
# Automatisches Lösen von Rätseln mit modernen SMT-Solvern

Professur für Künstliche Intelligenz
der Goethe-Universität Frankfurt

**Masterarbeit**

zur Erlangung des akademischen Grades
Master of Science

vorgelegt von

**Harald Winkler**
geboren am 13.04.1988 in Frankfurt

im Februar 2016

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß

Selbstständigkeitserklärung gemäß §24 Abschnitt 12 der Masterordung

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

Frankfurt, den 09.Februar 2016

Harald Winkler

# Zusammenfassung / Abstract


## Abstract

Die vorliegende Masterarbeit befasst sich mit dem automatischen Lösen von Logikrätseln unter Verwendung des modernen SMT-Solvers „Yices“. Dazu wurden zunächst die theoretischen Grundlagen von SMT („Satisfiability Modulo Theories“)-Solvern behandelt und darauf die Funktionsweisen des speziellen SMT-Solver Yices erläutert. Dieser ist in der Lage das Erfüllbarkeitsproblem gekoppelt mit verschiedenen Hintergrundtheorien zu lösen.

Für den Aufbau von Logikrätseln wurde eine getypte Prädikatenlogik auf endliche Mengen, die „Rätsellogik“, entwickelt, die neben den üblichen auch anzahlbeschränkte Existenzquantoren enthält.

Ziel dieser Arbeit war es einen Parser zu entwickeln, der in einer Textdatei verfasste Logeleien in die von Yices verlangte Eingabesprache transformieren soll. Dadurch sollte es auch dem Normalbenutzer ermöglicht werden, die Logikrätsel in einer möglichst natürlichsprachlichen Form aufzustellen. Sobald das Logikrätsel in die Eingabesprache von Yices verwandelt worden ist, kann der SMT-Solver überprüfen, ob dieses Rätsel lösbar ist und gegebenenfalls ein Lösungsmodell ausgeben.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

Anfang der 1960er Jahre wurden erste Algorithmen mit Hilfe von Entscheidungsprozeduren für das Lösen des aussagenlogischen Erfüllbarkeitsproblems entwickelt. Diese wurden als SAT-Solver bezeichnet und basieren fast alle auf dem resolutions-basierenden DPLL-Algorithmus. Sie implementieren einen Algorithmus, der als Eingabe eine aussagenlogische Formel einliest und diese anschließend zu beweisen versucht. Als Ausgabe wird je nach Ergebnis „erfüllbar“ oder „widersprüchlich“ zurückgegeben.
Das Erfüllbarkeitsproblem fand eine Reihe von Einsatzmöglichkeiten. Dazu gehören die Erkennung von künstlicher Intelligenz, das Model Checking, ein Verfahren zur vollautomatischen Verifikation einer Systembeschreibung gegen eine Spezifikation, und die automatische Planung.
Nach langer Zeit begann man das SAT-Problem zu erweitern. SAT-Solver wurden immer häufiger erfolgreich mit anderen Entscheidungstheorien kombiniert, mit dem Ziel, entscheidbare Formeln der Prädikatenlogik erster Stufe auf Erfüllbarkeit zu prüfen, da die meisten automatischen Entscheidungsprozeduren diese genannte Logik verstehen.
Das sich ergebende Entscheidungsproblem für logische Formeln wurde als ein SMT-Problem (Satisfiability Modulo Theories) bezeichnet, mit dem man sozusagen die „Erfüllbarkeit modulo Theorien“ betrachtet hat.
Die ersten SMT-Solver wurden schließlich Mitte der 1990er Jahre entwickelt mit dem Ziel prädikatenlogische Formeln zu lösen. Dazu kodierten sie das Problem entweder in ein reines SAT-Problem oder sie verwendeten einen SAT-Solver in Kombination mit verschiedenen Hintergrundtheorien, die jeweils von einem eigenen Theorie-Solver bearbeitet wurden. Zu den kombinierbaren Theorien gehörten *Gleichheit uninterpretierter Funktionssymbole* (EUF), *lineare Arithmetik*, *Äquivalenzlogik*, *Differenzlogik*, *Arrays*, *Listen* und *Bitvektoren*. Für die Kombination dieser Theorien sind Methoden wie die Nelson-Oppen-Methode zuständig.
SMT-Solver werden u.a. beim Model Checking, dem Scheduling, der Testfall-Generierung und dem Äquivalenz Checking eingesetzt.
Sie können aber auch für andere Zwecke genutzt werden, wie zum Beispiel dem Lösen von logischen Rätseln, sogenannten Logeleien. Genau dazu soll in meiner Arbeit der SMT-Solver **Yices** benutzt werden.
Das Ziel dieser Arbeit ist die Entwicklung eines Programms, das verwendet werden kann, um Logikrätsel automatisch zu lösen. Das eigentliche Finden der Lösung soll eben durch den modernen SMT-Solver **Yices** erfolgen. Zusätzlich soll das Programm auch für Privatnutzer relativ einfach und intuitiv zu benutzen sein.

## 1.1 Motivation

Die Motivation dieser Arbeit liegt daher in der Kombination der Benutzung von SMT-Solvern und der Erstellung eines Parsers, der die Eingaben der Logikrätsel in die Eingabesprache von Yices verarbeitet. Dieser neue Code kann schließlich vom Yices-Tool bearbeitet und das eingegebene Rätsel auf Erfüllbarkeit überprüft werden.
Da die Eingabe eines Logikrätsels für den Normalbenutzer in die Eingabesprache von Yices, die auf der Prädikatenlogik erster Stufe basiert, zu kompliziert und zeitaufwändig ist, wurde eine Eingabesprache entwickelt, mit der es dem Nutzer möglich sein soll, Logeleien möglichst einfach, intuitiv und natürlichsprachlich eingeben zu können.
Es wird im Rahmen dieser Arbeit auf eine neue Logik zugegriffen, die speziell für Logikrätsel entwickelt wurde. Diese neue Logik ist eine auf endliche Mengen eingeschränkte und um Sorten erweiterte Prädikatenlogik und wird in dieser Arbeit als „Rätsellogik“ bezeichnet. Zu dieser Logik werden ihre Syntax und ihre Semantik, welche beide sowohl auf der Aussagenlogik als auch auf der Prädikatenlogik größtenteils basieren, vorgestellt.
Die Rätsellogik wird als Zwischenschicht der eigentlichen Benutzereingabe und dem SMT-Solver verwendet, eignet sich aber eher nicht als direkte Eingabeschnittstelle für Benutzer, da diese hierfür Kenntnisse der formalen Sprache der Logik beherrschen müssen. Daher wird eine Eingabeschnittstelle entworfen werden, die Eingaben verarbeiten kann, die näher an der Umgangssprache liegen. Zudem soll in die Eingabeschnittstelle eine präzise Fehlererkennung integriert sein, die detaillierte Fehlermeldungen ausgibt und dem Nutzer hilft diese Fehler zu verbessern.
Die Rätsellogik beinhaltet neben den aussagenlogischen Junktoren und den prädikatenlogischen Quantoren eine erweiterte Form der Existenzquantoren, nämlich die anzahlbeschränkten Existenzquantoren. Diese ermöglichen es dem Nutzer Eingaben der Form „es existieren mindestens x ...“, „... höchstens y ...“ oder „... genau z ...“.

Die Zielgruppe, der in dieser Arbeit entwickelten Eingabe- und Auswertungsmöglichkeit für Logikrätsel, sind zum einen Entwickler solcher Logeleien, die damit die Erfüllbarkeit und Eindeutigkeit ihrer konstruierten Rätsel verifizieren können und Knobler, die eventuell nicht zum Ziel kommen, ihre Fehler finden oder ihre Ergebnisse überprüfen wollen.

Diese Arbeit verfolgt somit drei Ziele: Ein Ziel ist es die theoretischen Grundlagen von SMT-Solvern zu klären, ein anderes Ziel ist es einen Parser zu entwickeln, der die Benutzereingabe in die Eingabesprache des SMT-Solvers Yices umwandelt, um mit diesem Logikrätsel zu lösen und das letzte Ziel ist es die Ergebnisse mit einem SAT-Solver und einer entwickelten intelligenten Benutzungsschnittstelle zu vergleichen und herauszufinden, welche dieser Algorithmen die eingegebenen Logeleien am effizientesten löst.

Der entwickelte Parser und verschiedene Logikrätsel und Anweisungen zur Benutzung der intelligenten Benutzerschnittstelle sind zu finden auf:
http://www.ki.informatik.uni-frankfurt.de/master/programme/logicals-smt/

### 1.1.1 Verwandte Arbeit

Ein verwandte Masterarbeit [Dem15] befasste sich mit dem Thema „Automatisches Lösen von Logikrätseln unter Verwendung von SAT-Solvern mit einer intelligenten Benutzungsschnittstelle“. Bei dieser Arbeit ging es darum ein Programm zu entwickeln, das verwendet werden kann, um Logikrätsel mit Hilfe eines SAT-Solvers automatisch zu lösen. Um das Programm auch für den Privatanwender einfach und intuitiv benutzbar zu machen, wurde in dieser Arbeit eine Benutzungsschnittstelle für die Kommunikation mit dem SAT-Solver entwickelt.
Auch in dieser Arbeit wurde eine Rätsellogik verwendet, die aber im Gegensatz zu meiner verwendeten Rätsellogik, nur logische Operatoren unterstützt. Es wurden einige Testfälle erstellt, dessen Auswertung hinsichtlich ihrer Erfüllbarkeit nach Laufzeit, Speicherverbrauch und Größe der Klauselmengen untersucht worden sind.

Zum Schluss meiner Arbeit werden unter anderem diese Testfälle wiederaufgegriffen und es werden Vergleiche mit den Ergebnissen in Bezug auf Laufzeit und verwendeten Speicherplatz der verwandten Arbeit gezogen.

## 1.2 Aufbau

Meine Arbeit ist in sechs Kapitel unterteilt worden: die Grundlagen, der SMT-Solver Yices, die Rätsellogik, die Implementierung des Parsers, Tests und Ergbnisse und der Schlussteil mit Fazit und Ausblick.

In Kapitel 2 werden alle grundlegenden Themen dieser Arbeit einführend vorgestellt. Dazu wird zunächst der Begriff und die Herkunft der formalen Logik in Abschnitt 2.1 erklärt. Im nächsten Abschnitt wird erklärt, was man unter Logikrätseln versteht und ein für diese Masterarbeit exemplarisches Rätsel vorgestellt, zu welchem und anderen Rätseln im sechsten Kapitel die Erzeugungen, Verarbeitungen und Lösungen vorgestellt werden.
Anschließend wird im Abschnitt 2.3 die boolsche Aussagenlogik eingeführt, beginnend mit Syntax und Grammatik der Aussagenlogik, gefolgt von der Semantik, sowie Deduktionstheoremen, wie der logischen Folgerung, syntaktischen Ableitung und logischen Äquivalenz. Zuletzt werden einige Beispielsätze dieser Logik aufgestellt, um abzugrenzen, welche Aussagen möglich sind und welche nicht.
In Abschnitt 2.4 wird das Erfüllbarkeitsproblem der Aussagenlogik vorgestellt. Dies ist ein Entscheidungsproblem, das testet, ob eine aussagenlogische Formel erfüllbar ist oder nicht. Die Erfüllbarkeit wird von einem SAT-Solver überprüft, der auf dem DPLL-Algorithmus (2.4.2) basiert.
Im fünften Themengebiet wird erweiternd zur Aussagenlogik die Prädikatenlogik vorgestellt, wobei zunächst wieder die Syntax und Grammatik dieser Logik in 2.5.1 beschrieben und dann die einzelnen Bestandteile erklärt werden. Auch hierzu werden am Ende des Abschnitt Beispiele aufgezählt, die die Logik besser skizzieren sollen.
Das letzte Themengebiet befasst sich mit den Grundlagen der „Satisfiability modulo Theo-

ries“. Hier wird die allgemeine Funktionsweise von SMT-Solvern beleuchtet.

In Kapitel 3 wird über den SMT-Solver Yices ein Einblick verschafft. Zu Beginn wird auf die Logik dieses Solvers eingegangen. Dazu gehören das Typsystem, Terme und Formeln und die unterstützten Theorien. Im zweiten Abschnitt wird die Architektur von Yices vorgestellt. Zum einen werden die Hauptkomponenten vorgestellt und es wird auf die Theorie-Solver eingegangen, die Yices unterstützt und zum anderen wird erklärt wie Kontexte konfiguriert werden können. In Abschnitt 3.3 wird schließlich geschlidert wie man das Tool benutzen kann und wie die Eingabesprache definiert ist.

In Kapitel 4 werden die Syntax und die Semantik der für diese Arbeit benutzten Rätsellogik vorgestellt. In Abschnitt 4.1.1 geht es um die Typregeln der Formeln, in Abschnitt 4.2.2 werden die anzahlbeschränkten Existenzquantoren vorgestellt.

Im fünften Kapitel geht es schließlich um die Implementierung. Zunächst wird erklärt, was Parser und Parsergeneratoren sind, wobei vor allem auf den Parsergenerator „Happy“ eingegangen wird und die Funktionsweise eines Shift-Reduce-Parsers erläutert wird.
Im zweiten Abschnitt wird gezeigt wie eine Benutzereingabe eines Logikrätsels aufgebaut ist und anhand einer Beispieleingabe verdeutlicht. Um vor der Transformation der Rätseleingabe in den Yices-Code potenzielle Fehler auszuschließen, wird entsprechend der Typregeln aus Abschnitt 4.1 ein Typcheck gemacht, welcher in Abschnitt 5.3.1 getestet wird.
In Abschnitt 5.4 steht die Implementierung des Parsers, wobei vor allem auf die Datentypen, die Direktiven, die Grammatik und den Lexer eingegangen wird. Daraufhin folgt die Umwandlung der Rätseleingabe in die Eingabesprache von Yices in Abschnitt 5.5. Im letzten Abschnitt des fünften Kapitels werden einige Tests des implementierten Parsers durchgeführt.

In Kapitel 6 werden dem Programm Logikrätsel als Eingabe übergeben und die Ergebnisse der Rätsel untersucht, ebenso wie die Laufzeit und der Speicherplatzverbrauch der einzelnen Testfälle. In Abschnitt 6.2 wird ein Fazit über die Ergebnisse der Tests gezogen und außerdem werden diese Ergebnisse mit denen einer in einer anderen Arbeit entwickelten intelligenten Benutzerschnittstelle für SAT-Solver bezüglich ihre Performance verglichen.

Abschließend wird in Kapitel 7 die Arbeit zusammengefasst, ein Fazit in Abschnitt 7.2 gezogen und ein Ausblick für zukünftige Arbeiten in Abschnitt 7.3 eröffnet.

# 2 Grundlagen

Diese Arbeit beschäftigt sich mit dem automatischen Lösen von Logikrätseln unter Verwendung des SMT-Solvers Yices. Dafür wird in diesem Kapitel auf die Grundlagen für Logikrätsel und das SMT-Problem eingegangen. Im ersten Abschnitt wird der Begriff der formalen Logik, der Syllogistik erklärt. Darauf folgt in Abschnitt 2.2 eine Einführung in die Welt der Logikrätsel anhand einiger Beispiele.
Ein Bestandteil des SMT-Problems ist das Erfüllbarkeitsproblem, welches auf der Aussagenlogik basiert. Die Aussagenlogik wird in Abschnitt 2.3 vorgestellt. Dafür wird zum einen die Syntax und zum anderen die Semantik dieser Logik definiert, bevor im vierten Abschnitt dieses Kapitels auf das Erfüllbarkeitsproblem eingegangen wird. Dazu wird die Vorgehensweise des DPLL-Algorithmus (2.4.2) beschrieben.
Da die Logik von Yices bzw. die verwendete Rätsellogik auf der Prädikatenlogik erster Stufe basiert, wird diese in Abschnitt 2.5 eingeführt. Auch zu dieser Logik werden die Syntax und die Semantik definiert.
Im letzten Abschnitt dieses Kapitels geht es schließlich um die allgemeine Struktur und Funktionsweise von SMT-Solvern. Es werden sowohl direkte als auch indirekte SMT-Solver und verschiedene Theorie-Solver vorgestellt. Außerdem wird eine Methode zur Kombination von verschiedenen Hintergrundtheorien erklärt, nämlich die Nelson-Oppen-Methode.

## 2.1 Logik

Im vierten Jahrhundert vor Christus begründete der griechische Philosoph Aristoteles die Syllogistik ([WIK15]), die heute als formale Logik bekannt ist. Die formale Logik ist ein Teilgebiet der Philosophie, der Mathematik und der Informatik. Sie befasst sich mit der Lehre der logischen Schlussfolgerung. Dabei werden aus Behauptungen bestimmter Prämissen, dem Ober- und dem Untersatz, einer Aussage Konsequenzen deduziert.
Aristoteles formulierte dafür folgendes Beispiel:

| | |
|---|---|
| Erste Prämisse (Obersatz): | „Alle Menschen sind sterblich.“ |
| Zweite Prämisse (Untersatz): | „Alle Griechen sind Menschen.“ |
| Schlussfolgerung: | „Alle Griechen sind sterblich.“ |

Die klassische Logik ist ein Teilgebiet der formalen Logik und beschäftigt sich mit Aussagen. Eine Aussage ist ein Satz, dem sich Eigenschaften zuschreiben lassen und der als wahr oder falsch beurteilt werden kann. Sie muss die folgenden beiden Bedingungen erfüllen:

- Prinzip der Zweiwertigkeit: Jede Aussage kann genau einen von zwei Wahrheitswerten annehmen.
- Kompositionalitätsprinzip: Der Wahrheitswert, von durch logische Junktoren zusammengesetzten Aussagen, lässt sich durch die Komposition und die Art der Zusammenstellung ihrer Teilaussagen eindeutig bestimmen.

Ein Junktor ist ein logischer Operator, der als Verknüpfung zwischen atomaren Aussagen innerhalb der Logik dient. Die Anzahl, der durch einen Junktor verbundenen atomaren Aussagen, definiert die Stelligkeit des Junktors. Zu den Junktoren der klassischen Logik gehören die Negation ($\neg$), die Konjunktion ($\wedge$), die Disjunktion ($\vee$), die Implikation ($\Rightarrow$) und die Äquivalenz ($\Leftrightarrow$).
Die drei wichtigsten Teilgebiete der klassischen Logik sind die Aussagenlogik, die Prädikatenlogik erster Stufe und die Prädikatenlogik höherer Stufe, wobei die ersten beiden Logiken in den Abschnitten 2.3 bzw. 2.5 dieses Kapitels erläutert werden.

## 2.2 Logikrätsel

Logikrätsel oder auch Logeleien[1] gehören zur Logikrätselgattung, was bedeutet, dass sie durch das Deduzieren gelöst werden können. Solche Rätsel bestehen aus einer Beschreibung der Aufgabenstellung, in welcher Gruppen mit gleich vielen Elementen vorgegeben werden, sowie einer Reihe von Hinweisen, die entweder direkt oder indirekt Aussagen darüber enthalten, welche der in der Beschreibung genannten Elemente zusammengehören und welche nicht. Der Rätselnde soll nun jedem Element einer Gruppe widerspruchsfrei genau ein Element jeder anderen Gruppe zuordnen.
Außerdem zählen zu den Logeleien auch mathematische Denkaufgaben und Zahlenrätsel wie Sudokus, Kakuros oder magische Quadrate.

**Beispiel 2.2.1:** *Magisches Quadrat*
*Magische Quadrate* gibt es in der Größe $n \times n$, wobei n die Kantenlänge ist. Es ist eine quadratische Anordnung der Zahlen 1, 2, ..., $n^2$, sodass die Summe der Zahlen aller Zeilen, Spalten und der beiden Diagonalen immer gleich ist. Diese Summe wird als die *magische Zahl* des magischen Quadrates bezeichnet.

Wir betrachten ein magisches Quadrat der Größe 5×5. Bei einer solcher Größe ist die Summer jeder Zeile, Spalte und Diagonale gleich 65.

| | | | | |
|---|---|---|---|---|
| | 24 | | 8 | 15 |
| | 5 | 7 | 14 | |
| 4 | 6 | | 20 | |
| 10 | | | | 3 |
| 11 | 18 | 25 | | |

Tabelle 2.1: Magisches Quadrat

Der Knobler muss nun herausfinden, welche Zahlen in die leeren Felder gehören.
Auflösung in Kapitel 6.1.5

Diese Arbeit bezieht sich neben zahlenbasierenden Rätsel hauptsächlich auf prädikatenlogische Probleme. Ein solches Problem ist das von Aristoteles in Abschnitt 2.1 beschriebende Beispiel. Dieses ist in Beispiel 2.2.2 etwas modifiziert worden.

[1] https://de.wikipedia.org/wiki/Logical

**Beispiel 2.2.2:** *Ist Aristoteles sterblich?*

| |
|---|
| „Alle Menschen sind sterblich.“<br>„Aristoteles, Sokrates und Platon sind Menschen.“ |
| Frage: „Sind Aristoteles, Sokrates und Platon sterblich? “ |

Es ist offensichtlich, dass Aristoteles, Sokratis und Platon sterblich sind. Doch was ist mit folgendem Rätsel:

**Beispiel 2.2.3:** *Auf der Suche nach Oona*

| |
|---|
| Im Südpazifik gibt es einige Inseln auf denen nur Ritter und Schurken wohnen. Ritter haben die Eigenschaft, dass sie immer die Wahrheit sagen, wohingegen Schurken immer lügen. Einige der Bewohner sind zusätzlich halb Mensch, halb Vogel. Eine von diesen ist Oona. Sie fliegt gerne von Insel zu Insel und sagt ihrem Mann, der sich selbst als Logiker bezeichnet, nie, auf welche Insel sie fliegt.<br>Eines Tages ist sie mal wieder weggeflogen und ihr Mann macht sich auf die Suche. Er fährt mit seinem Kanu und fragt die Einwohner der Inseln, ob sie Oona gesehen hätten. Als er auf eine Insel mit fünf Bewohner kam, bekam er von ihnen folgende Antworten: |
| A: Oona ist auf dieser Insel.<br>B: Oona ist nicht auf dieser Insel.<br>C: Oona war gestern hier.<br>D: Oona ist heute nicht hier und sie war auch gestern nicht hier.<br>E: Entweder ist D ein Schurke oder C ist ein Ritter.<br><br>Der Logiker dachte eine Weile darüber nach, konnte aber nichts damit anfangen und fragt, ob einer der Bewohner noch eine Aussage machen könnte. Daraufhin erwidert A: Entweder ist E ein Schurke oder C ist ein Ritter.<br><br>Ist Oona auf der Insel? |

Tabelle 2.2: Auf der Suche nach Oona

Bei einem solchen Rätsel kann man die Lösung nicht sofort schlussfolgern.
In dieser Masterarbeit werden sowohl einfache, wie das Aristoteles-Rätsel, als auch schwierige Logeleien betrachtet. Dies geschieht in Kapitel 6.

## 2.3 Aussagenlogik

Als Grundlage der in Kapitel 4 definierten Rätsellogik werden in diesem Abschnitt die syntaktischen und semantischen Grundlagen der Aussagen- sowie im nächsten die der Prädikatenlogik erster Stufe definiert.
Die Aussagenlogik[1] wurde im Jahr 1847 von George Boole entwickelt und beschäftigt sich mit Aussagen und der Kombination mehrerer Aussagen mittels der logischen Operatoren (Junktoren). Die Semantik zusammengesetzter Aussagen lässt sich auf die Interpretation der atomaren Aussagen zurückführen. Die Aussagen in der Sprache der Aussagenlogik bezeichnet man als aussagenlogische Formeln. Diese bestehen aus mehreren Atomen, die für einfache Aussagen stehen und immer entweder wahr oder falsch sind. Der Wahrheitswert einer Aussage ist somit ausschließlich abhängig von den Wahrheitswerten ihrer Teilaussagen.

### 2.3.1 Syntax der Aussagenlogik

Die Syntax der Aussagenlogik besteht aus folgenden drei Komponenten:

- den Atomen (Aussagenvariablen) *0, 1, p, q, r...*
- den Junktoren $\neg$, $\wedge$, $\vee$, $\oplus$, $\Rightarrow$, $\Leftrightarrow$
- und den Hilfszeichen (, )

#### Grammatik der aussagenlogischer Formeln

**Definition 2.3.1** (Grammatik der aussagenlogischer Formeln AL): *Der Aussagenkalkül ist durch folgende Grammatik bestimmt, wobei A ein Nichtterminal für Atome ist und* $\mathrm{F}_1$, $\mathrm{F}_2$, $\mathrm{F}_3$,*... aussagenlogische Formeln repräsentieren:*

$$F ::= A \mid \neg\ F \mid \mathrm{F}_1 \wedge \mathrm{F}_2 \mid \mathrm{F}_1 \vee \mathrm{F}_2 \mid \mathrm{F}_1 \Rightarrow \mathrm{F}_2 \mid \mathrm{F}_1 \Leftrightarrow \mathrm{F}_2 \mid \mathrm{F}_1 \otimes \mathrm{F}_2$$

Aussagenlogische Formeln werden häufig auch als Ausdrücke bezeichnet. Jeder Ausdruck kann eindeutig zerlegt werden: Er ist entweder ein Atom, eine Negation, eine Konjunktion, eine Disjunktion, eine Implikation oder eine Äquivalenz. Seine Bestandteile nennt man Teilformeln.

#### Junktoren

In der folgenden Tabelle 2.1 sind die Junktoren aufgezählt, die benutzt werden, um Aussagen in der Aussagenlogik miteinander zu verknüpfen:

[1] Vgl. [Let13] und [Wag15a] für den gesamten Abschnitt 2.3

| **Bezeichnung** | *Bedeutung* | **Symbol** |
|---|---|---|
| Negation | *nicht* | $\neg$ |
| Konjunktion | *und* | $\wedge$ |
| Disjunktion | *oder* | $\vee$ |
| Kontravalenz | *entweder-oder* | $\otimes$ |
| Implikation | *wenn ..., dann* | $\Rightarrow$ |
| Äquivalenz | *genau dann, wenn* | $\Leftrightarrow$ |

Tabelle 2.3: Konnektoren der Aussagenlogik

In Tabelle 2.2 sind die Wahrheitswerte der einzelnen Junktoren in einer Wahrheitstafel abgebildet.

| | | Negation | Konjunktion | Disjunktion | Antivalenz | Implikation | Äquivalenz |
|---|---|---|---|---|---|---|---|
| p | q | $\neg$p | p $\wedge$ q | p $\vee$ q | p $\oplus$ q | p $\rightarrow$ q | p $\leftrightarrow$ q |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |

Tabelle 2.4: Wahrheitstafel

Die **Negation** invertiert den Wert einer Variable. So wird durch die Negation aus einer 1 eine 0 und umgekehrt.
Die **Konjunktion** erwartet zwei Eingaben p und q und liefert genau dann eine 1, wenn beide Eingaben eine 1 beinhalten und in allen anderen Fällen eine 0.
Für zwei Eingaben p und q liefert die **Disjunktion** eine 1, wenn mindestens eine der beiden Eingaben eine 1 ist und sonst eine 0.
Die **Antivalenz** ist ähnlich der Disjunktion, liefert aber nur dann eine 1, wenn genau eine der beiden Eingaben eine 1 enthält.
Die **Implikation** ergibt immer eine 1, es sei denn p = 1 und q = 0 .
Bei der Äquivalenz müssen beide Variablen dieselbe Eingabe haben, um eine 1 zu liefern.

Für die in Tabelle 2.2 genannten Junktoren gelten die folgenden Bindungsregeln:
Die Negation $\neg$ bindet am stärksten. Die Konjunktion $\wedge$ bindet stärker als die Disjunktion $\vee$. Sowohl die Antivalenz als auch die Implikation und die Äquivalenz binden gleich stark, aber schwächer als die vorherigen genannten Junktoren.

### Länge aussagenlogischer Formeln

In einer Formel F kann eine endliche Anzahl an Atomen auftreten. Die Menge dieser Atome bezeichnet man mit *atoms*(F).
Ein Literal ist ein Atom oder ein negiertes Atom ($\neg$A). Ein Atom A heißt auch positives

Literal und ¬A negatives Literal. *literals*($A$) steht für die Menge der Literale über einer Atommenge $A$, *literals*(F) für die Menge der Literale über den Atomen einer Formel F.

Es gibt zwei Möglichlichkeiten die Länge einer Formel F zu definieren:

**Definition 2.3.2** (Länge einer aussagenlogischen Formel): *Die Länge |F| einer aussagenlogischen Formel F wird induktiv definiert durch:*

1. Für jedes Atom A gilt |A|:=1.
2. Für jede Formel F gilt |(¬F)|:=|F|.
3. Für Formeln $F_1$ und $F_2$ gilt $|(F_1 \vee F_2)|:= |(F_1 \wedge F_2)|:=|F_1| + |F_2|$

**Beispiel 2.3.1**:
Die Formeln ¬((A ∨ ¬B)(¬A ∨ ¬B ∧ ¬¬C)) und A ∧ B ∧ C ∧ D ∧ E haben beide die Länge 5. Das Längenmaß nimmt also wenig Rücksicht auf die Struktur einer Formel.

*Eine zweite mögliche Definition der Formellänge zählt die zur Bildung der Formel verwendeten Zeichen, d.h. Klammern, Operatorzeichen und Zeichen für die Namen der Atome.*

Wenn Formeln zu lang werden, kann man sie durch *Substitution* kürzen und in die konjunktive Normalform bringen.
Die *Substitution* ist ein Teil des DPLL-Algorithmus, der in Kapitel 2.5.2 besprochen wird.

### 2.3.2 Semantik der Aussagenlogik

Die Semantik legt in der Aussagenlogik fest, wie die Formeln zu ihren Wahrheitswerten ausgewertet werden. Dafür werden nun die Auswertungen der möglichen Interpretationen bzw. Bewertungen betrachtet.
Atome stehen für elementare Aussagen, für die ein Wahrheitswert zugeordnet werden kann. Unter Berücksichtigung der logischen Operatoren kann der Wahrheitswert einer Formel bezüglich dieser atomaren Bewertung bestimmt werden.

**Definition 2.3.3** (Bewertung von Atomen). *Eine Bewertung oder Interpretation I ist eine Variablenbelegung, die jede aussagenlogische Formel auf ein Element der Menge {0, 1} abbildet.*

$$I:\{ A_0,A_1,A_2,... \} \rightarrow \{0,1\}$$

Eine Bewertung ist eine Zuordnung der Wahrheitswerte an alle Atome. Eine solche Bewertung lässt sich auf eine Bewertung aussagenlogischer Formeln erweitern.

**Definition 2.3.4** (Bewertung aussagenlogischer Formeln). *Sei I eine Bewertung der Atome. Diese wird zu einer Bewertung für aussagenlogische Formeln erweitert:*

$$\mathrm{I}:\{\mathrm{F}|\mathrm{F}\text{ aussagenlogische Formel}\} \to \{0,1\}$$

*Und zwar durch folgende Regeln:*

$$\mathrm{I}(\neg \mathrm{F}) = \begin{cases} 1 & \text{falls } I(F) = 0 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \wedge \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = 1 \text{ und } I(\mathrm{F}_2) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \vee \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = 1 \text{ oder } I(\mathrm{F}_2) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \Rightarrow \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = 0 \text{ oder } (I(\mathrm{F}_1) = 1 \text{ und } I(\mathrm{F}_2) = 1) \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \Leftrightarrow \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = I(\mathrm{F}_2) \\ 0 & \text{sonst} \end{cases}$$

Aus der Bewertung der Atome einer Formel wird also nach der obigen Vorschrift der Wahrheitswert der Formel unter dieser Bewertung berechnet. Dazu werden auch nur die in der Formel vorkommenden Atome zur Bestimmung ihres Wahrheitswertes benötigt. Für eine Formel F müssen zwei Bewertungen, die auf *atoms*(F) gleich sind, auch den gleichen Wahrheitswert liefern.

**Lemma 2.1** (Koinzidenzlemma). *Seien* $\mathrm{I}_1$ *und* $\mathrm{I}_2$ *Bewertungen und F eine aussagenlogische Formel. Dann gilt:*

$$(\forall \mathrm{A} \in \mathrm{atoms(F)}\colon \mathrm{I}_1(\mathrm{A}) = \mathrm{I}_2(\mathrm{A})) \Rightarrow \mathrm{I}_1(\mathrm{F}) = \mathrm{I}_2(\mathrm{F})\ )$$

Sei nun beispielsweise I(A) = 0, I(B) = 1 und I(C) = 1 gegeben, so ergibt sich I($\neg$A $\wedge$ (B $\vee$ C)) = 1. Da die Menge atoms(F) einer Formel F endlich ist, lassen sich in einer Wahrheitstabelle alle möglichen Bewertungen für atoms(F) systematisch aufzählen.

**Beispiel 2.3.2:** Eine Wahrheitstabelle für die Formel F = $\neg$A $\wedge$ (B $\vee$ C):
Die oben genannte Bewertung I(A) = 0, I(B) = 1 und I(C) = 1 entspricht der vierten Zeile und liefert I(F) = 1 als Ergebnis.

**Definition 2.3.5** (Modell). *Wenn I(F) = True gilt, so nennt man I ein Modell für die Formel F.*

**Definition 2.3.6.** *Eine aussagenlogische Formel F ist*

| A | B | C | ¬A | B ∨ C | ¬A ∧ (B ∨ C) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |

Tabelle 2.5: Wahrheitstabelle

- allgemeingültig genau dann, wenn für alle Interpretationen I gilt: I(F) = True.
- erfüllbar genau dann, wenn es eine Interpretation I gibt mit: I(F) = True.
- falsifizierbar genau dann, wenn es eine Interpretation I gibt mit: I(F) = False.
- widersprüchlich genau dann, wenn für alle Interpretationen I gilt: I(F) = False.

