

Studienarbeit

**Ein Entscheidungsverfahren
für
die Presburger–Arithmetik**

von
Stefan K. Baur

Betreuer:

Dr. Wolfgang Degen
Prof. Dr. Ulrich Rüde

Lehrstuhl für Informatik 10 (Systemsimulation)
Friedrich–Alexander–Universität Erlangen–Nürnberg

Erlangen, im November 2007

Erlangen, den 30. November 2007

Ich, STEFAN K. BAUR, versichere an Eides statt, dass ich die vorliegende Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

STEFAN K. BAUR

Danksagung Ich bedanke mich für interessante Gespräche zum Thema vorliegender Arbeit bei meinen Freunden und bei Jenen, die mir besonders gut zum Ziel verholfen haben. *Thomas Bertz*, der mir immer wieder gute Tipps zum Beweiser gegeben und meine Ausarbeitung korrigiert hat. *Arne Becker*, der mich immer wieder zum Weitermachen motiviert hat. *Hellmuth Klohs*, der mir eine völlig neue Sicht auf die Arbeit gegen hat. *Dr. Degen*, mein Betreuer, der mich immer wieder auf das Wesentliche zurückgeführt und dabei immer wieder Geduld bewiesen hat.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Thema	1
1.1.1	Beispielsätze	1
1.1.2	Rolle der Peano–Arithmetik	2
1.2	Presburger–Arithmetik	2
1.2.1	Axiomatische Darstellung	2
1.2.2	Semantische Darstellung	3
1.3	Quantorenelimination	3
1.3.1	Totale Quantorenelimination	4
1.3.2	Nicht–totale Quantorenelimination	4
1.3.3	Quantoreneliminationsansätze für $Th(\mathbb{Z}^<)$	4
2	Hauptteil	7
2.1	Anforderungsdefinition	7
2.1.1	Eingabe	7
2.1.2	Ausgabe	7
2.1.3	Projektziel	8
2.1.4	Qualitätsanforderungen	8
2.2	Problemanalyse	9
2.2.1	Entscheidbarkeit	9
2.2.2	Formeln mit freien Variablen	9
2.2.3	Sprachfindung	9
2.3	Programmübersicht	11
2.3.1	Frontend	11
2.3.2	Proving	11
2.3.3	Test	12

2.3.4	UI	12
2.4	Sprache $\mathcal{L}_{\mathbb{Z}^<}$	12
2.5	Syntaxbaum	13
2.5.1	Boolesche Knotentypen	13
2.5.2	Relationsknotentypen	14
2.5.3	Termknotentypen	15
2.5.4	Quantorenknotentyp	15
2.5.5	Knoteneigenschaften	16
2.6	Visitoren	16
2.6.1	Proofer	17
2.6.2	Simplifier	17
2.6.3	BooleanReducer	18
2.6.4	IsQuantifierFree	19
2.6.5	QuantifierInsider	19
2.6.6	QuantifierEliminator	20
2.6.7	RemoveUnusedQuantifiers	22
2.6.8	Basics	22
2.6.9	DNFer	23
2.6.10	Existentializer	23
2.6.11	Flatter	23
2.6.12	NotEliminator	24
2.6.13	RelationSideSwitcher	24
2.6.14	RelationTransformer	24
2.6.15	MinusReducer	25
2.6.16	Multiplier	25
2.6.17	ProductEliminator	25
2.6.18	Renamer	26
2.6.19	Substituter	26
2.6.20	SuccessorEliminator	26
2.6.21	VariablesReducer	26
2.6.22	BoundVariables	27
2.6.23	FreeVariables	27
2.6.24	CanEvaluate	27

2.6.25	Evaluator	28
2.6.26	ClassCounter	28
2.6.27	ContainsClass	28
2.6.28	SemantikChecker	28
2.6.29	VariableCounter	29
3	Schluss	30
3.1	Zusammenfassung	30
	Literaturverzeichnis	33

Kapitel 1

Einleitung

1.1 Thema

Vorliegende Arbeit behandelt die Presburger–Arithmetik. Es wird das von Monk [Mon76] skizzierte Entscheidungsverfahren in Java programmiert. Dieses monksche Entscheidungsverfahren beruht auf einer effektiven Quantorenelimination, d.h. es wird eine quantorenbehaftete Formel sukzessive in eine äquivalente, quantorenfreie Formel überführt. Ist die Inputformel ohne freie Variablen, dann kann der Wahrheitswert des durch die Quantorenelimination hergestellten Satzes sofort entschieden werden.

1.1.1 Beispielsätze

Es gibt einige entscheidbare Theorien, jedoch kann man mit den meisten dieser Theorien nur wenig ausdrücken wie beispielsweise mit der Sukzessor–Arithmetik. Die Presburger–Arithmetik hingegen zählt zu den Wenigen der entscheidbaren Theorien, mit der man einigermaßen interessante Aussagen treffen kann. Neben den trivialen Aussagen

- 6 ist durch 2 teilbar:

$$\exists \alpha. \alpha + \alpha = 6$$

- 6 ist durch 3 teilbar:

$$\exists \alpha. \alpha + \alpha + \alpha = 6$$

- Wenn eine Zahl durch 2 und durch 3 teilbar ist, dann ist sie auch durch 6 teilbar:

$$\exists \beta. (\exists \alpha. \alpha + \alpha = \beta \wedge \exists \alpha. \alpha + \alpha + \alpha = \beta \longrightarrow \exists \alpha. \alpha + \alpha + \alpha + \alpha + \alpha + \alpha = \beta)$$

kann man eben auch folgende, interessante Aussage, dessen Wahrheitswert zunächst nicht ganz ersichtlich ist, treffen:

- Für zwei gegebene, teilerfremde Zahlen, insbesondere Primzahlen (hier 2 und 5), gibt es eine Zahl n , ab der jede Zahl sich als Linearkombination darstellen lässt (Frobenius):

$$\exists n. \forall \alpha. (n < \alpha \longrightarrow \exists \beta. \exists \gamma. (\alpha = \beta + \beta + \gamma + \gamma + \gamma + \gamma + \gamma))$$

In [Smo91] werden weitere, interessante Additionsformeln aufgeführt.

Der allgemeine Begriff *Primzahl* zum Beispiel lässt sich mit der Presburger-Arithmetik leider nicht definieren.

1.1.2 Rolle der Peano-Arithmetik

Es gibt viele Theorien, die nicht entscheidbar sind. Meist sind genau diejenigen Theorien, für die man sich ein Entscheidungsverfahren wünschen würde, unentscheidbar wie beispielsweise die Peano-Arithmetik. Doch ist es teilweise auch möglich durch gezielte sprachliche Restriktionen aus einer unentscheidbaren Theorie eine andere, aber entscheidbare Theorie zu gewinnen.

Würde man die Peano-Arithmetik um die Multiplikation erleichtern, erhielte man eben die entscheidbare Presburger-Arithmetik.

1.2 Presburger-Arithmetik

Die Presburger-Arithmetik, auch lineare Arithmetik genannt, bezeichnet die von dem polnischen Logiker Presburger 1929 konstruierte, entscheidbare Theorie der natürlichen bzw. ganzen Zahlen mit Addition.

Die Presburger-Arithmetik kann man demnach sowohl über \mathbb{N} als auch über \mathbb{Z} betrachten.

Mit \mathbb{N} kann man die Ordnung $<$ mit der zur Verfügung stehenden Addition $+$ definieren:

$$\sigma < \tau : \iff \exists \alpha. \alpha + 1 + \sigma = \tau.$$

Dies ist im Fall von \mathbb{Z} nicht möglich. Will man die Ordnung $<$ im Fall von \mathbb{Z} haben, so muss man ein primitives Symbol $<$ explizit in die Theorie der Presburger-Arithmetik aufnehmen, was im Folgenden der Fall ist.

Die so erweiterte Presburger-Arithmetik kann auf zweierlei Arten gegeben werden, zum Einen axiomatisch oder zum Andern semantisch, nämlich als eine Menge aller wahren Sätze.

