach die Negation der bisher gemachten Entscheidungen als Backjump-Klausel zu verwenden. Bei Verwendung einer solchen Backjump-Klausel simuliert man in diesem Fall nichts anderes als die `Backtrack` Regel aus dem klassischen DPLL System.

5.6.2 Learn

Die meisten modernen DPLL Implementierungen fügen die ermittelte Backjump-Klausel, wie bereits erwähnt, als *gelernte Klausel* zusätzlich zur Formel F hinzu. Die gelernte Klausel $\overline{1} \vee \overline{5}$ im Beispiel 4.3.2 würde jeden Konflikt verhindern, dessen Ursache in einer Belegung von 1 und 5 in M läge. Sie hat also das Ziel ähnliche in der Zukunft auftretende Konflikte zu verhindern. Daher wird das Lernen einer Backjump-Klausel durch die Erweiterung am klassischen DPLL System mit einer `Learn` Regel wie folgt erweitert:

Definition 5.6.2.

`Learn`:

$$M \mid\mid F \quad \Longrightarrow M \mid\mid F,\ C \qquad \textbf{wenn} \begin{cases} \text{Jedes Atom aus } C \text{ kommt in } F \text{ oder } M \text{ vor,} \\ F \models C \end{cases}$$

In der `Learn` Regel bezeichnet man die Klausel C als gelernte Klausel, wenn sie nicht bereits in F vorkommt. Sie erlaubt es jeder beliebige Klausel C, die durch F abgeleitet werden kann, zur aktuellen Formel hinzuzufügen, solange alle Atome von C entweder in F oder M vorkommen.

5.6.3 Forget

In der Praxis wird eine gelernte Klausel auch wieder aus der Formel entfernt, wenn ihre Relevanz (siehe z.B. [JS97]) oder ihre Nützlichkeit unter eine bestimmte Schranke fällt. Die Nützlichkeit kann beispielsweise dadurch beeinflusst werden, wie oft aus der gelernten Klausel eine Unit-Klausel oder eine Konflikt-Klausel [GN02] entsteht. Das klassische DPLL System wird daher um folgende `Forget` Regel erweitert:

Definition 5.6.3.

`Forget`:

$$M \mid\mid F,\ C \quad \Longrightarrow M \mid\mid F \qquad \textbf{wenn} \begin{cases} F \models C \end{cases}$$

Eine Klausel gilt demnach als *entfernt*, wenn sie im Zuge der `Forget` Regel aus F gelöscht wird. Im Grunde genommen kann die `Forget` Regel dazu verwendet werden, jede beliebige Klausel aus F zu entfernen, die durch die übrigen Klauseln in F abgedeckt werden. Die `Forget` Regel soll also nicht nur solche Klauseln entfernen, die im Rahmen der `Learn` Regel kürzlich zu F hinzugefügt wurden. In der Praxis ist die Anwendung der `Forget` Regel jedoch beschränkt, weil es durchaus rechenintensiv werden kann, zu bestimmen, ob und welche Klauseln zu welchem Zeitpunkt entfernt werden können.

5.6.4 Restart

Zusätzlich starten moderne DPLL Implementierungen die DPLL Prozedur neu, wenn die Suche nicht genügend Fortschritte macht bezüglich eines bestimmten Maßstabes. Unter dem Vorbehalt nämlich, dass nicht alle gelernten Klauseln gelöscht werden, trägt das zusätzliche Wissen über den Suchraum dazu bei, dass nach einem restart die Suche mehr und mehr eingeschränkt wird. Die Kombination aus `Learn` und einer `Restart` Regel haben sich sowohl in der Praxis als auch in der Theorie als äußerst mächtig erwiesen. Im Rahmen des Abstrakten DPLL Frameworks kann die `Restart` Regel wie folgt im modernen DPLL Algorithmus modelliert werden:

Definition 5.6.4.

`Restart`:
$M \mid\mid F \qquad \Longrightarrow \emptyset \mid\mid F$

Wichtig ist es bei der Implementierung des `Restart`s zusammen mit der `Forget` Regel, dass im abstrakten DPLL System ein Endzustand erreicht wird, was in der Praxis dadurch sichergestellt ist, dass die Größe eines `Restart` Intervalls periodisch erhöht wird.

6 Implementierung

Ziel dieses Kapitels ist es eine endgültige Variante der Implementierung eines modernen DPLL Algorithmus in Haskell zu beschreiben. Wir werden dabei nur an den Stellen, wo es als sinnvoll erscheint auch im Detail auf einzelne Teile des Programmcodes eingehen. Dabei verwendet die Implementierung des modernen DPLL Algorithmus die ebenfalls implementierte Variante des klassischen DPLL Algorithmus als Grundgerüst.

Insgesamt basiert die Implementierung auf der Theorie des abstrakten DPLL Frameworks aus [NOT06]. Anders als das abstrakte Framework[1] aus [NOT06] jedoch, in der $M \mid\mid F \Longrightarrow M' \mid\mid F$ gilt, verändern wir aus Effizienzgründen die Formel F mit, um eine Unit-Klausel oder ein Konflikt leichter zu finden (siehe u.a. Unterabschnitt 6.2.5). Zusätzlich lässt sich eine Abbruchbedingung[2] dadurch einfacher überprüfen (siehe Unterabschnitt 6.3.1).

6.1 Implementierte Varianten

Mit dem Modul `Dpll_ChronoBacktrack_State` haben wir im Rahmen dieser Diplomarbeit zunächst einmal den klassischen DPLL Algorithmus mit chronologischem-backtracking Verfahren[3] implementiert, welcher aus den folgenden 6 Tasks besteht:

```
*****************************************
  – Task 1: UnitPropagate                –
  – Task 2: Decide                       –
  – Task 3: Zustände der DPLL Prozedur   –
  – Task 4: DPLL-Prozedur (Top-Level)    –
  – Task 5: Chronologisches Backtracking –
  – Task 6: Tests                        –
*****************************************
```

[1] verändert nur das Modell M von einem Zustand zum anderem

[2] SAT und ein Modell, oder UNSAT und gegebenenfalls lazy Ausgabe der jeweiligen (Teil)-Belegung

[3] Der Quellcode befindet sich auf der beigefügten CD im Verzeichnis klassischer_DPLL/`Dpll_ChronoBacktrack_State.hs`

Es folgt dann eine Erweiterung an dem klassischen Algorithmus durch die folgenden Tasks, welche in dem Modul `Dpll_backjump_mit_Learn` resultiert:

```
****************************************
  - Task 5: Konflikt-Analyse             -
  - Task 6: Zustand nach gelernter Klausel -
  - Task 7: Learn                        -
****************************************
```

Für die Konflikt-Analyse aus *Task 5* importiert das Modul `Dpll_backjump_mit_Learn` zusätzlich noch ein weiteres Modul `AnalyseConflict`, welches die folgenden Tasks beinhaltet:

```
************************************************************
  - Task 1: Knoten und Kanten im Implikationsgraph          -
  - Task 2: Konstruiere Implikationsgraph                   -
  - Task 3: Analysiere Konflikt                             -
      - Task 3a: Berechne Λ(x)                              -
      - Task 3b: Berechne Σ(x)                              -
      - Task 3c: Berechne die „conflicting assignment“ causes_of(κ) -
      - Task 3d: Berechne die „Backjump-Klausel“ ωc(κ)      -
      - Task 3e: Berechne den „backjump-level“ β            -
************************************************************
```

Die Module `Dpll_backjump_mit_Learn` und `AnalyseConflict` bilden zusammen die erste Variante des Modernen DPLL Algorithmus mit nicht-chronologischem-backtracking Verfahren und dem conflict-driven-learning[4].

Anschließend wurde die nächste Variante des modernen DPLL Algorithmus durch das Modul `AnalyseConflict_firstUIP` um eine Berechnung des first UIP cuts[5] erweitert:

```
************************************************************
  - Task 4: Berechne Backjump-Klausel durch First UIP Lernschema -
      - Task 4a: Ermittle first UIP                         -
      - Task 4b: Berechne causes_of(x, u_i)                 -
************************************************************
```

[4] siehe CD Verzeichnis moderner_DPLL/DPLL_mit_Backj_u_Learn/`Dpll_backjump_mit_Learn.hs` und moderner_DPLL/DPLL_mit_Backj_u_Learn/`AnalyseConflict.hs`

[5] siehe CD Verzeichnis moderner_DPLL/firstUIP/`AnalyseConflict_firstUIP.hs`

Darüberhinaus haben wir die vorangegangene first UIP Variante mit den folgenden Tasks um einige Entscheidungsheuristiken ausgeweitet:

```
*********************************************************
  - Task 2b: Decision Heuristiken                        -
          - (statisch: erste Literal aus der ersten Klausel) -
          - DLIS: dynamic largest individual sum         -
          - Literal, das am wenigsten vorkommt           -
          - Variante der MOM'S Heuristik                 -
*********************************************************
```

An dieser Stelle sei erwähnt, dass wir im Rahmen der Experimente in Kapitel 7, die verschiedenen Heuristiken auch in Verbindung mit zuvor implementierten Varianten untersuchen werden.

Alle Varianten lassen sich zudem durch ein zusätzlich implementiertes Modul `Test`[6] überprüfen, mit dessen Hilfe man auch größere Problem Instanzen aus einer DIMACS Eingabedatei [7] einlesen und für jede Problem Instanz die verbrauchte CPU Laufzeit bestimmen kann.

