Hochschule
München
University of
Applied Sciences

Fakultät für Informatik und Mathematik

Bachelorarbeit

Mehr entscheiden und kombinieren

-

Ein Haskell SMT Solver

Prüfer: Prof. Dr. Matthias Güdemann

Vorgelegt von:
Florian Obermaier, Matrikelnummer: 49039020
Wittelsbacherplatz 15, 82256 Fürstenfeldbruck
f.obermaier@hm.edu
B.Sc. Informatik

Abgabe:
München, 17.03.2024

Inhaltsverzeichnis

1 Einleitung

> Die kürzesten Wörter, nämlich „ja“ und „nein“, erfordern das meiste Nachdenken.
> *Pythagoras von Samos (570 - 500 v. Chr.) zugesprochen*[1]

Entscheidungen zu treffen, der Frage nachzugehen, ob etwas wahr oder falsch ist bzw. die Frage „Ist diese Aussage wahr?“ mit „ja“ oder „nein“ zu beantworten, beschäftigt die Menschen, wie man am Zitat erkennen kann, seit jeher. Bemerkenswert ist, dass schon Pythagoras von Samos mit der eindeutigen Zustimmung „ja“ bzw. der Ablehnung „nein“ den Fall einer ausweichenden Antwort „vielleicht“, mit der die Kenntnislosigkeit oder Unsicherheit der antwortenden Person kaschiert werden soll und die uns heute allerorten begegnet, unerwähnt lässt. Dies zeigt, wie stark die antiken Griechen bereits dem mathematischen Denken bzw. der Logik verhaftet waren, Bereichen, in denen es nur eine absolute Richtigkeit bei der Antwort auf eine Entscheidungsfrage gibt.

In der Mathematik lassen sich aus Entscheidungsfragen Aussagen formulieren, die im mathematischen Sinn überprüft werden können. „Ich bin in München und ich bin nicht in München.“ ist eine solche Aussage, die sich - unabhängig davon, dass sie offensichtlich falsch ist - umschreiben lässt als:

Beispiel 1.1.

$$A \wedge \neg A$$

Dabei ist A die Aussage „Ich bin in München“ und „¬“ bewirkt die Negierung der Aussage.

Die Entscheidungsfindung oder besser gesagt die Beantwortung der Frage „Ist diese Aussage wahr?“ mit Hilfe der Informatik zu erleichtern, ist die Aufgabe dieser Bachelorarbeit. Will man Aussagen der Art wie in Beispiel 1.1 auf ihren Wahrheitsgehalt hin überprüfen, so kann man unter anderem einen Algorithmus anwenden, der in sogenannten *Satisfiability Solvern*[2] (kurz: SAT-Solver) implementiert ist. Dabei wird eine Lösung durch systematisches Probieren gesucht. Wird eine gefunden, ist die Aussage erfüllbar, ansonsten unerfüllbar.

Besteht die gesamte Aussage aus mehreren Teilaussagen, also mathematischen Konstrukten, die entweder wahr oder falsch sein können, so müssen zusätzlich zur Überprüfung, ob das aussagenlogische Grundgerüst der Aussage wahr ist, noch die mathematischen Konstrukte untereinander auf Konsistenz überprüft werden. Einen Algorithmus, der zu dieser Konsistenzprüfung fähig ist, nennt man *Entscheidungsprozedur*. Zusammen mit dem SAT-Solver bildet die Entscheidungsprozedur einen *Satisfiability Modulo Theories Solver* (kurz: SMT-Solver), der die gesamte Aussage entscheiden kann (siehe Kapitel 2.2).

Zu diesen mathematischen Konstrukten, die im Folgenden *Theorien* genannt werden, ist die Gleichheit und Ungleichheit von Variablen und uninterpretierten Funktionen (kurz: EUF-Logik) zu zählen (siehe Kapitel 3.1). Anhand des folgenden Beispiels soll eine Aussage für diese Theorie exemplarisch dargestellt werden.

Beispiel 1.2.

$$(x = y) \vee (x \neq y)$$

Für das Beispiel 1.2 kann der SAT-Solver drei Lösungen liefern, nämlich $\{(x = y) \mapsto \text{TRUE}, (x \neq y) \mapsto \text{FALSE}\}$, $\{(x = y) \mapsto \text{FALSE}, (x \neq y) \mapsto \text{TRUE}\}$ und $\{(x = y) \mapsto \text{TRUE}, (x \neq y) \mapsto \text{TRUE}\}$, wobei nur zwei davon auch in der EUF-Logik wahr sind. Denn der Fall, dass sowohl die Aussage $x = y$ als auch die Aussage $x \neq y$ wahr sind, ist in der Theorie nicht konsistent, da x nicht

[1] https://www.aphorismen.de/zitat/6920
[2] Alle in der Einleitung vorkommenden Begriffe werden in den kommenden Kapitel genau definiert und erläutert.

gleichzeitig gleich und ungleich y sein kann. Im SMT-Solver werden deswegen die Antworten, die ein SAT-Solver liefert, nochmal überprüft, ob sie auch innerhalb der verwendeten Theorie, hier die der EUF-Logik, eine Lösung sind.

Da der SAT- , der SMT-Solver für die EUF-Logik und somit auch deren Entscheidungsprozedur bereits implementiert wurden, wird in der Arbeit eine weitere Theorie, die der Differenzenlogik, erst theoretisch eingeführt und anschließend in der Programmiersprache Haskell implementiert (siehe Kapitel 4.2.1). Somit können Aussagen der Form $x - y \leq 4$ formuliert werden. Um Aussagen der Differenzenlogik auch entscheiden zu können, ist es ein weiteres Ziel, den SMT-Solver um eine Entscheidungsprozedur für die Differenzenlogik so zu erweitern, dass dieser automatisch erkennt, aus welcher Theorie eine Aussage ist, und diese auf ihre Richtigkeit überprüfen kann.

Zusätzlich zu den Aussagen der EUF-Logik sind somit auch Aussagen der Differenzenlogik entscheidbar. Doch es besteht zunächst weiterhin eine Einschränkung, nämlich die Tatsache, dass man bei der Formulierung einer Aussage die Theorien nicht „vermischen“ darf. Das heißt die gesamte Aussage darf bei der Lösung mit dem SMT-Solver nur Teilaussagen aus der einen oder der anderen Theorie beinhalten. Diese Arbeit macht es jedoch möglich, durch eine spezielle Prozedur, die Nelson-Oppen-Kombinationsprozedur, diese Einschränkung aufzuheben und das nur unter Nutzung der implementierten Entscheidungsprozeduren des SMT-Solvers (siehe Kapitel 4.3).

2 Theoretische Grundlagen

Zunächst werden in der Arbeit oft benutzte Begriffe der zugrundeliegenden Theorie der Logik und der Entscheidungsprozeduren definiert, damit deren Verwendung im Folgenden klar ersichtlich ist. Im Anschluss daran wird die grundsätzliche Funktionsweise eines SAT- und eines SMT-Solvers erklärt, da beide Solver in der Implementierung (Kapitel 4) einen wichtigen Bestandteil bilden. Des Weiteren werden in den Kapiteln 4.2 und 4.3 die Theorie zur Differenzenlogik und deren Entscheidungsprozedur, sowie die Theorie zur Nelson-Oppen-Kombinationsprozedur behandelt.

2.1 Begriffsklärung: Logik und Entscheidungsprozeduren

Eine **Formel** in der Aussagenlogik besteht aus der Verknüpfung von sogenannten **Atomen** mit den Logikoperatoren $\neg$ (Negation), $\vee$ (Disjunktion), $\wedge$ (Konjunktion), $\Longrightarrow$ (Implikation), $\Longleftrightarrow$ (Äquivalenz). Dabei können die beiden letzteren durch die ersten drei dargestellt werden:

$$\begin{aligned} A \Rightarrow B &= \neg A \vee B \\ A \Leftrightarrow B &= (A \Rightarrow B) \wedge (B \Rightarrow A) \\ &= (\neg A \vee B) \wedge (A \vee \neg B) \end{aligned}$$

Definition 2.1 (Formel)**.** *Sei eine Menge von Atomen gegeben, dann kann eine Formel in der Aussagenlogik durch die folgenden Regeln definiert werden:*

Formel: *Formel* $\wedge$ *Formel* | *Formel* $\vee$ *Formel* | $\neg$*Formel* | *(Formel)* | *Atom*

Atom: *Boolsche Variable* | *TRUE* | *FALSE*

[vgl. KS16, S.27]

Atome sind also die kleinstmöglichen Formeln der jeweiligen Theorie. Im Falle der Aussagenlogik, für die die obige Definition gilt, sind die Atome die Wahrheitswerte TRUE und FALSE oder Variablen, die für diese stehen. Im Falle der EUF-Logik sind die Atome z.B. von der Form $x = y$

oder $x \neq y$, in der Differenzenlogik z.B. $x - y < 3$. Eng zusammenhängend damit sind die Literale:

Definition 2.2 (Literal)**.** *Ein Literal ist ein Atom oder ein negiertes Atom. Ersteres wird als positives Literal bezeichnet, letzteres als negatives [vgl. KS16, S.8].*

Als nächstes wird die Semantik einer Formel in der Aussagenlogik geklärt, d.h. wann eine Formel als TRUE oder FALSE ausgewertet wird. Dazu dient der Begriff der **Belegung**:

Definition 2.3 (Belegung)**.** *Eine Belegung ist eine Abbildung α, die jedes Atom einer Formel φ auf einen Wahrheitswert aus der Menge {TRUE, FALSE} abbildet [vgl. EP02, S.4].*

Zu dieser Definition findet man auf Seite 4 ein Beispiel (siehe Bsp. 2.1). Unter der Belegung für eine Formel φ versteht man den Wahrheitswert die gesamte Formel. Der Wahrheitswert der Belegung für eine Formel kann mit Wahrheitstafeln und mit als bekannt vorausgesetzten Regeln für die Logikoperatoren bestimmt werden. Ist der Wahrheitswert für die Belegung von φ TRUE, bezeichnet man die Belegung als ein **Modell**, man schreibt $\alpha \models \varphi$. Mit diesen Begriffen können nun Formeln in drei Kategorien eingeteilt werden:

Definition 2.4 (erfüllbar, unerfüllbar, gültig)**.** *Besitzt eine Formel mindestens ein Modell, dann ist sie **erfüllbar**. Ist das nicht der Fall wird sie **unerfüllbar** genannt. Tritt der Fall ein, dass jede Belegung ein Modell ist, so handelt es sich um eine **gültige** Formel. Man spricht auch von einer Tautologie [vgl. EP02, S.39].*

Um den Umgang mit Formeln, die Logikoperatoren enthalten, zu vereinfachen gibt es die sogenannten Normalformen, die eine bestimmte syntaktische Struktur vorgeben. Dabei ist wichtig, dass die ursprüngliche Formel und die daraus abgeleitete Formel in Normalform **äquivalent** sind.

Definition 2.5 (Äquivalenz von Formeln)**.** *Zwei Formeln φ, ψ sind äquivalent, wenn für alle Belegungen α gilt:*

$$\alpha \models \varphi \iff \alpha \models \psi$$

[vgl. EP02, S.42]

Sind in beiden Formeln teilweise unterschiedliche Atome enthalten, so spielt das für die Äquivalenz keine Rolle, solange obige Bedingung erfüllt ist. Ein einfaches Beispiel für die Äquivalenz sind die Formeln x und $x \wedge (y \vee \neg y)$. Diese Bemerkung ist im weiteren Verlauf wichtig, nämlich für die Tseitin-Transformation, da hier Hilfsvariablen eingeführt werden, die in der ursprünglichen Formel nicht vorkommen.

Eine der in der Arbeit verwendeten Normalformen ist die **konjunktive Normalform** (kurz: KNF), die man durch die Tseitin-Transformation erlangen kann.

Definition 2.6 (Konjunktive Normalform)**.** *Eine Formel φ ist in konjunktiver Normalform genau dann, wenn es sich um eine Konjunktion von Disjunktionen von Literalen L_{ij} handelt. In mathematischer Schreibweise:*

$$\varphi = \bigwedge_i (\bigvee_j L_{ij})$$

[vgl. KS16, S.12]

Vereinfachend wird eine Disjunktion von Literalen als **Klausel** bezeichnet.

Bei der zuvor genannten **Tseitin-Transformation** soll auf eine exakte Definition verzichtet und die Vorgehensweise nur umrissen werden. Schreibt man eine aussagenlogische Formel als Syntaxbaum entsprechend ihrer Präzedenz auf, d.h. jeder Logikoperator ist ein Knoten, während die Atome die Blätter sind, so ist die Wurzel der letzte Operator, den man ausführt, und darunter liegen jene, die man davor ausführt. Für jeden Knoten, der kein Blatt ist, wird eine Hilfsvariable eingeführt. Anschließend bildet man Äquivalenzen der Hilfsvariablen mit den Formeln, die sich aus

dem Operator, aus dem sie hervorgehen, und den beiden Nachfolgern ergeben. Ist der Nachfolger ein Blatt, handelt es sich um ein Literal, das verwendet werden kann, handelt es sich aber um einen inneren Knoten, so handelt es sich um einen weiteren Operator, der in der Formel durch seine Hilfsvariable dargestellt werden muss. Die sich daraus ergebenden Äquivalenzen werden in KNF umgewandelt und durch Konjunktion zu einer resultierenden Formel zusammengefügt, zusammen mit der Hilfsvariable, die die Wurzel des Syntaxbaums ist [vgl. KS16, S.12f].

2.2 SAT- und SMT-Solver

Wie bereits im Titel und in der Einleitung erwähnt, beschäftigt sich die Arbeit mit einem SMT-Solver. Was das ist und wie er sich von einem SAT-Solver unterscheidet, wird in diesem Abschnitt erklärt. Dazu bezeichnen wir die Menge aller erfüllbaren Formeln als SAT, die aller unerfüllbaren als UNSAT und die der gültigen als TAUT [vgl. EP02, S.41]. Mithilfe dieser Mengen kann man das **Entscheidungsproblem** nun in folgender Frage formulieren: Ist eine Formel $\varphi \in TAUT$? Um diese Frage zu beantworten, kann man einen Algorithmus finden, eine sogenannte **Entscheidungsprozedur**. Existiert eine solche, so nennt man die Theorie entscheidbar [vgl. KS16, S.6f].

