Spezifikation und maschinelle Verifikation von Konstantenfaltung in Übersetzern

Jan Olaf Blech

Studienarbeit

Betreuerin
Dr. Sabine Glesner

Verantwortlicher
Prof. Dr. Goos

9. Mai 2003

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Studienarbeit befasst sich mit der Spezifikation und der maschinellen Verifikation der Konstantenfaltung, wie sie in der Optimierungsphase von Übersetzern stattfindet.

Konstantenfaltung ist das Ausrechnen konstanter Ausrücke zur Übersetzungszeit. Rechnerarithmetiken und Sprachstandards schreiben in der Regel bestimmte Genauigkeiten beim Ausrechnen von Ausdrücken vor, die von Rechner zu Rechner und von Datentyp zu Datentyp in jeder Sprache verschieden sein können. Probleme können auftreten, wenn Zielmaschinenarithmetik beziehungsweise die von einem Sprachstandard vorgeschriebene Arithmetik und die Übersetzerarithmetik verschieden sind. Dies kann bei den sogenannten Cross-Compilern der Fall sein. Cross-Compiler übersetzen für eine andere Plattform als die, auf der sie laufen. Korrekt ist die Konstantenfaltung nur dann, wenn das gleiche Ergebnis herauskommt, wie wenn die Berechnung direkt auf der Zielmaschine, beziehungsweise mit den Regeln, die der Programmiersprachenstandard vorschreibt, durchgeführt worden wäre.

Hier befassen wir uns mit allgemeinen algebraischen Kriterien unter denen Arithmetiken durch andere ersetzt werden können, ohne das Ergebnis zu verfälschen. Speziell geben wir so ein Kriterium für Ganzzahlarithmetiken an. Wir vergleichen die Ganzzahlarithmetiken von C und Java und zeigen in einer Fallstudie wann Konstantenfaltung in gängigen Übersetzern durchgeführt wird. Weiterhin geben wir einen Ausblick auf die Probleme, die beim Ersetzen von Fließkommaarithmetiken auftreten.

1.1 Beispiel und Motivation

Folgendes Beispiel könnte irgendwo in einem Programm auftauchen. Wir können Konstantenfaltung in diesem Beispiel benutzen, um den Wert von ERDDURCHMESSER schon zur Übersetzungszeit auszurechnen. Er ist nämlich nur von Konstanten abhängig.

```
ERDUMFANG = 40000 (*km*)
PI = 3.14159265 ERDDURCHMESSER = ERDUMFANG / PI
```

ERDDURCHMESSER bekommt nämlich gerade den Wert des Ausdrucks ERDUMFANG / PI zugewiesen. Dieser kann schon vom Übersetzer berechnet werden. Es ist dann nicht nötig, ihn bei jedem Programmlauf neu zu berechnen.

Es kann jedoch vorkommen, dass der Übersetzer in einer anderen Sprache geschrieben wurde als die, für die er übersetzt. Auch kann der Übersetzer für eine andere Maschine (mit anderem Prozessor, anderer Registerbreite) übersetzen.

Dies beinhaltet einen gewissen Indeterminismus. Mit welchen arithmetischen Eigenschaften soll ERDDURCHMESSER = ERDUMFANG / PI berechnet werden? Die, die die Maschine auf der der Übersetzer läuft aufweist? Oder soll der Ausdruck in Zielmaschinenarithmetik bzw. mit den Regeln der Sprache, in der das Programm geschrieben wurde, ausgewertet werden? Vielleicht kann man sich darauf einigen, dass man immer der genaueren Arithmetik den Vorzug gibt? Oder sollte man immer mit voller Genauigkeit arbeiten?

Was ist aber, wenn es zwei Übersetzer gibt und der eine führt Konstantenfaltung durch, der andere nicht. Dann könnte das gleiche Quellprogramm in zwei verschiedenen Übersetzungen auf der Zielmaschine unterschiedliches Laufzeitverhalten zeigen! Dieses Verhalten kann nicht erwünscht sein. Wir können Konstantenfaltung daher nur als korrekt ansehen, wenn das Ergebnis immer das gleiche ist. Egal ob Konstantenfaltung durchgeführt wurde oder nicht.

Wenn die Zielmaschine zum Beispiel überhaupt keine Gleitkommazahlen unterstützt und die Nachkommastellen von PI einfach abschneidet (und der Sprachstandard dies zuläßt), ERDDURCHMESSER also den Wert 12000 hätte, dann müßte ein Übersetzer, der ERDDURCHMESSER schon vorweg ausrechnet das ganz genauso machen. Selbst dann, wenn er viel genauer Gleitkommazahlen berechnen kann.

In manchen Programmiersprachbeschreibungen (z.B. Java) werden genaue Bedingungen für das Berechnen von Ausdrücken festgelegt. Das hat für uns den Vorteil, dass wir zur Überprüfung der Korrektheit der Konstantenfaltung oft nur gucken müssen, ob die Bedingungen der Sprachbeschreibung eingehalten wurden. Eine Betrachtung der reinen Rechnerarithmetik kann damit oft für uns entfallen.

Insgesamt gilt also: Konstantenfaltung darf durchgeführt werden, wenn sich dadurch das Laufzeitverhalten des Programms nicht ändert.

1.2 Konstantenfaltung in Übersetzern

Konstantenfaltung kann prinzipiell in verschiedenen Phasen des Übersetzers gemacht werden. In der Regel wird man die Konstantenfaltung aber im Compiler-Backend als eine der ersten Optimierungen auf der Zwischensprache durchführen. Konstantenfaltung kann auch mehrfach durchgeführt werden. Im Gegensatz zu vielen anderen Optimierungen benötigt Konstantenfaltung keine spezielle Zwischensprache oder Vorarbeiten wie Datenflussanalyse. Wir müssen jedoch die Konstante in ihrer ursprünglichen Form bis in die Zwischensprache "retten". Dies geschieht durch eine eigene Konstantentabelle, in der wir die Konstante zum Beispiel als String abgespeichern.

Der überwiegende Teil konstanter Ausdrücke ist gar nicht im Programmtext selbst zu finden, sondern wird erst bei der Adressabbildung erzeugt[1].

[1] Es ist daher vorteilhaft, die Konstantenfaltung erst nach der Adressabbildung oder mehrfach durchzuführen.

Zum Beispiel wird bei einem Reihungszugriff

```
int a[10];

a[5] = a[6];
```

in manchen Programmiersprachen die Addresse von a[5] durch $Addr(a) + 5 * length(int)$ berechnet, wobei $Addr(a)$ die Adresse des Anfangs der Reihung a ist und damit oft zur Übersetzungszeit bekannt, $length(int)$ die Länge eines Integers in Byte und damit in der Regel konstant 4 ist. Bei Verbunden oder wenn Reihungen nicht mit Untergrenze 0 beginnen oder wenn sie mehr als eine Dimension besitzen können wir noch mehr "falten".

Man beachte, dass bei Adressberechnungen in der Regel nur die Operatoren +,- und * auftreten und dass Adressberechnungen in Ganzzahlarithmetik durchgeführt werden.

Weiteres zur Einbettung der Konstantenfaltung in den Übersetzer ist unter anderem in [1], [6], [8] und [12] zu finden.

Um korrekte Konstantenfaltung in der Praxis zu gewährleisten, unterscheiden wir drei Fälle

- Die Übersetzerarithmetik unterscheidet sich nicht von der der zu übersetzenden Sprache bzw. von der der Zielmaschine. Dann dürfen wir Konstantenfaltung natürlich durchführen.
- Die Übersetzerarithmetik simuliert die Arithmetik der Zielmaschine bzw. die durch den Sprachstandard des zu Übersetzenden Programms vorgeschriebene Arithmetik. Auch in diesem Fall darf Konstantenfaltung natürlich durchgeführt werden
- Übersetzerarithmetik und Zielmaschinen bzw. durch Sprachstandard vorgeschriebene Arithmetik sind verschieden, trotzdem kann sichergestellt werden, dass das Ergebnis das gleiche ist; egal ob der Ausdruck in Übersetzerarithmetik berechnet wird und dann in das Ergebnis in Zielmaschinenarithmetik konvertiert wird, oder ob gleich in Zielmaschinenarithmetik gerechnet wird. Hierbei können wir entweder allgemeingültige Regeln angeben, wann Ausdrücke in einer anderen Arithmetik berechnet werden dürfen. Wir können aber auch von Fall zu Fall unterscheiden und zum Beispiel beim Auftreten eines Überlaufs in Integerarithmetiken auf Konstantenfaltung verzichten.

In dieser Studienarbeit befassen wir uns mit dem letzten Fall erste Alternative: Übersetzer- und Zielmaschinenarithmetik sind verschieden. Trotzdem stellen wir allgemeine Kriterien vor, unter denen Konstantenfaltung in Übersetzerarithmetik durchgeführt werden darf. Diese Kriterien haben wir von Hand bewiesen. Für die in Programmiersprachen typischste Form der Ganzzahlarithmetik haben wir diese Kriterien maschinell mit dem Isabelle/HOL System verifiziert. Wir betrachten verschiedene Probleme, die bei sättigenden- und bei Fließkommaarithmetiken auftreten. Schließlich zeigen wir anhand von Fallstudien, wie Konstantenfaltung in verschiedenen Übersetzern durchgeführt wird.

1.3 Übersicht über die Arbeit

In Kapitel 2 führen wir algebraische Grundlagen ein, die für das weitere Verständnis unerlässlich sind. In Kapitel 3 sind Kriterien für die Austauschbarkeit von Arithmetiken zu finden. Mit Kapitel 4 zeigen wir wann Ganzzahlarithmetiken durch andere ersetzt werden dürfen. Wir geben ein notwendiges und ein hinreichendes Kriterium an. Kapitel 5 gibt einen Überblick über sättigende- und Fließkommaarithmetiken. In Kapitel 6 ist schließlich der in Isabelle/HOL formal geführte Beweis für die Austauschbarkeit bestimmter Integerarithmetiken zu finden. In Kapitel 7 betrachten wir die Ganzzahlarithmetiken von Java und C bezüglich ihrer Austauschbarkeit zur Übersetzungszeit. Wir haben in Kapitel 8 Fallstudien, bezüglich der Durchführung von Konstantenfaltung in verschiedenen Übersetzern, erstellt. In Kapitel 9 haben wir verwandte Arbeiten diskutiert und Kapitel 10 gibt eine Zusammenfassung und einen Ausblick.

Eine Vorversion dieser Arbeit ist bereits veröffentlicht[3].

Kapitel 2

Algebraische Grundlagen

In diesem Kapitel definieren wir Grundlagen aus der Algebra, wie sie zum Beispiel in [2] zu finden sind.

Definition 1 (Operationen) *Eine Abbildung $f : A^n \longrightarrow A$ mit $n \in \mathbb{N} \cup \{0\}$ und $A \neq \emptyset$ heißt n-stellige Operation auf A. n ist die Stelligkeit von f. Die Menge aller n-stelligen Operationen wird mit $Op_n(A)$ bezeichnet. Eine Operation mit Stelligkeit 0 auf A wird als Konstante bezeichnet.*

Definition 2 (Typ einer Algebra) *Der Typ einer Algebra ist eine Menge $\mathcal{F}$ von Funktionssymbolen. Jedem Element $f \in \mathcal{F}$ wird eine nicht negative ganze Zahl - die Stelligkeit - zugeordnet.*

Zu einer gegebenen Algebra A bezeichnen wir ihren Typ mit $Type(A)$.

Definition 3 (Algebra) *Ist $\mathcal{F}$ Typ eine Algebra, dann ist eine Algebra A vom Typ $\mathcal{F}$ ein Tupel (A, F), wobei $A \neq \emptyset$ und F eine Menge von Operationen auf A ist, die mit den Funktionssymbolen aus $\mathcal{F}$ derart bezeichnet werden, dass jedem n-stelligen Funktionssymbol $f \in \mathcal{F}$ eine n-stellige Operation $f^A \in F$ zugeordnet wird. Die Menge A wird auch als Universum bezeichnet.*

Ist $F = \{f_1, ..., f_k\}$ endlich, schreiben wir auch $(A, f_1, ...f_k)$ anstelle von (A, F). Wenn wir die Menge der Operationen einer Algebra $\mathcal{A}$ auf eine Bestimmte Menge M einschränken wollen schreiben wir $\mathcal{A}|_M$.

Klassische Beispiele von Algebren sind Gruppen und Ringe: Eine Gruppe ist eine Algebra $\mathcal{G} = (G, \cdot, ^{-1}, 1)$, in der folgende Gesetze gelten:

$x \cdot (y \cdot z) = (x \cdot y) \cdot z$ Assoziativität
$e \cdot x = x \cdot e = x$ Neutrales Element
$x \cdot x^{-1} = x^{-1} \cdot x = e$ Inverse Elemente

In einer abelschen Gruppe gilt auch noch das Kommutativgesetz:
$x \cdot y = y \cdot x$

In diesem Beispiel ist $Op_0(G) = \{1\}$,$Op_1(G) = \{^{-1}\}$ und $Op_2(G) = \{\cdot\}$. Funktionssymbole und Funktionen werden hier nicht weiter unterschieden. Das ist im allgemeinen unproblematisch. Wenn es wirklich wichtig ist können wir die Funktionen mit 1_G, $^{-1}{}_G$ bzw. $\cdot_G$ kennzeichnen. Ihnen sind hier die Funktionssymbole 1, $^{-1}$ bzw. $\cdot$ zugeordnet.