Es wird nun kurz auf die zugrundeliegende Datenstruktur eingegangen, bevor dann der Aufbau der Implementierung in Haskell anhand eines Flussdiagramms in Abbildung 6.2 beschrieben wird, auf die dann unsere nachfolgenden Erläuterungen basieren werden.

6.2 Datenstruktur

6.2.1 Literal

Das Literal beschreibt aussagenlogische Variablen, die in einer booleschen Formel entweder negiert oder nicht negiert vorkommen können. In unserer Implementierung werden diese als Integer dargestellt. Integer mit einem negativen Vorzeichen repräsentieren negierte Variablen, und Integer ohne Vorzeichen, positive Variablen.

```
1 type Literal          = Int
```

[6] siehe Quellcode CD Verzeichnis klassischer_DPLL/`Test_Zeitmessung.hs`
[7] siehe dazu Abschnitt 7.1

6.2.2 Klausel

Eine Klausel ist eine Disjunktion von Literalen. Sie wird in unserer Implementierung als eine Liste von Literalen dargestellt.

```
type Clause           =  [Literal]
```

6.2.3 Formel F

Die Formel beschreibt die Konjunktion von Klauseln. Wie wir bereits wissen, kann diese in einer abgekürzten Darstellung als Menge definiert werden. In unserer Implementierung wird die Formel als eine Liste von Klauseln dargestellt.

```
type ClauseSet    = [Clause]
```

Zusätzlich definieren wir einen weiteren Typ für die Formel, mit der wir jede einzelne Klausel innerhalb der Formel indizieren, sodass bei Bedarf mit Hilfe eines Index auf jede Klausel zugegriffen werden kann. Wir stellen diese mit Hilfe einer Liste von Tupeln der Form (Index, Klausel) dar:

```
type Formel = [(Int, Clause)]
```

6.2.4 Belegung M

Eine (Teil)Belegung stellt den Wahrheitswert einer Variablenbelegung dar. In unserer Implementierung besteht diese aus einer Liste von Quadrupeln, welche linksseitig erweitert wird, im Gegensatz zur Darstellung im abstrakten Framework [NOT06], welche die Belegung M rechtsseitig erweitert.

```
type Model    = [(Literal, LFlag, Int, Int )]
data LFlag    = I | D | A deriving (Eq, Show)
```

Das Quadrupel hat als erstes Element ein Literal als Typ. Das Literal stellt gleichzeitig mit seinem Vorzeichen den Wahrheitswert der Variable dar. Das zweite Element des Quadrupels, gekennzeichnet mit `LFlag`, enthält die Information, wodurch das Literal belegt wurde. Dabei steht `I` für implied-literal (durch `UnitPropagate` impliziert), `D` für decision-literal (durch `Decide` ausgewählt und belegt) und `A` für asserted-literal (nach einem `Backjump` mit `flipLit`[8] gesetzt). Mit dem dritten Element des Quadrupels führen wir den Index der Klausel mit, aus der die Belegung des Literals stammt (der Index wird später beim Aufbau des Implikationsgraphen benötigt). Das letzte Element schließlich, speichert immer den aktuellen decision-level des Literals. Der decision-level wird dabei bei jedem `Decide` inkremeniert und

[8] Die Funktion `flipLit` revidiert eine Belegung nach dem backjumping im backjump-level β

bei jedem `UnitPropagate` sowie nach einem `Backjump` und dem damit verbundenem Umkehren der Belegung durch `flipLit` beibehalten.

Weitere verwendete Datenstrukturen werden im Quellcode immer dort definiert, wo sie zum ersten Mal benutzt werden.

6.2.5 Unterschied der Datenstruktur zum imperativen Ansatz

Wir wollen einmal grob den „traditionellen Ansatz“ der Datenstruktur in imperativen Implementierungen beschreiben. Hier sind zu den Literalen (als Liste) in der Datenstruktur der Klausel zusätzlich zwei Zähler mit der Anzahl der positiven und negativen Literale (um Unit-Klauseln zu erkennen) vorhanden. Darüberhinaus existiert in der Datenstruktur der Klausel ein Zeiger auf diejenige Variable, durch die die entsprechende Klausel zuerst subsumiert[9] wurde. Der Zeiger zeigt auf NIL, falls die Klausel noch nicht subsumiert ist. Des Weiteren enthält die Datenstruktur einer Variablen drei Zustände, die den Wahrheitswert der Variable kennzeichnen: 0 für *false*, 1 für *true* und *offen*, falls die Variable noch nicht belegt ist. Darüberhinaus speichert die Variable mit zwei Zählern jeweils die Anzahl der positiven und negativen Variablen sowie ein Zeiger zu denjenigen Klauseln, in denen die Variable als positives bzw. negatives Literal vorkommt.

Betrachten wir nun unter einer solchen Datenstruktur eine Formel mit m Klauseln und n Variablen, wobei wir annehmen, dass jede Klausel l Literale enthält, dann müssen im Durchschnitt $l * m/n$ Zähler aktualisiert werden, wenn eine Variablenbelegung stattfindet. Genauso müssen im Zuge eines Backjumps nach einem Konflikt $l * m/n$ Zähler wieder zurückgesetzt werden.

Ein wesentlicher Unterschied in der Datenstruktur unserer Haskell Implementierung gegenüber imperativen DPLL Algorithmen [MSS99], die Listen benutzen (siehe u.a. [CA93]) liegt darin, die Belegung M und die Formel F als ein Zustandspaar $M \mid\mid F$ zu betrachten. Dadurch wird jede Änderung (durch `Decide`, `UnitPropagate` oder durch `flipLit` nach einem `Backjump` und `Learn`) an der Belegung M zusammen mit ihrer Auswirkung auf die Formel F mitgeführt. Dabei überführt jeweils jede Anwendung der oben genannten Prozeduren das Zustandspaar $M \mid\mid F$ in einen anderen Zustand $M' \mid\mid F'$.

Das hat den Vorteil, wenn wir nach einem Konflikt zu einem bestimmten level springen, so haben wir mit dem Zustandspaar immer gleichzeitig Zugriff auf die Belegung

[9] ein Klausel ist subsumiert, wenn mindestens ein Literal in ihr *true* und somit auch die Klausel als Ganzes *true* ist

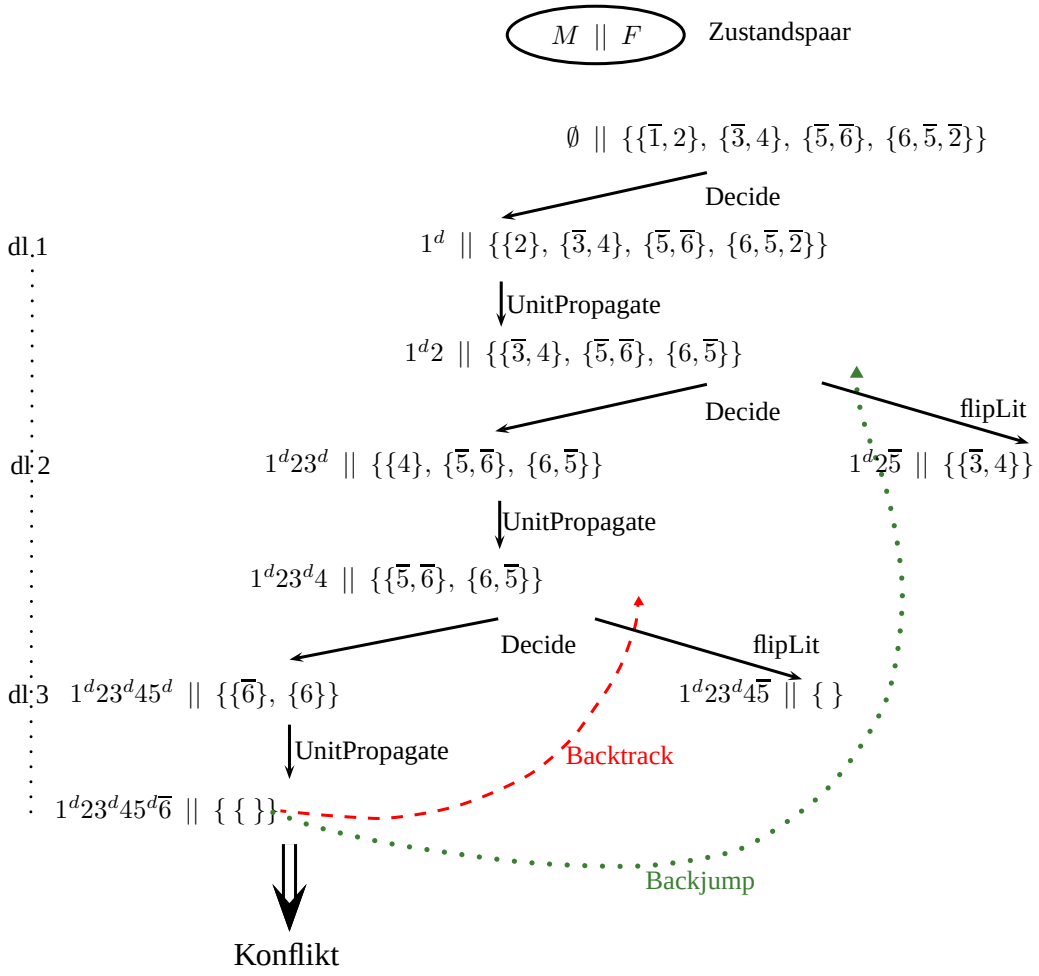

Abbildung 6.1: Beispiel zur Idee der Datenstruktur als Zustandpaar

M und die dazu korrespondierende Formel F des jeweiligen levels. Wir wollen diese Idee anhand eines kleinen Beispiels in Abbildung 6.1 veranschaulichen.