Ein **SAT-Solver** löst die Frage, ob eine aussagenlogische Formel $\varphi \in SAT$ ist. Ist φ erfüllbar, dann liefert der SAT-Solver auch gleich eine mögliche Belegung der einzelnen Atome, sodass die Formel wahr ist. Eine mögliche Belegung für eine aussagenlogische Formel φ ist in Beispiel 2.1 dargestellt.

Beispiel 2.1.

$$\begin{aligned} &\varphi = A \wedge (B \vee C) \\ &\quad \big\downarrow \text{SAT-Solver} \\ &\alpha = \{A \mapsto \text{TRUE},\ \ B \mapsto \text{FALSE},\ \ C \mapsto \text{TRUE}\} \end{aligned}$$

Zusammen mit folgendem Satz [vgl. EP02, S.39] kann man mit einem SAT-Solver sogar herausfinden, ob eine Formel φ gültig ist.

Satz 2.1. *Gegeben sei eine Formel φ, dann gilt:*

$$\varphi \in TAUT \iff \neg\varphi \in UNSAT$$

Dazu bildet man die Negation der Formel φ und lässt einen SAT-Solver entscheiden, ob diese negierte Formel unerfüllbar ist. Ist dies der Fall, ist die eigentliche Formel φ gültig.

Der in der Arbeit verwendete SAT-Solver[3] basiert auf dem CDCL-Framework (Conflict-Driven Clause-Learning). Dabei werden nacheinander die Atome beliebig mit Wahrheitswerten belegt, Klauseln, wenn es möglich ist, vereinfacht und darauf untersucht, ob es einen Konflikt für die gewählte Belegung gibt. Gibt es am Ende eine Belegung von allen Atomen, ohne dass es einen Konflikt gibt, so ist die Formel erfüllbar. Gibt es dagegen einen Konflikt, ist dies vorerst kein Problem. Es wird ein sogenanntes Backtracking gemacht. Dabei wird die Belegung auf einen Zustand vor dem Konflikt gesetzt und eine Klausel hinzugefügt, die dafür sorgt, dass die unerfüllbare Belegung nicht erneut auftritt. Erst wenn kein Backtracking mehr möglich ist, ist die Formel nicht erfüllbar [vgl. KS16, S.31ff].

Um nun auch Formeln einer beliebigen anderen Theorie, wie z.B. der Theorie der Gleichheit oder der Differenzenlogik, zu entscheiden, verwendet man einen **SMT-Solver**. Dessen Umsetzung ba-

[3] Quellcode: https://github.com/mgudemann/sat-solver

siert vor allem auf einer Kombination aus SAT-Solver und den Entscheidungsprozeduren der jeweiligen Theorien [vgl. KS16, S.60].

In der Arbeit wird der Begriff der Entscheidungsprozedur nicht immer im engen Sinne eines Algorithmus zur Entscheidung, ob eine Formel gültig ist, benutzt, sondern oft steht er für einen Algorithmus zur Entscheidung, ob eine Formel erfüllbar oder unerfüllbar ist. Allerdings sind diese beiden Verwendungen des Begriffs der Entscheidungsprozedur mithilfe des Satzes 2.1 fast identisch, da auch, wie im vorherigen Abschnitt über die Unerfüllbarkeit der negierten Formel die Gültigkeit der nicht negierten Formel gezeigt werden kann. Desweiteren sind die Entscheidungsprozeduren, die zur Verwendung im SMT-Solver implementiert wurden, nur auf den Fall beschränkt, dass die Atome mit Konjunktionen verbunden sind. Diese Einschränkung kann aufgrund der Verwendung der Entscheidungsprozeduren innerhalb des SMT-Solvers gemacht werden, da nach der Durchführung des SAT-Solvers die von der Entscheidungsprozedur zu prüfende Formel nur in dieser Form vorliegt (siehe Ende des ersten Schrittes auf Seite 6) .

Um den Begriff der **Theorie** zu definieren, ist es hilfreich zuerst die **Prädikatenlogik** zu erläutern. Dazu wird in einem ersten Schritt auf die Syntax und in einem zweiten Schritt auf die Semantik eingegangen. Die Bestandteile der Syntax einer Prädikatenlogik lassen sich in drei Bereiche einteilen [vgl. Sch13, S.32f].

1. Logische Symbole: Dazu zählen Variablen, Junktoren ($\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$) und Quantoren ($\exists, \forall$).
2. Nichtlogische Symbole: Diese werden in der **Signatur** Σ, bestehend aus Funktionssymbolen (dazu gehören auch Konstanten) und Prädikatensymbolen, zusammengefasst.
3. Regeln der Syntax (zusätzlich zu denen der Aussagenlogik in Definition 2.1)

Die Semantik der Prädikatenlogik bezieht sich, neben den schon aus der Aussagenlogik bekannten Regeln für die Junktoren, auf die richtige Interpretation der nichtlogischen Symbole Σ. In der **Struktur** fasst man diese Interpretation zusammen mit dem Grundbereich D und der Belegung der Variablen mit Werten aus D [vgl. KS16, S.15].

Unter einer Theorie versteht man letztendlich eine Prädikatenlogik, wobei jeweils besondere Beschränkungen gelten für die Signatur und die Struktur.

Beispiel 2.2. Am Beispiel der Theorie der Gleichheit ist dies:

- Signatur $\Sigma = \{=, \neq\}$
- Struktur:
  - Grundbereich $D = \mathbb{Z}$
  - Interpretationen von Σ:
    - Für das Prädikat "=": Sind die beiden Atome gleich, liefere TRUE und vice versa.
    - Für das Prädikat "$\neq$": Sind die beiden Atome gleich, liefere FALSE und vice versa.
  - Die Belegung der Variablen erfolgt mit Werten aus $\mathbb{Z}$, falls die Formel erfüllbar ist.

Der verwendete SMT-Solver arbeitet in **zwei Schritten**. Diese sind in untenstehendem Vorgangsdiagramm (Abbildung 1) veranschaulicht. In beiden ist der boolsche Encoder e ein wichtiger Bestandteil. Er bildet eine Formel einer Theorie T mit Signatur Σ auf eine aussagenlogische Formel ab, indem er jedes Atom der Formel, das nun auch Symbole aus der Signatur Σ enthalten kann, auf eine boolsche Variable abbildet [vgl. KS16, S.61f]. Sei φ ein Element aus der Menge aller Formeln der Theorie T mit der Signatur Σ (also $\varphi \in Form_\Sigma$), $a_i(\varphi)$ das i-te Atom der Formel φ und $\circ_i$ der

i-te Logikoperator, dann gilt:

$$\begin{aligned} e(\varphi) &= e(a_0(\varphi) \ \circ_0 \ a_1(\varphi) \ \circ_1 ... \circ_{n-1} \ a_n(\varphi)) \\ &= e(a_0(\varphi)) \ \circ_0 \ e(a_1(\varphi)) \ \circ_1 ... \circ_{n-1} \ e(a_n(\varphi)) \\ &\text{mit } e(\varphi) \in Form_{AL} \end{aligned}$$

$Form_{AL}$ ist dabei die Menge aller Formeln der Aussagenlogik.

Im **ersten Schritt** kann diese neu gewonnene Formel $e(\varphi)$, die dem aussagenlogischen Grundgerüst von φ entspricht, nun mit einem SAT-Solver gelöst werden. Es können zwei Fälle auftreten:

1. Liefert der SAT-Solver als Ergebnis, dass die Formel unerfüllbar ist, so ist φ das auch, unabhängig davon, was der nächste Schritt liefert, und der SMT-Solver kann als Ergebnis die Unerfüllbarkeit der Formel φ ausgeben.

2. Ist die Formel erfüllbar, so erhält man eine Belegung α, deren Elemente die Zuordnung der von e enkodierten Atome a_i zu Wahrheitswerten sind.

Man konstruiert nun aus der Belegung eine Formel ψ, die aus der Konjunktion dieser Atome besteht, wobei je nach zugeordnetem Wahrheitswert wie folgt verfahren wird: Bei TRUE erscheint das Atom in seiner ursprünglichen Form in der Konjunktion, bei FALSE in seiner negierten Form.

In einem **zweiten Schritt** wird die Formel ψ in der Entscheidungsprozedur der Theorie gelöst. Auf diese Weise findet man heraus, ob die Formel ψ, die aus der vom SAT-Solver gelieferten Belegung gebildet wird, in der Theorie erfüllbar ist oder nicht. Dabei können wiederum zwei Fälle auftreten:

1. Im ersten Fall ist die neu konstruierte Formel ψ erfüllbar, dann ist die Formel in ursprünglicher Form φ auch erfüllbar.

2. Im zweiten Fall ist ψ nicht erfüllbar. Dies heißt aber noch nicht, dass die Formel φ nicht erfüllbar ist, sondern nur für die vom SAT-Solver gelieferte Belegung. Um bei einer Wiederholung des Verfahrens doch noch eine Lösung für φ zu finden, muss die Belegung, die zu keiner Lösung geführt hat, ausgenommen werden. Dies erreicht man, indem man ψ mit e enkodiert, dann negiert und so eine neue Einschränkung ϵ bildet, die an die Formel $e(\varphi)$ durch Konjuktion angehängt wird. Anschließend kehrt man erneut zum ersten Schritt zurück.[vgl. KS16, S.61ff]

Um die Einschränkung im zweiten Fall besser[4] zu machen, wurde in der Implementierung ein Weg gewählt, der nicht sofort die komplette Formel ψ enkodiert und verneint. Anstatt dessen geht man schrittweise vor. Dazu wird systematisch jeweils nur ein Atom entfernt und anschließend mit der Entscheidungsprozedur überprüft, ob die Formel immer noch UNSAT ist. Wenn das der Fall ist, wird dieser Schritt wiederholt, wenn nicht, wird das entfernte Atom wieder hinzugefügt und ein anderes Atom entfernt. Der Algoritmus endet, wenn kein Atom mehr entfernt werden kann, sodass die Formel bei erneuter Anwendung des Algorithmus nicht wahr wird. Man erhält als Ergebnis eine verkleinerte Formel ψ', die dann enkodiert und verneint werden kann.

[4] Besser heißt hier, dass durch Konjunktion einer negierten Formel mit weniger konjugierten Variablen mehr Belegungen eingeschränkt werden, als mit einer Formel mit mehr Variablen. Z.B. verhindert $\varphi \wedge \neg(A \wedge B \wedge C)$ nur die Belegung $\alpha = \{A \mapsto \text{TRUE}, \ B \mapsto \text{TRUE}, \ C \mapsto \text{TRUE}\}$, wobei $\varphi \wedge \neg(A \wedge B)$ zusätzlich zu α auch noch die Belegung $\{A \mapsto \text{TRUE}, \ B \mapsto \text{TRUE}, \ C \mapsto \text{FALSE}\}$ verhindert.

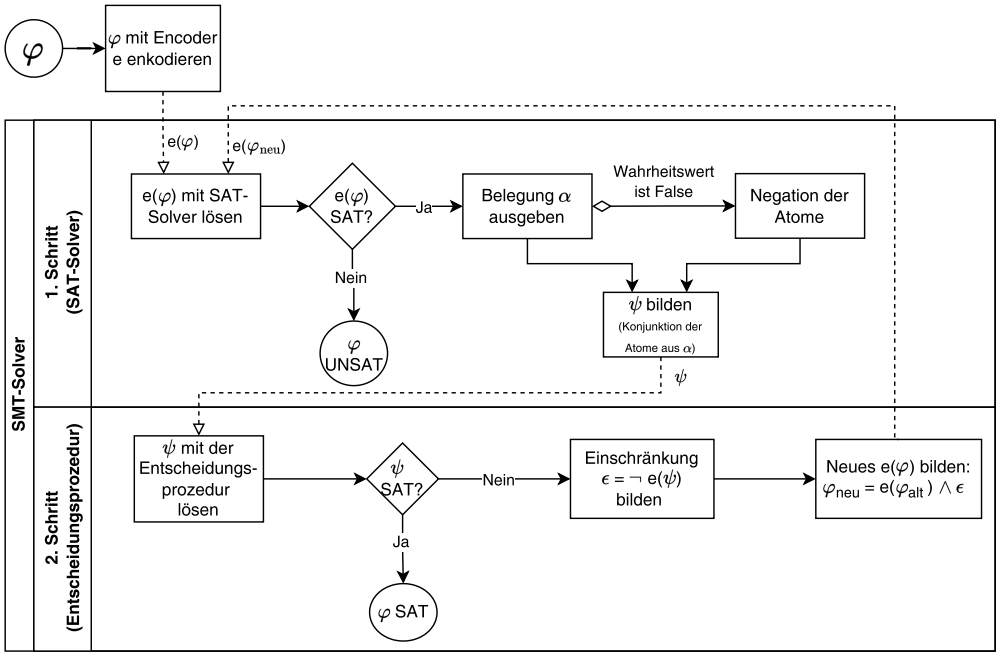

Abbildung 1: Diagramm des SMT-Solvers

3 Ausgangslage

Wie in der Einleitung erwähnt, basiert die Arbeit auf bereits vorhandenem Code, der von Herrn Prof. Dr. Güdemann im Rahmen der Lehrveranstaltung „Theoretische Informatik 2 “ angefertigt wurde. Welche Funktionalitäten bereits implementiert waren und welche Erweiterungen im Rahmen der Bachelorarbeit gemacht wurden ist in den folgenden beiden Unterkapiteln aufgeführt.

3.1 Start: Existierender Code

In den Dateien *EqTerm.hs* und *Equality.hs* ist bereits die EUF-Logik und deren Entscheidungsprozedur mit der Funktion `consistentEq` implementiert, genauso wie der SMT-Solver, basierend auf dem Vorgehen, wie es in Kapitel 2.2 erklärt wurde, mit der Funktion `solveT`.

Die Entscheidungsprozedur für die EUF-Logik wird durch das Bilden von Äquivalenzklassen, die unter Kongruenz abgeschlossen sind, erreicht. Verwendet wird für die Entscheidungsprozedur der englische Begriff **Congruence Closure**. Sei eine Formel

$$\varphi = a_0 \wedge a_1 \wedge ... \wedge a_n \in \text{ EUF} \qquad \text{mit } n \in \mathbb{N}$$

gegeben. Die einzelnen Atome $a_i = t_{2i} \circ t_{2i+1}$ mit $i \in [0, n]$ sind dann entweder Gleichheiten oder Ungleichheiten, d.h. „$\circ$“ $\in$ {„$=$“ , „$\neq$“ } und t_{2i}, t_{2i+1} sind entweder Variablen oder uninterpretierte Funktionen. Um nun die Erfüllbarkeit für φ zu entscheiden, geht man in zwei Schritten vor [vgl. KS16, S.85].