Ein Ring $(R, +, -, 0, \cdot)$ ist eine Algebra mit $+$ und $\cdot$ als binäre Operationen und $-$ als unäre Operation. $(R, +, -, 0)$ ist eine abelsche Gruppe. $\cdot$ ist assoziativ

und es gelten die Distributivgesetze:
$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
$(x + y) \cdot z = (x \cdot z) + (y \cdot z)$

Definition 4 (Termmenge) *Eine Termmenge $T(\mathcal{F}, V)$ zu einer Variablenmenge V und dem Typ einer Algebra $\mathcal{F}$ ist induktiv definiert: wenn f ein n-stelliges Funktionssymbol ist und $t_1, ..., t_n$ Terme sind, dann ist auch $f(t_1, ..., t_n)$ ein Term.*

In unserem Beispiel oben sind zum Beispiel "$1 \cdot 1$" oder " $1 + 1$ " Terme, da 1 ein Term ist und $\cdot$ bzw. $+$ zweistellige Funktionssymbole sind.

Ausdrücke können wir also auch als Terme bezeichnen.

Definition 5 (Termalgebra) *Eine Termalgebra $\mathcal{T}(V)$ vom Typ $\mathcal{F}$ mit Variablenmange V ist eine Algebra $\mathcal{T}(V) = (T(\mathcal{F}, V), \mathcal{F})$, so dass für jedes $f \in \mathcal{F}$ gilt: $f_{\mathcal{T}(V)}(t_1, ..., t_n) = f(t_1, ..., t_n)$.*

Definition 6 (Homomorphismus) *Eine Abbildung zwischen zwei Algebren $\mathcal{A}$, $\mathcal{B}$ mit gleichem Typ $\alpha : A \longrightarrow B$ heißt Homomorphismus von $\mathcal{A}$ nach $\mathcal{B}$, wenn $\alpha(f_{\mathcal{A}}(a_1, ..., a_n) = f_{\mathcal{B}}(\alpha(a_1), ..., \alpha(a_n))$ gilt.*

Der Ker von α, $\ker(\alpha)$, ist definiert durch $\ker(\alpha) = \{(a, a') \mid a, a' \in A \wedge \alpha(a) = \alpha(a')\}$.

Wenn $\alpha : A \longrightarrow B$ und $\beta : B \longrightarrow C$ Homomorphismen: von $\mathcal{A}$ nach $\mathcal{B}$ bzw. von $\mathcal{B}$ nach $\mathcal{C}$, dann ist die Verkettung von α und β: $\beta \circ \alpha$, mit $\beta \circ \alpha(a) = \beta(\alpha(a))$, $a \in A$ auch ein Homomorphismus.

Definition 7 (Kongruenzrelation) *Eine Kongruenzrelation θ auf einer Algebra $\mathcal{A} = (A, F)$ ist eine Äquivalenzrelation. Zusätzlich gilt für alle $f \in F$: Ist f eine n-Stellige Funktion und $(t_i, t_i') \in \theta$, $1 \leq i \leq n$ dann ist auch $(f(t_1, ..., t_n), f(t_1', ..., t_n')) \in \theta$.*

Wenn $\alpha : A \longrightarrow B$ ein Homomorphismus von $\mathcal{A}$ nach $\mathcal{B}$ ist, dann ist $\ker(\alpha)$ eine Kongruenzrelation auf A.

Definition 8 (Restklasse) *Eine Restklasse ist eine Teilmenge von $\mathbb{Z}$: $n\mathbb{Z} + r := \{x \in \mathbb{Z} | x \bmod n = r\}$, für beliebiges aber fest gewähltes n und $0 \leq r \leq n-1$. r ist der Standardrepräsentant einer Restklasse.*

Bei einem festen n bilden also die Elemente $0, 1, ..., n-1$ die Standardrepräsentanten für die Restklassen. Die Restklassen sind Kongruenzklassen folgender Kongruenzrelation über $\mathbb{Z}$: $\theta_n : x \theta_n y$ genau dann wenn $x \bmod n = y \bmod n$.

Bild 2.1 veranschaulicht die Beziehung zwischen Standardrepräsentanten einer Restklasse und Nichtstandardrepräsentanten. Mit den Spalten stellen wir jeweils eine Restklasse dar. Welches Element eine Restklasse letztendlich repräsentiert ist eigentlich egal.

Theorem 1 *Sei $\mathcal{A}$ eine Algebra,* Con $\mathcal{A}$ *die Menge aller Kongruenzrelationen auf $\mathcal{A}$. Im Kongruenzrelationenverband* **Con** $\mathcal{A}$ *von $\mathcal{A}$ mit Universum* Con $\mathcal{A}$ *sind $\wedge$ und $\vee$ folgendermaßen Berechenbar:*

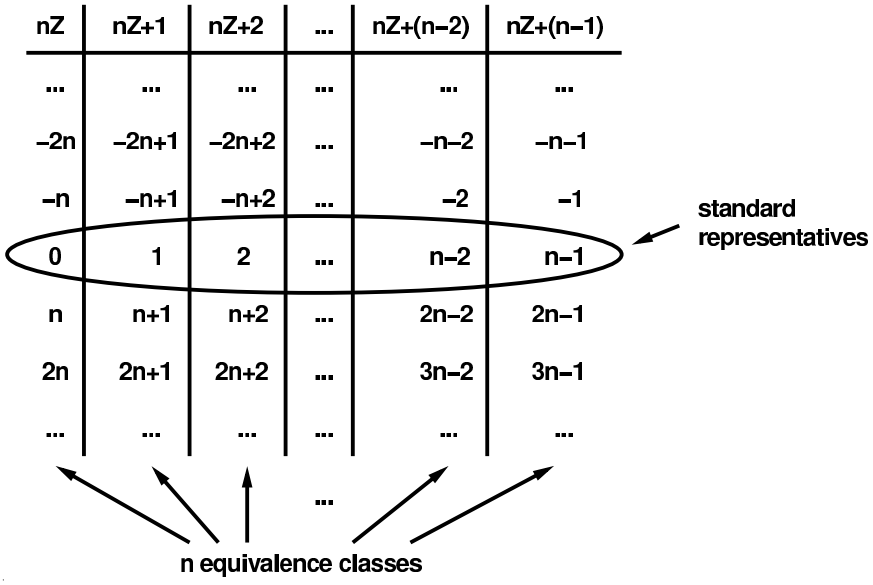


Abbildung 2.1: Restklassen von $\mathbb{Z}$

- $\theta_1 \wedge \theta_2 = \theta_1 \cap \theta_2$
- $\theta_1 \vee \theta_2 = \theta_1 \cup (\theta_1 \circ \theta_2) \cup (\theta_1 \circ \theta_2 \circ \theta_1) \cup (\theta_1 \circ \theta_2 \circ \theta_1 \circ \theta_2) \cup \cdots$, *das ist gleichbedeutend mit* $(a, b) \in \theta_1 \vee \theta_2$ *gdw. es Elemente* $a = c_1, c_2, \ldots, c_n = b$ *gibt, so dass* $(c_i, c_{i+1}) \in \theta_1$ *oder* $(c_i, c_{i+1}) \in \theta_2$ *für* $1 \leq i \leq n-1$.

Kapitel 3

Austauschbarkeit von Algebren

Dieses Kapitel zeigt allgemeine Bedingungen auf, unter denen Algebren ersetzt werden dürfen. Wir stellen Arithmetik als Beispiele für Algebren vor. Wir beweisen Kriterien unter denen wir eine Algebra durch eine andere ersetzen dürfen. Unter dem Ersetzen von Algebren verstehen wir, dass wir Ausdrücke, die in einer Algebra zur Berechnung anstehen in eine andere transformieren können und dort ausrechnen. Dann transformieren wir das Ergebnis wieder zurück.

3.1 Beispiele für Algebren: die Arithmetiken

Wir betrachten Rechnerarithmetiken als Spezialfälle von Algebren.

Definition 9 (Arithmetik) *Eine Arithmetik ist eine Algebra A mit Typ $\mathcal{F}$ wobei $+, -, \cdot, 0$ in $\mathcal{F}$ enthalten sind. $+, \cdot$ haben die Stelligkeit 2, $-$ die Stelligkeit 1 und die 0 die Stelligkeit 0 (Konstante).*

Diese Definition einer Arithmetik ist eine Minimaldefinition. Sie wird nur gebraucht, um die folgenden bekannten Rechnerarithmetiken formal zu fassen.

Als **gewöhnliche Ganzzahl-/Integerarithmetik** $(\mathbb{Z}, +, -, \cdot)$bezeichnen wir in dieser Studienarbeit die Arithmetik, die den Axiomen des Rechnens in den ganzen Zahlen genügt. Wenn nichts anderes erwähnt ist hat sie die Operatoren $(+, -, \cdot)$ mit ihren standard Bedeutungen.

Definition 10 (Ganzzahl- / Integerarithmetik) *Eine Ganzzahlarithmetik (A, F) ist eine Arithmetik dessen Universum eine Teilmenge der Ganzenzahlen ist. Falls A eine echte Teilmenge der Ganzenzahlen ist gibt es $MININT$, $MAXINT \in A$ mit $MININT \leq MAXINT$ und $\forall i \in Z\ MININT \leq i \leq MAXINT \Rightarrow i \in A$*

Eine Integerarithmetik hat als Universum eine zusammenhängende Teilmenge der Ganzenzahlen. Diese Teilmenge schreiben wir auch als $[MININT, MAXINT]$. Die Definition erfasst keine Arithmetiken, deren Universum entweder nur ein MININT oder nur ein MAXINT enthält[1]. Da diese Arithmetiken für uns unbedeutend sind können wir damit leben.

[1] also zum Beispiel $\mathbb{N}$

Ein weiteres Beispiel (neben der gewöhnlichen Ganzzahlarithmetik) für eine Arithmetik ist die Integerarithmetik, die Modulo 8 ($\mathbb{Z}_8$) rechnet:
$\mathcal{F} = \{+, \cdot, -, 0, 1\}$
$A = \{0, 1, 2, 3, 4, 5, 6, 7\}$
$F = \{+_8, \cdot_8, -_8, 0, 1\}$
(A, F) Ist eine Arithmetik, die alles modulo 8 rechnet. $a +_8 b$ entspricht der Operation $(a + b)$ *mod* 8 in der gewönlichen Integerarithmetik . Die Ziffern 0,1,2,3,4,5,6,7 sind Elemente des Universums A. 0 und 1 sind (eher wilkürlich gewählt) als Konstanten (Nullstellige Funktionen) in F vorhanden und damit auch als Nullstellige Funktionssymbole in F

Der Typ einer Arithmetiken wird oft neben dem unären $-$ auch ein binäres $-$ umfassen, dem die Operation $a - b = a + (-b)$ zugeordnet wird.

Definition 11 (Restklassen oder ModuloArithmetik) *Eine Restklassenarithmetik* $\mathbb{Z}_n = (\{0, 1, .., n-1\}, +, -, \cdot, +_{\mathbb{Z}_n}, -_{\mathbb{Z}_n}, \cdot_{\mathbb{Z}_n})$ *ist eine Ganzzahlarithmetik, wobei* $+_{\mathbb{Z}_n}, -_{\mathbb{Z}_n}, \cdot_{\mathbb{Z}_n}$ *folgendermaßen definiert sind:*
$a +_{\mathbb{Z}_n} b = a +_{\mathbb{Z}} b \ mod \ n$
$-_{\mathbb{Z}_n} a = -_{\mathbb{Z}} a \ mod \ n$
$a \cdot_{\mathbb{Z}_n} b = a \cdot_{\mathbb{Z}} b \ mod \ n$
Die $+_{\mathbb{Z}}, -_{\mathbb{Z}}, \cdot_{\mathbb{Z}}$ *sind dir entsprechenden Operationen in der gewöhnlichen Ganzzahlarithmetik.*

Restklassenarithmetiken sind die Arithmetiken, die Ausdrücke modulo einer Zahl n ausrechnen.

Ein Beispiel für die eine Restklassenarithmetik ist die Arithmetik, die modulo 8 rechnet, die wir oben vorgestellt haben (n = 8). Wir haben sie ja auch, nach dieser Definition korrekt, als $\mathbb{Z}_8$ bezeichnen.

Die gewöhnliche Ganzzahlarithmetik $\mathbb{Z}_\infty$ kürzen wir, wenn Missverständnisse ausgeschlossen sind auch mit $\mathbb{Z}$ ab.

3.2 Algebren und ihre Austauschbarkeit

Um zu einem Ersetzbarkeitsbegriff zu kommen, brauchen wir eine "Genauigkeitsrelation" auf Algebren.