Man sieht in Abbildung 6.1, wie beim `Backjump` die Entscheidung 3^d inklusive der daraus resultierenden Implikation 4, im Gegensatz zum `Backtrack` übersprungen wird[10]. Bei einem `flipLit` wird im Gegensatz zu `Decide` die andere noch nicht verwendete Alternative der Belegung des entsprechenden Literals verwendet. Dabei ist es wichtig, dass das Umkehren des Literals durch `flipLit` im richtigen zuvor berechnetem backjump-level[11] stattfindet. Beachte, dass bei einem `flipLit` nach einem `Backtrack` der backjump-level immer um eins kleiner ist als die des aktuellen decision-levels. Die Abbildung 6.1 zeigt zudem, wie sich entsprechend beim Auführen

[10] der decision-level 2 ist nicht an der Entstehung des Konfliktes beteiligt
[11] decision-level, in dem die Suche nach einem Konflikt fortgeführt werden soll

von `Decide`, `UnitPropagate` oder `flipLit`, das Modell M und die Formel F jeweils von Zustand zu Zustand verändert.

6.3 DPLL Prozedur

Die DPLL Prozedur ist der Einstiegspunkt des Algorithmus, sie erhält als Eingabe ein Zustandspaar wie in Abbildung 6.1 dargestellt. Wir definieren das Zustandspaar dabei durch das `data`-Konstrukt in Haskell wie folgt:

```
1 data State = State Model Formel
2  deriving Show
```

Die DPLL Prozedur unterscheidet vor seiner Verarbeitung eines Zustandspaars immer, ob ein Endzustand erreicht wurde, oder ob ein Backtrackzustand vorliegt (zu den Einzelheiten zum Endzustand bzw. Backtrackzustand siehe Unterabschnitt 6.3.1). Dies passiert mit Hilfe des rekursiven Aufrufs der DPLL Prozedur nach jedem `Decide`, `UnitPropagate` und `flipLit`. Um diese Unterscheidung zu gewährleisten kapseln wir die Ausgabe, indem wir ein entsprechendes `data`-Konstrukt `data BacktrackResult a` definieren. Der Datentyp `BacktrackResult` besitzt als erste Alternative `Result a` und als zweite Alternative `Backtrack Int ClauseSet`. `Int` steht für die Rückgabe eines berechneten backjump-levels `blevel`, und in `ClauseSet` wird durch das Lernen, die Formel um eine ermittelte Backjump-Klausel erweitert.

```
1 data BacktrackResult a = Result a | Backtrack Int ClauseSet
2  deriving Show
```

Wir verwenden diesen Datentyp anschließend in der Funktionsdefinition der DPLL Prozedur:

```
1 dpll :: State → (BacktrackResult State)
2 dpll state
3  | isCloseState state = Result state
4  | isBacktrackState state = Backtrack (analyseConflict state)
5                             ((getclSet state) ++ [(getConflictInducedCl state)])
6  ...
7  | doDecide state = ...
8  ...
9  | otherwise = let newstateP = unitPropagate state
```

Falls weder ein Endzustand durch `isCloseState state`, noch ein Backtrackzustand durch `isBacktrackState state` registriert wird, können wir mit dem Auswählen der nächsten Variablen durch `doDecide` fortfahren und gegebenenfalls weitere Literale durch `UnitPropagate` implizieren. Dadurch kommt es zu entsprechenden Veränderungen im Zustandspaar $M \parallel F$ (ähnlich zu Abbildung 6.1).

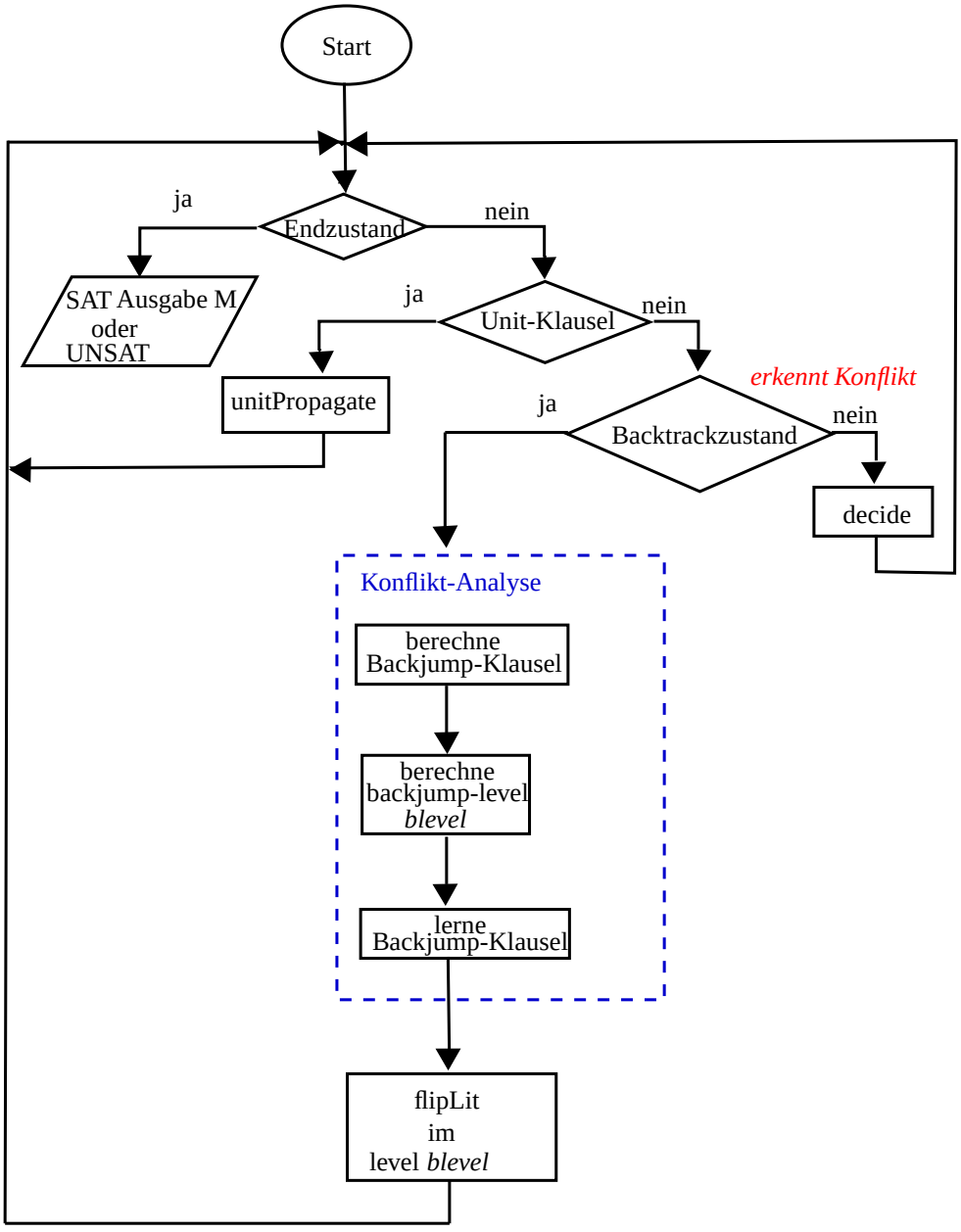

Abbildung 6.2: Aufbau der Implementierung des modernen DPLL Algorithmus in Haskell

6.3.1 Zustände der DPLL Prozedur

Ob ein Endzustand erreicht ist, wird über die Funktion `isCloseState state` geprüft.

```
1 isCloseState :: State → Bool
2 isCloseState (State m c c_0)
3       | c == []                          = True
4       | (not(checkM m) && (checkC c))    = True
5       | otherwise                        = False
```

Einerseits stellt derjenige Zustand mit der leeren Formel in F einen Endzustand dar (SAT (Zeile 3)), und andererseits derjenige Zustand, in der F die leere Klausel enthält und in M kein decision-literal existiert (UNSAT (Zeile 4)). In allen anderen Fällen liegt kein Endzustand vor (Zeile 5).

Wenn also kein Endzustand vorliegt, dann wird der Zustand auf ein Backtrackzustand hin überprüft (Zeile 4 `dpll` Funktion):

```
1 isBacktrackState (State m c c_0) = (checkM m) && (checkC c)
```

Hier wird der DPLL Prozedur signalisiert, dass ein Backtrack durchgeführt werden muss. Das ist dann der Fall, wenn F eine leere Klausel und M mindestens ein decision-literal enthält, sodass für den Algorithmus noch die Möglichkeit besteht, eine Entscheidung in M zu revidieren, um dennoch eine erfüllbare Belegung zu finden.