1.2.1 Axiomatische Darstellung

Apelt schlägt in [Ape66] das vollständige sowie unabhängige Axiomensystem Ξ_{PrA} zur Presburger-Arithmetik vor.

$$\Xi_{PrA} = \{ \quad \forall \alpha. \forall \beta. \quad \alpha + \beta = \beta + \alpha, \quad (1.1)$$

$$\forall \alpha. \forall \beta. \forall \gamma. \quad \alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma, \quad (1.2)$$

$$\forall \alpha. \quad 0 + \alpha = \alpha, \quad (1.3)$$

$$\forall \alpha. \forall \beta. \exists \gamma. \quad \alpha + \gamma = \beta, \quad (1.4)$$

$$\forall \alpha. \exists \beta. \bigvee_{i=0}^{p-1} p * \beta + i = \alpha, \quad (1.5)$$

$$\forall \alpha. \forall \beta. (\alpha < \beta \longrightarrow \neg(\beta < \alpha + 1)), \quad (1.6)$$

$$\forall \alpha. \forall \beta. \forall \gamma. (\alpha < \beta \wedge \beta < \gamma \longrightarrow \alpha < \gamma), \quad (1.7)$$

$$\forall \alpha. \forall \beta. \alpha < \beta \vee \alpha = \beta \vee \beta < \alpha, \quad (1.8)$$

$$\forall \alpha. \forall \beta. \forall \gamma. \alpha < \beta \longrightarrow (\gamma + \alpha < \gamma + \beta), \quad (1.9)$$

$$0 < 1 \}, \quad (1.10)$$

wobei der Parameter p im Schema 1.5 alle Primzahlen durchläuft, was für die Unabhängigkeit sorgt.

Die Axiome von 1.6 bis 1.10 sind für die hinzufinierte Ordnung notwendig.

1.2.2 Semantische Darstellung

Eine semantische Darstellung der Presburger–Arithmetik ergibt sich als die Theorie $Th(\mathbb{Z}^<)$ der algebraischen–relationalen Struktur $\mathbb{Z}^< = (\mathbb{Z}; 0, 1, +, -, <)$, wobei allgemein die Theorie einer Struktur die Menge der in der Struktur wahren Sätze ist.

$$Th(\mathbb{Z}^<) = \{ \varphi : \mathbb{Z}^< \models \varphi \}$$

Die Symbole der Struktur $\mathbb{Z}^<$ entsprechen ihrer natürlichen Interpretation.

1.3 Quantorenelimination

Als Quantorenelimination bezeichnet man das Überführen der zu entscheidenden Proposition mit kalkülartigen, semantisch korrekten Transformationsregeln in eine äquivalente, quantorenfreie Darstellung. Nach der Booleschen Auswertung des quantorenfreien Resultats erhält man einen Wahrheitswert, der angibt, ob die zu entscheidende Proposition ein Theorem der Theorie ist.

Beispielsweise können Sätze der folgenden Theorien mit der Methode der Quantorenelimination entschieden werden.

- Gleichheitstheorie (Löwenheim 1915).
- Theorie endlich vieler Mengen
- Sukzessor–Arithmetik
- Presburger–Arithmetik (Presburger 1929)

Die Quantorenelimination als solches kann in zwei Arten, die totale und die nicht–totale Quantorenelimination, unterteilt werden.

1.3.1 Totale Quantorenelimination

Eine *totale Quantorenelimination* eliminiert die Quantoren einer Formel φ vollständig, d.h. das Eliminationsresultat enthält in keiner Weise Quantoren und ist äquivalent zur Ausgangsformel φ .

Beispielsweise kann ein Entscheidungsverfahren zur Sukzessor-Arithmetik mit einer totalen Quantorenelimination realisiert werden.

1.3.2 Nicht-totale Quantorenelimination

Eine *nicht-totale Quantorenelimination* kapselt die zu eliminierenden Quantoren mit bestimmten Formeln, so genannten Basisformeln.

Eine *nicht-totale Quantorenelimination* eliminiert nur die außerhalb von so genannten Basisformeln vorkommenden Quantoren. Die außerhalb vorkommenden Quantoren werden mit Hilfe der Basisformeln, die Quantoren einkapseln, eliminiert.

Die Reelle Arithmetik $\mathbb{R} = (\mathbb{R}; 0, 1, +, *)$ kann beispielsweise mit der Methode der nicht-totalen Quantorenelimination entschieden werden, wohingegen die Reelle Arithmetik mit Ordnung $\mathbb{R}^< = (\mathbb{R}; 0, 1, +, *, <)$ mit der Methode der totalen Quantorenelimination entschieden werden kann.

Eine Basisformel kann einen Quantor oder mehrere Quantoren einkapseln, derart dass die Darstellung im Groben zwar quantorenfrei ist, aber die Basisformeln als solche immer noch quantorenbefahet sind.

Nebenbei bemerkt, setzt jede Quantorenelimination die Definition einer Menge von Basisformeln voraus, die durchaus wohl eingekapselte Quantoren haben können.

1.3.3 Quantoreneliminationsansätze für $Th(\mathbb{Z}^<)$

In der Literatur wird die Quantorenelimination, insbesondere die nicht-totale Quantorenelimination, als Kern für ein Entscheidungsverfahren für $Th(\mathbb{Z}^<)$ in Betracht gezogen. In diesem Fall wird die Struktur $\mathbb{Z}^<$ um eine bestimmte Relation, die einen Quantor makroartig kapselt, erweitert (zum Beispiel die Kongruenz bezüglich eines festen Moduls). Bei der Quantorenelimination werden nicht wirklich die Quantoren eliminiert. Die Proposition wird in eine quantorenfreie Darstellung überführt, die Quantoren werden gegebenenfalls indirekt unter Verwendung der hinzudefinierten, makroartigen Relation beibehalten.

Nun werden zwei nicht-totale Quantoreneliminationsansätze für $Th(\mathbb{Z}^<)$ vorgestellt.

Rautenbergs Ansatz

Rautenberg schlägt in [Rau96] als zusätzliche Relation, die einstellige Teilerrelationen $c \mid$ mit einem konstanten Teiler $c > 1$, vor.

$$c \mid \sigma \iff \exists \alpha. \alpha * c = \sigma$$

$$\longleftrightarrow \exists \alpha. \underbrace{\alpha + \alpha + \cdots + \alpha}_{c \text{ mal } \alpha} = \sigma$$

Beispielweise würde der quantorenbehaftete Satz $\exists \alpha. \alpha + \alpha + \alpha = 6$ nach der Rautenbergschen Quantorenelimination in die äquivalente Basisformel $3|6$ transformiert werden.

Monks Ansatz

Monk hingegen gebraucht für die Quantorenelimination die zweistellige Kongruenzrelationen \equiv_m mit einem konstanten Modulwert $m > 1$.

$$\begin{aligned} \sigma \equiv_m \tau &\longleftrightarrow \exists \alpha. (\alpha * m + \sigma = \tau) \\ &\longleftrightarrow \exists \alpha. \underbrace{(\alpha + \alpha + \cdots + \alpha)}_{m \text{ mal } \alpha} + \sigma = \tau \end{aligned}$$

Beispielweise würde ähnlich Rautenbergs Ansatz der quantorenbehaftete Satz $\exists \alpha. (\alpha + \alpha + 5 = 7)$ nach der Monkschen Quantorenelimination in die äquivalente Basisformel $5 \equiv_2 7$ transformiert werden.

Die Kunst liegt darin, eine beliebige quantorenbehaftete Inputformel, in eine Boolesche Kombination von Basisformeln zu transformieren. Wie das geht, skizziert Monk in [Mon76] auf nur fünf Seiten.

Nebenbei bemerkt, sind die Ansätze von Rautenberg und Monk gar nicht so unterschiedlich, denn eine Monksche, zweistellige *Kongruenzrelation* mit einem festen Modul $m > 1$ lässt sich auch mit einer Rautenbergschen, einstelligen *Teilerrelation* ausdrücken:

$$\sigma \equiv_m \tau \longleftrightarrow m \mid \sigma - \tau$$

Kapitel 2

Hauptteil

2.1 Anforderungsdefinition

Gefordert ist ein Entscheidungsprogramm (Computerprogramm), welches ein Entscheidungsverfahren für die Presburger–Arithmetik realisiert.

2.1.1 Eingabe

Der Eingabebereich des Entscheidungsprogramms muss beliebige Formeln (Inputformeln) aus einer intuitiv verständlichen Sprache akzeptieren. Die Inputformeln müssen wohlgeformt sein und können Quantoren $\mathcal{Q} \in \{\exists, \forall\}$ sowie freie Variablen enthalten.

2.1.2 Ausgabe

Als Entscheidungsergebnis werden im Allgemeinen folgende Formeln (Outputformeln) erwartet:

- *wahr*: Jede Belegung der freien Variablen macht aus der gegebenen Formel einen wahren Satz.
Beispielsweise wird für die Inputformel $\exists \alpha. (\alpha + 2 < 9 \wedge \beta = \beta)$ mit der freien Variable β als Output *wahr* erwartet, denn jede Belegung für β macht die Inputformel wahr.
- *falsch*: Es gibt keine Belegung der freien Variablen, die aus gegebener Formel einen wahren Satz macht.
Beispielsweise wird für die Inputformel $\exists \alpha. (\alpha + 2 < 9 \wedge \beta \neq \beta)$ mit der freien Variable β als Output *falsch* erwartet, denn es gibt keine Belegung für β , die die Inputformel wahr macht.
- *quantorenfreie Formel mit eventuell freien Variablen*: Für eine beliebige Belegung der freien Variablen werden die Inputformel und die zugehörige Outputformel zu äquivalenten Sätzen.

