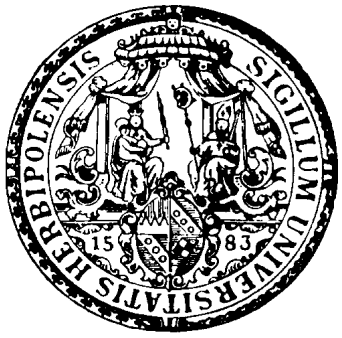
# Effiziente Intervallarithmetik mit C++

**Diplomarbeit im Fach Informatik**

vorgelegt von

Marco Nehmeier

16. Februar 2009




Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik II
Programmiersprachen und Programmiermethodik

betreut von

Prof. Dr. Jürgen Wolff von Gudenberg

# Inhaltsverzeichnis

# Vorwort

Im Bereich des wissenschaftlichen Rechnens stößt die Intervallarithmetik in den letzten Jahren auf immer mehr Akzeptanz. Eigenschaften, wie das sichere Einschließen der Ergebnisse bei der Intervallauswertung, machen sie zu einem guten Mittel um mit den Rundungsfehlern der Gleitkommazahlen umzugehen. Auch können Berechnungen auf Wertemengen mit garantierten Grenzen durchgeführt werden. Ebenso wird in Anwendungsgebieten wie der globalen Optimierung mittels "Branch and Bound" Algorithmen, der Robotik oder der Computergraphik die Intervallrechnung mit Erfolg eingesetzt.

Im Gegensatz zu der vom "Institute of Electrical and Electronics Engineers" (IEEE) standardisierten Gleitkommaarithmetik [37, 38], existiert für die Intervallrechnung jedoch keine Norm oder Implementierungsvorschrift. Viele verschiedene Implementierungen und Konzepte wurden über die Jahre entwickelt und auch eingesetzt. Eine einheitliche Semantik wie bei den Gleitkommazahlen ist jedoch nicht gegeben. Um die Unterstützung der Intervallarithmetik auf Rechnern zu verbessern, aber auch um die Akzeptanz bei Benutzern weiter zu erhöhen wird aktuell an einem IEEE Standard für Intervallrechnung [19] gearbeitet.

Diese Diplomarbeit beschäftigt sich mit den Möglichkeiten einer effizienten Implementierung der Intervallarithmetik auf Computersystemen mittels der Programmiersprache C++.

# Teil I

# Grundlagen

# Kapitel 1

# Reellwertige Intervallrechnung

Im ersten Kapitel dieser Arbeit wird in die reelle Intervallrechnung durch ihre mathematischen Definitionen und Eigenschaften eingeführt. Hierbei handelt es sich lediglich um einen Überblick über dieses Themengebiet, um die Grundlagen für spätere Kapitel zu vermitteln. Eine ausführliche Abhandlung über die reelle Intervallarithmetik findet sich unter anderem im Buch "Einführung in die Intervallrechnung" [3] von Götz Alefeld und Jürgen Herzberger.

## 1.1 Reellwertige Intervalle

In der Intervallrechnung werden, wie der Name vermuten lässt, Berechnungen nicht wie gewohnt auf Zahlen, sondern auf Zahlenmengen, den Intervallen, durchgeführt.

**Definition 1 (Abgeschlossene reellwertige Intervalle)** *Die Teilmenge* $A := \{x \in \mathbb{R} \mid a_1 \leq x \leq a_2\} \subseteq \mathbb{R}$ *für beliebige* $a_1, a_2 \in \mathbb{R}$ *heißt abgeschlossenes reellwertiges Intervall. Im Weiteren nur kurz Intervall genannt. Man schreibt* $A = [a_1\,,\, a_2]$.

Ein Intervall stellt somit, wie in Abbildung 1.1 für das Intervall $[1.2\,,\, 3.7]$ verdeutlicht, einen kontinuierlichen, abgeschlossenen Bereich der Zahlengerade von $\mathbb{R}$ dar und wird durch eine untere Grenze $a_1 \in \mathbb{R}$ sowie eine obere Grenze $a_2 \in \mathbb{R}$ beschränkt. Durch die Wahl der Grenzen $a_1, a_2 \in \mathbb{R}$ werden über das Inter-

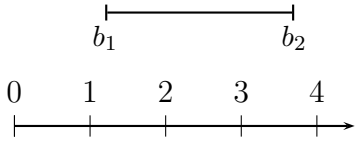



Abbildung 1.1: Intervall $[1.2\,,\, 3.7]$

vall $A = [a_1\,,\, a_2]$ drei verschiedene Arten von Mengen und somit auch Intervalle definiert:

**Punktintervall:** Gilt für die beiden Grenzen $a_1, a_2 \in \mathbb{R}$ die Gleichheit $a_1 = a_2$, so schließt das Intervall $A = [a_1\,,\, a_2]$ lediglich den reellen Punkt $a = a_1 = a_2$ ein. Das Intervall entspricht somit der einelementigen Menge $A = \{a\}$ und wird als Punktintervall bezeichnet.

**Echtes Intervall:** Sind die beiden Grenzen $a_1, a_2 \in \mathbb{R}$ verschieden und es gilt $a_1 < a_2$, so existieren unendlich viele $x \in \mathbb{R}$, welche die Bedingung $a_1 \leq x \leq a_2$ erfüllen. Das Intervall $A = [a_1\,,\, a_2]$ entspricht somit der Menge $\mathrm{A} = \{x \in \mathbb{R} \mid a_1 \leq x \leq a_2\}$ und heißt echtes Intervall.

**Leere Menge:** Im dritten möglichen Fall gilt für die beiden Grenzen $a_1, a_2 \in \mathbb{R}$ die Ungleichheit $a_1 > a_2$. Ein Intervall $A = [a_1\,,\, a_2]$ enthält somit keinen Punkt und entspricht der leeren Menge $A = \emptyset$.

Für arithmetische Operationen ist eine Grundmenge notwendig, auf welcher diese Operationen definiert werden können. Diese Menge bildet der Zusammenschluß aller Punktintervalle, echten Intervalle sowie der leeren Menge.

**Definition 2** $\mathbb{IR}$ *bezeichnet die Menge aller abgeschlossenen reellwertigen Intervalle.*

Mit Definition 1 der Intervalle als abgeschlossene, kontinuierliche und beschränkte Teilmenge der reellen Zahlen $\mathbb{R}$ lässt sich auch sofort folgender Bezug

$$\mathbb{IR} \subset \wp(\mathbb{R}) \tag{1.1}$$

zur Potenzmenge $\wp(\mathbb{R}) := \{x \mid x \subseteq \mathbb{R}\}$ der reellen Zahlen herstellen. Die echte Teilmengenrelation "$\subset$" liegt hier vor da die Elemente der Potenzmenge $\wp(\mathbb{R})$ im Gegensatz zu Intervallen aus $\mathbb{IR}$, nicht zwingend abgeschlossen, kontinuierlich oder beschränkt sind und somit eine Gleichheit ausgeschlossen ist.

### 1.1.1 Mittelpunkt-Radius Darstellung

Neben der Darstellung eines Intervalls $A = [a_1\,,\, a_2]$ über seine beiden Grenzen $a_1$ und $a_2$ können Intervalle natürlich auch über ihren Mittelpunkt

$$\mu A := \frac{a_1 + a_2}{2} \tag{1.2}$$

sowie den zugehörigen Radius

$$\rho A := \frac{a_2 - a_1}{2} \tag{1.3}$$

wie folgt definiert werden:

$$A := [\mu A\,;\, \rho A] = [\mu A - \rho A\,,\, \mu A + \rho A] \tag{1.4}$$

Wir unterscheiden die Mittelpunkt-Radius Notation hierbei nur durch einen Strichpunkt von der gebräuchlicheren Notation über Unter- und Obergrenze.

## 1.2 Relationen

Nach der Definition der abgeschlossenen reellwertigen Intervalle und deren Zusammenschluß zur Menge $\mathbb{IR}$ ist natürlich auch von Interesse, wie einzelne Intervalle aus $\mathbb{IR}$ zueinander in Bezug stehen. Hierzu müssen einige Relationen für Intervalle definiert werden.

### 1.2.1 Teilmengenrelationen

Durch die Definition der Menge $\mathbb{IR}$ als Teilmenge der Potenzmenge $\wp(\mathbb{R})$ können viele Eigenschaften der Potenzmenge übernommen werden. So gelten die aus der Mengentheorie bekannten Teilmengenrelationen $\subset, \subseteq, \supset$ sowie $\supseteq$ auch für Intervalle $A, B \in \mathbb{IR}$ wie gewohnt.

$$A \subseteq B :\Leftrightarrow \forall x \in A : x \in B \tag{1.5}$$
$$A \subset B :\Leftrightarrow A \subseteq B \wedge \exists x \in B : x \notin A \tag{1.6}$$
$$A \supseteq B :\Leftrightarrow B \subseteq A \tag{1.7}$$
$$A \supset B :\Leftrightarrow B \subset A \tag{1.8}$$

### 1.2.2 Mengentheoretische Gleichheitsrelation

Mit Hilfe der Relation $\subseteq$ aus (1.5) kann nun, analog zur mengentheoretischen Gleichheitsrelation, die in der Intervallarithmetik wichtige Gleichheitsrelation zwischen zwei Intervallen $A, B \in \mathbb{IR}$ definiert werden.

**Definition 3 (Gleichheit)** *Für die Intervalle $A, B \in \mathbb{IR}$ gilt die Relation $A = B$ genau dann, wenn die beiden Bedingungen*

$$A \subseteq B \tag{1.9}$$
$$B \subseteq A \tag{1.10}$$

*erfüllt sind.*

Natürlich ist die Gleichheitsrelation auf $\mathbb{IR}$ wie auch auf der Potenzmenge $\wp(\mathbb{R})$ transitiv, reflexiv und symmetrisch.

Durch die Defintion der Intervalle als kontinuierliche und abgeschlossene Menge mittels ihrer Ober- und Untergrenze, kann die Gleichheit zwischen zwei nichtleeren Intervallen $A = [a_1\,,\, a_2] \in \mathbb{IR}$ und $B = [b_1\,,\, b_2] \in \mathbb{IR}$ auf die Gleichheit der jeweiligen Grenzen vereinfacht werden. Ein elementweises Vergleichen der beiden Mengen ist nicht notwendig.

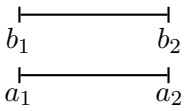



Abbildung 1.2: Gleichheit bei echten Intervallen

Wie in Abbildung 1.2 am Beispiel zweier echter Intervalle dargestellt, gilt die Gleichheit $A = B$ genau dann, wenn die beiden Bedingungen

$$a_1 = b_1 \tag{1.11}$$
$$a_2 = b_2 \tag{1.12}$$

erfüllt sind.

### 1.2.3 Vergleichsrelationen

Die, bei den reellen Zahlen üblichen, Ungleichheitsrelationen wie $<$ oder $>$ sind aufgrund der Definition der Intervalle als Mengen, nicht problemlos auf die Intervallarithmetik übertragbar. Betrachtet man beispielsweise die Relation $A < B$ für Intervalle $A, B \in \mathbb{IR}$, so sind mehrere verschiedene Interpretationen möglich. Soll die Relation für alle Paare aus $A \times B$ gelten oder ist ein Paar $(a, b) \in A \times B$ für eine erfüllte Relation ausreichend? Diese Frage kann nicht allgemein beantwortet werden, vielmehr muss man hier von Anwendungsfall zu Anwendungsfall unterscheiden.

So macht es bei einer Implementierung der Intervallarithmetik auf Rechnern durchaus Sinn die Ungleichheitsrelationen $<$, $\leq$, $>$ und $\geq$, aber auch die Gleichheitsrelation $=$, auf mehrere verschiedene Arten in Hard- oder Software zu realisieren, um einen Großteil der denkbaren Anwendungsfälle abzudecken.

#### “certainly”-Relationen

Eine mögliche Interpretation ist die strikte Umsetzung der reellwertigen Relation $\circ \in \{=, <, \leq, >, \geq\}$ auf Intervalle, so dass für zwei Intervalle $A, B \in \mathbb{IR}$ folgende Bedingung erfüllt ist.

$$A \circ B :\Leftrightarrow \forall a \in A, \forall b \in B : a \circ b \tag{1.13}$$

Die reellwertige Relation $\circ$ ist also für jede Elementkombination aus $A \times B$ “sicher” oder “gewiss” erfüllt und wird deshalb oft als “certainly”-Relation bezeichnet.

Für eine Realisierung auf Rechnern ist es durch die Definition der Intervalle als kontinuierliche und abgeschlossene Mengen jedoch nicht notwendig jede Elementkombination zu prüfen. Vielmehr genügt bei nicht leeren Intervallen ein Vergleich der Grenzen.

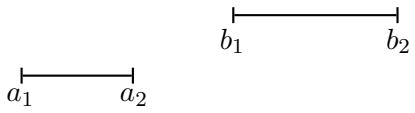



Abbildung 1.3: Beispiel für $A < B$ als "certainly"-Relation

Wie in Abbildung 1.3 dargestellt, genügt es beispielsweise für die Ungleichung $A < B$ die Überprüfung nur auf die obere Grenze $a_2$ von $A$ sowie die untere Grenze $b_1$ von $B$ zu beschränken.

**Hinweis:** Die Gleichheit $=$ entspricht als "certainly"-Relation der mengentheoretischen Gleichheit aus Definition 3.

### "possibly"-Relationen

Aber auch die Forderung, dass eine reellwertige Relation $\circ \in \{=, <, \leq, >, \geq\}$ für ein $a \in A$ und ein $b \in B$ gilt, ist berechtigt und in bestimmten Anwendungsfällen sinnvoll. In diesem Fall wird eine Relation $\circ$, welche folgende Bedingung erfüllt,

$$A \circ B :\Leftrightarrow \exists a \in A, \exists b \in B : a \circ b \tag{1.14}$$

auch als "possibly"-Relation bezeichnet, da nur für ein Zahlenpaar aus der Menge $A \times B$ die reellwertige Relation erfüllt sein muss und somit auf Intervallen nur möglicherweise zutrifft. Natürlich muss bei der "possibly"-Relation, wie auch schon bei der "certainly"-Relation, nicht jede Kombination aus $A \times B$ geprüft werden. Es genügt auch hier eine Betrachtung der Intervallgrenzen.


Abbildung 1.4: Zwei Beispiele für $A < B$ als "possibly"-Relation

Abbildung 1.4 zeigt am Beispiel der Relation $<$, dass durch die Beschränkung auf nur ein Zahlenpaar aus $A \times B$, im Vergleich zum reellwertigen Gegenstück, auch etwas "unerwartete" Intervallkombinationen die Relation erfüllen können.

## 1.3 Operationen

Um mit der Intervallarithmetik, analog zu den reellen Zahlen, Berechnungen durchführen zu können, sind natürlich auch die gewohnten Rechenoperationen Addition, Subtraktion, Multiplikation und Division sowie Standardfunktionen wie beispielsweise Sinus, Kosinus oder die Exponentialfunktion notwendig.

### 1.3.1 Grundrechenarten

Durch die Definition von $\mathbb{IR}$ als Teilmenge der reellen Potenzmenge $\wp(\mathbb{R})$ ergibt sich die Verknüpfung $\circ : \mathbb{IR}^2 \to \mathbb{IR}$ von zwei Intervallen $A, B \in \mathbb{IR}$

$$A \circ B := \{a \circ b \mid a \in A, b \in B\} \tag{1.15}$$

durch paarweises Anwenden der zugrunde liegenden reellen Verknüpfung $\circ : \mathbb{R}^2 \to \mathbb{R}$. Handelt es sich bei $\circ : \mathbb{R}^2 \to \mathbb{R}$ um eine stetige Verknüpfung auf der Menge $A \times B$, so ist auch $A \circ B$ eine abgeschlossene, kontinuierliche und beschränkte Menge, also ein Intervall. Die vier reellen Operationen $+, -, \cdot, \div$ lassen sich somit problemlos auf die Intervallrechnung abbilden.

**Satz 1 (Rechenregeln)** *Es sei $\circ \in \{+, -, \cdot, \div\}$ eine zweistellige stetige Operation für reelle Zahlen. Dann gilt für $A, B \in \mathbb{IR}$*

$$A \circ B = \{a \circ b \mid a \in A, b \in B\} \in \mathbb{IR}. \tag{1.16}$$

*Bei der Division $\div$ wird $0 \notin B$ vorausgesetzt.*

**Beweis:** Folgt direkt aus der Stetigkeit der reellen Verknüpfungen. □

Durch die Stetigkeit der reellen Operationen $+, -, \cdot, \div$ kann auf die Auswertung von unendlich vielen Zahlenpaaren verzichtet werden. Vielmehr genügt es die Grenzen der Intervalle zu betrachten.

**Folgerung 1** *Für $A = [a_1\,,\, a_2] \in \mathbb{IR}$ und $B = [b_1\,,\, b_2] \in \mathbb{IR}$ gelten folgende Rechenregeln:*

$$A + B = [a_1 + b_1\,,\, a_2 + b_2] \tag{1.17}$$

$$A - B = [a_1 - b_2\,,\, a_2 - b_1] \tag{1.18}$$

$$A \cdot B = [\min\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}\,,\, \max\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}] \tag{1.19}$$

$$A \div B = [a_1\,,\, a_2] \cdot [\frac{1}{b_2}\,,\, \frac{1}{b_1}] \quad \textit{mit} \quad 0 \notin B \tag{1.20}$$

### 1.3.2 Standardfunktionen

Für viele Berechnungen mit Intervallen werden die vier Grundrechenarten nicht ausreichen. Vielmehr ist es das Ziel die reellwertige Arithmetik auf die Intervallarithmetik zu überführen. Hierzu ist es notwendig auch reellwertige Funktionen $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$ auf Intervalle abzubilden. Da es sich bei Intervallen aber um Mengen handelt, müssen diese Funktionen für Mengen definiert werden. Dies wird über die Wertemenge einer Funktion realisiert.

**Definition 4** *$f(X) = \{f(x) \mid x \in X\}$ bezeichnet die Wertemenge einer Funktion $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$ über dem Intervall $X \in \mathbb{IR}$ mit $X \subseteq D_f$.*

Handelt es sich dabei um eine auf dem Intervall $X \subseteq D_f$ stetige Funktion $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$, so ist der Wertebereich von $f(X)$ ebenfalls abgeschlossen, kontinuierlich und beschränkt, und somit ein Intervall.

**Satz 2** *Ist die Funktion $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$ stetig über dem Intervall $X \in \mathbb{IR}$ mit $X \subseteq D_f$, so gilt*

$$f(X) = [\min\{f(x)_{x\in X}\}, \max\{f(x)_{x\in X}\}] \in \mathbb{IR}. \tag{1.21}$$

**Beweis:** Folgt direkt aus der Stetigkeit der Funktion $f$. □

## 1.4 Intervallauswertung

Durch die Überführung der Grundrechenarten und Standardfunktionen auf Intervalle wurde die Basis für die Intervallrechnung geschaffen. Jedoch treten in der reellen Arithmetik oft Funktionen $f : \mathbb{R} \to \mathbb{R}$ auf, welche über die Möglichkeiten der Grundrechenarten und Standardfunktionen hinausgehen. Somit muss eine allgemeine Funktionsauswertung für reelle stetige Funktionen $f : \mathbb{R} \to \mathbb{R}$ auf Intervallen geschaffen werden. Wir beschränken uns hier auf den eindimensionalen Raum, eine Verallgemeinerung auf mehrere Dimensionen ist jedoch problemlos möglich.

**Definition 5 (Intervallauswertung)** *Die Intervallauswertung $F : \mathbb{IR} \to \mathbb{IR}$ einer reellen Funktion $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$ über ein Intervall $X$ erhält man, indem jede Operation und jede Standardfunktion durch ihre intervallwertige Version ersetzt wird und $F : \mathbb{IR} \to \mathbb{IR}$ auf dem Intervall $X$ definiert ist.*

Man erhält $F(X)$ also durch ein Überführen der Funktion $f(x)$, indem man jede Variable $x$ durch das Intervall $X$ sowie die Operationen und Standardfunktionen durch die intervallwertigen Gegenstücke ersetzt. Natürlich kann $F(X)$ nur definiert sein wenn $X \subseteq D_f$ gilt. Jedoch folgt aus $X \subseteq D_f$ nicht zwingend

die Gültigkeit von $F(X)$. Das Problem ist die syntaktische Formulierung einer Funktion $f(x)$ und der daraus folgenden Intervallauswertung $F(X)$. Nimmt man folgende reelle Funktion als Beispiel für eine Intervallauswertung

$$f_1(x) = \frac{1}{x \cdot x + 2} \tag{1.22}$$

auf dem Intervall $X = [-2\,,\,2]$, so wird die Multiplikation $x \cdot x$ für Intervalle durch $X \cdot X = [-2\,,\,2] \cdot [-2\,,\,2] = [-4\,,\,4]$ ersetzt. Demnach entspricht der Nenner der Funktion dem Intervall $[-4\,,\,4] + [2\,,\,2] = [-2\,,\,6]$ und es würde bei der Auswertung der Funktion eine nicht zulässige Division mit 0 durchgeführt. Jedoch kann durch Umformung des Ausdrucks eine äquivalente Funktion

$$f_2(x) = \frac{1}{x^2 + 2} \tag{1.23}$$

gefunden werden, für welche eine gültige Intervallauswertung existiert. Nach Satz 2 wird $x^2$ für ein Intervall $X = [-2\,,\,2]$ durch $X^2 = [0\,,\,4]$ ersetzt. Es findet somit keine Division durch 0 statt und die Intervallauswertung der Funktion ist definiert.

### 1.4.1 Hauptsätze der Intervallarithmetik

Die Intervallauswertung ist nun für Funktionen definiert. Doch wie sieht das Ergebnis einer Auswertung von $F(X)$ im Vergleich zum Wertebereich $f(X)$ einer Funktion $f : \mathbb{R} \to \mathbb{R}$ auf dem Intervall $X$ aus? Die Antwort liefert der folgende fundamentale Satz der Intervallarithmetik.

**Satz 3 (Einschließungseigenschaft)** *Ist die Intervallauswertung $F : \mathbb{IR} \to \mathbb{IR}$ einer reellen Funktion $f$ auf dem Intervall $X$ definiert, so gilt*

$$f(X) \subseteq F(X). \tag{1.24}$$

**Beweis:** Siehe [3]. □

Durch eine Intervallauswertung ist also sichergestellt, dass durch das Ergebnis die komplette Wertemenge der Funktion abgedeckt wird. Für jedes $x \in X$ folgt somit auch $f(x) \in F(X)$, was sich auch im zweiten wichtigen Satz der Intervallarithmetik widerspiegelt.

**Satz 4 (Teilmengeneigenschaft)** *Für Intervalle $X \subseteq Y$ gilt bei einer Intervallauswertung $F$ folgende Inklusion*

$$F(X) \subseteq F(Y). \tag{1.25}$$

**Beweis:** Siehe [3]. □

## 1.5 Erweiterte Intervallrechnung

Eine Intervallrechnung ist durch die in Definition 5 eingeführte Intervallauswertung nun ermöglicht. Jedoch bereiten partiell definierte Funktionen Probleme, was eine Auswertung vieler Ausdrücke verbietet. So ist beispielsweise die Division durch Null oder die Quadratwurzel einer Negativen Zahl nicht definiert und die Intervallauswertung von

$$[5\,,\,7] \div [-1\,,\,1] \tag{1.26}$$

oder

$$\sqrt{[-1\,,\,1]} \tag{1.27}$$

würde bei einer rechnergestützten Auswertung mit einer Ausnahme abbrechen. Ein Anwender müsste in diesen Fällen den jeweiligen Ausdruck anpassen, um eine gültige Intervallauswertung zu erhalten.

Jedoch ist es in der Praxis oft gewünscht eine ausnahmefreie Auswertung durchzuführen. Um diesen Ansprüchen gerecht zu werden, wird die erweiterte Intervallrechnung mittels loser Funktionsauswertung auf einer topologischen Erweiterung der reellen Zahlen verwendet [55].

### 1.5.1 Topologische Erweiterung der reellen Zahlen

Ziel dieser Erweiterung ist es auch undefinierte Ausdrücke und Funktionen wie Division durch Null oder die Quadratwurzel negativer Zahlen über Grenzwertbetrachtung und lose Funktionsauswertung zu realisieren. Um jedoch alle Werte als Ergebnis zulassen zu können, müssen die reellen Zahlen $\mathbb{R}$ um die Symbole $-\infty$ und $+\infty$ für unendliche Grenzwerte erweitert werden.

**Definition 6 (Erweiterte reelle Zahlen)**

$$\mathbb{R}^* := \mathbb{R} \cup \{-\infty\} \cup \{+\infty\} \tag{1.28}$$

Im Raum der erweiterten reellen Zahlen $\mathbb{R}^*$ können nun auch Grenzwertbetrachtungen durchgeführt werden. Verknüpfungen zweier Elemente $a, b \in \mathbb{R}^*$ werden dann mittels Verknüpfungen auf den Grenzwerten ausgewertet [53].

$$a \circ b := \{ \lim_{k\to\infty} (a_k \circ b_k) \mid \lim_{k\to\infty} a_k = a, \lim_{k\to\infty} b_k = b\} \tag{1.29}$$

Eine Verknüpfung liefert also nicht wie bei den reellen Zahlen ein Element als Ergebnis, sondern die Menge aller Verknüpfungen der Grenzwerte. Die vier Grundrechenarten können somit, wie in den in Tabellen 1.1, 1.2, 1.3 und 1.4, realisiert werden. Analog zu den Grundrechenarten werden auch die Standardfunktionen durch Grenzwertbetrachtung ausgewertet [53].

| $+$ | $-\infty$ | $b$ | $+\infty$ |
|---|---|---|---|
| $-\infty$ | $-\infty$ | $-\infty$ | $\mathbb{R}^*$ |
| $a$ | $-\infty$ | $a+b$ | $+\infty$ |
| $+\infty$ | $\mathbb{R}^*$ | $+\infty$ | $+\infty$ |

Tabelle 1.1: Erweiterte Addition

| $-$ | $-\infty$ | $b$ | $+\infty$ |
|---|---|---|---|
| $-\infty$ | $\mathbb{R}^*$ | $-\infty$ | $-\infty$ |
| $a$ | $+\infty$ | $a-b$ | $-\infty$ |
| $+\infty$ | $+\infty$ | $+\infty$ | $\mathbb{R}^*$ |

Tabelle 1.2: Erweiterte Subtraktion

| $\cdot$ | $-\infty$ | $b<0$ | $0$ | $b>0$ | $+\infty$ |
|---|---|---|---|---|---|
| $-\infty$ | $+\infty$ | $+\infty$ | $\mathbb{R}^*$ | $-\infty$ | $-\infty$ |
| $a<0$ | $+\infty$ | $a\cdot b$ | $0$ | $a\cdot b$ | $-\infty$ |
| $0$ | $\mathbb{R}^*$ | $0$ | $0$ | $0$ | $\mathbb{R}^*$ |
| $a>0$ | $-\infty$ | $a\cdot b$ | $0$ | $a\cdot b$ | $+\infty$ |
| $+\infty$ | $-\infty$ | $-\infty$ | $\mathbb{R}^*$ | $+\infty$ | $+\infty$ |

Tabelle 1.3: Erweiterte Multiplikation

| $\div$ | $-\infty$ | $b<0$ | $0$ | $b>0$ | $+\infty$ |
|---|---|---|---|---|---|
| $-\infty$ | $[0\,,\,+\infty]$ | $+\infty$ | $\{-\infty,+\infty\}$ | $-\infty$ | $[-\infty\,,\,0]$ |
| $a<0$ | $0$ | $a\div b$ | $\{-\infty,+\infty\}$ | $a\div b$ | $0$ |
| $0$ | $0$ | $0$ | $\mathbb{R}^*$ | $0$ | $0$ |
| $a>0$ | $0$ | $a\div b$ | $\{-\infty,+\infty\}$ | $a\div b$ | $0$ |
| $+\infty$ | $[-\infty\,,\,0]$ | $-\infty$ | $\{-\infty,+\infty\}$ | $+\infty$ | $[0\,,\,+\infty]$ |

Tabelle 1.4: Erweiterte Division

### 1.5.2 Erweiterte Intervalle

Nachdem die erweiterten reellen Zahlen $\mathbb{R}^*$ sowie die grundlegende Arithmetik definiert wurden, kann nun auch die Definition für Intervalle angepasst werden. Analog zur Definition 1 der reellwertigen Intervalle folgt die Definition für erweiterte reellwertige Intervalle.

**Definition 7 (Erweiterte reellwertige Intervalle)** *Die Teilmenge* $A := \{x \in \mathbb{R}^* \mid a_1 \leq x \leq a_2\} \subseteq \mathbb{R}^*$ *für beliebige* $a_1, a_2 \in \mathbb{R}^*$ *heißt erweitertes reellwertiges Intervall oder kurz erweitertes Intervall. Man schreibt* $A = [a_1\,,\, a_2]$.

Wie auch bei den reellwertigen Intervallen können die erweiterten Intervalle in Abhängigkeit der Grenzen $a_1, a_2 \in \mathbb{R}^*$ wieder in den folgenden drei verschiedenen Formen auftreten.

- Punktintervalle $[a\,,\, a]$ für $a = a_1 = a_2$,
- echte Intervalle $[a_1\,,\, a_2]$ für $a_1 < a_2$
- sowie die leere Menge $\emptyset$ für $a_1 > a_2$

Die Menge der reellwertigen Intervalle $\mathbb{IR}$ wird für erweiterte Intervalle durch die Menge $\mathbb{IR}^*$ ersetzt.

**Definition 8** $\mathbb{IR}^*$ *bezeichnet die Menge aller erweiterten reellwertigen Intervalle.*

**Hinweis:** Bei Anwendern und Wissenschaftlern im Bereich der Intervallarithmetik existieren unterschiedliche Meinungen über die Verwendung der Werte $\pm\infty$ als Intervallgrenzen. Diese werden auch im Rahmen der IEEE Standardisierung diskutiert [36]. Hierbei werden hauptsächlich die drei folgenden Ansätze vertreten:

1. Intervalle werden als Teilmenge der reellen Zahlen angesehen. Die Werte $\pm\infty$ symbolisieren nur uneigentliche Grenzwerte und sind nicht im Intervall enthalten. Dementsprechend sind die drei unbeschränkten Intervalle $(-\infty\,,\, +\infty)$, $(-\infty\,,\, a]$ und $[a\,,\, +\infty)$ mit $a \in \mathbb{R}$ möglich.

2. Die Werte $\pm\infty$ sind Teil des Intervalls. Die Grundrechenarten werden für die Werte $\pm\infty$ entsprechend den Tabellen 1.1, 1.2, 1.3 und 1.4 durchgeführt.