Sobald ein Backtrackzustand erfüllt ist, wird die Konflikt-Analyse angestoßen, welche dann mit Hilfe des Implikationsgraphen einen backjump-level `blevel` zurückgibt (Zeile 4 `analyseConflict state` in der `dpll Funktion`) und eine berechnete Backjump-Klausel zu F hinzufügt (Zeile 5 in der `dpll` Funktion).

6.4 Konflikt-Analyse

Wie wir bereits in Abschnitt 5.2 erwähnt haben, berechnen wir eine Backjump-Klausel mit Hilfe des Implikationsgraphen. Er wird in der Implementierung erst dann aufgebaut, wenn er benötigt wird, nämlich dann, wenn die Konflikt-Analyse angestoßen wird. Wir schildern dabei grob unsere Vorgehensweise beim Aufbau des Implikationsgraphen im nächsten Unterabschnitt.

6.4.1 Aufbau des Implikationsgraphen

Die Knoten in einem Implikationsgraphen repräsentieren alle Literale in M zum Zeitpunkt eines Konfliktes. Sie werden nach dem LIFO Prinzip aus M entfernt und

suksessive als Knoten im Implikationsgraphen verwendet. Jedes Literal im Datentyp `type Model` hat einen Zeiger zu derjenigen Klausel, in der es impliziert worden ist. Alle ursprünglichen Literale dieser Klausel (aus der Original-Formel), werden bis auf das implizierte Literal negiert und mit einer Kante zum implizierten Literal versehen. Betrachten wir als Beispiel einmal einen Ausschnitt aus einer Original-Formel F und einer Belegung M:

$$F = \quad \{\{\overline{9}, \overline{6}, 7, \overline{8}\}_1, \ \cdots \ , \{6\}_8, \{\overline{7}\}_9\}$$

$$M = \quad 6_8 \overline{7}_9 9^d \overline{8}_1$$

Jede Klausel in F ist mit einem Index versehen und jedes implizierte Literal in M ebenfalls. Wir würden dann zunächst einmal für jedes Literal in M einen Knoten bilden. Aufgrund der Übersichtlichkeit verzichten wir hier in den Knoten auf die Darstellung des decision-levels:

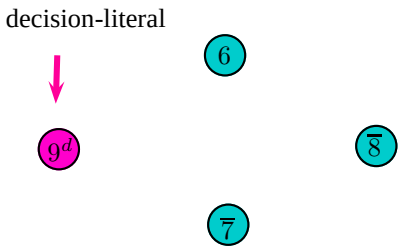

Abbildung 6.3: Knoten des Implikationsgraphen

Wir schauen uns nun gemäß dem LIFO Prinzip das erste implizierte Literal in M an, das ist die $\overline{8}_1$ (beachte, dass M linksseitig erweitert wird). Mit dem Index von $\overline{8}_1$ wissen wir, dass es durch die Original-Klausel 1 impliziert worden ist. Klausel 1 hilft uns jetzt die Kanten des Implikationsgraphen zu bilden. Dabei ziehen wir von allen Literalen der Klausel 1, die wir vorher negieren, eine Kante zu dem Knoten $\overline{8}$. Die Kanten werden zusätzlich mit dem Index der Klausel gekennzeichnet, aus der das Literal $\overline{8}$ stammt. Dabei entsteht ein Implikationsgraph, wie in Abbildung 6.4 dargestellt.

Beachte, dass decision-literale keine eingehenden Kanten besitzen, da sie nicht impliziert, sondern vielmehr durch `Decide` belegt werden. Literale, die in der Original-Formel bereits eine Unit-Klausel waren, besitzen ebenfalls keine eingehenden Kanten (wie Unit-Klauseln $\{6\}_8$ und $\{\overline{7}\}_9$).

Insgesamt wird der Implikationsgraph also nach dem Enstehen eines Konfliktes mit Hilfe von M und der Indizes, die als Zeiger zu den Original-Klauseln dienen, wie oben geschildert, von der Konfliktbelegung ausgehend rückwärts aufgebaut.

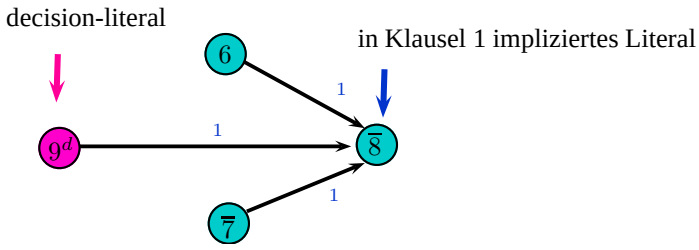

Abbildung 6.4: Aufbau des Implikationsgraphen anhand der Belegung M

6.4.2 Backjumping und Learn

In der Art und Weise der Implementierung des Modells M und der Formel F als Zustandspaar, muss einiges nach einem Backjump und Learn beachtet werden, bevor die Suche fortgeführt wird. Da sich im Laufe des Algorithmus die Formel F durch Streichen von Literalen und / oder Klauseln verändert, erweitern wir unsere Datenstruktur, indem wir zusätzlich die Original-Formel inklusive der gelernten Klauseln in `ClauseSet` mitführen. Dadurch lässt sich später der Verlauf der Implikationen durch das Aufbauen des Implikationsgraphen, wie in Unterabschnitt 6.4.1, rekonstruieren.

```
1 data State = State Model Formel ClauseSet
2  deriving Show
```

Angenommen der momentane Zustand ist

$$St_k = M_k \mid F_k \mid C_k.$$

Sei die Suche im Zustand St_i auf ein Konflikt gestoßen, sodass die Suche an der Stelle

$$St_i = M_i \mid F_i \mid C_i$$

mit einer Backjump-Klausel bCl absteigt. Angenommen die Konflikt-Analyse berechnet einen backjump-level k. Jetzt ist C_i nicht unbedingt gleich zu C_k, denn vielleicht wurde schon einmal zwischen k und i ein Backjump ausgeführt und Klauseln so auf diesem Weg zu St_i dazugelernt. (Es sollte aber gelten, dass $C_i = C_k + xs$ für irgendeine Formel xs). Jetzt sollte das Backjump Resultat sein:

$$\texttt{Backjump}\ k\ (C_i + bCl)$$

Wenn das den Zustand St_k erreicht, muss also mit folgendem Zustand fortgesetzt werden:

$$St_k = M_k \mid F_{k'} \mid (C_i + bCl),$$

wobei $F_{k'}$ die Anwendung des Modells M_k auf die Formel $(C_i + bCl)$ ist, welche wir mit $M_k(C_i + bCl)$ bezeichnen.

Etwas ineffizient ist so die Berechnung von $F_{k'}$, weil es die Information von F_k nicht benutzt. Eigentlich weiß man, dass $C_i = C_k + xs$ und $F_k = M_k(C_k)$, d.h. effizienter wäre es F'_k zu berechnen als

$$F'_k = F_k + M_k(xs + bCl).$$

6.5 First UIP Lernschema

Wir berechnen den first UIP analog zu der im Abschitt 5.2.4 definierten Gleichung:

$$causes\_of(x, u_i) = \begin{cases} u_i & \text{wenn } x = u_i \text{ oder } A(x) = \emptyset \\ \Lambda(x) \cup \left[\bigcup_{y \in \Sigma(x)} causes\_of(y, u_i) \right] & \text{sonst,} \end{cases} \tag{6.1}$$

Für jeden Knoten x aus der Menge der Konfliktbelegung (alle Knoten, die unmittelbar auf den Knoten κ zeigen), gehen wir im Implikationsgraphen um $causes\_of(x, u_i)$ zuerst für die unmittelbaren Vorgänger von κ zu berechnen, wie folgt vor:

```
causeof_x_ui graph x ui lsn@(lambda_list,sigma_list,nds) =
 let
     fnsigma x = fromJust "sigma" (lookup x sigma_list)
     fnlambda x = fromJust "lambda" (lookup x lambda_list)
     fnkappa x = fromJust "kappa" (lookup x result)
     result =  [(x,if (x == ui) || null (pre graph x)  then [x] else (fnlambda x
          `union` concat [fnkappa y | y ← fnsigma x])) | x ← nds]
 in (fnkappa x)
```

Um jetzt auf der Basis von `causeof_x_ui` den Implikationsgraphen vom Konfliktknoten κ ausgehend zum letzten decision-literal zu travesieren, definieren wir uns eine weitere Funktion `causeof_x_ui_all`, die anhand eines berechneten UIPs `ui` den entsprechenden UIP cut ermittelt. Die Funktion liefert also diejenigen Knoten bzw. Literale mit einer direkten Kante zur conflict-side.

```
causeof_x_ui_all :: Model → Formel → ClauseSet → Gr Int Int → [Literal]
                     → [Literal]
causeof_x_ui_all m c c_0 graph pre_kappa =
 let max_delta = maximum [delta y graph | y ← pre_kappa]
     lambdak    = [y | y ← pre_kappa, delta y graph < max_delta]
     sigmak    = [y | y ← pre_kappa, delta y graph == max_delta]
     nds = nodes graph
```

```
8       lambda_list = [(x, (lambda x graph)) | x ← nds]
9       sigma_list =  [(x, (sigma x graph)) | x ← nds]
10      lsn = (lambda_list,sigma_list,nds)
11      ui   = firstUIP m c c_0
12      result = [ui] `union` (lambdak `union` (concat [causeof_x_ui graph y ui lsn
            | y ← sigmak]))
13    in  trace2h  (  "cause_of_kappa prekappa:" ++ show pre_kappa ++ "\n"
14                  ++ "cause_of_kappa lambdak:" ++ show lambdak ++ "\n"
15                  ++ "cause_of_kappa sigmak:" ++ show sigmak ++ "\n"
16                  ++ "cause_of_kappa:" ++ show result
17                )
18       result
```

Handelt es sich bei der Variable `ui` um ein first UIP, wie sie durch `firstUIP m c c_0` (Zeile 11) ermittelt wird, dann wird ein cut gemäß dem first UIP Lernschema ermittelt. Diese Schnittstelle bietet weitere Möglichkeiten einen cut z.B. gemäß des last UIPs oder andere auszutesten.