Beispielsweise wird für die Inputformel $\exists\alpha.(\alpha + 2 < 9 \wedge \beta = 0)$ die Outputformel $\beta = 0$ erwartet.

Insbesondere wird für jeden gegebenen Satz (Formel ohne freie Variablen) als Ergebnis ein Wahrheitswert (wahr, falsch) gefordert. Die Angabe einer Inputformel (Formel mit freien Variablen) soll als Anfrage zu verstehen sein, die eine quantorenfreie Form der Formel gegebenenfalls mit freien Variablen zurückgibt.

2.1.3 Projektziel

Mit diesem Entscheidungsprogramm soll dem Anwender eine unkomplizierte sowie rasche Überprüfung, ob ein gegebener Satz ein Element der Theorie der Presburger–Arithmetik $Th(\mathbb{Z}^<)$ ist, ermöglicht werden. Durch die Benutzung des Entscheidungsprogramms soll der Anwender auf eventuell neue Erkenntnisse kommen können.

2.1.4 Qualitätsanforderungen

Robustheit

Das Programm muss *robust* sein. Keine durch den Anwender ausgelöste Aktion darf zum Programmabsturz führen.

Korrektheit

Auf das Ergebnis des Entscheidungsprogramms muss man sich verlassen können, deshalb wird besonders großen Wert auf die *Korrektheit* gelegt. Falsche Sätze dürfen nicht als wahr entschieden werden und umgekehrt.

Benutzerfreundlichkeit

Die *Benutzerfreundlichkeit* ist auch wichtig, der Anwender soll nicht gezwungen werden, seine Formel selbstständig umzuformulieren. Die Eingabesprache muss also so mächtig sein, dass beliebige Boolesche Kombinationen von Formeln angegeben werden können.

Effizienz

Die *Effizienz* darf vernachlässigt werden.

2.2 Problemanalyse

2.2.1 Entscheidbarkeit

Die Hauptfrage ist, ob $Th(\mathbb{Z}^<)$ entscheidbar ist oder nicht, ob es ein Entscheidungsverfahren für $Th(\mathbb{Z}^<)$ gibt. Nach Monk, der sogar ein Entscheidungsverfahren für die Presburger-Arithmetik in [Mon76] skizziert, ist $Th(\mathbb{Z}^<)$ entscheidbar. Monk schlägt zu diesem Zwecke eine *nicht-totale* Quantorenelimination vor. Die Menge der Basisformeln beschränkt sich auf folgende, zweistellige Relationen:

- $\sigma = \tau$ (Gleichheitsrelation),
- $\sigma < \tau$ bzw. $\sigma > \tau$ (simple Ordnungsrelationen),
- $\sigma \equiv_m \tau$ (Kongruenzrelation bezüglich eines festen Moduls m),

wobei σ und τ Terme sind.

2.2.2 Formeln mit freien Variablen

Können Formeln mit freien Variablen mit einem Entscheidungsverfahren in entsprechende Formeln überführt werden? Bei Monks Eliminationsvorschlag werden die Quantoren einer beliebigen Formel sukzessive von innen nach außen eliminiert (ein innerer Quantor hat in seinem Bindungsbereich keine weiteren Quantoren).

Dies wird mit einem elementaren Eliminationsschritt erreicht: eine Formel $Q\alpha.\varphi$ wird in eine äquivalente, quantorenfreie Darstellung ψ überführt, wobei der Quantor $Q \in \{\exists, \forall\}$ ist, die Teilformel φ quantorenfrei ist und die Teilformel φ durchaus freie Variablen enthalten kann. Die an den Quantor Q gebundene Variable α kommt syntaktisch in ψ nicht mehr vor.

Aus diesem Grunde kann das von Monk skizzierte Entscheidungsverfahren auch auf Formeln mit freien Variablen angewandt werden.

2.2.3 Sprachfindung

Ziel ist es eine möglichst große Sprache $\mathcal{L}_{\mathbb{Z}^<}$ zu finden, die eine intuitive Eingabe einer beliebigen Formel ermöglicht.

Boolesche Operationen

Im Prinzip kommt man mit einer relativ kleinen Menge von Booleschen Operationen aus, jede beliebige Aussage zu formulieren. Beispielsweise käme man nur mit der einstelligen Operation \neg und den zweistelligen Operationen \wedge und \vee aus. Bevor man aber bei der Eingabe mit vielen Klammern zu kämpfen hat, kann man die zweistelligen Operationen auf n -stellige Operationen mit $n > 1$ ausweiten.

Des Weiteren sollte man auch die Operation wie \longrightarrow und \longleftrightarrow als n -stellige Operation mit $n > 1$ hinzunehmen.

true und *false* werden als nullstellige Operationen mit in die Sprache aufgenommen, so kann ein resultierender Wahrheitswert auch als Formel der zu findenden Sprache dargestellt werden.

Erweiterung der Relationenmenge

Auf der Suche nach einer möglichst großen Relationenmenge muss man sich an die gewählte Menge von Basisformeln orientieren. Für jede einzelne Relation, die nicht in der Menge der Basisformeln liegt, die man aber syntaktisch der Sprache hinzufügen möchte, muss es eine Überführung in eine Basisformel geben. Folgende Relationen können in die von Monk vorgeschlagenen Basisformeln überführt werden.

- $\sigma \leq \tau \longleftrightarrow \sigma < \tau \vee \sigma = \tau$ (Kleiner-Gleich-Relation),
- $\sigma \geq \tau \longleftrightarrow \sigma > \tau \vee \sigma = \tau$ (Größer-Gleich-Relation),
- $c \mid \sigma \longleftrightarrow \sigma \equiv_c 0$ (Teilerrelation),
- $\sigma \neq \tau \longleftrightarrow \neg \sigma = \tau$ (negierte Gleichheitsrelation),
- $\sigma \not\equiv_m \tau \longleftrightarrow \neg \sigma \equiv_m \tau$ (negierte Kongruenzrelation),
- $c \nmid \sigma \longleftrightarrow \neg \sigma \equiv_c 0$ (negierte Teilerrelation),

wobei σ, τ Terme und c, m Konstanten > 1 sind.

Die hier aufgeführten Relationen werden in die Sprache mit aufgenommen.

Erweiterung der Funktionsmenge

Da die hier behandelte Presburger-Arithmetik der Struktur $\mathbb{Z}^<$ zugrundeliegt, hat man auch negative Zahlen zur Verfügung. Also spricht auch nichts dagegen, das Minus $-$ als unäres Funktionssymbol einzuführen.

Das Funktionssymbol $+$ als n -stelliges Symbol mit $n > 0$ wird ebenfalls eingeführt, um sich unnötige Klammersetzungen ersparen zu können.

Der Term $\sigma + 1$ kann als Sukzessor von σ betrachtet und mit σ' notiert werden.

Der Term $\sigma + \sigma$ kann als Produkt mit der Konstante 2 betrachtet und mit $2 * \sigma$ notiert werden.

Das neue Funktionssymbol $*$ kann auch als n -stelliges Symbol mit $n > 1$ eingeführt werden, wobei ein Produkt nicht mehr als eine einzige Variable enthalten darf, um dies zu prüfen, kann man einen *Semantikcheck* implementieren.

Beachtung der Qualitätsanforderungen

Die *Robustheit* eines Programms kann man eigentlich gar nicht richtig garantieren. Ein Programm ist nicht robust, wenn es Programmfehler gibt, die das laufende Programm in einen inkonsistenten Zustand versetzt.

Verwendet man aber eine typensichere, objektorientierte Hochsprache, die eine saubere Ausnahmenbehandlung ermöglicht, kann man die Robustheit eines Programms so einigermaßen sicherstellen.

Aus diesem Grunde fällt die Wahl der Programmiersprache für die Realisierung des Entscheidungsprogramms auf Java.

Die *Korrektheit* eines Programms (Programm arbeitet gemäß seiner Spezifikation) kann man auch nicht garantieren. Bekanntlich kann nach Dijkstra kein Programm korrekt gemäß seiner spezifizierten Anforderung sein. Man kann als Entwickler mit einer ordentlichen Programmverifikation nur sein Bestes geben, das Programm nahezu korrekt zu halten.

Die *Benutzerfreundlichkeit* wird durch die geeignete Wahl der Sprache gegeben.