Definition 12 (Genauigkeitsrelation auf Algebren: $\succeq$) *Eine Algebra* $\mathcal{A} = (A, F_A)$ *ist genauer* $(\mathcal{A} \succeq \mathcal{B})$ *als eine Algebra* $\mathcal{B} = (B, F_B)$ *genau dann wenn* $Type(\mathcal{A}) \subseteq Type(\mathcal{B})$ *und ein surjektiver Homomorphismus* $f : \mathcal{A}|_{Op(\mathcal{B})} \to \mathcal{B}$ *existiert.*

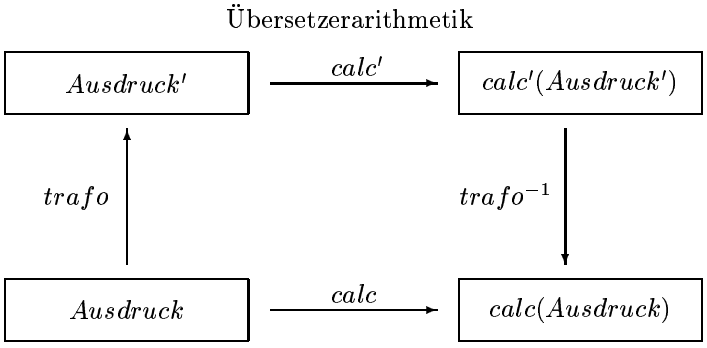
Die Genauigkeitsrelation auf Algebren benutzen wir um unsere oben definierten Arithmetiken Vergleichbar zu machen. Zum Beispiel ist die gewöhnliche Ganzzahlarithmetik genauer als die Arithmetik $\mathbb{Z}_8$: Es gilt nämlich $Type(\mathbb{Z}_8) = Type((\mathbb{Z}, +, -, \cdot)) = \{+, -, \cdot\}$. Der surjektive Homomorphismus $f : \mathbb{Z} \longrightarrow \{0, 1, 2, 3, 4, 5, 6, 7\}$ nimmt alles modulo 8:
$f(x) = x \mod 8$
$f(x + y) = x + y \mod 8$
$f(-x) = -x \mod 8$
$f(x \cdot y) = x \cdot y \mod 8$

Wobei wir hier jeweils die Operationen aus der gewöhnlichen Ganzzahlarithmetik genommen haben[2].

Um Konstantenfaltung zu verifizieren müssen wir auch noch definieren, was ein konstanter Ausdruck eigentlich ist. Wir definieren konstante Ausdrücke einer Algebra $\mathcal{A} = (A, F)$ als die Termalgebra $\mathcal{T}(\mathcal{A}) = (T(Type(\mathcal{A}), A), Type(\mathcal{A}))$ mit gleichen Typ wie $\mathcal{A}$ und der "Variablenmenge" A. Wir können jede Funktion $f : A \longrightarrow B$ in eine Funktion $f : \mathcal{T}(\mathcal{A}) \longrightarrow \mathcal{T}(\mathcal{B})$ überführen, indem wir jedes $a \in A$ auf $f(a)$ und jedem Term $g(t_1, ...t_n)$ auf $g(f(t_1), ..., f(t_n)$ abbilden.

Weiterhin brauchen wir eine Auswertungsfunktion: $eval_{\mathcal{A}} : \mathcal{T}(\mathcal{A}) \to \mathcal{A}$ sie weist jedem konstanten Ausdruck seinen Wert zu und ist daher induktiv folgendermaßen definiert:
$eval_{\mathcal{A}}(t) = a$, falls $t = a$ mit $a \in A$
$eval_{\mathcal{A}}(t) = h_{\mathcal{A}}(eval_{\mathcal{A}}(t_1), ..., eval_{\mathcal{A}}(t_n))$ falls $t = h(t_1, ..., t_n)$

In der folgenden Abbildung transformiert $trafo$ den (konstanten) *Ausdruck* in eine andere Algebra: die Übersetzerarithmetik, wo er mit $calc'$ berechnet wird. Zurücktransformiert in die Ursprungsalgebra: die Zielarithmetik soll das gleiche herauskommen, wie wenn der Ausdruck gleich in Zielmaschinenarithmetik berechnet worden wäre.



Theorem 2 (Ersetzbarkeit) *Für zwei Algebren $\mathcal{A} = (A, F_A)$ und $\mathcal{B} = (B, F_B)$ mit $\mathcal{A} \succeq \mathcal{B}$ mit Homomorphismus $f : \mathcal{A}|_{Op(\mathcal{B})} \to \mathcal{B}$ existiert eine Funktion $f^{-1} : T(Type(\mathcal{B}), B) \to T(Type(\mathcal{B}), A)$, so dass für alle $t \in T(Type(\mathcal{B}), B)$ folgendes gilt: $f(eval_{\mathcal{A}}(f^{-1}(t))) = eval_{\mathcal{B}}(t)$*

Beweis: Als erstes definieren wir uns eine Funktion $f^{-1} : \mathcal{B} \longrightarrow \mathcal{A}|_{Op(\mathcal{B})}$, die in eine Funktion $f^{-1} : T(Type(\mathcal{B}), B) \to T(Type(\mathcal{B}), A)$ überführt werden kann. Wir zeigen, dass dieses f^{-1} alle verlangten Eigenschaften hat.

Sei $\hat{f}^{-1} : \mathcal{B} \longrightarrow \mathcal{A}$ folgendermaßen definiert: $\hat{f}^{-1}(b) = \{a | f(a) = b\}$. Weil f surjektiv ist gilt $\hat{f}^{-1}1 \neq \emptyset$ für alle $b \in B$. Sei $f^{-1}(b) \in \hat{f}^{-1}(b)$, dann gilt, unabhängig davon wie $f^{-1}(b) \in \hat{f}^{-1}(b)$ gewählt wurde, immer $f(f^{-1}(b)) = b$

Wir zeigen jetzt mit vollständiger Induktion, dass für alle $t \in T(Type(\mathcal{B}), B)$ folgendes gilt: $f(eval_{\mathcal{A}}(f^{-1}(t))) = eval_{\mathcal{B}}(t)$

I.A. :Sei $t = b$, $b \in B$, $a' = f^{-1}(b)$. $f(eval_{\mathcal{A}}(f^{-1}(t))) = f(eval_{\mathcal{A}}(a') = f(a') =$

[2]Mit denen haben wir ja auch $\mathbb{Z}_8$ definiert

$f(f^{-1}(b)) = b = eval_{\mathcal{B}}(t)$

I.S. :

$f(eval_{\mathcal{A}}(f^{-1}(h(t_1, ..., t_n)))) =$	
$f(eval_{\mathcal{A}}(h(f^{-1}(t_1), ..., f^{-1}(t_n)))) =$	Überführen in Termalgebra
$= f(h(eval_{\mathcal{A}}(f^{-1}(t_1)), ..., eval_{\mathcal{A}}(f^{-1}(t_n))))$	Definition von $eval_{\mathcal{A}}$
$= h(f(eval_{\mathcal{A}}(f^{-1}(t_1))), ..., f(eval_{\mathcal{A}}(f^{-1}(t_n))))$	f ist ein Homomorphismus
$= h(eval_{\mathcal{B}}(t_1), ..., eval_{\mathcal{B}}(t_n))$	I.A.
$= eval_{\mathcal{B}}(h(t_1, ..., t_n))$	Definitions von $eval_{\mathcal{B}}$

Oben haben wir gezeigt, dass die Gewöhnliche Ganzzahlarithmetik genauer $\succeq$ als $\mathbb{Z}_8$ ist. Nach diesem Theorem dürfen wir nun Ausdrücke, die in $\mathbb{Z}_8$ zur Berechnung anstehen, mit f^{-1} [3] in die gewöhnliche Ganzzahlarithmetik transformieren. Dort können wir sie Ausrechnen und das Ergebnis mit f zurücktransformieren.

[3] was hier gerade die Identität ist

Kapitel 4

Verband der Restklassenarithmetiken

In diesem Kapitel beschreiben wir die Besonderheiten, die bei Restklassenarithmetiken auftauchen und wie wir sie in einem Verband anordnen können.

4.1 Anforderungen der Praxis

In der Praxis treten Integerarithmetiken in der Regel als Restklassenarithmetiken auf. Integerarithmetiken in real existierenden Programmiersprachen haben wir in Kapitel 7 betrachtet. Typischerweise gibt es zwei Varianten:

- Die unsigned - Variante: mit einem Universum $\{0, 1, ..., n-1\}$,wobei n eine Zweierpotenz ist. Da Register und Speicherstellen aus Bits aufgebaut sind, würde Platz verschenkt werden, wenn n keine Zweierpotenz wäre.
- Die signed - Variante: das Universum einer solchen Arithmetik umfasst die Elemente $\{-n/2, -n/2+1, ..., -1, 0, 1, ...n/2-2, n/2-1\}$. n ist auch hier typischerweise eine Zweierpotenz. Normalerweise werden signed - Zahlen im Zweierkomplement dargestellt, daher gibt es eine negative Zahl mehr als positive.

Wir betrachten sowohl die unsigned-Arithmetiken als auch signed- Arithmetiken als Restklassenarithmetiken. Für die unsigned Variante ist das nicht weiter Problematisch. Die Zahlen im Universum $\{0, ..., n-1\}$ sind ja gerade die Standardrepräsentanten ihrer Restklassen. Für die signed-Arithmetiken auch nicht! Die positiven Zahlen $\{0, ..., n/2-1\}$ sind Standardrepräsentanten ihrer jeweiligen Restklasse. Die negativen $\{-n/2, ..., -1\}$ zwar nicht, aber jede repräsentiert trotzdem eine andere Restklasse. $-n/2$ repräsentiert die Restklasse $\{..., -n/2, n/2, (3n/2), ...\}$ mit Standardrepräsentant $n/2$. $-n/2+1$ hat entsprechend den Standardrepräsentanten $n/2+1$. -1 ist in der Restklasse $\{..., -1, n-1, 2n-1, ...\}$ und hat damit den Standardrepräsentanten $n-1$.

4.2 Austauschen von Restklassenarithmetiken

Im nachfolgenden Theorem geben wir ein Kriterium an, unter dem Ausdrücke in einer Restklassenarithmetik auch in einer anderen Restklassenarithmetik ausgerechnet werden können:

Theorem 3 (Austauschen von Restklassenarithmetiken) *Wenn n m teilt mit $n, m \in \mathbb{N}^+$, dann ist die Restklassenarithmetik $\mathbb{Z}_m = (\{0, ..., m-1\}, +, -, \cdot)$ genauer als $\mathbb{Z}_n = (\{0, ..., n-1\}, +, -, \cdot)$: $\mathbb{Z}_m \succeq \mathbb{Z}_n$.*

Beweis: Zwei Bedingungen sind nach der Definition von $\succeq$ zu zeigen: $Type(\mathbb{Z}_n) \subseteq Type(\mathbb{Z}_m)$ ist trivialerweise war.
Um die zweite Bedingungzu verifizieren definieren wir eine Funktion:
$f : \mathbb{Z}_m \longrightarrow \mathbb{Z}_n$
und zeigen, dass dieses f ein surjektiver Homomorphismus ist. Da n Teiler von m ist, können wir $m = n \cdot_{\mathbb{Z}} p$ setzen. Jedes $x \in \{0, ..., m-1\}$ kann als $x = r +_{\mathbb{Z}} l \cdot_{\mathbb{Z}} n$ mit $l \in \{0, ..., p-1\}$ und $0 \leq r \leq n-1$ dargestellt werden. Sei $f(x) = r$. f ist surjektiv, da $x \in \{0, ..., n-1\}$. Auch gilt damit $f(x) = x$.

Damit f ein Homomorphismus ist müssen wir noch folgende Eigenschaften nachweisen:

1. $f(0_{\mathbb{Z}_m}) = 0_{\mathbb{Z}_n}$
2. $f(x +_{\mathbb{Z}_m} y) = f(x) +_{\mathbb{Z}_n} f(y)$
3. $f(-_{\mathbb{Z}_m} x) = -_{\mathbb{Z}_n} f(x)$
4. $f(x \cdot_{\mathbb{Z}_m} y) = f(x) \cdot_{\mathbb{Z}_n} f(y)$

Seien $x = r +_{\mathbb{Z}} l \cdot_{\mathbb{Z}} n$, $y = r' +_{\mathbb{Z}} l' \cdot_{\mathbb{Z}} n$ mit $0 \leq r \leq n-1$, $0 \leq r' \leq n-1$ und $l, l' \in \{0, ..., p-1\}$.