1.Schritt: Man bildet die unter Kongruenz abgeschlossenen Äquivalenzklassen.

- Dazu untersucht man zuerst jedes Atom $a_i = t_{2i} \circ t_{2i+1}$ mit $i \in [0, n]$ und bildet, falls das Prädikatensymbol „$\circ$“ gleich „$=$“ ist, die Äquivalenzklasse $\{t_{2i}, t_{2i+1}\}$. Falls „$\circ$“ gleich „$\neq$“ ist, werden die beiden Äquivalenzklassen $\{t_{2i}\}$ und $\{t_{2i+1}\}$ gebildet.
- Für alle Äquivalenklassen A_j mit $j \in \mathbb{N}$ wird überprüft, ob $A_j \cap A_k$ mit $j \neq k$ eine leere Menge ergibt oder nicht. Ist die leere Menge nicht leer, werden die beiden Äquivalenzklassen zu einer einzigen vereinigt: $A_{jk} = A_j \cup A_k$. Dieser Schritt wird so oft wiederholt, bis keine

Äquivalenzklassen mehr vereinigt werden können.

- Um letztlich die unter Kongruenz abgeschlossenen Äquivalenzklassen zu erhalten, überprüft man, ob es t_i, t_j gibt, die in der gleichen Äquivalenzklasse liegen und zugleich als Argument derselben uninterpretierten Funktion z.B. F vorkommen. Anschließend fügt man die Äquivalensklasse, in der $F(t_i)$ liegt, mit der Äquivalenzklasse, in der $F(t_j)$ vorkommt, zusammen und wiederholt den Vorgang so lange, bis auf diese Weise keine Äquivalenzklassen mehr verknüpft werden können.

2. Schritt: Überprüfe für alle Ungleichheiten $t_i \neq t_j$, die in der Formel φ enthalten sind, ob t_i und t_j Elemente der gleichen Äquivalenzklasse sind. Ist dies für mindestens eine Ungleichheit der Fall, ist die Formel φ unerfüllbar. Trifft das für keine einzige Ungleichheit zu, ist die Formel φ erfüllbar.

Der im SMT-Solver verwendete SAT-Solver ist in der Datei *SAT.hs* vorhanden. Darin ist die Funktionalität in der Funktion `solve` enthalten. Weitere für den SMT-Solver und v.a. auch für die Tseitin-Transformation notwendige Datentypen sind in der Datei *Types.hs* aufgeführt. Die Tseitin-Transformation selbst ist durch die Funktion `tseitin` in der Datei *Tseitin.hs* implementiert, während auch Funktionen aus den Dateien *CNF.hs* genutzt werden, um z.B. eine Formel mit `isCNF` daraufhin zu überprüfen, ob sie in KNF vorliegt.

Ein Teil der genannten Funktionen werden im Unterkapitel 4.2.3 in der Art angepasst, dass sie nicht nur für die EUF-Logik, sondern auch für die Differenzenlogik benutzt werden können. Dort wird auch auf deren Verwendung genauer eingegangen.

3.2 Ziel: Erweiterung um Differenzenlogik und Nelson-Oppen-Kombinationsprozedur

Wie in der Einleitung bereits erwähnt, besteht ein Teil der Arbeit aus der Implementierung einer Entscheidungsprozedur für die Differenzenlogik in der Programmiersprache Haskell. Die Umsetzung davon ist im Kapitel 4.2.2 nachzulesen. Zuerst aber wird in Kapitel 4.2.1 in die Theorie der Differenzenlogik eingeführt. Da sowohl die Entscheidungsprozedur der Differenzenlogik als auch der EUF-Logik auf die gleiche Weise im SMT-Solver benutzt werden, kann man im Code einige Verallgemeinerungen machen, die in Haskell leicht programmiert werden können, sodass redundanter Code vermieden wird (siehe Kapitel 4.2.3).

Mit den beiden Theorien kann man bereits Formeln lösen, die nur aus Atomen einer Theorie bestehen. Weiter kann man aus den Entscheidungsprozeduren dieser beiden Theorien durch die Nelson-Oppen-Kombinationsprozedur eine Prozedur schaffen, die es ermöglicht eine Formel zu entscheiden, die sowohl Atome aus der einen Theorie als auch Atome aus der anderen Theorie enthält und zusätzlich auch kombinierte Atome enthält, z.B. eine uninterpretierte Funktion in einem ansonsten der Differenzenlogik zuordenbaren Atom. Die Funktionsweise und die Implementierung der Nelson-Oppen-Komibinationsprozedur wird im Kapitel 4.3 erläutert, wobei insbesondere auf die drei Bestandteile, die Purification (siehe Kapitel 4.3.1), die Propagation (siehe Kapitel 4.3.2) und das Splitting (siehe Kapitel 4.3.3) eingegangen wird.

4 Implementierung

In diesem Kapitel wird die Implementierungsarbeit, die für die Fertigstellung des SMT-Solvers für die Differenzenlogik und die Entwicklung der Nelson-Oppen-Kombinationsprozedur nötig war, dargestellt. Die Besonderheit dieser Implementierung ist, dass der gesamte Code in der funktionalen Programmiersprache Haskell verfasst wurde. Die sich daraus ergebenden Herausforderungen und Probleme, aber auch deren Lösung, werden in den einzelnen Unterkapiteln erläutert, wobei zuerst auf die Besonderheiten von Haskell eingegangen wird.

4.1 Haskell

Bei Haskell handelt es sich um eine rein funktionale Programmiersprache, d.h. es wird mit Funktionen gearbeitet, deren Ergebnis, wie in der Mathematik, nur von den eingegebenen Werten abhängt und nicht von einem Zustand außerhalb der Funktion. Es gibt somit auch keine Nebeneffekte, wie z.B. die Veränderung eines externen Zustandes. Damit Funktionen dennoch auf einem definierten, durch die Funktion veränderbaren Zustand arbeiten können, ist es möglich den Zustand als Parameter und als Rückgabewert der Funktion zu definieren.

Wieso bietet sich gerade Haskell als Programmiersprache an? Ein großer Vorteil von Haskell ist die Typsicherheit, die dank der statischen Typisierung bereits bei der Kompilierung sichergestellt wird. Um anzugeben, welche Typen eine Funktion als Eingabewerte annimmt und welche sie ausgibt, dient die **Signatur**. Mit zwei Doppelpunkten folgt sie nach dem Funktionsnamen. Arbeitet man in der Signatur nur mit Typvariablen, so spricht man von Polymorphismus, d.h. die Funktion ist für mehrere Typen überladen. Mit sogenannten Typklassen kann diese Typvariable auf Instanzen einer Klasse eingeschränkt werden. Im Folgenden ist die Notation und die Zusammensetzung der Signatur einer Funktion `f` aus Typvariablen und Typklassen zu sehen.

```
f :: (Num a) => a -> a -> a
```

Dabei definiert eine **Klasse** Funktionen, die von ihren **Instanzen** implementiert werden müssen. Dies entspricht der Beziehung zwischen Interface und Implementierung in Java. Zur Verdeutlichung dient der Programmcode 1.

```
1  class F a where
2    f :: a -> String
3
4  instance F Integer where
5    f 1 = "eins"
6    f 2 = "zwei"
7    f _ = "viele"
```

Programmcode 1: Beispiel für Klasse und Instanz

Für die Typvariable kann man **Typen** einsetzen. Dies sind neben den bereits implementierten, wie `String`, `Char`, `Integer`, `Double`, `Bool`, ... auch eigene Datentypen. Man kann diese mit dem Wort *data* erzeugen. Im Programmcode 2 ist das verdeutlicht.

```
1  data DataName a = Constructor1 a
2                  | Constructor2 a Int
3                  | Constructor3
```

Programmcode 2: Beispiel für eigenen Datentyp

Es gibt also Konstruktoren mit und ohne Argumente, wobei die Argumente wiederum parametrisiert sein können. Die Konstruktoren liefern bei Verwendung einen Wert vom jeweiligen Datentyp.

Ein weiteres Konstrukt ist das Currying, was darauf beruht, dass Funktionen in Haskell nicht immer komplett ausgewertet werden müssen. So wird aus einer teilweise ausgewerteten Funktion eine neue Funktion. Siehe dazu Programmcode 3.

```
1 f :: Int -> Int -> Int
2 f a b = a + b
3
4 let g = f 3
5
6 g :: Int -> Int
7 g a = 3 + a
```

Programmcode 3: Beispiel für Currying

4.2 Differenzenlogik und deren Entscheidungsprozedur

4.2.1 Differenzenlogik

Bei der Differenzenlogik handelt es sich um eine Theorie, die nach folgenden Regeln aufgebaut ist [vgl. KS16, S. 126].

$$\textit{Formel}: \quad \textit{Formel} \wedge \textit{Formel} \quad | \quad \textit{Atom} \tag{1}$$

$$\textit{Atom}: \quad \textit{Differenz} \quad \textit{Operator} \quad \textit{Konstante} \tag{2}$$

$$\textit{Differenz}: \quad \textit{Variable} - \textit{Variable} \tag{3}$$

$$\textit{Operator}: \quad \leq \quad | \quad < \tag{4}$$

Die Signatur Σ besteht aus dem Funktionssymbol „$-$“, den beiden Prädikatensymbolen „$<$“ und „$\leq$“ und den Konstanten, die aus $\mathbb{Q}$ oder $\mathbb{Z}$ stammen.

Im Folgenden wird nur der Zahlenbereich $\mathbb{Z}$ verwendet. Diese Einschränkung kann gemacht werden, da jede Zahl aus $\mathbb{Q}$ endlich oder periodisch ist und somit als Bruch dargestellt werden kann. Sind in einem Atom Zahlen aus $\mathbb{Q}$ vorhanden, kann man den Hauptnenner bestimmen, dann für die Berechnungen nur mit den Zählern, die in $\mathbb{Z}$ liegen, fortfahren und die Lösungen zum Schluss wieder durch den Hauptnenner teilen.

Die Struktur ist deshalb gegeben durch:

- den Grundbereich $D = \mathbb{Z}$,
- die Interpretation des Funktionssymbols „$-$“ als die bekannte arithmetische Interpretation des Minuszeichens der ganzen Zahlen,
- die Interpretation der Prädikatensymbole „$<$“ und „$\leq$“ als die entsprechenden Vergleichsoperatoren der ganzen Zahlen und
- die Belegung der Variablen mit Werten aus $\mathbb{Z}$.

Obwohl in den Regeln nur die Operatoren „$<$“ und „$\leq$“ vorkommen, lassen sich damit durch Umformungen auch weitere Vergleichsoperatoren ausdrücken.

Seien x, y Variablen und $c \in \mathbb{Z}$, dann gilt:

$$x - y > c \quad \Leftrightarrow \quad y - x < -c \tag{5}$$
$$x - y \geq c \quad \Leftrightarrow \quad y - x \leq -c \tag{6}$$
$$x - y = c \quad \Leftrightarrow \quad x - y \leq c \quad \wedge \quad y - x \leq -c \tag{7}$$
$$x - y \neq c \quad \Leftrightarrow \quad (x - y < c \quad \vee \quad y - x < -c) \quad \wedge \quad \neg(x - y < c \quad \wedge \quad y - x < -c) \tag{8}$$

Die Umschreibung des Operators „ $\neq$ “ entspricht dem exklusiven Oder:

$$A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$$

Zusätzlich wird in der Arbeit mit folgender Regel der Operator „$<$“ aus der Definition der Differenzenlogik (siehe S. 10, Regel (4)) entfernt.

$$x - y < c \quad \Leftrightarrow \quad x - y \leq c - 1 \tag{9}$$

Dieses Vorgehen wird vor allem in Hinblick auf die später in Kapitel 4.2.2 vorgestellte Entscheidungsprozedur vorgenommen.

Falls die Differenz, d.h. die linke Seite des Atoms, nur aus einer Variable bestehen soll, so kann dies durch folgende Regel unter Hinzufügen einer Hilfsvariable a_{aux} im Code umgesetzt werden:

$$x \leq c \quad \Leftrightarrow \quad x - a_{aux} \leq c, \quad \text{wobei } a_{aux} = 0 \tag{10}$$

Der Datentyp für die Differenzenlogik ist in der Implementierung in der *DL.hs* Datei festgelegt (siehe Programmcode 4). Hierbei kann man klar den Aufbau erkennen, wie er am Anfang dieses Unterkapitels beschrieben wurde.

```
1  data DLVar = DLVar String
2             | DLVarAux
3    deriving (Eq, Ord)
4
5  data DLOp = Strict
6            | NonStrict
7    deriving (Eq, Ord, Show)
8
9  data DLDiff = DLDiff DLVar DLVar
10             | DLMono DLVar
11   deriving (Eq, Ord)
12
13 type DiffVal = Integer
14
15 data DLAtom = DLAtom DLDiff DLOp DiffVal
16   deriving (Eq, Ord)
17
18 type DLTerm = TermT DLAtom
```

Programmcode 4: Datenytp für Differenzenlogik

Eine Besonderheit ist, dass der Datentyp, der aufgrund der Regel (10) zur Darstellung von Differenzenatomen mit einer einzigen Variable eingeführt wurde, mit nur einer Hilfsvariable `DLVarAux`

auskommt (siehe Codezeile 2, Programmcode 4).

Besteht die Formel φ, wobei X die Menge der Variablen in φ darstellt, aus der Konjunktion von n Atomen der Form $x_i \leq c_i$ mit $i \in [1, n]$ und m Atomen der Form $x_j - x_k \leq c_j$ mit $j \in [n+1, m]$ und $k \in [m+1, 2m]$, wobei $x_i, x_j, x_k \in X$ und $c_i, c_j \in \mathbb{Z}$, so benötigt man dennoch nur eine einzige Hilfsvariable, wie im Folgenden gezeigt wird. Es liefere die Entscheidungsprozedur das Modell α mit $\alpha(a_{aux}) = a$ und da in der Regel (10) gefordert wird, dass die Hilfsvariable a_{aux} den Wert Null hat, muss man am Ende der Lösungsprozedur von jeder Belegung den Wert a der Belegung der Hilfsvariable a_{aux} abziehen. D.h. es gilt am Ende für die „bereinigte“ Belegung $\beta(x) = \alpha(x) - a$ für alle $x \in X$ und für die m Atome mit zwei Variablen:

$$\begin{aligned} & \beta(x_i) - \beta(x_j) \leq c \\ \Leftrightarrow \quad & \alpha(x_i) - a - (\alpha(x_j) - a) \leq c \\ \Leftrightarrow \quad & \alpha(x_i) - \alpha(x_j) \leq c \\ \Leftrightarrow \quad & \text{TRUE, da } \alpha \text{ ein Modell ist.} \end{aligned}$$

Für die n Atome mit einer Variablen gilt:

$$\begin{aligned} & \beta(x) \leq c \\ \Leftrightarrow \quad & \alpha(x) - a \leq c \\ \Leftrightarrow \quad & \alpha(x) - \alpha(a_{aux}) \leq c \\ \Leftrightarrow \quad & \text{TRUE, da } \alpha \text{ ein Modell ist.} \end{aligned}$$

Somit ist auch $\beta(x) = \alpha(x) - a$ ein Modell.