2.3 Programmübersicht

Vorliegende Arbeit realisiert das von Monk skizzierte Entscheidungsverfahren mit der Programmiersprache Java. Das Programm besitzt im Wesentlichen die Form eines Interpreters.

In diesem Abschnitt werden alle notwendigen Programmbestandteile erklärt.

2.3.1 Frontend

Frontend ist die Interpretationskomponente, die eine Inputformel in einen entsprechenden abstrakten Syntaxbaum (AST) überführt. Um dies realisieren zu können, muss für die Menge der Inputformeln eine Sprache existieren. Die für das Entscheidungsprogramm verwendete Sprache wird im Folgenden als $\mathcal{L}_{\mathbb{Z}^<}$ bezeichnet und wird an späterer Stelle definiert. Liegt die Inputformel nicht in dieser Sprache wird eine Fehlermeldung zurückgegeben.

Die zentralen Komponenten des Frontend heißen *Scanner*, *Parser* und *ParseTreeKit*.

Der Scanner wandelt die Inputformel, der als Bytestrom vorliegt, in einen Tokenstrom (Liste von Terminalen der Sprache $\mathcal{L}_{\mathbb{Z}^<}$) um. Der Parser erzeugt aus dem vom Scanner erkannten Tokenstrom mit Hilfe des *ParseTreeKit* gemäß der Sprache $\mathcal{L}_{\mathbb{Z}^<}$ einen abstrakten Syntaxbaum.

2.3.2 Proving

Proving ist die Entscheidungskomponente, die einen abstrakten Syntaxbaum in einen eventuell verschiedenen, abstrakten Syntaxbaum transformiert, der für die weitere Bear-

beitung erforderlich ist. Die Komponente *Proving* enthält alle erforderlichen Methoden, um die von Monk vorgeschlagenen Transformationen realisieren zu können.

2.3.3 Test

Ohne Tests funktioniert Software eigentlich nie!

Test ist die Verifikationskomponente, mit der stichprobenartig geprüft wird, ob das Entscheidungsprogramm auch wirklich das tut, was es tun soll.

Die Komponente *Test* prüft jede Methode der Komponenten *Frontend* und *Proving* besonders umfassend, da die *Korrektheit* des Entscheidungsprogramms ein gefordertes Qualitätsmerkmal ist.

Die Tests werden mit dem Framework *JUnit* realisiert. Es stellt die einzige Fremdkomponente des Entscheidungsprogramms dar.

2.3.4 UI

UI ist die Benutzerschnittstelle (User-Interface) des Entscheidungsprogramms. Ein Programm macht wenig Sinn, wenn keine Möglichkeit zur Nutzung gegeben ist.

2.4 Sprache $\mathcal{L}_{\mathbb{Z}^<}$

Die Sprache $\mathcal{L}_{\mathbb{Z}^<}$ umfasst die Inputformeln φ sowie für die Outputformeln φ .

Die zugrundeliegende Grammatik dieser Sprache ist *kontextfrei* und wird mittels *Backus-Naur-Form* wie folgt definiert.

```

//Produktionen
equivalences ::= implications { "<->" implications } .
implications ::= disjunctions { "->" disjunctions } .
disjunctions ::= conjunctions { ("or"|";") conjunctions } .
conjunctions ::= formula { ("and"|"&") formula } .
formula
  ::= "not" formula
  | "forall" VAR { "," VAR } [":"|"."] formula
  | "exists" VAR { "," VAR } [":"|"."] formula
  | "true"
  | "false"
  | sum {
    | "=" // gleich
    | "!="|"<>" // ungleich
    | "<"|">"|"<="|">=" // klar!
    | "#" NAT // kongruent
    | "!#" NAT // nicht kongruent
    | "|" // teilt
    | "!"|" // teilt nicht
  } sum }+
  | "(" equivalences ")" .
sum ::= ["+"|"^-"] product { ("+"|"^-") product } .
product ::= successor { ["*"] successor } .
successor ::= term { "" } .
term ::= NAT | VAR | "(" sum ")" .

// Startsymbol einer Formel
equivalences.

// Startsymbol eines Terms
sum.

// Besondere Terminale
VAR steht für einen typischen Variablenbezeichner.
NAT steht für eine natürliche Zahl.

```

Listing 2.1: Kontextfreie Grammatik zur Sprache $\mathcal{L}_{\mathbb{Z}^<}$

In dieser Definition werden streng genommen zwei Sprachen definiert, zum Einen die bereits erwähnte Sprache $\mathcal{L}_{\mathbb{Z}^<}$ unter Nutzung des Startsymbols *equivalences* und zum Anderen die Sprache der Terme der Presburger–Arithmetik unter Nutzung des Startsymbols *sum*.

2.5 Syntaxbaum

Die Aufgabe des *Frontend* ist es ja, eine Inputformel in einen entsprechenden, abstrakten Syntaxbaum zu überführen. Der Parser der Komponente *Frontend* stellt sicher, dass ein Syntaxbaum bezüglich der zugrundeliegenden Inputformel der Inputformel entspricht. Ist im Folgenden von einer Formel oder von einem Term die Rede, so ist indirekt auch die Rede vom entsprechenden Syntaxbaum.

Es gibt drei abstrakte Knotentypen des Syntaxbaumes gemäß des Entwurfsmusters *Composite–Pattern*:

- **Node**: Jeder Knoten ist vom Typ *Node*. *Node* bringt die Basisfunktionalität eines Syntaxbaumknotens mit.
- **LeafNode**: Manche Knoten enthalten keine Kindelemente wie zum Beispiel nullstellige Funktionen oder Relationen.
- **CompositeNode**: Andere Knoten müssen Kindelemente, die vom Knotentyp *Node* sind, enthalten wie zum Beispiel mehrstellige Funktionen oder Relationen.

Im Folgenden werden für insgesamt 23 geeignete, ausgezeichnete, sprachliche Fragmente entsprechende, konkrete Knotentypen des Syntaxbaumes definiert werden. Diese Knotentypen erben entweder von *LeafNode* oder von *CompositeNode*.

2.5.1 Boolesche Knotentypen

- **EquivalencesNode** repräsentiert n -stellige Boolesche Äquivalenzoperationen

$$\varphi_1 \longleftrightarrow \cdots \longleftrightarrow \varphi_n.$$

- **ImplicationsNode** repräsentiert n -stellige Boolesche Implikationsoperationen

$$\varphi_1 \longrightarrow \cdots \longrightarrow \varphi_n.$$

Eine mehrstellige Implikation wird vom Entscheidungsprogramm als rechts–assoziativ angesehen.

- **ConjunctionsNode** repräsentiert n -stellige Boolesche Konjunktionsoperationen

$$\varphi_1 \wedge \cdots \wedge \varphi_n.$$

- **DisjunctionsNode** repräsentiert n -stellige Boolesche Disjunktionsoperationen

$$\varphi_1 \vee \cdots \vee \varphi_n.$$

- **NegationNode** repräsentiert einstellige Boolesche Negationsoperationen

$$\neg \varphi.$$

- **BooleanNode** repräsentiert nullstellige Wahrheitswerte, von denen es nur zwei gibt:

$$true, false.$$

true notiert Monk als $0 = 0$.

2.5.2 Relationsknotentypen

- **EqualNode** repräsentiert zweistellige Gleichheitsrelationen

$$\sigma = \tau.$$

- **NotEqualNode** repräsentiert zweistellige negierte Gleichheitsrelationen

$$\sigma \neq \tau.$$

- **LowerThanNode** repräsentiert zweistellige Kleiner-Relationen

$$\sigma < \tau.$$

- **LowerEqualNode** repräsentiert zweistellige Kleiner-Gleich-Relationen

$$\sigma \leq \tau.$$

- **GreaterThanNode** repräsentiert zweistellige Größer-Relationen

$$\sigma > \tau.$$

- **GreaterEqualNode** repräsentiert zweistellige Größer-Gleich-Relationen

$$\sigma \geq \tau.$$

- **CongruentNode** repräsentiert zweistellige Kongruenzrelationen

$$\sigma \equiv_m \tau$$

mit einem festen Modul $m > 1$.

- **NotCongruentNode** repräsentiert zweistellige negierte Kongruenzrelationen

$$\sigma \not\equiv_m \tau$$

mit einem festen Modul $m > 1$.

- **DividesNode** repräsentiert einstellige Teilerrelationen

$$c \mid \sigma$$

mit einem festen Teiler $c > 1$.

- **NotDividesNode** repräsentiert einstellige Teilerrelationen

$$c \nmid \sigma$$

mit einem festen Teiler $c > 1$.