3. Wie Punkt 2, jedoch werden die Intervalle $[-\infty\,,\, -\infty]$ sowie $[+\infty\,,\, +\infty]$ nicht zugelassen.

### 1.5.3 Auswertung mit Containment Sets

Eine Arithmetik mit erweiterten Intervallen kann durch Containment Sets[1] [44] realisiert werden. Hierbei werden die Operationen $+, -, \cdot, \div$ auf erweiterten Intervallen wie in Satz 1 ausgewertet. Jedoch wird die Verknüpfung durch die, unter (1.29) angegebene, Verknüpfung für Elemente aus $\mathbb{R}^*$ ersetzt. Die vier Grundrechenarten werden bei erweiterten Intervallen also über die Menge aller Häufungspunkte ausgewertet und als Ergebnis vereinigt. Bei Funktionen auf $\mathbb{IR}^*$ wird eine lose Funktionsauswertung über die Häufungspunkte durchgeführt.

**Definition 9 (Containment Set)** *Für eine Funktion $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$ ist das Containment Set $f^* : \wp(\mathbb{R}^*) \to \wp(\mathbb{R}^*)$ definiert durch*

$$f^*(X) := \{f(x) \mid x \in X \cap D_f\} \cup \{\lim_{x \to x^*} f(x) \mid x \in D_f, x^* \in X\} \subseteq \mathbb{R}^* \tag{1.30}$$

Als Ergebnis der vier Grundrechenarten oder Standardfunktionen mit erweiterten Intervallen wird das kleinste Intervall verwendet, welches das Containment Set der Operation oder Funktion einschließt. Analog zur Intervallauswertung in Definition 5 kann nun eine Auswertung auf Containment Sets durchgeführt werden.

**Definition 10 (Auswertung mit Containment Sets)** *Eine Auswertung auf Containment Set $F^* : \mathbb{IR}^* \to \mathbb{IR}^*$ einer reellen Funktion $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$ über einem Intervall $X$ erhält man, indem jeder Operator und jede elementare Funktion durch ihre erweiterte intervallwertige Version ersetzt wird.*

Durch die lose Funktionsauswertung und die Betrachtung der Häufungspunkte ist nun eine Intervallauswertung auch für partiell definierte Funktionen gegeben. Weiter ist durch die Definition der Containment Sets auch sichergestellt, dass der exakte Wertebereich der Funktion auch im Ergebnisintervall der Containment Set Auswertung enthalten ist.

**Satz 5** *$F^*(X)$ ist immer definiert und es gilt*

$$f(X) \subseteq f^*(X) \subseteq F^*(X) \tag{1.31}$$

**Beweis:** Siehe [28]. □

Der Vorteil einer ausnahmefreien Auswertung von Intervallausdrücken mit Containment Sets bringt aber auch Nachteile mit sich. Der in der Intervallrechnung häufig verwendete Satz von Brouwer zur Bestimmung von Fixpunkten setzt die Stetigkeit von Funktionen voraus.

[1] Containment Sets werden oft auch nur kurz CSets genannt.

**Satz 6 (Brouwers Fixpunktsatz)** *Ist eine Funktion $f : A \to A$ auf der Menge $A$ definiert und stetig, so hat $f$ einen Fixpunkt in $A$.*

**Beweis:** Ohne Beweis. □

Jedoch wird bei der losen Funktionsauswertung mittels Containment Sets keine Rücksicht auf die Stetigkeit genommen. So ist die folgende Funktion

$$f(x) = \sqrt{x} - 1 \tag{1.32}$$

nur im Bereich $D_f = [0\,,\, +\infty)$ definiert und, wie in Abbildung 1.5 zu sehen, frei von Fixpunkten. Eine lose Funktionsauswertung mittels Containment Sets ist

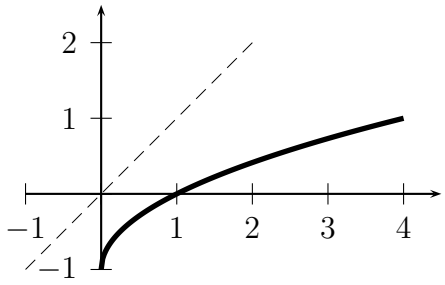



Abbildung 1.5: Funktion $f(x) = \sqrt{x} - 1$

aber auch auf Intervallen wie $[-4\,,\, 4]$ definiert und es gehen somit Informationen über die Stetigkeit verloren.

Bei einer Implementierung der erweiterten Intervallarithmetik sollte somit ein Flag berücksichtigt werden, welches über die Unstetigkeit der ausgewerteten Funktion informiert. Ein Anwender könnte so zum Beispiel bei einem Intervall Newtonverfahren [24] auf die Unstetigkeit eingehen und diese durch Bisektion der Intervalle behandeln.

# Kapitel 2

# Gleitkommazahlen

Durch den endlichen Zustandsraum eines Computers stößt man bei maschinellen Berechnungen zwangsläufig auf ein Problem. Die reelle Arithmetik lässt sich nicht auf einen Computer überführen. Es ist einfach nicht möglich eine (überabzählbar) unendliche Menge, wie die reellen Zahlen $\mathbb{R}$, durch endlich viele Zustände eines Rechners abzubilden. Es muss auf Computern also mit einer Approximation der reellen Zahlen gearbeitet werden.

## 2.1 $\beta$-adische Darstellung

Natürlich ist es so, dass durch die Einschränkung auf ein binäres Speichersystem reelle Zahlen nicht im uns geläufigen Dezimalsystem verarbeitet werden können. Man muss reelle Zahlen, analog zum Dezimalsystem, in einer für den Rechner verständlichen Form kodieren. Für diesen Zweck wird der "abstrakte" Wert einer Zahl nicht wie gewohnt zur Basis 10, sondern zu der von einem Computer darstellbaren Basis 2 kodiert.

Beide Systeme verwenden jedoch mit der $\beta$-adischen Darstellung [9] das gleiche Verfahren für die Kodierung. Sie unterscheiden sich lediglich durch die gewählte Basis $\beta \in \mathbb{N}$. Die Darstellung ist somit nicht auf das Binärsystem beschränkt.

Für die Kodierung einer reelle Zahl $x > 0$ kann man diese Zahl in zwei Summanden unterteilen.

$$x = n + r. \tag{2.1}$$

Zum einen in eine natürliche Zahl $n \in \mathbb{N}_0$, welche dem ganzen Teil der Zahl entspricht. Sowie eine reelle Zahl $r \in \mathbb{R}$ mit $0 \leq r < 1$ als gebrochenen Teil. Eine positive natürliche Zahl $n \in \mathbb{N}$ kann nun durch eine endliche Folge von Ziffern $n_0 \dots n_p$ zur gewählten Basis $\beta$ dargestellt werden.

$$n = \sum_{i=0}^{p} n_i \beta^i \tag{2.2}$$

Die Ziffern $n_i$ können hierbei ganzzahlige Werte von 0 bis $\beta - 1$ annehmen. Somit entspricht beispielsweise die natürlich Zahl $123_{10}$[1] im Dezimalsystem mit Basis $\beta = 10$ der Kodierung $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$ und im Dualsystem mit Basis $\beta = 2$ der Kodierung $1111011_2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$.

Auf ähnliche Weise kann auch eine positive reelle Zahl $r < 1$ über eine Folge von Ziffern dargestellt werden.

$$r = \sum_{i=1}^{\infty} r_i \beta^{-i} \tag{2.3}$$

Jedoch kann hier eine unendliche Folge von Ziffern $0 \leq r_i < \beta$ nötig sein.

Eine positive reelle Zahl $x \in \mathbb{R}$ kann somit über die beiden Reihen (2.2) und (2.3) als Folge von Ziffern

$$n_p \ldots n_0 \bullet r_{-1} r_{-2} \ldots \tag{2.4}$$

zu einer Basis $\beta$ kodiert werden. Die Zugehörigkeit der Ziffern zu der jeweiligen Reihe wird in der Notation wie gewohnt über einen Punkt oder ein Komma geregelt. Eine Dezimalzahl wie etwa $105.34_{10}$ oder eine Dualzahl wie $110.01_2$ kann somit über die beiden Reihen realisiert werden. Jedoch besteht hier immer noch das Problem, dass unendlich viele Ziffern notwendig sein können oder bei einer endlichen Folge von Ziffern der Speicherplatz eines Rechners nicht ausreicht.

Die reellen Zahlen müssen also auf eine endliche Menge beschränkt werden. Man könnte für eine maschinelle Darstellung einfach die Anzahl der Ziffern $n_i$ sowie $r_i$ jeweils auf eine bestimmte Anzahl festlegen. Dies würde, da die Trennung zwischen dem ganzen und dem gebrochenen Teil immer an der gleichen Stelle der Kodierung stattfindet, einer sogenannten “Festkommazahl” entsprechen. Allerdings ergibt sich hierdurch der Nachteil eines sehr eingeschränkten Bereichs der darstellbaren Zahlen.

## 2.2 Normalisierte Darstellung

Den Nachteilen der Festkommazahlen wirkt die Notation einer Zahl über eine Mantisse und einen Exponenten entgegen. Die Mantisse entspricht hierbei den einzelnen Ziffern einer Zahl, während der Exponent die Position des Kommas, also die Trennung zwischen ganzem und gebrochenem Teil, angibt. Im Dezimalsystem entspricht beispielsweise die Darstellung über die Mantisse 295.376 sowie dem Exponenten 2 der Dezimalzahl

$$29537.6 = 295.376 \cdot 10^2 \tag{2.5}$$

[1] Zur Identifikation der verwendeten Basis $\beta$ des Zahlensystems schreibt man die Basis oft auch tiefgestellt an eine Zahl.

und wird wegen der Verschiebung des Kommas als "Gleitkommazahl" bezeichnet. Im angegebenen Beispiel (2.5) fällt jedoch sofort auf, dass hier mehrere mögliche Kodierungen existieren.

$$\ldots = 29537.6 \cdot 10^0 = 2953.76 \cdot 10^1 = 295.376 \cdot 10^2 = \ldots \tag{2.6}$$

Um eine eindeutige Kodierung zu erreichen, kann man den Wertebereich der Mantisse auf positive Werte kleiner 1 festlegen und für die erste Ziffer einen Wert ungleich 0 voraussetzen. Im Beispiel (2.6) wäre somit nur $0.295376 \cdot 10^5$ eine gültige Notation. Dies nennt man eine normalisierte Darstellung [17].

**Satz 7 ($\beta$-adische normalisierte Darstellung)** *Eine reelle Zahl $r \in \mathbb{R}$ mit $r > 0$ lässt sich bezüglich einer Basis $\beta \in \mathbb{N}$ durch eine Folge von Ziffern $r_i \in \{0, \ldots \beta - 1\}$ mit $r_1 \neq 0$ sowie einer Ganzzahl $e \in \mathbb{Z}$ eindeutig darstellen und es gilt:*

$$r = \underbrace{\sum_{i=1}^{\infty}(r_i \beta^{-i})}_{Mantisse} \underbrace{\beta}_{Basis} \underbrace{e}_{Exponent} \tag{2.7}$$

**Beweis:** Ohne Beweis. □

## 2.3 Gleitkommasysteme

Bei der normalisierten $\beta$-adischen Darstellung kann die Mantisse jede reelle Zahl einnehmen, so dass $\beta^{-1} \leq Mantisse < 1$ gilt. Man muss für eine Maschinendarstellung also die Mantisse wie auch den Exponenten $e \in \mathbb{Z}$ jeweils durch darstellbare Zustände approximieren. Dies wird über eine festgelegte Anzahl an Bits als Ziffern in der Binärdarstellung gelöst. Abbildung 2.1 veranschaulicht dies für eine

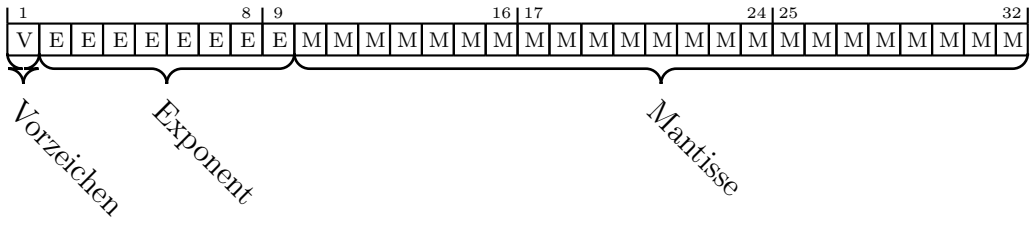



Abbildung 2.1: Gleitkommazahl mit Mantissenlänge 23, Expontentenlänge 8 und Vorzeichen

Gleitkommazahl mit einer Mantissenlänge 23 und einer Expontentenlänge 8. Ein zusätzliches Bit wird in diesem Beispiel für das Vorzeichen verwendet, um auch

negative Zahlen darstellen zu können. Über die endliche Reihe $\sum_{i=1}^{23}(r_i\beta^{-i})$ können somit für die Mantisse Werte aus dem Bereich $[\beta^{-1}, 1)$ verwendet werden. Für den Exponenten stehen $2^8$ verschiedene ganzzahlige Werte zur Verfügung. Der kleinste mögliche Exponent wird $e_{min}$, der größte Exponent $e_{max}$ genannt. Die, über die Mantisse der Länge $n$ sowie den Exponenten im Bereich $[e_{min}, e_{max}]$, darstellbaren Gleitkommazahlen bilden zusammen mit der Zahl 0 ein Gleitkommasystem [53].

**Definition 11 (Gleitkommasystem)** *Die reellen Zahlen in normalisierter $\beta$-adischer Darstellung mit Basis $\beta$, Mantissenlänge $n$ sowie den minimalen und maximalen Mantissenwerten $e_{min}$ und $e_{max}$ bilden zusammen mit der 0 das Gleitkommasystem $R = R(\beta, n, e_{min}, e_{max})$. Es gilt:*

$$R(\beta, n, e_{min}, e_{max}) := \{\pm \sum_{i=1}^{n}(r_i\beta^{-i})\beta^e \mid 0 \leq r_i < \beta, r_1 > 0, e_{min} \leq e \leq e_{max}\} \cup \{0\} \tag{2.8}$$

## 2.4 Genauigkeit und Rundung

Reelle Zahlen kann man über ein endliches Gleitkommasystem approximieren. Jedoch kann zwangsläufig nicht jede reelle Zahl in diesem endlichen System dargestellt werden. Sie müssen gegebenenfalls auf eine benachbarte Gleitkommazahl gerundet werden. Ausschlaggebend für die Genauigkeit einer Gleitkommazahl ist in erster Linie die Länge der Mantisse, welche dann mittels des Exponenten auf den gewünschten Wert der Zahl gebracht wird. Die Genauigkeit einer Gleitkommazahl lässt sich somit über die Einheit der letzten Mantissenstelle angeben.

**Definition 12 (Unit last place)** *Für eine $\beta$-adische normalisierte Gleitkommazahl $x$ zur Basis $\beta$ mit Mantissenlänge $n$ und Exponent $e_x$ bezeichnet $ulp(x)$ die Einheit in der letzten Mantissenstelle. Sie wird als "unit last place" oder kurz ulp bezeichnet. Es gilt:*

$$ulp(x) := \beta^{e_x - n} \tag{2.9}$$

Für eine Gleitkommazahl $x$ gibt $ulp(x)$ daher den Abstand zu den beiden benachbarten Gleitkommazahlen an. Der Vorgänger $pred(x)$ sowie der Nachfolger $succ(x)$ einer Gleitkommazahl $x$ kann somit über die Einheit der letzten Mantissenstelle berechnet werden.

$$pred(x) := \begin{cases} x - ulp(x) & x > 0 \\ x - ulp(x - ulp(x)) & x < 0 \end{cases} \tag{2.10}$$

$$succ(x) := \begin{cases} x + ulp(x) & x < 0 \\ x + ulp(x + ulp(x)) & x > 0 \end{cases} \tag{2.11}$$

Alle reellen Zahlen, welche zwischen einer Gleitkommazahl $x$ und ihrem Vorgänger $pred(x)$ beziehungsweise Nachfolger $succ(x)$ liegen, kann man demnach nicht über das zugrunde liegende Gleitkommasystem $R$ abbilden und müssen gerundet werden. Die auftretenden Rundungsfehler können hierbei für eine monotone Rundungsfunktion $\square : \mathbb{R} \rightarrow R$ in zwei Kategorien eingeteilt werden [53]. Den absoluten Rundungsfehler $\delta_x := x - \square x$ sowie den relativen Rundungsfehler $\epsilon_x := \delta_x / x$. Bei einer Rundung zur nächsten Gleitkommazahl[2] gilt für den relativen Rundungsfehler

$$\epsilon_x = \frac{1}{2}\beta^{1-n} \tag{2.12}$$

für andere monotone Rundungsfunktionen kann

$$\epsilon_x = \beta^{1-n} \tag{2.13}$$

als relativer Fehler der Rundung angegeben werden [53].

## 2.5 IEEE 754 Standard

Durch den, 1985 verabschiedeten, Standard IEEE 754 [37] wird eine einheitliche binäre Gleitkommaarithmetik für Rechner definiert. Der Standard spezifiziert dabei, die zu implementierenden

- Binären Gleitkommaformate
- Rundungsmodi
- Operationen
- Konversionen
- Ausnahmen

für Computerarchitekturen. Im August 2008 wurde mit dem Standard IEEE 754-2008 [38] eine überarbeitete Version verabschiedet, welcher auch dezimale Gleitkommaformate berücksichtigt. Parallel zum IEEE 754 Standard existiert noch der Standard IEEE 854 [20] für basisunabhängige Gleitkommazahlen. In der vorliegenden Arbeit wird jedoch nur der auf verbreiteten Rechnerarchitekturen vorrangig verwendete binäre Standard von 1985 besprochen.

[2] Gleitkommazahl $\tilde{x}$ mit dem geringsten Abstand $|x - \tilde{x}|$ zur reellen Zahl $x$.

### 2.5.1 Binäre Gleitkommaformate

Der IEEE 754 Standard legt für die vier definierten Gleitkommaformate `single`, `single extended`, `double` und `double extended` nur die darstellbare Zahlenmenge fest[3]. Die, bei einer Implementierung, verwendeten Kodierungen in Hard- oder Software lässt der Standard offen.

Die darstellbaren Zahlenformate werden mittels eines Gleitkommasystems gebildet, welche über die Mantissenlänge $n$ sowie den minimalen Exponenten $e_{min}$ und den maximalen Exponenten $e_{max}$, wie in Tabelle 2.1 aufgeführt, spezifiziert werden.

| | `single` | `single ext.` | `double` | `double ext.` |
|---|---|---|---|---|
| $n$ | 24 | $\geq 32$ | 53 | $\geq 64$ |
| $e_{min}$ | $-126$ | $\leq -1022$ | $-1022$ | $\leq -16382$ |
| $e_{max}$ | $+127$ | $\geq +1023$ | $+1023$ | $\geq +16383$ |

Tabelle 2.1: Parameter der IEEE 754 Gleitkommaformate

#### Unendlich und "Not a Number"

Neben den über ein Gleitkommasystem abgebildeten Zahlen ist es aber auch notwendig Werte für "Unendlich" oder "Nicht-Zahlen" vorzusehen. Durch den endlichen Wertebereich der Gleitkommazahlen auf Rechnern werden dann bei Operationen die beiden mathematischen Symbole $+\infty$ und $-\infty$ als entsprechende Werte bei einem Über- oder Unterlauf verwendet. Für jede (echte) Zahl $x$ aus dem Gleitkommasystem gilt folgende Bedingung:

$$-\infty < x < \infty \tag{2.14}$$

Bei unzulässigen Operationen sind als Ergebnisse signalisierende und ruhige NaNs im IEEE 754 Standard definiert. NaN steht hierbei für "Not a Number" und beschreibt einen Wert der nicht für eine Zahl steht. Bei Operationen lösen signalisierende NaNs Ausnahmen aus, während ruhige NaNs ohne Ausnahme weiterverarbeitet werden können. Vergleiche mit NaNs liefern immer `false` zurück und es gilt:

$$\texttt{NaN} \neq \texttt{NaN} \tag{2.15}$$

#### Kodierung

Für die Kodierung der Gleitkommazahlen verwendet der IEEE 754 Standard eine zu Satz 7 leicht modifizierte normalisierte Darstellung mit einer Mantisse im

[3]Wobei für die beiden erweiterten Formate `single extended` sowie `double extended` nur der mindestens darstellbare Bereich spezifiziert ist.

Bereich [1 , 2). Durch die Normalisierung kann jedoch das erste Bit, welches bei einer Binärdarstellung immer den Wert 1 hat, weggelassen werden. Man bezeichnet dies als "Hidden Bit". Der IEEE 754 Standard sieht jedoch auch denormalisierte Zahlen vor. Dies bedeutet, dass auch Mantissen mit führenden Nullen zugelassen sind[4]. Das erste Bit der Mantisse hat in diesem Fall den Wert 0. Speziell reservierte Werte des Exponenten werden hierbei verwendet, um denormalisierte Zahlen aber auch $-\infty$, $+\infty$, NaNs sowie die Zahl 0 zu kodieren.

In den beiden Basis-Formaten `single` und `double` wird die Gleitkommazahl dann aus folgenden drei Komponenten zusammengesetzt:

**Vorzeichen $s$:** Das Vorzeichen $s$ wird als ein Bit gespeichert und steht je nach Wert 0 oder 1 für eine positive oder negative Gleitkommazahl.

**Exponent $e_b$ in Bias-Darstellung:** Der Exponent $e$ muss jede Ganzzahl aus dem Bereich $[e_{min}\,,\, e_{max}]$ enthalten. Zusätzlich muss er noch die Werte $e = e_{min} - 1$ zum kodieren von denormalisierten Zahlen und den Zahlen $\pm 0$[5] sowie $e = e_{max} + 1$ zum kodieren von $\pm\infty$ sowie NaNs abdecken. Der Exponent wird mit Bias $b$ dargestellt. Es wird $e_b = e + b$ gespeichert.

**Mantisse $m$ mit Hidden Bit:** Bei einer normalisierten binären Gleitkommazahl müsste eine Mantisse im Bereich [1 , 2) mit Mantissenlänge $n$ immer einer Kodierung der Form $m = 1.m_1m_2m_3m_4 \ldots m_{n-1}$ mit $m_i \in \{0, 1\}$ entsprechen. Auf die führende 1 kann somit verzichtet werden und die Mantisse wird in der Form $m = .m_1m_2m_3m_4 \ldots m_{n-1}$ mit $m_i \in \{0, 1\}$ gespeichert.

| | `single` | `single ext.` | `double` | `double ext.` |
|---|---|---|---|---|
| $n$ | 24 | $\geq 32$ | 53 | $\geq 64$ |
| $e_{min}$ | $-126$ | $\leq -1022$ | $-1022$ | $\leq -16382$ |
| $e_{max}$ | $+127$ | $\geq +1023$ | $+1023$ | $\geq +16383$ |
| $b$ | $+127$ | $-$ | $+1023$ | $-$ |
| Exponentenbreite in Bits | 8 | $\geq 11$ | 11 | $\geq 15$ |
| Formatbreite in Bits | 32 | $\geq 43$ | 64 | $\geq 79$ |

Tabelle 2.2: Parameter der IEEE 754 Gleitkommaformate

Wie in Tabelle 2.2 zu sehen, sind im IEEE 754 Standard die beiden erweiterten Formate `single extended` und `double extended` nicht genau spezifiziert. Es sind lediglich Mindestanforderungen gegeben, welche bei einer Implementierung erfüllt werden müssen. Für die beiden Basis-Formate sind die Richtlinien

[4]Die eindeutige Darstellung der Gleitkommazahlen geht hier verloren.

[5]Der IEEE 754 Standard sieht sowohl $-0$ als auch $+0$ vor. In einigen Operationen, wie zum Beispiel Division durch 0, werden diese unterschiedlich behandelt. Jedoch gilt beim Vergleich $-0 = +0$.

jedoch vorgegeben. Lediglich die genaue Implementierung und Kodierung, wie beispielhaft für das "single"-Format in Abbildung 2.2 aufgezeigt, bleibt offen.

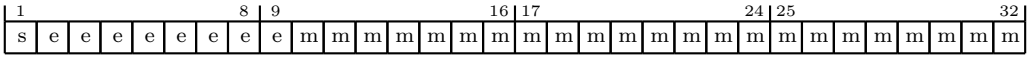



Abbildung 2.2: Format der Gleitkommazahl `single`

Eine dargestellte Zahl $x$ im Format `single` bildet sich hierbei aus folgenden Regeln:

- $e_b = 255 \wedge m \neq 0 \Rightarrow x = \texttt{NaN}$ (unabhängig vom Vorzeichen $s$)
- $e_b = 255 \wedge m = 0 \Rightarrow x = (-1)^s \cdot \infty$
- $0 < e_b < 255 \Rightarrow (-1)^s \cdot 2^{e_b - 127} \cdot (1.m)$
- $e_b = 0 \wedge m \neq 0 \Rightarrow (-1)^s \cdot 2^{-126} \cdot (0.m)$ (denormalisierte Zahlen)
- $e_b = 0 \wedge m = 0 \Rightarrow (-1)^s \cdot 0$

Analog bilden sich die Zahlen des Formats `double`:

- $e_b = 2047 \wedge m \neq 0 \Rightarrow x = \texttt{NaN}$ (unabhängig vom Vorzeichen $s$)
- $e_b = 2047 \wedge m = 0 \Rightarrow x = (-1)^s \cdot \infty$
- $0 < e_b < 2047 \Rightarrow (-1)^s \cdot 2^{e_b - 1023} \cdot (1.m)$
- $e_b = 0 \wedge m \neq 0 \Rightarrow (-1)^s \cdot 2^{-1022} \cdot (0.m)$ (denormalisierte Zahlen)
- $e_b = 0 \wedge m = 0 \Rightarrow (-1)^s \cdot 0$

### 2.5.2 Rundung

Durch die eingeschränkte Genauigkeit der endlichen Gleitkommazahlen definiert der IEEE 754 Standard eine Rundung für Zahlen, die nicht mittels des gewählten Gleitkommaformats darstellbar sind.

Als Standardmodus ist eine Rundung zur nächstgelegenen Gleitkommazahl festgesetzt. Hierbei wird die zu rundende Zahl $x$ unendlich genau angenommen und als Ergebnis die Gleitkommazahl $\tilde{x}$ mit dem geringsten Abstand $|x - \tilde{x}|$ verwendet. Ist der Abstand zu den beiden benachbarten Gleitkommazahlen jedoch identisch, so wird zur nächsten geraden Gleitkommazahl gerundet.

Zusätzlich stellt der IEEE 754 Standard noch drei, vom Benutzer wählbare, gerichtete Rundungsmodi zur Verfügung. Jeweils die Rundung gegen $-\infty$ und $+\infty$ sowie die Rundung gegen 0. Tabelle 2.3 zeigt die vier unterstützten Rundungverfahren und die in dieser Arbeit dafür verwendeten Symbole.

| Bezeichnung | Symbol |
|---|---|
| Rundung zur nächsten Zahl | $\bigcirc$ |
| Rundung in Richtung $-\infty$ | $\bigtriangledown$ |
| Rundung in Richtung $+\infty$ | $\bigtriangleup$ |
| Rundung in Richtung 0 | $\square$ |

Tabelle 2.3: Unterstützte Rundungen

Eine in der Intervallrechnung oft wichtige Eigenschaft gibt der folgende Satz an.

**Satz 8** *Zwischen den gerichteten Rundungen* $\bigtriangledown$ *und* $\bigtriangleup$ *gelten folgende Beziehungen:*

$$\bigtriangledown a = - \bigtriangleup (-a) \tag{2.16}$$

$$\bigtriangleup a = - \bigtriangledown (-a) \tag{2.17}$$

**Beweis:** Siehe [53]. □

### 2.5.3 Operationen

Der IEEE 754 Standard definiert die arithmetischen Operationen Addition, Subtraktion, Multiplikation und Division für die vier verschiedenen Rundungsmodi. Weiter sind noch die Berechnung der Quadratwurzel sowie des Restbetrags bei einer Division mit einer Ganzzahl vorgesehen. Im Gegensatz zur Quadratwurzel muss die Berechnung des Restbetrags ohne Rundung exakt erfolgen.

Für Vergleiche stehen die, aus der Mathematik üblichen, Operationen $<$, $\leq$, $=$, $\neq$, $\geq$ sowie $>$ wie gewohnt zur Verfügung. Zusätzlich ist noch die Vergleichsoperation "unordered" vorgesehen, um nicht anordbare Gleitkommazahlen zu bestimmen. Dies ist bei einem NaN als Operand, welches bei einem Vergleich immer verschieden zum anderen Operanden ist, der Fall.

Konversionen sind zwischen allen unterstützten Gleitkommaformaten gefordert. Wird in eine kleinere Genauigkeit konvertiert, so wird im verwendeten Rundungsmodus eine exakte Rundung auf die entsprechende Gleitkommazahl durchgeführt. Zusätzlich sind Konversionen mit Ganzzahlen oder Zahlen im Dezimalformat definiert.

### 2.5.4 Ausnahmen

Der IEEE 754 Standard definiert fünf verschiedene Kategorien von Ausnahmen:

**Überlauf** tritt auf, wenn die im verwendeten Gleitkommaformat größtmögliche endliche Zahl überschritten wird. Das Ergebnis der Operation ist, je nach Rundungsmodus, $+\infty$ oder die größte endliche Zahl.

**Unterlauf** tritt auf, wenn die im verwendeten Gleitkommaformat kleinstmögliche endliche Zahl unterschritten wird. Das Ergebnis der Operation ist, je nach Rundungsmodus, $-\infty$ oder die kleinste endliche Zahl.

**Division durch Null** wird bei einer Division einer endlichen Zahl $x \neq 0$ durch 0 verwendet. Das Ergebnis der Operation ist in Abhängigkeit der Vorzeichen von Dividend und Divisor entweder $-\infty$ oder $+\infty$[6].

**Ungültige Operation** kennzeichnet eine Operation die mit ungültigen Operanden ausgeführt wird. Das Ergebnis der Operation ist NaN.

**Ungenau** tritt auf wenn eine Zahl nicht durch das Gleitkommaformat dargestellt werden kann und gerundet wird.

Für jede Ausnahme ist ein eigenes Flag im Standard vorgesehen, welches vom Benutzer gelesen und geschrieben werden kann. Bei den Flags handelt es sich um sogenannte "Sticky-Flags", sind sie einmal gesetzt bleiben sie bis zum Zurücksetzen erhalten.

Neben den Statusflags sieht der Standard auch Traphandler vor, welche anstatt der Flags verwendet werden können. Tritt eine Ausnahme auf, so wird dann nicht das Flag gesetzt, sondern der aktivierte Traphandler aufgerufen. Der Traphandler ist dann für die weitere Fehlerbehandlung zuständig.

## 2.6 Intervallarithmetik mit Gleitkommazahlen

Auf Rechnern lässt sich die Intervallarithmetik mittels Gleitkommazahlen realisieren. Gleitkommaintervalle werden analog zu den reellwertigen Intervallen aus Definition 1 oder den erweiterten reellwertigen Intervallen aus Definition 7 verwendet. Lediglich bei den Grenzen $a_1$ und $a_2$ eines Intervalls $[a_1\,,\,a_2]$ handelt es sich um (endliche) Gleitkommazahlen. Das Intervall selbst repräsentiert jedoch eine kontinuierliche Teilmenge der reellen oder erweiterten reellen Zahlen.