Allgemeingültige Formeln nennt man Tautologien, widersprüchliche Formeln werden Kontradiktionen genannt.

**Lemma 2.2** *Sei F eine Formel, dann gilt:*
*F ist widersprüchlich* $\Leftrightarrow$ *F ist nicht erfüllbar*

**Satz 2.1** (Endlichkeitssatz). *Sei M eine unendliche Menge von Formeln. Dann ist M erfüllbar genau dann, wenn jede endliche Teilmenge von M erfüllbar ist.*

Es wird häufig die negierte Form dieser Äquivalenz verwendet: denn es genügt eine endliche widerspruchsvolle Teilmenge von M zu bestimmen, um zu zeigen, dass M widerspruchsvoll ist.

**Definition 2.3.7** (Semantischer Folgerungsbegriff). *Seien* $F_1$ *und* $F_2$ *Formeln, dann folgt* $F_2$ *semantisch aus* $F_1$ *(*$F_1 \vDash F_2$*) genau dann, wenn für alle Bewertungen I gilt:*

$$I(F_1)=1 \Rightarrow I(F_2)=1$$

Eine Formel folgt also semantisch aus einer anderen, wenn diese für jede Bewertung wahr ist, für die auch jene wahr ist.
Die Folgerung $F_1 \vDash F_2$ kann man überprüfen, indem man $(F_1 \wedge \neg F_2)$ auf Widerspruch testet.

**Satz 2.2** (Deduktionstheorem für $\vDash$). *Seien* $F_1$ *und* $F_2$ *aussagenlogische Formeln und M eine Menge von Formeln, dann gilt:*

$$M \cup (F_1) \models F_2 \Rightarrow M \models (F_1 \rightarrow F_2)$$

**Satz 2.3** (Interpolationstheorem). *Seien* $F_1$ *und* $F_2$ *Formeln, dann ist* $F_1$ *widerspruchsvoll oder* $F_2$ *eine Tautologie oder es gilt:*
$F_1 \models F_2 \Rightarrow$ *Es gibt eine Formel* $F_3$ *(Interpolante) mit atoms(*$F_3$*)*$\subseteq$*atoms(*$F_1$*)*$\cap$*atoms(*$F_2$*) und* $F_1 \models F_2$ *und* $F_3 \models F_2$.

**Definition 2.3.8** (Logische Äquivalenz). *Zwei Formeln* $F_1$ *und* $F_2$ *heißen logisch äquivalent, abgekürzt* $\approx$*, genau dann, wenn für jede Bewertung I gilt: I(*$F_1$*)=I(*$F_2$*), d.h., wenn gilt* $\models F_1 \leftrightarrow F_2$

Vererbungsregeln für diesen Äquivalenzbegriff:

| | |
|---|---|
| Vererbung | $F_1 \approx F_2 \Rightarrow \neg F_1 \approx \neg F_2$<br>$F_1 \approx F_2 \Rightarrow F_3 \vee F_1 \approx F_3 \vee F_2$ |
| Negation | $\neg\neg F_1 \approx F_1$ |
| Idempotenz | $F_1 \vee F_1 \approx F_1$<br>$F_1 \wedge F_1 \approx F_1$ |
| Kommutativität | $F_1 \vee F_2 \approx F_2 \vee F_1$<br>$F_1 \wedge F_2 \approx F_2 \wedge F_1$ |
| Assoziativität | $(F_1 \wedge F_2) \wedge F_3 \approx F_1 \wedge (F_2 \wedge F_3)$<br>$(F_1 \vee F_2) \vee F_3 \approx F_1 \vee (F_2 \vee F_3)$ |
| Distributivität | $(F_1 \wedge F_2) \vee F_3 \approx (F_1 \vee F_3) \wedge (F_2 \vee F_3)$<br>$(F_1 \vee F_2) \wedge F_3 \approx (F_1 \wedge F_3) \vee (F_2 \wedge F_3)$ |
| De Morgan | $\neg(F_1 \wedge F_2) \approx \neg F_1 \vee \neg F_2$<br>$\neg(F_1 \vee F_2) \approx \neg F_1 \wedge \neg F_2$ |

Tabelle 2.6: Vererbungsregeln

Die Anwendung dieser Umformungsgesetze erfolgt in der Regel lokal innerhalb der Formel. Die Zuverlässigkeit solcher Anwendungen formuliert das folgende Lemma.

**Lemma 2.3** *Sei* $F_1$ *eine aussagenlogische Formel,* $F_3$ *eine Teilformel von* $F_1$ *und* $F_2$ *eine weitere Formel mit* $F_3 \approx F_2$*. Ist* $F_2$ *die Formel, die aus der Ersetzung eines Vorkommens von* $F_3$ *in* $F_1$ *durch* $F_2$ *entsteht, so gilt auch* $F_1 \approx F_2$.

Zu beachten ist hier, dass die Äquivalenz von Ausgangs- und Zielformel bei jeder einzelnen Ersetzung gilt. Als Folgerung von Lemma 2.3 erhält man dann

$$F_3 \approx F_2 \Rightarrow F_1 \approx F_1(F_3\ F_2)$$

**Definition 2.3.9** (Erfüllbarkeitsäquivalenz). *Zwei Formeln* $F_1$ *und* $F_2$ *heißen erfüllbarkeitsäquivalent, genau dann, wenn gilt:*

$$F_1 \text{ ist erfüllbar} \Leftrightarrow F_2 \text{ ist erfüllbar}$$

### 2.3.3 Beispielsätze

In diesem Abschnitt befinden sich einige Beispielsätze aus dem Buch „Logik-Ritter und andere Schurken“([SB91]):

*Ritter lügen nicht.*
$p$ = *Ritter lügen nicht.*
In Aussagenlogik: $\neg p$

*Ritter sagen die Wahrheit und Schurken lügen.*
$p$ = *Ritter sagen die Wahrheit.*
$q$ = *Schurken lügen.*
In Aussagenlogik: $p \wedge q$

*Wenn A ein Ritter ist, ist B auch ein Ritter.*
$p$ = *A ist ein Ritter.*
$q$ = *B ist ein Ritter.*
In Aussagenlogik: $p \rightarrow q$

*Wenn A oder B ein Schurke ist, dann ist C ein Ritter.*
$p$ = *A ist ein Schurke.*
$q$ = *B ist ein Schurke.*
$r$ = *C ist ein Ritter.*
In Aussagenlogik: $(p \vee q) \rightarrow r$

*Entweder ist Oona auf der Insel oder A ist ein Schurke und B ist ein Ritter.*
$p$ = *A ist ein Schurke.*
$q$ = *B ist ein Ritter.*
$r$ = *Oona ist auf der Insel.*
In Aussagenlogik: $r \leftrightarrow (p \wedge q)$

In nächsten Abschnitt geht es um das Erfüllbarkeitsproblem der Aussagenlogik und um Algorithmen, die dieses Problem lösen können.

## 2.4 Das Erfüllbarkeitsproblem

In diesem Kapitel geht es um das Erfüllbarkeitsproblem[1] (Satisfiability-Problem) der Aussagenlogik und um einen Algorithmus (SAT-Solver), der dieses Problem lösen kann. Das Erfüllbarkeitsproblem ist ein NP-vollständiges Problem. Dabei geht es darum, eine Belegung für die Variablen einer gegebenen aussagenlogischen Formel zu finden, die beweist, dass diese Formel erfüllbar ist. Gibt es keine solche Belegung, so ist die Formel unerfüllbar.

### 2.4.1 SAT-Solver

Die meisten SAT-Solver arbeiten auf der Repräsentation der boolschen Funktion als aussagenlogische Formel $\varphi$, wobei diese in konjunktiver Normalform (KNF) gegeben ist. In KNF besteht $\varphi$ aus der Konjunktion von Klauseln. Jede Klausel besteht wiederum aus der Disjunktion von Literalen, wobei ein Literal eine Variable oder die Negation einer Variable ist. Um eine aussagenlogische Formel in KNF zu erfüllen, muss jede Klausel und somit mindestens ein Literal in jeder Klausel den boolschen Wahrheitswert *1* annehmen.
SAT-Solver verlangen typischerweise eine Eingabe in konjunktiver Normalform. Dafür verwendet man ein standardisiertes Format: DIMACS (Center for Discrete Mathematics and Theoretical Computer Science [Uni89]).
Eigenschaften von DIMACS:

- Variablen sind natürliche Zahlen $\geq 1$
- Literale werden durch Integer bezeichnet, z.B. $A_7$=3
- Klausel ist Liste von Integern, 0 markiert Klauselende
- KNF ist Liste von Klauseln
- Kommentare im Header (c ...)
- spezielle Headerzeile (p cnf ...) gibt Anzahl verwendeter Klauseln und Variablen an

**Beispiel 2.4.1:** Die KNF

$$(\neg A \vee B \vee C) \wedge (B \vee \neg C) \wedge \neg D \wedge (A \vee D) \wedge (\neg B \vee \neg C \vee \neg D)$$

kann im DIMACS-Format so repräsentiert werden:

```
-1 2 3 0
2 -3 0
-4 0
1 4 0
-2 -3 -4 0
```

Alle modernen SAT-Solver basieren auf dem DPLL-Algorithmus.

[1] Vgl. [UJ12] für den gesamten Abschnitt 2.4

### 2.4.2 DPLL-Algorithmus

Dieses Verfahren wurde benannt nach Martin Davis, Hilary Putnam, George Logemann und Donald Loveland (Davis-Putnam-Logeman-Loveland-Algorithmus). Sie entwickelten dieses resolutions-basierte Verfahren von 1960 bis 1962 [GS03].
Es ist im wesentlichen ein Backtracking-basierter Suchalgorithmus, der die Erfüllbarkeit einer aussagenlogischen Formel, sofern sie in konjunktiver Normalform ist, entscheidet. Zusätzlich liefert DPLL ein Modell, falls die Formel erfüllbar ist und einen Beweis, falls sie unerfüllbar ist.

**DPLL-Algorithmus:**

1. **procedure** DPLL($M$)
2. **while** $M$ enthält eine Klausel der Form $\{L\}$ **do**
3. $M \leftarrow$ vereinfache($M$,$L$)
4. **if** $M = \{\}$ **then**
5. **return true**
6. **if** $\{\} \in M$ **then**
7. **return false**
8. $L \leftarrow$ wähle-Literal($M$)
9. **if** DPLL(vereinfache($M$,$L$)) **then**
10. **return true**
11. **else**
12. **return** DPLL(vereinfache($M$,$\neg L$))

Der Eingabeparameter $M$ ist eine endliche Menge von Klauseln, wobei eine Klausel für die Disjunktion (Veroderung) ihrer Literale steht. Das Ergebnis von DPLL($M$) ist **true**, falls $M$ erfüllbar ist und **false**, falls nicht.
Ein wichtiger Punkt der DPLL-Prozedur sind Klauseln der Form $\{L\}$. Solch eine Klausel, in der nur ein einziges Literal enthalten ist, nennt man **1-Klausel**. Sobald es ein Modell $I$ für $M$ gibt mit $I \vDash L$, ist jede Klauselmenge $M$, die eine 1-Klausel $\{L\}$ enthält, erfüllbar. Deshalb wird der Wahrheitswert des Literals L mit *wahr* festgelegt und $M$ dementsprechend vereinfacht.
Das heißt jede Klausel C mit L $\in$ C kann gelöscht werden und jede Klausel C mit $\neg$L $\in$ C wird zu C / $\neg$L.
Dafür ist der Aufruf vereinfache($M$,$L$) in Zeile 3 zuständig. Diese Prozedur sieht wie folgt aus:

1. **procedure** vereinfache($M$,$L$)
2. $M$' $\leftarrow M \setminus \{C \in M | L \in C\}$ Lösche alle Klauseln die $L$ enthalten
3. $M$" $\leftarrow \{C \setminus \{\neg L\} | C \in M'\}$ Lösche ¬L von den verbleibenden Klauseln
4. **return** $M$"

Es kann sein, dass vereinfache($M$,$L$) neue 1-Klauseln liefert. Im nächsten Schritt werden durch die Schleife in den Zeilen 2 bis 3 dann alle 1-Klauseln eliminiert. Die Prozedur vereinfache($M$,$L$) enthält nun keine Literale mehr. Weil jede Klausel eine endliche Menge von Literalen enthält und jeder Aufruf von *vereinfache* alle Vorkommen von mindestens einem Literal eliminiert, terminiert die Schleife.
Das Vereinfachen der Menge von Klauseln bezeichnet man als **Unit Propagation** (dt. Einheitsresolution). Der DPLL-Algorithmus versucht $M$ so lange wie möglich zu Vereinfachen, in dem die durch 1-Klauseln erzwungenen Wahrheitswerte propagiert werden. In den Zeilen 4-7 folgt nach der Einheitsresolution die Ausgabe des Ergebnisses des aktuellen Aufrufs, aber nur dann, wenn $M$ leer ist, oder wenn $M$ die leere Klausel enthält. Ist $M$ leer, so ist es erfüllbar, enthält $M$ die leere Klausel, ist es unerfüllbar.
Die Hilfsprozedur *wähle-Literal* in Zeile 8 liefert ein in $\phi$ vorkommendes Literal, welches beliebig gewählt werden kann.
Haben die Tests noch kein Ergebnis des aktuellen Aufrufs geliefert, wird dies nun durch eine rekursive Suche gelöst. Dazu wird in den letzten vier Zeilen eine Fallunterscheidung bezüglich der möglichen Wahrheitswerte eines beliebigen in $M$ vorkommenden Literals L vorgenommen. Der Aufruf DPLL(vereinfache($M$,$L$)) in Zeile 9 untersucht implizit die Klauselmenge $M \cup \{\neg L\}$ auf Erfüllbarkeit.

$M$ ist genau dann erfüllbar, wenn $M \cup \{L\}$ oder $M \cup \{\neg L\}$ erfüllbar ist, was sofort die Korrektheit des Algorithmus liefert.

Mittlerweile sind SAT-Solver in der Lage Instanzen der Größenordnung von $10^5$ Variablen und $10^6$ Klauseln lösen zu können.
Einsatzgebiete für SAT-Solver sind unter anderen die Hardware-Verifikation, Planungsprobleme in der Künstlichen Intelligenz und das Constraint-Solving.

## 2.5 Prädikatenlogik erster Stufe

Die Prädikatenlogik[1] wurde gegen Ende des 19. Jahrhunderts von Gottlob Frege und Charles Sanders Peirce entwickelt. Sie erweitert die von Boole entwickelte Aussagenlogik um Prädikate, Funktionen und Quantoren.
Die Basis dieses Kapitels bilden die Syntax und die Semantik der Prädikatenlogik erster Stufe (PL1), um die Erfüllbarkeit, Gültigkeit und Allgemeingültigkeit zu definieren.

### 2.5.1 Syntax

Die Prädikatenlogik übernimmt und erweitert die Syntax der Aussagenlogik.

**Definition 2.5.1** (Variablen und Alphabet der PL1)
Seien $V = \{v_i | i \in N\}$ die Menge aller (Individuen-)Variablen der Form $v_i$ und $\Sigma$ eine Signatur. Das Alphabet $A_\Sigma$ der Prädikatenlogik über $\Sigma$ ist definiert durch:

- die Menge aller Variablen V
- die Symbole in der Signatur $\Sigma$
- die Wahrheitswerte 1 und 0
- die Junktoren
- den Allquantor $\forall$ und Existenzquantor $\exists$
- die Klammern (,) und das Komma ,

mit $A_\Sigma = V \cup \Sigma \cup \{\forall, \exists\} \cup \{\neg, \wedge, \vee, \rightarrow, \leftrightarrow, (, )\} \cup \{,\}$

**Definition 2.5.2** (Terme der Prädikatenlogik)
Sei $\Sigma$ eine Signatur. Die Menge $T_\Sigma$ der $\Sigma$-Terme ist rekursiv über das Alphabet $A_\Sigma$ wie folgt definiert:

- Jedes Konstantensymbol $k \in \Sigma$ ist ein $\Sigma$-Term ($k \in T_\Sigma$ ). Konstantensymbole sind 0-stellige Funktionssymbole.
- Jede Variable $v_i \in V$ ist ein Term ($v_i \in T_\Sigma$).
- Ist f ein Funktionssymbol mit Stelligkeit k und sind $t_1, ..., t_k$ Terme, so ist auch $f(t_1, ..., t_k)$ ein Term.

Ein Grundterm ist ein Term, in dem keine Variable auftritt.

**Definition 2.5.3** (Formeln der Prädikatenlogik)
Sei $\Sigma$ eine Signatur. Die Menge Fo aller Formeln der Prädikatenlogik über der Signatur $\Sigma$ wird auch rekursiv über das Alphabet $A_\Sigma$ wie folgt definiert:

[1] Vgl. [Lug05] und [Wag15b] für den gesamten Abschnitt 2.5

- Sind $t_1$ und $t_2$ Terme ($t_1, t_2 \in T_\Sigma$), dann ist auch $t_1 = t_2$ eine $\Sigma$-Formel ($t_1 = t_2 \in PL1[\Sigma]$).
- Sind $t_1$,...,$t_n$ Terme ($t_1, ..., t_n \in T_\Sigma$) und $P \in \Sigma$ ein n-stelliges Prädikatensymbol, so ist auch $P(t_1,...,t_n)$ eine $\Sigma$-Formel ($P(t_1,...,t_n) \in Fo$).
- Ist p eine $\Sigma$-Formel, so ist auch $\neg p$ eine $\Sigma$-Formel (Negation).
- Sind p und q zwei $\Sigma$-Formeln, so sind es auch
  - $(p \wedge q) \in Fo$ (Konjunktion)
  - $(p \vee q) \in Fo$ (Disjunktion)
  - $(p \rightarrow q) \in Fo$ (Implikation)
  - $(p \leftrightarrow q) \in Fo$ (Äquivalenz)

  wobei $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ als Junktoren bezeichnet werden.
- Ist p eine $\Sigma$-Formel und $x \in V$, so sind es auch
  - $\forall x\ p \in Fo$ (Generalisation)
  - $\exists x\ p \in Fo$ (Partikulation)

  wobei $\forall, \exists$ als Quantoren bezeichnet werden.

Die Hierarchie der oben genannten Junktoren und Quantoren ist folgende:
$\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall, \exists$

**Definition 2.5.4** (Teilformel)
Die Formel q ist eine $\Sigma$-Teilformel der $\Sigma$-Formel p, wenn q sowohl eine $\Sigma$-Formel als auch ein Teilwort von p ist.

### Gebundene und freie Variablen

Ein Vorkommen der Variable $v_i$ ist *gebunden* in der Formel Fo, falls $v_i$ in einer Teilformel von $Fo$ vorkommt, die die Gestalt $\exists v_i.P(p_i)$ oder $\forall v_i.P(p_i)$ hat. In diesem Fall wird $v_i$ durch den Existenzquantor ($\exists$) bzw. den Allquantor ($\forall$) *gebunden*. Andernfalls ist das Vorkommen von $v_i$ *frei* in Fo.
Eine Formel ohne freies Vorkommen von Variablen heißt *geschlossen*. Geschlossene Formeln werden auch als Aussagen bezeichnet.

**Beispiel 2.5.1:**
Bei $\exists v_i.P(v_i, F(v_i, v_j))$ ist $v_i$ eine gebundene und $v_j$ eine freie Variable.

### 2.5.2 Semantik der Prädikatenlogik

Die Semantik ist die Bedeutung (Interpretation) der Formeln.
In diesem Abschnitt wird die Bedeutung der wohldefinierten prädikatenlogischen Ausdrücke im Hinblick auf Objekte, Eigenschaften und Beziehungen festgelegt. Die Semantik der Prädikatenlogik stellt eine formale Grundlage zur Bestimmung des Wahrheitswertes wohlgeformter Ausdrücke zur Verfügung.

**Definition 2.5.6** (Struktur)
Eine Σ-Struktur ist ein Paar $\mathcal{A} = (\mathcal{U}_A, \mathcal{I}_A)$, wobei:

- $\mathcal{U}_A$ eine beliebige nicht-leere Menge ist.
- $\mathcal{U}_A$ ist die Grundmenge (das Universum) der Struktur $\mathcal{A}$

Die Abbildung $\mathcal{I}_A$ ordnet

- jeder Variablen $v_i$ aus $\mathcal{I}_A$ ein Element der Grundmenge $\mathcal{U}_A$ zu.
- jedem n-stelligen Prädikatensymbol P aus $\mathcal{I}_A$ ein n-stelliges Prädikat auf $\mathcal{U}_A$ zu. Das Gleichheitssymbol ist hier definiert durch die Äquivalenzrelation auf der Struktur.
- jedem n-stelligen Funktionssymbol f aus $\mathcal{I}_A$ eine n-stellige Funktion auf $\mathcal{U}_A$ zu.

Die Abbildung $\mathcal{I}_A$ interpretiert jede Variable, jedes Prädikaten- und jedes Funktionssymbol für die $\mathcal{I}_A$ definiert ist.

**Definition 2.5.7** (Passende Struktur)
Seien p eine Fo-Formel und $\mathcal{A} = (\mathcal{U}_A, \mathcal{I}_A)$ eine Struktur.
Ist die Struktur $\mathcal{A} = (\mathcal{U}_A, \mathcal{I}_A)$ einer Formel p so definiert, dass $\mathcal{I}_A$ für alle in p vorkommenden freien Variablen, Prädikatensymbolen und Funktionssymbolen definiert ist, dann ist diese Struktur zu der Formel p *passend*.

**Definition 2.5.8** (Belegung)
Sei $\mathcal{A} = (\mathcal{U}_A, \mathcal{I}_A)$ eine Σ-Struktur. Eine Belegung in $\mathcal{A}$ ist eine partielle Funktion $\beta : V \to \mathcal{U}_A$.
$\beta$ ordnet jeder Variablen $v_i \in Dom(\beta)$ ein Element $\beta(v_i)$ aus der Grundmenge $\mathcal{U}_A$ zu.
Eine Belegung $\beta : V \to \mathcal{U}_A$ ist eine Belegung für eine Fo-Formel p, wenn $\beta$ für alle freien p vorkommenden Variablen definiert ist, d.h. wenn die Struktur $\mathcal{A}$ zu der Formel p passend ist.

**Definition 2.5.9** (Interpretation)
Eine Σ-Interpretation ist ein Paar $\mathcal{I} = (\mathcal{A}, \beta)$ bestehend aus einer Struktur $\mathcal{A} = (\mathcal{U}_A, \mathcal{I}_A)$ und einer Belegung $\beta : V \to \mathcal{U}_A$. Die Interpretation $\mathcal{I}$ ordnet jedem Σ-Term t mit V(t) ⊆ einen Wahrheitswert zu.
$\mathcal{I} = (\mathcal{A}, \beta)$ ist eine Interpretation für eine Fo-Formel p, wenn $\beta$ zu einer Formel p passend ist.

**Definition 2.5.10** (Semantik von Σ-Formeln)

I(false) = 0 und I(true) = 1

$$\mathrm{I}(P(\mathrm{t}_1,...,\mathrm{t}_\mathrm{n})) = \begin{cases} 1 & \text{falls } (I(\mathrm{t}_1),...,I(\mathrm{t}_\mathrm{n})) \in P^\mathrm{A} \\ 0 & \text{falls } (I(\mathrm{t}_1),...,I(\mathrm{t}_\mathrm{n})) \notin P^\mathrm{A} \end{cases}$$

$$\mathrm{I}(\neg \mathrm{p}) \;=\; \begin{cases} 1 & \text{falls } I(p) = 0 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(p \wedge q) = \begin{cases} 1 & \text{falls } I(p) = 1 \text{ und } I(q) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(p \vee q) = \begin{cases} 1 & \text{falls } I(p) = 1 \text{ oder } I(q) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(p \Rightarrow q) = \begin{cases} 1 & \text{falls } I(p) = 1 \text{ oder } 0 \text{ und } I(q) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(p \Leftrightarrow q) = \begin{cases} 1 & \text{falls } I(p) = I(q) \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\forall x : p) = \begin{cases} 1 & \text{falls } I[a/x](p) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\exists x : p) = \begin{cases} 1 & \text{falls } I[a/x](p) = 1 \\ 0 & \text{sonst} \end{cases}$$

Notation dabei: $I[a/x]$ entspricht $I(y)$, falls y und x ungleich sind $a$, falls y und x gleich sind.

**Definition 2.5.11:** *Modell, Erfüllbarkeit, Allgemeingültigkeit*
Sei $\mathcal{I} = (\mathcal{A}, \beta)$ eine zu p passende Σ-Interpretation. $\mathcal{I}$ erfüllt p unter der Belegung $\beta$ (oder $(\mathcal{A}, \beta)$ ist ein Modell von p), falls $\mathcal{I} \vDash$ p oder $(\mathcal{A}, \beta) \vDash$ p.

Ist p ein Satz, so hängt die Erfüllbarkeit von p unter der Interpretation $\mathcal{I} = (\mathcal{A}, \beta)$ nicht von der Belegung $\beta$, sondern nur von der Struktur $\mathcal{A}$ ab. Man schreibt dann: $\mathcal{A} \vDash p[\beta]$.

Falls jede zu p passende Σ-Interpretation $\mathcal{I} = (\mathcal{A}, \beta)$ ein Modell zu p ist, dann heißt p allgemeingültig: $\vDash$ p.

Eine Formel p mit freien Variablen $v_1, ..., v_k$ ist erfüllbar genau dann, wenn ihr existenzieller Abschluss erfüllbar ist.

Eine Formel p mit freien Variablen $v_1, ..., v_k$ ist allgmeingültig genau dann, wenn ihr universeller Abschluss allgemeingültig ist.

**Definition 2.5.12:** (Äquivalenz)
Zwei Formeln p und q sind logisch äquivalent, falls für alle zu p und q passenden Interpretationen $\mathcal{I} = (\mathcal{A}, \beta)$ gilt:

$$\mathcal{I} \vDash p \Leftrightarrow \mathcal{I} \vDash q \text{ (p} \equiv \text{q)}$$

Um zu überprüfen, ob eine prädikatenlogische Formel p erfüllbar ist, überprüft man für alle Interpretationen $\mathcal{I} = (\mathcal{A}, \beta)$, ob $\mathcal{I} \vDash$ p.

Fast jeder natürlichsprachliche deutsche Satz lässt sich mit Hilfe der Symbole, Junktoren und Variablensymbole in der Prädikatenlogik erster Stufe darstellen. Es gibt jedoch keine eindeutige Abbildung von Sätzen auf prädikatenlogische Ausdrücke.

### Beispiele für prädikatenlogische Sätze

- Aristoteles und Sokratis sind Griechen.
  $grieche(Aristoteles) \wedge grieche(Sokratis)$
- Wolfswürmer essen keine Igelwürmer.
  $\neg isst(wolfswurm, igelwurm)$
- Ritter oder Schurken lügen.
  $lügt(Ritter) \vee lügt(Schurke)$
- Wenn Lanzelot kein Ritter ist, ist Oona auf der Insel.
  $\neg ist(lanzelot, ritter) \rightarrow auf(oona, insel)$
- Bok ist genau dann ein Marsmann, wenn Cep eine Venusfrau ist.
  $ist(Bok, Marsmann) \leftrightarrow ist(Cep, Venusfrau)$
- Jeder Schurke lügt.
  $\forall X(person(X) \wedge lügt(X, schurke)$
- Manche Leute mögen Oona.
  $\exists X(person(X) \wedge mag(X, oona))$
- Niemand mag die Inselsteuern.
  $\neg \exists X mag(X, inselsteuern)$

## 2.6 Satisfiability Modulo Theories

Dieses Kapitel befasst sich mit den Grundlagen von SMT-Solvern[1] bevor es im nächsten Kapitel schließlich um den speziellen SMT-Solver Yices geht.
Die ersten SMT-Solver wurden Mitte der 1990er Jahre von Leonardo de Moura entwickelt, nachdem es große Fortschritte bei der Implementierung von SAT-Solvern gab. Sie wurden immer häufiger erfolgreich mit anderen Theorie-Solvern kombiniert, mit dem Ziel, entscheidbare Formeln der Prädikatenlogik erster Stufe auf Erfüllbarkeit zu prüfen, da es bisher nur möglich war Formeln der Aussagenlogik auf Erfüllbarkeit zu prüfen. Dabei wird eine Formel nicht hinsichtlich aller möglichen Interpretationen auf Erfüllbarkeit überprüft, sondern lediglich bezüglich einer oder mehrerer sogenannter Hintergrundtheorien. Man überprüft somit die **Erfüllbarkeit modulo Theorien** (engl. Satisfiability Modulo Theories, SMT). Die Programme, die solche Probleme lösen, werden als **SMT-Solver** bezeichnet. Sie werden beim Model Checking, dem Scheduling, der Testfall-Generierung, dem Equivalence Checking, usw. angewandt.
Zu den Hintergrundtheorien zählen u.a. die Theorien der Äquivalenz, der uninterpretierte Funktionen (UF), der lineare Arithmetik, der Differenzlogik, der Bitvektoren und der Arrays, wobei es möglich ist mehrere dieser Hintergrundtheorien miteinander zu kombinieren, wofür die Nelson-Oppen-Methode (Abschnitt 2.6.4) benutzt wird. Jede dieser Theorien hat seinen eigenen Theorie-Solver hat. Diese Solver müssen folgende Eigenschaften erfüllen:

- Dadurch, dass die Solver **inkrementell** sind, müssen sie nicht, wenn sie die Erfüllbarkeit einer dynamisch wachsenden Formel $\phi$ überprüfen, ständig neu gestartet werden, vor allem dann nicht, wenn eine Teilformel der Formel $\phi$ schon als erfüllbar bewertet wurde.

- Durch einfaches **Backtracking** können sie einen konsistenten Zustand wiederherstellen, ohne einen Neustart des Theorie-Solvers vornehmen zu müssen.

- Mit Hilfe von **Theorie-Deduktionen** können sie eine Menge von Literalen l der Formel $\phi$ erkennen, die unter dem aktuellen Modell $\mu$ konsistent bzw. inkonsistent geworden sind.

Man unterscheidet zwischen zwei Arten von SMT-Solvern: Dem direkten und dem indirekten SMT-Solver.

### 2.6.1 Direkter SMT-Solver

Direkte SMT-Solver transformieren die gegebenen Formeln in aussagenlogische Formeln, da solche Formeln vom SAT-Solver auf Erfüllbarkeit überprüft werden können. Hierbei spricht man auch von einem „eager“-SMT-Solver, also einem eifrigen Solver. Ein Vorteil dieses Ansatzes ist, dass SAT-Solver normalerweise schneller arbeiten SMT-Solver. Das Problem dabei

[1] Vgl. [Hau10] und [BFT15] für den gesamten Abschnitt 2.6

ist, dass die Übersetzung oft sehr schwierig ist und die übersetzte Formel exponentiell groß werden kann, was dazu führt, dass der „eager“-Ansatz langsamer terminiert als der „lazy“-Ansatz bei indirekten SMT-Solvern.
Arithmetische Operationen, wie die Multiplikation und die Theorie der Arrays lassen sich beispielsweise lediglich mit erheblichem Aufwand mit dem „eager“-Ansatz umsetzen, da die dabei entstandenen SAT-Probleminstanzen exponentiell mit der Wortbreite wachsen.
Trotz dieser Nachteile gibt es einige moderne SMT-Solver die diesen Ansatz benutzen, wie u.a. UCLID, Beaver und SWORD und einige Probleme, die damit gelöst werden können (n-Damen-Problem, Sudokus usw.).

### 2.6.2 Indirekter SMT-Solver

Häufiger als die direkten werden die sogenannten **indirekten SMT-Solver** verwendet, die auch als „lazy“-SMT-Solver bezeichnet werden. Sie werden u.a. von MathSAT, ICS, CVC-Lite, CVC4 und Yices verwendet.
Sie kombinieren den auf den DPLL-Algorithmus basierenden SAT-Solver mit bestehenden Theorie-Solvern. Der SAT-Solver belegt die atomaren Formeln mit boolschen Werten und beauftragt den Theorie-Solver regelmäßig, die Erfüllbarkeit der Formeln der Hintergrundtheorie zu überprüfen. Abbildung 2.1 zeigt die Arbeitsweise eines solchen indirekten SMT-Solvers für prädikatenlogische Formeln.

Als Eingabe erwartet der SMT-Solver eine prädikatenlogischen Formel $\varphi$. Diese Formel $\varphi$ wird in eine aussagenlogische Formel abstrahiert, wobei die atomaren prädikatenlogischen Formeln durch boolsche Variablen repräsentiert werden. Die entstandene aussagenlogische Formel $\varphi_{\mathrm{a}}$ wird dem SAT-Solver übergeben. Dieser überprüft sie auf ihre Erfüllbarkeit.
Wenn $\varphi_{\mathrm{a}}$ nicht erfüllbar ist, gibt der Solver das Ergebnis `UNSAT` aus. Daraus folgt, dass auch die prädikatenlogische Formel $\varphi$ unerfüllbar ist. Findet der SAT-Solver hingegen eine konsistente Belegung für $\varphi_{\mathrm{a}}$, so werden die atomaren prädikatenlogischen Formeln $\phi$ an den Theorie-Solver übergeben.
Dessen Aufgabe ist es nun für die Konjunktion der übergebenen Formeln eine konsistente Belegung entsprechend der verwendeten Hintergrundtheorie $\tau$ zu finden. Wenn er keine konsistente Belegung für $\varphi$ findet, startet er eine Konfliktanalyse und verfeinert die abstrahierte aussagenlogische Formel entsprechend und übergibt diese zunächst wiederum dem SAT-Solver.
Dieser Vorgang wird solange durchgeführt bis entweder der SAT-Solver für die Formel $\varphi$ unerfüllbar ausgibt oder der Theorie-Solver eine konsistente Belegung findet. Wenn dies der Fall ist, bildet die Vereinigungsmenge dieser Belegung zusammen mit der Belegung der atomaren aussagenlogischen Formeln aus $\varphi$ die konsistente Belegung für $\varphi$.