2.5.3 Termknotentypen

- **SumNode** repräsentiert n -stellige Summen

$$\sigma_1 + \cdots + \sigma_n$$

mit $n > 0$.

- **ProductNode** repräsentiert n -stellige Produkte

$$\sigma_1 * \cdots * \sigma_n$$

mit $n > 1$, wobei in diesem Produkt maximal eine Variable enthalten sein darf, sonst hätte man es mit einer Formel der Peano–Arithmetik zu tun.

- **MinusNode** repräsentiert einstellige, inverse Additionen

$$-\sigma.$$

- **SuccessorNode** repräsentiert einstellige Sukzessoren

$$\sigma'.$$

- **VariableNode** repräsentiert (nullstellige) Variablen

$$\alpha.$$

- **UnsignedNode** repräsentiert (nullstellige) Konstanten wie beispielsweise

$$\cdots, -2, -1, 0, 1, 2, 3, \cdots.$$

2.5.4 Quantorenknotentyp

- **QuantifierNode** repräsentiert einen Quantor, der entweder ein Existenzquantor \exists oder ein Allquantor \forall sein kann. Dieser Knoten enthält den Namen der zu bindenden Variable.

2.5.5 Knoteneigenschaften

Diese aufgelisteten Knotentypen sind so gut wie funktionslos. Doch besitzen sie zwei wichtige Eigenschaften:

- **Besuchbarkeit:** Jeder Knoten ist besuchbar gemäß des Entwurfsmusters *Visitor-
Pattern*.
- **Rekodierbarkeit:** Jeder Knoten kann sich mit samt seinen Kindknoten wieder als Code der Sprache $\mathcal{L}_{\mathbb{Z}^<}$ ausgeben.

Ein Knoten ist softwaretechnisch besuchbar, wenn er folgende Schnittstelle implementiert.

```
public interface Visitable {
    <R, A> R accept(Visitor<R, A> visitor, A argument);
}
```

Listing 2.2: Schnittstelle eines besuchbaren Knotens

2.6 Visitoren

Eine beliebige Operation, insbesondere eine Transformation, für den Syntaxbaum wird hier als *Visitor* gemäß des Entwurfsmusters *Visitor-
Pattern* realisiert. Dazu muss diese Operation folgende Schnittstelle implementieren:

```
public interface Visitor<R, A> {
    R visit(EquivalencesNode node, A argument);
    R visit(ImplicationsNode node, A argument);
    R visit(DisjunctionsNode node, A argument);
    R visit(ConjunctionsNode node, A argument);
    R visit(NegationNode node, A argument);
    R visit(QuantifierNode node, A argument);
    R visit(EqualNode node, A argument);
    R visit(NotEqualNode node, A argument);
    R visit(LowerThanNode node, A argument);
    R visit(LowerEqualNode node, A argument);
    R visit(GreaterThanNode node, A argument);
    R visit(GreaterEqualNode node, A argument);
    R visit(CongruentNode node, A argument);
    R visit(NotCongruentNode node, A argument);
    R visit(DividesNode node, A argument);
    R visit(NotDividesNode node, A argument);
    R visit(BooleanNode node, A argument);
    R visit(SumNode node, A argument);
    R visit(MinusNode node, A argument);
    R visit(ProductNode node, A argument);
    R visit(SuccessorNode node, A argument);
    R visit(VariableNode node, A argument);
    R visit(UnsignedNode node, A argument);
}
```

Listing 2.3: Schnittstelle eines Besuchers

Jede für das Entscheidungsverfahren erforderliche Visitoren überführt den Syntaxbaum in einen eventuell umstrukturierten Syntaxbaum je nach dem, was der jeweilige Visitor zu leisten hat. Unter Transformation \mathcal{T} ist eine Programmfunktion wie folgt zu verstehen:

$$\mathcal{T} : \mathcal{L}_{\mathbb{Z}^<} \longrightarrow \mathcal{L}_{\mathbb{Z}^<}$$

2.6.1 Proofer

Proofer implementiert die Schnittstelle *Visitor* und stellt die Haupttransformation des Entscheidungsprogramms dar. Eine Inputformel in Form eines Knotens *node* wird wie folgt in seine quantorenfreie Darstellung gebracht:

```
Node output = input;
output = Simplifier.visit(output);
output = BooleanReducer.visit(output);
if (!IsQuantifierFree.visit(output)) {
    output = QuantifierEliminator.visit(output, false);
    output = output.accept(this, argument); // Rekursion
}
return output;
```

Als Output ergibt sich gemäß der Anforderungsdefinition einer der folgenden Fälle:

- Ist $\psi = true$, so ist φ wahr.
- Ist $\psi = false$, so ist φ falsch.
- Sonst, so verfügt φ über freie Variablen, die nicht akzident wegfallen.

Der Visitor *Proofer* verwendet folgende *Visitoren*, die an späterer Stelle genauer beschrieben sind:

- **Simplifier** vereinfacht eine Formel. Bevor ein Eliminationsschritt auf die Formel losgelassen wird, sollte man aus Effizienzgründen die Formel durch Vereinfachungen möglichst gut vereinfachen.
- **BooleanReducer** vereinfacht die Boolesche Kombination einer Formel ebenfalls aufgrund der Effizienz.
- **IsQuantifierFree** prüft die Quantorenfreiheit einer Formel. Eine Quantorenelimination ist nicht besonders sinnvoll, wenn die Formel keine Quantoren (mehr) enthält.
- **QuantifierEliminator** eliminiert Quantoren einer Formel.

2.6.2 Simplifier

Die Hilfstransformation *Simplifier* implementiert die Schnittstelle *Visitor* und vereinfacht eine gegebene Formel oder einen gegebenen Term.

Beispielsweise wird mit Hilfe des *Simplifier* die umständlich formulierte Formel

$$1 + 2 - (-(x * 4'' - (4 - 3 * x))) = 0$$

in die wesentlich einfachere Darstellung

$$9 * x = 1$$

transformiert.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **Flutter** entfernt im Prinzip unnötige Klammerungen. Eine klammerlose Darstellung wird als Vereinfachung angesehen.
- **CanEvaluate** prüft, ob ein Term arithmetisch ausgewertet werden kann. Bevor man Terme auswertet, muss natürlich geprüft werden, ob diese Variablen enthalten.
- **Evaluator** wertet eine arithmetische Kombination mit Konstanten aus. Ein Term ohne Variablen wie zum Beispiel $2' * (2 + 3)$ kann man vereinfacht, als Konstante darstellen.
- **VariablesReducer** reduziert die Vorkommen der Variablen. Je weniger Variablen vorkommen, desto einfacher wird der Ausdruck.
- **ProductEliminator** wandelt Produkte in Summen um. Das Vereinfachungsverfahren eliminiert zunächst die Produkte, um bezüglich der Summen besser vereinfachen zu können.
- **SuccessorEliminator** ersetzt Sukzessoren durch Addition mit 1. Ein Sukzessor ist als syntaktischer Zucker anzusehen und ist im Weiteren nicht von Nöten.
- **MinusReducer** vereinfacht, indem doppelte Minussymbole entfernt werden.
- **FreeVariables** gibt alle ungebundenen Variablen zurück. Innerhalb eines jeden Terms gibt es nur freie Variablen, die erst außerhalb einer Relation abgebunden werden können. Die Hilfsfunktion *FreeVariables* ist eine reine Hilfsfunktion, die immer wieder gebraucht wird.

2.6.3 BooleanReducer

Der Visitor *BooleanReducer* ist eine Hilfstransformation und vereinfacht die Boolesche Kombination einer gegebenen Formel.

Beispielsweise wird mit Hilfe des *BooleanReducer* die Formel mit einem überflüssigen Booleschen Wert

$$1 = 1 \longleftrightarrow (2 = 2 \longrightarrow \neg\neg false)$$

in die einfachere Darstellung

$$1 = 1 \longleftrightarrow \neg 2 = 2$$

transformiert.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **ContainsClass** prüft, ob ein bestimmter Knotentyp (hier *BooleanNode*) vorhanden ist.
- **Basics** wandelt gegebene Formel in eine Boolesche Kombination von (Monks) Basisformeln um.

2.6.4 IsQuantifierFree

Der Visitor *IsQuantifierFree* ist eine Hilfsfunktion und prüft, ob eine gegebene Formel quantorenfrei ist.

Beispielsweise enthält die Formel

$$\neg(\exists\alpha.\alpha = 0)$$

mindestens einen Quantor $\mathcal{Q} \in \{\exists, \forall\}$.

2.6.5 QuantifierInsider

Bevor der Visitor *QuantifierEliminator* erklärt werden kann, muss sein Vorfahr *QuantifierInsider*, der die Quantorenelimination mundgerecht vorbereitet, erklärt werden.