Um die Einschließungseigenschaft der Intervalle auch für Gleitkommaintervalle sicherzustellen müssen die Rechenoperationen aus Folgerung 1 mit gerichteten Rundungen wie folgt ausgeführt werden:

$$A + B = [a_1 \mathbin{\overset{\nabla}{+}} b_1\,,\; a_2 \mathbin{\overset{\Delta}{+}} b_2] \tag{2.18}$$

$$A - B = [a_1 \mathbin{\overset{\nabla}{-}} b_2\,,\; a_2 \mathbin{\overset{\Delta}{-}} b_1] \tag{2.19}$$

$$\begin{aligned} A \cdot B = \; & [\min\{a_1 \mathbin{\overset{\nabla}{\cdot}} b_1, a_1 \mathbin{\overset{\nabla}{\cdot}} b_2, a_2 \mathbin{\overset{\nabla}{\cdot}} b_1, a_2 \mathbin{\overset{\nabla}{\cdot}} b_2\}, \\ & \max\{a_1 \mathbin{\overset{\Delta}{\cdot}} b_1, a_1 \mathbin{\overset{\Delta}{\cdot}} b_2, a_2 \mathbin{\overset{\Delta}{\cdot}} b_1, a_2 \mathbin{\overset{\Delta}{\cdot}} b_2\}] \end{aligned} \tag{2.20}$$

$$A \div B = [a_1\,,\; a_2] \cdot [\frac{1}{b_2}\,,\; \frac{1}{b_1}] \quad \text{mit} \quad 0 \notin B \tag{2.21}$$

[6] Beispielsweise gilt: $1/-0 = -\infty$

Durch die Rundung der Intervallgrenzen jeweils nach Außen wird sichergestellt, dass der tatsächliche Wert der jeweiligen Grenze im Intervall enthalten ist. Allgemein muss für jede Operation auf Gleitkommaintervallen die Berechnung mit gerichteten Rundungen $\bigtriangledown$ und $\bigtriangleup$ ausgeführt werden, um die Einschließungseigenschaft zu erhalten. Die beiden Hauptsätze, Satz 3 über die Einschließungseigenschaft sowie Satz 4 über die Teilmengeneigenschaft, sind somit auch für Gleitkommaintervalle gültig.

# Teil II

# Intervallarithmetik auf Rechnern

# Kapitel 3

# Bestehende Systeme

Bevor in dieser Arbeit Anforderungen und Implementierungsmöglichkeiten der Intervallarithmetik auf Rechnern diskutiert werden, ist es natürlich von Interesse, wie andere existierende C++ Implementierungen diese Anforderungen lösen. Hierzu werden entscheidende Implementierungsdetails der beiden C++ Bibliotheken Boost Interval Arithmetic Library und filib++ untersucht. Vorerst wird jedoch ein Vorschlag für Intervallarithmetik in der C++ Standardbibliothek besprochen.

Hauptaugenmerk haben hierbei die jeweiligen Konzepte der Implementierung, die verfügbaren Berechnungsmodelle und Rundungsstrategien sowie die gebotenen Möglichkeiten bei der Anwendung. Eine Diskussion der einzelnen Methoden und Schnittstellen soll hier nicht stattfinden. In diesem Punkt erfüllen alle drei Lösungen die an eine Intervallbibliothek gestellten Anforderungen auf ähnliche Weise.

## 3.1 Vorschlag für Intervallarithmetik in der C++ Standardbibliothek

Aktuell liegt der Vorschlag von Sylvain Pion, Hervé Brönnimann und Guillaume Melquiond, Intervallarithmetik in die C++ Standardbibliothek aufzunehmen, in einer zweiten überarbeiteten Version, vor [41]. Bei den drei Autoren handelt es sich um Entwickler aus dem Intervallarithmetikprojekt [6] der Boost Bibliothek [5]. Die Arbeit an der Boost Intervallarithmetik Bibliothek war auch der ausschlaggebende Punkt für den oben genannten Vorschlag, jedoch weicht er in vielen Designentscheidungen von der Boost-Implementierung ab.

Hauptziel ist es die Intervallarithmetik im C++ Standard als reine Erweiterung der Standardbibliothek zu integrieren, ohne dabei eine Änderung am Sprachstandard zu benötigen. Eine mit aktuellen C++ Compilern lauffähige Reverenzimplementierung [39] steht ebenfalls schon zur Verfügung.

### 3.1.1 Datentyp für Intervalle

Als Datentyp für Intervalle sieht der Vorschlag eine generische Klasse `interval<T>` vor. Über den Templateparameter `T` kann dann der für die Unter- sowie Obergrenze verwendete Datentyp ausgeprägt werden. Jedoch wird der gewählte Datentyp für den Parameter `T`, genau wie bei `std::complex<T>` für komplexe Zahlen in der C++ Standardbibliothek [48], auf die drei C++ Gleitkommazahlen

- `float`
- `double`
- `long double`

über Spezialisierungen [49] der Templateklasse `interval<T>` eingeschränkt.

Die interne Repräsentation der Intervalle wird für `interval<T>` nicht weiter festgelegt. Jedoch legt die Wahl der Schnittstellen eine Darstellung über Unter- und Obergrenze nahe. So existieren nur Konstruktoren mit einem Parameter für die Erzeugung von Punktintervallen sowie mit zwei Parametern für die Erzeugung über Unter- und Obergrenze, was gegen eine Impelementierung in Mittelpunkt-Radius Darstellung spricht.

Weiter wird über die Genauigkeit der Intervalle nichts gesagt. Es wird lediglich gefordert, dass die Einschließungseigenschaft aus Satz 3 zu jeder Zeit erfüllt wird. Auch die Menge der darstellbaren Intervalle bleibt offen. Lediglich die leere Menge $\emptyset$, die Menge der reellen Zahlen $\mathbb{R}$ sowie die Punktintervalle $[x\,,\,x]$ für Gleitkommazahlen $x$ aus der Menge der über `T` darstellbaren Zahlen, wird von einer Implementierung vorausgesetzt.

Durch die Forderung, dass die Menge der reellen Zahlen abgebildet werden kann, sind bei der Konstruktion eines Intervalls natürlich $-\infty$ als Unter- sowie $+\infty$ als Obergrenze zulässig. allgemein muss die Invariante

$$\text{Untergrenze} \leq \text{Obergrenze} \tag{3.1}$$

bei nichtleeren Intervallen des Typs `interval<T>` stets erfüllt sein. Aus diesem Grund ist auf die Grenzen auch nur lesender Zugriff über die beiden Methoden `lower()` und `upper()` vorgesehen, um nicht durch Verändern der Grenzwerte die Invariante (3.1) zu verletzen.

#### Leeres Intervall

Auch die genaue Darstellung der leeren Menge als leeres Intervall wird in diesem Vorschlag der jeweiligen Implementierung offen gelassen. Jedoch wird auf die zwei Möglichkeiten

- $\emptyset = [\texttt{NaN}\,,\,\texttt{NaN}]$
- $\emptyset = [a\,,\,b]$ mit $a > b$

für eine Implementierung verwiesen.

### 3.1.2 Operationen

Die arithmetischen Operationen Addition, Subtraktion, Multiplikation sowie Division stehen, wie in Kapitel 1.5.3 zur Auswertung von erweiterten Intervallen definiert, zur Verfügung. Es wird jedoch nur gefordert, dass das Ergebnisintervall einer Operation $X \circ Y$ für zwei Intervalle $X$ und $Y$ mit $\circ \in \{+, -, \cdot, \div\}$ die jeweilige Menge

- $\{x + y \mid x \in X, y \in Y\}$ für die Addition
- $\{x - y \mid x \in X, y \in Y\}$ für die Subtraktion
- $\{x \cdot y \mid x \in X, y \in Y\}$ für die Multiplikation
- $\{x \div y \mid x \in X, y \in Y, y \neq 0\}$ für die Division

einschließt. Die Genauigkeit einer Operation ist der Implementierung überlassen. Hier muss bei der Implementierung zwischen benötigter Laufzeit und Qualität der Ergebnisse abgewägt werden. Gleiches gilt für die Liste der geforderten mathematischen Funktionen, wie zum Beispiel Sinus, Kosinus, Quadratwurzel oder Exponentialfunktion. Hier wird ebenfalls eine lose Funktionsauswertung über den Definitionsbereich der Funktion ausgeführt und das Ergebnisintervall muss die Wertemenge lediglich beinhalten. Einzig entscheident ist nur die Erfüllung der Einschließungseigenschaft. Außnahmen bilden Operationen und Funktionen, die mit einem leeren Intervall ausgeführt werden. Hier muss als Ergebnis stets ein leeres Intervall zurückgegeben werden.

Natürlich sind die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division über Operatorüberladung definiert. Der Benutzer kann diese vier Operationen auf die in der Gleitkommaarithmetik gewohnte Weise verwenden.

#### Partiell definierte Funktionen und Unstetigkeit

Wie im Kapitel 1.5.3 angesprochen, sind für einige Anwendungsfälle stetige Funktionen vorausgesetzt. Eine lose Funtkionsauswertung ist jedoch auch auf partiell definierten Funktionen gültig. Um in diesen Anwendungsfällen Information über die Auswertung von unstetigen oder partiell definierten Funktionen zu haben, wird für jede partiell definierte Funktion und für die Division eine zusätzliche

Implementierung gefordert. Diese Implementierung hat als zusätzlichen Parameter einen boolschen Wert, welcher als Referenz übergeben wird. Wird die Funktion nun über einen nicht definierten Bereich ausgewertet oder wird eine Division durch Null durchgeführt, so wird der Wert auf `true` gesetzt. Im anderen Fall bleibt der Wert dieses Parameters unverändert. Auf diese Weise kann eine boolsche Variable für die Auswertung eines Ausdrucks aus mehreren Funktionen verwendet werden und der Benutzer hat am Ende der Auswertung die Information über den kompletten Ausdruck. Listing 3.1 zeigt die Verwendung dieses Flags für den Ausdruck $x/\sqrt{y}$.

```
1 std::interval<double> x(1.0), y(-1.0, 1.0);
2 bool f = false;
3
4 std::interval<double> r = std::divide(x,
5         std::sqrt(y, f), f);
6
7 if (!f)
8         ...
```

Listing 3.1: Verwendung eines Flags bei der Auswertung von $x/\sqrt{y}$

**"Interval mathematical relations"**

Zusätzlich, zu den schon erwähnten Funktionen und Operationen, sind einige Funktionen im sogenannten "Backward" oder "Reverse" Verfahren [34] definiert. Der C++ Vorschlag nennt dies "interval mathematical relations".

Bei "Backward" oder "Reverse" Operationen wird die Operation um ein weiteres Intervall als Parameter ergänzt, für welches die Umkehrfunktion der Operation ausgeführt wird. Das Ergebnis der Umkehrfunktion wird dann mit dem Intervall des eigentlichen Operanden geschnitten. Die Funktion

- `interval<T> asin_rel(interval<T> X, interval<T> R)`

liefert beispielsweise als einschließendes Intervall die Menge

$$\{r \in R \mid sin(r) \in X\} \tag{3.2}$$

zurück. Analog werden die anderen definierten "Backward" oder "Reverse" Operationen ausgewertet. Lediglich die Division

- `interval<T> div_rel(interval<T> X, interval<T> Y)`

fällt mit ihrer Definition

$$\{r \in \mathbb{R} \mid \exists y \in Y, r \cdot y \in X\} \tag{3.3}$$

etwas aus der Reihe.

#### Rundung

Wie in Kapitel 2.6 schon angesprochen, ist es bei der Gleitkommaintervallarithmetik wichtig, die Berechnung der Intervallgrenzen mit gerichteten Rundungen gegen $-\infty$ und $+\infty$ auszuführen. Dies stellt die, für die Intervallarithmetik essentielle, Einschließungseigenschaft aus Satz 3 sicher.

Auf vielen verbreiteten Rechnerarchitekturen ist der Rundungsmodus jedoch nur global zu setzen und dementsprechend für alle Operationen gültig [2, 22]. Für verschiedene Rundungen ist somit ein oft laufzeitintensiver Wechsel des Rundungsmodus nötig. Aus diesem Grund legen einige Implementierungen der Intervallarithmetik die Kontrolle des Rundungsmodus für Optimierungszwecke teilweise oder ganz in die Hände der Benutzer. Dies resultiert zwar oft in besserer Laufzeit jedoch birgt es auch die Gefahr von fehlerhaften Berechnungen und Programmverhalten auf Grund eines falsch gesetzten Rundungsmodus.

Der Vorschlag spricht sich, aus den genannten Gründen, somit gegen eine benutzerseitige Kontrolle über die verwendete Rundung aus. Wechsel des Rundungsmodus müssen hier vom Benutzer gekapselt behandelt werden und dürfen niemals Programmcode außerhalb von Intervalloperationen betreffen.

### 3.1.3 Vergleichsoperationen

Aus den in Kapitel 1.2.3 angesprochenen Problemen bei Vergleichen von Intervallen, definiert der Vorschlag vier verschiedene Modelle von Vergleichsoperationen. Jedes der Modelle verwendet hierzu die Überladung der Operatoren `==`, `!=`, `<`, `<=`, `>` sowie `>=`, welche jeweils in einem eigenen Namensraum implementiert werden.

Ein Benutzer kann sich somit den, für den Anwendungsfall, gewünschten Vergleichsmodus, wie in Listing 3.2 gezeigt, in den jeweiligen Gültigkeitsbereich über die `using` Direktive einbinden und die überladenen Operatoren verwenden.

```
1 using namespace std::certainly_ops;
2
3 std::interval<double> x(1.0), y(-1.0, 1.0);
4
5 if (x < y)
6         ...
```

Listing 3.2: `using` Direktive für Vergleichsoperationen

#### Teilmengenrelationen

Der erste Vergleichsmodus ist die Verwendung der Teilmengenrelationen über den Namensraum `std::set_inclusion_ops`.

Die einzelnen Relationen werden dann, gemäß den aus Tabelle 3.1 entsprechenden Regeln, als Operatoren umgesetzt.

| Operator | Teilmengenrelation |
|---|---|
| == | $=$ |
| != | $\neq$ |
| < | $\subset$ |
| <= | $\subseteq$ |
| > | $\supset$ |
| >= | $\supseteq$ |

Tabelle 3.1: Zuordnung von Operatoren zu Teilmengenrelationen

#### Boolset Vergleiche

Um dem Problem mit Vergleichsoperationen bei Intervallen entgegenzuwirken, führen die Autoren einen Datentyp `bool_set` [42] ein, welcher als Ergebnis von Vergleichen über den Namensraum `std::bool_set_ops` verwendet wird. Ein `bool_set` repräsentiert hierbei vier verschiedene Werte: `empty`, `true`, `false` und `indeterminate`. Bei einem Vergleich von zwei Intervallen werden dann die entsprechenden Werte

`true`: Der Vergleich ist für jedes Wertepaar der beiden Intervalle gültig.

`false`: Der Vergleich ist für kein Wertepaar der beiden Intervalle gültig.

`empty`: Mindestens eines der beiden Intervalle ist die leere Menge.

`indeterminate`: Keiner der anderen drei Fälle trifft zu.

zurückgeliefert. Bei `indeterminate` ist der Vergleich somit teilweise richtig und auch teilweise falsch. Hier kann der Benutzer dann dem Anwendungsfall entsprechend entscheiden.

#### "certainly" und "possibly" Vergleiche

Über die Namensräume `std::certainly_ops` und `std::possibly_ops` stehen die beiden im Kapitel 1.2.3 definierten Vergleichsmodelle "certainly" und "possibly" zur Verfügung. Die beiden Modelle könnten auch, indem man die vier Werte eines `bool_set` entsprechend auf `true` und `false` abbildet, durch `bool_set` Vergleiche simuliert werden. Jedoch werden "certainly" und "possibly" Vergleiche in vielen Anwendungen verwendet und sind dementsprechend als eigene Modelle definiert.

## 3.2 Boost Interval Arithmetic Library

Die weit verbreitete C++ Bibliothekssammlung Boost [5] beinhaltet auch eine als Unterbibliothek implementierte Intervallarithmetik. Die Boost Interval Arith-

metic Library [6] stellt hierbei eine sehr vielseitige und komfortable Bibliothek dar, welche durch generische Datentypen und das Konzept der Policy Klassen [4] an unterschiedliche Anforderungen angepasst werden kann.

### 3.2.1 Policy Klassen und Datentyp für Intervalle

Das Konzept der Boost Implementierung ist es, das Verhalten der Intervallarithmetik über sogenannte Policy Klassen auszuprägen. Die Policy Klasse beeinflusst hierbei nicht die Datenrepräsentation des eigentlichen Intervalls über Unter- und Obergrenze. Es werden lediglich einige verwendete Algorithmen durch die Policy Klasse beschrieben [43]. Intervalle können dann über den Datentyp `interval<T, Policies>` erzeugt werden.

Der Templateparameter `T` gibt hierbei den für die Unter- und Obergrenze verwendeten Datentyp an. Die Boost Impelementierung unterstützt standardmässig die drei C++ Gleitkommazahlen:

- `float`
- `double`
- `long double`

Jedoch ist es durch das offene Design der Bibliothek mittels Policy Klassen auch möglich andere Datentypen als Grenzwerte zu verwenden, sie müssen lediglich bestimmte Eigenschaften erfüllen [6]. Hierzu ist es aber nötig eine entsprechende Policy Klasse für diesen Datentyp bereitzustellen. Dies kann entweder durch spezialisieren der vorhandenen oder durch implementieren von neuen Policies realisiert werden.

Die über den Templateparameter `Policies` für die Ausprägung der Klasse `interval<T, Policies>` verwendete Policy muss dann dem Typ `policies<Rounding, Checking>` entsprechen. Der Typ `policies<Rounding, Checking>` enthält wiederum nur Referenzen auf die beiden Klassen `Rounding` und `Checking`. Das Verhalten der Policy Klasse wird also in zwei einzelne Policies aufgeteilt [40]. Diese beiden als Templateparamter `Rounding` und `Checking` übergebenen Policies implementieren dann aber wichtiges Verhalten für die Intervallrechnung. So sind durch das Policy Konzept viele Algorithmen nur als Grundgerüst implementiert und bei der Auswertung dieser Algorithmen wird auf Routinen der verwendeten Policies zurückgegriffen. Es entspricht somit dem aus der objektorientierten Programmierung bekannten Strategie Muster [15]. Jedoch sind die Policy Klassen zur Übersetzungszeit bekannt und können durch den Compiler effizient eingebunden werden.

Die Klasse `interval<T, Policies>` verwendet für den Templateparameter jedoch eine Voreinstellung. Man kann Instanzen somit nur durch Angabe des Typs `T` erzeugen. Für `Policies` wird dann die Standard Policy verwendet.

### Rounding Policy

Die als Parameter `Rounding` übergebene Rounding Policy ist in der Boost Intervall Bibliothek für das Berechnen von korrekt gerundeten Werten für die Untersowie Obergrenze vom Typ `T` zuständig. Dies wird erreicht, indem jede Intervalloperation von einem Objekt der Rounding Policy gekapselt wird. Die Rounding Policy übernimmt hierbei drei Aufgaben:

1. Zu Beginn der Operation den aktuellen Rundungsmodus sichern.
2. Berechnungen korrekt gerundet ausführen.
3. Am Ende der Operation den gespeicherten Rundungsmodus wieder herstellen.

Punkt 1 und 3 übernimmt der Konstruktor und Destruktor der Rounding Policy. Bei der Instanzierung wird der aktuelle Rundungsmodus für ein späteres Rücksetzen gespeichert. Beim Vernichten des Objekts wird dann der alte Zustand über den Destruktor wieder hergestellt.

Die gerundeten Berechnungen der Grenzen werden dann über entsprechende Operationen und Funktionen der Rounding Policy durchgeführt. Für die Addition existieren beispielsweise die zwei Funktionen

- `T add_down(T, T)`
- `T add_up(T, T)`

um für den gewählten Datentyp der Intervallgrenzen eine Addition von zwei Werten mit gerichteter Rundung gegen $-\infty$ oder $+\infty$ durchzuführen. Der eigentliche Operator für die Addition von zwei Intervallen verwendet dann für die Berechnung der Grenzen die beiden Methoden der erzeugten Rounding Policy. Somit ist es möglich den allgemeinen Algorithmus für die Addition mit spezifischem Verhalten für die Berechnung der Grenzen auszuprägen und mit verschiedenen Strategien für die Rundung zu versehen.

Am Ende der Operation wird die erzeugte Rounding Policy wieder vernichtet und der ursprüngliche Zustand für die Rundung wieder hergestellt. Programmcode außerhalb von Intervalloperationen ist somit nicht von Rundungswechseln betroffen.

Die Boost Intervall Bibliothek liefert standardmässig Rounding Policy Klassen mit verschiedenen Rundungsstrategien für die drei Gleitkommatypen `float`, `double` und `long double`. Die Standardstrategie ist ein Wechsel des Rundungsmodus in die, jeweils durch die Operation der Policy Klasse, benötigte Rundung. Die Methode `T add_down(T, T)` setzt demnach die Rundung gegen $-\infty$, die Methode `T add_up(T, T)` analog in Richtung $+\infty$.

Eine weitere, optimierte Strategie ist die Anwendung der Rechenregel $\bigtriangledown a = - \bigtriangleup (-a)$ aus Satz 8 zum Einsparen von laufzeitintensiven Rundungswechseln. Hier wird die Berechnung nur mit einer gerichteten Rundung gegen $+\infty$ durchgeführt.

Zusätzlich existiert noch eine Policy, die komplett auf Rundung verzichtet und dementsprechend für die drei Gleitkommatypen die Einschließungseigenschaft nicht erfüllt.

Für andere Datentypen als `float`, `double` und `long double` oder für abweichende Rundungsstrategien kann der Benutzer eigene Rounding Policies, welche die benötigten Schnittstellen erfüllen, verwenden.

### Checking Policy

Über die als Templateparameter `Checking` angegebene Checking Policy kann, analog zur Rounding Policy, verschiedenes Verhalten für die Intervallrechnung ausgeprägt werden. Im Gegensatz zur Rounding Policy ist sie aber nicht für die eigentliche Berechnung zuständig. Ihre Aufgabe ist es für den Typ `T` bei speziellen Werten eine entsprechende Behandlung zu realisieren.

So sind beispielsweise die Intervalle $(-\infty\,,\,+\infty)$, $(-\infty\,,\,a]$, $[a\,,\,+\infty)$ in der Intervallrechnung nicht immer gewünscht und müssen für bestimmte Anwendungen durch geeignete Ausnahmen im Programm behandelt werden. Aber auch wenn von der Anwenderseite nichts gegen Intervalle mit unendlichen Grenzen spricht, können die Werte $-\infty$ und $+\infty$ aber auch NaNs Probleme bereiten. Diese drei Werte stehen zwar über den IEEE 754 Standard für Gleitkommazahlen zur Verfügung, jedoch gibt es Systeme die den Standard nicht oder nur teilweise erfüllen[1].

Die Checking Policy muss somit für anwendungsbedingte aber auch für systembedingte Sonderfälle eine entsprechende Behandlung realisieren. Ähnlich wie `std::numeric_limits<T>` [48] stellt die Checking Policy Methoden zum Zugriff auf bestimmte Werte bereit.

- `static T pos_inf()` : Wert für $+\infty$
- `static T neg_inf()` : Wert für $-\infty$
- `static T nan()` : Wert für `NaN`
- `static T empty_lower()` : Wert der unteren Grenze eines leeren Intervalls
- `static T empty_upper()` : Wert der oberen Grenze eines leeren Intervalls

Zum Erzeugen der benötigten Werte wird dann die entsprechende Methode der Checking Policy verwendet. In Abhängigkeit von System und Anwendungsfall

[1] Der von IBM, Sony und Toshiba gemeinsam entwickelte Cell Prozesseor unterstützt beispielsweise für den Datentyp `single` weder NaNs noch $\pm\infty$ [12].

kann die verwendete Policy dann Werte wie $+\infty$, $-\infty$ oder `NaN` zurückliefern oder es wird eine Ausnahme ausgelöst.

Über die beiden Methoden `empty_lower()` und `empty_upper()` können unterschiedliche Darstellungen für leere Intervalle realisiert werden. Die Werte der Grenzen bei leeren Intervallen sind in der Boost Implementierung nicht auf eine bestimmte Darstellung beschränkt. Sie könnten je nach Datentyp `T` und verwendeter Rechnerarchitektur zum Beispiel als [`NaN` , `NaN`] oder [1 , 0] erzeugt werden. Durch das offene Design sind aber natürlich auch Methoden zum Überprüfen auf leere Intervalle oder auf NaNs implementiert. Dies ist notwendig, um bei Methoden Intervalle und Parameter des Typs `T` auf ihre Werte zu prüfen. So wird zum Beispiel eine Addition eines Intervalls mit einem `NaN` verhindert.

Standardmässig liefert die Boost Intervall Bibliothek verschiedene Implementierungen von Checking Policies für die Gleitkommatypen `float`, `double` und `long double`. Für andere Datentypen oder für spezielle Anwendungsfälle können eigene Checking Policies verwendet werden. So kann beispielsweise bei Anwendungen, die nie ein leeres Intervall erzeugen, die Überprüfung auf leere Intervalle in der Methode `bool is_empty(const T&, const T&)` durch einen leeren Methodenrumpf[2] ersetzt werden.

### 3.2.2 Operationen

Operationen und Funktionen sind in der Boost Intervall Bibliothek, wie für erweiterte Intervalle in Kapitel 2.6 definiert, umgesetzt. Und auch Funktionsauswertungen werden, wie gewohnt, über lose Funktionsauswertung realisiert. Als Ergebnis werden immer die kleinst möglichen einschließenden Intervalle zurückgeliefert [43]. Operationen und Funktionen auf leeren Intervallen liefern aber auch stets leere Intervalle zurück.

"Backward" oder "Reverse" Operationen [34] oder auch eine spezielle Behandlung von unstetigen oder partiell definierten Funktionen gibt es keine. Es wird immer ein Einschluß durch lose Funktionsauswertung zurückgeliefert, ohne dabei irgendeine Information an den Anwender weiterzureichen. Ein Flag, wie etwa bei dem in Kapitel 3.1 besprochenen Vorschlag für Intervallarithmetik in der C++ Standardbibliothek, existiert nicht. Lediglich die Division mit Null findet eine besondere Beachtung.

Wird der normale Operator `/` für die Division mit Null verwendet, so wird hier das Ergebnis von der Art des Divisors abhängig gemacht. Entspricht der Divisor dem Punktintervall [0 , 0], so ist das Ergebnis der Division das leere Intervall. Enthält der Divisor neben der Zahl 0 noch weitere Zahlen so wird das kleinste Intervall zurückgeliefert, welches alle paarweisen Ergebnisse der Division ohne der

[2]Durch die Bekanntgabe von Policy Klassen zur Übersetzungszeit können leere Methodenrümpfe durch den Compiler optimiert werden.

Null enthält [6].

Zusätzlich enthält die Boost Bibliothek aber noch zwei weitere Methoden für die Division mit Null.

- `interval<T, Policies> division_part1(const interval<T, Policies>& x, const interval<T, Policies>& y, bool& b)`
- `interval<T, Policies> division_part2(const interval<T, Policies>& x, const interval<T, Policies>& y, bool b = true);`

Ist das Ergebnis einer Division durch ein Intervall darstellbar, so liefert die Methode `division_part1` dieses Intervall als Ergebnis zurück und setzt den Wert, der als Referenz übergebenen, boolschen Variable `b` auf den Wert `false`. Ist das Ergebnis die Vereinigung zweier disjunkter Intervalle, wie etwa bei $[1\,,\,1]/[-1\,,\,1] = (-\infty\,,\,-1] \cup [1\,,\,+\infty)$, so liefert `division_part1` das erste (negative) Intervall und setzt `b` auf den Wert `true`. Im diesem Fall muss dann das zweite (positive) Intervall über die Methode `division_part2` berechnet werden.

### 3.2.3 Ungeschützte Rundung

Wie bei den Rounding Policies beschrieben, werden die Wechsel des Rundungsmodus bei Intervalloperation für den Benutzer transparent durchgeführt. Es wird nie Programmcode außerhalb der Intervalloperation von diesen Rundungen betroffen. Dies bedeutet aber auch für ein Konzept mit Operatorüberladung, dass zu Beginn des Operators die aktuelle Rundung gesichert, die Operation mit den benötigten Rundungswechseln durchgeführt und vor dem Verlassen des Operators die Rundung wieder auf den alten Zustand zurückgesetzt wird. Werden nun aber mehrere Intervalloperationen hintereinander ausgeführt, so wird zwischen den Operationen immer unnötig in den alten Rundungsmodus zurückgewechselt. Dies hat für das Programm große Laufzeiteinbussen zur Folge.

Für diesen Zweck bietet die Boost Intervall Bibliothek einen "unprotected" Modus für die Berechnung an. Hierzu müssen Intervalle mit einer speziellen "unprotected Rounding Policy" erzeugt werden und der Benutzer muss den Rundungsmodus für die Berechnungen, durch Erzeugen eines entsprechenden Rundungsobjekts, initialisieren [6]. Die Berechnungen an sich werden dann ohne das unnötige Rücksetzen der Rundungsmodi durchgeführt. Am Ende der Berechnung wird dann, durch Vernichten des erstellten Rundungsobjekts, der alte Zustand wieder hergestellt.

Der Programmcode aus Listing 3.3 zeigt den groben Ablauf im "unprotected" Modus. Der Typ `I` bezeichnet hierbei einen Intervalldatentyp mit einer "gesicherten" Rundungsstrategie. Über `I::traits_type::rounding` kann dann die aktuelle Rundung gesichert und der Rundungsmodus für die Intervallrechnung initialisiert werden. Den zu `I` kompatiblen Intervalldatentyp mit einer "unprotected Rounding Policy" liefert dann

`boost::numeric::interval_lib::unprotect<I>::type`. Nach der Berechnung mit "ungesicherten" Intervallen wird durch Vernichten des Rundungsobjekts `rnd` wieder der alte Rundungszustand hergestellt.

```
1  {
2      // Aktuelle Rundung sichern und für
3      // Intervallberechnung initialisieren
4      typename I::traits_type::rounding rnd;
5
6      // Intervall des Typs I als unprotected definieren
7      typedef typename boost::numeric::interval_lib
8          ::unprotect<I>::type R;
9
10     // Intervalle des ungeschuetzten Typs R erzeugen
11     const R& a = ...
12
13     // Berechnungen durchführen
14     ...
15
16     // Durch die Vernichtung von rnd den alten
17     // Rundungsmodus wieder herstellen
18 }
```

Listing 3.3: Struktur bei Verwendung des "unprotected" Modus

Wichtig ist hierbei, dass die aufgezeigte Struktur eingehalten wird und keine anderen Gleitkommaberechnungen innerhalb dieses Blocks ausgeführt werden. Fehlerhafte Berechnungen wären sonst die Folge.