Wir haben bei manchen Funktion im Programmcode zudem die Technik der *Memoisation* verwendet [Hug85]. Dadurch konnte das Programm erheblich beschleunigt werden, indem die Rückgabewerte einer Funktion zwischengespeichert wurden, anstatt sie bei jedem Aufruf immer wieder neu zu berechnen. Mit dieser Technik wurden z.B. in der Funktion $causes\_of(x, u_i)$ die Rückgabewerte der Mengen $\Lambda(x)$ und $\Sigma(x)$ nur einmal pro x und nicht immer wieder neu berechnet.

6.6 Entscheidungsheuristik

Wir verwenden in der Implementierung unter anderem eine Entscheidungsheuristik, die dasjenige Literal auswählt, welches am häufigsten vorkommt. Die Auswahl einer Variable geschieht dabei in der `decide` Funktion. Wichtig ist, dass bei einem möglichen Umkehren der Variablenbelegung später in der `flipLit` Funktion nach einem Backjump auch genau dieselbe Variable, die vorher in der `decide` Funktion ausgewählt wurde, umgekehrt wird. Insbesondere ist dies in der Implementierung ebenfalls sichergestellt. Daher besteht durchaus die Möglichkeit auch andere Entscheidungsheuristiken, wie etwa eine randomisierte Auswahl einer Variablen, bequem anzuknüpfen und auszutesten.

```
1  decide :: State → (State,Int)
2  decide (State m c c_0) = let lit = chooseLiteral2 c in
3                ((State ((lit, D, chooseLitIndex c, inc_d_level m): m) ((assignTrue
                     2 lit) c) c_0),lit)
```

Wir können also durch eine neue Definition der Funktion `chooseLiteral2` (Zeile 2) im obigen Ausschnitt des Quellcodes eine andere Entscheidungsheuristik leicht an

den Algorithmus anklemmen. Diesbezüglich werden wir im Rahmen unserer Tests im Kapitel 7 einige verschiedene Entscheidungsheuristiken untersuchen.

7 Experimentelle Ergebnisse

Die nachfolgenden Experimente wurden auf einem Intel Pentium M (Centrino) 2.13 GHz Notebook mit 1GB RAM Arbeitsspeicher durchgeführt. Alle Varianten der DPLL Algorithmen sind in Haskell implementiert und werden mit dem GHC Kompiler, Version 6.10.1 [JHH+93] kompiliert und ausgeführt.

7.1 Die DIMACS CNF-Darstellung

Bevor wir zu den eigentlichen Testergebnissen unseres implementierten SAT-Solvers kommen, führen wir noch kurz die weitverbreitete und zu Testzweckem oft verwendete DIMACS CNF-Darstellung als Eingabedatei in SAT-Solver ein [JT93], welche wir ebenfalls im Rahmen unserer Experimente verwendet haben. Eine dafür erforderliche Schnittstelle zum Einlesen und Verarbeiten einer DIMACS-Datei[1] ist ebenfalls implementiert.

Die DIMACS Eingabedatei, wie in Abbildung 7.1, vereinfacht und standardisiert die Formulierung eines SAT-Problems dabei wie folgt:

- eine Zeile, die mit „c“ beginnt, ist eine Kommentarzeile und wird während der Verarbeitung durch den SAT-Solver ignoriert.
- die nächste Zeile beginnt mit einem „p“ und signalisiert damit die Problemzeile, in der nacheinander zunächst das „cnf“ Format, die Anzahl der n Variablen und die Anzahl der Klauseln angegeben wird.
- direkt nach der Problemzeile folgt die Kodierung des eigentlichen Problems:
  - eine Variable wird als Zahl x von 1 bis n dargestellt und ihre Negation als $-x$.
  - jede Klausel ist mit einer Folge von Zahlen (ihren Literalen), zwischen denen immer eine Leertaste frei ist, in einer Zeile getrennt. Das Ende einer Klausel wird durch eine 0 markiert.
  - eine solche gespeicherte Datei hat dann immer die Endung „`.cnf`“

[1] siehe CD Verzeichnis Benchmarks/dimacs/

Für die Formel $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4 \vee x_3) \wedge (x_5 \vee x_1 \vee \neg x_2)$ lautet die DIMACS CNF-Darstellung:

```
c Eingabedatei generiert aus der Formel F
p cnf 5 3
1 2 -3 0
-2 4 3 0
5 1 -2 0
```

Abbildung 7.1: Beispiel für eine DIMACS Eingabedatei

7.2 DIMACS Benchmark Resultate

7.2.1 AIM Benchmark-Tests

Zuerst wird die implementierte Variante des klassischen DPLL Algorithmus (nachfolgend mit kl. DPLL abgekürzt) anhand der Benchmark Klasse *AIM* mit den implementierten Varianten des modernen DPLL Algorithmus verglichen (fUIP ist eine Abkürzung für first UIP, DLIS und MOM bezeichnen Enscheidungsheuristiken, siehe Abschnitt 5.3). Dabei wird die CPU Rechenzeit jeweils in Sekunden angegeben.

Diejenige Variante, die sich auf der Grundlage AIM-100 Benchmark-Tests als beste Variante hervorhebt, wird anschließend anhand der implementierten Entscheidungsheuristiken noch einmal eingehend auch an größeren Problem Instanzen untersucht. Zusätzlich wird das Verhalten dieser Variante an unterschiedlichen Benchmark Klassen wie *DUBOIS*, *HOLE*, *II32*, *BF*, *HANOI*, *SSA*, *PAR* und *PRET* getestet. Es folgen dann einige Tests, die die verschiedenen Entscheidungsheuristiken an einigen ausgewählten DPLL Varianten ausprobieren.

Innerhalb einer Tabelle sind die Testfälle wiederum nach ihrer Erfüllbarkeit sortiert, wobei der Infix -no- eine unerfüllbare und -yes- eine erfüllbare Probleminstanz[2] kennzeichnet. Weiterhin deutet die Zahl nach der Benchmarkbenennung auf die zugrundeliegende Anzahl der Variablen in einer DIMACS Datei hin, z.B. liegen den Eingabedateien *AIM-50* insgesamt 50 Variablen zugrunde, in *AIM-100* sind es 100 Variablen und in *AIM-200* insgesamt 200 Variablen. Die Anzahl der Klauseln liegt dabei in einigen Fällen zwischen 80 und in manchen AIM-200 Testfällen bei 1200

[2] Auf der beigefügten CD befinden sich zusätzlich die log-Dateien der durchgeführten Tests: im Falle der Erfüllbarkeit können die Modelle daher im Verzeichnis Benchmarks/Tests/ eingesehen werden

Klauseln. Für eine genaue Information über die Anzahl der Klauseln sei im Einzelfall auf die folgende Referenz verwiesen [JT93].[3]

Benchmark	kl. DPLL	B. u. L.	first UIP	fUIP/DLIS
aim-50-1_6-no-1	0.078125	0.015625	0.015625	0.046875
aim-50-1_6-no-2	0.046875	0.0	0.015625	0.140625
aim-50-1_6-no-3	0.109375	0.015625	0.03125	0.09375
aim-50-1_6-no-4	0.015625	0.0	0.015625	0.046875
aim-50-2_0-no-1	0.078125	0.0	0.03125	0.03125
aim-50-2_0-no-2	0.03125	0.015625	0.046875	0.1875
aim-50-2_0-no-3	0.015625	0.015625	0.015625	0.046875
aim-50-2_0-no-4	0.015625	0.015625	0.015625	0.078125
aim-50-1_6-yes1-1	0.0	0.015625	0.03125	0.109375
aim-50-1_6-yes1-2	0.0	0.0	0.015625	0.109375
aim-50-1_6-yes1-3	0.0	0.015625	0.03125	0.109375
aim-50-1_6-yes1-4	0.0	0.015625	0.015625	0.015625
aim-50-2_0-yes1-1	0.0	0.015625	0.03125	0.109375
aim-50-2_0-yes1-2	0.0	0.0	0.0	0.109375
aim-50-2_0-yes1-3	0.0	0.015625	0.015625	0.015625
aim-50-2_0-yes1-4	0.0	0.015625	0.015625	0.109375
aim-50-3_4-yes1-1	0.0	0.03125	0.03125	0.125
aim-50-3_4-yes1-2	0.03125	0.078125	0.125	0.0625
aim-50-3_4-yes1-3	0.03125	0.09375	0.0625	0.234375
aim-50-3_4-yes1-4	0.03125	0.078125	0.125	0.03125
aim-50-6_0-yes1-1	0.015625	0.015625	0.015625	0.25
aim-50-6_0-yes1-2	0.03125	0.046875	0.0625	0.078125
aim-50-6_0-yes1-3	0.015625	0.046875	0.046875	0.125
aim-50-6_0-yes1-4	0.015625	0.03125	0.03125	0.125
$\sum$ AIM-50	0.5625	0.5625	1	2.390625