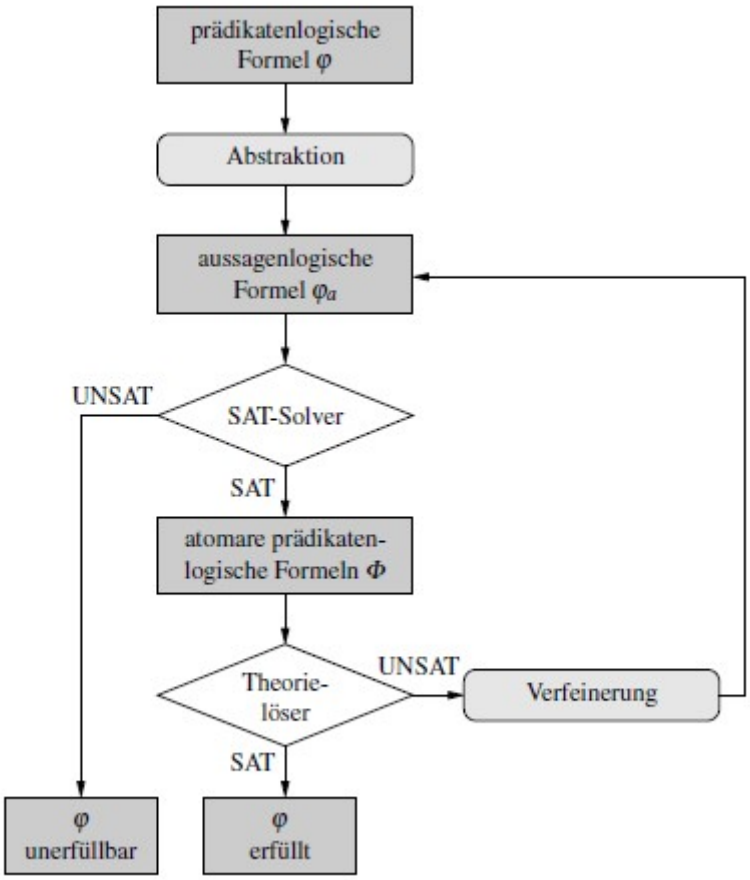



Abbildung 2.1: Indirekter SMT-Solver

Das Beispiel 2.6.1 skizziert die Vorgehensweise eines indirekten SMT-Solvers.

**Beispiel 2.6.1:**
Gegeben ist folgende prädikatenlogische Formel $\varphi$:

$$
\begin{aligned}
&\varphi := (a_1 \vee (u - w \leq 5)) \wedge \\
&(a_2 \vee (v + w \leq 6)) \wedge \\
&(a_3 \vee (z = 0)) \wedge \\
&(a_4 \vee (u + v \geq 12)) \wedge \\
&(\neg a_3 \vee \neg a_4) \wedge \\
&((x = z + 1) \vee (x = z + 3) \vee (x = z + 5) \vee (x = z + 7) \wedge \\
&((y = z + 2) \vee (y = z + 4) \vee (y = z + 6)) \wedge \\
&(u + v - 4 * x - 4 * y = 0)
\end{aligned}
$$

Der erste Schritt ist die aussagenlogische Abstraktion. Hierzu sollen die boolschen Variablen $b_i$ die atomaren prädikatenlogischen Formeln repräsentieren. Die letzte atomare Formel $\tau$

aus der folgenden Gleichung muss erfüllt sein:

$b_1 \Rightarrow (u - w \leq 5)$
$b_2 \Rightarrow (v + w \leq 6)$
$b_3 \Rightarrow (z = 0)$
$b_4 \Rightarrow (u + v \geq 12)$
$b_{51} \Rightarrow (x = z + 1)$
$b_{52} \Rightarrow (x = z + 3)$
$b_{53} \Rightarrow (x = z + 5)$
$b_{54} \Rightarrow (x = z + 7)$
$b_{61} \Rightarrow (y = z + 2)$
$b_{62} \Rightarrow (y = z + 4)$
$b_{63} \Rightarrow (y = z + 6)$
$\tau \Rightarrow (u + v - 4 * x - 4 * y = 0)$

Das Resultat ist eine aussagenlogische Formel $\varphi_a$:

$$\varphi_a = (a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge (a_3 \vee b_3) \wedge (a_4 \vee b_4) \wedge (\neg a_3 \vee \neg a_4) \wedge (b_{51} \vee b_{52} \vee b_{53} \vee b_{54}) \wedge (b_{61} \vee b_{62} \vee b_{63}) \wedge (\tau)$$

Diese Formel wird dem SAT-Solver als Eingabe übergeben, damit er die aussagenlogische Formel $\varphi_a$ auf Erfüllbarkeit testen kann. Eine mögliche Belegung $\beta$, die $\varphi_a$ erfüllt, wäre beispielsweise

$$\beta = \{\neg\alpha_1, \neg\alpha_2, \neg\alpha_3, \neg\alpha_4, b_5, b_6\}.$$

Durch Unit Propagation (siehe DPLL, Abschnitt 2.4.2) vervollständigt sich die Belegung zu:

$$\beta = \{\neg\alpha_1, b_1, \neg\alpha_2, b_2, \neg\alpha_3, b_3, \neg\alpha_4, b_4, b_4, b_5, b_6\}.$$

Dies impliziert die folgende Menge atomarer prädikatenlogischer Formeln:

$$\phi = \{(u - w \leq 5), (v + w \leq 6), (z = 0), (u + v \geq 12), (x = z + 1), (y = z + 2), (u + v - 4 * x - 4 * y = 0)\}$$

Der Theorie-Solver (in diesem Fall ein LRA-Solver [1]) überprüft die nun in konjunktiver Normalform gegebene Formel $\phi$. Da $\phi$ unerfüllbar ist, führt er eine Konfliktanalyse (Abbildung 2.2) durch. Das Resultat ist, dass die Belegung $b_1 = b_2 = b_4 := \tau$ zu einem Konflikt geführt hat. Deshalb verfeinert der LRA-Solver die aussagenlogischen Formel $\varphi_a$ mit der Klauselmenge $\neg b_1 \vee \neg b_2 \vee b_4 \vee (\neg(b_1 \wedge b_2 \wedge b_4))$.

Auch nach der Konfliktanalyse konnte der Theorie-Solver keine konsistente Belegung der Variablen der prädikatenlogischen Formeln finden. Das bedeutet, dass das Modell der aussagenlogischen Formel $\tau$-inkonsistent ist. Hätte er eine konsistente Belegung gefunden, hieße das Modell $\tau$-konsistent.

[1] Ein LRA-Solver ist ein Solver für die lineare reellwertige Arithmetik

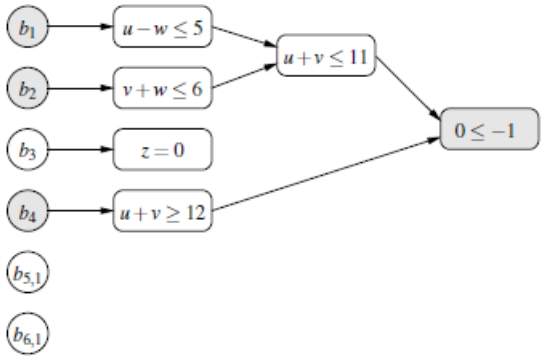



Abbildung 2.2: Konfliktanalyse des LRA-Solvers

In dem oben berechneten Beispiel werden nur Modelle der aussagenlogischen Formel auf $\tau$-Konsistenz geprüft. Eine mögliche Verbesserung des SMT-Solvers besteht darin auch partielle Belegungen auf $\tau$-Konsistenz zu testen.
Dies wird im Beispiel verdeutlicht, nachdem die Klausel $\neg b_1 \wedge \neg b_2 \wedge \neg b_4$ vom SMT-Solver deduziert wurde. In Abbildung 2.3 werden die einzelnen Entscheidungen des SAT-Solvers sowie die Interaktion des SAT-Solvers mit dem Theorie-Solver dargestellt.

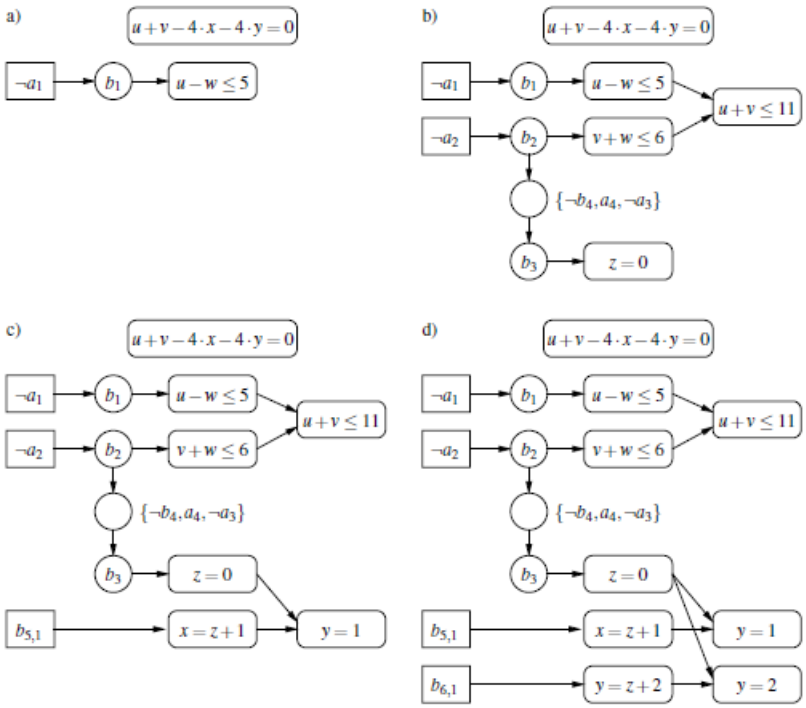



Abbildung 2.3: Schnelle Konfliktanalyse

In Schritt a) belegt der SAT-Solver die boolsche Variable $a_1$ mit `false`. Durch die Einheitsresolution (Unit Propagation) wird die Variable $b_1$ mit dem Wert $\tau$ belegt. Da $b_1 = \tau$ eine atomare prädikatenlogische Formel impliziert, wird sofort der LRA-Solver aufgerufen. Neben den Formeln $(u - w \leq 5)$ muss der LRA-Solver auch die unbedingte Formel $(u + v - 4 * x - 4 * y = 0)$ erfüllen. Da dies möglich ist, wird die Kontrolle an den SAT-Solver zurückgegeben.
In Schritt b) weist der SAT-Solver der Variablen $a_2$ den Wert `false` zu. Unit Propagation führt zur Zuweisung $b_2 = \tau$. Diese Belegung führt wiederum zu einem Aufruf des LRA-Solver, der nun zusätzlich die Formel $(u + w \leq 6)$ erfüllen muss. Da auch dies möglich ist, wird die Kontrolle an den SAT-Solver zurückgegeben. Noch im selben Schritt führen die Implikationen dazu, dass $\{\neg b_4, a_4, \neg a_3, b_3\}$ zur Belegung der boolschen Variablen hinzugefügt wird, wobei $b_3$ wiederum zu einer Implikation führt und den Aufruf des Theorie-Solvers erzwingt. Dieser bekommt als zusätzliche Formel $(z = 0)$, welche zusammen mit den bereits implizierten Formeln erfüllbar ist.
In Schritt c) wird der boolschen Variablen $b_{51}$ der Wert $\tau$ durch den SAT-Solver zugewiesen. Dies führt dazu, dass zu $\phi$ die Formel $(x = z + 1)$ hinzugefügt wird. Diese Formel vereinfacht sich mit $(z = 0)$ zu $(x = 1)$. Auch jetzt ist weiterhin die konjunktive Verknüpfung von atomaren prädikatenlogischen Formeln erfüllbar.
Im letzten Schritt weist der SAT-Solver der Variablen $b_{61}$ den Wert $\tau$ zu, was wiederum eine Implikation und damit einen Aufruf des LRA-Solvers nach sich zieht. Auch in diesem Fall ist die Konjunktion der Formeln durch den LRA-Solver erfüllbar. Da auch ebenfalls ein Modell der aussagenlogischen Formel $\varphi_a$ gefunden wurde, ist dieses auch LRA-konsistent. Dies bedeutet, dass die prädikatenlogische Formel $\varphi$ aus der Gleichung erfüllbar ist.

### 2.6.3 Theorie-Solver für Hintergrundtheorien

Es gibt eine reichhaltige Anzahl an Hintergrundtheorien und den dazugehörigen Theorie-Solvern, für die SMT-Solver. Beispiele für Hintergundtheorien sind u.a. die Theorie der *Gleichheit uninterpretierter Funktionssymbole* (EUF), *lineare Arithmetik*, *Äquivalenzlogik*, *Differenzlogik*, *Arrays*, *Listen* und *Bitvektoren*.
In diesem Abschnitt wird exemplarisch auf die Theorie der *Differenzlogik* eingegangen.

#### Theorie der Differenzlogik (IDL)

Bei der Theorie der Differenzlogik (Integer Difference Logic) wird jede Teilformel in der Form x - y $\diamond$ c dargestellt, wobei x und y Variablen sind, c eine Konstante ist und $\diamond \in (=,>,<,\geq,\leq)$.

Dieses Verfahren umfasst drei Schritte:

1. Alle Teilformeln umschreiben nach $\leq$
2. Graphen erstellen mit einem Knoten für jede Variable und pro Literal x - y $\leq$ c wird eine Kante x $\xrightarrow{c}$ y hinzugefügt.
3. Teilformeln sind erfüllbar $\Leftrightarrow$ Es gibt keinen Zyklus in dem die Summe der Kantengewichte negativ ist.

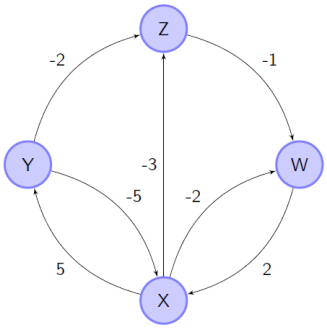
**Beispiel 2.6.3.1:** Gegeben ist folgende Formel:

$$x - y = 5 \wedge z - y \geq 2 \wedge z - x > 2 \wedge w - x = 2 \wedge z - w < 0$$

Im ersten Schritt werden alle Formeln zu $\leq$-Termen umgeformt:

$$x - y = 5 \Rightarrow x - y \leq 5 \wedge y - x \leq -5$$
$$z - y \geq 2 \Rightarrow y - z \leq -2$$
$$z - x > 2 \Rightarrow x - z \leq -3$$
$$w - x = 2 \Rightarrow w - x \leq 2 \wedge x - w \leq -2$$
$$z - w < 0 \Rightarrow z - w \leq -1$$

Im zweiten Schritt wird der Graph erstellt:



Im letzten Schritt wird überprüft, ob der erstellte Graph Zyklen enthält, deren Summe negativ ist.
Dies ist beispielsweise bei $X \to Z \to W \to Z = -3 - 2 + 1 = -4$ der Fall. Somit sind die Teilformeln unerfüllbar, woraus folgt, dass die Ursprungsformel auch unerfüllbar ist.

### 2.6.4 Kombination von Theorien

Siehe [KSB08].
Bisher wurde nur einzelne Hintergrund-Theorien betrachtet. SMT-Solver können aber auch mehrere verschiedene Theorien kombinieren. Eine Möglichkeit dazu bietet die Nelson-Oppen-Methode.
Das Nelson-Oppen-Entscheidungsvefahren ist ein Verfahren zur Lösung des Erfüllbarkeitsproblems disjunkter kombinierter Theorien. Es basiert auf der Propagierung von Gleichungen zwischen den einzelnen Theorie-Solvern. Das Verfahren kann in drei Schritten zusammengefasst werden:

1. **Formelabstraktion**:
   Mit Hilfe einer Abstraktionsfunktion wird die Formel $\sigma$ aus den kombinierten Theorien $\bigcup_i \tau$ in eine Konjunktion von puren $\tau_i$-Formeln $\sigma_1 \wedge \sigma_2 \wedge ... \wedge \sigma_n$ umgeschrieben. Jede Teil-Formel $\sigma_i$ wird einer Theorie $\tau_i$ zugeordnet.

2. $\tau_i$**-Erfüllbarkeit**:
   Im zweiten Schritt wird zu jeder der Teilformeln separat eine $\tau_i$-Erfüllbarkeitsüberprüfung durchgeführt. Sobald eine dieser Teilformeln nicht $\tau_i$-erfüllbar ist, ist die gesamte Formel und somit auch die Formel $\sigma$ in der kombinierten Theorie nicht erfüllbar.
   Wenn jedoch alle $\sigma_i$ Formeln $\tau_i$-erfüllbar sind, dann werden die bei jeder Theorie generierten Gleichungen über die gemeinsam verwendeten Variablen in die anderen Theorie-Solver propagiert.

3. **Gleichungspropagierung**:
   Wird infolge der $\tau_i$-Erfüllbarkeitsüberprüfung $\tau_i \vDash x = y$ generiert, dann wird die Gleichung $x = y$ in jede $\tau_j$-Theorie mit $i \neq j$ propagiert und dabei die Formel $\sigma_j$ durch $\sigma_j \wedge x = y$ ersetzt. Anschließend springt sie zur Theorie-Erfüllbarkeitsprüfung zurück und führt die beiden letzten Schritte rekursiv durch, bis keine neue Gleichung mehr generiert wird. Dann ist die Formel $\sigma$ aus der kombinierten Theorie $\tau$ erfüllbar.
   Wurde $\tau_j \vDash x_k = y_k$ infolge der $\tau_i$-Erfüllbarkeitsprüfung generiert, was bei nichtkonvexen Theorien oft vorkommt, dann wird eine Fallunterscheidung durchgeführt, bei der jede Gleichung $x_k = y_k$ in alle anderen Theorie-Solver propagiert wird. Ist einer der Zweige erfüllbar, dann endet das Verfahren und die Formel $\sigma$ ist erfüllbar. Sind alle Zweige nicht erfüllbar, endet das Verfahren und die Formel $\sigma$ ist unerfüllbar.

Dieses Verfahren ist korrekt bzw. vollständig, wenn die $\tau_i$-Theorien stabil unendlich bzw. konvex sind. Ist jeder der $\tau_i$-Theorie-Solver in polynomieller Zeit lösbar und jede $\tau_i$-Theorie konvex, dann ist das Verfahren in polynomieller Zeit lösbar.

**Beispiel 2.6.4.1** Nelson-Oppen-Methode:

$(x_2 \geq x_1) \wedge (x_1 - x_3 \geq x_2) \wedge (x_3 \geq 0) \wedge f(f(x_1) - f(x_2)) \neq f(x_3)$

Schritt 1: Formelabstraktion:

$(x_2 \geq x_1) \wedge (x_1 - x_3 \geq x_2) \wedge (x_3 \geq 0) \wedge f(a_1) \neq f(x_3) \wedge$
$a_1 = a_2 - a_3 \wedge$
$a_2 = f(x_1) \wedge$
$a_3 = f(x_2)$

Schritt 2: Gleichheit propagieren:

| Arithmetik | EUF |
|---|---|
| $x_2 \geq x_1$ | $f(a_1 \neq x_3$ |
| $x_1 - x_3 \geq x_2$ | $a_2 = f(x_1)$ |
| $x_3 \geq 0$ | $a_3 = f(x_2)$ |
| $a_1 = a_2 - a_3$ | |
| propagiere $x_3 = 0$ | setze $x_3 = 0$ |
| propagiere $x_1 = x_2$ | setze $x_1 = x_2$ |
| setze $a_2 = a_3$ | propagiere $a_2 = a_3$ |
| propagiere $a_1 = 0$ | setze $a_1 = 0$ |

Tabelle 2.7: Nelson-Oppen

Nachdem letzten Schritt sind $a_1$ und $x_3 = 0$, es gilt aber: $f(a_1) \neq f(x_3)$ und somit ist die Ursprungsformel unerfüllbar.

Da Nelson-Oppen keine Gleichungen liefert, braucht man einen SMT-Solver. Der Kern eines SMT-Solvers basiert auf Nelson-Oppen (z.B. der Kern von Yices).

**Beispiel 2.6.4.2** Nelson-Oppen-SMT:
Gegeben sei folgende Formel, die sich zusammensetzt aus den Theorien zur *linearen Arithmetik* und zur *Gleichheit uninterpretierter Formeln*:

$g(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$

Nach der Formelabstraktion ergibt sich:

$x_1 = g(a) = c \wedge$
$\neg x_2 = f(g(a)) \neq f(c) \wedge$
$x_3 = g(a) = d \wedge$

$x_4 = c \neq d$

Um diese Formeln zu lösen geht der Nelson-Oppen-SMT-Solver wie folgt vor:

- Sende $\{x_1, \neg x_2 \vee x_3, \neg x_4\}$ an SAT-Solver.
- SAT-Solver liefert erstes Modell: $\{x_1, \neg x_2, \neg x_4\}$.
- Theorie-Solver findet $\{x_1, \neg x_2\}$ unerfüllbar.
- Sende an SAT-Solver $\{x_1, \neg x_2 \vee x_3, \neg x_4, \neg x_1 \vee x_2\}$.
- SAT-Solver liefert Modell $\{x_1, x_2, x_3, \neg x_4\}$.
- Theorie-Solver findet $\{x_1, x_2, x_3, \neg x_4\}$ unerfüllbar.
- Sende $\{x_1, \neg x_2 \vee x_3, \neg x_4, \neg x_1 \vee x_2, \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4\}$ an SAT-Solver.
- SAT-Solver berichtet: unerfüllbar.

# 3 Yices

In diesem Kapitel werden die Logik, die Architektur und das Tool des SMT-Solvers Yices2[1] beschrieben. In Abschnitt 4.1 wird eine Einleitung über Yices2 gegeben. Im zweiten Abschnitt geht es um die Logik von Yices2. Dazu wird auf das Typsystem, die Terme und Formeln und die Theorien eingegangen. Abschnitt 4.3 beschäftigt sich mit der Architektur, wobei die Hauptkomponenten, die verschiedenen eingebauten Solver und die Kontextkonfiguration beschrieben werden. Im letzten Teil dieses Kapitels geht es um das Yices-Tool. Es wird erklärt, wie man es benutzen kann, wie man einige Quantorenprobleme lösen kann und wie die Eingabesprache von Yices aussieht.

**Bemerkung:**
Da der Code für Yices1 und Yices2 in Bezug auf die Rätsellogik (Kapitel 4) fast identisch ist, werden beide zur Lösung der Rätsel benutzt, wobei Yices1 durch eine bessere Lösung von quantifizierten Formeln mehr Rätsel lösen kann als Yices2.
Dieses Kapitel befasst sich ausschließlich mit den Komponenten von Yices2, wobei die meisten dieser Komponenten auch vom ersten Yices-SMT-Solver verwendet werden.
Wenn in diesem Kapitel von Yices die Rede ist, ist damit immer Yices2 gemeint.
Der einzige Unterschied bezüglich des für die Logikrätsel erstellten Yices-Codes ist, dass bei Yices2 das Kommando `(show-model)` am Ende des Codes stehen muss, wobei bei Yices1 der Befehl `-e` in der Konsole eingegeben werden muss, um ein Modell auszugeben.

[1] Vgl. [DM06a], [DM06b] und [Dut06] für den gesamten Abschnitt 2.3

## 3.1 Einleitung

Yices ist ein Entscheidungsverfahren, das am *Computer Science Laboratory SRI International*[1] von Leonardo de Moura und Bruno Dutertre entwickelt wurde. Es entscheidet über die Erfüllbarkeit von Formeln, die in der Prädikatenlogik erster Stufe formuliert sind.
Yices ist zum einen eine eigenständiges Werkzeug, zum anderen kann es als Library benutzt werden. Die Hauptaufgabe ist das Lesen einer oder mehrerer Formeln aus einer Datei mit Endung „.ys“ und die Überprüfung, ob diese Formeln erfüllbar sind. Sind sie erfüllbar, so kann Yices optional ein mögliches Modell ausgeben.
Yices benutzt die Nelson-Oppen-Methode um verschiedene Entscheidungsverfahren (Theorien) miteinander zu kombinieren. Zu diesen Theorien gehören:

- Uninterpretierte Funktionen: $f(f(f(x))) = x$
- Lineare Arithmetik (reeller Zahlen und Integerwerte): $x + 1 \leq y$
- Extensionale Arrays: $a$
- Bit-Vektoren mit festgelegten Größen: $concat(\mathrm{bv}_1, \mathrm{bv}_2 = \mathrm{bv}_3)$
- Quantoren: $\forall x \exists y$
- Skalartypen
- Rekursive Datentypen, Tupel, Records
- Lambda-Ausdrücke
- Abhängige Typen

Yices2 verwendet vier Tools zur Entscheidung über die Erfüllbarkeit von Formeln:

- `yices` ist der allgemeine SMT-Solver. er kann Eingaben der für Yices2 spezifizierten Sprache lesen und verarbeiten.
- `yices-smt` ist ein Solver für Eingaben die in der SMT-LIB 1.2 [2] geschrieben sind.
- `yices-smt2` ist ein Solver für Eingaben die in der SMT-LIB 2.0 ([BFT15]) geschrieben sind.
- `yices-sat` ist ein boolscher Erfüllbarkeitssolver, der Eingaben im DIMACS-Format verarbeiten kann.

[1] http://www.csl.sri.com/
[2] SMT-LIB ist eine internationale Initiative mit dem Ziel die Forschung und Entwicklung bei Satisfiability Modulo Theories zu erleichtern.

## 3.2 Logik

Yices2 besitzt die selbe getypte Logik wie Yices1 mit dem Unterschied, dass einige komplexe Typkonstrukte entfernt wurden. In diesem Abschnitt werden die eingebauten Typen, der Aufbau von Termen und Formeln und die von Yices unterstützten Theorien betrachtet.

### 3.2.1 Typsystem

Yices hat einige eingebaute Typen von primitiven Objekten: Dazu gehören die arithmetischen Typen `int` und `real`, der boolschen Typ `bool` und der Typ `(bitvector k)`, wobei k die Größe des Bitvektors angibt.
Diese Typen sind alle atomare Datentypen. Neben ihnen gibt es die *uninterpretierten Typen* und die *Skalartypen*. Ein uninterpretierter Typ bezeichnet eine nicht-leere Menge von Objekten ohne Kardinalitätsvorgaben. Ein Skalartyp bezeichnet eine nicht-leere, endliche Menge von Objekten. Die Kardinalität der Skalartypen wird definiert, wenn der Typ erstellt wird.

Zusätzlich zu den eben aufgezählten Typen liefert Yices Konstruktoren für Tupel und Funktionstypen. Die Menge aller Yices-Typen kann induktiv definiert werden:

- Jeder atomare Typ $\tau$ ist ein Typ.
- Wenn $n > 0$ und $\sigma_1$, ..., $\sigma_n$ n Typen sind, dann ist $\sigma = (\sigma_1 \times ... \times \sigma_n)$ ein Typ. Objekte des Typs $\sigma$ sind Tupel $(x_1, ..., x_n)$, wobei $x_i$ ein Objekt des Typs $\sigma_i$ ist.
- Wenn $n > 0$ und $\sigma_1$, ..., $\sigma_n$ $\tau$ Typen sind, dann ist $\sigma = (\sigma_1 \times ... \times \sigma_n \to \tau)$ ein Typ. Objekte des Typs $\sigma$ sind Funktionen der Domäne $\sigma_1 \times ... \times \sigma_n$ und der Reichweite $\tau$.

Bei ihrer Erstellung sind all diese Typen nicht-leer. Die Logik von Yices unterscheidet nicht zwischen Arrays und Funktionen und hat deshalb keinen spezifischen Typkonstruktor. Ein Array bestehend aus Integern ist beispielsweise einfach eine Funktion des Typs `int`.

Yices benutzt eine einfache Form von Untertypen (Subtypen). Sind zwei Typen $\sigma$ und $\tau$ gegeben, bedeutet $\sigma \sqsubset \tau$, dass $\sigma$ ein Subtyp von $\tau$ ist.

**Definition 3.1.** *Regeln der Beziehung der Subtypen:*

- $\tau \sqsubset \tau$ (jeder Typ ist ein Untertyp von sich selbst
- $int \sqsubset real$ (die Integerwerte formen einen Subtypen der reellen Zahlen)
- Wenn $\sigma_1 \sqsubset \tau_1, ..., \sigma_n \sqsubset \tau_n$, dann $(\sigma_1 \times ... \times \sigma_n) \sqsubset (\tau_1 \times ... \times \tau_n)$.
- Wenn $\tau \sqsubset \tau'$ dann $(\sigma_1 \times ... \times \sigma_n \to \tau) \sqsubset (\sigma_1 \times ... \times \sigma_n \to \tau')$

So ist der Typ (`int` × `int`) beispielsweise ein Subtyp von (`real` × `real`).

Die beiden Typen $\tau$ und $\tau'$ sind kompatibel, wenn sie einen gemeinsamen Obertypen (Supertyp) haben, was dann der Fall ist, wenn ein Typ $\sigma$ existiert, für den gilt: $\tau \sqsubset \sigma$ und $\tau' \sqsubset \sigma$. Das bedeutet, dass dort ein einzigartiger minimaler Supertyp um alle gemeinsamen Supertypen herum existiert. Dieser minimale Supertypen von $\tau$ und $\tau'$ heißt: $\tau \sqcup \tau'$. Daraus folgt:

$$\tau \sqsubset \sigma \text{ und } \tau' \sqsubset \sigma \Rightarrow \tau \sqcup \tau' \sqsubset \sigma.$$

Nehmen wir an, es seien zwei kompatible Typen gegeben: $\tau = (real \times int \times int)$ und $\tau' = (int \times int \times real)$. Ihr minimaler Supertyp ist $\tau \sqcup \tau' = (real \times int \times real)$. Der Typ $(real \times real \times real)$ wäre auch ein gemeinsamer Supertyp von $\tau$ und $\tau'$, aber kein minimaler.

### 3.2.2 Terme und Formeln

In Yices schließen die atomaren Terme neben den arithmetischen und Bitvektorkonstanten auch die boolschen Konstanten (`true` und `false`) mit ein.

Mit `(define-type ...)` deklariert man einen Typen. Wenn ein Skalartyp $\tau$ der Kardinalität n deklariert wird, dann sind die n unterschiedlichen Konstanten $c_1, ..., c_n$ des Typs $\tau$ auch implizit definiert. In der Yices-Syntax wird das in folgender Form gemacht:

```
(define-type tau (scalar c1 ... cn))
```

Eine äquivalente Funktionalität liefert die **API**[1] von Yices. Die API erlaubt es einen neuen Skalartypen zu erstellen und auf n Konstanten des Typs zuzugreifen.

Es ist ebenfalls möglich uninterpretierte Konstanten von beliebigen Typen zu definieren. Normalerweise werden uninterpretierte Konstanten des Typs $\tau$ wie globale Variablen behandelt, aber Yices unterscheidet zwischen Variablen des Typs $\tau$ und uninterpretierten Konstanten des Typs $\tau$. Es werden nur gebundene Variablen in Aussagen unterstützt.

Die Termkonstruktoren beinhalten alle boolschen Operatoren, also die Konjunktion, die Disjunktion, die Implikation, die Negation, die Äqui- und die Antivalenz, einen if-then-else-Konstruktor, Gleichungen, Funktionsanwendungen und Tupelkonstruktoren und Projektionen. Zusätzlich liefert Yices einen `update` Operator, welcher bei beliebigen Funktionen angewandt werden kann.
Die Notation $t :: \tau$ bedeutet: *Term t hat den Typ* $\tau$.

Es gibt keine getrennte Syntax oder Konstruktoren für die Formeln. In Yices ist eine Formel einfach ein Term eines boolschen Typs.

[1] http://yices.csl.sri.com/doc/api-types.html

Die Semantik der meisten dieser Operatoren ist Standard. Der `update`-Operator für Funktionen wird von den folgenden Axiomen charakterisiert:

$$((\text{update f } t_1...t_n \text{ v}) \ t_1...t_n) = \text{v}$$
$$u_1 \neq t_1 \vee ... \vee \Rightarrow((\text{update f } t_1...t_n \text{ v}) \ u_1...u_n) = (f u_1...u_n)$$

Anders ausgedrückt ist die Funktion ((update f $t_1$...$t_n$ v) äquivalent in allen Punkten bis auf ($t_1$,...,$t_n$). Wenn f als ein Array interpretiert wird, dann entspricht die `update`-Funktion einer Speicherung von v an Position $t_1$,...,$t_n$ im Array. Wenn man den Inhalt des Arrays betrachtet, stellt man fest, dass dort nichts außer der Funktionsanwendung ist: (f $i_1$...$i_n$) ist der Inhalt des Arrays an Position $i_1$...$i_n$.

### 3.2.3 Theorien

Zusätzlich zu den allgemeinen Operatoren gibt es die arithmetischen Operatoren und eine Vielzahl an Operatoren für Bitvektoren.

**Arithmetische Operatoren**
Zu den arithmetische Konstanten gehören beliebige Integer-Werte und rationale Zahlen. Obwohl Yices eine exakte Arithmetik benutzt, können rationale Konstanten in Fließpunkt-Notation geschrieben werden. Yices wandelt die Fließkommazahlen in rationale Zahlen um. Zum Beispiel der Ausdruck $3.04e-1$ wird in $38/125$ umgewandelt.