### 3.2.4 Vergleichsoperationen

Die Vergleichsrelationen werden in der Boost Implementierung ebenfalls über die Operatoren `==`, `!=`, `<`, `<=`, `>` sowie `>=` realisiert. Der Standard für eine Vergleichsoperation $\circ$ zwischen zwei Intervallen $A$ und $B$ entspricht hierbei folgenden Regeln:

- $\forall a \in A, \forall b \in B$ ist die Bedingung $a \circ b$ erfüllt, so ist das Ergebnis des Vergleichs `true`.
- $\forall a \in A, \forall b \in B$ ist die Bedingung $a \circ b$ nicht erfüllt, so ist das Ergebnis des Vergleichs `false`.
- Es wird eine Ausnahme ausgelöst, wenn keine der beiden Fälle eintritt oder es sich bei $A$ oder $B$ um ein ungültiges oder leeres Intervall handelt.

Analog zum im Kapitel 3.1 besprochenen Vorschlag für Intervallarithmetik in der C++ Standardbibliothek können aber andere Vergleichsmodelle über eigene Namensräume eingebunden werden. Die Teilmengenrelationen sowie die "certainly" und "possibly" Vergleiche entsprechen hierbei den, in Kapitel 3.1.3, vorgestellten Modellen und werden durch die Namensräume

- `boost::numeric::interval_lib::compare::set`
- `boost::numeric::interval_lib::compare::certain`
- `boost::numeric::interval_lib::compare::possible`

implementiert. Weiter stehen noch lexikographische Vergleiche, ein zu den Bool-Set Vergleichen ähnliches Modell mit 3 Zuständen, sowie explizite Vergleichsfunktionen zur Verfügung.

### Lexikographische Vergleiche

Bei lexikographischen Vergleichen werden zuerst die unteren Grenzen der beiden Intervalle verglichen. Reicht dies schon für eine Beurteilung der Vergleichsfunktion aus, so wird der entsprechende boolsche Wert zurückgeliefert. Ist dies nicht der Fall, so werden zusätzlich noch die oberen Grenzen in den Vergleich mit einbezogen. Für lexikographischen Vergleiche wird der Namensraum `boost::numeric::interval_lib::compare::lexicographic` verwendet.

### Tribool Vergleiche

Ähnlich wie beim Datentyp `bool_set` [42] werden durch den Datentyp `tribool` [7] aus der Boost Bibliothek neben den boolschen Werten `true` und `false` noch unbestimmbare Werte als `indeterminate` abgebildet. Der vierte Wert `empty` des Datentyps `bool_set` existiert hier nicht.

Tribool Vergleiche entsprechen dann den Regeln der Standardvergleichsoperatoren, lediglich die Ausnahmefälle werden auf den Wert `indeterminate` abgebildet. Ungültige Intervalle erzeugen bei Tribool Vergleichen aber dennoch eine Ausnahme. Für Tribool Vergleiche wird der Namensraum `boost::numeric::interval_lib::compare::tribool` verwendet.

### Explizite Vergleichsfunktionen

Über die Header Datei `boost/numeric/interval/compare/explicit.hpp` können zusätzlich noch explizite Vergleichsfunktionen in den Namensraum `boost::numeric::interval_lib` eingebunden werden. Hier stehen dann die jeweiligen "certainly" und "possibly" Vergleiche durch explizite Funktionsnamen bereit. Die verwendeten Funktionsnamen setzen sich aus `cer` für "certainly" oder `pos` für "possibly" sowie den Bezeichnern aus Tabelle 3.2 zusammen.

Die Funktion

- `bool cerlt(const interval<T, Policies1>& x, const interval<T, Policies2>& y)`

würde beispielsweise den Operator `<` aus dem Namensraum für "certainly" Vergleiche entsprechen.

| Vergleichsoperator | Bezeichner |
|---|---|
| == | eq |
| != | ne |
| < | lt |
| <= | le |
| > | gt |
| >= | ge |

Tabelle 3.2: Bezeichner für explizite Vergleichsfunktionen

## 3.3 filib++

Die Intervallbibliothek filib++ (fast interval library) [13] ist ein Gemeinschaftsprojekt des Lehrstuhls für Programmiersprachen und Programmiermethodik der Universität Würzburg und dem Institut für wissenschaftliches Rechnen der Universität Wuppertal. Bei filib++ handelt es sich um eine Erweiterung der, ursprünglich an der Universität Karlsruhe entwickelten, Intervallbibliothek filib. Der große Unterschied zur filib ist die Möglichkeit zur Nutzung der erweiterten Intervallrechnung und ein modernes Softwaredesign mit Templates und Traits Klassen [33].

### 3.3.1 Datentyp für Intervalle

Der Datentyp für Intervalle wird in filib++ durch die Klasse `interval<N, K, E>` dargestellt. Analog zu den zwei vorher erläuterten Lösungen in Kapitel 3.1 und 3.2, ist es über einen Templateparameter `N` möglich den Basistyp für die beiden Intervallgrenzen generisch auszuprägen. Aktuell ist filib++ für die Gleitkommatypen

- `float`
- `double`

implementiert. Eine Erweiterung auf `long double` oder andere Datentypen ist theoretisch durch einige Spezialisierungen von Traits Klassen möglich. Jedoch

werden die Standardfunktionen aus filib verwendet und müssten somit auch für die neuen Typen implementiert werden [29].

Die interne Repräsentation der Intervalle geschieht über Unter- und Obergrenze, wobei die Bedingung

$$\text{Untergrenze} \leq \text{Obergrenze} \tag{3.4}$$

für nicht leere Intervalle gilt. Leere Intervalle werden über NaNs in der Form [NaN , NaN] dargestellt. In der erweiterten Intervallrechnung sind für die Grenzen auch die Werte $-\infty$ und $+\infty$ erlaubt. Jedoch sind hier die beiden Punktintervalle $[-\infty\,, -\infty]$ und $[+\infty\,, +\infty]$ nicht zulässig.

### Traits Klassen

Wie schon angesprochen verwendet filib++ Traits Klassen für die Implementierung. Die Klasse `fp_traits<N, K>` stellt hierbei alle für den Basistyp `N` grundlegenden Funktionen für eine Intervallrechnung, wie etwa gerichtete Operationen für die Addition, Subtraktion, Multiplikation und Division, aber auch Zugriffsfunktionen auf bestimmte Werte, bereit. Sie ähnelt somit von der Funktionalität der Policy Klasse aus der Boost Intervall Bibliothek, ist jedoch nicht so leicht wie eine Policy Klasse austauschbar. In filib++ wird die entsprechende Spezialisierung von `fp_traits<N, K>` über die beiden Templateparameter `N` und `K` der Intervallklasse `interval<N, K, E>` bestimmt. Der Parameter `K` spezifiziert hierbei die für die Berechnung verwendete Rundungsstrategie, welche über Werte des Aufzählungstyps `rounding_strategy` identifiziert wird. Die Tabelle 3.3 zeigt die aktuell implementierten Kombinationen der Parameter `N` und `K` für die Klasse `fp_traits<N, K>`.

| Basistyp | Aufzählungstyp `rounding_strategy` |
|---|---|
| `float` | `native_switched` |
| `float` | `native_directed` |
| `float` | `multiplicative` |
| `float` | `no_rounding` |
| `float` | `native_onesided_global` |
| `double` | `native_switched` |
| `double` | `native_directed` |
| `double` | `multiplicative` |
| `double` | `no_rounding` |
| `double` | `native_onesided_global` |
| `double` | `pred_succ_rounding` |

Tabelle 3.3: Existierende `fp_traits<N, K>` Implementierungen

Bei einer Intervalloperation, wie etwa der vereinfacht dargestellten Addition in

Listing 3.4, wird dann für die Berechnung der Intervallgrenzen die jeweilige Operation mit gerichteter Rundung der ausgeprägten Traits Klasse verwendet.

```
1 interval<N,K,E> & interval<N,K,E>::operator+=(
2     interval<N,K,E> const & o)
3 {
4     INF = fp_traits<N,K>::downward_plus(INF,o.INF,false);
5     SUP = fp_traits<N,K>::upward_plus(SUP,o.SUP,true);
6
7     return *this;
8 }
```

Listing 3.4: Verwendung von `fp_traits<N, K>` bei einer vereinfachten Addition

Im Beispiel wird über die Methoden `downward_plus` und `upward_plus` die Unter- und Obergrenze mit entprechender Rundung gegen $-\infty$ und $+\infty$ durchgeführt. Der boolsche Parameter signalisiert hierbei der Funktion der Traits Klasse ob der Rundungsmodus nach der Operation auf eine Rundung zur nächsten Zahl gesetzt werden soll. Das Setzen des Rundungsmodus liegt dann aber an der für die Ausprägung der Traits Klasse verwendeten Rundungsstrategie `K`. Der Wert `true` gibt lediglich die Information, dass auf Rundung zur nächsten Zahl gewechselt werden sollte, an die Traits Klasse weiter. Je nach Rundungsstrategie `K` wird diese Aufforderung dann umgesetzt oder auch ignoriert. Wichtig ist dann vor allem, dass filib++, in der aktuell vorliegenden Version [13], die Rundung nicht auf den Zustand vor der Operation sondern auf eine Rundung zur nächsten Zahl zurücksetzt. Dies ist in einem Großteil der Anwendungen vernachlässigbar, da es sich bei der Rundung zur nächsten Zahl um den Standardmodus für Gleitkommaberechnungen handelt. Jedoch muss man diesem Verhalten, bei einer Mischung der Intervallrechnung mit Gleitkommaberechnungen in einem abweichenden Rundungsmodus, besondere Beachtung schenken.

Für das entsprechende Verändern des Rundungsmodus ist dann die jeweilige Ausprägung von `fp_traits<N, K>` selbst zuständig. Hierzu stehen verschiedene Hilfsklassen wie etwa `rounding_control<N, C>` bereit. Diese Klasse stellt eine einheitliche Schnittstelle zum Ändern des Rundungsmodus der jeweils verwendeten Gleitkommaumgebung dar[3]. Der Templateparameter `N` spezifiziert hier wieder den Gleitkommadatentyp. Der boolsche Parameter `C` gibt an, ob der angesprochenen Aufforderung zum Rücksetzen des Rundungsmodus auf eine Rundung zur nächsten Zahl über die Methode `rounding_control<N, C>::reset()` nachgekommen werden soll. Verschiedene Rundungsstrategien mit oder ohne Rücksetzen können dann leicht über eine Ausprägung des Parameters `C` realisiert werden.

Zum Ändern des Rundungsmodus stellt `rounding_control<N, C>` dann die

[3]Unterschiedliche hardwareabhängige Implementierungen können zum Beispiel über Präprozessor-Direktiven realisiert werden.

vier Methoden

- `downward()`
- `upward()`
- `tozero()`
- `tonearest()`

für die, dem IEEE 754 Standard entsprechenden, Rundungen bereit. Weiter beinhaltet die Klasse `rounding_control<N, C>` noch eine Methode `setup()`. Diese Methode wird dann von `fp_traits<N, K>::setup()` verwendet um die Gleitkommaumgebung für eine Intervallrechnung zu initialisieren. Die Methode `fp_traits<N, K>::setup()` muss somit immer aufgerufen werden, wenn bezüglich den Paramtern `N` und `K` unterschiedliche Instanzen der Intervallklasse `interval<N, K, E>` im Programm verwendet werden [28]. Dies gilt vor allem auch für den Programmstart.

Eine Spezialisierung von `fp_traits<N, K>` muss jedoch, durch das offene Konzept der Traits Klassen, nicht zwingend die Klasse `rounding_control<N, C>` verwenden. Einige Ausprägungen benutzen zum Beispiel die Klasse `rounding_control_stub`, welche zwar die gleiche Schnittstelle wie `rounding_control<N, C>` bietet, die Methoden aber nur aus leeren Rümpfen bestehen. Diese Klasse wird vor allem bei Rundungsstrategien verwendet welche nicht auf hardwareseitige Unterstützung angewiesen sind.

### Berechnungsmodell

Von den drei Templateparametern `N`, `K` und `E` wurden bis jetzt nur die ersten beiden besprochen. Der dritte Parameter `E` dient zur Steuerung des Berechnungsmodells. Bei der Instanzierung eines Intervalls kann der Parameter `E` die aufgeführten Werte des Aufzählungstypen `interval_mode` annehmen, um zwischen der "normalen" und der erweiterten Intervallrechnung zu wählen:

`i_mode_normal`: "Normale" Intervallauswertung, wie in Kapitel 1.4, mit Programmabbruch bei ungültigen Operationen, wie etwa Division durch Null oder Funktionsauswertung außerhalb des Definitionsbereichs. Dies ist auch das Standardberechnungsmodell in filib++. Man kann also auf die Angabe von `i_mode_normal` als Parameter `E` verzichten.

`i_mode_extended`: Erweiterte Intervallauswertung, wie in Kapitel 1.5, ohne Programmabbruch.

`i_mode_extended_flag`: Wie `i_mode_extended`, jedoch wird ein Flag gesetzt wenn die "normale" Intervallauswertung einen Fehler erzeugen würde.

Die einzelnen Intervalloperationen, wie etwa die vereinfacht dargestellte Addition aus Listing 3.4, müssen dann in Abhängigkeit des gewählten Berechnungsmodells `E` gegebenenfalls eine Ausnahme für einen Programmabbruch auslösen oder das Flag setzen.

### 3.3.2 Rundungsstrategien

Die Traits Klasse `fp_traits<N, K>` implementiert über den Parameter `K` mehrere verschiedene Rundungsstrategien. Die Rundungsstrategie wird hierbei über einen der folgenden Werte des Aufzählungstyp `rounding_strategy` ausgeprägt:

`native_switched:` Die zugrunde liegenden Gleitkommaoperationen werden hardwareseitig ausgeführt. Es muss also eine Gleitkommaumgebung nach IEEE 754 Standard zu Verfügung stehen. Für die Berechnung der Intervallgrenzen wird der Rundungsmodus jeweils entsprechend auf eine Rundung gegen $-\infty$ oder $+\infty$ gewechselt. Am Ende der Operation wird dann auf Rundung zur nächsten Zahl zurückgesetzt[4]. In filib++ ist `native_switched` die standardmässig verwendete Rundungsstrategie, muss also bei einer Instanzierung eines Intervalls nicht angegeben werden.

`native_directed:` Entspricht der Rundungsstrategie `native_switched`, jedoch wird hier nach einer Operation nicht auf eine Rundung zur nächsten Zahl gewechselt. Dies spart pro Operation einen teuren Rundungswechsel ein, kann aber bei nachfolgenden Gleitkommaberechnungen zu falschen Ergebnissen führen. Bei dieser Rundungsstrategie muss der Benutzer somit nach der Intervallrechnung den Rundungsmodus eigenhändig auf den benötigten Zustand zurücksetzen.

`multiplicative:` Ergebnisse werden durch Multiplizieren mit den Vorgänger $pred(1.0)$ oder Nachfolger $succ(1.0)$ der Zahl 1 im verwendeten Gleitkommaformat berechnet.

$$\triangle x := \begin{cases} x \cdot succ(1.0) & x > 0 \\ x \cdot pred(1.0) & x \leq 0 \end{cases} \tag{3.5}$$

$$\triangledown x := \begin{cases} x \cdot pred(1.0) & x > 0 \\ x \cdot succ(1.0) & x \leq 0 \end{cases} \tag{3.6}$$

Hierdurch kann auf eine gerichtete Rundung verzichtet werden. Voraussetzung ist jedoch, dass der Rundungsmodus mit einer Rundung zur nächsten Zahl arbeitet, was aber über die Methode `fp_traits<N, K>::setup()` sichergestellt wird.

[4]Dies muss aber nicht dem Rundungsmodus vor der Intervalloperation entsprechen.

`no_rounding`: Die Berechnungen werden ohne gerichtete Rundung ausgeführt. Die Einschließungseigenschaft aus Satz 3 wird in dieser Strategie nicht eingehalten.

`native_onesided_global`: Es wird hier durch Anwenden der Formel $\triangle a = -\bigtriangledown(-a)$ aus Satz 8 nur mit einer gerichteten Rundung gegen $-\infty$ ohne Rücksetzen gearbeitet. Somit können die Wechsel der Rundungsmodi sehr stark reduziert werden. Über `fp_traits<N, K>::setup()` wird auch hier die Rundung für die Intervallrechnung initialisiert. Für gemischte Intervall- und Gleitkommarechnungen gelten die gleichen Einschränkungen wie bei `native_directed` und der Benutzer muss gegebenenfalls die Rundung für die Gleitkommaberechnung eigenhändig anpassen.

`pred_succ_rounding`: Rundung wird nach der Berechnung der Grenzwerte $x_1$ und $x_2$ über den Vorgänger $pred(x_1)$ beziehungsweise dem Nachfolger $succ(x_2)$ der Ergebnisse simuliert. Wichtig ist hier, dass die Ergebnisse $x_1$ und $x_2$ auf ein ulp genau berechnet wurden, um die Einschließungseigenschaft beizubehalten.

### 3.3.3 Operationen

Operationen stehen sowohl mit "normaler" als auch erweiterter Intervallauswertung, wie in den Kapiteln 1.4 und 1.5 definiert, bereit.

Die vier Grundrechenarten entsprechen hierbei den in Kapitel 2.6 angegebenen Regeln. Bei einer Division mit Null[5] wird im erweiterten Modus das Intervall $[-\infty\,,\,+\infty] = \mathbb{R}^*$ als Ergebnis zurückgegeben.

Standardfunktionen liefern ein einschließendes Intervall der möglichen Werte der Funktion. Jedoch werden hier nicht zwangsläufig die kleinstmöglichen einschließenden Intervalle erzeugt. Weiter sind sie auch nur für den Basisdatentyp `double` über die filib Bibliothek implementiert [28]. Ein "Backward" oder "Reverse" Modus [34] steht in filib++ nicht zur Verfügung.

#### Flag bei erweiterter Intervallrechnung

Über den Berechnungsmodus `i_mode_extended_flag` bekommt der Benutzer Informationen über Operationen mitgeteilt, welche bei einer "normalen" Intervallauswertung zu einem Programmfehler geführt hätten. Dies tritt immer dann auf, wenn eine Intervalloperation ein NaN, einen Überlauf einer Gleitkommzahl oder ein leeres Intervall erzeugt. Natürlich zählen auch Funktionsaufrufe außerhalb des Definitionsbereichs oder die Division durch Null zu diesen ungültigen Operationen.

[5] Unabhängig davon ob $[0\,,\,0]$ oder $[a\,,\,b]$ mit $a < 0$ und $b > 0$.

Im Berechnungsmodus `i_mode_extended` oder `i_mode_extended_flag` der erweiterten Intervallrechnung werden diese Operationen jedoch ohne Programmfehler ausgeführt. Der Modus `i_mode_extended_flag` verwendet zusätzlich zu `i_mode_extended` noch ein globales Flag, durch welches er "ungültige" Intervalloperationen kennzeichnet. Der Anwender kann dann über die zwei folgenden Methoden auf dieses Flag zugreifen:

`bool getExtendedErrorFlag():` Gibt den aktuellen Zustand des globalen Flags zurück.

`void setExtendedErrorFalg():` Setzt das globale Flag wieder auf den "fehlerfreien" Zustand (`false`) zurück.

Intervalloperationen verändern den Wert des Flags nur bei "ungültigen" Operationen und setzen dabei den boolschen Wert auf `true`. Dieser Wert bleibt dann bis zum händischen Eingreifen des Benutzers erhalten. Für ein Zurücksetzen auf `false` ist der Anwender über die Methode `setExtendedErrorFalg()` selbst zuständig.

Über diese Funktionalität kann der Benutzer somit beliebige Blöcke von Intervallberechnungen mit einem einzigen Flag versehen und jeweils zu Beginn des Blocks den Wert zurücksetzen. Jedoch muss sich der Benutzer auch über die Konsequenz eines globalen Flags bewußt sein. So können im Berechnungsmodell `i_mode_extended_flag` nicht mehrere Intervallberechnungen auf verschiedenen Threads parallel ausgeführt werden. Ein für jeden Thread konsistenter Zustand des globalen Flags kann hier nicht sichergestellt werden.

### 3.3.4 Vergleichsoperationen

Die Vergleichsoperationen entsprechen den expliziten Vergleichsfunktionen aus der Boost Intervallbibliothek in Kapitel 3.2.4. Es stehen "certainly" und "possibly" Vergleiche aus Kapitel 1.2.3 über Funktionen zur Verfügung. Die Funktionsnamen setzen sich hierbei aus den Buchstaben `c` für "certainly" und `p` für "possibly" Vergleiche sowie den Bezeichnern aus Tabelle 3.4 für die jeweilige Relation zusammen.

| Vergleich | Bezeichner |
|---|---|
| $=$ | `eq` |
| $\neq$ | `ne` |
| $<$ | `lt` |
| $\leq$ | `le` |
| $>$ | `gt` |
| $\geq$ | `ge` |

Tabelle 3.4: Bezeichner für "certainly" und "possibly" Vergleiche

Die Funktionsnamen für die angebotenen Mengenrelationen bilden sich aus dem Buchstaben `s` sowie den Bezeichnern aus Tabelle 3.4. Eine Mengenrelation gilt hierbei als erfüllt, wenn jeweils die Untergrenzen und Obergrenzen die angegebene Vergleichsrelation aus Tabelle 3.4 erfüllen [29]. Beispielsweise ist die Mengenrelation `sge` für zwei Intervalle `X` und `Y` wie folgt definiert:

$$\texttt{X.sge(Y)} := \texttt{X.inf()} \geq \texttt{Y.inf()} \wedge \texttt{X.sup()} \geq \texttt{Y.sup()} \tag{3.7}$$

Die anderen Funktionen werden analog definiert. Alternativ können für die beiden Funktionen `seq` und `sne` auch die Operatoren `==` und `!=` verwendet werden. Zusätzlich sind auch die Relationen $\subseteq$, $\subset$, $\supseteq$ und $\supset$ über die Methoden `subset`, `proper_subset`, `superset`, `proper_superset` sowie durch Überladung der Operatoren `<=`, `<`, `>=` und `>` implementiert.

# Kapitel 4

# Anforderungen und Möglichkeiten einer Implementierung

Grundlegend für eine Implementierung der Intervallarithmetik auf Rechnern ist die Definition eines Datentyps `Interval`. Je nach Anforderung kann, analog zu den Kapiteln 1.4 und 1.5, der Datentyp für eine "normale" oder erweiterte Intervallauswertung entworfen werden. Jedoch überwiegen die Vorteile der ausnahmefreien erweiterten Intervallrechnung, so dass man sich beim Entwurf des Datentyps nicht auf die "normale" Intervallrechnung einschränken sollte[1].

## 4.1 Darstellbare Mengen

Bei den beiden Grenzwerten $\pm\infty$ handelt es sich nicht um Elemente der reellen Zahlen $\mathbb{R}$ und sollten folglich nicht als Werte eines Intervalls aufgefasst werden [26]. Die Werte $-\infty$ und $+\infty$ werden für den Datentyp `Interval` nur als Symbole verwendet, um unbeschränkte Intervalle darzustellen. Die Kodierung $[-\infty\,,\,a_2]$, $[a_1\,,\,+\infty]$ sowie $[-\infty\,,\,+\infty]$ mit den Symbolen $\pm\infty$ führt zusammen mit den beiden endlichen Gleitkommazahlen $a_1$ und $a_2$ zu den drei folgenden Intervallen:

- $(-\infty\,,\,a_2] := \{a \in \mathbb{R} \mid a \leq a_2\}$
- $[a_1\,,\,+\infty) := \{a \in \mathbb{R} \mid a \geq a_1\}$
- $(-\infty\,,\,+\infty) = \mathbb{R}$

Die beiden Darstellungen $[-\infty\,,\,-\infty]$ und $[+\infty\,,\,+\infty]$ machen, durch die Betrachtung der Symbole $\pm\infty$ als uneigentliches Infimum oder Supremum eines Intervalls, wenig Sinn und sind als Punktintervalle auch nicht gewünscht [26, 34, 54].

[1] Diese könnte, wenn gewünscht, softwareseitig durch spezielle Policy oder Traits Klassen, ähnlich der Boost Intervall Bibliothek oder filib++, auch mit dem erweiterten Datentyp realisiert werden.

Wendet man diese Einschränkungen auf die erweiterten Intervalle aus Definition 7 an, so muss der Datentyp `Interval` folglich die aufgelisteten Mengen

- $[a_1\,,\,a_2]$
- $(-\infty\,,\,a_2]$
- $[a_1\,,\,+\infty)$
- $(-\infty\,,\,+\infty)$
- $\emptyset$

hard- oder softwareseitig abbilden können. Bei den beiden Werten $a_1$ und $a_2$ handelt es sich hierbei um zwei endliche Gleitkommazahlen. Der Datentyp `Interval` entspricht somit der in [26, 54] definierten Menge der erweiterten Gleitkommaintervalle $\mathbb{I}$F.

**Definition 13** *Für beliebige endliche Gleitkommazahlen $a_1$ und $a_2$ eines Gleitkommasystems bildet*

$$\begin{aligned}\mathbb{I}F := \; & \{[a_1\,,\,a_2] \mid a_1 \leq a_2\} \cup \\ & \{(-\infty\,,\,a_2] \mid a_2 \geq -\infty\} \cup \\ & \{[a_1\,,\,+\infty) \mid a_1 \leq +\infty\} \cup \\ & \{(-\infty\,,\,+\infty)\} \cup \\ & \{\emptyset\}\end{aligned} \tag{4.1}$$

*die Menge der erweiterten Gleitkommaintervalle.*

Verwendet man, analog zu [26, 54], die angegebenen Tabellen 4.1, 4.2, 4.3, 4.4 sowie 4.5 für die Implementierung der vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division, so bildet $\mathbb{I}$F ein abgeschlossenes System[2].

**Satz 9** *Das System $\mathbb{I}F$ ist unter den vier Grundrechenarten abgeschlossen.*

**Beweis:** Siehe [26, 54]. Die Beweisidee kann den Tabellen 4.1, 4.2, 4.3, 4.4 sowie 4.5 entnommen werden. □

[2]Operationen auf leeren Mengen liefern hierbei stets die leere Menge.

| **Addition** | $(-\infty\,,\,b_2]$ | $[b_1\,,\,b_2]$ | $[b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
|---|---|---|---|---|
| $(-\infty\,,\,a_2]$ | $(-\infty\,,\,a_2 \overset{+}{\triangle} b_2]$ | $(-\infty\,,\,a_2 \overset{+}{\triangle} b_2]$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$ | $(-\infty\,,\,a_2 \overset{+}{\triangle} b_2]$ | $[a_1 \overset{+}{\triangledown} b_1\,,\,a_2 \overset{+}{\triangle} b_2]$ | $[a_1 \overset{+}{\triangledown} b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $[a_1 \overset{+}{\triangledown} b_1\,,\,+\infty)$ | $[a_1 \overset{+}{\triangledown} b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

Tabelle 4.1: Addition erweiterter Intervalle

| **Subtraktion** | $(-\infty\,,\,b_2]$ | $[b_1\,,\,b_2]$ | $[b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
|---|---|---|---|---|
| $(-\infty\,,\,a_2]$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,a_2 \overset{-}{\triangle} b_1]$ | $(-\infty\,,\,a_2 \overset{-}{\triangle} b_1]$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$ | $[a_1 \overset{-}{\triangledown} b_2\,,\,+\infty)$ | $[a_1 \overset{-}{\triangledown} b_2\,,\,a_2 \overset{-}{\triangle} b_1]$ | $(-\infty\,,\,a_2 \overset{-}{\triangle} b_1]$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$ | $[a_1 \overset{-}{\triangledown} b_2\,,\,+\infty)$ | $[a_1 \overset{-}{\triangledown} b_2\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

Tabelle 4.2: Subtraktion erweiterter Intervalle

### 4.1.1 "Not an Interval"

Analog zu `NaN` bei den Gleitkommazahlen, ist `NaI` als "Nicht-Intervall" für Intervalle denkbar. Der Wert `NaI` symbolisiert hierbei, dass es sich um ein unzulässiges Intervall handelt. `NaI` ist somit kein Element der Menge der Gleitkommaintervalle $\mathbb{IF}$.

Anders als bei den Gleitkommazahlen, wo durch ungültige Operationen NaNs als Ergebnis erzeugt werden, kann bei der erweiterten Intervallauswertung keine unzulässige Intervalloperation ausgeführt werden. Durch lose Funktionsauswertung sind Intervalloperationen für alle Intervalle aus $\mathbb{IF}$ definiert. Im schlechtesten Fall wird als Ergebnis lediglich die leere Menge berechnet. Demnach können unzulässige Intervalle ausschließlich bei der Initialisierung durch Konstruktoren, der Konvertierung aus anderen Datentypen oder durch andere Eingabe-Methoden erzeugt werden. Hier muss man dann entscheiden, ob das Programm abbricht und eine Ausnahme auslöst oder ob ein, für den Fehlerfall, definiertes Intervall erzeugt wird. Als zu erzeugende Intervalle in einem dieser Fehlerfälle sind das leere Intervall $\emptyset$ sowie ein, für diesen Anwendungsfall vorgesehener, Wert `NaI` denkbar.