Der Visitor *QuantifierInsider* ist eine Hilfstransformation und klopft einen Quantor so weit wie möglich in den Ausdruck hinein. Zielzustand: Die Formel muss dergestalt transformiert werden, dass alle inneren Quantoren mit dem dazugehörigen Bindungsbereich folgende mundgerechte Form besitzen:

$$\exists\alpha.(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n),$$

wobei hier die Formeln φ_1 bis φ_n Basisformeln sind. Jede dieser Basisformel darf nicht frei von der Variable α sein.

Beispielsweise wird mit Hilfe des *QuantifierInsider* die Formel

$$\exists\alpha.((1 = 1 \vee (2 = \alpha \vee (0 < \alpha \wedge \alpha < 3))) \vee 3 = 3 * \alpha)$$

in die Darstellung

$$1 = 1 \vee ((\exists\alpha.2 = \alpha) \vee (\exists\alpha.(0 < \alpha \wedge \alpha < 3))) \vee (\exists\alpha.3 = 3 * \alpha)$$

transformiert.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **Basics** wandelt eine Formel in eine Boolesche Kombination von Basisformeln um.
- **DNFer** wandelt eine Formel in eine disjunktive Normalform (DNF) um.
- **Existentializer** wandelt Allquantoren einer gegebenen Formel unter Einhaltung der Umwandlungsregeln in Existenzquantoren um. Für einen (Monkschen) Eliminationsschritt müssen die inneren Quantoren Existenzquantoren sein.
- **NotEliminator** klopft die Booleschen Negationen vollständig, gegebenenfalls werden die Basisformeln negiert, was wiederum eine Boolesche Kombination von Basisformeln gibt.
- **RemoveUnusedQuantifiers** entfernt Quantoren, deren jeweilige Variable nicht im Bindungsbereich vorkommt.

- **IsQuantifierFree** prüft die Quantorenfreiheit einer Formel.
- **RelationTransformer** formt eine Relation nach einer bestimmten Variable um.
- **VariableCounter** zählt die Vorkommen einer bestimmten Variable.

2.6.6 QuantifierEliminator

Der Visitor *QuantifierEliminator* ist eine Hilfstransformation und eliminiert einen inneren Existenzquantor des Inputbaums $\exists\alpha.\varphi$ mit einer beliebigen, aber quantorenfreien Formel φ .

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **Substituter**
- **Multiplier**
- **VariableCounter**
- **RelationTransformer**

Da der *QuantifierEliminator* von *QuantifierInsider* abgeleitet ist, beschränkt sich das Problem der Eliminierung eines inneren Existenzquantors auf das Problem

$$\exists\alpha.(\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n),$$

wobei die Formeln φ_1 bis φ_n Basisformeln sind. Jede einzelne dieser Basisformeln enthält die gebundene Variable α .

Die Konjunktion der Formeln φ_1 bis φ_n wird nach den Relationentypen ($=$, $<$, $>$, \equiv) geordnet:

$$\exists\alpha.(\varphi_{\wedge=} \wedge \varphi_{\wedge<} \wedge \varphi_{\wedge>} \wedge \varphi_{\wedge\equiv}),$$

wobei $\varphi_{\wedge=}$ eine Konjunktion der Gleichheitsrelationen, $\varphi_{\wedge<}$ eine Konjunktion der Kleiner-Relationen, $\varphi_{\wedge>}$ eine Konjunktion der Größer-Relationen und $\varphi_{\wedge\equiv}$ eine Konjunktion der Kongruenzrelationen darstellen.

Es ergibt sich folgende Formel Φ :

$$\begin{aligned} \exists\alpha.(& n_0 * \alpha = \xi_0 \wedge \cdots \wedge n_{i-1} * \alpha = \xi_{i-1} \\ & \wedge n_i * \alpha < \xi_i \wedge \cdots \wedge n_{j-1} * \alpha < \xi_{j-1} \\ & \wedge n_j * \alpha > \xi_j \wedge \cdots \wedge n_{k-1} * \alpha > \xi_{k-1} \\ & \wedge n_k * \alpha \equiv_{m_k} \xi_k \wedge \cdots \wedge n_{h-1} * \alpha \equiv_{m_{h-1}} \xi_{h-1}), \end{aligned}$$

wobei $0 \leq i \leq j \leq k \leq h > 0$, $n_0 > 0, \dots, n_{h-1} > 0$, $m_k > 1, \dots, m_{h-1} > 1$ und α -freie Terme ξ_0, \dots, ξ_{h-1} .

Jede Relation wird dergestalt erweitert, dass auf der linken Seite ein Produkt mit der Variable α und dem konstanten Faktor p steht, wobei p das kleinste, gemeinsame Vielfache (kgV) von n_0, \dots, n_{h-1} darstellt ($p = \text{kgV}(n_0, \dots, n_{h-1})$). Es ergibt sich die Formel Φ' :

$$\begin{aligned} \exists \alpha. (& p * \alpha = \vartheta_0 \wedge \dots \wedge p * \alpha = \vartheta_{i-1} \\ & \wedge p * \alpha < \vartheta_i \wedge \dots \wedge p * \alpha < \vartheta_{j-1} \\ & \wedge p * \alpha > \vartheta_j \wedge \dots \wedge p * \alpha > \vartheta_{k-1} \\ & \wedge p * \alpha \equiv_{m_k} \vartheta_k \wedge \dots \wedge p * \alpha \equiv_{m_{h-1}} \vartheta_{h-1}), \end{aligned}$$

wobei $\vartheta_0 = \Delta(p/n_0) * \xi_0, \dots, \vartheta_{h-1} = \Delta(p/n_{h-1}) * \xi_{h-1}$.

Im nächsten Schritt ergibt sich folgende Formel Φ'' :

$$\begin{aligned} \exists \alpha. (& \alpha = \vartheta_0 \wedge \dots \wedge \alpha = \vartheta_{i-1} \\ & \wedge \alpha < \vartheta_i \wedge \dots \wedge \alpha < \vartheta_{j-1} \\ & \wedge \alpha > \vartheta_j \wedge \dots \wedge \alpha > \vartheta_{k-1} \\ & \wedge \alpha \equiv_{m_k} \vartheta_k \wedge \dots \wedge \alpha \equiv_{m_{h-1}} \vartheta_{h-1} \\ & \wedge \alpha \equiv_p 0). \end{aligned} \tag{2.1}$$

Die Zeile (1.1) sagt aus, dass α durch p teilbar ist. Ist $p = 1$ kann die neue Zeile auch weggelassen werden.

Die Formel Φ'' ist immer noch nach Relationen geordnet:

$$\exists \alpha. (\varphi_{\wedge=} \wedge \varphi_{\wedge<} \wedge \varphi_{\wedge>} \wedge \varphi_{\wedge\equiv}).$$

Im nächsten Schritt geht es darum, die Anzahl der Konjunktionsglieder von $\varphi_{\wedge\equiv}$ auf (maximal) eine Kongruenzrelation zu reduzieren.

Man pickt sich zwei Kongruenzrelationen ($\alpha \equiv_m \sigma, \alpha \equiv_n \tau$) heraus. Die Konjunktion dieser beiden Kongruenzrelationen lässt sich in

$$\sigma \equiv_{\text{ggT}(m,n)} \tau \wedge \alpha \equiv_p \Delta(c * p/m) * \sigma + \Delta(d * p/n) * \sigma$$

umrechnen, wobei p das kleinste, gemeinsame Vielfache (kgV) von m und n darstellt ($p = \text{kgV}(m, n)$) und die Konstanten c, d ermittelt werden, derart dass $c * p/m + d * p/n = 1$ gilt. Die zwei herausgepickten Kongruenzrelationen werden gegen die neu gewonnene Kongruenzrelation $\alpha \equiv_p \Delta(c * p/m) * \sigma + \Delta(d * p/n) * \sigma$ ausgetauscht.

Die Anzahl der Kongruenzrelation in $\varphi_{\wedge\equiv}$ verkleinert sich mit diesem Schritt um eins. Die α -freie Kongruenzrelation $\sigma \equiv_{\text{ggT}(m,n)} \tau$, die sich dabei ergibt, wird dem Eliminationsresultat konjunktiv angehängt.

Dieser Berechnungsschritt wird solange wiederholt bis (maximal) eine einzige Kongruenzrelation in $\varphi_{\wedge\equiv}$ steht.

Im letzten Schritt wird in folgende Fälle unterschieden:

- Ist $\varphi_{\wedge=}$ nicht leer, so pickt man sich eine Basisformel heraus $\alpha = \sigma$ und substituiert jedes α -Vorkommen mit σ .