Die Yices-Sprache unterstützt die traditionellen arithmetischen Operatoren, wie Addition und Multiplikation mit der Ausnahme, dass keine Division einer Nicht-Konstanten erlaubt ist, um Probleme bei der Division durch Null zu vermeiden. $(5x + 3y)/4$ ist beispielsweise erlaubt, $4/(5x+3y)$ hingegen nicht. Die arithmetischen Prädikate sind die gewöhnlichen Vergleichsoperatoren, was strikte und nicht-strikte Ungleichheiten miteinschließt.

**Boolsche Operatoren:**

$$\frac{t :: \texttt{bool}}{(\texttt{not } t) :: \texttt{bool}} \quad \frac{t_1 :: \texttt{bool} \ \ t_2 :: \texttt{bool}}{(\texttt{implies } t_1 t_2) :: \texttt{bool}}$$

$$\frac{t_1 :: \texttt{bool} ... t_n :: \texttt{bool}}{(\texttt{or } t_1...t_n) :: \texttt{bool}} \quad \frac{t_1 :: \texttt{bool} ... t_n :: \texttt{bool}}{(\texttt{and } t_1...t_n) :: \texttt{bool}}$$

**Gleichheit:**

$$\frac{t_1 :: \tau_1 \ \ t_2 :: \tau_2}{(t_1 = t_2) :: \texttt{bool}}, \text{ wenn } \tau_1 \text{ und } \tau_2 \text{ kompatibel sind.}$$

**If-then-else:**

$$\frac{c :: \texttt{bool}\ \ \mathrm{t}_1 :: \tau_1\ \mathrm{t}_2 :: \tau_2}{(\texttt{ite}\ \ c\ \mathrm{t}_1 = \mathrm{t}_2) :: \tau_1 \sqcup \tau_2}, \text{ wenn } \tau_1 \text{ und } \tau_2 \text{ kompatibel sind.}$$

**Tupelkonstruktor und Projektion:**

$$\frac{\mathrm{t}_1 :: \tau_1 ... \mathrm{t}_\mathrm{n} :: \tau_\mathrm{n}}{(\texttt{tuple}\ \ \mathrm{t}_1 ... \mathrm{t}_\mathrm{n}) :: (\tau_1\ \ \times ... \times\ \ \tau_\mathrm{n})} \quad \frac{t :: (\tau_1\ \ \times ... \times\ \ \tau_\mathrm{n})}{(\texttt{select}_\mathrm{i}\ t) :: \tau_\mathrm{i}}$$

**Funktionsapplikation:**

$$\frac{f :: (\tau_1\ \ \times ... \times \tau_\mathrm{n} \to \tau)\ t_1 :: \sigma_1 ... t_\mathrm{n} :: \sigma_\mathrm{n}\ \ \sigma_1 \sqsubset \tau_1 ... \sigma_\mathrm{n} \sqsubset \tau_\mathrm{n}}{(f\ \ \mathrm{t}_1 ... \mathrm{t}_\mathrm{n}) :: \tau}$$

**Funktionsupdate:**

$$\frac{f :: (\tau_1\ \ \times ... \times \tau_\mathrm{n} \to \tau)\ t_1 :: \sigma_1 ... t_\mathrm{n} :: \sigma_\mathrm{n}\ \ v :: \sigma\ \ \sigma_\mathrm{i} \sqsubset \tau_\mathrm{i} ... \sigma \sqsubset \tau}{(\texttt{update}\ \ f\ \ \mathrm{t}_1 ... \mathrm{t}_\mathrm{n}\ \ v) :: (\tau_1\ \ \times ... \times\ \ \tau_\mathrm{n} \to \tau)}$$

Die Sprache erlaubt auch nicht-lineare Polynome. Diese werden allerdings aktuell noch nicht komplett vom Tool unterstützt. Yices kann zwar Probleme linearer Arithmetik mit reellen Zahlen und Integer-Werten lösen, was aber noch nicht einen Solver für nicht-lineare Arithmetik miteinschließt.

Seit der Version Yices 2.4 gibt es zudem einige weitere arithmetische Operatoren:

- `abs`: absolute (allgemeingültige) Werte
- `floor,ceil`: Auf- und Abrunden von Integerwerten
- `div,mod`: Division und Restrechnung von Integerwerten
- `divides,is-int`: Überprüfung der Teilbarkeit und Integralität

**Bitvektoren**
Yices unterstützt alle Bitvektoren, die in der SMT-LIB ([BFT15]) aufgelistet sind. Die am häufigsten genutzten Operatoren werden in Abbildung 3.1 aufgelistet. Sie beinhalten die Bitvektorarithmetik, logische Operatoren, wie `OR` oder `AND`, logische und arithmetische Shifts (Verschiebungen), Konkatenation und Extraktion von Subvektoren.

Die Semantik all dieser Bitvektor-Operatoren ist in der SMT-LIB definiert. Yices hält sich bis auf eine Ausnahme, nämlich bei der Division durch Null, an den Standard. In der SMT-LIB ist das Ergebnis einer Division von Null ein unspezifizierter Wert, wobei gewährleistet werden muss, dass die Divisions-Operatoren funktional sind. Anders ausgedrückt spezifiziert die SMT-LIB nicht das Ergebnis von (*bvudivab*), wenn b der Nullvektor ist, aber (*bvudivab*) und (*bvudivcb*) müssen gleich sein, sobald $a = c$ gilt. Dies gilt sogar, wenn b der Nullvektor ist. Yices benutzt dem gegenüber eine einfachere Semantik:

**Unsignierte Division:** Wenn b der Nullvektor von n Bits ist, dann ist
`(bvudiv a b) = 0b111...1`
`(bvurem a b) = a`

Allgemeinerweise ist der Quotient `(bvudiv a b)` der größte unsignierte Integerwert, der mit n Bits dargestellt werden kann. Er ist kleiner als a/b und die folgende Identität gilt für alle Bitvektoren a und b:
`a = (bvadd (bvmul (bvudiv a b) b) (bvurem)).`

**Signierte Division:** Wenn b der Nullvektor von n Bits ist, dann ergibt
`(bvsdiv a b) = 0b000...01`, wenn a negativ ist,
`(bvsdiv a b) = 0b111...1`, wenn a nicht-negativ ist,
`(bvsrem a b) = a` und
`(bvsmod a b) = a`.

| Operator und Typ | Bedeutung |
|---|---|
| `bvadd::((bv n) x (bv n) → (bv n))` | Addition |
| `bvsub::((bv n) x (bv n) → (bv n))` | Subtraktion |
| `bvmul::((bv n) x (bv n) → (bv n))` | Multiplikation |
| `bvneg::((bv n) → (bv n))` | Komplement |
| `bvudiv::((bv n) x (bv n) → (bv n))` | Quotient in unsignierter Division |
| `bvudiv::((bv n) x (bv n) → (bv n))` | Rest in unsignierter Division |
| `bvsdiv::((bv n) x (bv n) → (bv n))` | Quotient in signierter Division |
| `bvsrem::((bv n) x (bv n) → (bv n))` | Rest in signierter Division |
| `bvsmod::((bv n) x (bv n) → (bv n))` | Rest in signierter Division (abgerundet) |
| `bvule::((bv n) x (bv n) → bool` | unsigniert Kleiner-Gleich |
| `bvuge::((bv n) x (bv n) → bool` | unsigniert Größer-Gleich |
| `bvult::((bv n) x (bv n) → bool` | unsigniert Kleiner |
| `bvugt::((bv n) x (bv n) → bool` | unsigniert Größer |
| `bvsle::((bv n) x (bv n) → bool` | signiert Kleiner-Gleich |
| `bvsge::((bv n) x (bv n) → bool` | signiert Größer-Gleich |
| `bvslt::((bv n) x (bv n) → bool` | signiert Kleiner |
| `bvsgt::((bv n) x (bv n) → bool` | signiert Größer |
| `bvand:: ((bv n) x (bv n) → (bv n))` | Bitweise and |
| `bvor:: ((bv n) x (bv n) → (bv n))` | Bitweise or |
| `bvnot:: ((bv n) → (bv n))` | Bitweise Negation |
| `bvxor:: ((bv n) x (bv n) → (bv n))` | Bitweise xor |
| `bvshl:: ((bv n) x (bv n) → (bv n))` | linker Shift |
| `bvlshr:: ((bv n) x (bv n) → (bv n))` | logischer rechter Shift |
| `bvashr:: ((bv n) x (bv n) → (bv n))` | arithmetischer rechter Shift |
| `bvconcat:: ((bv n) x (bv n) → (bv n+m))` | Konkatenation |
| $\texttt{bvextract}_{i,j}$ `((bv n) → (bv m))` | extrahiere Bits von i nach j |

Abbildung 3.1: Bitvektor-Operatoren

Neben den Operationen der SMT-LIB beinhaltet Yices zwei Operatoren, die Arrays von boolschen Variablen in Arrays von Bitvektoren umwandeln können und umgekehrt.

- (bool-to-bv $b_1 ... b_n$) ist der Bitvektor, der bei der Konkatenation von n boolschen Termen $b_1, ....b_n$ erhält. Das höherwertige Bit ist $b_1$ und das niedrigerwertige ist $b_n$. Der Ausdruck

  ```
  (bool-to-bv true false false false)
  ```

  ist zum Beispiel der gleiche wie die Bitvektorkonstante 0b1000.

- (bit a i) extrahiert das i-te Bit des Bitvektors a zu einem boolschen Term. Wenn a n Bits hat, dann muss i einen Index zwischen 0 und n-1 haben. Das niedrigerwertige Bit hat den Index 0 und das höherwertige den Index n-1. Zum Beispiel gilt

  ```
  (bit (bool-to-bv false b true true) 2) = b,
  ```

  wobei b ein boolscher Term ist.

## 3.3 Architektur

Yices hat eine modulare Architektur. Man kann zwischen einer spezifische Kombination von diversen Theorie-Solvern mit der API[1] oder mit der ausführbaren Yices-Datei wählen. Mit der API können mehrere unabhängige Zusammenhänge parallel mit verschiedene Solvern und Einstellungen unterstützen werden.

### 3.3.1 Hauptkomponenten

Die Software von Yices besteht aus drei Hauptmodulen:

- **Term-Datenbank:** Yices verwaltet eine globale Datenbank, in der alle Variablen und Typen gespeichert werden. In dieser Datenbank stellt Yices eine API für das Erstellen von Variablen, Formeln und Typen bereit.
- **Kontextmanagement:** Ein Kontext ist eine zentrale Datenstruktur, die aufgestellte Formeln speichert. Jeder Kontext beinhaltet eine Menge von Aussagen, dessen Erfüllbarkeiten getestet werden sollen. Die API des Kontextmanagements unterstützt Operationen für das Erstellen und Initialisieren von Kontexten, für das Aufstellen von Formeln in einem Kontext und für das Überprüfen der Erfüllbarkeit von aufgestellten Formeln. Wahlweise kann ein Kontext auch Operationen für das Zurücknehmen von Aussagen unterstützen, wozu man den *Push/Pop-Mechanismus* benutzen muss. Einige Kontexte können unabhängig voneinander konstruiert und bearbeitet werden.
  Kontexte sind anpassbar. Jeder Kontext kann konfiguriert werden, um eine spezifische Theorie zu unterstützen und einen spezifischen Solver oder eine Kombination aus mehreren Solvern zu benutzen.
- **Modellmanagement:** Falls die Menge der aufgestellten Formeln in einem Kontext erfüllbar ist, dann kann ein Modell dieser Formeln erstellt werden. Dieses Modell verbindet die Symbole der Formeln, um die Werte zu konkretisieren. In der API stehen die Funktionen zum Erstellen und Überprüfen der Modelle.

Abbildung 3.2 zeigt die Top-Level-Architektur von Yices. Sie ist in drei Hauptmodule unterteilt. Jeder Kontext besteht aus getrennten Komponenten: Der Solver verwendet einen boolschen Erfüllbarkeitssolver und Entscheidungsprozeduren, um zu bestimmen, ob die aufgestellten Formeln erfüllbar sind. Der *Vereinfacher/Internalisierer* wandelt das Format, das von der Termdatenbank genutzt wird, in ein internes Format, das vom Solver benutzt wird. Er transformiert alle Formeln in die konjunktive Normalform, welche vom internen SAT-Solver benutzt werden.

[1] http://yices.csl.sri.com/doc/api-types.html

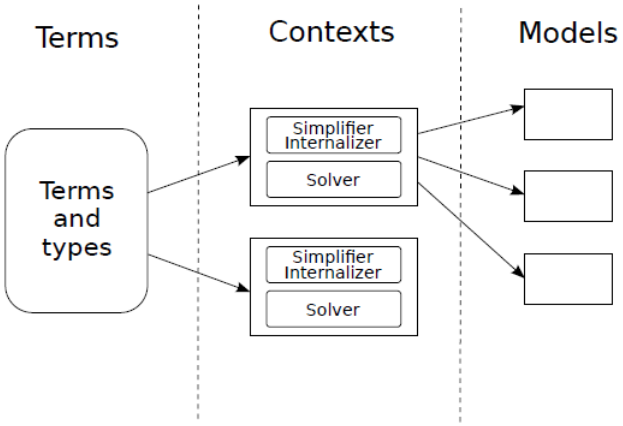



Abbildung 3.2: Top-Level-Yices2-Architektur

### 3.3.2 Solver

In Yices ist es möglich zwischen verschiedenen Solvern (oder einer Kombination aus Solvern) für das Lösen von Problemen zu wählen. Jeder Kontext kann so für eine spezifische Klasse von Formeln konfiguriert werden. Man kann zum Beispiel einen speziellen Solver für lineare Arithmetik benutzen oder einen Solver, der die komplette Yices-Sprache unterstützt. Abbildung 3.3 stellt die Architektur, der am meisten in Yices genutzten verfügbaren Solver, dar. Eine wichtige Komponente dieser Solver ist ein SAT-Solver, der auf der **Conflict-Driven-Clause Learning-Prozedur**[1] basiert. Der SAT-Solver ist verbunden mit einem oder mehreren sogenannten Theorie-Solvern. Jeder Theorie-Solver implementiert eine Entscheidungsprozedur für eine besondere Theorie. Aktuell beinhaltet Yices vier Haupt-Theorie-Solver:

- Der **UF-Solver** wird benutzt, wenn uninterpretierte Formeln auftauchen.
- Der **Arithmetik-Solver** löst Formeln, die arithmetische Operatoren enthalten.
- Der **Bitvektor-Solver** ist für Formeln zuständig, die Bitvektoren enthalten.
- Der **Array-Solver** behandelt die Hintergrundtheorie der Arrays.

Es ist möglich einige Komponenten aus Abbildung 3.3 zu entfernen, um einfachere und effizientere für Klassen und Formeln spezialisierte Solver zu erstellen. Ein Solver, der nur für die lineare Artihmetik zuständig ist, könnte beispielsweise direkt mit dem CDCL-SAT-Solver ([UJ12]) verbunden werden. Ähnlich dazu könnte Yices für reine Bitvektor-Probleme spezialisiert werden. Das gilt auch für Probleme beim Kombinieren von uninterpretierten Funktionen, Arrays und Bitvektoren.
Yices kombiniert einige Methoden der Nelson-Oppen-Methode [NO78]. Der UF-Solver koordiniert dabei verschiedene Theorie-Solver und gewährleistet globale Konsistenz. Die anderen

[1] CDCL ist eine Algorithmus, der das Erfüllbarkeitsproblem für aussagenlogische Formeln lösen kann.

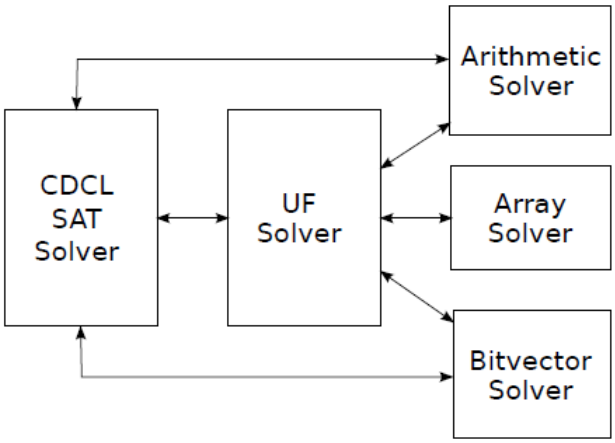



Abbildung 3.3: Die Solverkomponenten

Solver kommunizieren aussschließlich mit dem zentralen UF-Solver und nie direkt miteinander. Diese Eigenschaft vereinfacht das Design und die Implementierung von Theorie-Solvern erheblich.

### 3.3.3 Kontextkonfigurationen

Ein Kontext kann konfiguriert werden, um verschiedene Solver zu verwenden und verschiedene Szenarien zu unterstützen. Die grundlegenden Operationen auf einem Kontext schließen

- das Aufstellen einer oder mehrerer Formeln,
- das Überprüfen der Erfüllbarkeit durch die Aussagen und
- das Bauen eines Modells, falls die Aussagen erfüllbar sind,

mitein.
Durch den **Push/Pop-Mechanismus** kann ein Kontext das Hinzufügen und Entfernen von Aussagen unterstützen. In diesem Fall verwaltet der Kontext einen Stack von organisierten Aussagen in aufeinanderfolgenden Leveln. Die Push-Operation startet ein neues Level und die Pop-Operation entfernt alle Aussagen des höchstens Levels. Push macht also die Einstellungen für einen Backtracking-Punkt und Pop stellt den Kontext-Zustand zu einem vorherigen Backtracking-Punkt wieder her.
Die Unterstützung für Push und Pop verursacht einige laufzeittechnische Unkosten und kann einige Vorverarbeitungen und Vereinfachungen von Aussagen deaktivieren. In manchen Fällen ist es dann wünschenswert einen Kontext zu nutzen, der keine Push- und Pop-Operationen zur Verfügung stellt, um so eine höhere Leistung zu gewährleisten. Yices erlaubt es Nutzern die Menge der Features zu steuern, was man im spezifischen **Operating Mode** machen kann.

- Der einfachste Modus ist **one-shot**. In diesem Modus kann man Formeln aufstellen und diese sofort überprüfen. Nach dem Aufruf zum Überprüfen sind keine neuen Aussagen mehr erlaubt. Dieser Modus ist der Effizienteste. Yices kann mit ihm mächtige Vorverarbeitungen und Vereinfachungen durchführen.
- Der nächste Modus nennt sich **multi-checks**. In diesem Modus sind mehrere Aufrufe, um Operationen zu überprüfen, erlaubt. Einer kann Formeln aufstellen, „check“ aufrufen, weitere Formeln aufstellen und wieder „check“ aufrufen. Dieser Vorgang geht so lange, wie der Kontext erfüllbar ist. Sobald ein „check“ `unsat` ausgibt, können keine Aussagen mehr hinzugefügt werden. Dieser Modus vermeidet die allgemeinen Unkosten bei der Verwaltung eines Stacks von Aussagen.
- Der Standardmodus ist **push-pop**. In diesem Modus unterstützt ein Kontext die Push- und Pop-Operationen. Aussagen werden in einem Stack organisiert, wie bereits erklärt wurde.
- Der letzte Modus ist **interactive**. Dieser Modus liefert dieselben Funktionalitäten wie **push-pop**, aber der Kontext ist zum Wiederherstellen konfiguriert, wenn eine „check“-Operation unterbrochen wird oder einen Timeout hat.

## 3.4 Tool

In Yices gibt es ein Tool, das die Eingaben, die in der Yices-Sprache geschrieben sind, verarbeiten kann. Man nennt es `yices` (oder yices.exe in Windows). Im folgenden Abschnitt werden die von `yices` unterstützte Syntax und die Kommandos aufgezählt und erklärt.

**Beispiel 3.4.1**

```
(define-type BV (bitvector 32))

(define a::BV)
(define b::BV)
(define c::BV (mk-bv 32 1008832))
(define d::BV)

(assert (= a (bv-or (bv-and (mk-bv 32 255)
      (bv-not (bv-or b (bv-not c))))
    (bv-and c (bv-xor d (mk-bv 32 1023))))))

(check)

(show-model)
(eval a)
(eval b)
(eval c)
(eval d)
```

Das Beispiel 3.4.1 soll die Vorgehensweise des Tools veranschaulichen.
Die erste Zeile definiert den Typen `BV`, der für einen Bitvektor der Größe 32 steht. Danach werden vier Variablen vom Typ `BV` deklariert. Die Variablen `a`,`b` und `d` sind uninterpretierte Konstanten. Die Konstante `c` ist definiert als eine Bitvektorrepräsentation mit dem Integerwert 1008832. In der nächsten Zeile steht eine Aussage (`assert`), die einen Constraint zwischen a,b,c und d ausdrückt. Das Kommmando `(check)` überprüft, ob diese Aussage erfüllbar ist. Wenn sie es ist, dann erzeugt das Kommando `(show-model)` eine Modell. Die Kommandos `(eval ...)` fragen nach dem Wert der vier Variablen im Modell.

Nach Berechnung liefert yices folgendes Ergebnis:

```
sat
(= d 0b00000000000000000000000000000000
(= b 0b00000000000000000000000000000000
(= a 0b00000000000000000000000011000000

0b00000000000000000000000011000000
0b00000000000000000000000000000000
```

```
0b00000000000011110110010011000000
0b00000000000000000000000000000000
```

Das `(check)`-Kommando gibt uns `sat` zurück. Das heißt, die Aussagen sind erfüllbar. Danach wird das Modell in Form einer Zuweisung der drei uninterpretierten Variablen `a`,`b` und `d` ausgegeben. In den letzten vier Zeilen stehen die Bitvektorkonstanten für jedes `(eval ...)`-Kommando.

### 3.4.1 Quantorenprobleme

Yices kann teilweise Quantorenprobleme, d.h. Aussagen, die *exists/forall* enthalten, lösen. Wie der Name bereits andeutet sind solche Probleme der folgenden Form:

$$\exists x_1, ..., x_n : \forall y_1, ..., y_m : P(x_1, ..., x_n, y_1, ..., y_m)$$

In vielen Anwendungen ist es das Ziel die Werte $a_1, ..., a_n$ für quantifizierte Variablen $x_1, ..., x_n$ zu finden und zwar solche, die die Formel

$$\forall y_1, ..., y_m : P(a_1, ..., a_n, y_1, ..., y_m)$$

gültig machen.

Yices kann solche Probleme, bei denen quantifizierte Variablen entweder endliche Typen haben oder reelle Variablen sind, lösen. Der Algorithmus dazu wird beschrieben in [GSD14].

**Beispiel 3.4.2:**

```
(define x::real)
(assert (forall (y::real))
    (=> (and (< (* -1 y) 0) (< (+ -10 y) 0))
       (< (+ -7 (* -2 x) y) 0))))

(ef-solve)
(show-model)
```

Beispiel 3.4.2 stellt dar, wie Quantorenprobleme in der Yices-Sprache spezifiziert werden. Globale Deklarationen, wie die Konstante `x` entsprechen den existentiellen Variablen. Constraints werden als Aussagen der Form `(forall (y ...) P)` angegeben, wobei `y` eine universelle Variable ist. Formeln dürfen viele verschiedene Aussagen dieser Form enthalten, genauso wie quantorenfreie Constraints bei globalen Variablen.

Das Kommando `(ef-solve)` ruft den Quantoren-Solver auf. Dieses Kommando ist ähnlich zu `(check)`. Er gibt `sat` aus, falls das Problem erfüllbar ist, `unsat`, falls nicht und

`unknown`, falls der Solver nicht innerhalb einer festgelegten Anzahl von Iterationen terminiert. Falls `(ef-solve) sat` ausgibt, bekommen wir dasselbe Modell, wie mit dem Kommando `(show-model)`. Yices gibt folgendes zurück:

sat
(= x 2)

Die erste Zeile ist das Ergebnis von `(ef-solve)`. Die zweite Zeile das Modell, dass uns zeigt, welchen Wert die globale Variable `x` annimmt.

### 3.4.2 Tool starten

Das folgende Kommando startet das Yices-Tool:

yices [option] <filename>
Wenn kein <filename> angegeben wird, läuft yices im `interactive`-Modus und liest die Standardeingaben. Die folgenden Optionen werden unterstützt:

- `logic=<name>`: Auswahl einer SMT-LIB-Logik
- `arith-solver=<solver>`: Auswahl eines Arithmetik-Solvers
- `mode=<mode>`: Auswahl von Solver-Eigenschaften:
  - `one-shot`: Nach dem `(check)`-Kommando sind keine Aussagen erlaubt. In diesem Modus kann Yices die Erfüllbarkeit eines Blocks von Aussagen überprüfen und möglicherweise ein Modell erzeugen, falls alle Aussagen erfüllbar sind.
  - `multi-checks`: In diesem Modus sind mehrere `(assert)`- und `(check)`-Kommandos erlaubt.
  - `push-pop`: Wie beim `multi-checks`-Modus sind auch in diesem Modus mehrere `(assert)`- und `(check)`-Kommandos erlaubt. Zusätzlich dazu gibt es die Kommandos `(push)` und `(pop)`.
  - `interactive`: Dieser Modus unterstützt die gleichen Eigenschaften wie `push-pop` mit einem anderen Vorgehen, falls `(check)` unterbrochen wird.
- `-version, -V`: Zeigt die Version von Yices an.
- `-help, -h`: Gibt eine Zusammenfassung der Optionen zurück.
- `-verbose, -v`: Startet den Verbose-Modus.

### 3.4.3 Eingabesprache

In diesem Abschnitt des Kapitels geht es um die Syntax der Eingabesprache von Yices. Hierbei werden die lexikalischen Elemente, Deklarationen, Typen, Terme und Kommandos vorgestellt.

Zu den lexikalischen Elementen gehören Kommentare, Strings, numerische und Bitvektor-Konstanten und Symbole.

**Lexikalische Elemente**

- **Kommentare:** Starten mit „; “ und enden am Ende der Zeile. Ein Befehl für Kommentare über mehrere Zeilen ist nicht vorhanden.
- **Strings:** Sie ähneln den Strings in C und stehen immer in Anführungszeichen. Die Kürzel \n und \t erzeugen einen Zeilenumbruch bzw. einen Tab.
- **Numerische Konstanten:** Dezimalzahlen (z.B. 3 oder -3), rationale Zahlen (z.B. -1/2), Fließpunktzahlen (z.B. -1.5e+9).
- **Bitvektorkonstanten:** Diese können im Binärformat mit dem Präfix 0b oder im Hexadezimalformat mit dem Präfix 0x benutzt werden, wie z.B. 0b10101010 oder 0x33.
- **Symbole:** Ein Symbol ist ein String, der kein Keyword ist und nicht mit einer Ziffer oder einem (Leer-)Zeichen beginnt. Falls der erste Buchstabe ein arithmetischer Operator ist, muss der zweite Buchstabe eine Ziffer sein. Beispiele:
  abc, X333, +a321, b/7

**Deklarationen**
Eine Deklaration führt entweder einen neuen Typen oder einen Term ein oder gibt einem existierendem Typen oder Term einen neuen Namen. Da Yices verschiedene Namensräume für Typen und Terme benutzt, ist es erlaubt denselben Namen für einen Typen und einen Term zu verwenden.

- **Typdeklaration:** Ein Kommando der Form `(define-type name)` oder `(define-type name type)`. Die erste Form erstellt einen neuen uninterpretierten Typen mit dem Namen `name`. Die zweite Form gibt einem existierenden Typen einen neuen Namen.
- **Termdeklaration:** Ein Kommando der Form `(define name :: type)` oder `(define name :: type term)`. Die erste Form deklariert einen neuen uninterpretierten Term eines gegebenen Typen. Die zweite Form weist einen Namen einem Term zu, welcher vom Typ `type` sein muss.

**Typen**
Yices beinhaltet einige vordefinierte Typen für die Arithmetik und die Bitvektoren. Einer

davon kann die Menge der atomaren Typen erweitern, indem er uninterpretierte Typen und Skalartypen erstellt. Zusätzlich zu den atomaren Datentypen liefert Yices Konstruktoren für Tupel und Funktionstypen.

- **Vordefinierte Typen:** `int,bool,real` und `(bitvector k)`, wobei k ein positiver Integerwert ist. Eine Bitvektor-Variable b mit 32 Bits wird wie folgt definiert:
  `(define b::(bitvector 32))`
- **Uninterpretierte Typen:** `(define-type T)`
- **Skalartypen:** `(define-type S (scalar X Y Z))`
  definiert einen neuen Skalartypen S, der drei verschiedene Konstanten X,Y,Z enthält.
- **Tupeltypen:** `(define-type Pairs (tuple int int))`
- **Funktionstypen:** `(define f::(-> int bool))`

**Terme**
Yices verwendet eine mit Lisp verwandte Syntax. Das Polynom `2x+y+3z` wird beispielsweise so geschrieben:

```
(+ (* 2 x) y (* 3 z))
```

Allgemein gilt, dass alle assoziativen Operatoren endlich viele Argumente haben können:

```
and A B C
```

- **If-Then-Else:** Dieses Konstrukt kann jeden Typen annehmen. Es gibt zwei verschiedene Formen:
  `(ite c t1 t2)` und `(if c t1 t2)`
  Beide Formen sind äquivalent und bedeuten so viel wie „if c then t1 else t2“. Die Bedingung c muss ein boolscher Term sein und die beiden Terme t1 und t2 müssen kompatible Typen haben.
- **Gleichheit und Ungleichheit:** Diese werden geschrieben als `(= t1 t2)` bzw. `(/= t1 t2)`, wobei t1 und t2 wiederum zwei zueinander kompatible Typen sein müssen. Diese Operatoren sind binär.
- **Boolsche Operatoren:** Die üblichen boolschen Konstanten und Funktionen sind verfügbar. Die assoziativen und kommutativen Operatoren `or,and,xor` können endliche Argumente haben, Äquivalenz `(<=>)` und Implikation `(=>)` müssen genau zwei haben. Der Äquivalenz-Operator ist im Gegensatz zu Yices2 in Yices1 nicht enthalten.
- **Basis-Arithmetik:** Arithmetische Konstanten können in Dezimal-, Rational- oder Fließpunktnotation geschrieben werden. Intern benutzt Yices eine exakte rationale Arithmetik. Die arithmetischen Basis-Operationen sind in Abbildung 3.4 dargestellt.

| Syntax | Bedeutung | |
|---|---|---|
| (+ a1 … a_n) | Summe | $a_1 + \ldots + a_n$ |
| (* a1 … a_n) | Produkt | $a_1 \times \ldots \times a_n$ |
| (- a1 a2 … a_n) | Differenz | $a_1 - a_2 - \ldots - a_n$ |
| (^ a k) | Exponentiation | $a^k$ |
| (/ a c) | Division | a / c |
| (<= a1 a2) | Ungleichheit | $a_1 \leq a_2$ |
| (>= a1 a2) | Ungleichheit | $a_1 \geq a_2$ |
| (< a1 a2) | Strikte Ungleichheit | $a_1 < a_2$ |
| (> a1 a2) | Strikte Ungleichheit | $a_1 > a_2$ |

Abbildung 3.4: Arithmetische Operationen

- **Arithmetische Funktionen:** Andere arithmetische Operationen sind in Abbildung 3.5 aufgelistet. Die Operationen `abs`, `floor` und `ceil` haben folgende Bedeutung:
  - `abs x` ist der absolute Wert von x.
  - `floor x` ist der größte Integerwert kleiner oder gleich x.
  - `ceil x` ist der kleinste Integerwert größer oder gleich x.