Die letzte Zeile des Programmcodes 4 definiert eine Formel mit den Logikoperatoren „$\wedge$“, „$\vee$“, „$\neg$“, „$\Rightarrow$“, „$\Leftrightarrow$“ und `DLAtom` als Atomen. Um zu ermöglichen, dass man Formeln mit den Vergleichsoperatoren „$<$“, „$\leq$“, „$>$“, „$\geq$“, „$=$“, „$\neq$“ in die Entscheidungsprozedur eingeben kann, wurde der Datentyp `GenDLAtom` geschaffen. Eine Formel vom Typ `GenDLTerm (= TermT GenDLAtom)` kann man mit der Funktion `normalizeDL :: GenDLTerm -> DLTerm` in eine Formel vom Typ `DLTerm` umwandeln, d.h. es kommen nur noch die Operatoren „$<$“ und „$\leq$“ vor.

Der nächste Schritt besteht nun darin die neue Datenstruktur in den bereits vorhandenen SMT-Solver, der in *CNFSolver.hs* implementiert ist, zu integrieren, sodass möglichst wenig redundanter Code entsteht. Im bereits existierenden Code war die Signatur der zum SMT-Solver gehörenden Funktionen noch auf die EUF-Logik beschränkt. Um den SMT-Solver auch auf die Differenzenlogik anwendbar zu machen, wird eine eigene Klasse `Atom a` erstellt. Die Instanzen müssen dafür die von der Klasse gefordeten Funktionen

- `negateTerm :: a -> a`,
- `cleanTerms :: [(a, Bool)] -> [(a, Bool)]` und
- `consistent :: [a] -> Bool`

implementieren. Man erkennt, dass die Signaturen der Funktionen den Parameter `a` enthalten. Dieser Parameter wird im Folgenden dafür eingesetzt, um die Funktionen sowohl für die EUF-Logik als auch für die Differenzenlogik zu verwenden. Dies kann man am Beispiel der Funktion

```
solveT :: (Ord a, Show a, Atom a) => TermT a -> Maybe (Valuation a, [a])
```

sehen, die die Funktion des eigentlichen SMT-Solver implementiert. Durch den Kontext `Atom a` ist festgelegt, dass es sich bei `a` um eine Instanz der Klasse `Atom a` handeln muss.

Um herauszufinden, welche Funktionen nur in der Signatur angepasst (Fall 1) und welche Funktionen in einer Instanz auf eine Theorie angepasst werden müssen (Fall 2), ist es das einfachste Vorgehen, sich zu überlegen, welche Funktionen die Atome der Theorie direkt verändern und somit deren Syntax kennen müssen. Im Folgenden wird für jeden Fall jeweils ein Beispiel angeführt:

1.Fall: Die Funktion `normalizeTermBool` ist im Kontext des SMT-Solvers die Funktion, die aus der Lösung des SAT-Solvers, die Atome extrahiert und sie je nach Wahrheitswert negiert oder unverändert lässt. Dies ist der unmittelbare Schritt bevor in Abbildung 1 die Formel ψ gebildet wird, die dann durch die Entscheidungsprozedur gelöst wird. Im Programmcode 5 ist mit `normalizeTermBool` ein Beispiel einer Funktion zu sehen, bei der eine einfache Anpassung der Signatur ausreicht. Hier wird aus `(Equality, Bool) -> Equality` die auch für die Differenzenlogik anwendbare Signatur `Atom a => (a,Bool) -> a`.

```
1 normalizeTermBool :: Atom a => (a, Bool) -> a
2 normalizeTermBool tb =
3   case tb of
4     (t, True)  -> t
5     (t, False) -> negateTerm t
```

Programmcode 5: Einfache Anpassung der Signatur

2. Fall: Die Funktion `negateTerm` wird von `normalizeTermBool` in Programmcode 5 in Zeile 5 dafür benutzt, die eigentliche Negation durchzuführen. Da das Vorgehen bei der Negation in der jeweiligen Theorie unterschiedlich ist und Pattern Matching von Datentypen erfordert, die nur in jeweils einer Theorie vorkommen, gibt es dafür zwei verschiedene Implementierungen `negateDLTerm` und `negateEqTerm`. Die Implementierung mit Pattern Matching ist am Beispiel der Funktion `negateDLTerm` (siehe Programmcode 6) zu sehen. Somit ist für die Logik der Gleichheit, wie auch für die Differenzenlogik, eine eigene Funktion implementiert, deren Implementierung in der Klasse `Atom a` als Funktion `negateTerm` gefordert wird.

```
1 negateDLTerm :: DLAtom -> DLAtom
2 negateDLTerm (DLAtom (DLDiff v1 v2) Strict n) =
3                                         DLAtom (DLDiff v2 v1) NonStrict (-n)
4 negateDLTerm (DLAtom (DLDiff v1 v2) NonStrict n) =
5                                         DLAtom (DLDiff v2 v1) Strict (-n)
```

Programmcode 6: Implementierung von `negateTerm`

Eine Verallgemeinerung wird ebenso vorgenommen bei der Funktion `consistent`, die überprüft, ob eine Liste von Atomen aus der Differenzenlogik, die die einzelnen Bestandteile einer nur aus Konjunktionen bestehenden Formel enthält, erfüllbar oder unerfüllbar ist. Um diese Funktionalität zu implementieren, bedarf es einer Entscheidungsprozedur für die Theorie der Differenzenlogik, deren Theorie und Umsetzung im nächsten Unterkapitel 4.2.2 erläutert wird.

4.2.2 Entscheidungsprozedur basierend auf dem Bellman-Ford-Algorithmus

Im Folgenden werden die Grundlagen der Entscheidungsprozedur für die Konjunktion der Differenzenatome, das sind Prädikate der Form $x_i - x_j < c_k$ oder $x_i - x_j \leq c_k$, beschrieben. Weil die Werte für c nur aus den ganzen Zahlen $\mathbb{Z}$ genommen werden und somit die Regel (9) auf Seite 11 angewendet werden kann, können die Atome mit dem Prädikatensymbol $<$ in solche mit Prädikatensymbol $\leq$ umgewandelt werden.

Die gewählte Entscheidungsprozedur basiert auf einem Graphenalgorithmus, für dessen Erläuterung ein paar Begriffe der Graphentheorie definiert werden müssen.

Definition 4.1 (Graph)**.** *Ein **Graph** $G(V, E)$ wird definiert durch die beiden Mengen V und E. Erstere ist die Menge aller Knoten, letztere die Menge aller Kanten, die eine Verbindung zwischen jeweils zwei Knoten darstellt.*

*Definiert man für die Verbindung eine Richtung, so spricht man von einem **gerichteten** Graphen. Die Menge E ist also eine Menge von Tupeln (v, w), wobei v den Startknoten bezeichnet und w den Endknoten. Man nennt w auch **Nachfolger** von v [vgl. HM22, S.55].*

Die Kanten eines gerichteten Graphen werden im Folgenden **Pfeile** genannt, um sie von den Kanten eines ungerichteten Graphen zu unterscheiden, auf dessen Definition hier verzichtet wird. Ist eine Gewichtsfunktion $c : E \to \mathbb{Z}$ gegeben, die jedem Pfeil ein **Gewicht** zuordnet, so spricht man von einem gewichteten Graphen.

Definition 4.2 (Weg)**.** *Ein **Weg** P ist ein Tupel der Knoten, die von einem Startknoten v_0 aus über Pfeile erreicht werden können. $P = (v_0, v_1, ..., v_r)$ ist ein Weg, wenn $(v_i, v_{i+1}) \in E$ für $i \in [0, ..., r]$ gilt [vgl. HM22, S.56].*

Definition 4.3 (Kreis)**.** *Ein **Kreis** ist ein Weg $P = (v_0, v_1, ..., v_r)$, für den gilt, dass $v_0 = v_r$ [vgl. Cor+09, S.1170].*

Sei $P = (v_0, v_1, ..., v_r)$ ein Weg, dann kann man diesem ein Gewicht zuordnen:

$$c(P) = \sum_{i=0}^{r-1} c(v_i, v_i + 1)$$

Der **kürzeste Weg** von v nach w ist definiert über die Wege, die das Gewicht

$$\delta(u, v) = \begin{cases} min\{\ c(P)\ \mid\ P \text{ ist Weg von u nach v}\} \\ \infty, \quad \text{falls es keinen Weg von u nach v gibt} \end{cases}$$

besitzen [vgl. HM22, S.643]. Mit diesen Begriffen kann man einen **negativen Kreis** definieren als einen Kreis, dessen Gewicht negativ ist.

Definition 4.4 (Graph der Differenzenlogik)**.** *Sei S eine Menge von Differenzenatomen der Form $x_i - x_j \leq c_k$. Dann ist der Graph der Differenzenlogik G(V,A) ein gerichteter, gewichteter Graph mit Pfeilen $(x_j, x_i) \in A$ und der Gewichtsfunktion $c : A \to \mathbb{Z}$ [vgl. KS16, S.128].*

Im Rest der Arbeit wird der Graph der Differenzenlogik mit der Bezeichnung **DL-Graph** abgekürzt.

Beispiel 4.1. Sei folgende Formel gegeben:

$$\varphi = (x_2 - x_1 \leq 4) \wedge (x_3 - x_2 < 2) \wedge (x_1 - x_2 \leq -2),$$

so formt man diese mit Regel (9) um in

$$\varphi = (x_2 - x_1 \leq 4) \wedge (x_3 - x_2 \leq 1) \wedge (x_1 - x_2 \leq -2).$$

Für die Gewichtsfunktion gilt dann

$$c(x_1, x_2) = 4,\ \ c(x_2, x_1) = -2,\ \ c(x_2, x_3) = 1$$

und der resultierende DL-Graph ist in Abbildung 2 dargestellt.

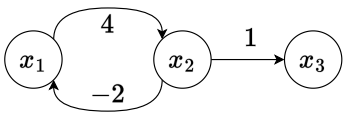

Abbildung 2: Einfaches Beispiel für einen DL-Graphen

Der folgende Satz kann genutzt werden, um mit dem DL-Graphen eine Entscheidungsprozedur für die Differenzenlogik zu konstruieren.

Satz 4.1. *Seien die Formel φ, bestehend aus Konjunktionen von Differenzenatomen, und der daraus resultierende DL-Graph G gegeben. Dann gilt:*

$$\varphi \in SAT \quad \Longleftrightarrow \quad G \text{ besitzt keine negativen Kreise}$$

[vgl. KS16, S.128]

Beweis. Zuerst wird die Richtung „⇐“ bewiesen. Seien also der der Formel φ entsprechende DL-Graph $G(V, A)$ und die Gewichtsfunktion c gegeben. Weiter wird ein Knoten v_0 in G eingefügt, der mit jedem anderen Knoten v_i mit $i \in [1, n]$ durch einen ausgehenden Pfeil (v_0, v_i) mit Kantengewicht $c(v_0, v_i) = 0$ verbunden ist. Da es im Graphen keinen negativen Kreis gibt, gibt es für jeden Knoten $v_i \in V$ und $i \in [1, n]$ einen kürzesten Weg von v_0 mit Gewicht $\delta(v_0, v_i)$. Aufgrund der Dreiecksungleichung ergibt sich, dass für einen beliebigen Pfeil (v_i, v_j), wie in Abbildung 3 dargestellt, gilt:

$$\delta(v_0, v_j) \leq \delta(v_0, v_i) + c(v_i, v_j)$$

Formt man diese Ungleichung um und ersetzt die kürzesten Wege durch die Variablen $x_i := \delta(v_0, v_i)$, so erhält man $x_j - x_i \leq c(v_i, v_j)$. Dies entspricht einem Differenzenatom und die kürzesten Wege $\delta(v_0, v_i)$ und $\delta(v_0, v_j)$ sind eine Belegung, die das Atom erfüllt. Somit ist die Formel φ erfüllbar.