Tabelle 7.1: CPU Rechenzeit für DPLL Varianten in Haskell bzgl. AIM-50 DIMACS Benchmark-Tests

Im Ergebnis lässt sich auf der Basis der AIM-50 Testfälle allgemein kein wesentlicher Unterschied zwischen der klassischen DPLL Variante mit chronologischem-backtracking und der modernen Varianten mit nicht-chronologischen-backtracking feststellen. Unter Umständen sind sogar die modernen Varianten aufgrund der zusätzlichen Berechnungen (z.B. Backjump-Klausel, backjump-level, finden des first UIPs) vereinzelt geringfügig langsamer als die klassische Variante. Aus diesem Grund

[3] eine detaillierte Auflistung der Benchmark-Dateien liegt als pdf-Dokument dem Inhalt der CD im Verzeichnis Benchmarks/Ueberblick_Benchmark.pdf bei

Benchmark	kl. DPLL	B. u. L.	first UIP	fUIP/DLIS
aim-100-1_6-no-1	13.34375	0.046875	0.34375	0.640625
aim-100-1_6-no-2	17.109375	0.21875	0.484375	0.921875
aim-100-1_6-no-3	12.5	0.0625	0.15625	1.0
aim-100-1_6-no-4	5.984375	0.046875	0.234375	0.59375
aim-100-2_0-no-1	0.015625	0.015625	0.015625	0.046875
aim-100-2_0-no-2	6.125	0.0625	0.203125	0.46875
aim-100-2_0-no-3	0.109375	0.03125	0.03125	0.09375
aim-100-2_0-no-4	0.171875	0.03125	0.0625	0.359375
aim-100-1_6-yes1-1	0.0	0.015625	0.015625	0.328125
aim-100-1_6-yes1-2	0.0	0.03125	0.078125	0.46875
aim-100-1_6-yes1-3	0.03125	0.171875	0.359375	1.296875
aim-100-1_6-yes1-4	0.0	0.015625	0.03125	0.40625
aim-100-2_0-yes1-1	0.03125	0.09375	0.15625	0.109375
aim-100-2_0-yes1-2	0.015625	0.09375	0.203125	1.015625
aim-100-2_0-yes1-3	0.0625	0.15625	0.234375	0.09375
aim-100-2_0-yes1-4	0.03125	0.109375	0.21875	0.34375
aim-100-3_4-yes1-1	0.265625	0.25	0.328125	2.390625
aim-100-3_4-yes1-2	0.078125	0.0625	0.125	0.53125
aim-100-3_4-yes1-3	1.015625	0.71875	0.640625	1.671875
aim-100-3_4-yes1-4	3.25	1.65625	1.53125	0.671875
aim-100-6_0-yes1-1	0.71875	0.203125	0.375	0.484375
aim-100-6_0-yes1-2	0.453125	0.296875	0.546875	0.828125
aim-100-6_0-yes1-3	0.234375	0.0625	0.109375	0.390625
aim-100-6_0-yes1-4	0.578125	0.3125	0.46875	1.328125
$\sum$ AIM-100	62.125	4.77	6.953	16.43

Tabelle 7.2: CPU Rechenzeit für DPLL Varianten in Haskell bzgl. AIM-100 DIMACS Benchmark-Tests

Benchmark	**kl. DPLL**	**B. u. L.**	**first UIP**	**fUIP/DLIS**
aim-200-1_6-no-1	> 300	0.109375	0.265625	3.28125
aim-200-1_6-no-2	> 300	0.125	0.703125	4.75
aim-200-1_6-no-3	> 300	1.03125	4.328125	7.15625
aim-200-1_6-no-4	13.234375	0.046875	0.125	2.28125
aim-200-2_0-no-1	137.609375	0.09375	0.453125	4.15625
aim-200-2_0-no-2	178.484375	0.0625	0.375	1.453125
aim-200-2_0-no-3	3.046875	0.046875	0.109375	1.703125
aim-200-2_0-no-4	15.46875	0.046875	0.09375	2.0
aim-200-1_6-yes1-1	0.015625	0.078125	0.140625	2.609375
aim-200-1_6-yes1-2	0.09375	0.234375	0.375	9.265625
aim-200-1_6-yes1-3	0.0	0.03125	0.078125	2.8125
aim-200-1_6-yes1-4	0.171875	0.125	0.265625	4.328125
aim-200-2_0-yes1-1	3.640625	0.734375	1.46875	10.203125
aim-200-2_0-yes1-2	0.046875	0.640625	1.453125	44.40625
aim-200-2_0-yes1-3	1.265625	1.09375	2.578125	10.84375
aim-200-2_0-yes1-4	0.859375	0.890625	2.8125	17.46875
aim-200-3_4-yes1-1	19.1875	2.515625	3.59375	19.59375
aim-200-3_4-yes1-2	> 300	45.359375	9.78125	27.875
aim-200-3_4-yes1-3	35.0625	2.5625	1.65625	27.1875
aim-200-3_4-yes1-4	4.046875	1.125	1.515625	28.78125
aim-200-6_0-yes1-1	0.859375	0.390625	0.421875	0.265625
aim-200-6_0-yes1-2	0.578125	0.765625	1.390625	3.484375
aim-200-6_0-yes1-3	0.734375	0.4375	0.6875	12.421875
aim-200-6_0-yes1-4	0.828125	0.21875	0.328125	1.484375
$\sum$ AIM-200/Ø	415.23/∼Ø67.30	58.79/Ø2.45	34.99/Ø1.46	221.93/Ø9.25

Tabelle 7.3: CPU Rechenzeit für DPLL Varianten in Haskell bzgl. AIM-200 DIMACS Benchmark-Tests

schauen wir uns das Verhalten noch einmal innerhalb der selben Benchmark Klasse jedoch mit einer größeren Anzahl an Variablen und Klauseln anhand von AIM-100 und AIM-200 an.

In der Summe hebt sich bei den AIM-100 Benchmark-Tests die moderne Variante (Backjump mit Learn) im Gegensatz zu der klassischen Variante sehr deutlich hervor. Während die klassische Variante im zweistelligen Sekundenbereich liegt, bleibt die Backjump und Learn Variante für dieselben Testfälle sogar im Bereich von nur wenigen hundertstel Sekunden. Das nicht-chronologische-backtracking liegt in allen Fällen gegenüber dem chronologischen-backtracking der klassischen Variante immer ganz deutlich vorne. Die first UIP Variante verglichen mit der Backjump und Learn Variante ist im Durchschnitt immer um ein paar zehntel Sekunden langsamer, sodass sich in der Summe nur ein kleiner Unterschied von einigen Sekunden ergibt.

Man sieht auch deutlich den Unterschied in der Variante mit der zusätzlichen Entscheidungsheuristik DLIS, verglichen mit der first UIP Variante ohne DLIS. Scheinbar ist die Entscheidungsheuristik in seiner Berechnung noch zu „teuer“ (liegt vorallem in der Verwendung der `nub` Funktion, die leider eine quadratische Laufzeit hat). Jedoch lassen sich mit einer modifizierten Implementierung der Version der DLIS Heuristik ohne die `nub` Funktion erstaunlich gute Verbesserungen bei ausreichend großen Problem Instanzen erreichen (siehe z.B. Tabellen 7.5, 7.11). Dennoch ist die implementierte Entscheidungsheuristik Variante in einem Fall auch schneller als alle anderen Varianten, wie man im Testfall aim-100-3_4-yes1-4 sieht. Zudem schneidet die first UIP Variante im Durchschnitt geringfügig besser ab, als die Backjump und Learn Variante bei den AIM-100 Benchmark-Tests.

Noch deutlicher wird die erhebliche Verbesserung in der modernen Variante gegenüber der klassischen Variante im AIM-200 Benchmark. Bei einigen Testfällen wurden die Läufe in der klassischen Variante nach einer Zeitbeschränkung von 300 Sekunden abgebrochen. Allgemein kann man sagen, dass auch im AIM-200 Bechnmark die vorher gemachten Beobachtungen zum Teil zutreffen, wobei die Differenzen hier deutlicher sichtbar sind.

7.2.2 Weitere Benchmark Klassen und Heuristiken

Nachfolgend werden zusätzlich weitere Testergebnisse mit verschiedenen Benchmark Klassen angegeben, welche anhand der DPLL Variante mit Backjump und Learn teilweise mit und ohne[4] Entscheidungsheuristiken getestet wurden.