1. Ist trivialerweise war.

2. $f(x +_{\mathbb{Z}_m} y) = f(x) +_{\mathbb{Z}_n} f(y)$**:**
$f(x +_{\mathbb{Z}_m} y) = f((r +_{\mathbb{Z}} l \cdot_{\mathbb{Z}} n) +_{\mathbb{Z}_m} (r' +_{\mathbb{Z}} l' \cdot_{\mathbb{Z}} n))$
$= f((r +_{\mathbb{Z}_m} l \cdot_{\mathbb{Z}_m} n) +_{\mathbb{Z}_m} (r' +_{\mathbb{Z}_m} l' \cdot_{\mathbb{Z}_m} n))$
$= f(r +_{\mathbb{Z}_m} r' +_{\mathbb{Z}_m} l \cdot_{\mathbb{Z}_m} n +_{\mathbb{Z}_m} l' \cdot_{\mathbb{Z}_m} n)$
$= \begin{cases} r +_{\mathbb{Z}_m} r' & r +_{\mathbb{Z}_m} r' \leq n-1 \\ r +_{\mathbb{Z}_m} r' -_{\mathbb{Z}_m} n & sonst \end{cases}$
$r +_{\mathbb{Z}_n} r' = f(x) +_{\mathbb{Z}_n} f(y)$

3. $f(-_{\mathbb{Z}_m} x) = -_{\mathbb{Z}_n} f(x)$
$f(-_{\mathbb{Z}_m} x) = f(m -_{\mathbb{Z}} x) = f(m -_{\mathbb{Z}} (r +_{\mathbb{Z}} l \cdot_{\mathbb{Z}} n))$
$= f(m -_{\mathbb{Z}} (l \cdot_{\mathbb{Z}} n) -_{\mathbb{Z}} r) = f(m -_{\mathbb{Z}} r)$
$f(p \cdot_{\mathbb{Z}} n -_{\mathbb{Z}} r) = f((p -_{\mathbb{Z}} 1) \cdot_{\mathbb{Z}} n +_{\mathbb{Z}} n -_{\mathbb{Z}} r)$
$= n -_{\mathbb{Z}} r = -_{\mathbb{Z}_n} r = -_{\mathbb{Z}_n} f(x)$

4. $f(x \cdot_{\mathbb{Z}_m} y) = f(x) \cdot_{\mathbb{Z}_n} f(y)$
Einerseits gilt $f(x) \cdot_{\mathbb{Z}_n} f(y) = r \cdot_{\mathbb{Z}_n} r' = (r \cdot_{\mathbb{Z}} r') \ mod \ n$
Auf der anderen Seite haben wir: $f(x \cdot_{\mathbb{Z}_m} y) = f((r +_{\mathbb{Z}} l \cdot_{\mathbb{Z}} n) \cdot_{\mathbb{Z}_m} (r' +_{\mathbb{Z}} l' \cdot_{\mathbb{Z}} n)) = f(r \cdot_{\mathbb{Z}_m} r')$

Sei nun $r \cdot_{\mathbb{Z}_m} r' = r''$,$r \cdot_{\mathbb{Z}} r' = l'' \cdot_{\mathbb{Z}} m +_{\mathbb{Z}} r''$,$0 \leq r'' \leq m-1$ und $(r \cdot_{\mathbb{Z}} r') mod\ n = r'''$, $r \cdot_{\mathbb{Z}} r' = l''' \cdot_{\mathbb{Z}} n +_{\mathbb{Z}} r'''$, $0 \leq r''' \leq n-1$.
Da n m teilt gibt es ein q mit $q \cdot_{\mathbb{Z}} l'' = l'''$ und $f(r'') = r''\ mod\ n = r'''$.

Diesen Beweis haben wir auch maschinell mit Isabelle/HOL geführt.

Theorem 4 *$\mathbb{Z}$ ist eine genauere Arithmetik als jedes $\mathbb{Z}_n$ mit $n \in \mathbb{N}^+$ ($\mathbb{Z} \succeq \mathbb{Z}_n$).*

Der Beweis ist trivial, wir haben ihn aber auch mit Isabelle/HOL geführt.

Theorem 5 (Verband der Restklassen-Arithmetiken) *Die Restklassenarithmetiken $\mathbb{Z}_n$und $\mathbb{Z}$ mit $n \in \mathbb{N}^+$ bilden einen Verband bezüglich der $\succeq$ Ordnung.*

Beweis: Es müssen lediglich die Verbandseigenschaften nachgewiesen werden.
inf und sup sind folgendermaßen definiert.

- $inf(\mathbb{Z}_n, \mathbb{Z}) = \mathbb{Z}_n$
- $sup(\mathbb{Z}_n, \mathbb{Z}) = \mathbb{Z}$
- $inf(\mathbb{Z}_n, \mathbb{Z}_m) = \mathbb{Z}_{ggT(n,m)}$
- $sup(\mathbb{Z}_n, \mathbb{Z}_m) = \mathbb{Z}_{kgV(n,m)}$

kgV ist das kleinste gemeinsame Vielfache, ggT der größte gemeinsame Teiler. Dass $\mathbb{Z}_{ggT(n,m)}$ gerade die Infimum Eigenschaft erfüllt beziehungsweise dass $\mathbb{Z}_{kgV(n,m)}$ die Supremumseigenschaft bezüglich $\succeq$ besitzt folt direkt aus den vorangegangenen Theoremen.

Top Element dieses Verbandes ist $\mathbb{Z}$, das Bottom Element ist $\mathbb{Z}_1$.

Theorem 6 *Der Verband der Ganzzahlarithmetiken(Int_Arith) ist isomorph zum dualen Verband* **Con** $\mathbb{Z}$ *der Kongruenzrelationen von* $\mathbb{Z}$.

Sei $\theta_n = \{(x, y) \mid x \bmod n = y \bmod n\}$. Wir definieren die Funktionen
$f : Int\_Arith \longrightarrow \text{Con } \mathbb{Z}$, $f(\mathbb{Z}_n) = \theta_n$ und $f(\mathbb{Z}) = \{(z, z) \mid z \in \mathbb{Z}\} =: \theta_\infty$
sowie
$f^{-1} : \text{Con } \mathbb{Z} \longrightarrow Int\_Arith$ with $f^{-1}(\theta_n) = \mathbb{Z}_n$,$f^{-1}(\theta_\infty) = \mathbb{Z}$.
f sind f^{-1} trivialerweise surjektive Funktionen und f^{-1} ist die inverse Funktion zu f. Wir müssen zeigen, dass f ein Isomorphismus ist, d.h., $\mathbb{Z}_m \succeq \mathbb{Z}_n$ gdw. $\theta_m \subseteq \theta_n$.
"⇒": Wir nehmen an: $\mathbb{Z}_m \succeq \mathbb{Z}_n$ und zeigen: $\theta_m \subseteq \theta_n$:
Angenommen $\mathbb{Z}_m \succeq \mathbb{Z}_n$. Dann existiert ein surjektiver Homomorphismus: $f : \mathbb{Z}_m \longrightarrow \mathbb{Z}_n$. Weiterhin existieren die surjektiven Homomorphismen $g : \mathbb{Z} \longrightarrow \mathbb{Z}_m$ and $h : \mathbb{Z} \longrightarrow \mathbb{Z}_n$ weil $\mathbb{Z} \succeq \mathbb{Z}_n$, $\mathbb{Z} \succeq \mathbb{Z}_m$. $h = f \circ g$ gilt. Auch existieren $\ker(g) = \theta_m$ and $\ker(h) = \theta_n$. Assume that $(x, y) \in \theta_m = \ker(g)$, aber $(x, y) \notin \theta_n = \ker(h)$. Dies ist ein Widerspruch, da alle Elemente die auf das gleiche Element durch g abgebildet werden, müssen auch auf das gleiche Element durch $f \circ g = h$ abgebildet werden. Also gilt: $\theta_m \subseteq \theta_n$.
"⇐": Wir nehmen an: $\theta_m \subseteq \theta_n$ und zeigen $\mathbb{Z}_m \succeq \mathbb{Z}_n$:
Angenommen $\theta_m \subseteq \theta_n$. Um zu zeigen, dass $\mathbb{Z}_m \succeq \mathbb{Z}_n$ gilt, beweisen wir, dass n m teilt. Wir nehmen das Gegenteil an: n teilt m nicht und führen einen Widerspruch herbei. $(m, 0) \in \theta_m$ aber $(m, 0) \notin \theta_n$, da n m nicht teilt. Dies ist ein Widerspruch zur Voraussetzung: $\theta_m \subseteq \theta_n$. Also schließen wir, dass n m teilt und damit auch $\mathbb{Z}_m \succeq \mathbb{Z}_n$ gilt.

Daraus folgt, dass die Teilereigenschaft nicht nur ein hinreichendes sondern auch ein notwendiges Kriterium, für $\succeq$ auf Restklassenarithmetiken, ist: m, $n \in \mathbb{N}^+$. n teilt m genau dann wenn $\mathbb{Z}_m \succeq \mathbb{Z}_n$.

Kapitel 5

Andere Integer- und Gleitkommaarithmetiken

In diesem Kapitel diskutieren wir Klassen von Arithmetiken für die es im allgemeinen nicht möglich ist, eine eine Relation "kann berechnet werden in" aufzustellen : Die sättigenden Integerarithmetiken und die Floatingpointarithmetiken.

5.1 Sättigende Arithmetiken

Sättigende Arithmetiken sind auch Ganzzahlarithmetiken, die $MAXINT$ und $MININT$ als größte bzw. kleinste Zahl im Universum enthalten. Im Gegensatz zu den Arithmetiken, die Modulo einer Zahl Rechnen wird ein Ausdruck, dessen Ergebnis größer als $MAXINT$ zu $MAXINT$ "abgerundet". Analog wird bei Unterschreitung von $MININT$ aufgerundet.[1]

Definition 13 *Eine Sättigende Ganzzahlarithmetik (A, F) ist eine Ganzzahlarithmetik mit endlichem Universum, mit vollgenden Eigenschaften: Die Operationen sollen im folgenden mit $+_s, -_s, \cdot_s$ für die sättigende Arithmetik, mit $+_z, -_z, \cdot_z$ für die gewöhnliche Ganzzahlarithmetik bezeichnet werden.*
$+_s, -_s, \cdot_s$ sind folgendermaßen definiert:

$$a +_s b = \begin{cases} MAXINT & a +_z b \geq MAXINT \\ MININT & a +_z b \leq MININT \\ a \cdot_z b & sonst \end{cases}$$

[1] Sättigende Arithmetiken kommen zum Beispiel im TriMedia Prozessor [11] von Philips zum Einsatz. Sie werden dort auch als "clipped" bezeichnet. Es gibt dort unter *anderem* folgende Befehle, die in Sättigender Arithmetik rechnen (auf 32 Bit Operanden): dspiabs: Berechnet den Betrag eines signed Werts (ja auch hier gibt es einen! Wert, wo es einen Unterschied macht), dspiadd : signed Add, dspuadd: unsigned Add, dspimul: signed Mul, dspumul : unsigned Mul, dspisub : signed Sub, dspusub : unsigned Sub.

Sättigende Arithmetiken eignen sich für gewisse Multimediatransformationen. Zum Beispiel können hohe Werte sehr selten auftreten. Für den Menschen sind sie vielleicht gar nicht mehr Wahrnehmbar, wie bei hohen Tönen. Dann ist es sinnvoll diese "Ausreißer" auch auf hohe Werte abzubilden und nicht modulo irgendetwas zu nehmen und damit wohlmöglich auf ganz niedrige Werte abzubilden.

$$a \cdot_s b = \begin{cases} MAXINT & a \cdot_z b \geq MAXINT \\ MININT & a \cdot_z b \leq MININT \\ a \cdot_z b & sonst \end{cases}$$

$$-_s b = \begin{cases} MAXINT & -_z b \geq MAXINT \\ MININT & -_z b \leq MININT \\ a -_z b & sonst \end{cases}$$

Sei $A = [-4, 3]$ Universum einer sättigenden Arithmetik (A, F_A), also $MININT = -4$ und $MAXINT = 3$. Dann gilt zum Beispiel folgendes

$$-4 + (-1)) + 1 = -4 + 1 = -3 \neq -4 = -4 + ((-1) + 1)$$

[nicht assoziativ]

Sei $B = [-8, 3]$ Universum einer anderen sättigenden Arithmetik (B, F_B). Es ist nicht möglich, Berechnungen aus (A, F_A) in (B, F_B) durchzuführen und das Ergebnis dann zurückzukonvertieren, wie folgendes Beispiel zeigt:

$$\text{Berechnung in } (A, F_A) : (2 + 2) - 3 = 3 - 3 = 0$$
$$\text{Berechnung in } (B, F_B) : (2 + 2) - 3 = 4 - 3 = 1$$

Die beiden Beispielarithmetiken stehen daher nicht in einer Relation "kann berechnet werden mit" .[2]

5.2 Die Fließkommazahlarithmetiken

Konstantenfaltung bei Fließkommazahlen ist im allgemeinen sehr viel schwieriger als bei den Ganzzahl Arithmetiken. Über den Fließkommazahlarithmetiken läßt sich leider kein Verband bezüglich einer Relation "Arithmetik A kann berechnet werden mit Arithmetik B" aufbauen. Im folgenden soll gezeigt werden, warum dies nicht möglich ist. Weiterhin ist es auch viel schwieriger eine spezielle Fließkommaarithmetik zu simulieren, als dies bei Ganzzahlarithmetiken der Fall wäre.