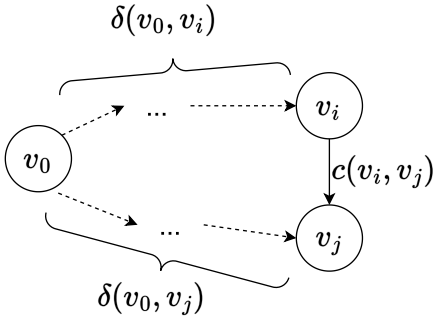

Abbildung 3: Darstellung der Dreiecksungleichung im Graphen

Die andere Richtung „⇒“des Satzes wird bewiesen, indem die Kontraposition „G besitzt einen negativen Kreis $\implies \varphi \in$ UNSAT“ widerlegt wird. Seien ein Graph $G(V, A)$, die Gewichtsfunktion c und der negative Kreis durch den Weg $P = (v_1, v_2, v_3, ..., v_n, v_1)$ gegeben. Wieder wird wie oben ein neuer Knoten v_0 eingefügt, der mit jedem anderen Knoten v_i mit $i \in [1, n]$ durch einen ausgehenden Pfeil (v_0, v_i) mit Kantengewicht $c(v_0, v_i) = 0$ verbunden ist. Zusätzlich wird auch

$x_i := \delta(v_0, v_i)$ mit $i \in [1, n]$ definiert. Dann gilt für die zugehörigen Differenzenatome:

$$
\begin{aligned}
& x_2 - x_1 \leq c(v_1, v_2) \\
& x_3 - x_2 \leq c(v_2, v_3) \\
& \quad \vdots \\
& x_1 - x_n \leq c(v_n, v_1), \text{ wobei } n \in \mathbb{N} \\
\Rightarrow \quad & \underbrace{x_2 - x_1 + x_3 - x_2 + ... + x_1 - x_n}_{=0} \leq \underbrace{c_0 + c_1 + ... + c_n}_{\substack{= \ C<0, \\ \text{da der Kreis negativ ist}}} \\
\Rightarrow \quad & 0 \leq C < 0
\end{aligned}
$$

Die letzte Zeile ist ein Widerspruch und somit ist die Kontraposition widerlegt und die Aussage „$\varphi \in$ SAT $\implies$ G besitzt keine negativen Kreise“ ist bewiesen [vgl. Cor+09, S.667f]. □

Mit einem Graphenalgorithmus, der sowohl negative Kreise erkennt als auch kürzeste Wege berechnet, kann man nun eine Entscheidungsprozedur konstruieren, die nicht nur entscheidet, ob die Formel erfüllbar ist, sondern bei Erfüllbarkeit auch gleich eine wahrmachende Belegung der Variablen liefert. Genau diese beiden erforderlichen Fähigkeiten besitzt der **Bellman-Ford-Algorithmus**. Dieser Algorithmus sucht den kürzesten Weg von einem Startknoten zu allen anderen Knoten, es handelt sich also um einen *Single-Source-Shortest-Path* Algorithmus. Dazu werden zwei Speicher T und P benötigt. Im ersten, nämlich T, werden alle gefundenen Entfernungen für jeden Knoten eines Graphen gespeichert, im zweiten, P, wird jeweils der Vorgänger zu einem Knoten gespeichert. Zum Start des Algorithmus wird ein Startknoten v_0 zum Graphen $G(V, A)$ hinzugefügt, der, wie oben beschrieben, mit jedem anderen Knoten v_i mit $i \in [1, n]$ durch einen ausgehenden Pfeil mit Kantengewicht $c(v_0, v_i) = 0$ verbunden ist. Dann werden beide Speicher initialisiert, alle Einträge p_v mit $v \in V$ in P sind leer, alle Einträge t_v mit $v \in V$ in T werden auf unendlich gesetzt mit Ausnahme des Eintrags des Startknotens. Dieser wird auf 0 gesetzt. Im Anschluss geht man $|V|-1$ mal alle Pfeile $(v_i, v_j) \in A$ durch und führt eine **Entspannung** durch, d.h. man vergleicht den gespeicherten Wert von t_{v_j} mit dem von t_{v_i} addiert mit dem Gewicht $c(v_i, v_j)$. Ist der erste Wert größer, so hat man einen kürzeren Weg (v_0, v_j) gefunden und speichert sowohl den neuen Wert $t_{v_j}^{neu} = t_{v_i} + c(v_i, v_j)$ in T als auch den neuen Vorgänger $p_{v_j} = v_i$ in P. Nachdem der Vorgang $|V|-1$ mal durchgeführt wurde, sind alle Pfeile entspannt und die gespeicherten Werte in T entsprechen den kürzesten Wegen $\delta(s, v_i)$ und die Werte in P den jeweiligen Vorgängern.

Durchläuft man den Algorithmus nun ein weiteres Mal, so sollten sich die Werte für die kürzesten Wege nicht ändern. Sind die Werte aber doch verändert, so weiß man, dass es im Graphen einen negativen Kreis geben muss, da dieser dafür sorgt, dass sich die kürzesten Wege im negativen Kreis bis ins negative Unendliche entspannen.

Der gesamte Bellman-Ford-Algorithmus wird im Pseudocode 1 dargestellt.

Für die konkrete Implementierung[5] in BellmanFord.hs, dargestellt im Programmcode 7, musste zuerst ein Datentyp für den Graphen eingeführt werden, der sich aus den Informationen der verwendeten Knoten als auch der bestehenden Pfeile zwischen diesen Knoten zusammensetzt. Zusätzlich wurde mit dem Parameter bei `Arc a` die Möglichkeit geschaffen für jeden Pfeil ein Gewicht mit beliebigem Datentyp anzugeben. Da in der Arbeit die Differenzenlogik auf $\mathbb{Z}$ definiert wurde (siehe S. 10) und somit auch die Gewichte aus $\mathbb{Z}$ sind, wurde der folgende Datentyp `IWeight` und die Klasse `Weight a` mit der Instanz `Weight IWeight` eingeführt. Dies wird im Programmcode 8

[5] Die konkrete Implementierung folgt zu großen Teilen der Implementierung von Jan van Eijck, der diese für die Vorlesung *Functional Specification of Algorithms: Fall 2014* an der ILLC (Institute for Logic, Language and Computation) bereitgestellt hat. https://staff.fnwi.uva.nl/d.j.n.vaneijck2/courses/14/fsa/lectures/FSA9.hs

Pseudocode 1 Bellman-Ford-Algorithmus

Input: DL-Graph $G(V, A)$, Startknoten $s \in V$ und Gewichtsfunktion c
Output: Menge T aller kürzesten Wege und Menge P aller Vorgänger

1: **procedure** BellmanFord($G(V, A)$, s, c)
2: $T[v] \leftarrow \infty$, $P[v] \leftarrow \{\}$ $\forall v \in V$
3: $T[s] \leftarrow 0$
4: **for** 1 **to** $|V| - 1$ **do**
5: **for all** $(u, v) \in A$ **do**
6: **if** $T[v] > T[u] + c(u, v)$ **then**
7: $T[v] \leftarrow T[u] + c(u, v)$
8: $P[v] \leftarrow u$
9: **return** T, P

```haskell
type Vertex = Integer

data Arc a = Arc Vertex Vertex a
  deriving (Show, Ord)

data Graph a = Graph [Vertex] [Arc a]
  deriving (Show, Eq)
```

Programmcode 7: Graphenbezogene Datentypen

dargestellt. Besonders ist hier die Definition des `IInfinity` Konstruktors, der die Zahl Unendlich darstellt und speziell für den Bellman-Ford-Algorithmus gebraucht wird, aber in Haskells `Integer` nicht vorhanden ist. Er wird in Zeile 2 des Pseudocodes 1 zur Initialisierung der kürzesten Wege genutzt.

```haskell
data IWeight = IWeight Integer
             | IInfinity
            deriving (Ord, Eq, Show)

class Weight a where
  mkZero :: a
  mkInf  :: a

instance Weight IWeight where
  mkZero = IWeight 0
  mkInf  = IInfinity
```

Programmcode 8: Datentyp für Gewichte aus $\mathbb{Z}$

Der eigentliche Bellman-Ford-Algorithmus wird durch die Funktion

```haskell
bf :: Weight a => [Arc a] -> Vertex -> (Vertex -> a, Vertex -> Vertex)
```

implementiert, die wiederum die Funktionen `bfInit` und `bfLoop` benutzt. Wie man an der Signatur erkennen kann, sind die Speicher T und P als Tupel von zwei Funktionen im Code umgesetzt, die im Fall von T einen Knoten `Vertex` auf das Gewicht `a` abbilden und im Fall von P auf einen weiteren Knoten `Vertex`. Die Funktion `bfInit` setzt die Anfangswerte für T auf Unendlich mit `mkInf` bzw. auf Null mit `mkZero`, wie im Pseudocode 1 in Zeile 2 beschrieben. `bfLoop` nutzt die durch `bfInit` initialisierten Funktionen und führt den Schritt in Zeile 5 bis 8 $|V| - 1$ mal aus.

Ob es einen negativen Kreis gibt, wird mit `bfCheck` in einem weiteren Schritt geprüft. Wie im Programmcode 9 in Zeile 3 zu sehen, wird die Funktion `d`, die die Knoten auf die Gewichte der

kürzesten Wege vom Startknoten aus abbildet, benutzt, um zu überprüfen, ob sich die Gewichte der Wege durch einen weiteren Durchgang des in Pseudocode 1 in den Zeilen 5 bis 8 gezeigten Verfahrens nochmal verkleinern. Falls dies der Fall ist, so gibt es einen negativen Kreis und die Funktion liefert `False` zurück.

```
1 bfCheck :: (Num a, Ord a, Weight a) => [Arc a] -> (Vertex -> a) -> Bool
2 bfCheck es d =
3   all (\(Arc u v w) -> d u + w >= d v) es
```

Programmcode 9: Überprüfung eines Graphen auf negative Kreise

Bevor `bf` für eine Entscheidungsprozedur genutzt werden kann, benötigt man noch die Umwandlung der Konjunktion der Differenzenatome, die vom SAT-Solver als Liste von `DLAtom` gelieferte wird, in einen DL-Graphen. Dies wird in *DL.hs* durch die Funktion

```
mkConstraintGraph :: [DLAtom] -> ([Arc IWeight], Bimap.Bimap DLVar Integer)
```

erreicht. Wie man sieht, wird dazu die Konjunktion der Differenzenatome als Liste eben dieser übergeben und es wird ein Tupel zurückgegeben, das aus den Pfeilen des DL-Graphen und einer bidirektionalen Zuordnung `Bimap` besteht. Die `Bimap` bildet die in der ursprünglichen Formel enthaltenen Variablen `DLVar` auf die in den Pfeilen vorkommenden, korrespondierenden Knoten ab, die in der Form von `Integer` gespeichert werden. Letztere Zuordnung wird benötigt, wenn man im Anschluss an den Bellman-Ford-Algorithmus eine wahrmachende Belegung erhalten will. Das Vorgehen dazu wird weiter unten durch die Funktionen `getAssignment` und `getWeights` erläutert.

Der für den Bellman-Ford-Algorithmus nötige Startknoten v_0, der mit allen anderen Knoten mit Pfeilen des Gewichts Null verbunden ist, wird mit der Funktion `insertStartVertex` eingefügt.

Die eigentliche Transformation geschieht in der Funktion `dlToEdge`, in der die Atome in Pfeile umgewandelt werden. Darin ist auch die in der Regel (9) auf Seite 11 genannte Umwandlung von „ $<$ “ zu „ $\leq$ “ implementiert.

Erwähnenswert ist die Funktion `normGraph`, die für den Fall, dass es für zwei Knoten mehrere Pfeile mit unterschiedlichen Gewichten gibt, den Pfeil mit dem Gewicht auswählt, der alle ableitbaren Bedingungen erfüllt und die anderen Pfeile entfernt.

Zusammen mit den oben genannten Funktionen kann man nun, wie auf Seite 13 erwähnt, die Entscheidungsprozedur, wie in Programmcode 10 dargestellt, beschreiben.

```
1 consistentDL :: [DLAtom] -> Bool
2 consistentDL dlList =
3   let
4     graph = fst (mkConstraintGraph dlList)
5   in
6     bfCheck graph (fst (bf graph 0))
```

Programmcode 10: Entscheidungsprozedur der Differenzenlogik

Um eine wahrmachende Belegung zu erhalten, wird in `getAssignment` und somit insbesondere in der Hilfsfunktion `getWeights` der Bellman-Ford-Algorithmus ausgeführt und aus der resultierenden Funktion `d`, die die Knoten auf ihre kürzesten Wege abbildet, werden die jeweiligen Belegungen aller Knoten ausgelesen. Die Knoten werden anschließend wieder auf ihre entsprechenden Variablen aus dem Differenzenatom abgebildet und als Liste von Tupeln `[(DLVar, IWeight)]` zurückgegeben. Will man die Belegung einer Formel φ, so ist eine erneute Überprüfung der Erfüllbarkeit der Formel

nicht erforderlich, da `getAssignment` auf das Ergebnis von der Funktion `solveT` ausgeführt wird und von dieser nur ein nicht leeres Ergebnis geliefert wird, wenn φ erfüllbar ist.

Die Signatur der beiden Funktionen `getAssignment` und `getWeights` enthält als Eingabe das Tupel `(Valuation DLAtom, [DLAtom])`, welches dem Ergebnis der Funktion `solveT` entspricht, die den SMT-Solver implementiert. Die Aufteilung in die zwei Teile des Tupels muss aufgrund der Verwendung der Funktion `CDCL.solve` des SAT-Solvers in `solve` (zu finden in *SAT.hs*) gemacht werden. Denn die Funktion `CDCL.solve` nimmt bei der Entscheidung auf Erfüllbarkeit Optimierungen vor und so kann es vorkommen, dass einige Variablen, die enkodierten Atomen entsprechen (siehe S. 5), wegfallen, da sie sowohl wahr als auch falsch sein können. Darum wird von `solve` sowohl die Belegung der Atome zurückgegeben als auch die weggelassenen Atome als Liste.

Die daraus gewonnene Belegung oder eine beliebige andere Belegung einer Formel φ kann mit der Funktion `evaluate` auf ihre Wahrheit überprüft werden. Erfüllt die Belegung die Formel φ, so gibt die Funktion `True` zurück, `False` anderenfalls. Durch die Möglichkeit mit `getAssignment` Belegungen aus den Lösungen von `solveT` zu gewinnen, die die Formel φ erfüllen sollen, und anschließend mit `evaluate` die Lösungen auf ihre Richtigkeit zu überprüfen, kann man Tests schreiben, die auf dem Prinzip des *property-based-testing* basieren.

Für das property-based-testing wird das Haskell Package *QuickCheck*[6] verwendet. Darin wird eine Spezifikation des zu testenden Programms erstellt, die von diesem erfüllt werden muss. Die Spezifikation wird über sogenannte Eigenschaften (engl. properties) festgelegt. Es handelt sich hierbei um Funktionen, die automatisiert, mit zufälligen Werten aufgerufen werden und einen Wahrheitswert zurückliefern. In der Datei *DLPropComplexSpec.hs* befindet sich solch ein Test, der zufällig erstellte Formeln erst mit `solveT` löst, dann eine Belegung mit `getAssignment` erhält und zu guter Letzt mit `evaluate` überprüft, ob die Belegung auch wirklich die Formel erfüllt. Als Ausgabe liefert der Test eine prozentuale Aufzählung der Fälle, die unerfüllbar sind, und eine der Fälle, die erfüllbar sind, jeweils mit Angabe der Anzahl von Atomen in der Formel.