  Für die Division und die Restrechnung von Integerwerten verwendet Yices die Konventionen der SMT-LIB, mit der Ausnahme, dass der Teiler `k` nicht Null sein darf und dass `div` und `mod` als reelle Zahlen deklariert werden müssen.
  Beim Teilbarkeitstest `divides k x` ist `k` eine rationale Konstante, die auch den Wert Null annehmen kann.

| Syntax | Bedeutung |
|---|---|
| (abs x) | Absoluter Wert |
| (floor x) | Abrunden |
| (ceil x) | Aufrunden |
| (div x k) | Integer-Division |
| (mod x k) | Modulo |
| (divides k x) | Teilbarkeitstest |
| (is-int x) | Integralitätstest |

Abbildung 3.5: Arithmetische Funktionen

- **Bitvektorkonstanten:** Eine Bitvektorkonstante kann in binärer oder hexadezimaler Form aufgeschrieben werden. Bei der binären Schreibweise ist die Anzahl der Bits äquivalent zur Anzahl der binären Ziffern. Die drei Terme `0b1, 0b01, 0b001` beschreiben

beispielsweise verschiedene Bitvektorkonstanten mit einem, zwei und drei Bits. In hexadezimaler Schreibweise ist die Anzahl der Bits äquivalent zur vierfachen Anzahl der hexadezimalen Ziffern.
Eine Bitvektorkonstante wird nach folgendem Ausdruck konstruiert:
`(mk-bv size value)`
In einem solchen Ausdruck müssen `size` und `value` Integerwerte sein, wobei `size` die Anzahl der Bits ($> 0$) in der Bitvektorkonstante und `value` der dezimale Wert ($\geq 0$) der Konstante ist. Beispiel:
`(mk-bv 4 15)`

- **Bitvektorarithmetik:**
  In Abbildung 3.6 werden alle arithmetischen und bitweisen Operatoren aufgelistet. Zu den arithmetischen Operatoren der Bitvektoren gehören die neben der Summe, dem Produkt, der Differenz und dem Komplement, der Negation auch die Exponentiation, wohingegen die Division nicht dazu gehört.
  Zu den bitweise Operatoren der Bitvektoren gehören die logischen Operatoren.

| Syntax | Bedeutung |
|---|---|
| `(bv-add u1 ... u_n)` | Summe |
| `(bv-mul u1 ... u_n)` | Produkt |
| `(bv-sub u1 ... u_n)` | Differenz |
| `(bv-neg u)` | Komplement |
| `(bv-pow u k)` | Exponentiation |
| `(bv-not u)` | Bitweise Komplement |
| `(bv-and u1 ... u_n)` | Bitweise and |
| `(bv-or u1 ... u_n)` | Bitweise or |
| `(bv-xor u1 ... u_n)` | Bitweise xor |
| `(bv-nand u1 ... u_n)` | Bitweise nand |
| `(bv-nor u1 ... u_n)` | Bitweise nor |
| `(bv-xnor u1 ... u_n)` | Bitweise xnor |

Abbildung 3.6: Bitvektoroperationen (arithmetisch und bitweise)

- **Bitvektor Shift und Rotate:**
  In Abbildung 3.7 sind die Shift- und Rotate-Operationen aufgelistet. Die Operationen in den ersten sieben Zeilen shiften einen Bitvektor `u` mit einer festgelegten Anzahl von `k` Bits. Die unteren drei Operatoren nehmen zwei Bitvektorargumente `u` und `v`, welche Bitvektoren derselben Größe `n` sein müssen.

| Syntax | Bedeutung |
|---|---|
| `(bv-shift-left0 u k)` | linker Shift, Padding mit 0 |
| `(bv-shift-left1 u k)` | linker Shift, Padding mit 1 |
| `(bv-shift-right0 u k)` | rechter Shift, Padding mit 0 |
| `(bv-shift-right1 u k)` | rechter Shift, Padding mit 1 |
| `(bv-ashift-right x k)` | arithmetischer Shift |
| `(bv-rotate-left x k)` | Rotation nach links |
| `(bv-rotate-right x k)` | Rotation nach rechts |
| `(bv-shl u v)` | linker Shift |
| `(bv-lshr u v)` | rechter logischer Shift |
| `(bv-ashr u v)` | arithmetischer Shift |

Abbildung 3.7: Bitvektoroperationen (Shift und Rotate)

- **Bitvektor strukturelle Operationen:**
  Mit den Operatoren aus Abbildung 3.8 können Extraktionen, Konkatenationen und andere strukturelle Operationen ausgeführt werden.

| Syntax | Bedeutung |
|---|---|
| `(bv-extract i j u)` | Subvektorextraktion |
| `(bv-concat u1 ... u_n)` | Konkatenation |
| `(bv-repeat u k)` | wdh. Konkatenation |
| `(bv-sign-extend u k)` | Zeichenerweiterung |
| `(bv-zero-extend u k)` | Nullerweiterung |
| `(bv-redor u)` | or-Reduktion |
| `(bv-redand u)` | and-Reduktion |
| `(bv-redcomp u v)` | Gleichheitsreduktion |

Abbildung 3.8: Bitvektoroperationen (strukturelle Operatoren)

- **Bitvektor Division:**
  In Abbildung 3.9 sind Divisions- und Restoperatoren aufgelistet, wobei `u` und `v` Bitvektoren derselben Größe `n` sein müssen.

- **Umwandlung zwischen Booleans und Bitvektoren:**
  Die zwei Operationen `(bool-to-bv b1 ... bn)` und `(bit u i)` wandeln boolsche Werte in Bitvektoren um und umgekehrt.
  Die boolschen Terme werden mit $b_1, ..., b_n$ dargestellt, u steht für den Bitvektor und i für das i-te Bit des Bitvektors. Die Umwandlung von `bool-to-bv true false true false` ergibt beispielsweise die Bitvektorkonstante `0b1010`. Wandelt man `(bit 0b1010 3)` in einen boolschen Wert um, wird `true` ausgegeben.

| Syntax | Bedeutung |
|---|---|
| (bv-div u v) | Quotient in unsignierter Division |
| (bv-rem u v) | Rest in unsignierter Division |
| (bv-sdiv u v) | Quotient in signierter Division |
| (bv-srem u v) | Rest in signierter Division |
| (bv-smod u v) | Rest in signierter Division (abgerundet) |

Abbildung 3.9: Bitvektoroperationen Division

- **Tupel:**
  Ein Term von Tupeln wird mit `(mk-tuple t1 ... tn)` erstellt, wobei $n \geq 1$ und `t1,...,tn` beliebige Terme sein können. Mit `(mk-tuple -1 1)` wird ein Paar von Integern konstruiert.
  Die Projektionsoperation extrahiert die i-te Komponente eines Tupels. Es wird mit `select t i` bezeichnet, wobei `t` ein Term eines Tupeltypen ist und `i` eine Integerkonstante. Falls ein Tupel `n` Komponenten hat, müssen es `i` zwischen 1 und `n` sein. Beispiele:

  ```
  (select (mk-tuple -1 1) 1) = -1
  (select (mk-tuple -1 1) 2) = 1
  ```

  Yices beinhaltet zusätzlich einen Tupel-Update-Operator. Der Ausdruck `(tuple-update t i v)` ist äquivalent zu Tupel `t` mit seiner i-ten Komponente ersetzt durch `v`. Der Typ von `v` muss ein Subtyp der i-ten Komponenten von `t` sein.

- **Funktionsupdates:**
  Ein Array oder Funktionsupdate wird durch `(update a (i1 ... in))` beschrieben. In diesem Ausdruck muss `a` ein Term mit einem Funktionstypen sein und `n` ist die Stelligkeit von `a`. Der Ausdruck konstruiert eine Funktion `b`, welche äquivalent zu `a` ist, bis auf, dass es `i1,...,in` zu `v` mappt.

- **Bitvektor Ungleichungen:**
  In Abbildung 3.10 sind die Operatoren für Ungleichungen von Bitvektoren aufgelistet. Bei allen Operationen wird ein boolscher Wert zurückgegeben.

| Syntax | Bedeutung |
|---|---|
| `(bv-ge u v)` | u ≥ v unsigniert |
| `(bv-gt u v)` | u > v unsigniert |
| `(bv-le u v)` | u ≤ v unsigniert |
| `(bv-lt u v)` | u < v unsigniert |
| `(bv-sge u v)` | u ≥ v signiert |
| `(bv-sgt u v)` | u > v signiert |
| `(bv-sle u v)` | u ≤ v signiert |
| `(bv-slt u v)` | u < v signiert |

Abbildung 3.10: Bitvektoroperationen (Vergleiche)

**Kommandos**
In Yices gibt es Kommandos für das Deklarieren von Typen und Terme, das Erstellen von Mengen von Aussagen, das Überprüfen ihrer Erfüllbarkeit und das Aufstellen von Modellen. Andere Kommandos setzen Parameter, die das Preprocessing und Heuristiken von verschiedenen Solvern steuern. In den Abbildungen 3.11-14 (am Ende dieses Kapitels) sind diese Kommandos aufgelistet.

- **Deklarationen:**
  Es gibt sowohl Typdeklarationen `(define-type name)`, `(define-type name type)`, als auch Termdeklarationen `(define name :: type)`, `(define name :: type term)`. Um eine Funktion zu deklarieren, kann man die `lambda`-Notation verwenden. Beispiel:

```
(define max::(-> real real real)
  (lambda (x::real y::real) (if (< x y) y x)))
```

  In diesem Beispiel wird die Funktion `max` definiert, die das Maximum zweier reeller Zahlen berechnet.

- **Aussagen:**
  Eine Aussage ist wie folgt aufgebaut: `(assert formula)`, wobei `formula` ein boolscher Term ist.
  Sobald eine Aussage unerfüllbar ist, wird jede andere Aussage als ein Fehler (error) behandelt.

- **Check:**
  Das `(check)`-Kommando überprüft, ob die aktuelle Menge der Aussagen erfüllbar ist und gibt `sat` oder `unsat` aus.
  Falls die Ausgabe `unknown` ist, ruft `(check)` den SMT-Solver auf, um festzulegen, ob die Aussagen erfüllbar sind.

- **Push, Pop, Reset:**
  Push/Pop werden von Yices im Modus `push-pop` und `interactive` unterstützt. In diesen Modi verwaltet der Kontext einen Stack von Aussagen, der nach aufsteigendem Level organisiert ist. Das `(push)`-Kommando beginnt mit einem neuen Aussagenlevel auf diesen Stack und `(pop)` entfernt alle Aussagen im aktuellen Level. Das Kommando `(assert f)` fügt eine Aussage `f` zum aktuellen Level hinzu. Diese Aussage ist solange Teil des Kontextes, bis das aktuelle Level entweder von einem `(pop)` oder von einem `(reset)` beendet wird. `(pop)` widerruft alle eingegangen Aussagen seit dem letzten Matching `(push)`. Das anfängliche Aussagenlevel beinhaltet alle Formeln, die vor dem ersten `(push)`-Kommando aufgestellt wurden. Solche Aussagen können von `(pop)` nicht widerrufen werden. Sie bleiben solange im Kontext bis `(reset)` aufgerufen wird.
  Die Kommandos `(reset)` und `(pop)` modifizieren die Aussagen im Kontext, wobei sie nicht die Term- und Typdeklarationen beeinflussen. Die folgende Sequenz ist gültig:

  ```
  (push)
  (define X::bool)
  (assert X)
  (check)
  (pop)
  (assert (not X))
  (check)
  ```

  Der Term `X` wird nach dem `(push)`-Kommando deklariert. Das `(pop)`-Kommando entfernt die ersten Aussagen, aber nicht die Deklaration. Folglich bleibt `X` als ein boolscher Term deklariert. Die zweite Aussage ist dann gültig. Beide `(check)`-Kommandos geben `sat` zurück.

- **Modell:**
  Das Kommando `(show-model)` stellt das aktuelle Modell dar, sofern kein Fehler vorkommt (in Yices1 wird durch die Eingabe `-e` ein Modell erzeugt). Es wird als eine Liste von Zuweisungen angezeigt, gefolgt von einer Liste von Funktionsdefinitionen.
  Eine Zuweisung hat die Form `= name value`, wobei `name` eine uninterpretierte Konstante und `value` eine Konstante ist. Dieses Format wird bei allen Termen von atomaren Typen benutzt.
  Das Kommando `(eval term)` berechnet den Wert, der dem `term` im aktuellen Modell zugewiesen wird.

  **Beispiel 3.4.3:**
  `(eval (x y))`

  **Eingabe:**
  `(define a::(-> int bool))`

```
(define b::(-> int bool))
(define c::(-> int bool))
(define x::int)
(define y::int)
(assert (and (a x) (b y)))
(assert (/= x y))
(assert (distinct a b c))
(check)
(show-model)
```

**Modell:**

```
(= y 0)
(= x -579)
(function c
 (type (-> int bool))
 (default true))
(function a
 (type (-> int bool))
 (= (a 1) false)
 (default true))
(function b
 (type (-> int bool))
 (= (b 0) true)
 (= (b 1) true)
 (default false))
```

- **Implikatoren:**
  Falls die Menge der Aussagen erfüllbar ist, kann ein Implikator erstellt werden. Der Implikator ist eine Menge von Literalen $l_1, ..., l_n$, sodass die Konjunktion $l_1 \wedge ... \wedge l_n$ erfüllbar ist und die Aussagen impliziert. Um einen solchen Implikator zu berechnen, erstellt Yices ein Modell M von Aussagen und konstruiert den Implikator von diesem Modell. Alle Literale $l_t$ sind `true` in M.
  Das Kommando, um einen Implikator anzuzeigen, ist `(show-implicant)`. Es ist nur im `one-shot`-Modus ausführbar. Der Implikator wird schließlich als ein Liste von Literalen angezeigt, jeweils eine pro Reihe.

**Beispiel 3.4.4:**
**Eingabe:**

```
(define x::int)
```

```
(define y::int)
(define z::int)
(assert (distinct x y z))
(assert (or (> x (+ z (* 2 y))) (< x (-z (* 2 y)))))
(check)
(show-implicant)
```

**Implikanten:**

```
(< (+ (* -1 y) z) 0)
(< (+ (* -1 x) (* 2 y) z) 0)
(< (+ (* -1 x) y) 0)
```

- **Exists/Forall Solver:**
  Das Kommando `(ef-solve)` überprüft die Erfüllbarkeit bei einem „exists/forall"-Problem. Dieses Kommando ist verfügbar, wenn Yices mit der Option `--mode=ef` läuft.

- **Parameter:**
  Eine Ansammlung von Parametern steuert das Preprocessing und die Vereinfachungen, die von Yices angewandt werden und die Heuristik, die vom CDCL-SAT-Solver und den Theorie-Solvern benutzt wird. Einige Kommandos erlauben es diese Parameter zu untersuchen und zu modifizieren.
  `(show-params)` zeigt die Liste aller verfügbaren Parameter an. Um spezifische Parameter zu sehen nutzt man `(show param name)`. Um einen Parameter zu bestimmen, schreibt man den Befehl `(set-param name value)`. Hilfe findet man mit `(help params)`.

- **Umwandlung ins DIMACS-Format:**
  Das Kommando `(export-to-dimcs file)` wandelt boolsche und Bitvektorprobleme ins DIMACS-Format. Yices unterstützt dieses Kommando, wenn es mit den Optionen `-logic=NONE` oder mit `-logic=QFBV` läuft.

**Beispiel 3.4.5:**
**Eingabe:**

```
(define a::(bitvector 4))
(define b::(bitvector 4))
(assert (bv-ge a b))
(export-to-dimacs "test.cnf")
(exit)
```

**Ergebnis:**

```
c Autogenerated by Yices
```

```
c
c a -> [6 7 8]
c b -> [3 4 5]
c
p cnf 10 14
1 0
-2 0
-3 9 0
-5 8 0
-5 -10 0
6 9 0
8 -10 0
-9 -6 3 0
-10 -7 4 0
-4 7 10 0
-10 9 4 0
-4 -9 10 0
-7 9 -10 0
7 10 -9 0
```

- **Timeout:**
  Standardmäßig nutzt `yices` keine Timeouts. Deshalb kann `(check)` lange zum Terminieren brauchen. Um diese Zeit zu begrenzen, kann man ein Timeout angeben:
  `(set-timeout 60)`

- **Echo:**
  Das `echo`-Kommando benutzt man, um einen String als eine Standardausgabe zu drucken.

  **Beispiel 3.4.6:**

```
(define a::bool)
(define b::bool)
(define c::bool)
(define d::bool)


(assert (= a (or b c)))
(assert (= d (and b c)))
(assert (= a d))
(echo "Diese Formeln sollten erfüllbar sein")
(check)
(show-model)
```

- **Include:**
  Mit `(include filename)` kann man Dateien in Yices einbauen. Beispiel:

`(include „example.ys")`
Dieser Befehl liest alle Kommandos dieser Datei und führt sie aus.

- **Hilfe:**
  Benötigt man Hilfe, sollte man die Befehle `(help)` oder `(help topic)` eingeben. Ohne Argument wird eine Zusammenfassung aller Yices-Kommandos wiedergegeben, mit Argument nur ein spezifischer `topic`.

- **Statistiken:**
  Der Solver speichert Spuren von verschiedenen Statistiken, die den Suchalgorithmus betreffen. Mit `(show-stats)` druckt man alle internen Statistiken aus. Yices speichert beispielsweise die Zeit, wie lange die Ausführung eines `(check)`-Befehls dauert.

- **Exit:**
  Der Solver kann zu jeder Zeit mit dem Befehl `(exit)` geschlossen werden.

In den folgenden vier Abbildungen 3.11-3.14 sind alle Keywords von Yices, alle Kommandos, alle Typen und alle Ausdrücke aufgelistet:

| | | |
|---|---|---|
| * | + | - |
| -> | / | /= |
| < | <= | <=> |
| = | => | > |
| >= | ^ | and |
| assert | bit | bitvector |
| bool | bool-to-bv | bv-add |
| bv-and | bv-ashift-right | bv-ashr |
| bv-comp | bv-concat | bv-div |
| bv-extract | bv-ge | bv-gt |
| bv-le | bv-lshr | bv-lt |
| bv-mul | bv-nand | bv-neg |
| bv-nor | bv-not | bv-or |
| bv-pow | bv-redand | bv-redor |
| bv-rem | bv-repeat | bv-rotate-left |
| bv-rotate-right | bv-sdiv | bv-sge |
| bv-sgt | bv-shift-left0 | bv-shift-left1 |
| bv-shift-right0 | bv-shift-right1 | bv-shl |
| bv-sign-extend | bv-sle | bv-slt |
| bv-smod | bv-srem | bv-sub |
| bv-xnor | bv-xor | bv-zero-extend |
| check | define | define-type |
| distinct | dump-context | echo |
| ef-solve | eval | exists |
| exit | export-to-dimacs | false |
| forall | help | if |
| include | int | ite |
| lambda | let | mk-bv |
| mk-tuple | not | or |
| pop | push | real |
| reset | reset-stats | scalar |
| select | set-param | set-timeout |
| show-implicant | show-model | show-param |
| show-params | show-stats | true |
| tuple | tuple-update | update |
| xor | | |

Abbildung 3.11: Yices-Keywords

```
<command>  ::=
           ( define-type <symbol> )
         | ( define-type <symbol> <typedef> )
         | ( define <symbol> :: <type> )
         | ( define <symbol> :: <type> <expression> )
         | ( assert <expression> )
         | ( exit )
         | ( check )
         | ( push )
         | ( pop )
         | ( reset )
         | ( show-model )
         | ( eval <expression> )
         | ( echo <string> )
         | ( include <string> )
         | ( set-param <symbol> <immediate-value> )
         | ( show-param <symbol> )
         | ( show-params )
         | ( show-stats )
         | ( reset-stats )
         | ( set-timeout <number> )
         | ( show-timeout )
         | ( dump-context )
         | ( help )
         | ( help <symbol> )
         | ( help <string> )
         | ( ef-solve )
         | ( export-to-dimacs <string> )
         | ( show-implicant )
         | EOS

<immediate-value> ::=
           true
         | false
         | <number>
         | <symbol>

<number> ::=
           <rational>
         | <float>
```

Abbildung 3.12: Yices-Syntax: Kommandos

```
<typedef> ::=
            <type>
          | ( scalar <symbol> ... <symbol> )

<type> ::=
            <symbol>
          | ( tuple <type> ... <type> )
          | ( -> <type> ... <type> <type> )
          | ( bitvector <rational> )
          | int
          | bool
          | real
```

Abbildung 3.13: Yices-Syntax: Typen

```
<expr> ::=
            true
          | false
          | <symbol>
          | <rational>
          | <float>
          | <binary bv>
          | <hexa bv>
          | ( forall ( <var_decl> ... <var_decl> ) <expr> )
          | ( exists ( <var_decl> ... <var_decl> ) <expr> )
          | ( lambda ( <var_decl> ... <var_decl> ) <expr> )
          | ( let ( <binding> ... <binding> ) <expr> )
          | ( update <expr> ( <expr> ... <expr> ) <expr> )
          | ( <function> <expr> ... <expr> )

<function> ::=
            <function-keyword>
          | <expr>

<var_decl> ::= <symbol> :: <type>

<binding> ::= ( <symbol> <expr> )
```

Abbildung 3.14: Yices-Syntax: Ausdruecke

# 4 Rätsellogik

In diesem Kapitel geht es um eine getypte Prädikatenlogik auf endliche Mengen. Diese Logik wird als **Rätsellogik** bezeichnet. Im ersten Abschnitt dieses Kapitels wird die Syntax dieser Logik behandelt. In Abschnitt 4.2 geht es um deren Semantik und um eine Erweiterung der Quantoren, nämlich um die „anzahlbeschränkten Existenzquantoren“.
Als Grundlage dieser Rätsellogik wird die Rätsellogik aus ([Dem15]) genommen.

## 4.1 Syntax der Rätsellogik

Die Syntax der Rätsellogik beinhaltet einige syntaktische Grundlagen der Aussagen- und der Prädikatenlogik erster Stufe, wobei es einige Unterschiede gibt. In der Prädikatenlogik erster Stufe können Individuenvariablen wie $x$ in $\forall x(P(x))$ jede beliebige Grundmenge für $x$ annehmen und jede beliebige Interpretation des Prädikats P ist möglich.
Im Gegensatz dazu geht es in dieser Arbeit um eine getypte Prädikatenlogik auf endlichen Mengen, welche feste vordefinierte Mengen der Individuenvariablen und bestimmte Interpretationen besitzen. Es werden nur Rätsel betrachtet bei denen der Kontext von vornherein gegeben ist und es klar ist, welche und wie viele Elemente vorhanden sind. Somit ist eine bestimmte, endliche Menge an Konstanten für $x$ gegeben.
Zusätzlich ermöglicht die Rätsellogik die Eingabe von Integerwerten und arithmetischen Operatoren. Diese können dazu genutzt werden, um neben dem Lösen, von in Aussagen verfassten Logeleien, auch Zahlenrätsel, wie beispielweise Sudokus oder magische Quadrate zu lösen.
Die Signatur der Rätsellogik ist wie folgt definiert:

**Definition 4.1** (Signatur). *Eine Signatur ist ein Tupel (K, S, P) wobei gilt:*

- *K ist eine nichtleere endliche Menge von Konstanten und enthält in seiner Größe beschränkte natürliche Zahlen (Integer).*
- *S ist eine nichtleere endliche Menge von Sorten S. Für jede Sorte $s \in S$ gibt es eine endliche nichtleere Menge von Konstanten konstanten(s) $\subseteq K$.*
- *P ist eine Menge von Prädikaten, wobei jedes Prädikat $p \in P$ eine Stelligkeit $ar(p) \geq 0$ hat. Jedes $p \in P$ hat einen Typ $p :: s_1 \to ...s_n \to \{True, False\}$.*

*Jede Konstante $k \in K$ gehört zu mindestens einer Sorte $k \in K$. Mit sorte(k) $\subseteq S$ bezeichnen wir die Sorten von k und fordern sorte(k) $\neq$ 0 für jedes $k \in K$. Wenn $s_i \in sorte(k)$, dann*

*schreiben wir auch* $k :: \mathrm{s_i}$.

**Definition 4.2** (Syntax der Rätsellogik). *Sei V eine abzählbar-unendliche Menge von Variablen. Ein Atom A über einer Signatur (K, S, P) ist ein Ausdruck der Form* $p(t_1, ..., t_n)$, *wenn* $p \in P$ *ein n-stelliges Prädikat ist und* $t_i$ *Terme sind. Wir nehmen an, dass S die Sorte Int enthält, und K entsprechende Konstanten (endlich viele) für beschränkte natürliche Zahlen. Formeln F, Terme T und Atome A sind durch die folgende Grammatik definiert, wobei* $x_i$ *in V,* $k_i$ *in K:*

$F ::= A \mid \neg\ F \mid \mathrm{F_1} \wedge \mathrm{F_2} \mid \mathrm{F_1} \vee \mathrm{F_2} \mid \mathrm{F_1} \Rightarrow \mathrm{F_2} \mid \mathrm{F_1} \Leftrightarrow \mathrm{F_2} \mid \forall \mathrm{x} \in s.F \mid \exists \mathrm{x} \in s.F$

$A ::= p(\mathrm{T_1}, ..., \mathrm{T_n})$

$T ::= \mathrm{x_i} \mid \mathrm{k_i} \mid T + T \mid T \text{ - } T \mid T * T \mid T \,/\, T$

### 4.1.1 Wohlgetyptheit von Formeln

Sei Γ eine Umgebung, die Variablen eine Sorte zuordnet (geschrieben als x::s, wenn x ∈ V und s ∈ S). Es gilt $\Gamma\ (\mathrm{x_i}) = \mathrm{s_i}$, wenn $\mathrm{x} :: \mathrm{s_i} \in \Gamma$.

**Definition 4.3** (Typisierung). *Eine Formel F ist wohl-getypt gdw. sich* $\theta \vdash F$ *mit den Typisierungsregeln herleiten lässt:*

$$\frac{p :: \mathrm{s_1} \to ... \to \mathrm{s_n} \to \{T, F\} \forall \mathrm{T_i} : \Gamma \vDash \mathrm{T_i} :: \mathrm{s_i}}{\Gamma \vdash p(\mathrm{T_1}, ..., \mathrm{T_n})}$$

$$\frac{\Gamma \cup \{x :: s\} \vdash F}{\Gamma \vdash \exists x \in s.F} \qquad \frac{\Gamma \cup \{x :: s\} \vdash F}{\Gamma \vdash \forall x \in s.F}$$

$$\frac{\Gamma \vdash F}{\Gamma \vdash \neg F} \qquad \frac{\Gamma \vdash \mathrm{F_1}, \Gamma \vdash \mathrm{F_2}}{\Gamma \vdash (\mathrm{F_1} \otimes \mathrm{F_2})}\ \otimes \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$$

$$\frac{\Gamma(x) = s}{\Gamma \vDash x :: s}$$

$$\frac{s \in Sorte(k)}{\Gamma \vDash k :: s} \qquad \frac{\Gamma \vDash \mathrm{T_1} :: Int, \Gamma \vDash \mathrm{T_2} :: Int}{\Gamma \vDash (\mathrm{T_1} \otimes \mathrm{T_2} :: Int)}\ \otimes \in \{+, -, *, /\}$$

**Beispiel 4.1:** *Typisierung*
Da die Rätsellogik Zahlen und arithmetische Operatoren enthalten kann, ist es mit ihr möglich Gleichungen aufzustellen und mehrere miteinander durch logische Operatoren zu verknüpfen.
Gegeben sei die Formel: $(x + y) == 10 \wedge (y \neq 8)$

Nach Anwendung der Typisierungsregeln ergibt sich:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(x) = Int}{\theta \vDash x :: Int} \quad \dfrac{\Gamma(y) = Int}{\theta \vDash y :: Int}}{\Gamma \vDash (x + y) :: Int} \quad \dfrac{Int \in Sorte(10)}{\Gamma \vDash == 10 :: Int}}{\theta \cup \{x :: Int, y :: Int\} \vdash (x + y == 10)} \quad \dfrac{\dfrac{\dfrac{\Gamma(y) = Int}{\theta \vDash y :: Int}}{\Gamma \vDash y} \quad \dfrac{Int \in Sorte(8)}{\Gamma \vDash \neq 8 :: Int}}{\theta \cup \{x :: Int, y :: Int\} \vdash (y \neq 8)}}{\theta \cup \{x :: Int, y :: Int\} \vdash (x + y == 10) \wedge (y \neq 8)}}{\theta \cup \{x :: Int\} \vdash \exists y \in Int (x + y == 10) \wedge (y \neq 8)}}{\theta \vdash \exists x \in Int . y \in Int : (x + y == 10) \wedge (y \neq 8)}$$

## 4.2 Semantik der Rätsellogik

Die Semantik der Rätsellogik ähnelt der Semantik der Prädikatenlogik mit einigen Änderungen bzw. Erweiterungen, wie beispielsweise den anzahlbasierten Existenzquantoren (siehe Kapitel 4.2.1).

Für eine Formel F steht $F[k/x]$ für die Ersetzung aller freien Vorkommen der Variablen x durch die Konstante k.
Sei F eine wohlgetypte Formel der Rätsellogik über einer Signatur (K, S, P) und einer Menge von Variablen x.

**Definition 4.4** (Interpretation). *Eine Interpretation (Bewertung) I bildet jedes Prädikat p ∈ P mit Stelligkeit n auf eine Menge von n-Tupeln typgerecht ab, d.h. I(p) ⊆ (konstanten(*$\mathrm{s}_1$*) ×...× konstanten(*$\mathrm{s}_n$*)), wenn p::*$\mathrm{s}_1 \rightarrow \ldots \rightarrow \mathrm{s}_n \rightarrow$ *{True, False}.*

$\mathrm{I}(==) = \{(k, k) | k \in K\}$
$\mathrm{I}(/=) = \{(k_1, k_2) | k_1 \in k, k_2 \in K\} \backslash \{(k, k) | k \in K\}$

*Die Erweiterung einer Interpretation auf wohlgetypte Formeln F ist definiert durch:*

$$\mathrm{I}(\mathrm{p}(\mathrm{t}_1, \ldots, \mathrm{t}_n)) = \begin{cases} 1 & \text{falls } (I(\mathrm{t}_1), \ldots, I(\mathrm{t}_n)) \in I(p) \\ 0 & \text{sonst} \end{cases}$$

$\mathrm{I}(\mathrm{k}_i) = \mathrm{k}_i$

$\mathrm{I}(\mathrm{t}_1 \otimes \mathrm{t}_\mathrm{n}) = \mathrm{n}$, wenn $I(\mathrm{t}_1) \otimes I(\mathrm{t}_\mathrm{n}) = \mathrm{n}$ für $\otimes \in \{+, -, *, /\}$ und n im Zahlenbereich liegt.

$$\mathrm{I}(\neg \mathrm{F}) \quad = \begin{cases} 1 & \text{falls } I(F) = 0 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \wedge \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = 1 \text{ und } I(\mathrm{F}_2) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \vee \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = 1 \text{ oder } I(\mathrm{F}_2) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \Rightarrow \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = 0 \text{ oder } I(\mathrm{F}_2) = 1 \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\mathrm{F}_1 \Leftrightarrow \mathrm{F}_2) = \begin{cases} 1 & \text{falls } I(\mathrm{F}_1) = I(\mathrm{F}_2) \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\exists x \in s.F) = \begin{cases} 1 & \text{falls es k} \in \text{konstanten(s) gibt mit I(F[k/x]) = True} \\ 0 & \text{sonst} \end{cases}$$

$$\mathrm{I}(\forall x \in s.F) = \begin{cases} 1 & \text{falls für alle k} \in \text{konstanten(s) gilt: I(F[k/x]) = True} \\ 0 & \text{sonst} \end{cases}$$

**Definition 4.5.** *Eine wohlgetypte Rätselformel ist*

- allgemeingültig genau dann, wenn für alle Interpretationen I gilt: I(F) = True.
- erfüllbar genau dann, wenn es eine Interpretation I gibt mit: I(F) = True.
- falsifizierbar genau dann, wenn es eine Interpretation I gibt mit I(F) = False.
- widersprüchlich genau dann, wenn für alle Interpretationen I gilt: I(F) = False.

Wenn I(F) = True gilt, so nennt man I ein Modell für F.

Es folgt ein Beispiel, um eine solche Formel F in Rätsellogik mit einer Signatur (K, S, P) zu sehen.

### 4.2.1 Beispiel

Sei ein Ritter-Schurken-Rätsel der folgenden Form gegeben:

| |
|---|
| A und B sind Ritter. |
| C und D sind Schurken. |
| Alle edlen Ritter bekämpfen einen Schurken. |
| B und A sind edel. |
| B kämpft nicht gegen C. |
| A kämpft nicht gegen D. |
| Wer bekämpft wen? |

Tabelle 4.1: Ritter-Schurken-Rätsel

*Sei F die dazugehörige Formel in Rätsellogik. Sei (K, S, P) die Signatur über der Formel F mit:*

$K = \{A,\ B,\ C,\ D\}$
$S = \{Ritter,\ Schurken\}$
$Ritter = \{A,\ B\}$
$Schurken = \{C,\ D\}$
$P = \{edel(x),\ bekämpft(x,y)\}$

$F = \forall x \in Ritter\ \exists y \in Schurken\ (edel(x) \Rightarrow bekämpft(x,y) \wedge edel(B) \wedge edel(A) \wedge \neg\ (bekämpft(B,C)) \wedge \neg\ bekämpft((A,D)))$

### 4.2.2 Anzahlbeschränkte Existenzquantoren

Neben den in 4.1.1 vorgestellten Quantoren beinhaltet die Rätsellogik die anzahlbeschränkten Existenzquantoren. Der gewöhnliche Existenzquantor stellt die Bedingung, dass mindestens ein Element wahr sein muss, um die Formel zu erfüllen. Anzahlbeschränkt bedeutet, dass eine festgelegte oder eingeschränkte Anzahl wahr sein muss, um die Formel zu erfüllen. Im Folgenden werden drei weitere Existenzquantoren mit diesen Bedingungen betrachtet.

**Definition 4.6** (Interpretation der anzahlbeschränkten Existenzquantoren)

Es existieren mindestens n Konstanten aus der Menge aller Konstanten: $\exists \geq$ n = `ExAtLeast n:`

$$\mathrm{I}(\exists \geq n\ x \in s.F) = \begin{cases} 1 & \text{gdw. es } \{\mathrm{k}_1, ..., \mathrm{k}_n\} \subseteq \mathrm{K(s)} \text{ gibt mit } |\{\mathrm{k}_1, ..., \mathrm{k}_n\}| \geq \mathrm{n} \\ & \text{und } \mathrm{I(F[k/x])} = \text{True} \\ 0 & \text{sonst} \end{cases}$$

Es existieren genau n Konstanten aus der Menge aller Konstanten: $\exists = \mathrm{n} =$ `ExExactly n:`

$$\mathrm{I}(\exists = n\ x \in s.F) = \begin{cases} 1 & \text{gdw. es } \{\mathrm{k}_1, ..., \mathrm{k}_n\} \subseteq \mathrm{K(s)} \text{ gibt mit } |\{\mathrm{k}_1, ..., \mathrm{k}_n\}| = \mathrm{n} \\ & \text{und } \mathrm{I(F[k/x])} = \text{True} \\ 0 & \text{sonst} \end{cases}$$

Es existieren höchstens n Konstanten aus der Menge aller Konstanten: $\exists \leq \mathrm{n} =$ `ExAtMost n:`

$$\mathrm{I}(\exists \leq n\ x \in s.F) = \begin{cases} 1 & \text{gdw. für alle } \{\mathrm{k}_1, ..., \mathrm{k}_n\} \subseteq \mathrm{K(s)} \text{ gibt mit } |\{\mathrm{k}_1, ..., \mathrm{k}_n| = \mathrm{n} + 1 \text{ gilt:}\} \\ & \mathrm{I(F[k/x])} = \text{False} \\ 0 & \text{sonst} \end{cases}$$

**Definition 4.7** (Erweiterte Typisierungsregeln). *Für die anzahlbeschränkten Existenzquantoren gelten die folgenden Typisierungsregeln:*

$$\frac{\Gamma \vdash \exists \in s.F}{\Gamma \vdash \exists \geq nx \in s.F} \qquad \frac{\Gamma \vdash \exists \in s.F}{\Gamma \vdash \exists = nx \in s.F} \qquad \frac{\Gamma \vdash \exists \in s.F}{\Gamma \vdash \exists \neq nx \in s.F}$$

### Beispiele mit anzahlbeschränkten Existenzquantoren

Aktuell sind sieben Mitglieder bei einem Treffen in Mars-Venus-Club: Arq, Bok, Cep, Dru, Eor, Fip und Gus. Es kommt ein Gast, der sie fragt, von welchem Planeten sie kommen und ob sie männlich oder weiblich sind. Drei von ihnen antworten:

Genau einer der Mitglieder ist ein Marsmann:
$\exists = 1$ x $\in$ Mitglieder $ist(x, Mann) \wedge von(x, Mars)$

Mindestens zwei sind Frauen:
$\exists \geq 2$ x $\in$ Mitglieder $ist(x, Frau)$

Höchstens drei kommen von der Venus:
$\exists \leq 3$ x $\in$ Mitglieder $von(x, Venus)$

# 5 Implementierung

In diesem Kapitel geht es um die Implementierung des Parsers, der aus einer Textdatei, in welcher die Benutzereingabe einer Logelei steht, eine Yices-Datei mit der Endung „.ys“ erzeugt. Diese wird vom Yices-Solver eingelesen, wobei dieser dann die Erfüllbarkeit eines solchen Logikrätsels überprüft und gegebenenfalls ein Modell erzeugt, insofern das Rätsel erfüllbar ist.