5.3 Der IEEE Floating - Point Standard

Fließkommazahlen werden im IEEE 754 Floating Point Standard[4] in folgender Form dargestellt: $s \cdot m \cdot 2^{e-N+1}$, wobei s das Vorzeichen, entweder -1 oder +1 annimmt, m die Mantisse eine positive ganze Zahl kleiner als 2^N und e eine ganze Zahl zwischen $E_{min} = -(2^{K-1} - 2)$ und $E_{max} = 2^{K-1} - 1$ ist.

Die Programmiersprache Java schreibt in ihrem Standard [5] folgende Werte von N, K, E_{min}, E_{max} vor.

[2] Natürlich ist das kein Beweis, dass es nicht doch zwei, sättigende Arithmetiken gibt, die in der Relation "kann berechnet werden mit" stehen. Ausser den trivialen Fällen, wo sich die Arithmetiken nicht oder nur in ihrem Typ unterscheiden, haben wir keine in dieser Relation stehenden sättigenden Arithmetiken gefunden.

Parameter	float	double
N	24	53
K	8	11
E_{max}	+127	+1023
E_{min}	-126	-1022

Neben Zahlen schreibt der Standard auch noch vor, dass bestimmte Werte: NaN (Not a Number), +infinity und -infinity darstellbar sein müssen.

Folgendes Java - Beispiel[3] soll zeigen, dass man Fließkommazahl - Berechnungen, selbst die Addition, nicht einfach in einer genaueren Arithmetik durchführen kann und das Ergebnis dann zurückkonvertieren kann:

```
double d;
float f;

  f = 0;
f += 2.0E38;
f += 2.0E38;
f -= 1.0E38;
System.out.println(f);

d = 0;
d += 2.0E38;
d += 2.0E38;
d -= 1.0E38;
System.out.println((float) d);
```

Sowohl in der float - Arithmetik als auch in der genaueren double - Arithmetik werden die gleichen Berechnungen durchgeführt. Die Zahl 4.0E38 (Ergebnis der zweiten Addition) läßt sich in float aber nicht mehr darstellen, deshalb wird hier der Wert Infinity angenommen, der auch durch eine Subtraktion nicht mehr verlassen wird. Das Ergebnis ist Infinity.

In der double - Arithmetik läßt sich die Zahl 4.0E38 sehr wohl noch darstellen. Das Endergebnis 3.0E38 läßt sich wieder in float darstellen. Zurückkonvertiert ergibt sich ein anderes Ergebnis, als wenn die ganze Berechnung in float-Arithmetik durchgeführt worden wäre. Das Ausgabe ist:

```
Infinity
3.0E38
```

Wenn sich aber nicht einmal bezüglich der Addition eine Relation "kann berechnet werden in" aufstellen läßt, geht das für die Subtraktion und die Multiplikation trivialerweise auch nicht.

Man könnte nun meinen, dass wenigstens Aussagen in der Form über die Genauigkeit einer Arithmetik möglich sind. Zum Beispiel könnten wir annehmen, dass Berechnungen auf dem java Datentyp double immer genauer sind als auf float. Folgendes Beispiel-Programm scheint dies zu bestätigen:

```
double d;
float f;
```

[3]Dieses und das nächste Beispiel sind in abgewandelter Form nach [5] erstellt worden

```
f = 1.0f / 41.0f;
f = f * 41.0f;
System.out.println(f);

d = 1.0d / 41.0d;
d = d * 41.0d;
System.out.println(d);
```

Die Ausgabe ist:

```
0.99999994
1.0
```

Hier tritt offensichtlich ein Rundungsfehler in der float Berechnung auf, der in der double Arithmetik nicht auftritt. Es gibt aber auch Rundungsfehler in der double Arithmetik, die in der Float-Arithmetik nicht zum Tragen kommen. Folgendes ganz ähnliche Programm belegt dies:

```
double d;
float f;

f = 1.0f / 49.0f;
f = f * 49.0f;
System.out.println(f);

d = 1.0d / 49.0d;
d = d * 49.0d;
System.out.println(d);
```

Die Ausgabe ist:

```
1.0
0.9999999999999999
```

Zwar ergibt 0,9999999999999999 nach float konvertiert den Wert 1. Dies kann jedoch kein Argument sein double Berechnungen von Ausdrücken als prinzipiell genauer anzusehen. Natürlich lassen sich Ausdrücke konstruieren, wo eine Vielzahl von solchen Rundungsfehlern zu weitaus größeren Fehlern führt.

Kapitel 6

Die Isabelle-Formalisierung

In diesem Kapitel haben wir mit dem Isabelle/HOL System die Austauschbarkeit von Restklassenintegerarithmetiken bewiesen. Isabelle/HOL wurde und wird an der TU-München und der Cambridge University entwickelt. Eine Einführung in Isabelle gibt [10].

6.1 Allgemeines zu Isabelle

Isabelle/HOL setzt auf der funktionalen Programmiersprache ML auf. Dementsprechend werden auch in Isabelle Beweise über Funktionen und die für funktionale Programmiersprachen typischen Datenstrukturen: Listen und hier insbesondere Bäumen geführt.

Ein typisches Beweisdokument in Isabelle wird durch das Schlüsselwort "theory" eingeleitet und setzt sich aus Datentypdefinitionen, Funktions- bzw. Konstantendefinitionen und Lemmata zusammen. Ein Lemma können wir mithilfe verschiedener nacheinander angewandter Beweisschritte beweisen. Zu den Beweistechniken, die wir in so einem Schritt angewenden können zählen: die Anwendung anderer Lemmata, die Fallunterscheidung und die Induktion. Die Induktion können wir insbesondere auch auf Datentypen wie Listen und Bäume anwenden.

Eine Vielzahl von Regeln sind in Isabelle selbst schon bewiesen bzw. axiomatisiert. Für uns sind davon insbesondere von Interesse (a sei vom Typ int)

- a = m * (a div m) + a mod m
- a mod 0 = a

6.2 Modellierung als Ausdrucksbaum

Um zu zeigen, dass Bedingungen für beliebige Ausdrücke gelten brauchen wir zunächst einmal eine geeignete Datenstruktur. Mit Hilfe von Kantorowitsch Bäumen lassen sich zum Beispiel gewöhnliche arithmetische Ausdrücke darstellen. Auch zur Modellierung der Ganzzahlarithmetikausdrücke ist es möglich auf eine Baumstruktur zurückzugreifen. Baumstrukturen haben zudem den Vorteil, dass sie sich in Isabelle sehr einfach modellieren lassen:

$$datatype\ Etree = Leaf\ int \mid Node\ operator\ Etree\ Etree \tag{6.1}$$

definiert den Datentyp Etree. So ein Etree ist entweder ein Blatt (Leaf), das einen Integer Wert enthält, oder ein Knoten (Node), der einen Operator und zwei weitere Bäume (Etree) enthält.

Als Operatoren sollen in diesem Beispiel nur +,- und * zur Verfügung stehen.

$$datatype\ operator = Add \mid Sub \mid Mult \tag{6.2}$$

Beispielsweise sieht der Ausdruck $(5 + 6) * 7$ als Ausdrucksbaum in Isabelle folgendermaßen aus:

$$Node\ (\ Node\ (\ Leaf\ 5)\ Add\ (\ Leaf\ 6\ )\ )\ Mult\ (\ Leaf\ 7\ ) \tag{6.3}$$

Neben dem Ausdrucksbaum ist natürlich noch mindestens eine Funktion die dessen Ergebnis berechnet nötig. Für Berechnungen, die den Regeln des Rechnens im Ring der ganzen Zahlen genügt definieren wir die Funktion calc[1].

$$\begin{array}{l} consts \\ calc\ ::\ \text{“}Etree \Rightarrow int\text{“} \\ primrec \\ \text{“}calc\ (Leaf\ a) = a\text{“} \\ \text{“}calc\ (Node\ ox\ a\ b) = (case\ ox\ of \\ Add \Rightarrow (calc\ a) + (calc\ b) \mid \\ Sub \Rightarrow (calc\ a) - (calc\ b) \mid \\ Mult \Rightarrow (calc\ a) * (calc\ b))\text{''} \end{array} \tag{6.4}$$

Weiterhin benötigen wir für die folgenden Ausführungen die Funktion calcm, sie berechnet den Wert eines Ausdrucksbaum, indem sie jeden Operanden modulo m nimmt und alle Operationen modulo m ausführt.

$$\begin{array}{l} consts \\ calcm\ ::\ \text{“}Etree \Rightarrow int \Rightarrow int\text{''} \\ primrec \\ \text{“}calcm\ (Leaf\ a)\ m = a\ mod\ m\text{''} \\ \text{“}calcm\ (Node\ ox\ a\ b)\ m = (case\ ox\ of \\ Add \Rightarrow (((calcm\ a\ m) + (calcm\ b\ m))mod\ m) \mid \\ Sub \Rightarrow (((calcm\ a\ m) - (calcm\ b\ m))mod\ m) \mid \\ Mult \Rightarrow (((calcm\ a\ m) * (calcm\ b\ m))mod\ m))\text{''} \end{array} \tag{6.5}$$

Weiterhin brauchen wir auch noch eine eigene Funktion, die sich wie calc m verhält, aber statt modulo m - modulo (m*n) rechnet.

[1] Die calc - Varianten, die wir hier vorstellen entsprechen den eval - Funktionen aus den ersten Kapiteln

$$
\begin{array}{l}
consts \\
calcmn\ ::\ \text{“}Etree \Rightarrow int \Rightarrow int \Rightarrow int\text{''} \\
primrec \\
\text{“}calcmn\ (Leaf\ a)\ m\ n = a\ mod\ (m * n)\text{''} \\
\text{“}calcmn\ (Node\ ox\ a\ b)\ m\ n = (case\ ox\ of \\
Add \Rightarrow ((calcmn\ a\ m\ n) + (calcmn\ b\ m\ n))mod\ (m * n) | \\
Sub \Rightarrow ((calcmn\ a\ m\ n) - (calcmn\ b\ m\ n))mod\ (m * n) | \\
Mult \Rightarrow ((calcmn\ a\ m\ n) * (calcmn\ b\ m\ n))mod\ (m * n))\text{''}
\end{array}
\tag{6.6}
$$

6.3 Beweis der hinreichenden Bedingung

In diesem Abschnitt führen wir folgende Beweise:

1. Einmal wollen wir zeigen, dass wir alle Ausdrucksbäume, die wir mit calcm berechnen, auch mit calc berechnen können. Wir müssen das Ergebnis von calc nur nocheinmal modulo m nehmen, damit immer die gleichen Ergebnisse herauskommen.

2. Dann wollen wir zeigen, dass alle Ausdrucksbäume, die mit calcm berechnet werden, auch mit calcmn berechnet werden können, allerdings unter der Voraussetzung, dass n größer als 0 ist[2]. Das Ergebnis von calcmn modulo m genommen -auf den selben Ausdrucksbaum angewandt- ist das Gleiche, wie das von calcm.

Im ersten Beweis behandeln wir alle Ausdrücke, die nur $(+, -, *)$ enthalten und in einer Arithmetik, die alles modulo m rechnet (also einer Restklassenarithmetik) zur Berechnung anstehen. Wir zeigen, dass sie auch in der gewöhnlichen Ganzzahlarithmetik berechnet werden können, indem man das Ergebnis dort modulo m nimmt.

Mit dem zweiten Beweis zeigen wir, dass hierzu auch eine Arithmetik genügt, die Modulo m*n rechnet, wenn n größer 0 ist.