Das leere Intervall hat den Vorteil, dass es sich um ein "vollwertiges" Element der verwendeten Gleitkommaintervalle handelt. Alle Operationen sind durch die lose Funktionsauswertung für diesen Wert definiert. Leere Intervalle werden somit durch komplette Intervallausdrücke propagiert und als Ergebnis des Ausdrucks geliefert. Das ausnahmefreie Konzept der erweiterten Intervallrechnung bleibt demnach erhalten. Jedoch ist für einen Anwender nicht mehr zu unterscheiden, ob das leere Intervall das Ergebnis eines unzulässigen Konstruktoraufrufs oder einer losen Funktionsauswertung außerhalb des Defintionsbereichs ist. Hier kann, wie beispielsweise in der C++ Bibliothek filib++ [28], ein Flag verwendet werden,

| **Multiplikation** | $[0\,,\,0]$ | $[b_1\,,\,b_2]$<br>$b_2 \leq 0$ | $[b_1\,,\,b_2]$<br>$b_1 < 0 < b_2$ |
|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 \leq 0$ | $[0\,,\,0]$ | $[a_2 \overline{\cdot} b_2\,,\,a_1 \underline{\cdot} b_1]$ | $[a_1 \overline{\cdot} b_2\,,\,a_1 \underline{\cdot} b_1]$ |
| $[a_1\,,\,a_2]$, $a_1 < 0 < a_2$ | $[0\,,\,0]$ | $[a_2 \overline{\cdot} b_1\,,\,a_1 \underline{\cdot} b_1]$ | $[\min\{a_1 \overline{\cdot} b_2,\ a_2 \overline{\cdot} b_1\},$<br>$\max\{a_1 \underline{\cdot} b_1,\ a_2 \underline{\cdot} b_2\}]$ |
| $[a_1\,,\,a_2]$, $a_1 \geq 0$ | $[0\,,\,0]$ | $[a_2 \overline{\cdot} b_1\,,\,a_1 \underline{\cdot} b_2]$ | $[a_2 \overline{\cdot} b_1\,,\,a_2 \underline{\cdot} b_2]$ |
| $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ |
| $(-\infty\,,\,a_2]$, $a_2 \leq 0$ | $[0\,,\,0]$ | $[a_2 \overline{\cdot} b_2\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 \geq 0$ | $[0\,,\,0]$ | $[a_2 \overline{\cdot} b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \leq 0$ | $[0\,,\,0]$ | $(-\infty\,,\,a_1 \underline{\cdot} b_1]$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \geq 0$ | $[0\,,\,0]$ | $(-\infty\,,\,a_1 \underline{\cdot} b_2]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $[0\,,\,0]$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

| **Multiplikation** | $[b_1\,,\,b_2]$<br>$b_1 \geq 0$ | $(-\infty\,,\,b_2]$<br>$b_2 \leq 0$ | $(-\infty\,,\,b_2]$<br>$b_2 \geq 0$ |
|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 \leq 0$ | $[a_1 \overline{\cdot} b_2\,,\,a_2 \underline{\cdot} b_1]$ | $[a_2 \overline{\cdot} b_2\,,\,+\infty)$ | $[a_1 \overline{\cdot} b_2\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 < 0 < a_2$ | $[a_1 \overline{\cdot} b_2\,,\,a_2 \underline{\cdot} b_2]$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 \geq 0$ | $[a_1 \overline{\cdot} b_1\,,\,a_2 \underline{\cdot} b_2]$ | $(-\infty\,,\,a_1 \underline{\cdot} b_2]$ | $(-\infty\,,\,a_2 \underline{\cdot} b_2]$ |
| $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ |
| $(-\infty\,,\,a_2]$, $a_2 \leq 0$ | $(-\infty\,,\,a_2 \underline{\cdot} b_1]$ | $[a_2 \overline{\cdot} b_2\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 \geq 0$ | $(-\infty\,,\,a_2 \underline{\cdot} b_2]$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \leq 0$ | $[a_1 \overline{\cdot} b_2\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \geq 0$ | $[a_1 \overline{\cdot} b_1\,,\,+\infty)$ | $(-\infty\,,\,a_1 \underline{\cdot} b_2]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

| **Multiplikation** | $[b_1\,,\,+\infty)$<br>$b_1 \leq 0$ | $[b_1\,,\,+\infty)$<br>$b_1 \geq 0$ | $(-\infty\,,\,+\infty)$ |
|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 \leq 0$ | $(-\infty\,,\,a_1 \underline{\cdot} b_1]$ | $(-\infty\,,\,a_2 \underline{\cdot} b_1]$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 < 0 < a_2$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 \geq 0$ | $[a_2 \overline{\cdot} b_1\,,\,+\infty)$ | $[a_1 \overline{\cdot} b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ |
| $(-\infty\,,\,a_2]$, $a_2 \leq 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,a_2 \underline{\cdot} b_1]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 \geq 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \leq 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \geq 0$ | $(-\infty\,,\,+\infty)$ | $[a_1 \overline{\cdot} b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

Tabelle 4.3: Multiplikation erweiterter Intervalle

| **Division** $0 \notin B$ | $[b_1\,,\,b_2]$ $b_2 < 0$ | $[b_1\,,\,b_2]$ $b_1 > 0$ |
|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 \leq 0$ | $[a_2 \overline{\div} b_1\,,\,a_1 \underline{\div} b_2]$ | $[a_1 \overline{\div} b_1\,,\,a_2 \underline{\div} b_2]$ |
| $[a_1\,,\,a_2]$, $a_1 < 0 < a_2$ | $[a_2 \overline{\div} b_2\,,\,a_1 \underline{\div} b_2]$ | $[a_1 \overline{\div} b_1\,,\,a_2 \underline{\div} b_1]$ |
| $[a_1\,,\,a_2]$, $a_1 \geq 0$ | $[a_2 \overline{\div} b_2\,,\,a_1 \underline{\div} b_1]$ | $[a_1 \overline{\div} b_2\,,\,a_2 \underline{\div} b_1]$ |
| $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ |
| $(-\infty\,,\,a_2]$, $a_2 \leq 0$ | $[a_2 \overline{\div} b_1\,,\,+\infty)$ | $(-\infty\,,\,a_2 \underline{\div} b_2]$ |
| $(-\infty\,,\,a_2]$, $a_2 \geq 0$ | $[a_2 \overline{\div} b_2\,,\,+\infty)$ | $(-\infty\,,\,a_2 \underline{\div} b_1]$ |
| $[a_1\,,\,+\infty)$, $a_1 \leq 0$ | $(-\infty\,,\,a_1 \underline{\div} b_2]$ | $[a_1 \overline{\div} b_1\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \geq 0$ | $(-\infty\,,\,a_1 \underline{\div} b_1]$ | $[a_1 \overline{\div} b_2\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

| **Division** $0 \notin B$ | $(-\infty\,,\,b_2]$ $b_2 < 0$ | $[b_1\,,\,+\infty)$ $b_1 > 0$ |
|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 \leq 0$ | $[0\,,\,a_1 \underline{\div} b_2]$ | $[a_1 \overline{\div} b_1\,,\,0]$ |
| $[a_1\,,\,a_2]$, $a_1 < 0 < a_2$ | $[a_2 \overline{\div} b_2\,,\,a_1 \underline{\div} b_2]$ | $[a_1 \overline{\div} b_1\,,\,a_2 \underline{\div} b_1]$ |
| $[a_1\,,\,a_2]$, $a_1 \geq 0$ | $[a_2 \overline{\div} b_2\,,\,0]$ | $[0\,,\,a_2 \underline{\div} b_1]$ |
| $[0\,,\,0]$ | $[0\,,\,0]$ | $[0\,,\,0]$ |
| $(-\infty\,,\,a_2]$, $a_2 \leq 0$ | $[0\,,\,+\infty)$ | $(-\infty\,,\,0]$ |
| $(-\infty\,,\,a_2]$, $a_2 \geq 0$ | $[a_2 \overline{\div} b_2\,,\,+\infty)$ | $(-\infty\,,\,a_2 \underline{\div} b_1]$ |
| $[a_1\,,\,+\infty)$, $a_1 \leq 0$ | $(-\infty\,,\,a_1 \underline{\div} b_2]$ | $[a_1 \overline{\div} b_1\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 \geq 0$ | $(-\infty\,,\,0]$ | $[0\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

Tabelle 4.4: Division erweiterter Intervalle mit $0 \notin B$

| **Division** $0 \in B$ | $[0\,,\,0]$ | $[b_1\,,\,b_2]$ $b_1 < b_2 = 0$ | $[b_1\,,\,b_2]$ $b_1 < 0 < b_2$ |
|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 < 0$ | $\emptyset$ | $[a_2 \overline{\div} b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 \leq 0 \leq a_2$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 > 0$ | $\emptyset$ | $(-\infty\,,\,a_1 \underline{\div} b_1]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 < 0$ | $\emptyset$ | $[a_2 \overline{\div} b_1\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 > 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 < 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 > 0$ | $\emptyset$ | $(-\infty\,,\,a_1 \underline{\div} b_1]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

| **Division** $0 \in B$ | $[b_1\,,\,b_2]$ $0 = b_1 < b_2$ | $(-\infty\,,\,b_2]$ $b_2 = 0$ | $(-\infty\,,\,b_2]$ $b_2 > 0$ |
|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 < 0$ | $(-\infty\,,\,a_2 \underline{\div} b_2]$ | $[0\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 \leq 0 \leq a_2$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 > 0$ | $[a_1 \overline{\div} b_2\,,\,+\infty)$ | $(-\infty\,,\,0]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 < 0$ | $(-\infty\,,\,a_2 \underline{\div} b_2]$ | $[0\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 > 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 < 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 > 0$ | $[a_1 \overline{\div} b_2\,,\,+\infty)$ | $(-\infty\,,\,0]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

| **Division** $0 \in B$ | $[b_1\,,\,+\infty)$ $b_1 < 0$ | $[b_1\,,\,+\infty)$ $b_1 = 0$ | $(-\infty\,,\,+\infty)$ |
|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 < 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,0]$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 \leq 0 \leq a_2$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 > 0$ | $(-\infty\,,\,+\infty)$ | $[0\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 < 0$ | $(-\infty\,,\,0]$ | $(-\infty\,,\,0]$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,a_2]$, $a_2 > 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 < 0$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $[a_1\,,\,+\infty)$, $a_1 > 0$ | $(-\infty\,,\,+\infty)$ | $[0\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |
| $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ | $(-\infty\,,\,+\infty)$ |

Tabelle 4.5: Division erweiterter Intervalle mit $0 \in B$

um die notwendige Information an den Benutzer weiter zu reichen.

Durch einen speziellen Wert `NaI` kann auf dieses Flag bei einer unzulässigen Eingabe oder Initialisierung verzichtet werden. Ansonsten ist für ein `NaI` aber das gleiche Verhalten wie für das leere Intervall gefordert. NaIs müssen ebenfalls durch komplette Intervallausdrücke propagiert werden. Jedoch sind, um die Information auch an den Benutzer weiter zu reichen, NaIs stärker als NaNs zu gewichten. NaIs sind deshalb als eine Kombination aus leerem Intervall und einem Fehlerflag anzusehen.

Für eine Implementierung eines Datentyps `Interval` mit NaIs empfiehlt es sich demnach, NaIs als Spezialfall der leeren Menge zu betrachten. Dies ist über eine Darstellung der leeren Menge als Intervall [`NaN` , `NaN`] leicht zu realisieren. Bei der Kodierung eines NaNs im IEEE 754 Standard [37] ist für die Mantisse nur ein Wert ungleich Null gefordert und auch das Vorzeichen wird nicht beachtet. Mit diesen Bits können dann verschiedene Kodierungen von NaNs zur Darstellung von unterschiedlichen leeren Intervallen, wie beispielsweise $\emptyset = [+\texttt{NaN}, +\texttt{NaN}]$ und $\texttt{NaI} = [-\texttt{NaN}, -\texttt{NaN}]$, verwendet werden.

## 4.2 Repräsentation von Intervallen

Für die Realisierung auf Rechnern bietet es sich natürlich an den Datentyp `Interval` auf den nach IEEE 754 standardisierten Gleitkommazahlen aufzubauen. Intervalle sind aber dennoch als kontinuierliche Teilmengen der reellen Zahlen anzusehen, lediglich bei den Grenzen handelt es sich um Gleitkommazahlen.

Es ist auch durchaus erstrebenswert Intervalle möglichst platzsparend zu kodieren. Die in Kapitel 1 eingeführten Schreibweisen über Unter- und Obergrenze oder über Mittelpunkt und Radius geben hier eine gute Hilfestellung. So genügen für einen Datentyp `Interval` zwei Gleitkommazahlen, um die benötigten Werte eines Intervalls abzubilden.

Die IEEE 754 Gleitkommazahlen beinhalten neben den normalisierten Gleitkommazahlen einige spezielle Werte. Hier muss vorher geklärt werden ob und wie diese Werte als Intervallgrenzen verwendet werden können:

**Denormalisierte Zahlen:** Bei einer denormalisierten Gleitkommazahl handelt es sich um eine zulässige endliche reelle Zahl. Aus diesem Grund spricht nichts gegen denormalisierte Zahlen als Intervallgrenze.

**$\pm 0$:** Bei den nach IEEE 754 standardisierten Gleitkommazahlen existiert sowohl der Wert $-0$ als auch $+0$. Auch hier sind beide Werte als Intervallgrenze zulässig. Bei Intervalloperationen wird jedoch nicht zwischen $\pm 0$ unterschieden. Der Wert muss, im Gegensatz zu den IEEE 754 Gleitkommazahlen[3], vorzeichenlos verarbeitet werden.

[3] Bei den IEEE 754 Gleitkommazahlen ist die Division durch Null vom Vorzeichen abhängig.

**$\pm\infty$:** Wie im Kapitel 4.1 angesprochen, sind $\pm\infty$ für Intervalle der Form $(-\infty\,,\,a_2]$, $[a_1\,,\,+\infty)$ und $(-\infty\,,\,+\infty)$vorgesehen.

**NaN:** NaNs sind nicht als numerischer Grenzwert eines Intervalls anzusehen. Jedoch können sie sehr gut zur Kodierung des leeren Intervalls verwendet werden.

### 4.2.1 Kodierung über Unter- und Obergrenze

Eine der geläufigsten Darstellungen von Intervallen ist die Verwendung von Unter- und Obergrenze $a_1$ und $a_2$. Die Speicherung wird dann über zwei Felder des verwendeten Gleitkommaformats, analog zu der im Kapitel 1 eingeführten Schreibweise für Intervalle, durchgeführt. Der Wert eines Intervalls $[a_1\,,\,a_2]$ mit $a_1 \leq a_2$ entspricht somit der maschinellen Darstellung $\langle a_1\,;\,a_2\rangle$[4].

Einzige Ausnahme bilden die leere Menge sowie möglicherweise verwendete NaIs. Für das leere Intervall bietet sich, wie schon bei den NaIs angesprochen, die Darstellung $\langle$`NaN`$\,;\,$`NaN`$\rangle$ über (stille) NaNs an. Dies hat den Vorteil, dass auf IEEE 754 konformen Systemen NaNs durch alle Gleitkommaoperationen weitergeleitet werden. Die vier Grundrechenarten könnten, wie in Kapitel 2.6 angegeben, ohne spezielle Behandlung der leeren Menge durch die zugrunde liegenden Gleitkommaoperationen realisiert werden. Ein weiterer Vorteil ist die Möglichkeit, durch unterschiedliche Kodierungen der NaNs zusätzliche Informationen in leeren Intervallen zu speichern. Dies kann die schon angesprochene Darstellung von NaIs sein, aber auch andere Debuginformation ist denkbar.

Eine andere Möglichkeit für die leere Menge bietet die Kodierung $\langle a\,;\,b\rangle$ über zwei Gleitkommawerte mit der Bedingung $a > b$. Jedoch muss hier bei Operationen eine spezielle Behandlung der leeren Menge vorgesehen werden. Wird die leere Menge beispielsweise über $\langle 1\,;\,-1\rangle$ dargestellt, und es findet eine Multiplikation mit dem Punktintervall $[1\,,\,1]$ statt, so wird mit der Multiplikation aus Kapitel 2.6 das Intervall $\langle -1\,;\,1\rangle \hat{=} [-1\,,\,1] \neq \emptyset$ als Ergebnis berechnet. Auch die oft verwendete Darstellung $\langle +\infty\,;\,-\infty\rangle$ bringt hier keine Besserung. Das Weiterreichen von leeren Intervallen muss demnach für diese Kodierung immer explizit realisiert werden. Jedoch hat diese Darstellung der leeren Menge auf einigen nicht IEEE 754 konformen Systemen ihre Berechtigung. Sind NaNs auf dem verwendeten System nicht für das zugrunde liegende Gleitkommaformat implementiert, stellt die Kodierung mit vertauschten Grenzen die einzig sinnvolle Möglichkeit dar, die leere Menge mit zwei Feldern des Gleitkommadatentyps zu realisieren.

#### Kodierung nach Lambov

Eines der größten Probleme der Intervallarithmetik auf Rechnern sind die benötigten gerichteten Rundungen der Gleitkommazahlen bei der Berechnung. Auf

[4]Hier sind auch die Werte $a_1 = -\infty$ sowie $a_2 = +\infty$ zulässig.

vielen Systemen wird die Rundung der Operationen global gesteuert und ein Wechsel in einen anderen Modus ist mit hohen Laufzeitkosten verbunden. Aus diesem Grund wird oft durch anwenden der Formel $\triangle a = - \bigtriangledown (-a)$ aus Satz 8 versucht Intervalloperationen nur mit einer gerichteten Rundung durchzuführen. Mit der angebenen Formel würde das bedeuten, dass für Intervalloperationen auf die Rundung $\bigtriangledown$ gewechselt wird. Die Rundung $\triangle$ wird dann über die angegebene Formel simuliert.

Für IEEE 754 Gleitkommzahlen ist eine Rundung zur nächsten Zahl der Standardmodus. Und auch die meisten Anwendungen und Bibliotheken setzen diesen Rundungsmodus voraus. Dies ist wohl auch der Grund für fehlerhafte Optimierungen durch einige C++ Compiler bei Verwendung der gerichteten Rundung. Für die standardmässige Rundung zur nächsten Zahl gilt die folgende Gleichung:

$$\bigcirc a = - \bigcirc (-a) \tag{4.2}$$

Auf die doppelte Negation kann somit bei der Rundung $\bigcirc$ verzichtet werden und viele C++ Compiler führen eine entsprechende Optimierung des Programmcodes durch. Bei den, in der Intervallarithmetik verwendeten, gerichteten Rundungen $\bigtriangledown$ und $\triangle$ ist diese Optimierung jedoch nicht zulässig, wird aber dennoch von einigen Compilern durchgeführt [27].

Um diese fehlerhaften Compileroptimierungen zu umgehen, verwendet Branimir Lambov in seiner Implementierung der Intervallrechnung [47] eine Darstellung mit negierter Obergrenze[5] [27] nach folgendem Schema:

$$[a_1\,,\, a_2] \hat{=} \langle a_1\,;\, -a_2 \rangle \tag{4.3}$$

Durch diese Darstellung kann die Regel $\triangle a = - \bigtriangledown (-a)$ auf Intervalloperationen angewendet werden, ohne dass der Compiler doppelte Negationen fehlerhaft optimieren kann.

Für die Kodierung der leeren Menge bieten sich die gleichen Möglichkeiten wie bei der schon angesprochenen "normalen" Kodierung über Unter- und Obergrenze an.

### Kodierung mit "Wraparound" Intervallen

Bei unstetigen Funktionen ist die Wertemenge natürlich nicht zwingend kontinuierlich und somit auch nicht als Intervall darstellbar. Abhilfe schafft hier das Einschließen der Wertemenge durch ein Intervall. Die, bei der Intervallrechnung, wichtige Einschließungseigenschaft bleibt erhalten.

Betrachtet man jedoch die Division durch Null aus Formel (4.4), so werden bei einer losen Funktionsauswertung zwei disjunkte Intervalle $(-\infty\,,\, -100]$ und

[5] Alternativ kann durch Anwendung von $\bigtriangledown$a = - $\triangle$(-a) auch die untere Grenze negiert werden.

$[100\,,\,+\infty)$berechnet.

$$[100\,,\,100] \div [-1\,,\,1] = (-\infty\,,\,-100] \cup [100\,,\,+\infty) \tag{4.4}$$

Das einzig mögliche einschließende Intervall ist hier $(-\infty\,,\,+\infty) = \mathbb{R}$ und würde das korrekte Ergebnis um den Bereich $(-100\,,\,100)$ überschätzen.

Um diese Überschätzung zu vermeiden, ist es denkbar auch zwei disjunkte Intervalle als Ergebnis zuzulassen [26, 44]. Die Kodierung der Menge $(-\infty\,,\,l] \cup [r\,,\,+\infty)$ mit $l < r$ kann ebenfalls mit den beiden Gleitkommawerten des Datentyps `Interval` als sogenanntes "Wraparound" Intervall realisiert werden:

$$(-\infty\,,\,l] \cup [r\,,\,+\infty) \hat{=} \langle r\,;\,l \rangle \tag{4.5}$$

Die beiden Grenzwerte $l$ und $r$ werden also vertauscht abgespeichert um sie von den "normalen" Intervallen mit Maschinendarstellung $\langle a_1\,;\,a_2\rangle$ mit $a_1 \leq a_2$ abzugrenzen[6].

Das Problem bei den "Wraparound" Intervallen ist aber, dass hier zwei disjunkte Intervalle in einem Datentyp `Interval` gespeichert werden. Es handelt sich demnach nicht mehr um ein Intervall und muss in allen Intervalloperationen berücksichtigt werden. Dies würde die einzelnen Intervalloperationen jedoch stark verkomplizieren und betrifft auch Berechnungen, die komplett ohne "Wraparound" Intervalle auskommen. Zusätzlich bekommt man, wenn mehr als zwei disjunkte Intervalle benötigt werden, ein Problem mit der Darstellung. Auch eine Darstellung von zwei disjunkten beschränkten Intervallen der Form $[a_1\,,\,a_2] \cup [b_1\,,\,b_2]$ mit $a_2 < b_1$ ist nicht möglich und die Umsetzung somit inkonsequent.

Dies sind Gründe, die gegen eine Implementierung von "Wraparound" Intervallen sprechen. Um der genannten Überschätzung entgegenzuwirken sind spezielle Methoden, welche jeweils nur eine Teilmenge berechnen[7] oder die Teilmengen als Feld zusammenschließen, besser geeignet. Auch die in filib++ [29] verwendeten Flags sind hier hilfreich und können zum Erkennen von Überschätzungen verwendet werden.

### 4.2.2 Kodierung über Mittelpunkt und Radius

Als Alternative zur Speicherung der Intervalle über Unter- und Obergrenze sowie den beiden Spezialfällen, bietet sich auch eine maschinelle Kodierung über den Mittelpunkt $m$ und den Radius $r \geq 0$ des Intervalls an:

$$[m\,;\,r] \hat{=} \langle m\,;\,r \rangle \tag{4.6}$$

[6] Natürlich ist es auch möglich "Wraparound" Intervalle mit der Kodierung von Lambov zu realisieren.

[7] Ähnlich zu `division_part1` und `division_part2` der Boost Intervallbibliothek [6]

Punktintervalle besitzen den Radus $r = 0$ und echte Intervalle den entsprechenden positiven Wert als Radius. Leere Intervalle können wieder über zwei NaNs abgebildet werden.

Jedoch bereiten Intervalle in Mittelpunkt-Radius Darstellung mittels Gleitkommazahlen einige Probleme. Arithmetische Operationen sind durch die zentrierte Darstellung schwerer mit Gleitkommazahlen zu realisieren. Rundungsfehler, die bei der Berechnung eines Mittelpunktes auftreten, müssen durch den Radius des neuen Intervalls abgedeckt werden [31, 35]. Andernfalls ist die Einschließungseigenschaft nicht sichergestellt. Und vor allem Methoden wie Multiplikation oder Division sind in der Mittelpunkt-Radius Darstellung um einiges aufwändiger als bei Intervallen mit Unter- und Obergrenze [31, 35].

#### Darstellung von einseitig unbeschränkten Intervallen

Ein weiterer Nachteil der Intervalle in Mittelpunkt-Radius Darstellung ist die Repräsentation der beiden Intervalle $(-\infty\,,\,a]$ und $[a\,,\,+\infty)$. Beide Intervalle besitzen den identischen Radius $r = \infty$ und können demnach nicht durch eine Kodierung mit Mittelpunkt und Radius dargestellt werden. Auch ist der Mittelpunkt für diese Intervalle nicht definierbar. Jede endliche Zahl $m$ würde durch den unendlichen Radius $r$ auf die Menge der reellen Zahlen $\mathbb{R}$ abbilden.

Dieses Problem lässt sich nur lösen, indem man für unbeschränkte Intervalle auf eine abweichende Kodierung zurückgreift. Die Intervalle der Form $[a\,,\,a]$ oder $[a_1\,,\,a_2]$ mit $-\infty < a, a_1, a_2 < +\infty$ lassen sich über einen endlichen Mittelpunkt $m$ sowie einem endlichen Radius $r \geq 0$ eindeutig darstellen. Für die drei verschiedenen unbeschränkten Intervalle kann man somit über die Symbole $\pm\infty$ die Kodierung folgendermassen realisieren:

- $(-\infty\,,\,a] \hat{=} \langle a\,;\,-\infty\rangle$
- $[a\,,\,+\infty) \hat{=} \langle a\,;\,+\infty\rangle$
- $(-\infty\,,\,+\infty) \hat{=} \langle -\infty\,;\,+\infty\rangle$

Jedoch werden die Intervalloperationen durch die abweichende Darstellung der unbeschränkten Intervalle noch weiter erschwert.

## 4.3 Flags

Wie in den Kapitel 3.1.2 und 3.3.3 schon angesprochen, ist es bei einer Intervallauswertung sinnvoll bestimmte Flags vorzusehen, um Informationen an den Anwender weiter zu reichen. Hierbei sind bei einer losen Funktionsauswertung zwei verschiedene Ursachen für das Setzen von Flags denkbar:

**Undefiniert:** Das Flag wird immer dann gesetzt, wenn eine Funktion außerhalb ihres Definitionsbereichs ausgewertet wird. Beispiele dieser Art sind folgende Funktionen:

- $[1\,,\,1] \div [-1\,,\,1]$
- $\sqrt{[-1\,,\,1]}$

**Unstetig:** Das Flag wird gesetzt, wenn eine Funktion innerhalb ihres Definitionsbereichs ausgewertet wird, sie jedoch Unstetigkeitsstellen aufweist. Als ein Beispiel ist hier die Signumfunktion[8] zu nennen:

- $sgn([-1\,,\,1])$

Bei der Verwendung von Flags für beide Anwendungsfälle gehen die Meinungen bei Anwendern, Entwicklern und Wissenschaftlern im Bereich der Intervallarithmetik auseinander. Auf der einen Seite gibt es Softwarebibliotheken, wie etwa die Boost Intervallbibliothek [6], welche gänzlich auf Flags verzichten. Dieser Verzicht stellt jedoch eine starke Einschränkung in der Verwendbarkeit der Software dar[9]. Dementsprechend ist eine Implementierung mit Flags gewünscht. Aber auch hier gibt es unterschiedliche Ansätze. filib++ [29] und auch der Vorschlag für Intervallrechnung in der C++ Standardbibliothek [41] sehen jeweils nur ein Flag vor, welches sowohl bei undefinierter Funktionsauswertung als auch bei unstetigen Funktionen gesetzt wird. Der im Rahmen der IEEE Standardisierung der Intervallarithmetik [19] von Arnold Neumaier eingereichte Vorschlag [34] definiert jedoch beide Flags.

### 4.3.1 Speicherung

Unabhängig von der Anzahl der benötigten Flags ist es interessant ob sich, ähnlich wie bei NaIs in Kapitel 4.1.1, die jeweils benötigte Information nur über die beiden verwendeten Gleitkommazahlen eines Intervalls kodieren lässt.

Ein Ansatz ist hier die, wie bei den "Wraparound" Intervallen aus Kapitel 4.2.1 verwendete, Speicherung mit vertauschten Intervallgrenzen. Jedoch sind bei "normalen" Intervallen die Bedingungen an die Grenzwerte nicht so streng wie bei den "Wraparound" Intervallen. Bei einem "Wraparound" Intervall $(-\infty\,,\,l] \cup [r\,,\,+\infty)$ wird $l < r$ für die beiden Grenzwerte gefordert. Intervalle dieser Art lassen sich demnach durch ihre maschinelle Darstellung $\langle r\,;\,l\rangle$ immer

[8] Die Signumfunktion spiegelt das Vorzeichen des Funktionsparameters wider und ist folgendermassen definiert: $sgn(x) := \begin{cases} +1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$

[9] Klassisches Beispiel ist hier das Intervall Newtonverfahren [24] zur Bestimmung von Nullstellen.

von den restlichen Intervallen mit einer Kodierung $\langle a_1\,;\,a_2\rangle$ mit $a_1 \leq a_2$ unterscheiden.

Möchte man bei einem Intervall $[a_1\,,\,a_2]\hat{=}\langle a_1\,;\,a_2\rangle$ über vertauschen der Unter- und Obergrenze ein weiteres Intervall

$$[a_1\,,\,a_2]^*\hat{=}\langle a_2\,;\,a_1\rangle \tag{4.7}$$

mit Flag realisieren, so gilt dies nur für echte Intervalle mit $a_1 < a_2$. Für Punktintervalle ist wegen der Gleichheit

$$[a\,,\,a]\hat{=}\langle a\,;\,a\rangle\hat{=}[a\,,\,a]^* \tag{4.8}$$

folglich keine Unterscheidung zwischen Intervallen mit oder ohne Flag möglich. Ein ähnliches Problem tritt auch bei der Kodierung nach Lambov aus Kapitel 4.2.1 auf. Hier gilt wegen der negierten Speicherung der Obergrenze folgende Gleichheit:

$$[-a\,,\,-a]\hat{=}\langle -a\,;\,a\rangle\hat{=}[a\,,\,a]^* \tag{4.9}$$

Der Ansatz über das Vertauschen der Intervallgrenzen muss somit wieder verworfen werden. Jedoch bieten Intervalle mit Darstellung über Mittelpunkt und Radius die Möglichkeit, Informationen über das Vorzeichen des Radius $r$ zu speichern. Durch den strikt positiven Radius $r \geq 0$ eines beidseitig begrenzten Intervalls und durch die redundante Darstellung der Null bei IEEE 754 Gleitkommazahlen lässt sich durch das Vorzeichen des Radius $r$ ein Flag realisieren. Beidseitig begrenzte Intervalle ohne Flag speichern dann, wie gewohnt, einen Radius mit positiven Vorzeichen, Intervalle mit Flag werden mit einen negativen Radius dargestellt.

Für unbegrenzte Intervalle sowie der leeren Menge muss dann noch eine zulässige Kodierung gewählt werden. Tabelle 4.6 zeigt eine mögliche Kodierung mit Berücksichtigung von NaIs.

| | ohne Flag | mit Flag |
|---|---|---|
| $[m\,;\,r]$ | $\langle m\,;\,r\rangle$ | $\langle m\,;\,-r\rangle$ |
| $(-\infty\,,\,a]$ | $\langle a\,;\,-\infty\rangle$ | $\langle -\infty\,;\,a\rangle$ |
| $[a\,,\,+\infty)$ | $\langle a\,;\,+\infty\rangle$ | $\langle +\infty\,;\,a\rangle$ |
| $(-\infty\,,\,+\infty)$ | $\langle -\infty\,;\,+\infty\rangle$ | $\langle +\infty\,;\,-\infty\rangle$ |
| $\emptyset$ | $\langle$`NaN`$\,;\,$`NaN`$\rangle$ | $\langle -$`NaN`$\,;\,-$`NaN`$\rangle$ |
| `NaI` | $\langle -$`NaN`$\,;\,$`NaN`$\rangle$ | $\langle$`NaN`$\,;\,-$`NaN`$\rangle$ |

Tabelle 4.6: Kodierung eines Intervalls mit Flag in Mittelpunkt-Radius Darstellung

Die Nachteile und Schwächen der Mittelpunkt-Radius Darstellung bleiben hier natürlich erhalten oder werden durch den zusätzlichen Aufwand für die Kodierung des Flags noch verschärft. Weiter ist es nur möglich ein Flag vorzusehen und würde Forderungen nach mehreren unterschiedlichen Flags nicht nachkommen.

## 4.4 Arithmetische Operationen

Neben den verschiedenen Repräsentationsmöglichkeiten der Intervalle ist es auch wichtig die benötigten arithmetischen Operationen bereitzustellen. Die jeweilige Implementierung der Operationen ist aber von der gewählten Darstellung der Intervalle abhängig. In dieser Arbeit wird der Darstellung über Unter- und Obergrenze der Vorzug gegeben. Eine Darstellung über Mittelpunkt und Radius birgt, durch die aufwändigeren Operationen, mehr Nach- als Vorteile. Leere Intervalle werden über [`NaN` , `NaN`] implementiert. Der Sonderfall `NaI` sowie "Wraparound" Intervalle werden nicht beachtet.

### 4.4.1 Grundrechenarten

Durch die Kodierung mit Unter- und Obergrenze ist es relativ einfach die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division den Tabellen 4.1, 4.2, 4.3, 4.4 und 4.5 entsprechend, auf IEEE 754 konformen Systemen zu implementieren[10]. Durch die Kodierung der leeren Menge als [`NaN` , `NaN`] bedarf diese keiner speziellen Behandlung. Sie wird durch die Verwendung der entsprechenden Gleitkommaoperationen wieder als Ergebnis propagiert.