- Andernfalls
 - Ist $\varphi_{\wedge <}$ oder $\varphi_{\wedge >}$ leer, ergibt das Eliminationsresult unabhängig von $\varphi_{\wedge \equiv}$ *true*.
 - Andernfalls, wird folgende, komplizierte Ergebnisformel ψ konstruiert.

$$\psi = \bigvee_{\substack{\alpha < \sigma \in \varphi_{\wedge <} \\ \alpha > \tau \in \varphi_{\wedge >}}} \bigwedge_{\substack{\alpha < \eta \in \varphi_{\wedge <} \\ \alpha > v \in \varphi_{\wedge >}}} (\sigma < v+1 \wedge \eta < \tau+1 \wedge (\bigvee_{0 \leq e < m} (\tau+e+1 < \sigma \wedge \tau+e+1 \equiv_m \xi))),$$

wobei m das Modul und ξ der α -freie Term der einzigen Kongruenzrelation in $\varphi_{\wedge \equiv}$ sind.

2.6.7 RemoveUnusedQuantifiers

Der Visitor *RemoveUnusedQuantifiers* ist eine Hilfstransformation und entfernt überflüssige Quantoren. Es stellt eine simple Form der Quantorenelimination dar, wenn die gebundene Variable nicht im Bindungsbereich vorkommt, kann der Quantor einfach weggelassen werden.

Beispielsweise wird mit Hilfe des *RemoveUnusedQuantifiers* die Formel

$$\exists \alpha. \forall \beta. \alpha = 0$$

in die Darstellung

$$\exists \alpha. \alpha = 0$$

transformiert.

Dabei kommt folgender *Visitor* zum Einsatz:

- **FreeVariables** gibt eine Liste der freien Variablen zurück. Ist keine dieser Variablen namentlich identisch mit der vom Quantor zu bindende Variable, kann dieser Quantor restlos entfernt werden.

2.6.8 Basics

Der Visitor *Basics* ist eine Hilfstransformation und wandelt eine Formel in eine Boolesche Kombination (mit \neg , \wedge und \vee) von Basisformeln um.

Beispielsweise wird mit Hilfe des *Basics* die Formel

$$2 \mid 4 \wedge 2 \neq 3 \vee 2 \nmid 5$$

in die Darstellung

$$4 \equiv_2 0 \wedge (2 < 3 \vee 2 > 3) \vee 5 + 1 \equiv_2 0$$

transformiert.

2.6.9 DNFer

Der Visitor *DNFer* ist eine Hilfstransformation und wandelt eine Formel in eine disjunktive Normalform (DNF) um.

Beispielsweise wird mit Hilfe des *DNFer* die Formel

$$1 = 1 \wedge (2 = 2 \vee 3 = 3) \wedge 4 = 4$$

in die Darstellung

$$(2 = 2 \wedge 1 = 1 \wedge 4 = 4) \vee (3 = 3 \wedge 1 = 1 \wedge 4 = 4)$$

transformiert.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **Basics**
- **NotEliminator**
- **Flutter**

2.6.10 Existentializer

Der Visitor *Existentializer* ist eine Hilfstransformation und wandelt Allquantoren einer gegebenen Formel unter Einhaltung der Umwandlungsregeln in Existenzquantoren um.

Beispielsweise wird mit Hilfe des *Existentializer* die Formel

$$\forall \alpha. (0 < \alpha \longrightarrow \forall \beta. (\beta < 0 \longrightarrow \alpha \neq \beta))$$

in die Darstellung

$$\neg(\exists \alpha. (\neg(0 < \alpha \longrightarrow (\neg(\exists \beta. (\neg(\beta < 0 \longrightarrow \alpha \neq \beta)))))))$$

transformiert.

2.6.11 Flutter

Der Visitor *Flutter* ist eine Hilfstransformation und klopft Ausdrücke flach, d.h. entfernt im Prinzip unnötige Klammerungen.

Beispielsweise wird mit Hilfe des *Flutter* die Formel

$$(true \wedge 1 = 2 * (x * 5)) \wedge (false \wedge 2 = 2 + (3 + 4))$$

in die Darstellung

$$true \wedge 1 = 2 * x * 5 \wedge false \wedge 2 = 2 + 3 + 4$$

transformiert.

2.6.12 NotEliminator

Der Visitor *NotEliminator* ist eine Hilfstransformation und eliminiert die Boolesche Negation durch Reinklopfen.

Beispielsweise wird mit Hilfe des *NotEliminator* die Formel

$$\neg true \longleftrightarrow \neg(2 \neq 3 \wedge 2 \mid 4)$$

in die Darstellung

$$false \longleftrightarrow (2 = 3 \vee 4 + 1 \equiv_2 0)$$

transformiert.

Dabei kommt der Visitor **Basics** zum Einsatz. Mit **Basics** reduziert sich das Problem die Negation einer beliebigen Relation auf die Negation einer Basisrelation wie folgt:

- $\neg \sigma = \tau \longleftrightarrow \sigma < \tau \vee \sigma > \tau$
- $\neg \sigma < \tau \longleftrightarrow \sigma = \tau \vee \sigma > \tau$
- $\neg \sigma > \tau \longleftrightarrow \sigma = \tau \vee \sigma < \tau$
- $\neg \sigma \equiv_m \tau \longleftrightarrow \sigma + 1 \equiv_m \tau \vee \dots \vee \sigma + \Delta(m - 1) \equiv_m \tau$

2.6.13 RelationSideSwitcher

Der Visitor *RelationSideSwitcher* ist eine Hilfstransformation und vertauscht die Seiten einer Relation unter Einhaltung des Wahrheitswertes.

Beispielsweise wird mit Hilfe des *RelationSideSwitcher* die Formel

$$\alpha \leq \beta$$

in die Darstellung

$$\beta \geq \alpha$$

transformiert.

2.6.14 RelationTransformer

Der Visitor *RelationTransformer* ist eine Hilfstransformation und formt eine Relation nach einer bestimmten Variable um.

Beispielsweise wird mit Hilfe des *RelationTransformer* die Formel

$$9 - 2 * \alpha > \alpha + 3$$

in die Darstellung

$$3 * \alpha < 6$$

transformiert.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **RelationTransformer**
- **Substituter** überflüssige Variablen werden durch 0 ersetzt.
- **Simplifier**
- **FreeVariables**

2.6.15 MinusReducer

Der Visitor *MinusReducer* ist eine Hilfstransformation und reduziert die Vorkommen von überflüssigen Subtraktionen; ein doppeltes Minus ergibt wiederum ein Plus.

Beispielsweise wird mit Hilfe des *MinusReducer* die Formel

$$-(-(-4)) \leq 5 + (2 - (-3))$$

in die Darstellung

$$-4 \leq 5 + (2 + 3)$$

transformiert.

2.6.16 Multiplier

Der Visitor *Multiplier* ist eine Hilfsfunktion und versieht einen Term mit einem Koeffizienten bzw. Faktor.

Beispielsweise wird mit Hilfe des *Multiplier* unter Angabe eines konstanten Faktors c der Term

$$\alpha + 2$$

wie folgt erweitert:

$$c * (\alpha + 2).$$

2.6.17 ProductEliminator

Der Visitor *ProductEliminator* ist eine Hilfstransformation und eliminiert die Produkte des Inputbaumes, in dem die Produkte in Summen umgeschrieben werden.

Beispielsweise wird mit Hilfe des *ProductEliminator* die Formel

$$3 * x = 2 * (5 * y)$$

in die Darstellung

$$x + x + x = (y + y + y + y + y) + (y + y + y + y + y)$$

transformiert.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **CanEvaluate**
- **Evaluator**

2.6.18 Renamer

Der Visitor *Renamer* ist eine Hilfsfunktion und benennt bestimmte, ungebundene Variablen um (Umbenennung durch Substitution). Die Hilfsfunktion *Renamer* erbt seine Funktionalität komplett von *Substituter*.

Beispielsweise wird mit Hilfe des *Renamer* unter Angabe eines alten Variablennamens α und eines neuen Variablennamens γ die Formel

$$2 * \alpha + \beta = \alpha$$

in die Darstellung

$$2 * \gamma + \beta = \gamma$$

transformiert.

2.6.19 Substituter

Der Visitor *Substituter* ist eine Hilfstransformation und substituiert bestimmte, ungebundene Variablen durch einen neuen Term.

Beispielsweise wird mit Hilfe des *Substituter* unter Angabe eines alten Variablennamens α und eines neuen Terms $3 * (5 + 7)$ die Formel

$$2 * \alpha + \beta = \alpha$$

in die Darstellung

$$2 * (3 * (5 + 7)) + \beta = 3 * (5 + 7)$$

transformiert.