“calc” modulo m entspricht “calcm”

In Isabelle wird die Forderung, dass alle Ausdrucksbäume berechnet mit calcm das gleiche Ergebnis haben, wie der selbe Ausdrucksbaum berechnet mit calc und das Ergebnis modulo m genommen, folgendermaßen formalisiert:

$$
lemma\ \text{“}(calc\ a)mod\ m = (calcm\ a\ (m :: int))\text{''} \tag{6.7}
$$

Der Erste Beweisschritt ist eine Induktion über den Ausdrucksbaum a:

[2] Das ist keine Einschränkung. Bei Arithmetiken werden m und n immer größer 0 sein

$$
\begin{aligned}
&1.\bigwedge int.\ calc\ (Leaf\ int) mod\ m = calcm\ (Leaf\ int)\\
&2.\bigwedge operator\ Etree1\ Etree2.\\
&[|calc\ Etree1\ mod\ m = calcm\ Etree1\ m;\\
&calc\ Etree2\ mod\ m = calcm\ Etree2\ m|]\\
&calc\ (Node\ operator\ Etree1\ Etree2) mod\ m =\\
&calcm\ (Node\ operator\ Etree1\ Etree2)\ m
\end{aligned}
\tag{6.8}
$$

Die Induktionsverankerung (1.) folgt direkt aus der Definition von calc bzw. calcm. Für den induktionsschritt (2.) werden die Definitionen von calc und calcm eingesetzt:

$$
\begin{aligned}
&\bigwedge operator\ Etree1\ Etree2.\\
&[|calc\ Etree1\ mod\ m = calcm\ Etree1\ m;\\
&calc\ Etree2\ mod\ m = calcm\ Etree2|]\\
&\Longrightarrow (case\ operator\ of\ Add \Rightarrow calc\ Etree1 + calc\ Etree\\
&|\ Sub \Rightarrow calc\ Etree1 - calc\ Etree\\
&|\ Mult \Rightarrow calc\ Etree1 * calc\ Etree2) mod\\
&m =\\
&(case\ operator\ of\ Add \Rightarrow (calcm\ Etree1\ m + calcm\ Etree2\ m) mod\ m\\
&|\ Sub \Rightarrow (calcm\ Etree1\ m - calcm\ Etree2\ m) mod\ m\\
&|\ Mult \Rightarrow calcm\ Etree1\ m * calcm\ Etree2\ m\ mod\ m)
\end{aligned}
\tag{6.9}
$$

Nun folgt eine Fallunterscheidung nach den Operatoren: Add, Sub und Mult. Exemplarisch sei dies für den Add Operator vorgeführt:

$$
\begin{aligned}
&\bigwedge Etree1\ Etree2.\\
&[|calc\ Etree1\ mod\ m = calcm\ Etree1\ m;\\
&calc\ Etree2\ mod\ m = calcm\ Etree2\ m|]\\
&\Longrightarrow (calc\ Etree1 + calc\ Etree2) mod\ m =\\
&(calcm\ Etree1\ m + calcm\ Etree2\ m) mod\ m
\end{aligned}
\tag{6.10}
$$

Das folgende Lemma hilft diesen Beweisschritt umzuformen:

$$
lemma\ ``(a + b) mod\ m = (a\ mod\ m + b\ mod\ m) mod\ (m :: int)''
\tag{6.11}
$$

Das Ergebnis sieht dann folgendermaßen aus und kann von Isabelle automatisch gelöst werden:

$$
\begin{aligned}
&\bigwedge Etree1\ Etree2.\\
&[|calc\ Etree1\ mod\ m = calcm\ Etree1\ m;\\
&calc\ Etree2\ mod\ m = calcm\ Etree2\ m|]\\
&\Longrightarrow (calc\ Etree1\ mod\ m + calc\ Etree2\ mod\ m) mod\ m =\\
&(calcm\ Etree1\ m + calcm\ Etree2\ m) mod\ m
\end{aligned}
\tag{6.12}
$$

Nachdem wir die drei Fälle (Add, Sub und Mult) bewiesen haben, ist der gesamte Beweis erbracht.

6.3.1 "calcmn" modulo m entspricht "calcm"

Die Forderung, dass alle Ausdrucksbäume berechnet mit calc das gleiche Ergebnis haben, wie der selbe Ausdrucksbaum berechnet mit calcmn und das Ergebnis modulo m genommen, wird folgendermaßen formalisiert:

$$lemma\ "(1 \leq n) \Longrightarrow (calcmn\ a\ m\ n) mod\ m = (calcm\ a\ (m :: int))" \quad (6.13)$$

Der Beweis geht Analog zu dem oben vorgeführten und ist wie auch alle anderen Beweise und Formalisierungen im Anhang zu finden.

6.3.2 Ein Verband entsteht

Im folgenden wollen wir die Annahme treffen: m ist größer gleich 0 und n immer größer 0. Oben wurde bewiesen, dass ein beliebiger Ausdrucksbaum, der in calcm berechnet wird, genausogut in calc berechnet werden kann. Dazu ist es allerdings notwendig, dass das Ergebnis nocheinmal modulo m genommen wird. Erst dadurch erhält man dann in jedem Fall den gleichen Wert. Wir können also sagen, dass calcm auch mit calc berechnet werden kann, umgekehrt gilt das trivialerweise nicht. Die Funktion calc genügt den Regeln der gewöhnlichen Ganzzahlaritmetik. Die Funktion calcm der der Restklassenarithmetik, die modulo m rechnet. Es kommt also schon einmal ein flacher Halbverband bezüglich der Relation "kann berechnet werden mit" -mit der gewöhnlichen Ganzzahlaritmetik als Top Element- zustande.

Jetzt benutzen wir die Tatsache, dass calcm auch durch calcmn bei gleichem m berechnet werden kann. Der Halbverband ist damit nicht mehr flach. Um einen Verband zu bekommen fehlt noch das Bottom-Element: calcm mit m = 1 bildet alle Elemente auf die 0 ab und die zugehörige Arithmetik erfüllt damit diese Eigenschaften.

Der gebildete Verband wurde ausschließlich aus den in Isabelle formalisierten Elementen gebildet!

Es stellt sich die Frage, warum wir nur die Operatoren +,- und * betrachtet haben, die Antwort ist einfach: für div und mod existiert ein solcher Verband nicht.

Folgendes Lemma ist falsch, auf der linken Seite wurde die gewöhnliche Ganzzahlaritmetik angewandt, auf der rechten die Ganzzahlaritmetik modulo 8: (linke Seite: 2, rechte Seite: 1)

$$lemma\ "(((4+5) mod\ 7) mod\ 8 = (((4+5) mod\ 8) mod\ 7) mod\ (8 :: int))" \quad (6.14)$$

Auch dieses Lemma ist falsch, links gewöhnliche Ganzzahlaritmetik, rechts modulo 8 Aritmetik: (linke Seite: 4, rechte Seite: 0)

$$lemma\ "(((4+5) div\ 2) mod\ 8 = (((4+5) mod\ 8) div\ 2) mod\ (8 :: int))" \quad (6.15)$$

6.4 Beweis der notwendigen Bedingung

Wir haben gezeigt, dass Ausdrücke aus einer Arithmetik, die Modulo m rechnet auch in einer Arithmetik berechnet werden können, die Modulo n rechnet. Dazu muß m n teilen und das Ergebnis Modulo m genommen werden. Das m Teiler von n ist, ist jedoch -wie in Kapitel 4 gezeigt- auch eine notwendige Bedingung um beliebige Ausdrücke dieser Art in einer anderen Modulo n Arithmeitk durchzuführen.

Folgendes Lemma zeigt das auch in Isabelle:

$$\begin{aligned} lemma\ \text{“}((a = (Leaf\ n)) \wedge ((n\ mod\ m \neq 0) \wedge ((calcm\ a\ m) = \\ (calcm\ a\ n)\ mod\ m))) = False'' \end{aligned} \tag{6.16}$$

Das m kein Teiler von n ist, haben wir durch $n\ mod\ m \neq 0$ formalisiert. Die Bedingung für das Ausrechnen beliebiger Ausdrücke ist der Ausdruck $((calcm\ a\ m) = (calcm\ a\ n)\ mod\ m)$. Das (generische) Gegenbeispiel ist $a = (Leaf\ n)$. Es ist Sofort zu sehen, dass der Ausdruck $((calcm\ a\ m) = (calcm\ a\ n)\ mod\ m)$ zu $((calcm\ (Leaf\ n)\ m) = (calcm\ (Leaf\ n)\ n)\ mod\ m) = 0\ mod\ m = 0$ ausgewertet wird, was der Bedingung widerspricht, dass m kein Teiler von n sein darf. Isabelle wertet das Lemma in einem Schritt aus.

Kapitel 7

Integerarithmetiken in Java und C

In diesem Kapitel übertragen wir die theoretischen Ergebnisse aus den vorangegangenen Kapiteln auf konkrete durch Sprachstandards [5] [7] definierte Restklassenarithmetiken.

Wärend der Java-Sprachstandard genau vorschreibt, wie groß bzw. genau ein Typ zu sein hat und wie gerechnet werden muß läßt der C-Standard einiges offen. Es werden allerdings Beziehungen zwischen den Typen definiert, wie zum Beispiel "short int ist kleiner als int". Auf den meisten Systemen können wir die Dateien "limits.h" für Integertypen und "float.h" für Floatingpointtypen finden. Hier sind einige Angaben über die primitiven Datentypen einer C - Implementierung auf der aktuellen Plattform zu finden (z.B. #define SHRT_MAX 32767).

Die strikte Definition der Datentypen und ihrer Operationen durch Java ist durch die Sprachphilosophie bedingt. Java-Programme -einmal in Java-Bytecode übersetzt- sollen auf einer Vielzahl unterschiedlicher Systeme laufen können. Hierzu benötigen sie nur einen Java Interpreter. Auf jedem dieser Systeme soll ein gleiches Laufzeitverhalten[1] gewährleistet werden können.

C hingegen ist sehr viel älter als Java[2]. Ein Programm wird für jeden Maschinentyp neu übersetzt. Hier wird versucht ein Maximum der Vorteile jeder Maschine zu nutzen. Daher sind Datentypen unterschiedlich genau, wenn eine Maschine zum Beispiel breitere Register und entsprechende Operationen besitzt als eine andere.

7.1 Betrachtete Arithmetiken

Wir überprüfen, welche der folgenden in der folgenden Tabelle angegebenen Arithmetiken durch eine andere ersetzt werden kann. Dabei beschränken wir uns auf beliebige Ausdrücke, die allerdings nur die Operatoren $(+, -, \cdot)$ enthalten. Die Tabelle gibt verschiedene Datentypen jeweils mit MININT und MAXINT an.

[1] wenn wir von Ressourcenbeschränkungen und ähnlichem absehen
[2] C: Anfang der 70er Jahre, Java: Anfang der 90er Jahre

Typ— Sprache	MININT	MAXINT
Java		
byte	-128	127
char	u 0000 = 0	u ffff = 65535
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
C (gcc auf IA32)		
char	-128	127
signed char	-128	127
unsigned char	0	255
short	-32768	32767
unsigned short	0	65535
int	-2147483648	2147483647
unsigned int	0	4294967295
long	-2147483648	2147483647
unsigned long	0	4294967295
Fortran (Gnu Fortran)		
INTEGER (KIND = 1)	-2147483648	2147483647
INTEGER (KIND = 2)	-9223372036854775808	9223372036854775807
INTEGER (KIND = 3)	-128	127

7.2 Arithmetikverbände in der Praxis

Auch die von realen Programmiersprachen vorgeschriebenen Arithmetiken für Ganzzahlberechnungen bilden einen Verband bezüglich "Ausdrücke in Arithmetik A können berechnet werden in Arithmetik B" (Abbildung 7.1). Wir müssen dazu die gewöhnliche Ganzzahlarithmetik als maximales Element und eine Arithmetik, die alles auf die 0 abbildet, als minimales Element nehmen. Voraussetzung ist allerdings auch hier, dass die Ausdrücke nur die Operatoren $+, -, *$ enthalten.

Die negativen Zahlen, die in signed Datentypen auftauchen können wir algebraisch als Nichtstandardrepräsentanten entsprechender Restklassen betrachten. Demnach können signed und entsprechende unsigned Datentypen algebraisch gleich behandelt werden. Es existiert jedoch keine Untertypsbeziehung zwischen der signed und der unsigned Variante eines Ganzzahltyps gibt. Dies spielt in der Praxis aber eine Rolle. Daher sind in der obigen Abbildung zusätzlich zu dem algebraischen $\succeq$ auch noch ein Enthaltensein aller möglichen Werte einer Arithmetik voraussetzt. Wir packen wir hier signed und entsprechende unsigned Werte daher in unterschiedliche Äquivalenzklassen.

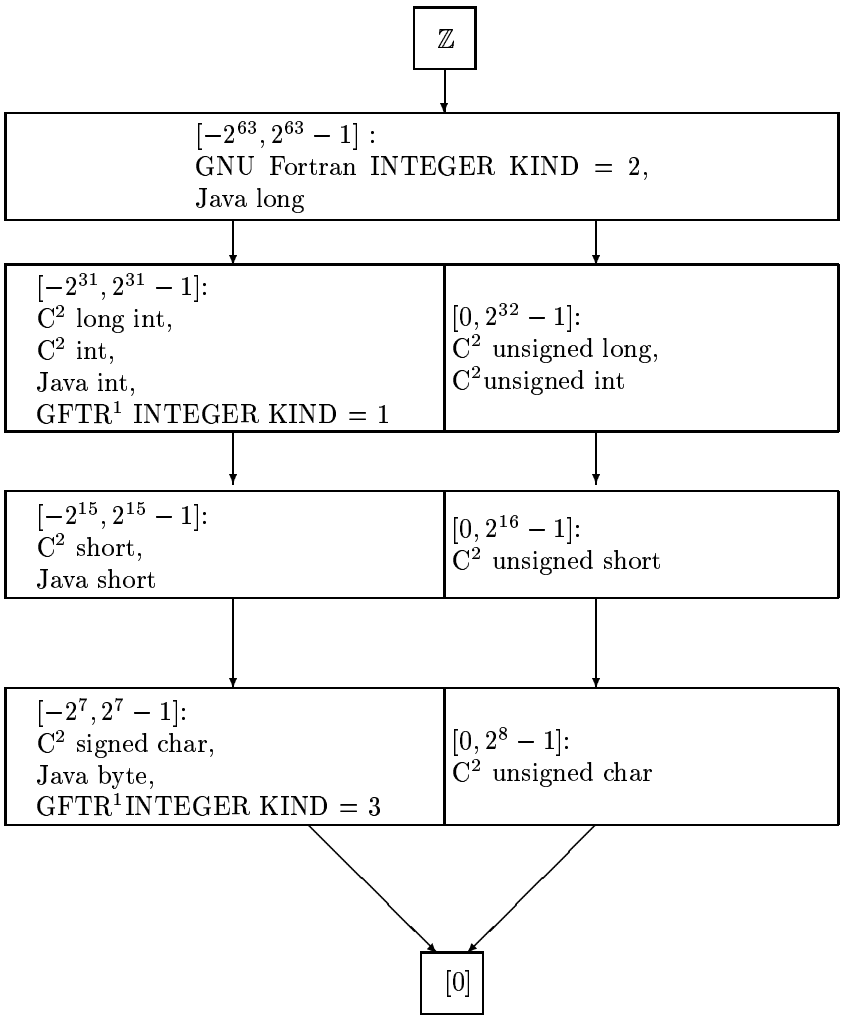


Abbildung 7.1: Verband der Integerarithmetiken

7.3 Anforderungen an C und Java in Übersetzern

Wir verlangen von einem Java Übersetzer, dass er, auf jeder Plattform Ausdrücke mit gleicher Genauigkeit behandelt, nämlich gerade die, die der Java - Sprachstandard vorgibt.