[4] es wird immer das erste Literal der ersten Klausel ausgewählt

	Backjump / Learn				
Benchmark	**# Var**	**# Klauseln**	**Erfüllbar?**	**ohne Heuristik**	**DLIS**
dubois20.cnf	60	160	nein	0.25	5.44
dubois21.cnf	63	168	nein	0.296875	6.69
dubois22.cnf	66	176	nein	0.328125	7.59
dubois23.cnf	69	184	nein	0.359375	10.41
dubois24.cnf	72	192	nein	0.421875	14.13
dubois25.cnf	75	200	nein	0.453125	15.34
dubois26.cnf	78	208	nein	0.515625	20.43
dubois27.cnf	81	216	nein	0.5625	19.17
dubois28.cnf	84	224	nein	0.625	30.73
dubois29.cnf	87	232	nein	0.6875	26.81
dubois30.cnf	90	240	nein	0.75	27.97
dubois50.cnf	150	400	nein	3.078125	557.69

Tabelle 7.4: CPU Rechenzeit für Backjump und Learn anhand der DUBOIS Benchmark Klasse

	Backjump / Learn				
Benchmark	**# Var**	**# Klauseln**	**Erfüllbar?**	**ohne Heuristik**	**DLIS**
hole6.cnf	42	133	nein	2.03125	0.44
hole7.cnf	56	204	nein	14.53125	2.39
hole8.cnf	72	297	nein	95.140625	10.28
hole9.cnf	90	415	nein	> 900	43.94
hole10.cnf	110	561	nein	> 900	190.22

Tabelle 7.5: CPU Rechenzeit für Backjump und Learn anhand der HOLE Benchmark Klasse mit und ohne Heuristik

	Backjump / Learn			
Benchmark	**# Variablen**	**# Klauseln**	**Erfüllbar?**	**CPU Zeit**
pret60_25.cnf	60	160	nein	0.0
pret60_40.cnf	60	160	nein	0.015625
pret60_60.cnf	60	160	nein	0.0
pret60_75.cnf	60	160	nein	0.015625
pret150_25.cnf	150	400	nein	0.0
pret150_40.cnf	150	400	nein	0.0
pret150_60.cnf	150	400	nein	0.0
pret150_75.cnf	150	400	nein	0.015625

Tabelle 7.6: CPU Rechenzeit für Backjump und Learn anhand der PRET Benchmark Klasse

	Backjump / Learn			
Benchmark	**# Variablen**	**# Klauseln**	**Erfüllbar?**	**CPU Zeit**
par8-1-c.cnf	64	254	ja	0.140625
par8-2-c.cnf	68	270	ja	0.484375
par8-3-c.cnf	75	298	ja	0.296875
par8-4-c.cnf	67	266	ja	0.125
par8-5-c.cnf	75	298	ja	0.421875

Tabelle 7.7: CPU Rechenzeit für Backjump und Learn anhand der PAR8 Benchmark Klasse

7.2.3 Große Problem Instanzen

	Backjump / Learn			
Benchmark	**# Variablen**	**# Klauseln**	**Erfüllbar?**	**CPU Zeit**
ii32b1.cnf	228	1374	ja	0.125
ii32b2.cnf	261	2558	ja	41.46875
ii32b3.cnf	348	5734	ja	82.203125
ii32b4.cnf	381	9618	ja	371.53125
hanoi4.cnf	718	4934	ja	386.421875
ssa0432-003.cnf	435	1027	nein	146.828125

Tabelle 7.8: CPU Rechenzeit für Backjump und Learn anhand der II32, HANOI und der SSA Benchmark Klasse

	Backjump / Learn			
Benchmark	**# Variablen**	**# Klauseln**	**Erfüllbar?**	**CPU Zeit**
ii8b1.cnf	336	2068	ja	0.27
ii8b2.cnf	576	4088	ja	84.8
ii8b3.cnf	816	6108	ja	212.36
ii8b4.cnf	1086	8214	ja	916.47
ii8c1.cnf	510	3065	ja	0.8
ii8c2.cnf	950	6689	ja	449.40
ii8d1.cnf	530	3207	ja	155.40
ii8d2.cnf	930	6547	ja	1152.88
ii8e1.cnf	520	3136	ja	9.70
ii8e2.cnf	870	6121	ja	679.8

Tabelle 7.9: CPU Rechenzeit für Backjump und Learn anhand der II8 Benchmark Klasse

	Backjump / Learn				
Benchmark	**# Variablen**	**# Klauseln**	**Erfüllbar?**	**DLIS**	**MOM**
bf0432-007.cnf	1040	3668	nein	592.03	1644.1
bf1355-075.cnf	2180	6778	nein	313.16	465.88
bf1355-638.cnf	2177	4768	nein	425.47	581.81
bf2670-001.cnf	1393	3434	nein	43.03	187.67

Tabelle 7.10: CPU Rechenzeit für Backjump und Learn anhand der BF Benchmark Klasse mit DLIS und MOM Heuristik

	Backjump / Learn				
Benchmark	**# Var**	**# Klauseln**	**Erfüllbar?**	**ohne Heuristik**	**DLIS**
ssa0432-003.cnf	435	1027	nein	147.31	5.23
ssa2670-130.cnf	1359	3321	nein	124.0	312.0
ssa2670-141.cnf	986	2315	nein	160.84	87.75
ssa6288-047.cnf	10410	34238	nein	1.0	1.03
ssa7552-038.cnf	1501	3575	ja	84.73	0.84
ssa7552-158.cnf	1363	3034	ja	16.14	6.34
ssa7552-159.cnf	1363	3032	ja	36.69	9.92
ssa7552-160.cnf	1391	3126	ja	42.42	4.39

Tabelle 7.11: CPU Rechenzeit für Backjump und Learn anhand der SSA Benchmark Klasse mit und ohne Heuristik

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde auf der Basis des klassischen DPLL Algorithmus [DLL62] eine in der Laufzeit wesentlich verbesserte Variante des DPLL Algorithmus in der funktionalen Programmiersprache Haskell implementiert, mit der sich boolesche Variablen auf (Un)Erfüllbarkeit hin überprüfen lassen. Bei einer Erfüllbarkeit lässt sich ein zugrundeliegendes Modell ausgeben. Die verbesserte Variante verwendet dabei Techniken aus aktuellen SAT-Solvern, wie etwa dem *backjumping*, eine Form des nicht-chronologischen-backtrackings, sowie dem *conflict-driven-learning* [MSS99], ein um Lernen von Klauseln erweitertes Schema, wobei beide Techniken zusammen zu bahnbrechenden Verbesserungen in modernen SAT-Solvern geführt haben. Im Rahmen weiterer Verbesserungen wurde das first UIP Lernschema implementiert, welches auf der Basis umfangreicher Experimente in [ZM01] als robust und effizient gilt und aus diesem Grund ebenfalls in den meisten modernen SAT-Solvern verwendet wird. Darüberhinaus wurden weitere mögliche Verbesserungen aus modernen SAT-Solvern vorgestellt und zudem verschiedene Entscheidungsheuristiken implementiert. Abschließend wurden die verschiedenen DPLL Varianten hinsichtlich ihrer CPU Rechenzeit anhand unterschiedlicher Benchmark-Tests untersucht.

8.2 Ausblick

8.2.1 Andere Entscheidungsheuristiken

Es wäre weiterhin interessant zu prüfen, wie sich andere Entscheidungsheuristiken aus dem Abschnitt 5.3 auf die Laufzeit des Algorithmus auswirken. Des Weiteren könnte man in diesem Zusammenhang die Datenstruktur derart verändern, dass die Auswahl einer Variablen, ähnlich wie beim VSIDS durch einen Zähler priorisiert wird, wobei der Zähler für eine Variable immer dann aktualisiert wird, wenn diese in einer kürzlich hinzugefügten Klausel vorkommt. Aktuell wird die Häufigkeit der Variable jedesmal für die Liste der Formel neu ermittelt, was bei einer zunehmenden Listengröße noch sehr rechenintensiv ist.

8.2.2 First UIP Lernschema ohne Implikationsgraph

Man könnte unter Umständen auch die Konflikt-Analyse gänzlich ohne einen Implikationsgraphen durchführen. Das Ermitteln der Backjump-Klausel mit einem first UIP basiert dabei auf Resolution und könnte das travesieren eines Implikationsgraphen ersparen [Rya04].

8.2.3 Restart

Weiterhin könnte man aufgrund von Problem Instanzen mit auffallend langer Laufzeit eine Restart Strategie implementieren, und diesbezüglich das Verhalten des Algorithmus untersuchen. Hierbei würde sich z.B. anbieten, auch randomisierte Restart Strategien auszuprobieren. In der jetzigen Implementierung hätte man bereits die Möglichkeit direkt mit einem backjump-level von 0 an die Wurzel des Entscheidungsbaums zu springen und dabei gleichzeitig die gelernten Klauseln beizubehalten, womit sich ein Restart bereits umsetzen ließe. Entscheidend wäre dann den Restart erst nach einem bestimmten Kriterium (z.B. nach einer bestimmten Anzahl von Konflikten, oder etwa zufällig) anzustoßen.

8.2.4 Erweiterung des DPLL mit SAT Modulo Theorien

Unser Algorithmus liefert insbesondere aufgrund der Implementierung in Haskell eine gute Grundlage um DPLL mit SAT Modulo Theorien (SMT) [NOT06] zu erweitern, mit dem Ziel der Konstruktion eines Deduktionssystems für Prädikatenlogik. DPLL mit Theorien kann man daher als eine Erweiterung der implementierten `dpll` Funktion für Aussagenlogik betrachten: die Formel wird dabei in zwei Teile $F \cup T$ zerlegt, sodass `dpll` auf F arbeitet, und es einen Algorithmus (Solver für T) bereits gibt, der von der `dpll` Funktion ab und zu befragt wird und dann mit den Anworten des T-Solvers weiter fortfährt. Dieser Anteil kann auch eine nicht-Aussagenlogik sein, z.B. die Prädikatenlogik erster Stufe.