Die Erstellung der Testfälle erfolgt über Instanzen der Klasse `Arbitrary` von QuickCheck. Die Instanz `Arbitrary GenDLAtom` z.B. erzeugt zufällige Atome der Differenzenlogik, deren Erzeugung in der Funktion `arbitrary` festgelegt werden muss. Dabei können beispielsweise mit den QuickCheck-Funktionen `elements` zufällige Elemente aus einer Liste gewählt werden und mit `choose(i,j)` wird eine Zahl aus einem Zahlenintervall $[i, j]$ gewählt. Obwohl Tests zwar nie die Fehlerfreiheit garantieren, können durch die automatisierte und zufällige Erstellung von Testfällen doch weitaus mehr Fälle abgedeckt werden, die man ansonsten möglicherweise übergangen hätte. Zusätzlich bietet die Klasse `Arbitrary` die Funktion `shrink`, die beschreibt, wie man im Fehlerfall das Objekt verkleinern kann, um herauszufinden, welcher Teil des Objekts tatsächlich zum Fehler geführt hat. Bei einer Formel, die aus mehreren Konjuktionen, Disjunktionen, usw. besteht, kann genau die Teilformel herausgefunden werden, die das Scheitern des Tests verursacht hat.

4.2.3 Weitere Verallgemeinerungen

Durch die Einführung der Klasse `Atom a` auf Seite 12 ist es nun möglich, um redundanten Code zu vermeiden, einige schon bestehende Funktionen zu verallgemeinern. Ein erster Schritt ist die Auslagerung aller verallgemeinerten Funktionen, die von beiden Theorien genutzt werden können, in die Datei *CNFSolver.hs*. Ein wichtiges Beispiel für eine verallgemeinerte Funktion ist die bereits auf Seite 12 erwähnte Funktion `solveT`, die das im Kapitel 2.2 auf Seite 5 beschriebene Vorgehen eines SMT-Solvers implementiert.

Ein weiterer Schritt ist die Einführung eines neuen Datentyps, wie im Programmcode 11 dargestellt.

[6] https://hackage.haskell.org/package/QuickCheck

Mit diesem Datentyp ist es möglich die Funktion `solve`, die die Funktionalität des SAT-Solvers

```
data B = B Int
       | AuxB Int
  deriving(Ord, Eq, Show)
```

Programmcode 11: Neuer Datentyp für die Lösung mit dem SAT-Solver

implementiert, auf die Formeln der Theorien anzuwenden, die Instanzen der Klasse `Atom a` sind. Der Konstruktor `B` dient zur Darstellung eines Atoms in der jeweiligen Theorie als eine Variable `B Int`, wobei die Zahl zur Verwendung des intern verwendeten SAT-Solvers `CDCL.solve` nötig ist. Der SAT-Solver benötigt die Formel in KNF und einem speziellen Format, bei dem die positiven Literale mit positiven Zahlen dargestellt werden, die negativen Literale mit negativen. Sind die Literale in einer Klausel, so sind die Zahlen in einer Liste enthalten. Die Listen untereinander sind in einer weiteren Liste enthalten und entsprechen den Klauseln, die über Konjunktionen verbunden sind. Diese Vorgehen ist in Beispiel 4.2 verdeutlicht.

Beispiel 4.2.

$$(x - y < 4 \ \vee \ \neg(x - z = 3)) \ \wedge \ (z - y > -2)$$
$$\downarrow \texttt{createBimap}$$
$$\texttt{Bimap}: x - y < 4 \leftrightarrow 1, \quad x - z = 3 \leftrightarrow -2, \quad z - y > -2 \leftrightarrow 3$$
$$\downarrow \texttt{convertClauses}$$
$$[[1, -2][3]]$$

Zuerst werden die Atome einer Formel, die einer beliebigen Theorie angehört, durch die Funktion `createBimap` in einer bidirektionalen Map gespeichert. D.h. jedem Atom wird eine Zahl zugeordnet, die in der Form `B Int` gespeichert wird. Im Anschluss sind die von `CDCL.solve` gelieferten Ergebnisse als Liste von Zahlen, die den Wahrheitswerten entsprechen, wieder den Atomen durch die Funktion `getSolution` zuordenbar. Positive Zahlen entsprechen dem Wahrheitswert TRUE, negative dem Wahrheitswert FALSE.

Beispiel 4.2 (Fortsetzung)**.**

$$[[1, -2][3]]$$
$$\downarrow \texttt{CDCL.solve}$$
$$[-1, -2, 3]$$
$$\downarrow \texttt{getSolution}$$
$$x - y < 4 \mapsto \text{FALSE}, \quad x - z = 3 \mapsto \text{FALSE}, \quad z - y > -2 \mapsto \text{TRUE}$$

Wie schon erwähnt muss die Formel in KNF vorliegen. Dazu ist die Anwendung der Tseitin-Transformation notwendig, falls die Formel nicht in KNF vorliegt. Die Hilfsvariablen der Tseitin-Transformation (siehe S. 3) werden im Format `BVar (AuxB i)` erzeugt. Diese Erzeugung erfolgt durch die Funktion `mkAuxVar`, deren Implementierung durch die Klasse `AuxVar a` gefordert wird (siehe Programmcode 12). Da die Tseitin-Transformation erst nach dem Aufruf der Funktion `createBimap` erfolgt, sind die Hilfsvariablen darin nicht gespeichert. Dieses Vorgehen nutzt man, um den Hilfsvariablen bei der Transformation mit der Funktion `convertClauses` in das Format `[[Int]]`, das die Funktion `CDCL.solve` benötigt, Zahlen in einem Zahlenbereich zuzuordnen, der diese eindeutig als Hilfsvariablen identifizierbar macht. Man addiert nämlich auf die Zahl i der

Hilfsvariable `AuxB i` die Mächtigkeit n der vor der Tseitin-Transformation erzeugten `Bimap`[7] und weiß so, dass ab dem Zahlenwert n die Hilfsvariablen beginnen. Am Beispiel 4.2 von vorhin, würden die Hilfsvariablen ab $n = 3$ beginnen. Zum Schluss werden die Zahlen in der Ergebnisliste, geliefert von `CDCL.solve`, mit Hilfe der `Bimap` und der Funktion `getSolution` als Liste von Atom-Wahrheitswert-Paaren zurückgegeben. Die Werte der Hilfsvariablen werden dafür ignoriert und nicht zurückgegeben.

```
1  class AuxVar a where
2   mkAuxVar :: Int -> a
```

Programmcode 12: Klasse der Hilfsvariablen der Tseitin-Transformation

4.3 Nelson-Oppen-Kombinationsprozedur

In den vorherigen Kapiteln wurde die Theorie der Differenzenlogik eingeführt, deren Entscheidungsprozedur und die Implementierung veranschaulicht. Die EUF-Logik und deren Entscheidungsprozedur wurden bereits implementiert (siehe Kapitel 3.1).

Mithilfe der Nelson-Oppen-Kombinationsprozedur, die im Folgenden behandelt wird, kann man nun Formeln lösen, in denen beide Theorien vorkommen. D.h. Variablen und Elemente der Signatur einer Theorie können nun auch mit Variablen und Elementen der Signatur der anderen Theorie verbunden werden. Atome, die auf diese Weise entstehen, werden im Folgenden **kombinierte Atome** genannt und die Formel, in der diese Atome enthalten sind, **kombinierte Formel**.

Im Beispiel 4.3 ist exemplarisch eine kombinierte Formel dargestellt. Als Minuend dient hier im ersten Atom eine uninterpretierte Funktion. Somit ist das Atom bis auf den Minuend der Differenzenlogik zuzuordnen. Das zweite Atom ist, bis auf die Differenz auf der linken Seite, der EUF-Logik zuzuordnen.

Beispiel 4.3.

$$f(x_0) - x_1 < 4 \quad \wedge \quad x_1 - x_2 = 5 \quad \wedge \quad f(x_0) \leq g(x_2)$$

Um eine Entscheidungsprozedur für eine kombinierte Formel aus beiden Theorien zu erhalten, werden die Entscheidungsprozeduren beider Theorien verbunden. Diese Fusion wird in der Nelson-Oppen-Kombinationsprozedur vorgenommen. Damit die Nelson-Oppen-Kombinationsprozedur überhaupt angewendet werden kann, müssen die kombinierten Theorien einige Voraussetzungen erfüllen. Diese Voraussetzungen sind, dass

- die Theorie konvex ist,
- es sich um eine Theorie mit Gleichheit handelt,
- die Signatur der einen Theorie unterschiedlich zu der der anderen Theorie ist (oder aber die gleichen Interpretationen für gleiche Elemente der Signatur besitzen),
- es eine Entscheidungsprozedur gibt und
- der Grundbereich unendlich ist [vgl. KS16, S.229ff].

Bis auf den ersten Punkt - im Gegensatz zur EUF-Logik ist die Differenzenlogik nicht konvex - erfüllen beide Theorien diese Voraussetzungen. Dass die Differenzenlogik nicht konvex ist, hat zur Folge, dass die Nelson-Oppen-Kombinationsprozedur für konvexe Theorien um einen weiteren

[7] Die Mächtigkeit der `Bimap` entspricht der Anzahl der unterschiedlichen Atome in der Formel vor der Tseitin-Transformation.

Schritt[8], nämlich das **Splitting**, ergänzt wird. Der theoretische Hintergrund und das Vorgehen dazu wird im Unterkapitel 4.3.3 behandelt.

Als weitere Bedingung fordert man, dass in der kombinierten Formel nur Konjunktionen als Logikoperatoren vorkommen. Das stellt jedoch kein Problem dar, denn wie beim Vorgehen des SMT-Solvers kann aus einer beliebigen Formel mittels eines SAT-Solvers eine Belegung der Atome der kombinierten Theorie geliefert werden, die dann in eine Formel bestehend nur aus Konjunktionen umgewandelt werden kann (siehe S.5 der Abschnitt am Ende des ersten Schrittes). Aus diesem Grund ist die Eingabe der Funktion, die die eigentliche Nelson-Oppen-Kombinationsprozedur implementiert, nur eine Liste der Atome `NOAtom`, deren Aufbau in folgenden Absätzen und im Programmcode 13 erklärt wird.

Da beide Theorien in einer Formel vorkommen können, muss es einen neuen Datentyp geben, der es ermöglicht Atome der Differenzenlogik, Atome der EUF-Logik und letztlich auch Atome, die Teile aus beiden Theorien enthalten, zu bilden. Diese Teile können Elemente der Signaturen oder Variablen sein. Wie man im Programmcode 13 erkennen kann, kann ein `NOVar` z.B. sowohl eine uninterpretierte Funktion mit einem Parameter sein (`NOFun1 String NOVar`) als auch eine Differenz (`NODiff NOVar NOVar`). Bei beiden Beispielen kann `NOVar` wiederum aus der jeweils anderen Theorie kommen.

Die Signatur wird - wenig überraschend - nur aus Prädikatensymbolen der Differenzenlogik gebildet, da die Symbole aus der EUF-Logik „=“ und „≠“ darin bereits enthalten sind und die gleichen Interpretationen haben.

```
1  data NOVar = NOVar String
2             | NOFun1 String NOVar
3             | NOFun2 String NOVar NOVar
4             | NODiff NOVar NOVar
5             | NOInt Integer
6   deriving (Eq, Ord)
7
8  data NOOp = L
9            | LE
10           | G
11           | GE
12           | EQU
13           | NEQU
14    deriving(Eq, Ord, Show)
15
16 type NODiffVal = Integer
17
18 data NOAtom = NOAtom NOVar NOOp NOVar
19   deriving (Eq, Ord)
```

Programmcode 13: Datentyp für die Nelson-Oppen-Kombinationsprozedur

Die Nelson-Oppen-Kombinationsprozedur lässt sich in drei funktionale Hauptbestandteile einteilen: **Purification**, **Propagation** und **Splitting**. Der Zusammenhang ist in Abbildung 4 zu sehen.

[8] im Vergleich zum Vorgehen bei konvexen Theorien

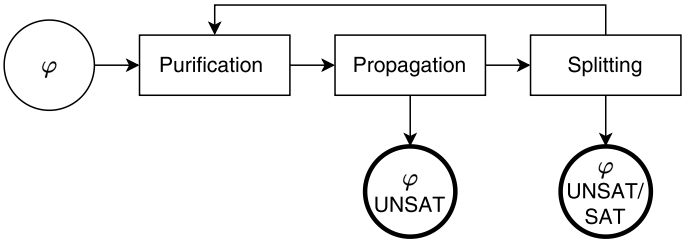

Abbildung 4: Schritte in der Nelson-Oppen-Kombinationsprozedur

Jedem dieser Bestandteile ist ein eigenes Unterkapitel gewidmet, in dem sowohl die Theorie als auch die Implementierung enthalten sind. Im Folgenden wird die Purification erläutert.

4.3.1 Purification

Das Ziel der Purification ist es, eine kombinierte Formel φ so mit Atomen zu erweitern und die Atome der Formel in der Weise zu bearbeiten, dass am Ende jedes Atom einer Theorie zugeordnet werden kann.

Dabei geht man so vor, dass man in allen Atomen die Elemente identifiziert, die verhindern, dass das Atom einer Theorie zugeordnet werden kann, und ersetzt diese Elemente durch Hilfsvariablen aus einer Theorie, sodass eine Zuordnung möglich ist. Anschließend wird die Hilfsvariable dem Element, das sie im ursprünglichen Atom ersetzt, gleichgesetzt und durch eine Konjunktion an die Formel angehängt (siehe Beispiel 4.4) [vgl. KS16, S.232]. Wendet man dieses Vorgehen auf alle kombinierten Atome an, so sind alle Atome genau einer Theorie zuordenbar und aus φ kann eine Formel φ' gemacht werden, die aus zwei Teilformeln besteht:

$$\varphi' = F_1 \wedge F_2$$

F_1 enthält alle Atome der einen Theorie und F_2 die der anderen Theorie, wobei Variablennamen und somit die Variablen selbst in beiden Formeln vorkommen können. Eine wichtige Tatsache, die im Rahmen dieser Arbeit nicht bewiesen wird, ist, dass φ und φ' äquivalent sind, d.h. φ ist erfüllbar in der kombinierten Theorie genau dann, wenn φ' erfüllbar ist (siehe Definition 2.5). Die Teilformeln F_1 und F_2 kann man mit der jeweiligen Entscheidungsprozedur der Theorie lösen, muss aber, bevor man auf die Erfüllbarkeit der Formel φ' und somit auch der Formel φ schließen kann, erst noch den zweiten Schritt, die **Propagation**, wiederholt ausführen (siehe Unterkapitel 4.3.2), um anschließend im dritten Schritt, dem **Splitting**, auf ein Ergebnis zu kommen (siehe Unterkapitel 4.3.3).