Ein C Übersetzer hingegen, muß die Genauigkeit der Zielmaschiene, für die er Übersetzt kennen und verwenden, um Konstantenfaltung durchführen zu können. Die erwähnten Dateien limits.h und float.h können von Plattform

[1] GNU Fortran
[2] gcc auf Intel 32 Bit Maschinen

zu Plattform verschieden sein. Ein C Cross-Compiler sollte daher die entsprechenden Dateien der Zielmaschine haben[3]

[3]zum Beispiel /usr/local/ia64-linux/include/ bzw. /usr/local/i686-pc-linux-gnu/include/ beim gcc als Cross-Compiler auf einem Linux-System

Kapitel 8

Fallstudien

In dieser Fallstudie untersuchen wir das Verhalten von Cross-Compilern für C, bezüglich Konstantenfaltung bei Integerwerten. Cross-Compiler sind Übersetzer, die für ein anderes System Übersetzen, als das, auf dem sie laufen.

Wir stellen in diesem Kapitel vor: den gcc Cross-Compiler xgcc[1], der auf einem Linux System auf einem 32 Bit-Rechner Code für ein 64 Bit Itanium System erzeugen soll. Weiterhin wird der lcc[2] betrachtet, der ebenfalls Code auf einer 32 Bit Maschine für eine 64 Bit Maschine erzeugen soll.

8.1 Testfall 1

Erster Testfall ist folgendes C Programm:

```
extern f(long int i);

int main(int argc,char **argv) {
  long int i1;
  long int i2;

  i1 = 2147483648; // = 2^31
  i2 = 2147483647 + 1; // = (2^31 - 1) + 1 = 2^31
  i3 = 2147483648 + 1; // = 2^31 + 1


  f(i1);
  f(i2);
  f(i3);
}
```

int und long int sind auf 32-Bit Systemen 32 Bit breit sind, decken also einen Zahlenbereich von $-2147483648 = -2^{31}$ bis $2147483647 = 2^{31} - 1$ ab. Wegen der Zweierkomplementdarstellung gibt es eine negative Zahl mehr, als positive Zahlen vorhanden sind. Auf 64 Bit Systemen ist int auch 32 Bit

[1] Version 3.2, http://gcc.gnu.org/
[2] Version 4.2, http://www.cs.princeton.edu/software/lcc/

breit, long int jedoch 64 Bit breit. long int deckt damit einen Zahlenbereich von $-9223372036854775808 = -2^{63}$ bis $9223372036854775807 = 2^{63} - 1$ ab.

Für die 32 Bit Integerarithmetik ist 2147483648 um eins zu groß. Ein Übersetzer, der für 32 Bit Systeme übersetzt sollte daher eine Warnung ausgeben und 2147483648 auf den Restklassenrepräsentanten -2147483648 abbilden. Auch 2147483647 + 1 sollte auf 32 Bit Maschinen zu -2147483648 ausgewertet werden. 2147483648 + 1 ist um zwei zu groß für die 32 Bit Arithmetik, sollte daher zu dem Restklassenrepräsentanten -2147483647 ausgewertet werden.

Auf 64 Bit Maschinen sollte 2147483648 nicht verändert werden.
Die 2147483647 + 1 sollten nicht wie man denken könnte zu 2147483648 ausgewertet werden, sondern -da 2147483647 und 1 beide noch int Werte sind (ohne long)- sollten sie in C auch in int-Arithmetik ausgewertet werden, nämlich zu -2147483648[3].

In 2147483648 + 1 ist einer der Operanden vom Typ long (2147483648). Daher sollte die Berechnung in long Arithmetik 2147483649 als Ergebnis liefern.

Die Funktion f ist nur vorhanden, um dem Übersetzer vorzugaukeln, dass i1, i2 und i3 noch benutzt werden. Wäre sie nicht vorhanden dürfte der Compiler das ganze Programm "wegoptimieren".

8.2 Testfall 2

Dieser Testfall dient dazu herauszufinden, ob Konstantenfaltung auf normalen int Werten durchgeführt wird. Kein Überlauf tritt bei der Berechnung 4009+702, die in jedem Fall 4711 als Ergebnis haben sollte. Die Zweite Berechnung findet mit int Werten im int-Bereich statt, wobei ein Überlauf auftritt.

```
extern g(int i);

int main(int argc,char **argv) {

  int i1;
  int i2;

  i1 = 4009+702;
  i2 = 2147483613+119;

  g(i1);
  g(i2);
}
```

8.3 Ergebnisse und Gegenüberstellung

Der gcc für 32 Bit Systeme und der xgcc von 32 auf 64 Bit Systeme führen die Konstantenfaltung im ersten Testfall korrekt durch (Siehe Assemblerlisting Anhang B).

[3]Das Schreibt der C Sprachstandard vor und ist in [7] zu finden

Der lcc für führt -im ersten Testfall- nur die Konstantenfaltung in der Berechnung 2147483648 + 1 durch, wenn er für ein 64 Bit System übersetzt (Alpha/OSF). 2147483647 + 1 bleibt unausgerechnet.

Der lcc verzichtet bei Auftreten eines Überlaufs von vornherein auf Konstantenfaltung, egal für welche Zielplattform er übersetzt. Sonst wird Konstantenfaltung durchgeführt, wie der zweite Testfall zeigt:

Übersetzt für ein Intel 32 Bit System wird in der ersten Berechnung der Ausdruch 4009+702 korrekt zu 4711 ausgewertet. In der Berechnung 2147483613+119 würde ein Überlauf auftreten und auf Konstantenfaltung wird verzichtet. Das entsprechende Assemblerlisting ist im Anhang B zu finden. Übersetzt für 64 Bit Systeme ändert sich nichts. int-Arithmetik sollte ja auch auf 32 wie auf 64 Bit Rechnern das gleiche liefern.

Der gcc bzw. xgcc führt auch im zweiten Testfall die Konstantenfaltung durch.

Intressant ist beim lcc, dass er auch die Fähigkeit hat die Zielarithmetik zu simulieren. Wird der erste Testfall mit dem lcc für Alpha/OSF[4] übersetzt wird die Konstantenfaltung bei dem Ausdruck der einen long Operanden hat (2147483648 + 1) korrekt durchgeführt. Bei (2147483647 + 1) treten zwei int Operanden auf, daher führt der lcc hier keine Konstantenfaltung durch.

Folgende Tabelle vergleicht Eigenschaften von lcc und gcc:

Übersetzer	lcc i32 nach alpha/osf	gcc i32 nach ia64
Konstantenfaltung (kein Überlauf)	√	√
Konstantenfaltung bei Überlauf	-	√
Simuliert 64 Bit Arithmetik	√	√

[4]int ist 32 Bit breit, long int 64 Bit breit

Kapitel 9

Verwandte Arbeiten

Die meisten Übersetzerbaubücher behandeln Konstantenfaltung auf nicht mehr als ein bis zwei Seiten. Als Korrektheitskriterium wird angegeben, dass der Übersetzer die Zielarithmetik simulieren muß oder sich zumindest genauso verhalten muß.

Die erste Arbeit, die die Korrektheit eines Übersetzers nachweist ist [9]. Hier wird bewiesen, dass ein Übersetzer für einfache arithmetische Ausdrücke korrekt arbeitet. Dieser Übersetzer setzt voraus, dass Quell- und Zielarithmetik identisch sind. Sie sollen zum Beispiel den Regeln des Rechnens in den reellen Zahlen genügen. Weiterhin ist der einzige Operator der Quellsprache der Additionsoperator. Der Beweis kann sich somit auf das korrekte Alloziieren von Speicherplatz von Zwischenergebnissen, auf die richtigen Operanden bei den Additionen und auf die richtige Reihenfolge der Operationen beschränken.

Eine Vorarbeit zu dieser Studienarbeit ist [3].

Kapitel 10

Zusammenfassung und Ausblick

In dieser Studienarbeit haben wir eine Bedingung erarbeitet unter der verschiedene Restklassenarithmetiken durch genauere ersetzt werden können. Ausdrücke brauchen nicht notwendigerweise in der Zielarithmetik ausgerechnet werden, sondern können oft auch in einer anderen berechnet werden. Dies ist vor allendingen bei den Cross-Compilern wichtig.

Den Beweis für die Vertauschbarkeit von Restklassenarithmetiken haben wir auf zwei Arten erbracht. Einmal konventionell per Hand. Dann maschinell mit dem Isabelle/HOL System. Man könnte argumentieren, dass ein Beweis ausreichen würde. Wir sind jedoch der Meinung, man sich bei einem Beweis der per Hand durchgeführt wurde leichter verschreibt oder Randbedingungen übersieht. Wir denken, dass ein maschineller Beweis zu einem von Hand geführten Beweis eine zusätzliche Absicherung ist. Die Chance, dass einem kein Fehler unterlaufen ist, steigt. Umgekehrt ist es oft Sinnvoll einen Beweis zuerst von Hand zu führen, da dies in der Regel schneller geht als einen Beweis maschinell zu führen. Die Bezeichnung automatisches Beweissystem ist in dieser Hinsicht irreführend, da das Beweissystem keine Arbeit abnimmt, sondern nur überwacht.

Es ist uns gelungen eine Austauschbarkeitsrelation bezüglich der Operatoren +,-,* aufzubauen. Wir haben gezeigt, dass es nicht möglich ist Operatoren wie div und mod miteinzubeziehen. Für Floatingpointarithmetiken und die sättigenden Integerarithmetiken konnten wir auch keine Austauschbarkeitsrelation aufstellen. In der Praxis besteht der ganz überwiegende Teil der Konstantenfaltung aus Adressberechnungen. Adressberechnungen werden aber gerade mit Restklassenarithmetiken und den Operatoren +,-,* durchgeführt. Damit deckt diese Studienarbeit den größten, praxisrelevanten Teil der Konstantenfaltung ab.

In zukünftigen Arbeiten könnte man noch weitere Arithmetiken untersuchen.

Interessant wären auch ausführlichere Fallstudien. Software für eingebettete Systeme -mit zum Beispiel 16 Bit Ganzzahlarithmetik- wird auf 32 Bit oder 64 Bit Systemen entwickelt. Gerade für diesen Fall ist bei den Operatoren +,-,* keine Simulation der 16 Bit Arithmetik nötig. Es wäre interessant zu sehen, wie groß der Anteil dieser Berechnungen, in einem Praxisrelevanten Projekt, ist. Vielleicht könnte man in einem einfachen Compiler den Großteil der Konstan-

tenfaltung durchführen ohne ein Simulationsmodul für 16 Bit Integerarithmetik zu entwickeln.

Literaturverzeichnis

[1] Andrew W. Appel: Modern Compiler Implementation in Java, Cambridge University Press 2002

[2] Stanley Burries, H.P. Sankappanavae: A Course in Universal Algebra

[3] Sabine Glesner, Jan Olaf Blech: Classifying and Formally Verifyin g Integer Constant Folding, Proceedings COCV-Workshop 2003. Electronic Notes in Theoretical Computer Science (ENTCS) Vol. 82 No 2, 2003

[4] IEEE Standard for Binary Floating-Point Arithmetic, IEEE 1985

[5] Java Language Specification (Second Edition), http://java.sun.com /docs/books/jls/second_edition/html/

[6] Uwe Kastens: Übersetzerbau, Oldenbourg 1990

[7] Kernighan, Ritchie: The C Programming Language, Prentice Hall 1988

[8] S. Muchnick: Advanced compiler design and implementation, Morgan Kaufmann 1997

[9] John McCarthy, James Painter: Correctness of a Compiler for Arithmetic Expressions, Proceedings of Symposia in Applied Mathematics Vol 19, 1967

[10] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel: Isabelle/HOL A Proof Assistent for Higher-Order Logic, LNCS 2283, Springer 2002

[11] TriMedia - Databook: TM - 1300 Media Processor, Philips Semiconductors 2000

[12] W.M. Waite, G. Goos: Compiler Cntruction, Springer 1984

Anhang A

Beweisskripte

Im folgenden sind die Beweisdateien aufgelistet.