#### Addition und Subtraktion

Die Addition und Subtraktion können durch die Darstellung der leeren Menge als [`NaN` , `NaN`] analog zu den in Kapitel 2.6 angegebenen Vorschriften

$$A + B = [a_1 \triangledown\!\!\!\!+ b_1 \, , \, a_2 \triangle\!\!\!\!+ \, b_2] \tag{4.10}$$

$$A - B = [a_1 \triangledown\!\!\!\!+ b_2 \, , \, a_2 \triangle\!\!\!\!+ \, b_1] \tag{4.11}$$

umgesetzt werden. Durch die speziellen Eigenschaften der Werte $\pm\infty$ sowie `NaN` werden alle in den Tabellen 4.1 und 4.2 möglichen Fälle abgedeckt und korrekt behandelt.

#### Multiplikation

Die in Tabelle 4.3 angegebene Multiplikation kann, durch die Eigenschaften der IEEE 754 Gleitkommazahlen, auf die in Tabelle 4.7 angegebenen neun Fälle reduziert werden. Jedoch muss man die beiden Sonderfälle

**Multiplikation mit** $[0\,,\,0]$**:** Liefert immer das Intervall $[0\,,\,0]$ als Ergebnis.

**Multiplikation mit** $(-\infty\,,\,+\infty)$**:** Liefert das Intervall $(-\infty\,,\,+\infty)$ als Ergebnis, außer es wird mit $[0\,,\,0]$ multipliziert.

[10] Gleiches gilt natürlich auch für die Darstellung nach Lambov. Hier müssen die Algorithmen jedoch an die negierte Obergrenze angepasst werden.

vorher behandeln, um die nicht definierte Multiplikation $\pm 0 \cdot \pm\infty$ [37] zu verhindern.

| **Multiplikation** | $[b_1\,,\,b_2]$<br>$b_2 \leq 0$ | $[b_1\,,\,b_2]$<br>$b_1 < 0 < b_2$ | $[b_1\,,\,b_2]$<br>$b_1 \geq 0$ |
|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 \leq 0$ | $[a_2 \mathbin{\overline{\cdot}} b_2\,,\, a_1 \mathbin{\underline{\cdot}} b_1]$ | $[a_1 \mathbin{\overline{\cdot}} b_2\,,\, a_1 \mathbin{\underline{\cdot}} b_1]$ | $[a_1 \mathbin{\overline{\cdot}} b_2\,,\, a_2 \mathbin{\underline{\cdot}} b_1]$ |
| $[a_1\,,\,a_2]$, $a_1 < 0 < a_2$ | $[a_2 \mathbin{\overline{\cdot}} b_1\,,\, a_1 \mathbin{\underline{\cdot}} b_1]$ | $[\min\{a_1 \mathbin{\overline{\cdot}} b_2,\ a_2 \mathbin{\overline{\cdot}} b_1\},$<br>$\max\{a_1 \mathbin{\underline{\cdot}} b_1,\ a_2 \mathbin{\underline{\cdot}} b_2\}]$ | $[a_1 \mathbin{\overline{\cdot}} b_2\,,\, a_2 \mathbin{\underline{\cdot}} b_2]$ |
| $[a_1\,,\,a_2]$, $a_1 \geq 0$ | $[a_2 \mathbin{\overline{\cdot}} b_1\,,\, a_1 \mathbin{\underline{\cdot}} b_2]$ | $[a_2 \mathbin{\overline{\cdot}} b_1\,,\, a_2 \mathbin{\underline{\cdot}} b_2]$ | $[a_1 \mathbin{\overline{\cdot}} b_1\,,\, a_2 \mathbin{\underline{\cdot}} b_2]$ |

Tabelle 4.7: Vereinfachte Multiplikation

### Division

Die Division aus den Tabellen 4.4 und 4.5 kann auf die in den Tabellen 4.8 und 4.9 angegebenen Fälle reduziert werden. Im Gegensatz zur Multiplikation sind hier keine Sonderfälle zu beachten.

| **Division**<br>$0 \notin B$ | $[b_1\,,\,b_2]$<br>$b_2 < 0$ | $[b_1\,,\,b_2]$<br>$b_1 > 0$ |
|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 \leq 0$ | $[a_2 \mathbin{\overline{\div}} b_1\,,\, a_1 \mathbin{\underline{\div}} b_2]$ | $[a_1 \mathbin{\overline{\div}} b_1\,,\, a_2 \mathbin{\underline{\div}} b_2]$ |
| $[a_1\,,\,a_2]$, $a_1 < 0 < a_2$ | $[a_2 \mathbin{\overline{\div}} b_2\,,\, a_1 \mathbin{\underline{\div}} b_2]$ | $[a_1 \mathbin{\overline{\div}} b_1\,,\, a_2 \mathbin{\underline{\div}} b_1]$ |
| $[a_1\,,\,a_2]$, $a_1 \geq 0$ | $[a_2 \mathbin{\overline{\div}} b_2\,,\, a_1 \mathbin{\underline{\div}} b_1]$ | $[a_1 \mathbin{\overline{\div}} b_2\,,\, a_2 \mathbin{\underline{\div}} b_1]$ |

Tabelle 4.8: Vereinfachte Division mit $0 \notin B$

| **Division**<br>$0 \in B$ | $[0\,,\,0]$ | $[b_1\,,\,b_2]$<br>$b_1 < b_2 = 0$ | $[b_1\,,\,b_2]$<br>$b_1 < 0 < b_2$ | $[b_1\,,\,b_2]$<br>$0 = b_1 < b_2$ |
|---|---|---|---|---|
| $[a_1\,,\,a_2]$, $a_2 < 0$ | $\emptyset$ | $[a_2 \mathbin{\overline{\div}} b_1\,,\, +\infty)$ | $(-\infty\,,\, +\infty)$ | $(-\infty\,,\, a_2 \mathbin{\underline{\div}} b_2]$ |
| $[a_1\,,\,a_2]$, $a_1 \leq 0 \leq a_2$ | $(-\infty\,,\, +\infty)$ | $(-\infty\,,\, +\infty)$ | $(-\infty\,,\, +\infty)$ | $(-\infty\,,\, +\infty)$ |
| $[a_1\,,\,a_2]$, $a_1 > 0$ | $\emptyset$ | $(-\infty\,,\, a_1 \mathbin{\underline{\div}} b_1]$ | $(-\infty\,,\, +\infty)$ | $[a_1 \mathbin{\overline{\div}} b_2\,,\, +\infty)$ |

Tabelle 4.9: Vereinfachte Division mit $0 \in B$

### 4.4.2 Fused multiply-add

Der 2008 erneuerte Standard IEEE 754 für Gleitkommaarithmetik [38] definiert eine als Funktion vereinte Multiplikation mit anschließender Addition. Die als "fused multiply-add" bezeichnete Funktion $fma : R^3 \to R$ für ein Gleitkommasystem $R$ entspricht somit der folgenden Definition:

$$fma(x, y, z) := (x * y) + z \tag{4.12}$$

Gegenüber einer normalen Multiplikation mit anschließender Addition wird bei der Verwendung von $fma$ jedoch nur einmal am Ende der Operation gerundet. Das Ergebnis weist somit eine höhere Genauigkeit auf und kann auf einer implementierenden Rechnerarchitektur durch eine optimierte Hardwareoperation angeboten werden. Dies kann, wie bei Intel's Itanium Prozesser, zu quasi der gleichen Latenzzeit wie eine "gewöhnliche" Multiplikation führen [18].

Der Vorteil der besseren Genauigkeit und Performance kommt dann vielen mathematischen Problemen zu Gute. Man denke nur an das Skalarprodukt, die Matrizenmultiplikation oder das Auswerten von Polynomen über das Hornerschema [9]. Da die genannten Berechnungen aber auch in der Intervallrechnung verbreitet sind, wäre es natürlich wünschenswert eine Funktion $fma : \mathbb{IF}^3 \to \mathbb{IF}$ auch für Intervalle zu definieren.

Betrachtet man die Multiplikation in Tabelle 4.7 so fällt auf, dass wegen der Monotonie der Addition

$$a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d \tag{4.13}$$

die in der Tabelle außenliegenden acht Fälle ohne Probleme als "fused multiply-add" für Intervalle implementiert werden können. Die Unter- und Obergrenze wird hierbei durch die beiden Funktionen $\bigtriangledown fma$ und $\bigtriangleup fma$ mit gerichteter Rundung auf Gleitkommaebene berechnet. In der folgenden Gleichung wird die Equivalenz[11] von $fma : \mathbb{IF}^3 \to \mathbb{IF}$ zur normalen Multiplikation mit anschließender Addition für den Fall $a_2, b_2 \leq 0$ gezeigt.

$$\begin{aligned}
& [a_1\,,\, a_2] \cdot [b_1\,,\, b_2] + [c_1\,,\, c_2] \\
\Leftrightarrow\ & [a_2 \mathbin{\bigtriangledown\!\!\cdot} b_2\,,\, a_1 \mathbin{\bigtriangleup\!\!\cdot} b_1] + [c_1\,,\, c_2] \\
\Leftrightarrow\ & [(a_2 \mathbin{\bigtriangledown\!\!\cdot} b_2) \mathbin{\bigtriangledown\!\!+} c_1\,,\, (a_1 \mathbin{\bigtriangleup\!\!\cdot} b_1) \mathbin{\bigtriangleup\!\!+} c_2] \\
\Leftrightarrow\ & [\bigtriangledown fma(a_2, b_2, c_1)\,,\, \bigtriangleup fma(a_1, b_1, c_2)] \\
\Leftrightarrow\ & fma([a_1\,,\, a_2], [b_1\,,\, b_2], [c_1\,,\, c_2])
\end{aligned} \tag{4.14}$$

Dies entspricht der ersten Zelle der Multiplikationstabelle 4.7 mit der Bedingung $a_2, b_2 \leq 0$. Die anderen sieben äußeren Zellen der Tabelle können analog behandelt werden[12].

[11] Das unter Umständen genauere Ergebnis durch verwenden von $fma$ auf Gleitkommaebene wird hier nicht berücksichtigt.

[12] Genau wie bei der Multiplikation müssen hier die beiden Sonderfälle $[0\,,\, 0]$ und $(-\infty\,,\, +\infty)$vorher abgefangen werden.

Aber auch die Minimum- und Maximumfunktionen werden durch die Monotonie der Addition nicht verletzt. Demnach gilt auch für den letzten Fall mit $a_1, b_1 < 0 < a_2, b_2$ im Inneren der Tabelle die Equivalenz.

$$\begin{aligned}
& [a_1\,,\,a_2] \cdot [b_1\,,\,b_2] + [c_1\,,\,c_2] \\
\Leftrightarrow\quad & \begin{array}{l}[\min\{a_1 \bigtriangledown b_2,\ a_2 \bigtriangledown b_1\}, \\ \quad \max\{a_1 \bigtriangleup b_1,\ a_2 \bigtriangleup b_2\}]\end{array} + [c_1\,,\,c_2] \\
\Leftrightarrow\quad & \begin{array}{l}[\min\{a_1 \bigtriangledown b_2,\ a_2 \bigtriangledown b_1\} \bigtriangledown c_1, \\ \quad \max\{a_1 \bigtriangleup b_1,\ a_2 \bigtriangleup b_2\} \bigtriangleup c_2]\end{array} \qquad (4.15) \\
\Leftrightarrow\quad & \begin{array}{l}[\min\{\bigtriangledown fma(a_1, b_2, c_1),\ \bigtriangledown fma(a_2, b_1, c_1)\}, \\ \quad \max\{\bigtriangleup fma(a_1, b_1, c_2\ \bigtriangleup fma(a_2, b_2, c_2)\}]\end{array} \\
\Leftrightarrow\quad & fma([a_1\,,\,a_2], [b_1\,,\,b_2], [c_1\,,\,c_2])
\end{aligned}$$

"Fused multiply-add" kann somit auch für Intervalle implementiert werden, wenn die verwendete Rechnerarchitektur $fma$ für Gleitkommazahlen nach IEEE 754 Standard mit den beiden gerichteten Rundungen $\bigtriangledown$ und $\bigtriangleup$ bereitstellt. Dies kann bei Berechnungen wie dem Skalarprodukt oder dem Hornerschema zu genaueren Ergebnissen führen. Auch ist auf entsprechenden Rechnerarchitekturen, wie beispielsweise Intel Itanium, für die acht der Gleichung (4.14) entsprechenden Fälle mit einer besseren Performance zu rechnen. Dies konnte jedoch in Ermangelung entsprechender Hardware nicht untersucht werden. Für den letzten Fall in Gleichung (4.15) ist es dann natürlich entscheident, wie sich die Funktion $fma$ für Gleitkommazahlen im Vergleich zur normalen Multiplikation und Addition performancemässig schlägt.

Für bestimmte Berechnungen, wie etwa der Intervallmatrizenmultiplikation mit rein positiven Werten, genügt es aber auf die acht Fälle aus (4.14) zurückzugreifen. Die Operationen können somit Eins zu Eins mit $fma$ auf Gleitkommaebene umgesetzt werden.

### 4.4.3 Behandlung von disjunkten Ergebnissen bei der Division

Ein Problem der erweiterten Intervallrechnung mit loser Funktionsauswertung ist das Überschätzen bei einigen Funktionen. Der klassische Fall ist hier die Division durch ein Intervall das die Zahl 0 enthält. Wie bei den "Wraparound" Intervallen schon gezeigt, gilt mit loser Funktionsauswertung folgende Gleichung:

$$[100\,,\,100] \div [-1\,,\,1] = (-\infty\,,\,-100] \cup [100\,,\,+\infty) \qquad (4.16)$$

Das einschließende Intervall $(-\infty\,,\,+\infty)$ überschätzt hierbei um $(-100\,,\,100)$ und kann in einigen Anwendungsfällen eine signifikante Verfälschung des Ergebnisses darstellen.

Für eine Implementierung der Intervallarithmetik gibt es neben den in Kapitel 4.2.1 vorgestellten "Wraparound" Intervallen weitere Möglichkeiten, um mit diesen Problem umzugehen.

**Flag:** Wird bei der Division eine Menge der Form $(-\infty\,,\,l]\cup[r\,,\,+\infty)]$ als Ergebnis berechnet, so liefert die Funktion das einschließende Intervall $(-\infty\,,\,+\infty)$ als Ergebnis zurück. Gleichzeitig wird ein Flag gesetzt um den Benutzer über die Überschätzung zu informieren. Der Benutzer kann dann auf dieses Flag eingehen und wenn nötig Funktionsaufrufe mit Teilintervallen durchführen um genauere Ergebnisse zu berechnen.

**Zwei Methoden:** Für die Division stehen zwei zusätzliche Implementierungen

- `Interval divL(Interval a, Interval b)`
- `Interval divR(Interval a, Interval b)`

zur Verfügung. Bei einer Division durch ein Intervall $[b_1\,,\,b_2]$ mit $b_1 < 0 < b_2$ liefert `divL` dann das negative Intervall $(-\infty\,,\,l]$ mit $l < 0$, `divR` das positive Intervall $(-\infty\,,\,r]$ mit $r > 0$.

In den anderen Fällen berechnet `divL` das komplette Ergebnis der Division und `divR` gibt die leere Menge zurück. Der Anwender kann somit bei Anwendungen, wie dem Intervall Newtonverfahren [24], durch verwenden der beiden Funktionen `divL` und `divR` die Stetigkeit der Auswertung sicherstellen.

**Eine Methode mit speziellen Rückgabewert:** Alternativ zu den zwei Funktionen `divL` und `divR` ist auch ein Zusammenschluß zu einer Methode

- `(Interval, Interval) div(Interval a, Interval b)`

denkbar. Hier werden die Ergebnisse von `divL` und `divR` in einem Feld zusammengefasst. Der Anwender kann dann über die jeweiligen Zellen des Feldes auf die entsprechenden Ergebnisse zugreifen.

In einer C++ Implementierung kann bei dem letzten Verfahren ein eigener Typ `DivResult` als spezieller Rückgabewert für die Division implementiert werden. Die Klasse `DivResult` enthält wie oben beschrieben die beiden disjunkten Ergebnisse sowie die entsprechenden Zugriffsmethoden

- `Interval DivResult::getL()`
- `Interval DivResult::getR()`

auf die beiden Intervalle. Listing 4.1 zeigt die Verwendung von `DivResult` mit Zugriff auf beide Intervalle.

```
1 Interval a(100, 100), b(-1, 1);
2
3 DivResult x = a / b;
4
5 Interval l = x.getL();
6 Interval r = x.getR();
```

Listing 4.1: Verwendung von `DivResult`

Durch bereitstellen eines speziellen Zuweisungsoperators

- `Interval& Interval::operator=(DivResult const& dr)`

und eines Konstruktors

- `Interval::Interval(DivResult const& dr);`

in der Klasse `Interval`, kann eine automatische Zuweisung des Typs `DivResult` an den Datentyp `Interval` realisiert werden. Hierbei wird für die beiden disjunkten Intervalle aus `DivResult` ein einschließendes Intervall erzeugt. Die Division kann somit bei einer zu vernachlässigenden Überschätzung wie gewohnt verwendet werden. Listing 4.2 zeigt dies am Beispiel der Gleichung 4.16. Das Ergebnis `x` entspricht hierbei dem Intervall $(-\infty, +\infty)$.

```
1 Interval a(100, 100), b(-1, 1);
2
3 Interval x = a / b;
```

Listing 4.2: Verwendung von `DivResult` über den Zuweisungsoperator

# Kapitel 5

# Implementierung mit C++

Bei der Sprache C++ [48] handelt es sich um eine mächtige und flexible Programmiersprache. Viele der in C++ bereitgestellten Techniken und Konzepte machen sie bei der Implementierung der Intervallarithmetik zur bevorzugten Sprache. Vor allem das Konzept der Expression Templates [50] bietet einige Möglichkeiten zur Reduzierung von teuren Rundungswechseln.

## 5.1 Rundungsstrategien

Eines der größten Probleme der Intervallarithmetik auf Rechnern ist die Verwendung von Gleitkommaoperationen mit direkter Rundung. Der IEEE Standard für Gleitkommazahlen [37, 38] definiert zwar die beiden notwendigen Rundungen

- Rundung gegen $-\infty$
- Rundung gegen $+\infty$

jedoch wird die Art der Implementierung nicht vorgeschrieben. So ist es auf vielen Rechnerarchitekturen, wie zum Beispiel der weit verbreiteten X86 Architektur, nicht möglich eine arithmetische Operation direkt mit einer der zur Verfügung stehenden Rundungen auszuführen. Vielmehr muss die gewünschte Rundung global gesetzt werden und hat dementsprechend Einfluss auf im Anschluss folgende arithmetische Operationen [2, 22].

Das globale Setzen der Rundungen auf Hardwareseite wäre soweit nicht das Problem. Dies könnte man vor dem Benutzer verbergen und zur Compilezeit lösen. Jedoch muss auf vielen Architekturen bei einem Wechsel der Rundung die Pipeline der FPU (Floating Point Unit) geleert werden [21], was hohe Latenzzeiten mit sich bringt. Zusätzlich können Intervalloperationen mit Rundungswechsel auch nicht von den Performancevorteilen des Pipelining [8] profitieren. Die Ausführung einer Intervalladdition $[a_1\,,\,a_2] + [b_1\,,\,b_2] = [a_1 \bigtriangledown b_1\,,\,a_2 \bigtriangleup b_2]$ ist deshalb um ein vielfaches langsamer als zwei Additionen im gleichen Rundungsmodus.

Beachtet man auch noch den Umstand, dass viele Funktionen der Gleitkommaarithmetik die standardmässige Rundung zur nächsten Zahl für ein ordnungsgemässes Verhalten voraussetzen, so erkennt man schnell, dass der Wechsel der Rundungsmodi einen kritischen Punkt in der Implementierung der Intervallarithmetik darstellt. Hier ist es zwingend notwendig die Rundung zwischen Intervalloperationen und Gleitkommaoperationen ständig in den benötigten Zustand zu setzen.

Mit diesen Voraussetzungen wird aber eine Implementierung der Intervallarithmetik als Softwarebibliothek für einen Entwickler zu einem Spagat zwischen Performance und Benutzerfreundlichkeit. Man kann die Software auf eine gute Performance ausrichten und auf automatische Wechsel des Rundungsmodus verzichten. Dieser Weg wurde zum Beispiel auch in der Boost Intervallbibliothek [6] über die "unprotected Rounding Policy" oder in filib++ [29] über die Rundungsstrategie `native_onesided_global` realisiert. Jedoch muss sich hier der Benutzer selbst um das Setzen des benötigten Rundungsmodus kümmern, was zum einen unkomfortabel und zum anderen fehlerträchtig ist.

Aus diesem Grund verwenden die beiden Bibliotheken standardmässig einen Modus, welcher nach jeder Intervalloperation die Rundung wieder auf den gleichen Rundungsmodus wie vor der Operation zurücksetzt. Dies ist zwar für den Benutzer sehr komfortabel, da er sich nicht um eine korrekte Rundung kümmern muss. Jedoch geht dies stark auf Kosten der Laufzeit, da viele unnötige Rundungswechsel durchgeführt werden. Betrachtet man den Intervallausdruck

$$[a_1\,,\,a_2] + [b_1\,,\,b_2] \tag{5.1}$$

so muss dieser durch die Software in Einzelbefehle zerlegt werden, um auf einer Rechnerarchitektur wie dem X86 mit Hilfe der Gleitkommaoperationen berechenbar zu sein. Der Ausdruck (5.1) für eine Auswertung mit beidseitiger Rundung kann grob durch folgenden Pseudocode dargestellt werden.

```
1 store_rounding();
2 set_rounding_downward();
3 r1 = a1 + b1;
4 set_rounding_upward();
5 r2 = a2 + b2;
6 reset_rounding();
```

Listing 5.1: Addition von zwei Intervallen mit beidseitiger Rundung

Wichtig sind bei dieser Auswertung vor allem die beiden Befehle `store_rounding()` in der ersten Zeile sowie `reset_rounding()` am Ende der Auswertung. Über `store_rounding()` wird zuerst der aktuelle Rundungsmodus vor der Ausführung des Intervallausdrucks zwischengespeichert und kann nach der Auswertung durch `reset_rounding()` wieder auf den alten Zustand zurückgesetzt werden. Dieses Vorgehen stellt sicher, dass durch die Auswertung von

Intervallausdrücken mit Hilfe von direkten Rundungen gegen $-\infty$ oder $+\infty$ andere Gleitkommaoperationen nach der Intervallauswertung nicht beeinträchtigt werden und wieder der ursprünglich global gesetzte Rundungsmodus verwendet wird[1]. Die Intervallauswertung findet quasi in einer abgesicherten Umgebung statt und die Rundung zur Berechnung der unteren Grenze $r_1 = a_1 \bigtriangledown b_1$ kann über `set_rounding_downward()` gegen $-\infty$ gesetzt werden. Nach Berechnung der unteren Grenze folgt dann, analog für die Berechnung der oberen Grenze $r_2 = a_2 \bigtriangleup b_2$, der Rundungswechsel gegen $+\infty$ über den Befehl `set_rounding_upward()`.

Diese Umsetzung einer Intervalloperation kann in einer Programmiersprache wie C++ problemlos durch Operatorüberladung realisiert werden. Der Benutzer kann die Intervallarithmetik somit auf gleiche Weise wie die Gleitkommaarithmetik verwenden, jedoch auf Kosten einer schlechteren Performance. Alleine für die Addition von zwei Intervallen wären, wie in Listing 5.1 dargestellt, drei Rundungswechsel mit jeweiliger Leerung der Pipeline nötig[2]. Und bei längeren Intervallausdrücken wie

$$[a_1\,,\,a_2] + [b_1\,,\,b_2] + [c_1\,,\,c_2] \tag{5.2}$$

nimmt, da die Codeblöcke aus Listing 5.1 im Prinzip nur mehrmals hintereinander ausgeführt werden, die Anzahl der Rundungswechsel linear mit der Anzahl der Intervalloperationen zu. Der folgende Pseudocode verdeutlicht dies für die Addition von drei Intervallen anhand der Auswertung des Ausdrucks (5.2).

```
1  store_rounding();
2  set_rounding_downward();
3  t1 = a1 + b1;
4  set_rounding_upward();
5  t2 = a2 + b2;
6  reset_rounding();
7  store_rounding();
8  set_rounding_downward();
9  r1 = t1 + c1;
10 set_rounding_upward();
11 r2 = t2 + c2;
12 reset_rounding();
```

Listing 5.2: Addition von drei Intervallen mit beidseitiger Rundung

Für die Addition von drei Intervallen werden schon insgesamt sechs Rundungswechsel durchgeführt und allgemein sind für einen Intervallausdruck mit $n$ Operationen $3n$ Rundungswechsel notwendig.

[1] filib++ speichert den ursprünglichen Zustand jedoch nicht und setzt immer auf den Standardrundungsmodus zurück.

[2] Unter der Voraussetzung, dass vor der Operation keine gerichtete Rundung $\bigtriangledown$ oder $\bigtriangleup$ verwendet wurde.

Besserung schafft hier die Intervallauswertung mit einseitiger Rundung durch anwenden der Rechenregel $\triangle a = -(\triangledown - a)$ aus Satz 8, um sich pro Operation einen Rundungswechsel zu sparen. Die Anzahl der Rundungswechsel wird hierdurch bei $n$ Operationen auf $2n$ reduziert. Der folgende Pseudocode zeigt dies für den Ausdruck (5.2).

```
1  store_rounding();
2  set_rounding_downward();
3  t1 = a1 + b1;
4  t2 = -(-a2 - b2);
5  reset_rounding();
6  store_rounding();
7  set_rounding_downward();
8  r1 = t1 + c1;
9  r2 = -(-t2 - c2);
10 reset_rounding();
```

Listing 5.3: Addition von drei Intervallen mit einseitiger Rundung

Aber auch diese Optimierung benötigt noch $2n$ Wechsel des Rundungmodus und hat mit der einfachen Realisierung aus Listing 5.2 einige völlig nutzlose Codezeilen gemeinsam. Betrachtet man die beiden Codes aus Listing 5.2 und 5.3, so fallen die identischen Codeblöcke

```
1 reset_rounding();
2 store_rounding();
3 set_rounding_downward();
```

Listing 5.4: Codeabschnitt mit Optimierungspotential

auf. Diese Codeblöcke treten immer zwischen zwei Intervalloperationen in einem Ausdruck auf und machen nichts anderes als durch `reset_rounding()` in den durch `store_rounding()` vorher gespeicherten, ursprünglichen Rundungsmodus zu wechseln. Anschließend wird der gleiche Zustand wieder über `store_rounding()` gesichert und die Rundung durch `set_rounding_downward()` gegen $-\infty$ gesetzt.

Bei der Implementierung mit beidseitiger Rundung in Listing 5.2 sind somit das `reset_rounding()` sowie das `store_rounding()` völlig nutzlos, da man sich, unabhängig davon ob die beiden Befehle ausgeführt wurden, nach dem `set_rounding_downward()` in identischen Zuständen befindet. In Listing 5.3 kann durch die einseitige Rundung auch auf `set_rounding_downward()` verzichtet werden.

Die unnötigen Rundungswechsel aus Codeabschnitt 5.4 bieten somit ein großes Potential für Optimierungsversuche. Jedoch sollten diese Optimierungen vor dem Benutzer weitestgehend versteckt werden. Dies kann in C++ über Expression Templates realisiert werden.

### 5.1.1 Intervallarithmetik mit Expression Templates

Bei Expression Templates handelt es sich um eine mächtige C++ Technik, welche es erlaubt den Compiler zur Übersetzungszeit als Interpreter zu verwenden, um damit optimierten Programmcode zu erzeugen [51]. Klassische Anwendungsgebiete für Expression Templates sind die Schleifenfusion [50] oder die Berechnung genauerer Skalarprodukte [30].

In der Intervallarithmetik kann man durch den Einsatz von Expression Templates den exzessiven Gebrauch der laufzeitintensiven Rundungswechsel einschränken ohne dabei den Benutzer durch eine komplizierte Software zu belasten. Vielmehr ist es möglich die Intervallarithmetik mit Expression Templates in gleicher Weise wie die Gleitkommaarithmetik zu verwenden. Der nachfolgende C++ Code zeigt die Addition von drei Intervallen mit Hilfe von Expression Templates.

```
1 Interval a(1.1, 2.3);
2 Interval b(2.7, 5.3);
3 Interval c(1.1, 1.3);
4
5 Interval r = a + b + c;
```

Listing 5.5: Addition von drei Intervallen in C++

Für den Benutzer unterscheidet sich, bis auf die Instanzierung, die Addition von drei Intervallen in Listing 5.5 nicht von der Addition von drei Gleitkommawerten in Listing 5.6.

```
1 double a = 1.1;
2 double b = 2.7;
3 double c = 1.3;
4
5 double r = a + b + c;
```

Listing 5.6: Addition von drei Double-Werten in C++

Die zur Übersetzungszeit durchgeführte Codeerzeugung mittels Expression Templates bleibt vom Benutzer genauso verborgen wie die Rundungswechsel, welche durch einfache Operatorüberladung in filib++ oder der Boost Intervallbibliothek zu Ergebnissen wie in Listing 5.2 oder 5.3 führen. Jedoch besteht mit Expression Templates die Möglichkeit, die unnötige Rundungswechsel aus Listing 5.4 zu vermeiden und bei der Auswertung mit beidseitiger Rundung sogar die Anzahl der restlichen Rundungen zu reduzieren.

Doch wie kann aus dem Ausdruck `Interval r = a + b + c`, mit Hilfe von Expression Templates, effizienter Code für die Berechnung erzeugt werden? Ausschlaggebend ist hier, die Operatoren nicht sequentiell über Operatorüberladung

auszuwerten, sondern für den Ausdruck zur Übersetzungszeit eine Baumdarstellung zu erzeugen und anschließend für den ganzen Ausdrucksbaum das Ergebnis zu berechnen. Dieser Ansatz ähnelt somit dem aus der objektorientierten Programmierung bekannten Interpreter Muster [15]. Jedoch wird hier, im Gegensatz zum objektorientierten Ansatz, der Baum zur Übersetzungszeit erstellt. Durch Compileroptimierungen und Inlining [49] kann effizienter Programmcode für die Auswertung erzeugt werden. Abbildung 5.1 zeigt die Baumstruktur für den Ausdruck `a + b + c`.

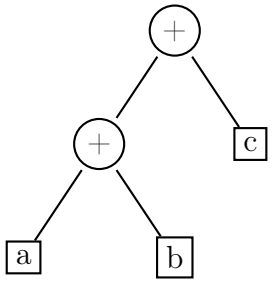



Abbildung 5.1: Baumdarstellung des Ausdrucks `a + b + c`

Der Vorteil des Ausdrucksbaums ist die Möglichkeit bei der Auswertung des Baumes das Besucher Muster [15] anzuwenden, um die jeweiligen Operationen sowie Wechsel des Rundungsmodus mit einer Flußanalyse zu optimieren.