Beispiel 4.4.

$$f(x_0) - x_1 < 4 \ \wedge \ x_1 - x_2 = 5 \ \wedge \ f(x_0) \leq g(x_2)$$

$$\downarrow \text{Purification}$$

$$\underbrace{aux_0 - x_1 < 4 \ \wedge \ x_1 - x_2 = 5 \ \wedge \ aux_0 \leq aux_1}_{\text{DL}} \ \wedge \ \underbrace{aux_0 = f(x_0) \ \wedge \ aux_1 = g(x_2)}_{EUF}$$

Die Umsetzung im Code erfolgt mit der Funktion

```
purifyT :: [NOAtom] -> ([TermT GenDLAtom],[TermT Equality]).
```

Als Eingabe dient die Liste von `NOAtom`, die die zu untersuchende Formel repräsentiert. Von dieser Funktion wird die Funktion

```
purify :: String -> NOAtom -> ([TermT GenDLAtom], [TermT Equality])
```

aufgerufen. Aus der Signatur lässt sich entnehmen, dass ein `String` benötigt wird. Diese Zeichenfolge wird zur Benennung der Hilfsvariablen benutzt, damit jede neu erzeugte Hilfsvariable innerhalb der Formel einen eindeutigen Namen hat. Dabei wird ein Zählprinzip genutzt, das an das Wort „NOAux“ zuerst die Zahl anhängt, die der Position des `NOAtoms` in der ursprünglichen Formel entspricht, und dann eine durch Punkte getrennte Folge von 1en und 2en anhängt. Weiter unten im Text wird das genaue Vorgehen erläutert und ein Beispiel gegeben.

Die Funktion `purify` nutzt ihrerseits wieder drei Funktionen, die je nach Aufbau des eingegebenen `NOAtom` über Pattern Matching ausgewählt werden. Ist der enthaltene Operator ein „=“ oder ein „≠“, kann es sich um ein Atom der EUF-Logik handeln und wird weiter durch die Funktion `purifyEquation` behandelt. Darin werden alle infrage kommenden `NOAtome` untersucht und, wenn möglich, gleich in Atome der EUF-Logik vom Typ `TermT Equality` umgewandelt. Falls jedoch eine Differenz oder eine Zahl darin vorkommt, kann es sich nur um ein Atom der Differenzenlogik handeln. Ein Atom, das eine Differenz enthält, wird durch die Funktion `purifyDiff` behandelt, ein Atom, das eine Zahl aber keine Differenz enthält, durch die Funktion `purify2Diff`. Natürlich können z.B. in der Differenz als Minuend oder Subtrahend wieder uninterpretierte Funktionen auftreten. Diese werden vorerst durch eine Variable aus der jeweiligen Theorie ersetzt und danach mit der Funktion `purifyFun` weiter untersucht.

Ist der Operator des `NOAtom` kein „=“ und kein „≠“, kann es sich um ein Atom der Differenzenlogik handeln. Das Atom wird, wie im Abschnitt zuvor, wenn darin eine Differenz vorkommt, durch `purifyDiff` behandelt und in allen anderen Fällen durch die Funktion `purify2Diff`. Bei der Funktion `purifyDiff` können zwei Fälle auftreten, die beide nicht in ein Atom der Differenzenlogik oder der EUF-Logik überführt werden können. Der erste Fall tritt auf, wenn es im `NOAtom` eine Differenz gibt, aber darin keine Zahl vorkommt. Der zweite Fall kommt zustande, wenn die einzige enthaltene Zahl des `NOAtom` der Minuend der Differenz ist.

Wird durch eine Funktion eine neue Hilfsvariable eingeführt wird an den bisherigen übergebenen Hilfsvariablennamen „.1“ angehängt. Bei den Funktionen `purifyEquation`, `purify2Diff` und `purifyFun` kann es dazu kommen, dass zwei Hilfsvariablen neu eingeführt werden müssen, da die Bestandteile eines `NOAtoms` genauer untersucht werden, weil es z.B. Bestandteile vom Typ `NOFun1` hat. Dazu wird an den Namen „.1 “ oder „.2 “ angehängt, je nachdem, ob es sich um die erste neu eingeführte Variable oder die zweite handelt. Am Beispiel von Abbildung 5 wird das veranschaulicht.

Nachdem man die Purification auf eine kombinierte Formel φ angewendet hat, kann man mit dem Ergebnis der Purification, das aus einem Tupel der zwei Listen F_1 und F_2 besteht, die jeweils nur noch Atome einer der beiden Theorien enthalten, den nächsten Schritt ausführen, bevor man zur **Propagation** kommt. Dieser nächste Schritt prüft jeweils mit der Entscheidungsprozedur der jeweiligen Theorie, ob die Formeln F_1 und F_2 erfüllbar sind. Sind sie das nicht, so ist die kombinierte Formel φ auch unerfüllbar. Ansonsten geht man zur Propagation über.

4.3.2 Propagation

Die Formeln F_1 und F_2 werden bei der Propagation einzeln daraufhin untersucht, ob sie Gleichheiten implizieren, und diese Gleichheiten werden an die jeweils andere Formel mit Konjunktion angehängt (d.h. propagiert) und mit der entsprechenden Entscheidungsprozedur erneut auf Erfüllbarkeit geprüft. Falls keine weiteren Gleichheiten mehr gefolgert werden können, so muss mithilfe des Splittings entschieden werden, ob die ursprüngliche Formel φ erfüllbar ist [vgl. KS16, S.233]. Um herauszufinden, welche Gleichheiten impliziert werden können, gibt es ein Vorgehen, das für

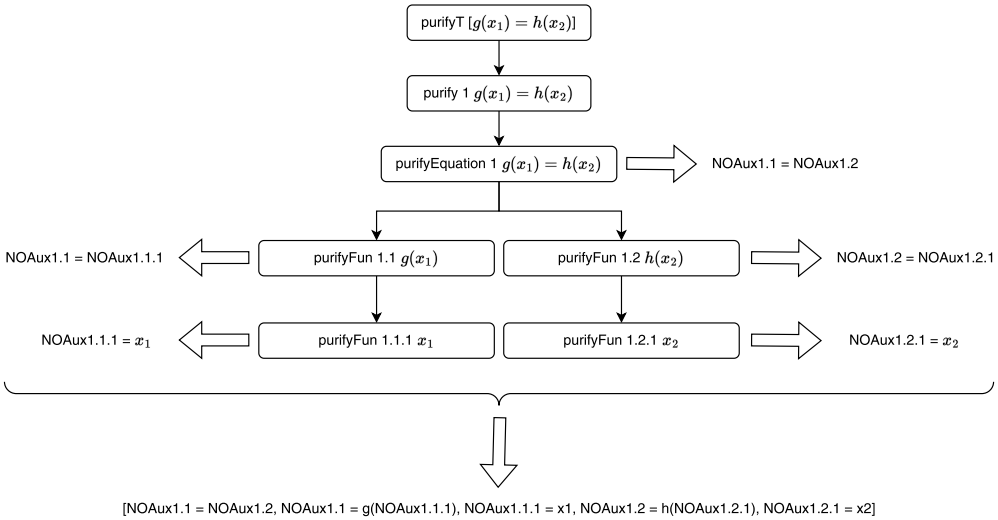

Abbildung 5: Einfaches Beispiel für `purifyT` und Benennung der Variablen

alle Theorien angewendet werden kann. Sei dazu eine Formel F einer Theorie gegeben und für zwei Variablen a_i und a_j soll untersucht werden, ob die Gleichheit $a_i = a_j$ impliziert werden kann. Es ist also zu prüfen, ob die Aussage

$$F \quad \Rightarrow \quad a_i = a_j$$

wahr ist. Formt man diese Aussage um, so erhält man:

$$\begin{gathered} \neg F \vee (a_i = a_j) \\ \downarrow \\ \neg(F \wedge \neg(a_i = a_j)) \\ \downarrow \\ \neg(F \wedge (a_i \neq a_j)) \end{gathered}$$

Folglich reicht es aus mit der Entscheidungsprozedur zu zeigen, dass $F \wedge (a_i \neq a_j)$ falsch ist. Um daraus einen Algorithmus zu bilden, werden zuerst die Variablen der Formel F_1 extrahiert, dann alle möglichen Kombinationen C von je zwei Variablen $c_{ij} = (a_i, a_j)$ gebildet. Anschließend werden nacheinander die Konjunktion bestehend aus F_1 und der Ungleichheit je einer Kombination c_{ij}, d.h. $F_1' = F_1 \wedge (a_i \neq a_j)$, mit der Entscheidungsprozedur geprüft. Falls die Entscheidungsprozedur ausgibt, dass die neu gebildete Formel F_1' erfüllbar ist, so wird die Gleichheit nicht impliziert und es wird mit der Ungleichheit von zwei anderen Variablen fortgefahren. Ist die neu gebildete Formel F_1' aber unerfüllbar, so wird die Gleichheit der beiden Variablen von F_1 impliziert und sie kann durch Konjunktion zu F_1 hinzugefügt werden, sodass gilt $F_1^{neu} = F_1 \wedge (a_i = a_j)$. Im Anschluss wird $a_i \wedge a_j$ auch an die Formel F_2 der anderen Theorie durch Konjunktion angehängt. Die neu gebildete Formel $F_2^{neu} = F_2 \wedge (a_i = a_j)$ wird auf Erfüllbarkeit überprüft. Ist sie unerfüllbar, so ist die gesamte Formel φ unerfüllbar. Ist sie erfüllbar, wird der Vorgang mit F_1^{neu} wiederholt, bis alle Variablenkombinationen geprüft wurden, um dann mit der Formel F_2^{neu} der anderen Theorie den gesamten Vorgang erneut zu durchlaufen. Da nach jeder Propagation bereits überprüfte Gleichheiten, die nicht impliziert wurden, auf einmal doch impliziert werden können, muss man erneut alle Gleichheiten überprüfen. Die folgende Abbildung 6 stellt den letztgenannten Punkt nochmal dar.

Das gesamte Vorgehen bei der Purification und der Propagation wird in Abbildung 7 verdeutlicht.

Im Code ist die Nelson-Oppen-Kombinationsprozedur in der Funktion `solveNO` implementiert. Diese nimmt die Ergebnisse der Purification als Eingabe entgegen und liefert `(Nothing,Nothing)`

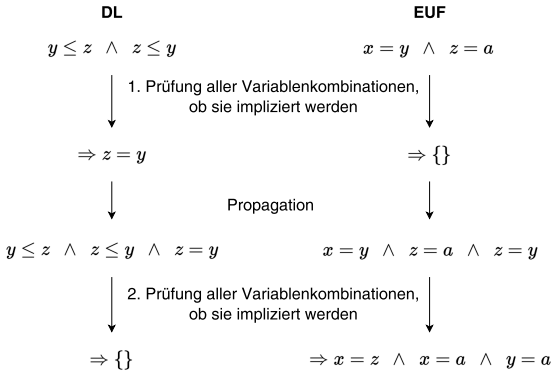

Abbildung 6: Beispiel für wiederholte Prüfung

zurück, wenn die kombinierte Formel nicht erfüllbar ist. Ansonsten liefert sie eine Zuordnung von Wahrheitswerten zu den einzelnen Atomen der Formel in der Form

```
(Maybe (Valuation DLAtom), Maybe (Valuation Equality))
```

nach der Purification zurück. Gibt es nach der Purification nur Atome einer Theorie, wird nur die Entscheidungsprozedur der jeweiligen Theorie aufgerufen und die Lösung ausgegeben. Gibt es aber Atome aus beiden Theorien, beginnt der Hauptteil der Nelson-Oppen-Kombinationsprozedur.

Dazu wird eine Funktion `solveNOentry` aufgerufen, die einen Durchgang der Propagation, wie oben beschrieben, macht und anschließend überprüft, ob ein erneuter Durchgang notwendig ist. Um das herauszufinden, gibt die Funktion zurück, welche Gleichheiten von Variablen nicht impliziert wurden, bei denen aber bei einer erneuten Prüfung der neu entstandenen Formeln die Möglichkeit besteht, doch noch impliziert zu werden. Hat sich die Anzahl der nochmal zu prüfenden Gleichheiten - im Code dargestellt als Tupel von Variablen `([(DLVar, DLVar)], [(Symbol, Symbol)])` - nicht geändert, so ist die Prozedur zu Ende und die Lösung kann ausgegeben werden. Ein Durchgang der Propagation wird durch die Funktion `solveNOrec` vollzogen. Dabei werden, wie oben beschrieben,

- alle Kombinationen von Variablen als Ungleichheiten zur Formel hinzugefügt,
- mit der Entscheidungsprozedur geprüft,
- je nach Ausgang dieser Prüfung entweder deren Gleichheiten hinzugefügt oder nicht
- und im Anschluss, falls nötig, die Gleichheiten in die jeweils andere Theorie durch Konjunktion propagiert.

Damit bei einem erneuten Durchgang nicht Gleichheiten geprüft werden, die schon impliziert wurden, wird eine Liste von Gleichheiten, aus der die bereits implizierten Gleichheiten entfernt wurden, beim erneuten Aufruf in die Funktion `solveNOrec` eingesetzt. Zusätzlich wird diese Liste auch der Ausgabe von `solveNOrec` hinzugefügt, damit der Vergleich in `solveNOentry` mit dem Zustand davor erfolgen kann.

4.3.3 Splitting

Wie oben erwähnt, muss die Nelson-Oppen-Kombinationsprozedur für konvexe Theorien um das Splitting erweitert werden, da es sich bei der Differenzenlogik um eine nicht konvexe Theorie handelt. Zunächst folgt die Definition, was es heißt, dass eine Theorie konvex bzw. nicht konvex ist.