A.1 Etree.thy

```
theory newmod = Main:

(* Basic stuff*)

lemma l1:"(a + b) mod m = (a mod m + b mod m) mod (m::int)"
apply (rule zmod_zadd1_eq)
done

lemma l2a:"(a::int) - b = a + -b"
apply (auto)
done

lemma l2b: "m - b mod (m::int) = m + (- (b mod m))"
apply (auto)
done

lemma l2c: "(m + - (b mod m)) mod m = (m mod m + - (b mod m) mod
m) mod (m::int)"
apply (rule zmod_zadd1_eq)
done

lemma l2d: "(a mod m - b mod m) mod m = (a mod m + (- (b mod m)))
mod (m::int) "
apply (auto)
done

lemma l2e: "(a mod m + (- (b mod m)) mod m) mod (m::int) = (a mod m
+ (- (b mod m))) mod (m::int) "
apply (subst zmod_zadd1_eq)
apply (auto)
```

```
done

lemma l2f: "((a::int) mod m - b mod m) mod m = (a - b) mod m"
apply (subst l2a)
apply (subst zmod_zadd1_eq)
apply (subst zmod_zminus1_eq_if)
apply (case_tac "b mod m = 0")
apply (auto)
apply (subst zmod_zadd1_eq)
apply (subst l2b)
apply (subst zmod_zadd1_eq)
apply (subst l2c)
apply (auto)
apply (subst l2e)
apply (auto)
done

lemma l2: "(a - b) mod m = ((a::int) mod m - b mod m) mod m "
apply (subst l2f)
apply (auto)
done

lemma l3a:"a mod m * (b mod m) mod (m::int) = a * b mod m"
apply (subst zmod_zmult1_eq)
apply (subst zmult_commute)
apply (subst zmod_zmult1_eq)
apply (subst zmult_commute)
apply (auto)
done

lemma l3:"(a * b) mod m = ((a mod m) * (b mod m)) mod (m::int)"
apply(subst l3a)
apply (auto)
done

lemma l4a: "(m * (a div m mod n) + a mod m) mod m = (m * (a div m
mod n) mod m + a mod m mod m) mod (m::int)"
apply (simp add: zmod_zadd1_eq)
done

lemma l4b: "m * (a div m mod n) mod m = (0::int)"
apply (auto)
done

lemma l4: "(1 ≤ n) ⟹ (a::int) mod m =a mod (m * n) mod m "
apply (subst zmod_zmult2_eq)
apply (auto)
apply (subst l4a)
apply (subst l4b)
apply (auto)
```

```
done
(*—-*)
(*Our Expression Tree*)
datatype operator = Add | Sub | Mult
datatype Etree = Leaf int | Node operator Etree Etree

(* Ordinary Integer Arithmetics*)
consts
calc :: "Etree ⇒ int"
primrec
"calc (Leaf a) = a"
"calc (Node ox a b) = (case ox of
Add ⇒ (calc a) + (calc b) |
Sub ⇒ (calc a) - (calc b) |
Mult ⇒ (calc a) * (calc b) )"

(* Ordinary Integer Aritnmetics mod m*)
consts
calcm :: "Etree ⇒ int ⇒ int"
primrec
"calcm (Leaf a) m = a mod m"
"calcm (Node ox a b) m = (case ox of
Add ⇒ (((calcm a m) + (calcm b m)) mod m) |
Sub ⇒ (((calcm a m) - (calcm b m)) mod m) |
Mult ⇒ (((calcm a m) * (calcm b m)) mod m) )"

lemma m1: "(calc a + calc b) mod m = (calc a mod m+ calc b mod m) mod
m"
apply (rule zmod_zadd1_eq)
done

lemma m1p: "(calc Etree1 - calc Etree2) mod m = (calc Etree1 mod m- calc
Etree2 mod m) mod m"
apply (rule l2)
done

lemma m1pp: "calc Etree1 * calc Etree2 mod m = (calc Etree1 mod m) *
(calc Etree2 mod m) mod m "
apply (rule l3)
done

(* You can either calculate an Expression Tree using normal integer arithmetics
taking the result mod m, but you can also use mod m - integer arithmetics,
giving the same result *)

lemma "(calc a) mod m = (calcm a (m::int))"
apply (induct_tac a)
apply (auto)
apply (case_tac "operator = Add")
apply (auto)
```

```
apply (subst m1)
apply (auto)
apply (case_tac "operator = Sub")
apply (auto)
apply (subst m1p)
apply (auto)
apply (case_tac "operator = Mult")
apply (auto)
apply (subst m1pp)
apply (auto)
apply (case_tac operator)
apply (auto)
done

consts
calcmn :: "Etree ⇒ int ⇒ int ⇒ int"
primrec
"calcmn (Leaf a) m n= a mod (m*n)"
"calcmn (Node ox a b) m n = ( case ox of
Add ⇒ ((calcmn a m n) + (calcmn b m n)) mod (m*n) |
Sub ⇒ ((calcmn a m n) - (calcmn b m n)) mod (m*n) |
Mult ⇒ ((calcmn a m n) * (calcmn b m n)) mod (m*n)
)
"

lemma m2: "(1 ≤ n) ⟹ (calcmn Etree1 m n + calcmn Etree2 m n) mod
(m * n) mod m = (calcmn Etree1 m n + calcmn Etree2 m n) mod m"
apply (subst l4)
apply (auto)
done

lemma m3: "(calcmn Etree1 m n + calcmn Etree2 m n) mod m = (calcmn
Etree1 m n mod m + calcmn Etree2 m n mod m) mod m"
apply (rule zmod_zadd1_eq)
done

lemma m2p: "(1 ≤ n) ⟹ (calcmn Etree1 m n - calcmn Etree2 m n) mod
(m * n) mod m = (calcmn Etree1 m n - calcmn Etree2 m n) mod m"
apply (subst l4)
apply (auto)
done

lemma m3p: "(calcmn Vtree1 m n - calcmn Vtree2 m n) mod m = (calcmn
Vtree1 m n mod m - calcmn Vtree2 m n mod m) mod m"
apply (rule l2)
done

lemma m2pp: "(1 ≤ n) ⟹ (calcmn Etree1 m n * calcmn Etree2 m n) mod
(m * n) mod m = (calcmn Etree1 m n * calcmn Etree2 m n) mod m"
apply (subst l4)
```

```
apply (auto)
done

lemma m3ppa: " ((calcmn Etree1 m n mod m) * (calcmn Etree2 m n mod
m)) mod m = (calcmn Etree1 m n * calcmn Etree2 m n) mod m "
apply (subst l3)
apply (auto)
done


lemma m3pp: "(calcmn Etree1 m n * calcmn Etree2 m n) mod m = ((cal-
cmn Etree1 m n mod m) * (calcmn Etree2 m n mod m)) mod m"
apply (subst m3ppa)
apply (auto)
done

(* You can either calculate an Expression Tree using integer arithmetics mod
m*n taking the result mod m, but you can also use mod m - integer arithmetics,
giving the same result *)

lemma "(1 ≤ n) ⟹ (calcmn a m n) mod m = (calcm a (m::int))"
apply (induct_tac a)
apply (auto)
apply (subst l4)
apply (auto)
apply (case_tac "operator = Add")
apply (auto)
apply (subst m2)
apply (auto)
apply (subst m3)
apply (auto)
apply (case_tac "operator = Sub")
apply (auto)
apply (subst m2p)
apply (auto)
apply (subst m3p)
apply (auto)
apply (case_tac "operator = Mult")
apply (auto)
apply (subst m2pp)
apply (auto)
apply (subst m3pp)
apply (auto)
apply (case_tac operator)
apply (auto)

done


(*——*)
```

lemma t2l4:"((b = (Leaf n)) ∧ ((n mod m ≠ 0) ∧ ((calcm b m) = (calcm b n) mod m)) = False) ⟹((n mod m ≠ 0) ∧ ((∀ a.(calcm a m) = (calcm a n) mod m)=False))"
apply (auto)
oops

(*If m does not divide n with remainder 0 we can't calculate every expression tree from "mod m"-Arithmetics using "mod n" -Arithmetics taking the result mod m. (Leaf n) for example will allways fail*)
lemma "((a = (Leaf n)) ∧ ((n mod m ≠ 0) ∧ ((calcm a m) = (calcm a n) mod m))) = False"
apply (auto)
done

end

A.2 counterex.thy

theory counterex = Main:

(* Consider an algebra that calculates expressions taking everything modulo 8
In general you can't calculate an expression -using the mod operator- of such an algebra in another algebra -e.g. the ordinary integer arithmetic- and take the result mod 8*)
lemma "¬ (((4 + 5) mod 7) mod 8 = (((4 + 5) mod 8) mod 7)mod (8::int))"
apply (auto)
done

(* Same with div*)
lemma "¬ (((4 + 5) div 2) mod 8 = (((4 + 5) mod 8) div 2) mod (8::int))"
apply (auto)
done
(*These examples show that (div,mod)-operations doesn't fit into the lattice of integer arithmetics that is defined only on operators +,- and *. *)

end

Anhang B

Assemblerlistings

B.1 gcc auf Intel 32 bit

```
        .file   "testcase2.c"
        .text
        .align 2
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $-2147483648, -4(%ebp)
        movl    $-2147483648, %eax
        movl    %eax, -8(%ebp)
        movl    $-2147483647, -12(%ebp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    f
        movl    -8(%ebp), %eax
        movl    %eax, (%esp)
        call    f
        movl    -12(%ebp), %eax
        movl    %eax, (%esp)
        call    f
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) 3.2"
```

B.2 xgcc Intel 32 Bit nach IA64

```
        .file   "testcase2.c"
        .pred.safe_across_calls p1-p5,p16-p63
        .text
        .align 16
        .global main#
        .proc main#
main:
        .prologue 14, 35
        .save ar.pfs, r36
        alloc r36 = ar.pfs, 2, 4, 1, 0
        .vframe r37
        mov r37 = r12
        adds r12 = -48, r12
        .save rp, r35
        mov r35 = b0
        .body
        ;;
        adds r14 = -32, r37
        ;;
        st4 [r14] = r32
        adds r14 = -24, r37
        ;;
        st8 [r14] = r33
        adds r15 = -16, r37
        movl r14 = 2147483648
        ;;
        st8 [r15] = r14
        adds r15 = -8, r37
        movl r14 = -2147483648
        ;;
        st8 [r15] = r14
        mov r15 = r37
        movl r14 = 2147483649
        ;;
        st8 [r15] = r14
        adds r14 = -16, r37
        ;;
        ld8 r38 = [r14]
        mov r34 = r1
        br.call.sptk.many b0 = f#
        ;;
        mov r1 = r34
        adds r14 = -8, r37
        ;;
        ld8 r38 = [r14]
        mov r34 = r1
        br.call.sptk.many b0 = f#
```

```
        ;;
        mov r1 = r34
        mov r14 = r37
        ;;
        ld8 r38 = [r14]
        mov r34 = r1
        br.call.sptk.many b0 = f#
        ;;
        mov r1 = r34
        mov r8 = r14
        mov ar.pfs = r36
        mov b0 = r35
        .restore sp
        mov r12 = r37
        br.ret.sptk.many b0
        ;;
        .endp main#
        .ident  "GCC: (GNU) 3.2"
```

B.3 lcc Testfälle

i32 nach i32:

```
.globl main
.text
.align 16
.type main,@function
main:
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi
movl %esp,%ebp
subl $8,%esp
movl $4711,-4(%ebp)
mov $2147483613,%edi
leal 119(%edi),%edi
movl %edi,-8(%ebp)
pushl -4(%ebp)
call g
addl $4,%esp
pushl -8(%ebp)
call g
addl $4,%esp
mov $0,%eax
.LC1:
movl %ebp,%esp
```

```
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
.Lf2:
.size main,.Lf2-main
.ident "LCC: 4.1"
```

i32 nach alpha/osf:

```
.globl main
.text
.text
.ent main
main:
ldgp $gp,0($27)
lda $sp,-96($sp)
.mask 0x4000000,-96
.frame $sp,96,$26,48
stq $26,0($sp)
stq $16,48($sp)
stq $17,56($sp)
.prologue 1
lda $27,0x80000000
stq $27,-64+96($sp)
lda $27,2147483647
lda $27,1($27)
sll $27,8*(8-4),$27
sra $27,8*(8-4),$27
stq $27,-72+96($sp)
lda $27,0x80000001
stq $27,-80+96($sp)
ldq $16,-64+96($sp)
jsr $26,f
ldgp $gp,0($26)
ldq $16,-72+96($sp)
jsr $26,f
ldgp $gp,0($26)
ldq $16,-80+96($sp)
jsr $26,f
ldgp $gp,0($26)
mov $31,$0
L.1:
ldq $26,0($sp)
lda $sp,96($sp)
ret
.end main
```