Zu Beginn dieses Kapitels geht es um einen Parser im Allgemeinen, um den Parsergenerator „Happy“, welcher einen Shift-Reduce-Parser erstellt und um die Erläuterung von dessen Eigenschaften.
In Abschnitt 5.2 wird erklärt in welcher Form der Benutzer ein Logikrätsel eingeben kann. Dazu wird die kontextfreie Grammatik dieser Eingabe definiert und es wird ein Beispiel gezeigt.
Darauf folgt das Typchecking und das Testen der Typchecks in Abschnitt 5.3, wozu einige Tests zur Korrektheit durchgeführt und Fehlermeldungen überprüft werden.
In Abschnitt 5.4 geht es um die Implementierung des Parser.

Außerdem wird in diesem Kapitel versucht diese Logik „natürlicher“ zu gestalten, indem Worte statt Symbole verwendet werden, sodass die Eingabe „lesbarer“ und näher an natürlichsprachlichen Sätzen ist. Dafür werden der Einfachheit halber englische Begriffe verwendet. Um dies zu gewährleisten ist ein Parser (Abschnitt 5.4) implementiert worden, der gültige Eingaben erkennt und falsche Eingaben zurückweist und schließlich gültige Eingaben in die Yices-Logik überführt.

In Abschnitt 5.5 wird der Code der Umwandlung von der Rätseleingabe in die Eingabesprache von Yices dargestellt und erläutert, bevor im letzten Abschnitt der Parser getestet wird.

## 5.1 Parser und Parsergenerator

### 5.1.1 Parser

Ein Parser ist ein Programm, das in der Informatik für die Zerlegung und Umwandlung einer Eingabe in ein für die Weiterverarbeitung geeignetes Format zuständig ist und die Gültigkeit einer Eingabe überprüft. Häufig werden Parser eingesetzt, um im Anschluss an den Analysevorgang die Semantik der Eingabe zu erschließen und daraufhin Aktionen durchzuführen.

Der Parser gibt die Analyse einer Eingabe in einer gewünschten Form aus und erzeugt zusätzlich Strukturbeschreibungen.
Die Vorverarbeitung der Eingabe ist in drei Phasen unterteilt:

1. lexikalische Analyse
2. syntaktische Analyse
3. semantische Analyse

Der lexikalische Scanner ist für die Analyse des Texts zuständig. Er wird auch Lexer genannt. Er zerlegt die Eingabedaten in Token, welche als atomare Eingabezeichen des Parsers dienen. Ein Token ist die kleinste zusammenhängende Einheit, die der Parser verarbeitet. Neben der Zerlegung von Token, entfernt der Lexer Leerzeichen und Kommentare und erkennt in der Eingabe Schlüsselwörter (TokenKeyword), Bezeichner (TokenIdentifier) und Symbole (TokenSymbol).
Die gesammelte Liste von Token übergibt er danach dem eigentlichen Parser, welcher sich um die Grammatik der Eingabe, also um die syntaktische Analyse kümmert. Er führt eine syntaktische Überprüfung der Eingabedaten durch, d.h. er überprüft die Grammatik der Eingabe und erstellt aus den Daten einen Ableitungsbaum, der schließlich zur semantischen Analyse benutzt wird.
In unserem Fall soll der Parser die Eingabe des Benutzers in die Eingabe für das Kernprogramm Yices umwandeln.
Dies geschieht mit Hilfe eines Parsergenerators, eines Programms, das einen Parser mithilfe der ihm gegebenen Spezifikationen erstellt. Da der Parser in der Sprache Haskell implementiert wird, bietet es sich an den Parsergenerator **Happy** ([Mar10]) zu verwenden, der speziell für Haskell entwickelt wurde und im Glasgow Haskell Compiler integriert ist.

### 5.1.2 Parsergenerator happy

Happy ist das für Haskell, was yacc für C ist. Sie sind beide Parsergeneratoren, die ähnlich aufgebaut sind. Sie erstellen aus einer Datei, deren Inhalt in der BNF [1] geschrieben ist, ein Haskell-Modul mit einem Parser für die entsprechende Grammatik. Wir benutzen Happy, da dort generierte Parser in der Regel schneller sind als andere.
Mit Happy hat man die Möglichkeit einen Lexer bzw. Tokenizer einzubinden, den man von einem Programm generieren lässt oder selbst schreibt. Happy kann aber theoretisch auch eine Folge von Zeichen direkt parsen, was aber meistens unpraktikabel ist.
Der Lexer unseres Parser wurde selbst erstellt, in den Parser eingebunden und in Abschnitt 5.4 wird unsere Vorgehensweise erläutert.
Happy-Dateien haben üblicherweise die Endung „.y“ und erzeugen eine Haskell-Datei mit der Endung „.hs“. Der Aufruf `happy Parser.y` erzeugt beispielsweise `Parser.hs`.

Happy erzeugt einen speziellen Parser: den Shift-Reduce-Parser. Auf diesen wird im nächsten Abschnitt näher eingegangen.

[1] Backus-Naur-Form - zur Darstellung kontextfreier Grammatiken

### 5.1.3 Shift-Reduce-Parser

Shift-Reduce-Parser gehören zu den LR-Parsern. Sie sind Bottom-Up-Parser für LR-Grammatiken sind. Das „L“ steht dafür, dass die Eingabe von links nach rechts verarbeitet wird, das „R“ dafür, dass eine Rechtsherleitung vorgenommen wird.

**Beispiel 5.1.** *Gegeben sei folgende kontextfreie Grammatik:*

E := E + E | E
E := E * E
E := (E)
E := a | b | c

Kontextfrei nennt man eine Grammatik, wenn auf der linken Seite der Regeln nur ein einziges Nichtterminal steht. Stehen auf der linken Seite mehrere Nichtterminale, muss nämlich der Kontext betrachtet werden, in dem Teile der Nichtterminale auftauchen.

**Definition 5.1.** *Eine kontextfreie Grammatik ist ein 4-Tupel G = (N,T,P,σ) mit*

1. *N: endliche Menge von Nichtterminalen*
2. *T: endliche Menge von Terminalen, wobei $N \cap T = 0$*
3. *$P \subseteq Nx(N \cup T)^*$ eine endliche Menge (Produktionssystem)*
4. *$\sigma \in N$ ist ein ausgezeichnetes Hilfszeichen (Startzeichen)*

*Es gibt zwei Rechtsherleitungen des Ausdrucks b - a * c. Die erste:*

E → E - E
→ E - E * E
→ E - E * c
→ E - a * c
→ b - a * c

*Zweite Möglichkeit:*

E → E * E
→ E * c
→ E - E * c
→ E - a * c
→ b - a * c

Da es mehrere Rechtsherleitungen geben kann, ist es wichtig, dass die Operatorprioritäten zusätzlich zur Grammatik in die Parserspezifikation geschrieben werden, damit keine falsche

Rechtsherleitung erzeugt wird.

**Shift-Reduce Analysemethode:**
Shift-Reduce steht für „Schiebe-Reduziere“und beschreibt die zwei wesentlichen Aktionen des Shift-Reduce-Parsers. Neben diesen gibt es noch die Aktionen „Accept“ und „Error“:

- Schieben: Liest ein Zeichen der Eingabe und schiebt dieses auf den Stack.
- Reduzieren: Ersetzt das oberste Stück des Stacks durch ein Nichtterminal.
- Akzeptieren: Akzeptiere, wenn auf dem Stack das Startsymbol steht und die Eingabe leer ist.
- Fehler: Gibt einen Fehler aus, wenn weder Schiebe- noch Reduzieraktion möglich ist und nicht akzeptiert werden kann.

Betrachten wir nochmals das Beispiel 5.1. Dieses mal benutzen wir einen Shift-Reduce-Parser, der eine Rechtsherleitung anhand einer Grammatik erzeugt, die in der Parserspezifikation steht:

| Stack | Eingabe | Aktion |
|---|---|---|
| $ | b-a*c$ | schiebe |
| $b | -a*c$ | reduziere mit E::=b |
| $E | -a*c$ | schiebe |
| $E- | a*c$ | schiebe |
| $E-a | *c$ | reduziere mit E::=a |
| $E-E | *c$ | schiebe |
| $E-E* | c$ | schiebe |
| $E-E*c | $ | reduziere mit E::=c |
| $E-E*E | $ | reduziere mit E::=E*E |
| $E-E | $ | reduziere mit E::=E-E |
| $E | $ | akzeptiere |

Tabelle 5.1: Beispiel Shift-Reduce-Parser

## 5.2 Benutzereingabe

Der Benutzer schreibt seine Rätseleingabe in eine Textdatei („rätsel.txt“). Der Inhalt einer solchen Datei ist in drei Abschnitte unterteilt. In den ersten Teil kommen die Sorten, in den zweiten die Prädikate und in den letzten Teil die Formeln des Logikrätsels. Dafür werden die Terminale `SET`, `RELATION` und `PROPOSITION` verwendet:

1. Sorten:
   `SET: EineSorte := {konstante1,konstante2}`
   Eine Sorte besteht aus einer endlichen Anzahl zugehöriger Konstanten.

2. Prädikate:
   `RELATION: prädikat SUBSET EineSorte`
   `RELATION: prädikat SUBSET (Sorte1 X Sorte2 X ... X Sorten`
   In einer Relation wird ein Prädikat definiert. `SUBSET` legt dabei fest, welchen Typ das Prädikat hat. `prädikat SUBSET EineSorte` bedeutet, dass das Prädikat eine Teilmenge der Sorte ist und `prädikat SUBSET (Sorte1 X Sorte2 X ... X Sorten` legt fest, dass das Prädikat eine n-stellige Relation über den entsprechenden Sorten ist.

3. Formeln:
   `PROPOSITION 1: Formel1` (6.2.1)
   `PROPOSITION 2: Formel2` (6.2.1)
   `PROPOSITION N: FormelN` (6.2.1)
   Es ist möglich eine endliche Anzahl von Formeln anzugeben. Die einzelnen Formeln werden in Form von durchnummerierten Aussagen angegeben.

Im nächsten Abschnitt wird näher auf die Eingabe der Formeln eingegangen.

### 5.2.1 Eingabe von Formeln

Formeln werden natürlichsprachig englisch formuliert. Sie werden durch die folgende kontextfreie Grammatik erzeugt, wobei *Formel, Atom, Predicate, Number, Sort, Term, Num* und *Variable* Nichtterminale sind und alle anderen Symbole Terminale sind:

*Formel* ::= *Atom*
| *Formel* `AND` *Formel*
| *Formel* `OR` *Formel*
| *Formel* `XOR` *Formel*
| `NOT` *Formel*
| *Formel* `IMPLIES` *Formel*
| *Formel* `IFF` *Formel*
| `EXISTS A` *Sort Variable* `SUCH THAT` *Formel*
| `EXISTS AT LEAST` *Number Sort Variable* `SUCH THAT` *Formel*
| `EXISTS AT MOST` *Number Sort Variable* `SUCH THAT` *Formel*
| `EXISTS EXACTLY` *Number Sort Variable* `SUCH THAT` *Formel*

| `FOR ALL` *Sort Variable* `HOLDS` *Formel*
| *(Formel)*

*Atom* ::= *Predicate*
| *Term* = *Term*
| *Term* /= *Term*

*Term* ::= *Num*
| *Term* * *Term*
| *Term* / *Term*
| *Term* + *Term*
| *Term* - *Term*

Die Grammatiken für die weiteren Nichtterminale werden informell beschrieben:

- *Number* und *Num* repräsentieren positive Ganzzahlen.
- *Variable* ist ein String aus Buchstaben beginnend mit einem Kleinbuchstaben.
- *Sort* ist ein String aus Buchstaben beginnend mit einem Großbuchstaben.
- Prädikate (Nichtterminale *Predicate*) werden in der Form `praedikatName(argument1,...,argumentN)` geschrieben, wobei nullstellige Prädikate als `praedikatName()` geschrieben werden. Sowohl der Name des Prädikats als auch die Argumente sind dabei Strings aus Buchstaben. Sie beginnen mit einem Kleinbuchstaben.

Da die Grammatik für Formeln in dieser Form mehrdeutig ist, müssen noch die Prioritäten und die Assoziativitäten der Operatoren festgelegt werden. Begonnen wird mit dem Operator mit der höchsten Priorität. In den Klammern jeweils steht die Assoziativität, also wie die Operatoren ausgewertet werden sollen:

`NOT` (gleichwertig),
= und /= (linksassoziativ),
* und / (linksassoziativ),
+ und - (linksassoziativ),
`AND` (linksassoziativ),
`OR` und `XOR` (linksassoziativ),
`IMPLIES` (ra), `IFF` (rechtsassoziativ),
`THAT` und `HOLDS` (gleichwertig)

## 5.3 Typcheck

Um vor der Transformation der Rätseleingabe in den Yices-Code potenzielle Fehler auszuschließen, wird entsprechend der Typregeln aus Abschnitt 4.1 ein Typcheck gemacht. In diesem Kapitel stehen nur Ausschnitte des Codes des Typchecks. Der gesamte Code ist zu finden auf http://www.ki.informatik.uni-frankfurt.de/master/programme/logicals-smt/ .

Der Typcheck eines Rätsels wird mit der Funktion `checkFormel` aufgerufen. Sie bekommt eine Formel mit ihrer jeweiligen Signatur übergeben und besteht aus den Hilfsfunktionen `typecheckPreamble` und `typecheckFormel`. Die erstere überprüft die Signatur und die letztere überprüft für die einzelnen Fälle, ob die verwendeten Sorten und Prädikate existieren, ob eine nicht definierte Sorte in der Formel vorkommt, ob ein Prädikat zu viele oder zu wenige Argumente übergeben bekommt und ob eine Variable eine falsche Sorte hat.

```
checkFormel (preamble,formel) =
  typecheckPreamble preamble &&
  typecheckFormel [] preamble formel
```

Die Funktion `typecheckPreamble` überprüft, ob ein Prädikat einer Sorte zugewiesen wurde, die nicht existiert.

```
typecheckPreamble (Preambel sorts predicates) =
 go predicates
 where
  go [] = True
  go ((p,typ):ps) = (check typ) && (go ps)
   where
    check [] = True
    check (t:ts) =
     case (lookup t sorts) of
      Just _ -> check ts
      Nothing ->
       error
        (unlines
         [,
         "*** Fehler waehrend des Typchecks:",
         "*** Das Praedikat "++ show p ++ "wurde mit dem Typ ",
         "*** "++ concat (intersperse > "(typ ++ ["Bool"])),
         "*** definiert, aber die Sorte "++ t ++ ëxistiert nicht."
         ])
```

Der Typcheck wird rekursiv auf alle Teilformeln angewendet. Für den logischen Operator der Implikation wird beispielsweise die Funktion `typcheckFormel` auf die linke Teilformel f1

und die rechte Teilformel f2 ausgeführt:

```
typecheckFormel gamma preamble (Impl f1 f2) =
 (typecheckFormel gamma preamble f1) &&
 (typecheckFormel gamma preamble f2)
```

Nach der Überprüfung der logischen Operatoren folgen die Quantoren. Hierbei wird exemplarisch an dem anzahlbeschränkten Existenzquantor „es existieren mindestens n“ getestet, ob die in der Formel verwendete Sorte in der Signatur existiert:

```
typecheckFormel gamma (Preambel sorts predicates) (ExAtLeast n x s f) =
 if s `elem` (map fst sorts) || s == "Int"
 then typecheckFormel ((x,s):gamma) (Preambel sorts predicates) f
 else
  error
   (unlines
     [,
     "*** Fehler waehrend des Typchecks:",
     "*** Die Sorte " ++ show s ++ ", die in",
     "*** EXISTS AT LEAST " ++ show n ++ " " ++ s ++ x ++ "...",
     "*** verwendet wurde, existiert nicht."
     ])
```

Bei den Gleichungen bzw. Ungleichungen wird überprüft, ob die beiden Terme einen gemeinsamen Typ haben:

```
typecheckFormel gamma (Preambel sorts predicates) (AtomF (Equality t1 t2)) =
 let termcheck = case t1 of
  Id x -> Left x
  other -> case checkAndGetTermTypeOperator gamma other of
   "Int" -> Right "Int"
   _ -> error "in typechecking"
 termcheck2 = case t2 of
  Id x -> Left x
  other -> case checkAndGetTermTypeOperator gamma other of
   "Int" -> Right "Int"
   _ -> error "in typechecking"
 go (Left a) (Left b) = if null [() | (s,elems) <- sorts, a `elem` elems,
   b `elem` elems] then case lookup a gamma of
    Nothing -> error ("freie variable " ++ a)
    Just typ -> case lookup b gamma of
     Nothing -> error ("freie variable " ++ b)
     Just typ2 -> if typ == typ2 then True else
```

```
     error ("type mismatch: die Terme " ++ show t1 ++ " " ++ show t2 ++
     " haben keinen gemeinsamen Typ")
   else True
```

Als nächstes wird überprüft, ob die in der Formel verwendeten Prädikate in der Signatur existieren und ob die Prädikate die richtige Stelligkeit haben:

```
typecheckFormel gamma (Preambel sorts predicates)(AtomF (Pred predi args))=
 case lookup predi predicates of
 Nothing ->
  error -- Praedikat existiert nicht
  (unlines
    [,
    "*** Fehler waehrend des Typchecks:",
    "*** Das in ",
    "*** " ++ predi ++ "(" ++ concat (intersperse "," args) ++ ")",
    "*** verwendete Praedikat " ++ show predi ++ " existiert nicht."
    ])
  Just typ -> check gamma sorts args typ
  where
   check gamma sorts args typ
    | length args /= length typ =
     error
      (unlines
     [,
     "*** Fehler waehrend des Typchecks:",
     "*** Praedikat " ++ show predi ++ " wird mit falscher Stelligkeit
     verwendet.",
     "*** Die Stelligkeit von " ++ predi ++ " ist " ++ show (length typ) ++ ",
     aber in",
     "*** " ++ predi ++ "(" ++ concat (intersperse "," args) ++ ")",
     "*** wird es mit Stelligkeit " ++ show (length args) ++ " verwendet.",
     "*** Bitte" ++ if length args < length typ then "mehr" else "weniger " ++
     "Argumente verwenden."
     ])
```

Zuletzt wird überprüft, ob die arithmetischen Operationen Terme vom Typ Integer verwenden. Dies wird exemplarisch an der Multiplikation gezeigt:

```
checkAndGetTermTypeOperator gamma (Num x) = "Int"
checkAndGetTermTypeOperator gamma (Times t1 t2)
 | checkAndGetTermTypeOperator gamma t1 == "Int"
 &&
```

```
 checkAndGetTermTypeOperator gamma t2 == "Int" = "Int"
 | checkAndGetTermTypeOperator gamma t1 /= "Int" = error
 "Typ von " ++ show t1 ++ "ist nicht Int, obwohl dieser erwartet war"
 | checkAndGetTermTypeOperator gamma t2 /= "Int" = error
 "Typ von " ++ show t2 ++ "ist nicht Int, obwohl dieser erwartet war"
```

### 5.3.1 Testen des Typchecks

In diesem Abschnitt wird der Typcheck anhand verschiedener Beispiele getestet. Bei den ersten beiden Testfällen sind absichtlich Fehler eingebaut, der letzte Testfall ist fehlerfrei und wird in Kapitel 6.1.1 von Yices gelöst.

#### Testfall 1:

```
testError1 :: Logical
testError1 = (preamble,formel)
 where
  preamble = Preambel sorts predicates
  sorts = [("Bewohner",["Bok","Ork"]),
    ("Planet", ["Mars", "Venus"])]
  predicates = [("fliegt",["Bewohner"])]
  formel = Pred "fliegt" ["Cep"]

*Main> checkFormel testError1
*** Exception:
*** Fehler waehrend des Typchecks:
*** Die Formel ist nicht geschlossen, da die Variable "Cep"
*** in fliegt(Cep)
*** frei vorkommt.
```

Der erste Testfall enthält die freie Variable `Cep`. Da keine freien Variablen erlaubt sind, wird die oben gezeigte Fehlermeldung ausgegeben. Um diesen Fehler zu beheben muss die Variable `Cep` beispielsweise der Sorte `Bewohner` als Konstante hinzugefügt werden.

#### Testfall 2:

```
testError2 :: Logical
testError2 = (preamble,formel)
 where
  preamble = Preambel sorts predicates
  sorts = [("Ritter",["Lancelot","Artus"]),
    ("Schurke", ["Hotzenplotz"])]
  predicates = [("sagtWahrheit",["Ritter"])]
```

```
  formel = Pred "sagtWahrheit" ["Artus","Lancelot"]

*Main> checkFormel testError2
*** Exception:
*** Fehler waehrend des Typchecks:
*** Praedikat ßagtWahrheit"wird mit falscher Stelligkeit verwendet.
*** Die Stelligkeit von sagtWahrheit ist 1, aber in
*** sagtWahrheit(Artus,Lancelot)
*** wird es mit Stelligkeit 2 verwendet.
*** Bitte weniger Argumente verwenden.
```

Gegeben ist das einstellige Prädikat `sagtWahrheit`. Die Formel enthält aber zwei Argumente, also eines zu viel. Um diesen Fehler zu beheben muss entweder ein Argument entfernt werden oder aus dem einstelligen Prädikat muss ein Zweistelliges gemacht werden.

**Testfall 3: Aristoteles:**

```
testAristoteles :: Logical
testAristoteles = (preamble,formel)
 where
 preamble = Preambel sorts predicates
 sorts = [("Mensch",["Aristoteles","Sokrates","Platon"]),
   ("Sterblichkeit", ["Sterblich","Unsterblich"])]
 predicates = [("ist",["Mensch","Sterblichkeit"])]
 formel = And [(rxor [AtomF $ Pred "ist" [Id "Aristoteles",Id "Sterblich"],
   AtomF $ Pred "ist" [Id "Aristoteles",Id "Unsterblich"]]),
     (rxor [AtomF $ Pred "ist" [Id "Sokrates",Id "Sterblich"],
   AtomF $ Pred "ist" [Id "Sokrates",Id "Unsterblich"]]),
     (rxor [AtomF $ Pred "ist" [Id "Platon",Id "Sterblich"],
   AtomF $ Pred "ist" [Id "Platon",Id "Unsterblich"]]),
     Forall "x" "Mensch" (AtomF $ Pred "ist" [Id "x",Id "Sterblich"])
     ]

*Main> checkFormel testAristoteles
True
```

Die Ausgabe ist `True`, d.h. es wurde kein Fehler während des Typchecks gefunden und die Transformation in die Yices-Logik kann beginnen.

## 5.4 Implementierung des Parsers

Im ersten Teils dieses Kapitels geht es um die Datentypen der Rätsellogik, gefolgt von den Direktiven, der Grammatik, den Schlüsselwörtern und dem Lexer.

### 5.4.1 Datentypen der Rätsellogik

Die Datentypen `Formula`, `Atom`, `Term` und `Preamble` sind Datentypen der Rätsellogik. Eine Formel der Rätsellogik wird durch den Datentyp `Formula` repräsentiert. Der Formeln werden durch die logischen Operatoren und die Quantoren geformt:

```
data Formula name = And [Formula name]
          | Or [Formula name]
          | Not (Formula name)
          | Impl (Formula name) (Formula name)
          | Equiv (Formula name) (Formula name)
          | Exists name Sortname (Formula name)
          | ExAtLeast Int name Sortname (Formula name)
          | ExAtMost Int name Sortname (Formula name)
          | ExExactly Int name Sortname (Formula name)
          | Forall name Sortname (Formula name)
          | AtomF (Atom name)
   deriving(Show,Eq)
```

Atome sind Prädikate (mit Argumenten), sowie spezielle Prädikate für den Gleichheits- und Ungleichheitstest:

```
data Atom name = Pred Predicate [Term name]
          | Equality (Term name) (Term name)
          | NEquality (Term name) (Term name)
   deriving(Show,Eq)
```

Die Terme der Rätsellogik formt man durch die Addition, Subtraktion, Multiplikation und Division:

```
data Term name = Id name
          | Num Int
          | Times (Term name) (Term name)
          | Divide (Term name) (Term name)
          | Plus (Term name) (Term name)
          | Minus (Term name) (Term name)
   deriving(Show,Eq)
```

Eine Formel der Rätsellogik besitzt eine Signatur (K,S,P), die im Programm durch den Datentyp `Preambel` dargestellt wird:

```
data Preambel = Preambel [(Sortname,[Element])] [(Predicate,[Sortname])]
```

In der Präambel sind somit die Konstanten, die Sorten und die Prädikate enthalten. Die Sorten werden hierbei durch Listen `[(Sortname,[Element])]` dargestellt, wobei jede Sorte durch ein Paar, das aus dem Sortennamen und der Liste der zugehörigen Konstanten `[Element]` besteht, dargestellt wird. Die Konstanten werden somit direkt ihren Sorten zugewiesen. Prädikate werden durch das Paar `[(Predicate,[Sortname])]` dargestellt, wobei `Sortname` den Typ des Prädikats dargestellt.
Die Typen `Predicate`, `Element` und `Sortname` sind Typsynonyme für Strings:

```
type Sortname = String
type Element = String
type Predicate = String
```

Der Datentyp `Logical` stellt ein in Rätsellogik kodiertes Rätsel dar. Er enthält die Preambel und die Formel selbst:

```
type Logical = (Preambel, Formula String)
```

### 5.4.2 Die Direktiven

Zu Beginn der Grammatik werden vier Parser definiert, wobei die Parser `parseSets`, `parseRels` und `parseProps` lediglich zum Testen der einzelnen Komponenten gedacht sind. Der Hauptparser, der die Direktive `START` als Startsymbol benutzt, heißt `parser`.

```
%name parser START
%name parseSets Sets
%name parseRels Rels
%name parseProps Props
```

Der Tokentyp `Token` ist der Tokentyp, den der Parser erkennen soll. Die Direktive %token definiert die möglichen Token. Die Symbole auf der linken Seite sind die Token, die auf den Rest der Grammatik verweisen. Rechts in den geschweiften Klammern stehen die Haskell Pattern, die die Token zuordnen:

```
%tokentype { Token }
%token
```

```
'{'                    { TokenSymbol "{"}
'}'                    { TokenSymbol "}"}
':='                   { TokenSymbol ":="}
'='                    { TokenSymbol -"}
'+'                    { TokenSymbol "+"}
'-'                    { TokenSymbol "}
'*'                    { TokenSymbol "*"}
'/'                    { TokenSymbol "/"}
'/='                   { TokenSymbol "/="}
':'                    { TokenSymbol ":"}
'('                    { TokenSymbol "("}
')'                    { TokenSymbol ")"}
'X'                    { TokenSymbol "X"}
','                    { TokenSymbol ","}
```

Ähnlich zu diesen TokenSymbols aufgebaut sind die TokenKeywords, die Tokenidentifier und die TokenInts.
Für eine korrekte Klammerung, werden folgende Bindungsstärken festgelegt:

```
%nonassoc 'THAT' 'HOLDS'
%left ',' 'X'
%right 'IFF'
%right 'IMPLIES'
%left 'OR' 'XOR'
%left 'AND'
%left '+' '-'
%left '*' '/'
%left '=' '/='
%nonassoc 'NOT'
%%
```

Die Begriffe %left, %right und %nonassoc geben an, ob ein Operator linksassoziativ, rechtssassoziativ oder gleichwertig ausgewertet werden soll. Der unterste Operator, in diesem Fall `NOT` bindet immer am stärksten. Umso höher die Operatoren stehen, umso schwächer binden sie. Bei den Junktoren bindet `AND` stärker als `OR` und `XOR`, wobei diese stärker als `IMPLIES` binden.

### 5.4.3 Grammatik

Die kontextfreie Grammatik des Parsers besitzt die Direktive `START` als Startsymbol und liefert ein Dreier-Tupel bestehend aus

- einer Liste von Mengenbeschreibung vom Typ Set,

- einer Liste von Relationen und
- einer Liste von nummerierten Aussagen

zurück.

```
START :: { ([Set], [Rel], [Prop]) }
START : Sets Rels Props            { ($1,$2,$3) }
```

Die Grammatik für Mengenausdrücke erlaubt Eingaben der Form:

```
SET: Mengenname := {konstante1,...,konstanteN}
```

Als Ausgabe wird der Datentyp Set verwendet, der im Grunde nur den Syntaxbaum darstellt. Später werden dann die erforderlichen Überprüfungen auf dem Syntaxbaum durchgeführt. Daher sieht die Grammatik wie folgt aus:

```
Sets :: { [Set] }
Sets : Set                         { [$1] }
| Set Sets                         { $1:$2 }
Set : 'SET' ':' IdG ':=' Objects   { Def ($3,$5) }

Objects : '{' List '}'             { $2 }
List : IdK                         { [$1] }
| IdK ',' List                     { $1:$3 }
```

Bei der Grammatik der Aussagen wird eine Aussage der Form `PROPOSITION Zahl: Formel` erwartet. `Formel` steht hierbei für beliebige Formeln der Rätsellogik. Die Rückgabe ist eine Liste von Aussagen, wobei jedes Listenelement ein Paar (Zahl,Formel) ist, da die Formeln nummeriert sind. Die Formel ist vom Typ `Formula String`.

```
Props : Prop                       { [$1] }
| Prop Props                       { $1:$2 }

Prop :'PROP' Number':'Formel       { As ($2,$4) }
```

Die Syntax der Grammatik für die Formeln orientiert sich an der englischen Sprache. Aussagenlogische Verknüpfungen werden mit `AND, OR, NOT, IMPLIES` und `IFF` dargestellt. Quantoren werden folgendermaßen dargestellt:

- Eine allquantifizierte Aussage der Form 'forall x in s.F' wird als '`FOR ALL S x HOLDS F`' repräsentiert, wobei `S` für die Sorte und `F` für die Formel steht.
- Eine Aussage mit Existenzquantor ohne Anzahlbeschränkung 'exists x in s.F' wird als '`EXISTS A S x SUCH THAT F`' repräsentiert.

- Der Existenzquantor mit Anzahlbeschränkung „mindestens“ wird als 'EXISTS AT LEAST n S x SUCH THAT F' dargestellt, wobei n die genaue Anzahl angibt.
- Der Existenzquantor mit Anzahlbeschränkung „höchstens“ wird als 'EXISTS AT MOST n S x SUCH THAT F' dargestellt, wobei n die genaue Anzahl angibt.
- Der Existenzquantor mit Anzahlbeschränkung „genau “ wird als 'EXISTS EXACTLY n S x SUCH THAT F' dargestellt, wobei n die genaue Anzahl angibt.

```
Formel :: { Formula String }
Formel : Predicate                  { $1 }
| Formel 'AND' Formel               { And [$1,$3] }
| Formel 'OR' Formel                { Or [$1,$3] }
| Formel 'XOR' Formel               { rxor [$1,$3] }
| 'NOT' Formel                      { Not $2 }
| Formel 'IMPLIES' Formel           { Impl $1 $3 }
| Formel 'IFF' Formel               { Equiv $1 $3 }
| '(' Formel ')'                    { $2 }
```

Prädikate werden in der Form `name(arg1,...,argn)` dargestellt. Es gibt auch nullstellige Prädikate `name()`. Die Grammatik der Prädikate sieht wie folgt aus:

```
Predicate : IdK '(' ')'             { AtomF (Rpredicate $1 []) }
| IdK '(' ArgList ')'               { AtomF (Rpredicate $1 $3) }
| Term '=' Term                     { AtomF (Equality $1 $3) }
| Term '/=' Term                    { AtomF (NEquality $1 $3) }

ArgList : IdK                       { [$1] }
| IdK ',' ArgList                   { $1:$3 }

Term : IdK                          { Id $1 }
| Term '+' Term                     { Plus $ 1 $3 }
| Term '-' Term                     { Minus $ 1 $3 }
| Term '*' Term                     { Times $ 1 $3 }
| Term '/' Term                     { Divide $ 1 $3 }
| Number                            { Num $1 }
```

### 5.4.4 Schlüsselwörter und Datentypen

In diesem Abschnitt werden sowohl die Datentypen der Mengenbeschreibungen, der „happy“-Error-Funktion und der Token als auch die Schlüsselwörter eingeführt.