9 Literaturverzeichnis

[BCCZ99] Biere, Armin ; Cimatti, Alessandro ; Clarke, Edmund M. ; Zhu, Yunshan: Symbolic Model Checking without BDDs. In: *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. London, UK : Springer-Verlag, 1999. – ISBN 3–540–65703–7, S. 193–207

[Bry86] Bryant, Randal E.: Graph-Based Algorithms for Boolean Function Manipulation. In: *IEEE Trans. Comput.* 35 (1986), Nr. 8, S. 677–691. – ISSN 0018–9340

[CA93] Crawford, James M. ; Auton, Larry D.: Experimental Results on the Crossover Point in Satisfiability Problems. In: *In Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993, S. 21–27

[Coo71] Cook, Stephen A.: The complexity of theorem-proving procedures. In: *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, 1971, S. 151–158

[DLL62] Davis, Martin ; Logemann, George ; Loveland, Donald: A machine program for theorem-proving. In: *Commun. ACM* 5 (1962), Nr. 7, S. 394–397. – ISSN 0001–0782

[DP60] Davis, Martin ; Putnam, Hilary: A Computing Procedure for Quantification Theory. In: *J. ACM* 7 (1960), Nr. 3, S. 201–215. – ISSN 0004–5411

[Fre95] Freeman, Jon W.: *Improvements to Propositional Satisfiability Search Algorithms*. Philadelphia, Departement of computer and Information science, University of Pennsylvania, Diss., 1995

[GN02] Goldberg, E. ; Novikov, Y.: BerkMin: A Fast and Robust Sat-Solver. In: *DATE '02: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA : IEEE Computer Society, 2002, S. 142

[GPFW96] Gu, Jun ; Purdom, Paul W. ; Franco, John ; Wah, Benjamin W.: Algorithms for the Satisfiability (SAT) Problem: A Survey. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1996, S. 19–152

[GSK98] Gomes, Carla P. ; Selman, Bart ; Kautz, Henry: Boosting Combinatorial Search Through Randomization. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*. Madison, Wisconsin, 1998, S. 431–437

[Hay79] Hayes, Patrick J.: The naive physics manifesto. In: Michie, Donald (Hrsg.): *Expert systems in the microelectronic age*. Edinburgh, Scotland : Edinburgh University Press, 1979

[Hua07] Huang, Jinbo: The Effect of Restarts on the Efficiency of Clause Learning. In: *IJCAI*, 2007, S. 2318–2323

[Hug85] Hughes, John: Lazy memo-functions. In: *Proc. of a conference on Functional programming languages and computer architecture*. New York, NY, USA : Springer-Verlag New York, Inc., 1985. – ISBN 3–387–15975–4, S. 129–146

[IBB92] Informatik, Reihe ; Buro, M. ; Büning, H. K.: *Report on a SAT competition*. 1992

[JHH+93] Jones, Simon L. P. ; Hall, Cordelia V. ; Hammond, Kevin ; Partain, Will ; Wadler, Philip: *The Glasgow Haskell compiler: a technical overview*. 1993 http://www.haskell.org/ghc/

[JS97] Jr., Roberto J. B. ; Schrag, Robert: Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In: *AAAI/IAAI*, 1997, S. 203–208

[JT93] Johnson, D. S. (Hrsg.) ; Trick, M. A. (Hrsg.): *Second DIMACS implementation challenge*. 1993 (DIMACS Series in Discrete Mathematics and Theoretical Computer Science). ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/

[JW90] Jeroslow, Robert G. ; Wang, Jinchang: Solving Propositional Satisfiability Problems. In: *Ann. Math. Artif. Intell.* 1 (1990), S. 167–187

[KS92] Kautz, Henry ; Selman, Bart: Planning as satisfiability. In: *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*. New York, NY, USA : John Wiley & Sons, Inc., 1992. – ISBN 0–471–93608–1, S. 359–363

[LMS01] Lynce, Inês ; Marques-Silva, João: *The Puzzling Role of Simplification in Propositional Satisfiability*. 2001

[LMS05] Lynce, Inês ; Marques-Silva, João: Efficient Data Structures for Backtrack Search SAT Solvers. In: *Annals of Mathematics and Artificial Intelligence* 43 (2005), Nr. 1-4, S. 137–152. – ISSN 1012–2443

[McM02] McMillan, Kenneth L.: Applying SAT Methods in Unbounded Symbolic Model Checking. In: *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. London, UK : Springer-Verlag, 2002. – ISBN 3–540–43997–8, S. 250–264

[MM97] M Manzano, Alonzo C.: his life, his work and some of his miracles. (1997), S. 211–232

[MMZ+01] Moskewicz, Matthew W. ; Madigan, Conor F. ; Zhao, Ying ; Zhang, Lintao ; Malik, Sharad: Chaff: engineering an efficient SAT solver. In: *DAC '01: Proceedings of the 38th conference on Design automation*. New York, NY, USA : ACM, 2001. – ISBN 1–58113–297–2, S. 530–535

[MS98] Mary Sheeran, Stålmark G.: A tutorial on Stålmark's proof procedure for propositional logic. In: *Volume 152 of Lecture Notes in Computer Science*, Springer Verlag, 1998, S. 82–99

[Ms99] Marques-silva, João: The impact of branching heuristics in propositional satisfiability algorithms. In: *In 9th Portuguese Conference on Artificial Intelligence (EPIA*, 1999, S. 62–74

[MSS99] Marques-Silva, Jo ao P. ; Sakallah, Karem A.: GRASP: A Search Algorithm for Propositional Satisfiability. In: *IEEE Trans. Comput.* 48 (1999), Nr. 5, S. 506–521. – ISSN 0018–9340

[NOT06] Nieuwenhuis, Robert ; Oliveras, Albert ; Tinelli, Cesare: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). In: *J. ACM* 53 (2006), Nr. 6, S. 937–977. – ISSN 0004–5411

[OSG08] O'Sullivan, Bryan ; Stewart, Donald ; Goerzen, John: *Real World Haskell*. O'Reilly Media, Inc., 2008. – ISBN 0596514980

[PeHeA+99] Peyton Jones [editor], Simon ; Hughes [editor], John ; Augustsson, Lennart ; Barton, Dave ; Boutel, Brian ; Burton, Warren ; Fraser, Simon ; Fasel, Joseph ; Hammond, Kevin ; Hinze, Ralf ; Hudak, Paul ; Johnsson, Thomas ; Jones, Mark ; Launchbury, John ; Meijer, Erik ; Peterson, John ; Reid, Alastair ; Runciman, Colin ; Wadler, Philip ; Peyton Jones, Simon (Hrsg.) ; Hughes, John (Hrsg.): *Haskell 98 — A Non-strict, Purely Functional Language*. 1999

[Pro93] Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. In: *Computational Intelligence, Volume 9, Number 3* (1993), S. 268–299

[Rec06] Rechenberg, Peter: *Informatik-Handbuch*. Hanser Verlag, 2006

[Rob65] Robinson, J. A.: A Machine-Oriented Logic Based on the Resolution Principle. In: *J. ACM* 12 (1965), Nr. 1, S. 23–41. – ISSN 0004–5411

[Rya04] Ryan, Lawrence: *Efficient Algorithms for Clause-learning Sat Solvers*, Diplomarbeit, 2004

[Sch06] Schmidt-Schauss, Manfred: *Skript zur Vorlesung „Einführung in die Methoden der Künstlichen Intelligenz“*. http://www.ki.informatik.uni-frankfurt.de/lehre/SS2006/KI/, (Sommersemester 2006)

[Sch09] Schmidt-Schauss, Manfred: *Skript zur Vorlesung „Einführung in die Funktionale Programmierung“*. http://www.ki.informatik.uni-frankfurt.de/lehre/WS2008/EFP/, (Wintersemester 2008 / 2009)

[SLM92] Selman, Bart ; Levesque, Hector ; Mitchell, D.: A new method for solving hard satisfiability problems. In: *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, 1992, S. 459–465

[Stå89] Stålmarck, G.: A system for determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are generated from a formula / European Patent Nr. 0403 454 (1995), US Patent Nr. 5 276 897, Swedish Patent Nr. 467 076 (1989). 1989. – Forschungsbericht

[Tar74] Tarjan, Robert E.: Finding Dominators in Directed Graphs. In: *SIAM J. Comput.* 3 (1974), Nr. 1, S. 62–89

[Tho97] Thompson, Simon: *Haskell: the craft of functional programming*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. – ISBN 0201403579

[Tse68] Tseitin, G. S.: On the complexity of derivations in the propositional calculus. In: *Studies in Mathematics and Mathematical Logic* Part II (1968), S. 115–125

[Zha97] Zhang, Hantao: SATO: An Efficient Propositional Prover. In: *CADE-14: Proceedings of the 14th International Conference on Automated Deduction*. London, UK : Springer-Verlag, 1997. – ISBN 3–540–63104–6, S. 272–275

[ZM01] Zhang, Lintao ; Madigan, Conor F.: Efficient conflict driven learning in a boolean satisfiability solver. In: *In ICCAD*, 2001, S. 279–285