2.6.20 SuccessorEliminator

Der Visitor *SuccessorEliminator* ist eine Hilfstransformation und eliminiert Sukzessoren durch entsprechende Additionen mit Konstanten.

Beispielsweise wird mit Hilfe des *SuccessorEliminator* die Formel

$$0 < \alpha \wedge \alpha < \beta \wedge \beta < \alpha'' \wedge \alpha'' < 0'''$$

in die Darstellung

$$0 < \alpha \wedge \alpha < \beta \wedge \beta < \alpha + 2 \wedge \alpha + 2 < 4$$

transformiert.

2.6.21 VariablesReducer

Der Visitor *VariablesReducer* ist eine Hilfstransformation und reduziert gegebenenfalls überflüssige Variablen.

Beispielsweise wird mit Hilfe des *VariablesReducer* die Formel

$$2 * x - (y + x + 3 + x) = -y + 2 * z$$

in die Darstellung

$$2 * 0 - (0 + 0 + 3 + 0) = -0 + 2 * z$$

transformiert.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **FreeVariables**
- **VariableCounter**
- **Substituter**

2.6.22 BoundVariables

Der Visitor *BoundVariables* ist eine Hilfsfunktion und ermittelt die gebundenen Variablen.

Beispielsweise wird mit Hilfe des *BoundVariables* die Formel

$$\alpha = 0 \wedge \forall \alpha \geq \gamma \vee \exists \beta. \alpha + \beta = 0$$

folgende Variablenliste

$$\{\beta\}$$

ermittelt.

2.6.23 FreeVariables

Der Visitor *FreeVariables* ist eine Hilfsfunktion und ermittelt die freien, ungebundenen Variablen, das Pendant zu *BoundVariables*.

Beispielsweise wird mit Hilfe des *FreeVariables* die Formel

$$\alpha = 1 \wedge \forall \alpha > \gamma \wedge \exists \beta. \alpha + \beta = 0$$

folgende Variablenliste

$$\{\alpha, \gamma\}$$

ermittelt.

2.6.24 CanEvaluate

Der Visitor *CanEvaluate* ist eine Hilfsfunktion und prüft, ob ein Term ausgewertet werden kann. Ein Term kann nur ausgewertet werden, wenn keine Variablen enthalten sind.

2.6.25 Evaluator

Der Visitor *Evaluator* ist eine Hilfsfunktion und wertet einen Term aus und gibt die Auswertung als Konstante zurück.

Beispielsweise ergibt sich mit Hilfe des *Evaluator* aus dem Term $4 + 4 * 5$ die Konstante 24.

Dabei kommt folgender *Visitor* zum Einsatz:

- **ContainsClass** prüft nach, ob sich im zu auszuwertenden Term eine Variable befindet.

2.6.26 ClassCounter

Der Visitor *ClassCounter* ist eine Hilfsfunktion und zählt einen bestimmten Knotentyp im gegebenen Syntaxbaum.

Beispielsweise erhält man mit Hilfe des *ClassCounter* zur Formel

$$true \wedge false$$

unter Angabe des Knotentypen *BooleanNode* als Ergebnis 2 und unter Angabe des Knotentypen *ConjunctionsNode* als Ergebnis 1. Für andere Knotentypen, die es in diesem Beispiel nicht gibt, erhält man jeweils als Ergebnis 0.

2.6.27 ContainsClass

Der Visitor *ContainsClass* ist eine Hilfsfunktion und prüft, ob ein gegebener Syntaxbaum einen bestimmten Knotentyp enthält.

2.6.28 SemantikChecker

Der Visitor *SemantikChecker* ist eine Hilfsfunktion und prüft, ob es sich bei der Inputformel um eine gültige Formel der Presburger Arithmetik handelt. Es müssen folgende Bedingungen gelten:

- der Teiler einer Teilt-Relation muss konstant sein.
- ein Produkt darf maximal nur eine Variable enthalten.

Beispielsweise meckert der *SemantikChecker*, wenn die Inputformel den Term $x*y$ enthält.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **CanEvaluate** prüft, ob der Teiler bzw. das Produkt sich als Konstante darstellen lässt.
- **Evaluator** ermittelt den Konstantenwert des Teilers bzw. des Produkts.

2.6.29 VariableCounter

Der Visitor *VariableCounter* ist eine Hilfsfunktion und zählt die Vorkommen einer bestimmten Variable im gegebenen Syntaxbaum.

Dabei kommen folgende *Visitoren* sinnvoll zum Einsatz:

- **FreeVariables**
- **ProductEliminator**

Kapitel 3

Schluss

3.1 Zusammenfassung

Vorliegende Arbeit zeigt dem Leser eine Möglichkeit auf, wie ein Entscheidungsverfahren für die Presburger–Arithmetik im Großen und Ganzen implementiert werden kann. Es entstand ein ganz brauchbares Entscheidungsprogramm, welches dem Anwender ermöglicht eine beliebige Formel φ der Presburger–Arithmetik einzugeben. Die Formel φ kann entschieden, bewiesen oder widerlegt, werden. Des Weiteren stehen dem Anwender noch weitere Transformationen für die gegebene Formel φ zur Verfügung wie zum Beispiel Transformationen zur Vereinfachung der Formel. Die gegebene Formel φ , wenn diese wohlgeformt gemäß der hier definierten Sprache $\mathcal{L}_{\mathbb{Z}^<}$ ist, wird stets als Syntaxbaum sichtbar gemacht. Dabei kann der Anwender auch auf jeden Knoten (Teilformeln sowie Teilterme) aus einer Liste der zur Verfügung stehenden Transformationen eine Transformation anwenden.

Die Realisierung des Entscheiders (Entscheidungsprogramm) richtet sich nach einer sehr prägnanten Skizzierung zum Entscheidungsverfahren für die Presburger–Arithmetik von Monk, siehe [Mon76]. Das Entscheidungsverfahren wird mit der Methode der nicht–totalen Quantorenelimination umgesetzt. Zu diesem Zweck werden zweistellige Kongruenzrelationen \equiv_m mit einem festen Modul m als Basisrelation in die Symbolmenge der Struktur der Presburger–Arithmetik aufgenommen.

Verwunderlich war während der Entwicklung des Entscheiders, dass eine nur fünfseitige Skizzierung eines möglichen Entscheidungsverfahrens für die Presburger–Arithmetik eine umfangreiche Implementierung (viel Code) erforderlich machte.

Bei der Anwendung des Entscheidungsprogramms stellt man ziemlich schnell fest, dass die Eingabesprache eine Art *Abfragesprache* (Query–Language) für Formeln der Presburger–Arithmetik ist. Formuliert man eine Formel mit freien Variablen liefert der Entscheider eine Lösung (Belegung) für die freien Variablen, zwar meist nicht die kleinstmögliche Lösung, aber immerhin. Nach Möglichkeit sollte man diese Abfragesprache als **Presburger–Arithmetic–Query–Language** (kurz: **PAQL**) bezeichnen!

Nichts desto trotz, vorliegender Entscheider kann für weitere Arbeiten unter dem Aspekt

der Effizienzmachung durchaus aufgegriffen werden. Das Entscheidungsverfahren für die Presburger–Arithmetik hat als untere Zeitschranke 2^{2^n} , wobei n die Anzahl der Symbolvorkommnisse der Inputformel ist, siehe [MY78]. Das Entscheidungsprogramm sollte derart umgestaltet werden, dass es sich in pathologischen Fällen dieser unteren Zeitschranke annähert. Des Weiteren wären auch sprachliche Erweiterungen interessant wie zum Beispiel die Erweiterung eines einstelligen Funktionssymbols für den *Absolutbetrag*. Wünschenswert wäre auch, eine sprachliche Eingabemöglichkeit für n -stellige Operationen mit Laufvariablen zur Verfügung zu haben. Neben den Erweiterungen für die **PAQL** würde man sich auch eine Definitionssprache (**PADL**) wünschen, die es einem Anwender ermöglicht weitere Funktions- und Relationssymbole hinzuzudefinieren.

Literaturverzeichnis

- [Ape66] Harry Apelt. Zeitschrift für mathematische Logik und Grundlagen der Mathematik, 1966.
- [Mon76] J. Donald Monk. *Mathematical Logic*. Springer – Verlag, 1976.
- [MY78] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Elsevier, North – Holland, 1978.
- [Rau96] Wolfgang Rautenberg. *Einführung in die Mathematische Logik*. Vieweg Verlagsgesellschaft mbH, 1996.
- [Smo91] Craig Smorynski. *Logical Number Theory 1*. Springer – Verlag, 1991.