Der Datentyp `Set` für die Mengenbeschreibungen besteht aus einem Sortennamen und eine Liste von Elementen:

```
data Set = Def (Sortname,[Element])
  deriving(Eq,Show)
```

Die „happy“-Error-Funktion erwartet eine Liste von Token und gibt den Typ a zurück:

```
happyError :: [Token] -> a
happyError [] = error "parse error: unerwartetes Ende"
happyError xs = error $ "parse error:"++ concat
(intersperse  (map showToken xs))
```

Die Datenstruktur für die Token ist wie folgt aufgebaut:

```
data Token = TokenKeyword String
    | TokenSymbol String
    | TokenIdentifier1 String
    | TokenIdentifier2 String
    | TokenInt Int
  deriving(Show)

showToken (TokenKeyword k) = k
showToken (TokenSymbol k) = k
showToken (TokenIdentifier1 k) = k
showToken (TokenIdentifier2 k) = k
showToken (TokenInt k) = show k
```

Bei diesen aufgezählten Tokens wird zwischen Schlüsselworten, Symbolen, Zahlen und Namen unterschieden, wobei TokenIdentifier1 für Namen verwendet wird, die mit einem Großbuchstaben beginnen und TokenIdentifier2 für Namen, die mit einem Kleinbuchstaben beginnen.

Es wurden eine Reihe von Schlüsselwörtern, `keywords`, benutzt:

```
keywords = ["SET", "SUBSET", "RELATION", "PROPOSITION", "EXISTS", "FOR",
"ALL", "SUCH", "THAT", "AND", "OR", "NOT", "LEAST", "MOST", "IMPLIES",
"IFF","AT", "A", "HOLDS", "XOR", "EXACTLY"]
```

### 5.4.5 Der Lexer

Zum Schluss wird der Lexer betrachtet. Dieser ist der vorverarbeitende Teil des Parsers. Er entfernt Leerzeichen zwischen den erlaubten Begriffen und verwandelt die Symbole in `TokenSymbols`. Kommentare werden mit -- eingeleitet, wobei der Lexer alle Zeichen des

Kommentars entfernt.

```
lexer :: String -> [Token]
lexer [] = []
lexer ('-':'-':xs) = lexer $ unlines $ tail (lines xs)
lexer cs
      | any (\k -> k 'isPrefixOf' cs && k == (takeWhile (isLetter) cs))
      keywords =
      let k = head (filter (\k -> k 'isPrefixOf' cs) keywords)
               l = length k
           in (TokenKeyword k):(lexer (drop l cs))

lexer ('X':xs) = (TokenSymbol "X"):(lexer xs)
lexer ('{':xs) = (TokenSymbol "{"):(lexer xs)
lexer ('}':xs) = (TokenSymbol "}"):(lexer xs)
lexer ('(':xs) = (TokenSymbol "("):(lexer xs)
lexer (')':xs) = (TokenSymbol ")"):(lexer xs)
lexer (',':xs) = (TokenSymbol ","):(lexer xs)
lexer ('=':xs) = (TokenSymbol ","):(lexer xs)
lexer ('+':xs) = (TokenSymbol ","):(lexer xs)
lexer ('-':xs) = (TokenSymbol ","):(lexer xs)
lexer ('*':xs) = (TokenSymbol ","):(lexer xs)
lexer ('/':xs) = (TokenSymbol ","):(lexer xs)
lexer ('/':'=':xs) = (TokenSymbol ","):(lexer xs)
lexer (':':'=':xs) = (TokenSymbol ":="):(lexer xs)
lexer (':':xs) = (TokenSymbol ":"):(lexer xs)

lexer (x:xs)
      | isSpace x = lexer xs
      | isUpper x = let (v,rest) = span isLetter xs
                in (TokenIdentifier1 (x:v)) : (lexer rest)
      | isLower x = let (v,rest) = span isLetter xs
                in (TokenIdentifier2 (x:v)) : (lexer rest)
      | isDigit x = let (v,rest) = span isDigit xs
                in (TokenInt (read $ x:v)) : (lexer rest)
      | otherwise = error ("parse error, can't lex symbol "++ show x)
```

Nach der lexikalischen Analyse beginnt die syntaktische Analyse durch den Parser.

Im nächsten Abschnitt wird erklärt, wie der Yices-Code erzeugt wird.

## 5.5 Umwandlung in die Yices-Logik

Wie in Kapitel 5.2 gesehen, geschieht die Eingabe eines Rätsels in der Form:

```
SET: Set1 := {a,b,c}
SET: Set2 := {d,e,f}

RELATION: Relation1 SUBSET (SET1 X SET2)

PROPOSITION X: FOR ALL ...
```

In diesem Kapitel geht es um die Umwandlung einer solchen Eingabe in einen für Yices ausführbaren Code. Die Hauptaufrufe für die Mengensätze, Relationen und Propositionen sind dafür `setToYices (Def (name,elems))`, `relToYices (D (symbol,elems))` und `propToYices (As (zahl, formula))`.

### 5.5.1 Mengensätze SET

`SET` definiert in der Rätsellogik die existierenden Sorten. Diese bestehen aus mehreren Konstanten.

```
setToYices (Def (name,elems)) = "(define-type " ++ name ++
     "(scalar " ++ concat (intersperse " " elems) ++ "))"
setsToYices xs = unlines (map setToYices xs)
printSets inp0 = putStrLn (setsToYices (parseSets (lexer inp0)))
```

Der Befehl `setToYices` wandelt Eingaben der Form `SET: Test :=` {a,b,c} in Yices-Code um. Sorten werden in Yices mit dem Befehl `(define-type ...)` definiert. In Zeile 2 werden Zeilenumbrüche erstellt, falls mehrere Sorten definiert wurden. Die Funktion `printSets` transformiert ausschließlich die Sorten in Yices-Code und dient lediglich zum Testen dieser Sorten.

### 5.5.2 Relationen

In den Relationen werden die Prädikate der Rätsellogik definiert.

```
relToYices (D (symbol,elems)) = "(define " ++ symbol ++ " ::
     (-> "++ concat (intersperse " " elems) ++ " bool))"
relsToYices xs = unlines (map relToYices xs)
printRels inp1 = putStrLn (relsToYices (parseRels (lexer inp1)))
```

In Yices braucht man dafür den Befehl `(define ...)`. In der ersten Zeile werden die Relationen definiert und wenn es mehrere gibt, erstellt `relsToYices` Zeilenumbrüche. Die Funktion

`printRels` transformiert ausschließlich die Relationen in Yices-Code und dient lediglich zu Testzwecken.

### 5.5.3 Propositionen

Nachdem die Sorten und Prädikate definiert wurden, werden in den Propositionen die eigentlichen Formeln definiert.

```
propToYices (As (zahl, formula))="(assert " ++ formulaToYices formula++")"

atomToYices (Rpredicate pname []) = pname
atomToYices (Rpredicate pname args) = "(" ++ pname ++ (concat
   (intersperse " " (map termToYices args))) ++ ")"
atomToYices (Equality t1 t2) ="(= " ++ termToYices t1 ++termToYices t2 ++")"
atomToYices (NEquality t1 t2) ="(/= " ++termToYices t1 ++termToYices t2 ++")"

termToYices :: Term String -> String
termToYices (Id name) = name
termToYices (Num z) = show z
termToYices (Times t1 t2) = "(* "++ termToYices t1 ++ termToYices t2 ++ ")"
termToYices (Plus t1 t2) = "(+ "++ termToYices t1 ++ termToYices t2 ++ ")"
termToYices (Minus t1 t2) = "(- "++ termToYices t1 ++ termToYices t2 ++ ")"
termToYices (Divide t1 t2) = "(/ "++ termToYices t1 ++ termToYices t2 ++ ")"

formulaToYices (AtomF a) = atomToYices a
formulaToYices (Not formula) = "(not " ++ formulaToYices formula ++ ")"
formulaToYices (Or list) = "(or" (map formulaToYices list)++")"
formulaToYices (And list) = "(and" (map formulaToYices list)++")"
formulaToYices (Impl f1 f2) = "(=>" (map formulaToYices [f1,f2])++")"
formulaToYices (Equiv f1 f2)="(and" (map formulaToYices[f1,f2])++")"
formulaToYices (Exists name sort f) =
   "(exists("++ name ++"::"++ sort ++")"++ formulaToYices f ++")"
formulaToYices (Forall name sort f) =
   "(forall("++ name ++"::"++ sort ++")"++ formulaToYices f ++")"
```

In Yices dient der Befehl `assert` zur Definition der Aussagen (Propositionen). Die Funktion `propToYices` ist für die Umwandlung der Eingabe `PROPOSITION x` in `(assert ...` zuständig. Die Formeln, die logische Operatoren enthalten, werden in der Funktion `formulaToYices` verarbeitet, Formeln mit arithmetischen Operatoren in der Funktion `termToYices`, wobei jene nur aufgerufen wird, wenn die Funktion `atomToYices` ein (Un-)Gleichung erstellt hat. Neben den logischen Operatoren, Quantoren oder arithmetische Operatoren können Formeln auch anzahlbeschränkten Existenzquantoren enthalten.

**Anzahlbeschränkte Existenzquantoren:**
Es gibt drei anzahlbeschränkte Existenzquantoren, nämlich:

- `ExAtLeast n`: es existieren mindestens n
- `ExExactly n`: es existieren genau n
- `ExAtMost n`: es existieren höchstens n

Der Existenzquantor für „es existieren mindestens n“ wird folgendermaßen erzeugt:

```
formulaToYices (ExAtLeast anzahl name sort f) =
let f' = formulaToYices f
   list = list1 ++ list2
   list1 = ["(/= x" ++ show i ++ " x" ++ show j ++ ")" | i <- [1..anzahl],
     j <- [i+1..anzahl], i /= j]
   list2 = ["(formula x" ++ show i ++ ")" | i <- [1..anzahl]]
in "(let ((formula (lambda (" ++ name ++ "::" ++ sort ++")" ++ f' ++ ")))" ++
   "(exists (" ++ concat (intersperse " "
     ["x" ++ show i)++"::" ++ sort ++ |
     i <- [1..anzahl]]) ++ ")"
   ++ " (and " ++ (concat (intersperse " " list)) ++ ")))"
```

Die Erzeugung des Existenzquantor für „es existieren höchstens n“ funktioniert ähnlich dazu:

```
formulaToYices (ExAtMost anzahl name sort f) =
let f' = formulaToYices f
   list = list1 ++ list2
   list1 = ["(/= x" ++ show i ++ " x" ++ show j ++ ")" | i <- [1..anzahl+1],
     j <- [i+1..anzahl+1], i /= j]
   list2 = ["(formula x" ++ show i ++ ")" | i <- [1..anzahl+1]]
in "(let ((formula (lambda (" ++ name ++ "::" ++ sort ++")" ++ f' ++ "))) " ++
   "(not (exists (" ++ concat (intersperse " "
     ["x" ++ show i)++"::" ++ sort ++ |
     i <- [1..anzahl+1]]) ++ ")"
   ++ " (and " ++ (concat (intersperse " " list)) ++ "))))"
```

Der Existenzquantor für „es existieren genau n“ ist schließlich die Konjunktion der beiden anderen Existenzquantoren:

```
formulaToYices (ExExactly anzahl name sort f) =
     "(and " ++ formulaToYices (ExAtLeast anzahl name sort f) ++
     formulaToYices (ExAtMost anzahl name sort f) ++ ")"
```

Die Funktion `toBinOp` wird aufgerufen, wenn mehrere gleiche Operationen durchgeführt werden:

```
toBinOp op [] =
toBinOp op [y] = y
toBinOp op (x:y:ys) = op ++ " (" ++ x ++ ") (" ++ toBinOp op (y:ys) ++ ")"
```

Der Äquivalenz-Operator (<=> für Yices2) ist in Yices1 nicht implementiert. Die Funktion `equiConv` erstellt diesen Operator, indem sie zwei Implikationen durchführt, wobei eine beispielsweise aus Aussage A Aussage B impliziert und die andere aus Aussage B Aussage A ($A \rightarrow B \wedge B \rightarrow A$). Die Funktion `equiConv` macht dies wie folgend:

```
equiConv eqc [] =
equiConv eqc [y] = y
equiConv eqc (x:y:ys) = eqc ++ " (=> (" ++ x ++ ") (" ++ equiConv eqc (y:ys)
     ++ "))" ++ " (=> (" ++ y ++ ") (" ++ equiConv eqc (x:ys) ++ "))"
```

Die Funktion `printProps` erzeugt eine Testausgabe der Formeln:

```
propsToYices xs = unlines (map propToYices xs)
printProps inp2 =
     putStrLn ((propsToYices (parseProps (lexer inp2))) ++ "(check)")
```

### 5.5.4 Yices-Datei erzeugen

Der Befehl `run str` ist zum Testen des gesamten Codes da, wobei die Funktion `allToYices` dafür zuständig ist, die Sorten, Relationen und Propositionen zu verknüpfen:

```
run str = putStrLn $ allToYices $ parser $ lexer str
allToYices (a,b,c) =
   concat[setsToYices a, relsToYices b, propsToYices c, ausgabe, ausgabe2]
```

Um schließlich eine komplette ausführbare Yices-Datei zu erzeugen, die schlussendlich die Rätsel auf Erfüllbarkeit überprüfen und ein Modell, wenn vorhanden, ausgeben soll, wird die Funktion `readAndConvert` aufgerufen. Diese liest eine Rätsel-Datei ein, verarbeitet den Inhalt und erzeugt eine Yices-Datei mit Endung „.ys“.

```
readAndConvert file = do
 c <- readFile file
 let out = allToYices $ parser $ lexer c
 writeFile (file ++ ".ys") out
```

## 5.6 Testen des Parsers

Ein wichtiger Punkt ist das Testen des Parsers. Es wird überprüft, welche Fehlermeldungen er ausgibt und welche Fehler nicht erkannt werden.

Für einfachere Tests ist der Parser in Einzelparser zerlegt worden:

```
parse1 = parseSets . lexer
parse2 = parseRels . lexer
parse3 = parseProps . lexer
```

Durch diese Zerlegung ist es möglich bestimmte Teile der Eingabe zu testen, wobei `parse1` die Mengeneingaben überprüft, `parse2` die Eingabe der Relationen und `parse3` die Propositionen.

**Tests der Mengeneingaben:**

```
*Parser> printSets "SET: Bewohner := {ritter}"
(define-type Bewohner (scalar ritter))
```

**Tests der Prädikate:**

```
*Parser> printRels "RELATION: istein SUBSET (Bewohner)"
(define istein :: (-> Bewohner bool))
```

**Tests der Propositionen:**

```
*Parser> printProps "PROPOSITION 1: NOT istein (ritter)"
(assert (not (istein ritter)))
```

Obwohl jeder Einzelparser getestet wurde, können trotzdem noch Fehler auftreten und zwar derart, dass Sorten, Prädikate oder Konstanten in Propositionen verwendet werden, die in der Signatur nicht definiert wurden. Deshalb wird der gesamte Parser getestet:

```
*Parser> run "SET: Bewohner := {ritter} RELATION: istein SUBSET
   Bewohner PROPOSITION 1:istein (ritter)"
(define-type Bewohner (scalar ritter))
(define istein :: (-> Bewohner bool))
(assert (istein ritter))
```

Im nächsten Kapitel werden verschiedene Logikrätsel betrachtet, in Yices-Code transformiert und die dazugehörigen Lösungsmodelle angegeben.

# 6 Tests und Ergebnisse

In diesem Kapitel werden verschiedene Rätsel mit unterschiedlichen Schwierigkeitsgraden vorgestellt, gelöst, deren Modelle gezeigt und hinsichtlich Laufzeit und Speicherplatzverbrauch betrachtet. Für jedes Rätsel wird zum einen die Eingabe und zum anderen der entstandene Yices-Code dargestellt.
Anschließend werden in Abschnitt 6.2 die Ergebnisse mit den Ergebnissen aus [Dem15] verglichen und überprüft, welche Möglichkeit Logikrätsel zu lösen für den Anwender am effizientesten ist.

Die folgenden Tests wurden mit dem Vierkernprozessor Intel(R) Core(TM) i5 CPU M430 mit 2,27GHz und einem Arbeitsspeicher von 4 GB auf einem 64 Bit-Betriebssystem (Linux) durchgeführt.

## 6.1 Testen verschiedener Rätsel

Es werden fünf Rätsel betrachtet: *Aristoteles* (6.1.1), *Auf der Suche nach Oona* (6.1.2), *Interplanetarische Verwicklungen* (6.1.3), *Wolfswürmer* (6.1.4), *Magisches Quadrat*. Zu jedem dieser Rätsel werden die folgenden vier Punkte abgearbeitet:

1. Beschreibung des Rätsels
2. Rätseleingabe
3. Rätsel in Yices-Code
4. Auswertung

**Zu beachten ist:**
Zwischen den Codes der beiden SMT-Solver Yices und Yices2 gibt es nur einen Unterschied. Um ein Modell auszugeben benötigt Yices `-e` als Eingabe in der Console. Bei Yices2 muss am Ende des Codes das Kommando `(show-model)` hinzugefügt werden.

Die Konvertierung eines Rätsels in die Eingabesprache von Yices unter Verwendung des GHC-Interpretes geschieht mit dem Befehl:
*Parser> readAndConvert "textdatei.txt"

Das Lösen eines Rätsels mit Hilfe des Yices-SMT-Solvers geschieht mit den Befehlen
*yices –e textdatei.ys* unter Windows und *.\yices –e textdatei.ys* unter Linux.

### 6.1.1 Aristoteles

Dieses Rätsel wurde vermutlich selbst von dem Philosophen Aristoteles entworfen. Es gilt als eines der ersten formal aufgestellten logischen Rätsel.

#### Beschreibung des Rätsels

| |
|---|
| Prämisse 1: Alle Menschen sind sterblich.<br>Prämisse 2: Aristoteles, Sokrates und Platon sind Menschen. |
| Sind die Griechen Aristoteles, Sokrates und Platon sterblich? |

#### Eingabe

```
SET: Mensch:= {aristoteles,sokrates,platon}
SET: Sterblichkeit:= {sterblich,unsterblich}

RELATION: ist SUBSET (Mensch X Sterblichkeit)

PROPOSITION 1: ist(aristoteles,sterblich) XOR ist(aristoteles,unsterblich)
PROPOSITION 2: ist(sokrates,sterblich) XOR ist(sokrates,unsterblich)
PROPOSITION 3: ist(platon,sterblich) XOR ist(platon,unsterblich)

PROPOSITION 4: FOR ALL Mensch x HOLDS ist(x,sterblich)
```

Mit `SET` werden die Sorten `Mensch` und `Sterblichkeit` definiert und zugleich auch die Konstanten `aristoteles,sokrates,platon` und `sterblich,unsterblich`. Somit ist auch die Zugehörigkeit der Konstanten zu den Sorten definiert.
`RELATION` definiert das zweistellige Prädikat `ist`, welches jeweils aus einer Konstante beiden Sorten besteht.
Die ersten drei Propositionen besagen, dass die drei genannten Menschen entweder sterblich oder unsterblich sind. Die letzte Proposition besagt, dass alle Menschen sterblich sind.

#### Yices-Code

Nachdem der Parser die Eingabe des Rätsels in die Logik von Yices transformiert hat, entsteht folgender Code:

```
(define-type Mensch (scalar aristoteles sokrates platon))
(define-type Sterblichkeit (scalar sterblich unsterblich))

(define ist :: (-> Mensch Sterblichkeit bool))

(assert (and (or (ist aristoteles sterblich) (ist aristoteles unsterblich))
  (not (and (ist aristoteles sterblich) (ist aristoteles unsterblich)))))
(assert (and (or (ist sokrates sterblich) (ist sokrates unsterblich)) (not
 (and (ist sokrates sterblich) (ist sokrates unsterblich)))))
(assert (and (or (ist platon sterblich) (ist platon unsterblich)) (not (and
  (ist platon sterblich) (ist platon unsterblich)))))
(assert (forall (x::Mensch)(ist x sterblich)))

(check)
```

**Auswertung**

```
==========================================
            Erfuellbarkeit + Modell:
==========================================
unknown
(= (ist aristoteles sterblich) true)
(= (ist aristoteles unsterblich) false)
(= (ist sokrates sterblich) true)
(= (ist sokrates unsterblich) false)
(= (ist platon sterblich) true)
(= (ist platon unsterblich) false)

==========================================
```

Das ausgegebene Modell gibt an, dass es wahr ist, das Aristoteles, Sokrates und Platon sterblich sind und dass es falsch ist, dass sie unsterblich sind.

Die Laufzeit beträgt 0,01 Sekunden und der Speicherverbrauch liegt bei 4,41MB.

### 6.1.2 Auf der Suche nach Oona

Dieses Rätsel stammt aus dem Buch „Logik-Ritter und andere Schurken“([SB91]).

#### Beschreibung des Rätsels

| |
|---|
| Im Südpazifik gibt es einige Inseln auf denen nur Ritter und Schurken wohnen. Ritter haben die Eigenschaft, dass sie immer die Wahrheit sagen, wohingegen Schurken immer lügen. Einige der Bewohner sind zusätzlich halb Mensch, halb Vogel. Eine von diesen ist Oona. Sie fliegt gerne von Insel zu Insel und sagt ihrem Mann, der sich selbst als Logiker bezeichnet, nie, auf welche Insel sie fliegt.<br>Eines Tages ist sie mal wieder weggeflogen und ihr Mann macht sich auf die Suche. Er fährt mit seinem Kanu und fragt die Einwohner der Inseln, ob sie Oona gesehen hätten. Als er auf eine Insel mit fünf Bewohner kam, bekam er von ihnen folgende Antworten: |
| A: Oona ist auf dieser Insel.<br>B: Oona ist nicht auf dieser Insel.<br>C: Oona war gestern hier.<br>D: Oona ist heute nicht hier und sie war auch gestern nicht hier.<br>E: Entweder ist D ein Schurke oder C ist ein Ritter.<br><br>Der Logiker dachte eine Weile darüber nach, konnte aber nichts damit anfangen und fragt, ob einer der Bewohner noch eine Aussage machen könnte. Daraufhin erwidert A: Entweder ist E ein Schurke oder C ist ein Ritter.<br><br>Ist Oona auf der Insel? |

Tabelle 6.1: Auf der Suche nach Oona

#### Eingabe

```
SET: Bewohner := {oona,a,b,c,d,e}
SET: Beruf := {ritter,schurke}
SET: Zeitpunkt := {heute,gestern,nie}

RELATION: ist SUBSET (Bewohner X Beruf)
RELATION: wann SUBSET (Bewohner X Zeitpunkt)

PROPOSITION 1: ist(a,ritter) IFF wann(oona,heute)
PROPOSITION 2: ist(b,ritter) IFF wann(oona,nie)
```

```
PROPOSITION 3: ist(c,ritter) IFF wann(oona,gestern)
PROPOSITION 4: ist(d,ritter) IFF wann(oona,nie)
PROPOSITION 5: ist(e,ritter) IMPLIES (ist(d,schurke) XOR ist(c,ritter))
PROPOSITION 6: ist(a,ritter) IMPLIES (ist(e,schurke) XOR ist(c,ritter))
PROPOSITION 7: FOR ALL Bewohner x HOLDS ist(x,ritter) XOR ist(x,schurke)
PROPOSITION 8: wann(oona,heute) XOR wann(oona,gestern) XOR wann(oona,nie)
```

Mit `SET` werden die Sorten `Bewohner`, `Beruf` und `Zeitpunkt` definiert und zugleich auch die Konstanten `oona,a,b,c,d,e`, `ritter,schurke` und `heute,gestern,nie`.
`RELATION` definiert die Prädikate `wann` und `ist` und legt fest, dass sie jeweils zweistellige Prädikate sind, die aus einem Tupel bestehen, welche jeweils eine Konstante einer Sorte und eine Konstante einer anderen Sorte enthalten.
So besagt beispielsweise die erste `PROPOSITION`, dass Oona genau dann heute auf der Insel ist, wenn A die Wahrheit sagt.

### Yices-Code

```
(define-type Bewohner (scalar oona a b c d e))
(define-type Beruf (scalar ritter schurke))
(define-type Zeitpunkt (scalar heute gestern nie))

(define ist :: (-> Bewohner Beruf bool))
(define wann :: (-> Bewohner Zeitpunkt bool))

(assert(and(=>(ist a ritter)(wann oona heute))(=>(wann oona heute)
 (ist a ritter))))
(assert(and(=>(ist b ritter)(wann oona nie))(=>(wann oona nie)
 (ist b ritter))))
(assert(and(=>(ist c ritter)(wann oona gestern))(=>(wann oona gestern)
 (ist c ritter))))
(assert(and(=>(ist d ritter)(wann oona nie))(=>(wann oona nie)
 (ist d ritter))))
(assert(=>(ist e ritter)(and(or(ist d schurke)(ist c ritter))(not(and
 (ist d schurke)(ist c ritter))))))
(assert(=>(ist a ritter)(and(or(ist e schurke)(ist c ritter))(not(and
 (ist e schurke)(ist c ritter))))))
(assert(forall(x::Bewohner)(and(or(ist x ritter)(ist x schurke))(not
 (and(ist x ritter)(ist x schurke))))))
(assert(and(or(and(or(wann oona heute)(wann oona gestern))(not(and
 (wann oona heute)(wann oona gestern))))(wann oona nie)) (not(and(and
 (or(wann oona heute)(wann oona gestern))(not(and(wann oona heute)
 (wann oona gestern))))(wann oona nie)))))
```

### Auswertung

```
==========================================
            Erfuellbarkeit + Modell:
==========================================
unknown
(= (ist a ritter) false)
(= (wann oona heute) false)
(= (ist b ritter) false)
(= (wann oona nie) false)
(= (ist c ritter) true)
(= (wann oona gestern) true)
(= (ist d ritter) false)
(= (ist e ritter) false)
(= (ist d schurke) true)
(= (ist e schurke) true)
(= (ist a schurke) true)
(= (ist b schurke) true)
(= (ist c schurke) false)
```

Der Logiker erkennt, dass C der einzige Ritter ist und die anderen Bewohner allesamt Schurken sind. Da Ritter immer die Wahrheit sagen, bedeutet dies, dass Oona gestern auf der Insel war und sich heute bereits woanders aufhält.

Trotz mehrerer Proposition braucht Yices auch bei diesem Rätsel nur 10 Millisekunden und verbraucht mit 4,5MB nur etwas 90kB mehr als das Aristoteles-Rätsel.

### 6.1.3 Interplanetarische Verwicklungen

Dieses Rätsel ist eine abgeänderte Logelei aus dem Buch „Logik-Ritter und andere Schurken“ ([SB91]).

#### Beschreibung des Rätsels

| |
|---|
| Auf einem Satelliten des Jupiters, namens Ganymed, gibt es den Mars-Venus-Club. Alle Mitglieder dieses Clubs stammen vom Mars oder von der Venus. An manchen Tagen sind auch Gäste zugelassen. Heute ist ein solcher Tag. Die Erdenbürger Oona und ihr Mann, ein Logiker, besuchen Ganymed. Leider ist es ihnen nicht vergönnt zu erkennen von welchem Planeten die Mitglieder Clubs kommen und welchen Geschlechts sie sind. Da sie neugierig sind fragen sie einige Bewohner nach Geschlecht und Herkunft. Sie bekommen folgende Antworten: |
| Arq: Soweit ich weiß sind drei oder vier von uns Frauen.<br>Bok: Ich zähle zwei Venusmänner.<br>Cep: Wenn Arq von der Venus ist, sind dies Fip und Bok ebenfalls.<br>Dru: Ich weiß, dass Eor vom Mars ist und Arq eine Frau ist.<br>Eor: Gus ist auch eine Frau und Arq wäre mir auf dem Mars bestimmt schon begegnet.<br>Fip: Bok ist genau dann ein Mann, wenn Dru oder Gus Frauen sind.<br>Gus: Wenn Cep vom Mars kommt, dann kommt Gus auch daher.<br><br>Diese Informationen reichen dem Logiker und Oona nicht aus, weshalb sie nochmal nachhaken: Wer sind denn die zwei Venusmänner?<br><br>Bok: Ich kann euch nur versichern, dass Dru und Eor nicht von der Venus stammende Männer sind. |

Tabelle 6.2: Interplanetarische Verwicklungen

#### Eingabe

```
SET: Name := {arq,bok,cep,dru,eor,fip,gus}
SET: Planet := {mars,venus}
SET: Geschlecht := {mann,frau}

RELATION: von SUBSET (Name X Planet)
RELATION: ist SUBSET (Name X Geschlecht)
```

```
PROPOSITION 1: EXISTS AT LEAST 3 Name x SUCH THAT ist(x,frau)
PROPOSITION 2: EXISTS AT MOST 4 Name x SUCH THAT ist(x,frau)
PROPOSITION 3: EXISTS EXACTLY 2 Name x SUCH THAT ist(x,mann)
AND von(x,venus)
PROPOSITION 4: von(arq,venus) IMPLIES (von(bok,venus) AND von(fip,venus))
PROPOSITION 5: von(eor,mars) AND ist(arq,frau)
PROPOSITION 6: ist(gus,frau) AND NOT von(arq,mars)
PROPOSITION 7: ist(bok,mann) IFF (ist(dru,frau) OR ist(gus,frau))
PROPOSITION 8: von(cep,mars) IMPLIES von(gus,mars)
PROPOSITION 9: ist(dru,mann) AND ist(eor,mann) AND NOT ist(dru,mars)
AND NOT ist(eor,mars)
PROPOSITION 10: FOR ALL Name x HOLDS ist(x,mann) XOR ist(x,frau)
PROPOSITION 11: FOR ALL Name x HOLDS von(x,mars) XOR von(x,venus)
```

`SET` definiert die Sorten `Name`, `Planet` und `Geschlecht` und zugleich auch die Konstanten `arq,bok,cep,dru,eor,fip,gus`, `mars,venus` und `mann,frau`.
`RELATION` definiert die Prädikate `von` und `ist` und legt fest, dass sie jeweils zweistellige Prädikate sind.
Die ersten drei Propositionen enthalten allesamt anzahlbeschränkte Existenzquantoren und besagen, dass zwischen drei und vier Mitgliedern Frauen sein müssen und genau zwei Mitglieder von der Venus stammende Männer sind.

### Yices-Code

```
(define-type Name (scalar ark bok cep dru eor fip gus))
(define-type Planet (scalar mars venus))
(define-type Geschlecht (scalar mann frau))

(define von :: (-> Name Planet bool))
(define ist :: (-> Name Geschlecht bool))

(assert(let ((formula (lambda (x::Name)(ist x frau)))) (exists (x1::Name
x2::Name x3::Name)(and (/= x1 x2) (/= x1 x3) (/= x2 x3) (formula x1)
(formula x2) (formula x3)))))
(assert(let((formula(lambda(x::Name)(ist x frau))))(not(exists (x1::Name
x2::Name ... x5::Name)(and (/= x1 x2)(/= x1 x3)(/= x1 x4)(/= x1 x5)
(/= x2 x3)(/= x2 x4)(/= x2 x5)(/= x3 x4)(/= x3 x5)(/= x4 x5)
(formula x1)(formula x2)(formula x3)(formula x4)(formula x5))))))
(assert(and(let((formula(lambda(x::Name)(and(ist x mann)(von x venus)))))
(exists(x1::Name x2::Name)(and(/= x1 x2)(formula x1)(formula x2))))
(let((formula(lambda(x::Name)(and(ist x mann)(von x venus)))))
(not(exists(x1::Name x2::Name x3::Name)(and(/= x1 x2)(/= x1 x3)
(/= x2 x3)(formula x1)(formula x2)(formula x3)))))))
```

```
(assert (=> (von ark venus) (and (von bok venus) (von fip venus))))
(assert (and (von eor mars) (ist ark frau)))
(assert (and (ist gus frau) (not (von ark mars))))
(assert (and (=> (ist bok mann)(or (ist dru frau)(ist gus frau)))
 (=> (or (ist dru frau)(ist gus frau)) (ist bok mann))))
(assert (=> (von cep mars) (von gus mars)))
(assert (and (and (and (ist dru mann) (ist eor mann))
 (not(ist dru mars))) (not (ist eor mars))))
(assert (forall (x::Name) (and (or (ist x mann)(ist x frau))
 (not (and (ist x mann) (ist x frau))))))
(assert (forall (x::Name) (and (or (von x mars) (von x venus))
 (not (and (von x mars) (von x venus))))))

(check)
```

### Auswertung

```
==========================================
            Erfuellbarkeit + Modell:
==========================================
unknown
(= (ist fip frau) true)
(= (ist gus frau) true)
(= (ist ark frau) true)
(= (ist bok mann) true)
(= (von bok venus) true)
(= (ist cep mann) true)
(= (von cep venus) true)
(= (von ark venus) true)
(= (von fip venus) true)
(= (von eor mars) true)
(= (ist dru mann) true)
(= (ist eor mann) true)
(= (von gus venus) true)
(= (von dru mars) true)
```

Arq, Fip und Gus sind Venusfrauen, Bok und Cep Venusmänner und Dru und Eor Marsmänner.

Die Laufzeit ist mit 1,02 Sekunden mehr als hundertfach so groß, wie die der vorangegangen Rätsel, der Speicherplatzverbrauch ist mit 9MB hingegen nur doppelt so groß.

### 6.1.4 Wolfswürmer

Dieses Rätsel ist eine Logelei des Online-Magazins Zeit-Online.