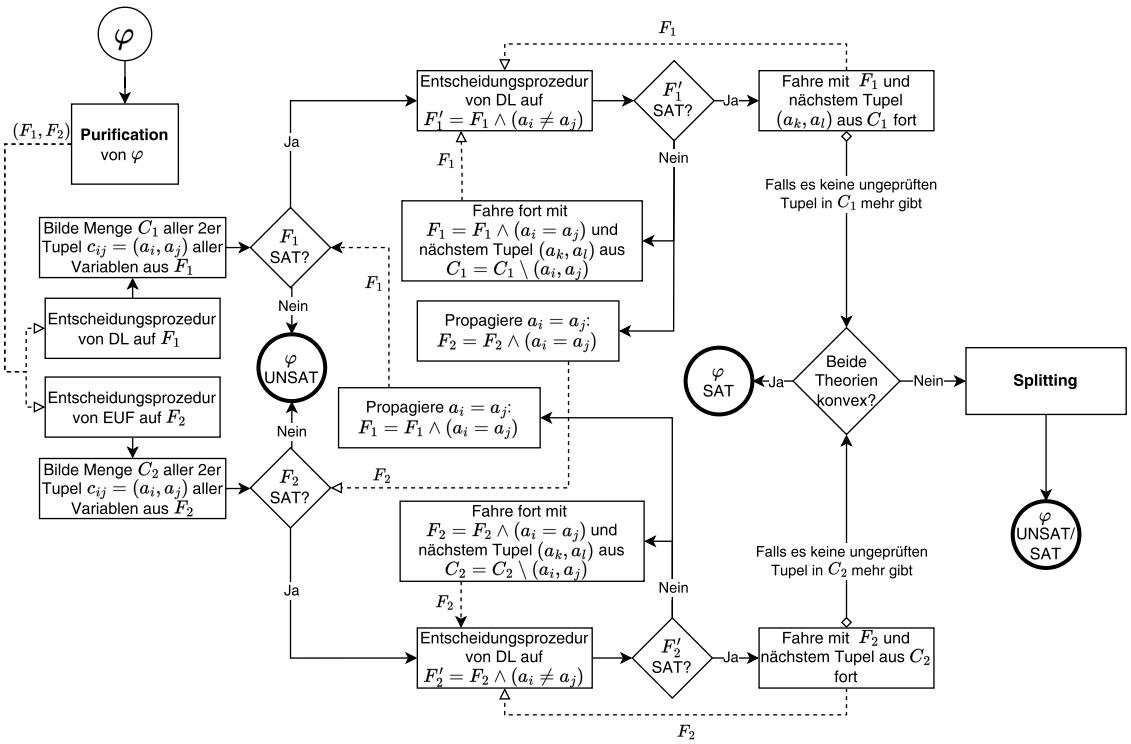

Abbildung 7: Ablaufdiagramm für die Nelson-Oppen-Kombinationsprozedur

Definition 4.5 (konvexe Theorie)**.** *Eine Theorie T ist konvex, wenn für jede Formel φ, die nur über Konjunktionen verknüpft ist, gilt, dass sie konvex ist. Gibt es eine Formel φ, die nicht konvex ist, so ist die Theorie T insgesamt nicht konvex [vgl. KS16, S.230].*

Um diese Definition zu verstehen, benötigt man die Definition von konvexen bzw. nicht konvexen Formeln.

Definition 4.6 (konvexe Formel)**.** *Eine Formel φ, die nur über Konjunktionen verknüpft ist, heißt nicht konvex, wenn gilt:*

$$\varphi \Rightarrow \bigvee_{i=1}^{n} (x_i = y_i)$$

aber es kein $i \in [1, n]$ gibt, sodass gilt:

$$\varphi \Rightarrow x_i = y_i$$

Gibt es mindestens ein i, so heißt die Formel konvex [vgl. Opp80, S.293].

Dass die Differenzenlogik eine nicht konvexe Theorie ist, kann man mit folgendem Beispiel [vgl. Opp80, S.293] zeigen.

Beispiel 4.5. Sei die Formel $\varphi = (x \geq 1) \wedge (x \leq 2) \wedge (y = 1) \wedge (z = 2)$ aus der Theorie der Differenzenlogik gegeben. Da x nur Werte aus $\mathbb{N}$ annehmen kann, gilt in diesem Fall $x = 1$ oder $x = 2$. Somit ergibt sich mit dem Rest der Formel φ, dass, je nachdem, welchen Wert x annimmt, entweder $x = y$ oder $x = z$ gilt. Es ist aber nicht möglich, dass aus φ nur eine Gleichheit gefolgert werden kann, d.h. es gilt $\varphi \Rightarrow x = y \vee x = z$, aber nicht nur $\varphi \Rightarrow x = z$ oder $\varphi \Rightarrow x = y$. Somit ist nach der Defintion 4.6 φ nicht konvex und nach Definition 4.5 die Theorie der Differenzenlogik nicht konvex.

Da bei der Nelson-Oppen-Kombinationsprozedur bisher nur die Implikationen von einzelnen Gleichheiten geprüft wurden (siehe Kapitel 4.3.2), aber nicht die Disjunktionen der Gleichheiten, führt man einen weiteren Schritt in die Nelson-Oppen-Kombinationsprozedur ein, der dies durch Re-

kursion mit dem schon vorhandenen Vorgehen aus der Propagation löst. Sei $\varphi' = F_1 \wedge F_2$ die zu untersuchende Formel φ nach der Purification, dann müssen für das anschließende Splitting folgende Schritte ausgeführt werden:

1. Prüfe für Gleichheiten, die nicht einzeln durch F_1 impliziert werden, ob Disjunktionen aus diesen Gleichheiten impliziert werden.
2. Führe die Nelson-Oppen-Kombinationsprozedur jeweils erneut aus mit den Formeln $\varphi' \wedge (x_1 = y_1), \varphi' \wedge (x_2 = y_2), ..., \varphi' \wedge (x_k = y_k)$
3. Ist keine von diesen Formeln erfüllbar, so ist die Formel φ unerfüllbar.

Im Code wird das Splitting durch den Aufruf der Funktion `solveSplitting` innerhalb der Funktion `solveNOentry` umgesetzt. Dazu wird der erste Schritt des Splittings mit Hilfe der Funktionen `listOfConjNegTuple` und `isImplicated` durchgeführt, indem alle Disjunktionen, bestehend aus zwei Gleichheiten, geprüft werden. Dabei geht man genauso vor, wie es im Unterkapitel 4.3.2 erklärt wird, mit dem einzigen Unterschied, dass diesmal zwei Ungleichheiten durch Konjunktion angeknüpft werden. Gibt es solche Disjunktionen von zwei Gleichheiten, die impliziert werden, so wird die Funktion `solveNOretry` mit der Formel aufgerufen, die um genau eine Gleichheit erweitert wird, wie es im Schritt zwei gefordert wird. Die Funktion liefert ein Ergebnis, wie es im dritten Schritt des Splittings beschrieben ist. Eine Übersicht der Schritte innerhalb des Splittings ist in Abbildung 8 zu sehen.

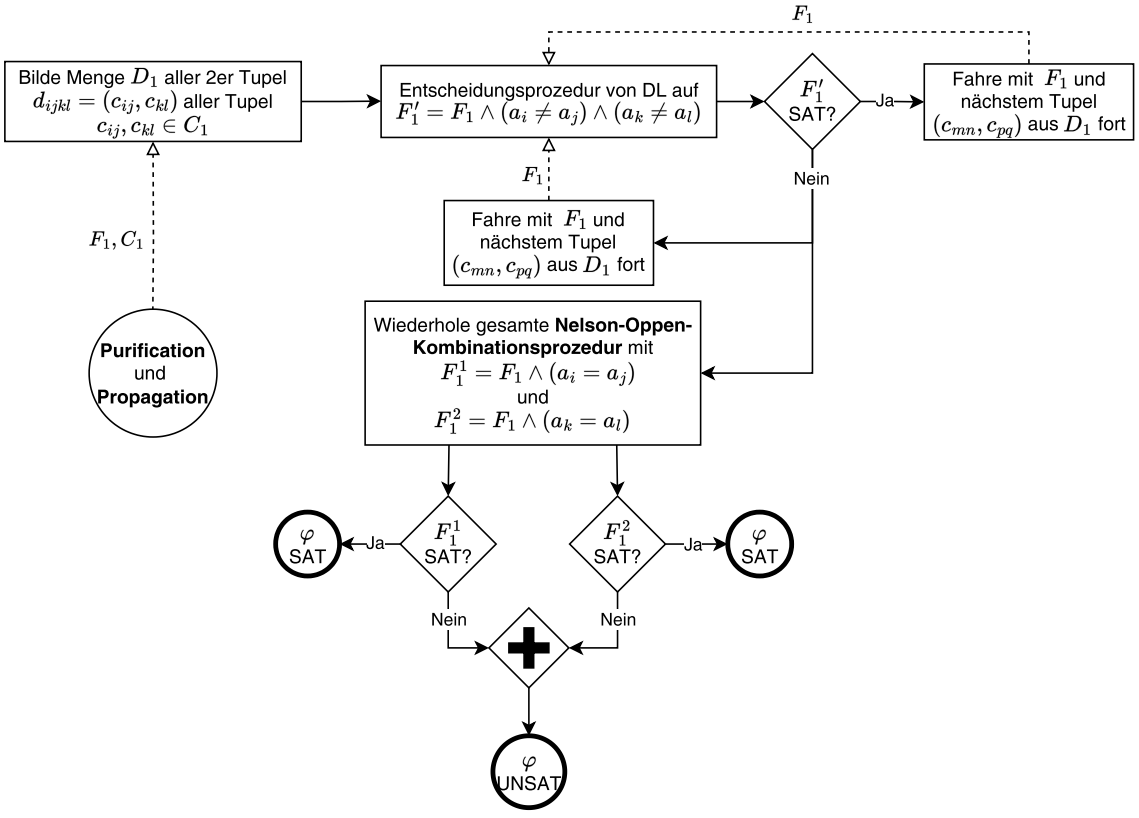

Abbildung 8: Ablaufdiagramm für den Schritt *Splitting* innerhalb der Nelson-Oppen-Kombinationsprozedur

5 Zusammenfassung und Ausblick

In diesem letzten Kapitel sollen die wichtigsten Punkt der Arbeit kurz resümiert, sowie auf im Arbeitsprozess auftretende Herausforderungen hingewiesen werden.

Der erste Teil der Arbeit beschäftigt sich mit der Einführung der Theorie der Differenzenlogik und dem Einfügen ihrer Entscheidungsprozedur in einen bereits bestehenden SMT-Solver. Die Besonderheit ist die Verwendung der Programmiersprache Haskell, woraus sich auch die erste Herausforderung ergab, und zwar die Erstellung eines gemeinsamen Datentyps, sodass der SMT-Solver sowohl für die EUF-Logik als auch für die Differenzenlogik funktioniert. Für die Entscheidungsprozedur der Differenzenlogik wird ein Algorithmus aus der Graphentheorie benutzt, auf dessen Grundlagen und Umsetzung näher eingegangen wird. Auch dort wird die besondere Anforderung der Implementierung in Haskell angesprochen.

Im zweiten Teil der Arbeit wird der Fokus auf eine Entscheidungsprozedur gelegt, die es ermöglicht Formeln zu entscheiden, die sowohl Bestandteile aus der EUF-Logik als auch aus der Differenzenlogik enthalten. Die Nelson-Oppen-Kombinationsprozedur schafft genau dies unter Verwendung der bereits schon implementierten Entscheidungsprozeduren für die beiden Theorien. Eine Besonderheit ist hier die Verwendung der Differenzenlogik über den ganzen Zahlen, da dies zur Folge hat, dass es sich um eine konvexe Theorie handelt und die Nelson-Oppen-Kombinationsprozedur um das Splitting erweitert werden muss.

Der Großteil der Herausforderungen bei der Arbeit mit Haskell im ersten, wie auch im zweiten Teil, liegt darin, dass es sich um eine funktionale Programmiersprache handelt. Somit wurden Operationen, die eine Speicherung eines Zustands benötigen, durch Rekursion oder die Verwendung der Zustände als Parameter und Rückgabewerte von Funktionen gelöst.

Richten wir zu guter Letzt unser Augenmerk darauf, inwieweit die durch die Arbeit gewonnenen Erkenntnisse erweitert bzw. vereinfacht werden können.

In einem weiteren Schritt könnten z.B. zum Auffinden von Fehlern im Code die Property Based Tests für die Entscheidungsprozedur der Differenzenlogik und des SMT-Solvers ausgebaut und für die Nelson-Oppen-Kombinationsprozedur eingeführt werden.

Da das Verfahren zum Herausfinden von implizierten Gleichheiten in der Nelson-Oppen-Kombinationsprozedur auf einem Verfahren basiert, das alle Möglichkeiten von Kombinationen der vorkommenden Variablen durchprobiert, ist es für Formeln mit einer großen Anzahl von Variablen sehr langsam. Hier können Algorithmen Abhilfe schaffen, die in der jeweiligen Theorie analytisch aus den Formeln die implizierten Gleichheiten liefern können. Für die EUF-Logik böten sich die Äquivalenzklassen nach dem Congruence Closure Verfahren an, da nach Abschluss des Verfahrens jeweils in einer Äquivalenzklasse alle Variablen enthalten sind, die gleich sind. Für die Differenzenlogik bietet sich wie bei der Entscheidungsprozedur ein graphentheoretischer Ansatz an [vgl. LM06, S.32f].

In der Nelson-Oppen-Kombinationsprozedur kann das Splitting erweitert werden, sodass die Implikation von mehr als zwei Gleichheiten getestet werden kann. Zusätzlich ist eine Suche nach einem sinnvollen gemeinsamen Datentyp in der Nelson-Oppen-Kombinationsprozedur sinnvoll, damit die Erweiterung um eine dritte Theorie, die sowohl konvex als auch nicht konvex sein kann, leichter möglich ist.

Literatur

[Cor+09] Thomas H. Cormen u. a. *Introduction to Algorithms - Third Edition*. 3. Aufl. Massachusetts, 2009.

[EP02] Katrin Erk und Lutz Priese. *Theoretische Informatik - Eine umfassende Einführung*. 2., erweit. Aufl. Berlin Heidelberg, 2002.

[HM22] Helmut Harbrecht und Michael Multerer. *Algorithmische Mathematik - Graphen, Numerik und Probabilistik*. 1. Aufl. Berlin Heidelberg, 2022.

[KS16] Daniel Kroening und Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. 2. Aufl. Berlin Heidelberg, 2016.

[LM06] Shuvendu K. Lahiri und Madanlal Musuvathi. „An Efficient Nelson-Oppen Decision Procedure for Difference Constraints over Rationals“. In: *Electronic Notes in Theoretical Computer Science* 144 (2 2006), S. 27–41.

[Opp80] Derek C. Oppen. „Complexity, convexity and combinations of theories“. In: *Theoretical Computer Science* 12 (1980), S. 291–302.

[Sch13] Michael Schenke. *Logikkalküle in der Informatik - Wie wird Logik vom Rechner genutzt?* 1. Aufl. Wiesbaden, 2013.

Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Fürstenfeldbruck, 17.03.2024
Ort, Datum


Unterschrift