So muss bei einer Intervallauswertung mit beidseitiger Rundung nicht zwingend die untere Grenze zuerst berechnet werden. Die einzige Voraussetzung für eine Auswertung mit beidseitiger Rundung ist, dass die unteren Grenzen mit Rundung gegen $-\infty$ und die oberen Grenzen mit Rundung gegen $+\infty$ berechnet werden.

Wertet man also den Ausdruck `Interval r = a + b + c` bei der Zuweisung an `r` mit Hilfe des Ausdrucksbaums aus Abbildung 5.1 in Postorder Reihenfolge [52] aus, so kann zu Beginn der Auswertung, wie in den Beispielen mit einfacher Operatorüberladung, der alte Rundungsmodus gespeichert werden, um dann die Rundung für die Intervallauswertung zu initialisieren. Bei der, im Anschluss durchgeführten, Auswertung des Ausdrucksbaums kann bei der Traversierung des Baums eine Information über den aktuellen Rundungsmodus an die einzelnen Knoten weitergereicht werden. Durch diese Information kann die durchzuführende Operation in einem Knoten durch Fallunterscheidung zuerst die untere oder obere Grenze berechnen, ohne dabei die Rundung zu ändern. Anschließend wird die Rundung für die Berechnung der anderen Grenze entsprechend gesetzt. Für die weitere Auswertung wird die Information über den neuen Rundungsmodus an den restlichen Baum weitergereicht. Am Ende der Auswertung kann dann wieder auf den gespeicherten, ursprünglichen Rundungsmodus zurückgesetzt werden. Die ausgeführten Rundungswechsel und arithmetischen Operationen entsprechen dann vom Ablauf dem folgenden Pseudocode.

```
1 store_rounding();
2 set_rounding_downward();
3 t1 = a1 + b1;
4 set_rounding_upward();
5 t2 = a2 + b2;
6 r2 = t2 + c2;
7 set_rounding_downward();
8 r1 = t1 + c1;
9 reset_rounding();
```

Listing 5.7: Optimierte Addition von drei Intervallen mit beidseitiger Rundung

Pro Operation müssen demnach nur ein Rundungswechsel sowie die beiden Rundungswechsel zu Beginn und am Ende der Auswertung, durchgeführt werden[3]. Die Anzahl der Rundungswechsel wird für eine Intervallauswertung mit beidseitiger Rundung bei $n$ Operationen von $3n$ auf $n + 2$ reduziert.

Noch effizienter ist die Intervallauswertung mit einseitiger Rundung. Hier muss nur zu Beginn der Auswertung die Rundung gegen $-\infty$ und am Ende wieder auf den ursprünglichen Modus gesetzt werden.

```
1 store_rounding();
2 set_rounding_downward();
3 t1 = a1 + b1;
4 t2 = -(-a2 - b2);
5 r1 = t1 + c1;
6 r2 = -(-t2 - c2);
7 reset_rounding();
```

Listing 5.8: Optimierte Addition von drei Intervallen mit einseitiger Rundung

Demnach können mit Hilfe von Expression Templates die Rundungswechsel bei Intervallausdrücken mit $n$ Operationen auf konstant zwei reduziert werden. Tabelle 5.1 zeigt hier die Vorteile der Intervallauswertung mit Expression Templates im Vergleich zur sequentiellen Auswertung mit Operatorüberladung.

| | Einfache Operatorüberladung | Expression Templates |
|---|---|---|
| Beidseitige Rundung | $3n$ | $n + 2$ |
| Einseitige Rundung | $2n$ | 2 |

Tabelle 5.1: Anzahl der Rundungswechsel bei Intervallausdrücken mit $n$ Operationen

[3]Eine weitere Optimierung durch hochziehen des Befehls `r1 = t1 + c1;` von Zeile 8 in Zeile 4 ist nicht für alle Operationen möglich und wird deshalb nicht beachtet.

Nachdem die Vorteile für eine Implementierung mit Expression Templates sprechen, muss natürlich auch die Realisierung geklärt werden. Hierbei beschränken wir uns nur auf die notwendigen Funktionalitäten, um nicht vom eigentlichen Konzept abzulenken. Die folgende Beispielimplementierung zeigt deshalb nur das Konzept der Intervallauswertung mit Expression Templates anhand der Addition. Andere Operationen können analog implementiert werden. Auch decken die gezeigten Methoden nicht alle an einen Datentyp `Interval` gestellten Anforderungen, wie etwa die Überprüfung der Grenzwerte bei einer Instanzierung, ab.

### Datentyp `Interval`

Für eine Beispielimplementierung genügt es vollkommen ein Intervall wie im nachfolgenden Listing zu implementieren.

```
1  class Interval {
2    private:
3      double _inf;
4      double _sup;
5
6    public:
7
8      Interval(double inf, double sup)
9          : _inf(inf), _sup(sup) {}
10
11     double inf() const {
12         return _inf;
13     }
14
15     double sup() const {
16         return _sup;
17     }
18 };
```

Listing 5.9: Klasse `Interval`

Die Intervallklasse enthält somit nur die Untergrenze `_inf` sowie die Obergrenze `_sup` vom Gleitkommatyp `double`. Bei einer Implementierung für den produktiven Einsatz wäre hier eine generische Lösung mit einen Templateparameter `N` für den Datentyp der Grenzen sinnvoller. Jedoch erschwert dies die Lesbarkeit des restlichen Codes erheblich.

### Baumrepräsentation

Wie in Kapitel 5.1.1 angesprochen wird bei Expression Templates aus einem Ausdruck ein Ausdrucksbaum aufgebaut und dieser am Stück ausgewertet. Nimmt

man als Beispiel die Addition von zwei Intervallen $a + b$, so kann man sich den Baum wie in Abbildung 5.2 vorstellen. Jeweils ein Intervall als Blatt des Baumes

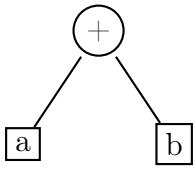



Abbildung 5.2: Ausdrucksbaum für die Addition von zwei Intervallen $a + b$

sowie die Addition als inneren Knoten. Man benötigt neben `Interval` also noch eine weitere Klasse für die Addition als inneren Knoten. Dies leistet die Klasse `BinaryIntervalExpr<OP, E1, E2>`.

```
1  template<typename OP, typename E1, typename E2>
2  class BinaryIntervalExpr {
3    private:
4      typedef EvalTraits<E1>  et1;
5      typedef EvalTraits<E2>  et2;
6
7      E1 _expr1;
8      E2 _expr2;
9
10   public:
11     BinaryIntervalExpr(E1 expr1, E2 expr2)
12         : _expr1(expr1), _expr2(expr2) {}
13
14     Interval eval(RndControl& rnd) const {
15         return OP::eval(rnd, et1::eval(rnd, _expr1),
16             et2::eval(rnd, _expr2));
17     }
18 };
```

Listing 5.10: Klasse `BinaryIntervalExpr<OP, E1, E2>`

Hier wird bewußt keine spezielle Klasse `Add` zur Addition von zwei Intervallen sondern eine allgemeine generische Klasse für eine zweistellige Operation implementiert. Die Gründe hierfür sind naheliegend. Zum einen wird man in einer Implementierung für den produktiven Einsatz mehrere zweistellige Operationen wie Subtraktion, Multiplikation oder Division benötigen. Zum anderen wird man nicht nur Ausdrücke mit genau zwei Intervallen verarbeiten. Bei einem Ausdruck wie `a + b + c` könnte man noch den Teilbaum für `a + b`, wie in Abbildung 5.2, mit dieser speziellen Klasse realisieren. Jedoch wäre die Addition dieses Teilbaums mit dem Intervall `c` nicht möglich, da zwei Intervalle als Nachfolgeknoten erwartet werden. Man müsste demnach mehrere verschiedene Implementierungen von

`Add` realisieren oder wie im Interpreter Muster [15] mit Vererbung und virtuellen Methoden arbeiten.

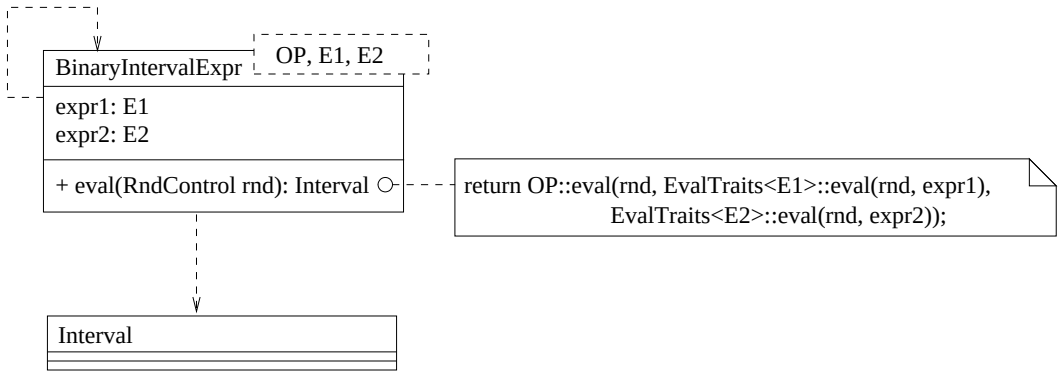



Abbildung 5.3: Klasse `BinaryIntervalExpr<OP, E1, E2>`

Die Klasse `BinaryIntervalExpr<OP, E1, E2>` geht hier einen eleganteren, und durch den Verzicht auf virtuelle Methoden auch effizienteren, Weg. So muss durch die als Templateparameter `OP` übergebene Policy Klasse [4] das Konzept der inneren Knoten nur einmal für alle zweistelligen Operationen implementiert werden. Die eigentlich auszuführende Operation wird als Funktorklasse übergeben und dann bei der Baumauswertung entsprechend verwendet. Einzige Voraussetzung an die übergebene Policy Klasse ist das Einhalten der Schnittstellenvereinbarungen. In diesem Beispiel ist dies lediglich das zur Verfügung stellen der folgenden Funktion für die Baumauswertung.

```
1 static Interval eval(RndControl& rnd,
2         Interval const& i1, Interval const& i2);
```

Listing 5.11: Schnittstellenvereinbarungen für Policy Klassen von zweistelligen Operationen

Diese Funktion führt, für die beiden als Referenz übergebenen Intervalle `i1` und `i2`, die Operation, wie etwa das Addieren von zwei Intervallen, aus. Das per Referenz übergebene Objekt `rnd` der Klasse `RndControl` dient dabei zur Steuerung des Rundungsmodus.

Die beiden Nachfolgeknoten, also die Operanden der auszuführenden Funktion, werden bei der Klasse `BinaryIntervalExpr<OP, E1, E2>` über die Templateparameter `E1` und `E2` ausgeprägt. Somit kann im Prinzip jeder beliebige Nachfolgeknoten verwendet werden. Um die Klasse `BinaryIntervalExpr<OP, E1, E2>` jedoch flexibel zu gestalten, wird für die Auswertung der Nachfolgeknoten eine Traits Klasse `EvalTraits<E>` verwendet.

```
1 template<typename E>
2 class EvalTraits {
3   public:
4     static inline Interval eval(RndControl& rnd,
5             E const& e) {
6         return e.eval(rnd);
7     }
8 };
```

Listing 5.12: Traits Klasse `EvalTraits<E>`

Die Klasse `BinaryIntervalExpr<OP, E1, E2>` kann die Auswertung der Nachfolger somit über die angepasste Methode `eval` der Traits Klassen durchführen. `BinaryIntervalExpr<OP, E1, E2>` benötigt somit kein spezielles Wissen über die beiden Nachfolgeklassen. Lediglich eine entsprechende Implementierung von `EvalTraits<E>` ist nötig. Die in Listing 5.12 angegebene allgemeine Implementierung kann für alle Nachfolgeklassen verwendet werden, welche folgende Schnittstellenvereinbarung einhalten.

```
1 Interval eval(RndControl& rnd) const;
```

Listing 5.13: Schnittstellenvereinbarungen an Nachfolgeknoten

Wie schon bei der Schnittstellenvereinbarung für die Policy Klassen in Listing 5.11 wird auch hier eine Referenz auf ein Objekt der Klasse `RndControl` übergeben. Somit ist es möglich ein Objekt der Klasse `RndControl` anhand des Besucher Musters [15] bei der Auswertung an jeden Knoten weiterzureichen, um damit den Rundungsmodus zu steuern. Betrachtet man jedoch den Code aus Listing 5.10 oder das Klassendiagramm aus Abbildung 5.3 so stellt man fest, dass die Klasse `BinaryIntervalExpr<OP, E1, E2>` selbst die Schnittstellenvereinbarung aus Listing 5.13 einhält, jedoch keine spezielle Traits Klasse für `Interval` angeboten wird. Die Klasse `Interval` enthält die geforderte Methode aus Listing 5.13 nicht und kann somit nicht als Nachfolgeknoten für `BinaryIntervalExpr<OP, E1, E2>` verwendet werden. Natürlich könnte man die Klasse `Interval` um die geforderte Methode erweitern. Eleganter ist jedoch eine Spezialisierung `EvalTraits<Interval>` der Traits Klasse.

```
1 template<>
2 class EvalTraits<Interval> {
3   public:
4     static inline Interval eval(RndControl& rnd,
5             Interval const& i) {
6         return i;
7     }
8 };
```

Listing 5.14: Traits Klasse `EvalTraits<Interval>`

Mit der Traits Klasse `EvalTraits<Interval>` sind nun alle vier Kombinationsmöglichkeiten aus `BinaryIntervalExpr<OP, E1, E2>` und `Interval` in einem Baum abgedeckt. Der einzige Unterschied bei der Spezialisierung zu der allgemeinen Traits Klasse ist die Auswertung. Für ein Intervall als Nachfolger findet nicht noch ein rekursiver Aufruf der Methode eval(RndControl& rnd) statt. Hier wird das Intervall gleich für den Aufruf der eigentlichen Operation verwendet.

### Steuerung der Rundung am Beispiel der Addition

In den Sprachen C und C++ ist es über die im C99 Standard [10] definierte Header-Datei `<fenv.h>` möglich, Rundungsmodi für Gleitkommazahlen des IEEE 754 Standard anzusprechen. Für diesen Zweck sind im C99 Standard die beiden Methoden

- `int fegetround()`
- `int fesetround(int round)`

zum Auslesen und Setzen des globalen Rundungsmodus implementiert. Die vier verschiedenen Rundungsmodi werden dabei über einen Integerwert identifiziert, für welchen die in Tabelle 5.2 aufgeführten Konstanten verwendet werden. Durch

| **Rundungsmodus** | **Integer-Konstante** |
|---|---|
| Rundung gegen $-\infty$ | `FE_DOWNWARD` |
| Rundung zur nächsten Zahl | `FE_TONEAREST` |
| Rundung gegen 0 | `FE_TOWARDZERO` |
| Rundung gegen $+\infty$ | `FE_UPWARD` |

Tabelle 5.2: Integer-Konstanten zum Ansprechen der Rundungsmodi

einbinden der Header Datei `<fenv.h>` stehen somit alle nötigen Funktionalitäten zur Verfügung, um den Rundungsmodus der Gleitkommazahlen für die Intervallarithmetik zu steuern. Bei der Intervallauswertung mit Expression Templates wird der Rundungsmodus aber nicht direkt über die beiden Methoden `int fegetround()` und `int fesetround(int round)` gesteuert, sondern über ein Objekt der Klasse `RndControl` gekapselt.

```
1 class RndControl {
2   private:
3     int _former;
4     int _current;
5
6   public:
7     RndControl() {
8         _former = fegetround();
```

```
9           _current = _former;
10      }
11
12      ~RndControl() {
13          setRounding(_former);
14      }
15
16      void setRounding(int rnd) {
17          if (rnd != _current
18              && fesetround(rnd) == 0)
19              _current = rnd;
20      }
21
22      int getRounding() {
23          return _current;
24      }
25  };
```

Listing 5.15: Klasse `RndControl`

Die Klasse `RndControl` hat den Vorteil, dass der ursprüngliche sowie der aktuelle Rundungsmodus in Membervariablen gespeichert werden können. Da aber das Objekt der Klasse `RndControl` bei der Baumauswertung die Rolle eines Besuchers einnimmt, stehen diese Informationen jedem Knoten des Baums während der Auswertung zur Verfügung. Hierdurch eröffnet sich der, für die Umsetzung einer Operation zuständigen, Policy Klasse die Möglichkeit, die Operation in Abhängigkeit der aktuellen Rundung durchzuführen.

Die Klasse `IntervalAdd` verdeutlicht nun auf einfache Weise die Möglichkeiten der Ausdrucksauswertung mittels Expression Templates. Durch den Zustand des übergebenen Objekts `rnd` vom Typ `RndControl` kann der Ablauf der Operation vom aktuellen Rundungsmodus abhängig gemacht werden.

```
1  class IntervalAdd {
2    public:
3      static Interval eval(RndControl& rnd,
4          Interval const& i1, Interval const& i2) {
5
6          double inf;
7          double sup;
8
9          switch(rnd.getRounding()) {
10           case FE_DOWNWARD:
11             inf = i1.inf() + i2.inf();
12             rnd.setRounding(FE_UPWARD);
```

```
13              sup = i1.sup() + i2.sup();
14              return Interval(inf,sup);
15
16            case FE_UPWARD:
17              sup = i1.sup() + i2.sup();
18              rnd.setRounding(FE_DOWNWARD);
19              inf = i1.inf() + i2.inf();
20              return Interval(inf,sup);
21
22            default:
23              rnd.setRounding(FE_DOWNWARD);
24              inf = i1.inf() + i2.inf();
25              rnd.setRounding(FE_UPWARD);
26              sup = i1.sup() + i2.sup();
27              return Interval(inf,sup);
28          }
29      }
30  };
```

Listing 5.16: Klasse `IntervalAdd`

Über die Methode `int getRounding()` des, als Besucher übergebenen, Objekts `rnd` kann geprüft werden ob sich die Rundung schon in einem der beiden gewünschten Modi `FE_DOWNWARD` oder `FE_UPWARD` befindet. Entsprechend kann dann erst die untere oder die obere Grenze berechnet werden. Der zur Berechnung der anderen Grenze notwendige Rundungsmodus wird dann ebenfalls über das Objekt `rnd` mittels der Methode `void setRounding(int rnd)` gesetzt. Diese Methode ändert dann die Rundung auf den gewünschten Modus. Speichert aber gleichzeitig auch die Information über den nun aktuellen Zustand, so dass nach Beendigung dieser Operation auch Nachfolger darauf zugreifen können.

Das Ausführen der Addition benötigt also im Idealfall nur einen Wechsel des Rundungsmodus. Einzig der Fall, dass sich die Rundung nicht in einem der beiden gewünschten Zustände befindet macht eine Berechnung mit zwei Rundungswechseln nach dem herkömmlichen Schema notwendig. Dieser Fall muss natürlich beachtet werden, da auch Intervalloperationen mit anderen Rundungsmodi nicht auszuschließen sind. Um unnötige Wechsel der Rundung zu vermeiden, sollten die bei der Auswertung verwendeten Funktorklassen nur die für die eigene Operation notwendigen Rundungswechsel durchführen.

Dieses Konzept spiegelt sich auch in der Funktionalität der Klasse `RndControl` wider. Im Konstruktor wird nur die Information über die aktuelle Rundung gespeichert. Für das Setzen des korrekten Modus ist dann die jeweilige Operation im Ausdrucksbaum zuständig. Die Klasse `RndControl` ist lediglich nach der Auswertung für das Zurücksetzen auf die ursprüngliche Rundung verantwortlich. Dies geschieht über den Destruktor beim Vernichten des Objekts. Für die Intervall-

auswertung erzeugt man demnach ein Objekt vom Typ `RndControl`, verwendet dieses Objekt für die Auswertung des Ausdrucksbaumes als Besucher Objekt und vernichtet es nach der Auswertung. Dieses Konzept zeigt die händisch durchgeführte Intervallauswertung am Beispiel der Addition von zwei Intervallen `a` und `b` im nachfolgenden Programmcode.

```
1 int main() {
2     Interval a(1.1, 2.3);
3     Interval b(2.7, 5.3);
4
5     BinaryIntervalExpr<IntervalAdd, Interval,
6         Interval> expr(a, b);
7
8     RndControl rnd;
9     Interval r = expr.eval(rnd);
10
11    return 0;
12 }
```

Listing 5.17: Beispiel für die Addition von zwei Intervallen

Entscheidend ist hierbei die Konstruktion des Ausdrucksbaums in Zeile 5. Die Addition der beiden Intervalle `a` und `b` muss wie in Abbildung 5.4 durch eine Instanz der Klasse `BinaryIntervalExpr<OP, E1, E2>` abgebildet werden. Für die

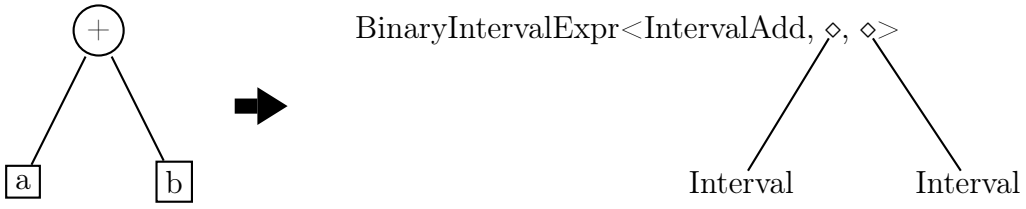



Abbildung 5.4: Templateausprägungen des Ausdrucksbaums für die Addition von zwei Intervallen

durchzuführende Addition wird die Instanz `expr` von `BinaryIntervalExpr<OP, E1, E2>` mit der Policy-Klasse `IntervalAdd` ausgeprägt. Die beiden Operanden für die Addition sind jeweils Intervalle und somit sind auch die beiden Templateparameter `E1` und `E2` jeweils vom Typ `Interval`. Das Objekt `expr` entspricht demnach dem Wurzelknoten des Ausdrucksbaums aus Abbildung 5.4 und enthält die beiden übergebenen Intervalle `a` und `b` als Blätter.

Für die Auswertung wird anschließend die lokale Instanz `rnd` der Klasse `RndControl` erzeugt, welche dem Wurzelobjekt `expr` des Ausdrucksbaums in Zeile 8 für die Steuerung der Rundung übergeben wird. In diesem Beispiel wird

nach der Auswertung des Ausdrucks die Methode `main()` und somit auch der Gültigkeitsbereich des lokalen Objekts `rnd` verlassen. Das Objekt `rnd` wird automatisch freigegeben und somit auch die Rundung über den Destruktor von `rnd` wieder auf den Ursprungszustand gesetzt. Ist dies jedoch nicht der Fall, dass mit dem Beendigen der Auswertung auch der Gültigkeitsbereich von `rnd` verlassen wird, so muss das Objekt explizit freigegeben oder der Auswertungsblock in den Zeilen 8 und 9 durch eine Klammerung { . . . } mit einem eigenen Gültigkeitsbereich versehen werden.

### Baumerzeugung

Wie bei dem in Listing 5.17 gezeigten Code ist es über Expression Templates möglich Intervallauswertungen vorzunehmen und dabei mittels eines Besucherobjekts die nötigen Rundungswechsel zu reduzieren. Nur ist es für einen Benutzer unkomfortabel und auch fehleranfällig die Ausdrucksbäume sowie deren Auswertung von Hand vorzunehmen. Man denke nur an längere Intervallausdrücke. Die hierzu nötigen, verschachtelten Templateausprägungen sind dabei nur schwer zu überblicken.

Auch das manuelle Verwalten der Instanzen vom Typ `RndControl` bringt keine Verbesserungen zur "unprotected Rounding Policy" der Boost Intervallbibliothek oder zum Modus `native_onesided_global` aus filib++.

Diese Umstände können jedoch durch Operatorüberladung sowie einer Wrapperklasse für den Ausdrucksbaum bereinigt werden. Die Klasse `IntervalExpr<E>` stellt diese Wrapperklasse dar.

```
1  template<typename E>
2  class IntervalExpr {
3    private:
4      E _expr;
5
6    public:
7      IntervalExpr(E expr)
8       : _expr(expr) {}
9
10     E& rep() {
11         return _expr;
12     }
13
14     E const& rep() const {
15         return _expr;
16     }
17
18     Interval eval() const {
19         RndControl rnd;
```

```
20          return _expr.eval(rnd);
21      }
22  };
```

Listing 5.18: Klasse `IntervalExpr<E>`

Sie kapselt den über den Templateparameter `E` parametrisierten Ausdrucksbaum und bietet mit der Methode

- `Interval eval() const`

die Möglichkeit diesen auszuwerten. Die Methode `eval()` entspricht dabei dem

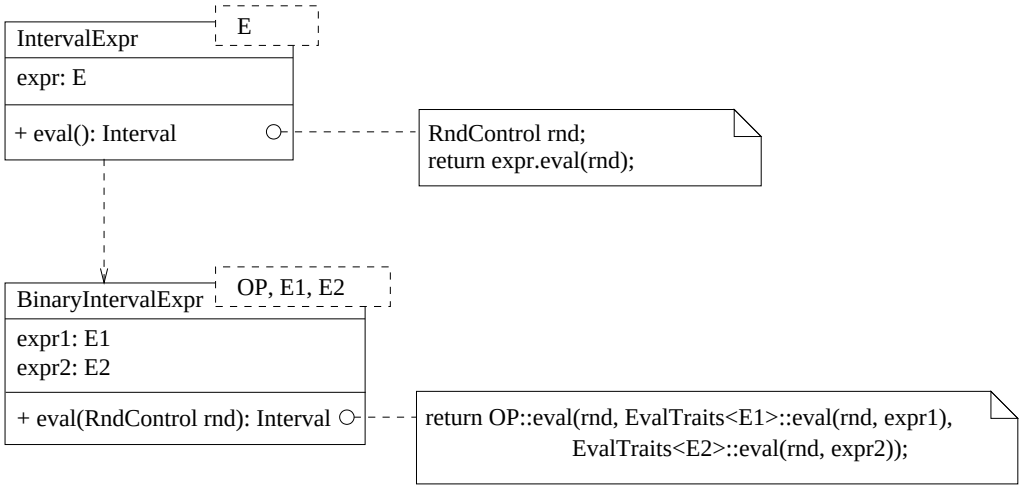



Abbildung 5.5: Klasse `IntervalExpr<E>`

Codeblock von Zeile 8 und 9 aus Listing 5.17. Der Baum wird somit mit einem eigenen Rundungsobjekt ausgewertet, welches am Ende der Auswertung durch Verlassen des Gültigkeitsbereichs wieder gelöscht und die Rundung somit zurückgestellt wird. Der Beispielcode aus Listing 5.17 kann nun wie folgt geändert werden.

```
1  int main() {
2      Interval a(1.1, 2.3), b(2.7, 5.3);
3
4      IntervalExpr<BinaryIntervalExpr<IntervalAdd, Interval,
5          Interval>> expr(BinaryIntervalExpr<IntervalAdd,
6          Interval, Interval>(a, b));
7
8      Interval r = expr.eval();
9
10     return 0;s
11 }
```

Listing 5.19: Addition von zwei Intervallen mit Hilfe der Wrapperklasse

Die Steuerung der Rundung wird durch die Wrapperklasse gekapselt und liegt somit nicht mehr in der Verantwortung des Benutzers. Jedoch ist nun die Konstruktion des Ausdrucksbaums in den Zeilen 4 bis 6 noch weiter erschwert. Implementiert man aber folgende Überladung des Operators `operator+`, so kann auch dieses komplizierte Konzept vor dem Benutzer verborgen werden.

```
1 IntervalExpr<BinaryIntervalExpr<IntervalAdd,
2 Interval, Interval>>
3 operator+(Interval const& i1, Interval const& i2) {
4     return IntervalExpr<BinaryIntervalExpr<IntervalAdd,
5         Interval, Interval>>(BinaryIntervalExpr<
6                 IntervalAdd, Interval, Interval>(i1,i2));
7 }
```

Listing 5.20: Operatorüberladung für die Addition von zwei Intervallen

Dieser Operator liefert beim Verknüpfen von zwei Objekten des Typs `Interval` exakt das gleiche Ergebnis wie die Konstruktion in den Zeilen 4 bis 6 von Listing 5.19. Problem ist aber, dass dieser Operator nur für zwei Intervalle greift. Bei einem längeren Ausdruck werden jedoch die Operatoren der Reihe nach ausgewertet und so müssen dann auch Intervalle und Ausdrucksbäume miteinander verknüpft werden. Dies leisten die weiteren Operatoren aus Listing 5.21.

```
1 template<typename E>
2 IntervalExpr<BinaryIntervalExpr<IntervalAdd,
3 Interval, E>>
4 operator+(Interval const& i, IntervalExpr<E> expr) {
5     return IntervalExpr<BinaryIntervalExpr<IntervalAdd,
6         Interval, E>>(BinaryIntervalExpr<IntervalAdd,
7             Interval, E>(i, expr.rep()));
8 }
9
10 template<typename E>
11 IntervalExpr<BinaryIntervalExpr<IntervalAdd,
12 E, Interval>>
13 operator+(IntervalExpr<E> expr, Interval const& i) {
14     return IntervalExpr<BinaryIntervalExpr<IntervalAdd,
15         E, Interval>>(BinaryIntervalExpr<IntervalAdd,
16              E, Interval>(expr.rep(), i));
17 }
18
19 template<typename E1, typename E2>
20 IntervalExpr<BinaryIntervalExpr<IntervalAdd, E1, E2>>
21 operator+(IntervalExpr<E1> expr1,
22         IntervalExpr<E2> expr2) {
```

```
23      return IntervalExpr<BinaryIntervalExpr<IntervalAdd,
24           E1, E2> >(BinaryIntervalExpr<IntervalAdd,
25                E1, E2>(expr1.rep(), expr2.rep()));
26 }
```

Listing 5.21: Operatorüberladung für die Addition mit `IntervalExpr<E>`

Der jeweilige Operator verwendet, bei den durch `IntervalExpr<E>` gekapselten Ausdrucksbäumen, nur den eigentlichen Ausdrucksbaum vom Typ `E`. Hierfür stellt die Klasse `IntervalExpr<E>` die Methode `rep()` für den Zugriff zur Verfügung. So kann durch sequentielle Anwendung der Operatoren der vollständige Ausdrucksbaum, wie in Abbildung 5.6 für die Addition von drei Intervallen verdeutlicht, als Ergebnis in einem Objekt vom Typ `IntervalExpr<E>` gekapselt werden.