#### Beschreibung des Rätsels

| |
|---|
| Tom, ein Biologiestudent, sitzt verzweifelt in der Klausur, denn er hat vergessen das Kapitel über Wolfswürmer zu lernen. Das einzige, was er weiß ist, dass er bei den Multiple-Choice-Aufgaben immer genau drei korrekte Antworten geben muss. Nun liegt folgende Antwortmöglichkeiten vor ihm: |
| a) Wolfswürmer werden oft von Igelwürmern gefressen.<br>b) Wolfswürmer meiden die Gesellschaft von Eselswürmern.<br>c) Wolfswürmer ernähren sich von Lammwürmern.<br>d) Wolfswürmer leben in der banesischen Tundra.<br>e) Wolfswürmer gehören zur Gattung der Hundswürmer.<br>f) Wolfswürmer sind grau gestreift.<br><br>Durch die Antwortmöglichkeiten g) bis l) kann er durch logisches Denken an die Lösung kommen:<br><br>g) Genau eine der beiden Aussagen b) und e) sind richtig.<br>h) Genau eine der beiden Aussagen a) und d) sind richtig.<br>i) Genau eine der beiden Aussagen c) und h) sind richtig.<br>j) Genau eine der beiden Aussagen f) und i) sind richtig.<br>k) Genau eine der beiden Aussagen c) und d) sind richtig.<br>l) Genau eine der beiden Aussagen d) und h) sind richtig.<br><br>Er weiß: Es gibt genau drei korrekte Antworten.<br>Können sie Tom helfen, die richtigen Antworten herauszufinden? |

Tabelle 6.3: Wolfswürmer-Rätsel

#### Eingabe

```
SET: Aufgabe:= {a,b,c,d,e,f,g,h,i,j,k,l}
SET: Loesung:= {richtig,falsch}
```

```
RELATION: ist SUBSET (Aufgabe X Loesung)

PROPOSITION 1: ist(g,richtig) IFF (ist(b,richtig) XOR ist(e,richtig))
PROPOSITION 2: ist(h,richtig) IFF (ist(a,richtig) XOR ist(d,richtig))
PROPOSITION 3: ist(i,richtig) IFF (ist(c,richtig) XOR ist(h,richtig))
PROPOSITION 4: ist(j,richtig) IFF (ist(f,richtig) XOR ist(i,richtig))
PROPOSITION 5: ist(k,richtig) IFF (ist(c,richtig) XOR ist(d,richtig))
PROPOSITION 6: ist(l,richtig) IFF (ist(d,richtig) XOR ist(h,richtig))
PROPOSITION 7: EXISTS EXACTLY 3 Aufgabe x SUCH THAT ist(x,richtig)
```

Die Sorten `Aufgabe` und `Loesung` werden mit `SET` definiert und genauso wie die Konstanten `a,b,c,d,e,f,g,h,i,j,k,l` und `richtig,falsch`.
`RELATION` definiert das zweistellige Prädikat `ist` bestehend aus einer Konstante der Sorte `Aufgabe` und einer Konstante der Sorte `Loesung`.
Die ersten sechs Propositionen enthalten allesamt einen Äquivalenz- und einen Exklusive-Oder-Operator. Die letzte Proposition beinhaltet einen anzahlbeschränkten Existenzquantor und besagt, dass genau drei Antwortmöglichkeiten korrekt sein müssen.

### Yices-Code

```
(define-type Aufgabe (scalar a b c d e f g h i j k l))
(define-type Loesung (scalar richtig falsch))

(define ist :: (-> Aufgabe Loesung bool))

(assert (and (=> (ist g richtig) (and (or (ist b richtig) (ist e richtig))
  (not (and (ist b richtig) (ist e richtig))))) (=> (and (or (ist b richtig)
  (ist e richtig))(not(and(ist b richtig)(ist e richtig))))(ist g richtig))))
(assert (and (=> (ist h richtig) (and (or (ist a richtig) (ist d richtig))
  (not (and (ist a richtig) (ist d richtig))))) (=> (and (or (ist a richtig)
  (ist d richtig))(not(and(ist a richtig)(ist d richtig))))(ist h richtig))))
(assert (and (=> (ist i richtig) (and (or (ist c richtig) (ist h richtig))
  (not (and (ist c richtig) (ist h richtig))))) (=> (and (or (ist c richtig)
  (ist h richtig))(not(and(ist c richtig)(ist h richtig))))(ist i richtig))))
(assert (and (=> (ist j richtig) (and (or (ist f richtig) (ist i richtig))
  (not (and (ist f richtig) (ist i richtig))))) (=> (and (or (ist f richtig)
  (ist i richtig))(not(and(ist f richtig)(ist i richtig))))(ist j richtig))))
(assert (and (=> (ist k richtig) (and (or (ist c richtig) (ist d richtig))
  (not (and (ist c richtig) (ist d richtig))))) (=> (and (or (ist c richtig)
  (ist d richtig))(not(and(ist c richtig)(ist d richtig))))(ist k richtig))))
(assert (and (=> (ist l richtig) (and (or (ist d richtig) (ist h richtig))
  (not (and (ist d richtig) (ist h richtig))))) (=> (and (or (ist d richtig)
  (ist h richtig))(not(and(ist d richtig)(ist h richtig))))(ist l richtig))))
```

```
(assert (and (let ((formula (lambda (x::Aufgabe)(ist x richtig))))
  (exists (x1::Aufgabe x2::Aufgabe x3::Aufgabe) (and (/= x1 x2) (/= x1 x3)
  (/= x2 x3) (formula x1) (formula x2) (formula x3)))) (let ((formula
  (lambda (x::Aufgabe)(ist x richtig)))) (not (exists (x1::Aufgabe
  x2::Aufgabe x3::Aufgabe x4::Aufgabe) (and (/= x1 x2) (/= x1 x3) (/= x1 x4)
  (/= x2 x3) (/= x2 x4) (/= x3 x4) (formula x1) (formula x2) (formula x3)
  (formula x4)))))))

(check)
```

**Auswertung**

```
==========================================
Erfuellbarkeit + Modell:
==========================================
unknown
(= (ist g richtig) false)
(= (ist b richtig) false)
(= (ist e richtig) false)
(= (ist h richtig) true)
(= (ist a richtig) false)
(= (ist d richtig) true)
(= (ist i richtig) false)
(= (ist c richtig) true)
(= (ist j richtig) false)
(= (ist f richtig) false)
(= (ist k richtig) false)
(= (ist l richtig) false)

==========================================
```

Somit muss Tom die Antworten c, d und h ankreuzen, um die volle Punktzahl für diese Aufgabe zu erhalten.

Diese Rätsel hat im Vergleich zu den anderen Rätseln mit 24,688 Sekunden mit großen Abstand die längste Laufzeit und auch der Speicherverbrauch ist mit 10,9MB am größten.

### 6.1.5 Magisches Quadrat

Dieses magische Quadrat ist ein 5×5-Quadrat, bei dem die Summe jeder Zeile, jeder Spalte und jeder Diagonale die Zahl 65 ergeben muss.

#### Beschreibung

| | | | | |
|---|---|---|---|---|
| | 24 | | 8 | 15 |
| | 5 | 7 | 14 | |
| 4 | 6 | | 20 | |
| 10 | | | | 3 |
| 11 | 18 | 25 | | |

Tabelle 6.4: Magisches Quadrat

#### Eingabe

```
SET: Quadrat := {zahlen}

RELATION: magisches SUBSET Quadrat

PROPOSITION 1:
EXISTS A Int a SUCH THAT (EXISTS A Int b SUCH THAT a + 24 + b + 8 + 15 = 65)
PROPOSITION 2:
EXISTS A Int c SUCH THAT (EXISTS A Int d SUCH THAT c + 5 + 7 + 14 + d = 65)
PROPOSITION 3:
EXISTS A Int e SUCH THAT (EXISTS A Int f SUCH THAT 4 + 6 + e + 20 + f = 65)
...
PROPOSITION 12:
EXISTS A Int g SUCH THAT (EXISTS A Int e SUCH THAT 11 + g + e + 14 + 15 = 65)
```

#### Yices-Code

```
(define-type Quadrat (zahlen))
(define magisches :: (-> Quadrat bool))

(assert (exists (a::Int)(exists (b::Int)(=(+(+(+(+ a 24) b) 8) 15) 65))))
(assert (exists (c::Int)(exists (d::Int)(=(+(+(+(+ c 5) 7) 14) d) 65))))
(assert (exists (e::Int)(exists (f::Int)(=(+(+(+(+ 4 6) e) 20) f) 65))))
(assert (exists (g::Int)(exists (h::Int)
  (exists (i::Int)(= (+(+(+(+ 10 g) h) i) 3) 65)))))
(assert (exists (j::Int)(exists (k::Int)(=(+(+(+(+ 11 18) 25) j) k) 65))))
(assert (exists (a::Int)(exists (c::Int)(=(+(+(+(+ a c) 4) 10) 11) 65))))
```

```
(assert (exists (g::Int)(=(+(+(+(+ 24 5) 6) g) 18) 65)))
(assert (exists (b::Int)(exists (e::Int)
  (exists (h::Int)(=(+(+(+(+ b 7) e) h) 25) 65)))))
(assert (exists (i::Int)(exists (j::Int)(=(+(+(+(+ 8 14) 20) i) j) 65))))
(assert (exists (d::Int)(exists (f::Int)
  (exists (k::Int)(=(+(+(+(+ 15 d) f) 3) k) 65)))))
(assert (exists (a::Int)(exists (e::Int)
  (exists (i::Int)(exists (k::Int)(=(+(+(+(+ a 5) e) i) k) 65))))))
(assert (exists (g::Int)(exists (e::Int)(=(+(+(+(+ 11 g) e) 14) 15) 65))))

(check)
```

### Auswertung

```
==========================================
Erfuellbarkeit + Modell:
==========================================
sat
(= a 17)
(= b 1)
(= c 23)
(= d 16)
(= e 13)
(= f 22)
(= g 12)
(= h 19)
(= i 21)
(= j 2)
(= k 9)
==========================================
```

Anhand dieser Werte ergibt sich folgende Lösung für das magische Quadrat:

| | | | | |
|---|---|---|---|---|
| 17 | 24 | 1 | 8 | 15 |
| 23 | 5 | 7 | 14 | 16 |
| 4 | 6 | 13 | 20 | 22 |
| 10 | 12 | 19 | 21 | 3 |
| 11 | 18 | 25 | 2 | 9 |

Tabelle 6.5: Magisches Quadrat

Die Lösung dieses magischen Quadrates hat eine Laufzeit von 0,08 Sekunden und einen Speicherverbrauch von 4,5MB.

## 6.2 Vergleiche und Fazit aller Tests

In Tabelle 6.5 ist eine Übersicht aller Laufzeiten und des benötigten Speicherplatzes von verschiedenen Rätseln mit verschiedenen Lösungsansätzen dargestellt. Auf der Seite http://www.ki.informatik.uni-frankfurt.de/master/programme/logicals-smt/ ist eine Vielzahl weiterer Rätsel nachzuschlagen.
Zwei der Lösungsansätze stammen aus ([Dem15]), bei denen der SAT-Solver SAT4J benutzt wurde. Beim ersten Ansatz (ohne intelligente Benutzerschnittstelle IBS) wurden die Logikrätsel direkt an das Kernprogramm übergeben. Beim zweiten Ansatz wurden die Rätsel zunächst als benutzerfreundliche Eingabe angegeben (mit intelligenter Benutzerschnittstelle IBS).
Die zwei anderen Lösungsansätze sind die beiden SMT-Solver Yices1 und Yices2, wobei Yices2 nicht für alle Rätsel Modelle liefern kann, da die Existenz- und Allquantoren nicht vollständig unterstützt werden. Ein solcher Fall ist in der Tabelle durch n.l. (nicht lösbar) gekennzeichnet. Bei einigen Rätseln, die Quantoren enthalten, wurde diese Quantoren im Code für Yices2 durch eine äquivalente Umformung von logischen Operatoren ersetzt, damit diese Rätsel zu lösen sind.

### 6.2.1 Analyse der Ergebnisse

Bei der Analyse der Ergebnisse werden zunächst einzelne Logikrätsel bezüglich ihrer Performance zwischen den SAT- und SMT-Solvern verglichen und danach werden die verschiedenen Rätsel miteinander verglichen.

Betrachten wir das *Bombenrätsel*, so sind die Ergebnisse des SAT-Solvers SAT4J unabhängig davon, ob sie mit oder ohne der intelligenten Benutzerschnittstelle berechnet wurden, fast gleich. Genauso verhält es sich mit den beiden SMT-Solvern Yices1 und Yices2. Vergleicht man jedoch den SAT- mit den SMT-Solvern, so fällt auf, dass letztere eine kürzere Laufzeit haben als die Varianten des SAT-Solvers. Bezüglich des Speicherverbrauchs sieht das anders aus. Dieser ist bei den SMT-Solvern vierfach so hoch, wie bei SAT4J.
Betrachten wir nun das *Einsteinrätsel*, so fallen erhebliche Unterschiede auf. Die Laufzeit bei SAT4J mit Benutzung der intelligenten Benutzerschnittstelle beträgt knapp 7,5 Minuten, wohingegen Yices dafür noch nicht einmal eine Sekunde braucht. Das kann daran liegen, dass Yices nur ein Modell sucht und sobald es eins gefunden hat, dieses ausgibt, und SAT4J hingegen alle möglichen Modelle berechnen will und nicht abbricht, wenn eins gefunden wurde. Ein weiterer Grund ist, dass SAT-Solver Formeln der Rätsellogik zuerst in die Aussagenlogik transformieren müssen, was zu einer deutlichen Steigerung der Klauselmengen führen kann. Auch hinsichtlich des Speicherplatzverbrauchs gibt es Unterschiede. Mit 106MB ist er beim SAT-Solver fast 20-fach so hoch wie beim SMT-Solver. Vergleicht man die Werte mit denen des Bombenrätsels, so haben sich diese bei Yices kaum erhöht (von 4MB auf 5,8MB). Bei SAT4J haben sich die Werte hingegen mehr als verhundertfacht (von 1MB auf 106MB).
Das Rätsel, das mit großem Abstand die meisten Kapazitäten verbraucht, ist das *Rätsel der Wolfswürmer*. SAT4J braucht 14199 Sekunden, also 3 Stunden 35 Minuten und 39 Sekunden

und verbraucht 404MB, wohingegen Yices 25 Sekunden braucht und 10,9MB Speicherplatz verwendet. Im Vergleich zu den anderen Rätseln ist die Diskrepanz der Laufzeiten, sowohl beim SAT-Solver als auch beim SMT-Solver, immens.
Das Wolfswürmer-Rätsel (Abschnitt 6.1.4) besteht aus sieben Propositionen, wobei sechs davon jeweils eine Äuivalenz und eine Antivalenz (XOR-Verknüpfung) enthalten und die letzte Proposition einen anzahlbeschränkten Existenzquantor. Da auch andere Rätsel diese Quantoren enthalten, liegt hierbei wohl nicht das Problem. Das Problem liegt vielmehr bei der Menge der auftretenden Äquivalenz- und Antivalenz-Verknüpfungen, die zu einer deutlichen Erhöhung der Klauselmengen führen, wodurch die Laufzeit stark ansteigt.
Auch Zahlenrätsel sind mit den SMT-Solvern schnell lösbar. Die Laufzeit für das Lösen des magischen Quadrats beträgt 0,08 Sekunden und der Speicherverbrauch ist mit 4,5MB ähnlich niedrig, wie der der anderen Rätsel. Sat-Solver sind nicht in der Lage solche Zahlenrätsel zu lösen.

Allgemein lässt sich feststellen, dass das Programm korrekt und genau wie erwartet läuft. Die Tests gaben immer die richtigen Lösungen aus und die Fehlermeldungen für die Falschtests waren ebenfalls korrekt. Der verwendete Speicherplatz liegt mit 4MB für das kleinste und 10,9MB für das größte Rätsel auf einem konstanten Level. Vergleicht man den Speicherplatzverbrauch zwischen SAT- und SMT-Solvern fällt auf, dass er bei kleineren Rätseln bei SAT4J geringer ist und bei größeren Rätseln bei Yices1 und Yices2. Bei der Laufzeit ergeben sich für Yices1 erhebliche Unterschiede. Auf der einen Seite braucht Yices1 für das „Aristoteles“-Rätsel nur 0,001 Sekunden und auf der anderen Seite für das „Wolfswürmer“-Rätsel mit 24,688 Sekunden ein Vielfaches der Laufzeit.

| Rätsel | Laufzeit Speicher | SAT Ohne IBS | Solver Mit IBS | SMT Yices | SMT Yices2 |
|---|---|---|---|---|---|
| **Aristoteles** | Laufzeit | 0,49s | 0,56s | 0,001s | 0,001s |
| | Speicher | 1MB | 1MB | 4,41MB | 4,8MB |
| **Bombe** | Laufzeit | 0,61s | 0,67s | 0,003s | 0,003s |
| | Speicher | 1MB | 1MB | 4,4MB | 4,9MB |
| **Bombe2** | Laufzeit | 3,9s | 4,3s | 0,003s | 0,004s |
| | Speicher | 1MB | 1MB | 4,4MB | 4,9MB |
| **Ehepaare** | Laufzeit | 5,9s | 8,84s | 0,277s | n.l. |
| | Speicher | 16MB | 21MB | 6,16MB | n.l. |
| **Einstein** | Laufzeit | 197s | 444s | 0,235s | n.l. |
| | Speicher | 93MB | 106MB | 5,8MB | n.l. |
| **Freunde** | Laufzeit | 0,62s | 0,83s | 0,009s | 0,01s |
| | Speicher | 1MB | 1MB | 4,5MB | 4,8MB |
| **Geburtstag** | Laufzeit | 1,9s | 2,6s | 0,012s | 0,031s |
| | Speicher | 3MB | 3,9MB | 4,5MB | 4,9MB |
| **Magisches Quadrat** | Laufzeit | n.l. | n.l. | 0,08s | n.l. |
| | Speicher | n.l. | n.l. | 4,5MB | n.l. |
| **Planet** | Laufzeit | 0,43s | 0,59s | 1,02s | n.l. |
| | Speicher | 5,1MB | 7MB | 9MB | n.l. |
| **Knusel** | Laufzeit | 9,2s | 11,4s | 0,06s | n.l. |
| | Speicher | 13MB | 18MB | 4,97MB | n.l. |
| **Knusien** | Laufzeit | 0,68s | 0,93s | 0,01s | 0,013s |
| | Speicher | 1MB | 2MB | 4,5MB | 4,8MB |
| **Oona** | Laufzeit | 1,99s | 2,31s | 0,01s | 0,031s |
| | Speicher | 1,7MB | 2MB | 4,5MB | 4,9MB |
| **Salz** | Laufzeit | 0,54 | 0,66s | 0,01s | n.l. |
| | Speicher | 1MB | 1MB | 4,5MB | n.l. |
| **Schrumbien** | Laufzeit | 0,61s | 0,77s | 0,026s | 0,046s |
| | Speicher | 5,2MB | 7MB | 4,5MB | 4,8MB |
| **Schwimmer** | Laufzeit | 0,53s | 0,66s | 0,009s | 0,01s |
| | Speicher | 1MB | 1MB | 4,5MB | 4,8MB |
| **Wolfswürmer** | Laufzeit | 11637s | 14199s | 24,688s | n.l. |
| | Speicher | 351MB | 404MB | 10,9MB | n.l. |

Tabelle 6.6: Übersicht

# 7 Zusammenfassung und Fazit

Das letzte Kapitel dieser Arbeit ist in drei Abschnitt unterteilt. Der erste Abschnitt fasst die Schritte und den Schwerpunkt der Arbeit zusammen. Im zweiten wird ein Fazit aus der Implementierung und den Tests gezogen. Im letzten Abschnitt wird ein Ausblick auf weitere Forschungen im Gebiet der SMT-Solver gegeben.

## 7.1 Zusammenfassung

Der Kernpunkt dieser Arbeit ist das Lösen von Logikrätseln unter Verwendung der SMT-Solver Yices1 und Yices2 und der Implementierung eines Parsers, der möglichst natürlichsprachliche Eingaben von Rätseln in die Logik eines solchen genannten Solvers transformiert.
Im ersten Kapitel wurde die Motivation für diese Arbeit erläutert und anschließend wurden die Grundlagen für das Verständnis dieser Arbeit in Kapitel 2 eingeführt.
Zuerst ging es in diesem Kapitel um den Begriff der Logik im allgemeinen, gefolgt von Beispielen von Logikrätseln, wie dem Aristoteles-Rätsel und dem magischen Quadrat.
Daraufhin wurde die Aussagenlogik in Bezug auf ihre Syntax und Semantik definiert, um im nächsten Abschnitt das Erfüllbarkeitsproblem und die dafür zugeschnittenen SAT-Solver zu erklären.
Die nächste Stufe der Aussagenlogik ist die Prädikatenlogik. Auch bei dieser Logik wurde deren Syntax und Semantik betrachtet. Im letzten Abschnitt dieses Kapitels ging es um die „Satisfiability modulo Theories“ und die Funktionsweise der dafür zugeschnittenen SMT-Solver.

In Kapitel 3 wurden mit Yices ein spezielle SMT-Solver eingeführt. Dazu wurden dessen Logik, Architektur, Eingabesprache und die Benutzung des Tools an sich beschrieben und Beispiele aufgeführt, die deren Funktionsweisen skizzieren sollten.

Mit den Grundlagen der Aussagen- und Prädikatenlogik war es möglich die sogenannte „Rätsellogik“ in Kapitel 4 einzuführen, die eine auf endliche Mengen eingeschränkte und um Sorten erweiterte Prädikatenlogik darstellt. Auch zu der Rätsellogik wurden ihre Syntax und ihre Semantik definiert, sowie die anzahlbeschränkten Existenzquantoren, die Ausdrücke der Form „es gibt mindestens / höchstens / genau n...“ ermöglichen. Außerdem wurden die Typregeln der Formeln der Rätsellogik definiert.

In Kapitel 5 ging es um die Implementierung des Parsers. Begonnen wurde mit der Funktionsweise von Parsern und dem Parsergenerator Happy. Darauf folgten die Definition der Benutzereingabe eines Logikrätsels und der Typcheck.

In Abschnitt 5.4 wurde die Implementierung des Parsers vorgestellt, gefolgt von der Umwandlung der Rätseleingabe in die Eingabesprache von Yices in Abschnitt 5.5. Im letzten Abschnitt wurde diese Umwandlung genauestens getestet.

Abschließend wurde im letzten Kapitel die Implementierung ausgiebig getestet, um die Funktionsweise experimentell zu überprüfen. Dazu wurden einige Rätsel ausgewählt, beschrieben, deren Eingaben angegeben, deren Yices-Code dargestellt und deren Lösungsmodelle angegeben. Außerdem wurden die Laufzeit und der Speicherverbrauch betrachtet und daraus Schlüsse gezogen.
Im zweiten Abschnitt wurden Vergleiche zwischen dem SAT-Solver SAT4J und den SMT-Solvern hinsichtlich dieser Rätsel in Bezug auf deren Performance gezogen.

## 7.2 Fazit

Nach ausführlichem Testen hat sich ergeben, dass Yices ein SMT-Solver ist, der besonders schnell und effizient Logikrätsel lösen kann, bei denen die Formeln ausschließlich aussagenlogische Junktoren enthalten. Die Laufzeit beträgt dabei meistens nur wenige Millisekunden. Doch dies unterscheidet ihn noch nicht von gewöhnlichen SAT-Solvern, die aussagenlogische Probleme ähnlich schnell lösen können. Im Gegensatz zu solchen Solvern sind Yices und auch andere SMT-Solver in der Lage auch prädikatenlogische Formeln auf ihre Erfüllbarkeit hin zu untersuchen. Dadurch ist es möglich, Logikrätsel aufzustellen, die sowohl Existenz- als auch Allquantoren enthalten und diese zu lösen. Dafür eignet sich vor allem der erste Yices-SMT-Solver. Bei allen getesteten Logikrätseln hat er ein Lösungsmodell ausgegeben, obwohl er teilweise beim Überprüfen der Erfüllbarkeit „unknown“ zurückgegeben hat. Diese Meldung erscheint immer dann, wenn der Solver innerhalb einer festgelegten Anzahl von Iterationen nicht terminiert und es fehlschlägt zu zeigen, dass eine Formel unerfüllbar ist. Doch trotzdem erstellt Yices ein potenzielles Lösungsmodell der Formeln, wobei der Nutzer nicht weiß, ob dieses Modell wirklich alle Formeln erfüllt.
Der SMT-Solver Yices2 hat dieses Problem der „unknown“-Ausgabe nicht. Sobald dieser herausfindet, ob ein Rätsel (un-)lösbar ist, gibt er „sat“ oder „unsat“ zurück. Leider hat Yices2 gegenüber Yices1 erhebliche Quantorenprobleme. Rätselformeln, die einen Existenzquantor enthalten, sind mit ihm kaum lösbar. Zwar hat er einen speziellen Modus für solche Formeln ($--mode = ef$), doch löst dieser nur Formeln die Allquantoren enthalten und keine mit Existenzquantoren.
So lässt sich schlussfolgern, dass Yices1, trotz des Problem der „unknown“-Ausgabe für Logikrätsel, die Quantoren enthalten nützlicher ist als Yices2. So kann er beispielsweise auch für Logikrätsel, die anzahlbeschränkte Existenzquantoren enthalten, Lösungsmodelle generieren.

Die Formeln, die in den Logikrätsel benutzt werden, beruhen auf einer speziell für diese Arbeit entwickelten Prädikatenlogik, der „Rätsellogik“. Diese enthält neben dem normalen, drei weitere Existenzquantoren, mit denen die Anzahl der Konstanten eingeschränkt werden kann. Es ist damit möglich, Aussagen wie „es existieren mindestens n“, „es existieren höchstens n“ und „es existieren genau n“ Konstanten in Rätseln aufzustellen. Yices1 ist im

Gegensatz zu Yices2 in der Lage auch für Logikrätsel, die solche anzahlbeschränkten Existenzquantoren enthalten, Lösungsmodelle zu generieren.

Vergleicht man SAT- und SMT-Solver im Allgemeinen, so haben letztere deutlich mehr Möglichkeiten. Sie können neben der Lösung von aussagen-, prädikaten- und rätsellogischen Problemen auch verschiedene Hintergrundtheorien miteinander kombinieren und die entstehenden Formeln auf Erfüllbarkeit hin überprüfen. So können sie beispielsweise arithmetische Operatoren mit logischen Operatoren verknüpfen. Dadurch war es uns möglich unsere Rätsellogik um einige arithmetische Operatoren zu erweitern. Diese konnten dazu genutzt werden, Zahlenrätsel wie das „Magische Quadrat“ zu lösen. Auch viele andere Zahlenrätsel, wie Sudokus oder Kakuros, kann man damit lösen.

Dadurch, dass bei jedem unserer Testfälle aus Kapitel 6 ein Lösungsmodell erstellt wird, lässt sich schlussfolgern, dass der in dieser Arbeit implementierte Parser korrekt arbeitet. Er transformiert erfolgreich, in einer Textdatei geschriebene Logikrätsel in die Eingabesprache von Yices, wodurch es erst möglich ist die Logikrätsel zu lösen.

Die intelligente Benutzerschnittstelle erleichtert es dem Normalbenutzer Logikrätsel in Form der Rätsellogik aufzustellen. Auch bei fehlerhaften Eingaben funktioniert sie und fängt potenzielle Fehler mit Fehlermeldungen ab.

In Bezug auf die Testfälle ließ sich beobachten, dass mit 4MB für kleinere und mit knapp 11MB für größere Rätsel der benötigte Speicherplatz auf einem konstanten Niveau ist. Bei den Laufzeitanalyse gab es Ergebnisse die zwischen einer Millisekunde und 25 Sekunden lagen, was einer Erhöhung um das 25000-fache entspricht. Es hat sich ergeben, dass dies vor allem an häufig auftauchenden von Äquivalenz- und Kontravalenzverknüpfungen lag. Die Laufzeit von 25 Sekunden für das „Wolsfwürmer“-Rätsel (Kapitel 6.1.4) ist aber trotzdem noch im Rahmen, vor allem, wenn man diese Laufzeit mit der eines SAT-Solvers vergleicht. SAT4J braucht nämlich für die Lösung dieses Rätsels mehr als 3,5 Stunden. Diese hohe Laufzeit liegt daran, dass SAT-Solver im Allgemeinen nur aussagenlogische Formeln akzeptieren und somit die Rätsellogik erst in die Aussagenlogik transformiert werden muss, wodurch um das Vielfache größere Klauselmengen entstehen können, wie bei SMT-Solvern.

Insgesamt lässt sich aus dieser Arbeit der Schluss ziehen, dass SMT-Solver ein gutes Werkzeug sind um effizient und schnell Logikrätsel zu lösen. Sie haben einige Vorteile gegenüber SAT-Solvern, es gibt aber trotzdem in einigen Punkten noch Verbesserungsbedarf.

## 7.3 Ausblick

Es fallen in Bezug auf SMT-Solver auch einige Punkte auf, die noch ausbaufähig wären und im Rahmen einer weiteren Arbeit umgesetzt werden könnten.

Ein im Fazit bereits besprochenes Problem ist das Problem der „unknown“-Ausgabe. Wenn Yices nach einer bestimmten Zeit nicht herausfindet, ob ein Modell unerfüllbar ist, kommt es zu jener Ausgabe. Trotzdem erstellt es noch ein Modell, welches zwar häufig richtig ist, aber theoretisch auch falsch sein kann. Um die Erfüllbarkeit zu verifizieren, sollte man das Modell zusätzlich zu den ursprünglichen Formeln des Rätsels nochmals von dem SMT-Solver überprüfen lassen. Sollte Yices dann „unsat“ ausgeben, so wäre das zuerst ausgegebene Modell falsch. Gibt Yices „sat“ aus, so wäre das Modell zumindest ein mögliches Lösungsmodell.

Auch für die intelligente Benutzerschnittstelle, die für eine möglichst einfache, natürlichsprachliche Eingabe eines Rätsels zuständig ist, gibt es einige Verbesserungsmöglichkeiten. Zum einen könnte man neben der infix-Verwendung von Prädikaten auch die präfix-, bzw. postfix-Verwendung von Prädikaten als weitere Eingabemöglichkeiten von Formeln einbauen. Zum anderen könnte man die Festlegung von relationalen Eigenschaften der Prädikate, wie beispielsweise Reflexivität, Transitivität, Injektivität, Surjektivität, einbauen.
Außerdem könnte man die Fehlererkennung und -ausgabe erweitern. Dazu gehören beispielsweise eine genaue Positionsangabe des Fehlers oder die Erkennung von Tippfehlern.

Diese genannten und weitere Verbesserungspunkte könnte man als Ausgangspunkt für weiterführende Arbeiten, die SMT-Solver verwenden, nehmen.

# Literaturverzeichnis


[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard version 2.5. *http://smtlib.cs.uiowa.edu/*, 20:15–118, 2015.

[Dem15] Aybike Demirsan. Automatisches lösen von logikrätseln unter verwendung von sat-solvern mit einer intelligenten benutzungsschnittstelle. *http://www.ki.informatik.uni-frankfurt.de/master/programme/logicals/*, 2015.

[DM06a] Leonardo De Moura. Yices 1.0: An efficient smt solver. *http://yices.csl.sri.com/old/slides-smtcomp-06.pdf*, 2006.

[DM06b] Leonardo De Moura. Yices 1.0: An efficient smt solver. *http://yices.csl.sri.com/old/slides-afm-tutorial.pdf*, 2006.

[Dut06] Bruno Dutertre. Yices manual version 2.3. *http://yices.csl.sri.com/papers/manual.pdf*, 2006.

[GS03] Günther Görz and Josef Schneeberger. *Handbuch der künstlichen Intelligenz* -. Oldenbourg Verlag, München, korrigierte auflage edition, 2003.

[GSD14] Adria Gascon, Pramod Subramanyan, and Bruno Dutertre. *Formal Methods in Computer-Aided Design (FMCAD 2014)*. Springer, Berlin, Heidelberg, 2014.

[Hau10] Christian Haubelt. *Digitale Hardware/Software-Systeme - Spezifikation und Verifikation*. Springer-Verlag, Berlin Heidelberg New York, 2010.

[KSB08] Daniel Kroening, Ofer Strichman, and R.E. Bryant. *Decision Procedures - An Algorithmic Point of View*. Springer-Verlag, Berlin Heidelberg, 2008.

[Let13] Theodor Lettmann. *Aussagenlogik: Deduktion und Algorithmen - Deduktion und Algorithmen*. Springer-Verlag, Berlin Heidelberg New York, 2013.

[Lug05] George Luger. *Artificial Intelligence - Structures and Strategies for Complex Problem Solving*. Pearson Education, Amsterdam, 5 sub edition, 2005.

[Mar10] Simon Marlow. Parserpenerator happy. *https://www.haskell.org/happy/*, 2010.

[NO78] G. Nelson and D. C. Oppen. *Simplification by Cooperating Decision Procedures* -. Defense Technical Information Center, New York, 1978.

[SB91] Raymond Smullyan and Thea Brandt. *Logik-Ritter und andere Schurken - diabolische Rätsel, interplanetarische Verwicklungen und Gödelsche Systeme*. Fischer-Taschenbuch-Verlag, Frankfurt am Main, 1991.

[UJ12] Schöning Uwe and Torán Jacobo. *Das Erfüllbarkeitsproblem SAT - Algorithmen und Analysen*. Lehmanns Media, Berlin, 2012.

[Uni89] Rutgers University. Dimacs. *http://dimacs.rutgers.edu/*, 1989.

[Wag15a] Karl H. Wagner. Aussagenlogik fb 10 universität bremen. *http://www.fb10.uni-bremen.de/khwagner/grundkurs2/kapitel3.aspx*, 2015.

[Wag15b] Karl H. Wagner. Prädikatenlogik fb 10 universität bremen. *http://www.fb10.uni-bremen.de/khwagner/grundkurs2/kapitel4.aspx*, 2015.

[WIK15] WIKIPEDIA. Syllogistik. *https://de.wikipedia.org/wiki/Syllogismus*, 2015.