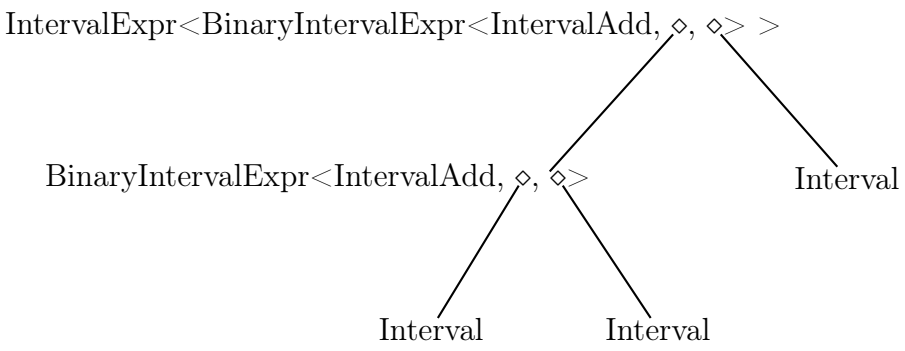



Abbildung 5.6: Templateausprägungen des Ausdrucksbaums für die Addition von drei Intervallen

### Baumauswertung

Erweitert man die Klasse `Interval` noch um einen Zuweisungsoperator sowie einen Konstruktor für Ausdrucksbäume, so ist auch eine automatische Auswertung von Ausdrücken gegeben.

```
1 template<typename E>
2 Interval(IntervalExpr<E> const& expr) {
3     Interval i = expr.eval();
4     _inf = i.inf();
5     _sup = i.sup();
6 }
7
8 template<typename E>
```

```
9  Interval& operator=(IntervalExpr<E> const& expr) {
10     Interval i = expr.eval();
11     _inf = i.inf();
12     _sup = i.sup();
13
14     return this;
15 }
```

Listing 5.22: Erweiterung der Klasse Interval

Die Addition von zwei Intervallen reduziert sich demnach auf folgenden Beispielcode.

```
1 int main() {
2     Interval a(1.1, 2.3);
3     Interval b(2.7, 5.3);
4
5     Interval r = a + b;
6
7     return 0;
8 }
```

Listing 5.23: Addition von zwei Intervallen mit Auswertung bei Zuweisung

### Zusammenfassung

Wie hier am Beispiel der Addition aufgezeigt, kann die Intervallarithmetik mit Hilfe von Expression Templates in C++ realisiert werden. Der Aufwand bei der Implementierung ist hierbei etwas größer als mit einfacher Operatorüberladung, jedoch kann durch das Konzept der Policy Klassen bei den Operationen ein großer Teil in der Funktionalität zentralisiert werden. So ist die Klasse `BinaryIntervalExpr<OP, E1, E2>` nur einmal nötig. Für weitere zweistellige Operationen wie Subtraktion, Division oder Multiplikation aber auch andere beliebige zweistellige Funktionen sind dann nur eine spezielle Funktorklasse sowie eine eigene Operatorüberladung oder spezielle Funktionen zur Konstruktion des Baums nötig.

Analog zur Implementierung von zweistelligen Operationen mit der Klasse `BinaryIntervalExpr<OP, E1,E2>`, kann auch eine Repräsentation von ein- oder mehrstelligen Operationen realisiert werden. Weiter ist es auch möglich die durch Expression Templates realisierten Intervalloperationen mit anderen, nicht auf Expression Templates basierenden, Methoden zu mischen.

Eine “klassisch” implementierte Methode

- `Interval sub(Interval const& i1, Interval const& i2)`

zur Subtraktion von zwei Intervallen kann beispielsweise zusammen mit der, über Expression Templates realsierten, Addition verwendet werden.

```
1 int main() {
2     Interval a(1.1, 2.3);
3     Interval b(2.7, 5.3);
4     Interval c(7.5, 2.9);
5     Interval d(1.8, 1.9);
6
7     Interval r = a + sub(b, c + d);
8
9     return 0;
10 }
```

Listing 5.24: Mischen von Expression Templates und "klassischen" Methoden

Hier werden dann nur die entsprechenden Teilausdrücke über Expression Templates ausgewertet. Im Beispiel aus Listing 5.24 sind dies die Addition `c + d` sowie die Addition von `a` mit dem Ergebnis der Subtraktion. Die bessere Rundungstrategie wird dann nur für die Teilausdrücke verwendet. Dies bedeutet, dass hier nach der Auswertung von `c + d` die Rundung wieder zurückgesetzt wird. Die Methode `sub` wird anschließend "klassisch" ausgeführt. Die letzte Addition kann dann wieder mit Expression Templates und verbesserter Rundungsstrategie durchgeführt werden.

Das Konzept der Intervallauswertung mit Expression Templates wurde im Rahmen dieser Arbeit sowohl mit beidseitiger Rundung, als auch mit einseitiger Rundung[4] für die Addition implementiert. Im Vergleich mit der Boost Intervallbibliothek (Standard Policy) und filib++ (`native_switched`, `native_directed`) konnte, vor allem bei der Implementierung mit einseitiger Rundung, ein großer Performancegewinn für längere Ausdrücke festgestellt werden. Ergebnisse von Laufzeitvergleichen sind in der Tabelle 5.3 sowie der Abbildung 5.7 aufgeführt. Zusätzlich ist zu erwähnen, dass für die Testimplementierung nur C++ Standardcode verwendet wurde. Optimierungen auf Assemblerebene, wie etwa bei filib++, wurden nicht angewendet.

| $n$ | ET einseitig | ET beidseitig | filib++ | filib++ native dir. | Boost |
|---|---|---|---|---|---|
| 1 | 7.44 | 11.11 | 4.41 | 3.34 | 7.73 |
| 2 | 7.56 | 14.75 | 8.65 | 6.33 | 15.37 |
| 3 | 7.76 | 18.42 | 13 | 9.53 | 23.08 |
| 4 | 7.75 | 22.41 | 17.43 | 13.07 | 30.74 |
| 5 | 8.02 | 26.16 | 21.73 | 16 | 38.34 |

Tabelle 5.3: Laufzeit für Ausdrücke mit $n$ Operationen

[4] Darstellung nach Lambov.

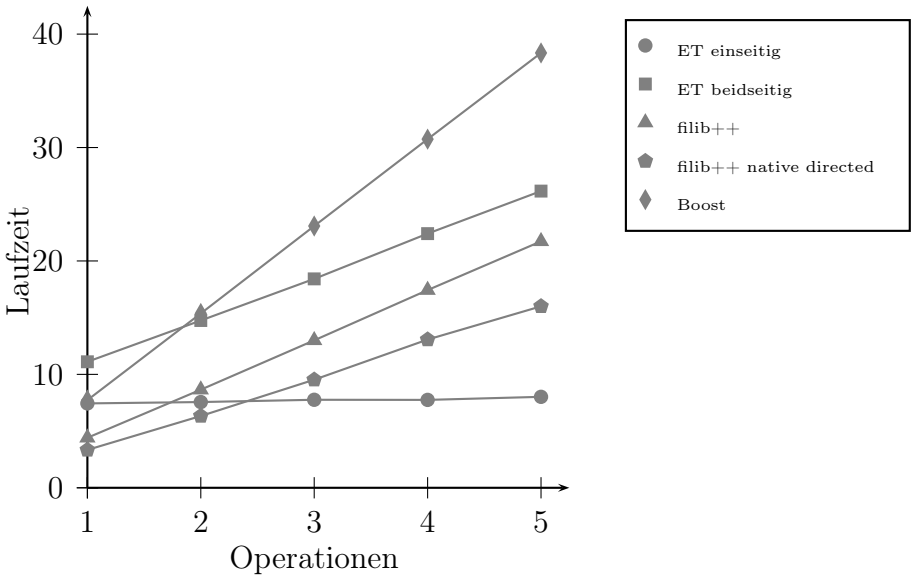



Abbildung 5.7: Laufzeit für Ausdrücke mit $n$ Operationen

Der Vorteil der Expression Templates ist somit vor allem für längere Ausdrücke gegeben. Für Ausdrücke mit wenigen Operationen ist die Laufzeit nur geringfügig besser oder sogar teilweise schlechter. Dies wirkt sich vor allem dann negativ aus, wenn viele dieser kurzen Ausdrücke berechnet werden. Hier kann man durch Erweitern der Klasse `IntervalExpr<E>` um eine zusätzliche Auswertungsmethode

```
1 Interval eval(RndControl& rnd) const {
2     return _expr.eval(rnd);
3 }
```

Listing 5.25: Zusätzliche Auswertungsmethode der Klasse `IntervalExpr<E>` für ungeschützte Rundung

eine zur "unprotected Rounding Policy" der Boost Intervallbibliothek ähnliche Rundungsstrategie anbieten. Die neue Methode entspricht hierbei der schon implementierten Methode `eval()`, jedoch wird hier auf die Kapselung durch ein Objekt des Typs `RndControl` verzichtet. Der Benutzer muss sich beim expliziten Aufruf dieser Methode selbst um die Bereitstellung eines gültigen Objekts des Typs `RndControl` kümmern.

```
1 {
2     RndControl rnd;
3     Interval a = (i1 + i2).eval(rnd);
4     Interval b = (i3 + i4).eval(rnd);
5     Interval c = (i5 + i6).eval(rnd);
6     Interval d = (i7 + i8).eval(rnd);
7 }
```

Listing 5.26: Verwendung der ungeschützten Rundung

Listing 5.26 zeigt dies am Beispiel mehrer Additionen. Wichtig ist hier, dass die Intervalloperationen durch ein eigenes Objekt vom Typ `RndControl` umschlossen werden. Dies wird im Beispiel durch einen eigenen Gültigkeitsbereich über die geschweiften Klammern erreicht. Die Intervalloperationen können somit alle von der besseren Rundungsstrategie profitieren und im Falle einer einseitigen Rundung mit zwei Wechseln des Rundungsmodus ausgeführt werden.

Die vorher implementierte automatische Auswertung von Expression Templates über den Zuweisungsoperator der Klasse `Interval` wird von der ungeschützten Rundungsstrategie nicht beeinflusst. Sie könnte durch ihre Kapselung der Auswertung auch innerhalb des Codeblocks in Listing 5.26 verwendet werden. Für normale Gleitkommaoperationen gilt dies wegen den abweichenden Rundungsmodi nicht unbedingt.

## 5.2 Flagverwaltung

Im Kapitel 3 wurden zwei verschiedene Konzepte der Flagverwaltung in Softwarebibliotheken angesprochen. Die Intervallbibliothek filib++ [29] verwendet mit einem globalen Flag ein, auf modernen Multithreading/Multicore Architekturen, schlechtes Konzept der Flagverwaltung. Das globale Flag wird nur bei einer Funktionsauswertung außerhalb des Definitionsbereichs auf `true` gesetzt. Ansonsten bleibt das Flag unangetastet. Der Wert `false` kann anschließend nur durch händisches Zurücksetzen wieder erreicht werden. Diese Technik ist sehr umständlich und auch fehlerträchtig. Ein nicht zurückgesetztes Flag kann unter Umständen einen nicht gewollten Programmablauf herbeiführen. Vor allem ist aber eine verteilte Berechnung mit diesem globalen Flag nicht möglich. Ein für mehrere Threads konsistenter Zustand kann nicht garantiert werden.

Der Vorschlag für Intervallarithmetik in der C++ Standardbibliothek [41] geht hier einen besseren Weg. Er verwendet spezielle, um einen boolschen Parameter erweiterte, Methoden zur Auswertung von partiell definierten Funktionen. Für eine Auswertung mit Flag kann der Benutzer eine Referenz einer Variable des Typs `bool` beim Methodenaufruf übergeben. Die boolsche Variable wird dann als Flag bei der losen Funktionsauswertung verwendet. Der Wert der Variable bleibt dabei ebenfalls unangetastet, solange die Auswertung den Definitionsbereich der

Funktion nicht verlässt. Findet jedoch eine Auswertung außerhalb des Definitionsbereichs statt, so wird der Wert auf `true` gesetzt. Listing 3.1 in Kapitel 3.1.2 zeigt die Verwendung dieser, durch den Benutzer zu verwaltenden, Flags an der Auswertung der Funktion $x/\sqrt{y}$. Dieses Konzept hat jedoch den Nachteil, dass, durch die um ein Flag erweiterten Methoden, die Lesbarkeit des Programmcodes stark leidet. Vergleicht man den jeweils nötigen Programmcode zur Auswertung des relativ einfachen Ausdrucks $x/\sqrt{y}$ in den beiden Listings,

```
1 interval<double> r = divide(x, sqrt(y, flag), flag);
```

Listing 5.27: Auswertung von $x/\sqrt{y}$ mit Flag

```
1 interval<double> r = x / sqrt(y);
```

Listing 5.28: Auswertung von $x/\sqrt{y}$ ohne Flag

so ist bei der Auswertung mit Flag nicht mehr viel von der intuitiven Notation in Listing 5.28 zu sehen. Weiter wird durch die erschwerte Notation der Auswertung mit Flags der zweite Nachteil dieses Konzepts zusätzlich verstärkt, der Benutzer ist selbst für die Verwaltung der Flags zuständig. Er muss dafür Sorge tragen, dass die verwendeten boolschen Variablen zum jeweils richtigen Zeitpunkt wieder initialisiert werden. Eine parallele Ausführung ist mit diesem Konzept der Flagverwaltung möglich, jedoch muss der Benutzer dann besonders auf eine konsistente Programmierung achten.

### 5.2.1 Ausdrucksbezogene Flags

Die beiden Konzepte zur Flagverwaltung über eine globale Variable oder einen boolschen Parameter haben einen gemeinsamen Nachteil, sie sind nicht auf einen Ausdruck beschränkt. Sie können von mehreren Ausdrücken verwendet werden und sind demnach von der Verwaltung des Benutzers abhängig.

Diese Abhängigkeit vom Benutzer kann aber durch ein Verknüpfen eines Ausdrucks mit einem eigenen Flag behoben werden. Ein Intervallausdruck liefert bei der Auswertung demnach nicht nur das Ergebnisintervall, sondern auch ein, für diese Ausdrucksauswertung, eigenes Flag. Durch einen neuen Datentyp `ExprResult`[5] können die beiden Werte zusammengefasst werden.

```
1 class ExprResult {
2   private:
3     Interval _i;
4     bool _flag;
5
6   public:
```

[5] Auf die Angabe eines Konstruktors mit Übergabe der benötigten Werte wird hier noch verzichtet. Die Definition der Konstruktoren wird später im Text bei der genauen Implementierung besprochen.

```
7       bool getFlag() const {
8           return _flag;
9       }
10
11      double getInf() const {
12          return _i.inf();
13      }
14
15      double getSup() const {
16          return _i.sup();
17      }
18
19      Interval getInterval() const {
20          return _i;
21      }
22  };
```

Listing 5.29: Klasse `ExprResult`

Das Ergebnis eines Intervallausdrucks ist demnach kein Intervall vom Typ `Interval`, sondern eine Kombination aus Intervall und Flag. Eine zum Codebeispiel in Listing 3.1 äquivalente Auswertung mit `ExprResult` könnte dann wie folgt durchgeführt werden.

```
1 Interval x(1.0, 1.0), y(-1.0, 1.0);
2
3 ExprResult r = x / sqrt(y);
4
5 if (!r.getFlag())
6         Interval i = r.getInterval();
7         ...
```

Listing 5.30: Verwendung des Datentyps `ExprResult` zur Auswertung von $x/\sqrt{y}$

Der Benutzer muss sich nun nicht mehr um die Verwaltung von Flags kümmern, jedoch ist jetzt kein direkter Zugriff auf das Ergebnisintervall mehr möglich.

Durch erweitern des Datentyps `Interval` aus Listing 5.9 kann dies jedoch durch eine automatische Zuweisung behoben werden. Der folgende Konstruktor sowie der Zuweisungsoperator realisieren diese Zuweisung eines Objekts vom Typ `ExprResult` an die Klasse `Interval`.

```
1 Interval::Interval(ExprResult const& e) {
2     Interval i = e.getInterval();
3     _inf = i.inf();
4     _sup = i.sup();
5 }
6
7 Interval& Interval::operator=(ExprResult const& e) {
8     Interval i = e.getInterval();
9     _inf = i.inf();
10    _sup = i.sup();
11
12    return *this;
13 }
```

Listing 5.31: Konstruktor und Operator der Klasse `Interval` für die Zuweisung von `ExprResult`

Wird das Flag des Ausdrucks nicht benötigt, so bleibt der Umweg über die Klasse `ExprResult` vor dem Benutzer verborgen und die Ausdrucksauswertung entspricht der gewohnten Notation. Ist aber beispielsweise die Stetigkeit des ausgewerteten Intervallausdrucks gefordert, so kann über den Typ `ExprResult` ein entsprechender Zugriff auf das Flag erfolgen. Der Benutzer profitiert demnach von einem flexiblen Konzept der Flagverwaltung.

### Implementierung

Das Konzept der Flagverwaltung kann auf die Ausdrucksauswertung über Expression Templates aus Kapitel 5.1.1 aufsetzen. Es muss nur an einigen Klassen eine Anpassung an die ausdrucksabhängige Flagverwaltung durchgeführt werden.

Die Klasse `IntervalExpr` repräsentiert bei der Auswertung mit Expression Templates die Wurzel des Ausdrucksbaums. Intervallauswertungen werden immer über die Methode `eval()` dieser Klasse ausgeführt. Dementsprechend muss für eine ausdrucksabhängige Flagverwaltung die Methode `eval()` wie folgt angepasst werden.

```
1 ExprResult eval() const {
2         RndControl rnd;
3         FlagControl flag;
4         return ExprResult(_expr.eval(rnd, flag), flag);
5 }
```

Listing 5.32: Angepasste Methode `eval()` der Klasse `IntervalExpr`

Der neue Rückgabetyp entspricht nun der neuen Klasse `ExprResult`. Im Vergleich zur alten Methode aus Listing 5.18 wird hier zusätzlich die Klasse `FlagControl`

verwendet. Diese wird bei der Auswertung, analog zu `RndControl`, als Besucherobjekt [15] durch den Baum gereicht. Die einzelnen Knoten des Ausdrucksbaums haben nun neben der Rundungssteuerung auch die Möglichkeit auf Daten des Obejekts `flag` zuzugreifen.

```
1 class FlagControl {
2   private:
3     bool _flag;
4
5   public:
6     FlagControl() : _flag(false) {}
7
8     void setFlag(bool flag) {
9         _flag = _flag || flag;
10    }
11
12    bool getFlag() {
13        return _flag;
14    }
15 };
```

Listing 5.33: Klasse `FlagControl`

Die Klasse `FlagControl` implementiert alle für die Verwaltung eines ausdrucksabhängigen Flags nötigen Funktionalitäten[6]. Die Methode `setFlag(bool)` realisiert hierbei das Setzen des Flags. Der übergebene Wert wird jedoch nicht direkt gespeichert, sondern mit dem aktuellen Wert `_flag` "verodert" um ein Rücksetzen zu verhindern.

Die Informationen aus `FlagControl` werden dann bei der Erzeugung eines Objekts vom Typ `ExprResult` entsprechend verwendet. Listing 5.34 zeigt die für die automatische Zuweisung eines Ausdrucksbaums an die Klasse `ExprResult` notwendigen Konstruktoren sowie den Zuweisungsoperator.

```
1     ExprResult(Interval i, FlagControl& flag)
2         : _i(i), _flag(flag.getFlag()) {}
3
4     template<typename E>
5     ExprResult(IntervalExpr<E> const& expr) {
6         ExprResult e = expr.eval();
7         _i = e.getInterval();
8         _flag = e.getFlag();
9     }
```

[6] In Listing 5.33 wird lediglich ein Flag umgesetzt, jedoch ist es problemlos möglich mehrere verschiedene Flags vorzusehen.

```
10
11      template<typename E>
12      ExprResult& operator=(IntervalExpr<E> const& expr) {
13          ExprResult e = expr.eval();
14          _i = e.getInterval();
15          _flag = e.getFlag();
16
17          return *this;
18      }
```

Listing 5.34: Konstruktoren und Zuweisungsoperator der Klasse `ExprResult`

Durch ein zusätzliches Besucherobjekt vom Typ `FlagControl` für die Baumauswertung muss die, im Baum verwendete, Schnittstelle für Nachfolgeknoten aus Listing 5.13 an den zusätzlichen Parameter angepasst werden.

```
1 Interval eval(RndControl& rnd, FlagControl& flag) const;
```

Listing 5.35: Schnittstellenvereinbarungen an Nachfolgeknoten

Das Objekt des Typs `FlagControl` wird als zweiter Parameter per Referenz übergeben. In der Implementierung betrifft diese Änderung die Klasse `BinaryIntervalExpr<OP, E1, E2>` aus Listing 5.10. Die Methode `eval` der Klasse `BinaryIntervalExpr<OP, E1, E2>` reicht hierbei, das neue Besucherobjekt analog zum Objekt `rnd` an die verwendete Policy Klasse `OP`, sowie an die beiden, für die Auswertung der Nachfolger zuständigen Traits Klassen `EvalTraits<E1>` und `EvalTraits<E2>`, weiter.

Die Policy Klasse `OP` muss den Parameter vom Typ `FlagControl` selbstverständlich auch in ihrer Schnittstelle berücksichtigen.

```
1 static Interval eval(RndControl& rnd, FlagControl& flag,
2         Interval const& i1, Interval const& i2);
```

Listing 5.36: Schnittstellenvereinbarungen für Policy Klassen mit Flag

Wenn nötig kann die Policy Klasse nun aber auf Methoden des Objekts `flag` zugreifen und demnach das Flag des Ausdrucks setzen. Unstetige oder nicht definierte Auswertungen können für den Benutzer sichtbar gemacht werden.

Schlußendlich muß nur noch die Traits Klasse durch das neue Konzept überarbeitet werden.

```
1  template<typename E>
2  class EvalTraits {
3    public:
4      static Interval eval(RndControl& rnd,
5           FlagControl& flag, E const& e) {
6          return e.eval(rnd, flag);
7      }
8  };
9
10 template<>
11 class EvalTraits<Interval> {
12   public:
13     static Interval eval(RndControl& rnd,
14          FlagControl& flag, Interval const& i) {
15         return i;
16     }
17 };
18
19 template<>
20 class EvalTraits<ExprResult> {
21   public:
22     static Interval eval(RndControl& rnd,
23          FlagControl& flag, ExprResult const& e) {
24         flag.setFlag(e.getFlag());
25         return e.getInterval();
26     }
27 };
```

Listing 5.37: Traits Klassen mit Flag

Die allgemeine Implementierung `EvalTraits<E>` arbeitet hierbei genau wie in der alten Implementierung. Die Parameter werden einfach an den Nachfolgeknoten weitergereicht.

Interessant ist die Implementierung der beiden Typen für Blätter des Ausdrucksbaums. Für Intervalle des Typs `Interval` wird, wie gewohnt, die Spezialisierung `EvalTraits<Interval>` angeboten. Auch hier gibt die Traits Klasse einfach das Intervall zurück. Das Objekt vom Typ `FlagControl` wird nicht beachtet. Anders bei der Spezialisierung für den Typ `ExprResult` hier wird der Wert des Flags in die Auswertung mit einbezogen. Hat das Flag des Objekts `e` den Wert `true`, so wird durch das "Verodern" in der Methode `setFlag(bool)` der Wert auch für diesen Ausdruck übernommen. Objekte vom Typ `ExprResult` können demzufolge auch als Operanden für Intervallausdrücke verwendet werden.

### Vorteile des Datentyps `ExprResult`

Durch die, oben angegebene, Implementierung der Klasse `ExprResult` ist es möglich ausdrucksbezogene Flags auch mit der effizienten Intervallauswertung über Expression Templates zu verwenden. Die Klasse `ExprResult` bleibt bei Nichtgebrauch der Flags vor dem Anwender verborgen, er kann wie gewohnt mit Objekten des Typs `Interval` arbeiten.

Ist das Flag jedoch für bestimmte Anwendungszwecke nötig, so kann die Auswertung direkt an eine Instanz von `ExprResult` zugewiesen werden. Für weitere Auswertungen bleibt das Flag in den Instanzen von `ExprResult` erhalten. Berechnungen mit Teilausdrücken können somit auch direkt über Objekte von `ExprResult` realisiert werden. Das Flag wird dabei in die jeweilige Rechnung mit einbezogen. Listing 5.38 zeigt diese Vorteile an einigen Beispielen auf.

```
1  Interval x(1.0, 1.0), y(-1.0, 1.0);
2
3  // .getFlag() liefert true
4  ExprResult r = sqrt(y);
5
6  // Flag steht nicht zur Verfügung
7  Interval ir = r;
8  Interval i = sqrt(y);
9
10 // .getFlag() liefert true
11 ExprResult a = x + sqrt(y);
12 ExprResult b = x + r;
13
14 // .getFlag() liefert false
15 ExprResult c = x + r.getInterval();
16 ExprResult d = x + ir;
17 ExprResult e = x + i;
18
19 // Flag steht nicht zur Verfügung
20 Interval f = x + sqrt(y);
```

Listing 5.38: Verwendung des Datentyps `ExprResult`

Durch die lokale ausdrucksbezogene Flagverwaltung ist dieses Modell auch besser für aktuelle Rechnerarchitekturen geeignet. Bei einer parallelen Auswertung können Ausdrucksbäume auf verschiedene Threads verteilt werden. Wird der jeweilige Baum dann in einem eigenen Thread über die Methode `eval()` aus Listing 5.32 ausgewertet steht das Flag threadlokal zur Verfügung [45]. Zugriffe durch mehrere Threads sind somit ausgeschlossen und die Konsistenz des Flags bleibt gewahrt.

Eine Auswertung eines Ausdrucksbaums auf mehreren Threads ist wegen des Konzepts der Rundungssteuerung über ein Besucherobjekt vom Typ `RndControl` nicht möglich. Es können jedoch Teilbäume, welche man anschließend in einem weiteren Ausdruck zusammenführt, als eigene Ausdrücke parallel berechnet werden.

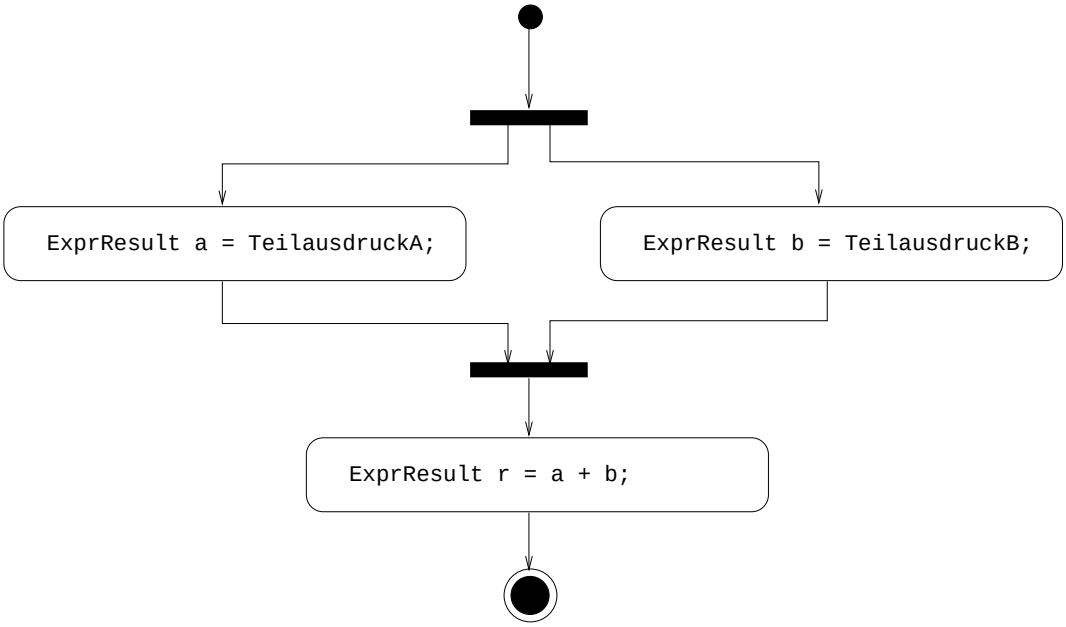



Abbildung 5.8: Parallele Berechnung eines Ausdrucks

## 5.3 Anmerkung

Durch eine Umsetzung der Operatoren für Addition, Subtraktion, Multiplikation und Division mit gerichteten Rundungen, könnte die Implementierung der Intervallarithmetik stark vereinfacht werden. Hierzu könnte man, ähnlich der Programmiersprache Pascal-XSC [1], neue Operatoren für die Ausführung mit gerichteter Rundung in der Sprache C++ aufnehmen. Tabelle 5.4 zeigt einen Vorschlag für die zwölf neuen Operatoren[7], zusammen mit den gewohnten Operatoren für die Grundrechenarten.

Diese neuen Operatoren haben auf Systemen mit direkt aufrufbaren gerundenten Operationen[8] den Vorteil, dass nicht mit einem globalen Rundungsmodus über die Funktionen `fegetround` und `fesetround` gearbeitet werden muss. Die Abbildung der Intervalloperationen, wie etwa der Addition, kann somit leicht über die beiden Operatoren `<+` sowie `>+` auf Hardware umgesetzt werden.

[7]Das Symbol für die Rundung wurde bewußt als erstes Zeichen gewählt um `>-` vom Zeiger `->` zu unterscheiden

[8]Der Itanium Prozessor [18] von Intel unterstützt Operationen mit Angabe der Rundung.

| Rundung | Addition | Subtraktion | Multiplikation | Division |
|---|---|---|---|---|
| ◯ | + | - | * | / |
| △ | >+ | >- | >* | >/ |
| ▽ | <+ | <- | <* | </ |
| □ | _+ | _- | _* | _/ |

Tabelle 5.4: Operatoren mit Rundung

Aber auch auf Systemen, wie dem X86, mit globaler Rundungssteuerung haben diese neuen Operatoren ihren Vorteil. Die Programmierung von Intervalloperationen wird durch den Verzicht auf die globale Steuerung durch den Programmierer stark vereinfacht und Fehlerquellen durch falsch gesetzte Rundungsmodi werden so verhindert. Zusätzlich kann der Compiler die, für diese Operatoren zu setzenden, globalen Rundungsmodi optimieren. Einen Ausdruck, wie etwa `r = a >+ b >+ c >+ d`, könnte der Compiler folglich wie im Pseudocode in Listing 5.39 umsetzen.

```
1 set_rounding_upward();
2 t1 = a + b;
3 t2 = t1 + c;
4 r = t2 + d;
```

Listing 5.39: Umsetzung des Ausdrucks `r = a >+ b >+ c >+ d`

# Kapitel 6

# Hardware

Die Implementierung der Intervallarithmetik kann natürlich nicht nur durch Software realisiert werden. Für die verwendete Gleitkommaarithmetik müssen entsprechende Hardwareunterstützungen existieren. Jedoch müssen für eine vorschriftsmässige Intervallauswertung vorher noch einige Punkte beachtet werden. Auch ist es von Interesse welche Möglichkeiten für die Intervallrechnung auf der verbreiteten X86 Architektur zur Verfügung stehen.

## 6.1 X86 Architektur

Die X86 Architektur stellt heute eine der am weitesten verbreiteten Rechnerarchitekturen da. Unter dem Begriff X86 werden Prozessoren zusammengefasst, welche zu der im Jahre 1978 von der Firma Intel vorgestellten 8086 Mikroprozessorarchitektur kompatibel sind [22]. Heute unterscheidet man hierbei noch zwischen einer älteren 32 Bit und einer neuen 64 Bit Architektur.

Der, für die Intervallarithmetik wichtige, IEEE 754 Standard für Gleitkommazahlen wird durch die Prozessorarchitektur für die drei Gleitkommaformate

- `single`
- `double`
- `double extended`

abgedeckt. Für das Format `double extended` wird ein 80 Bit Register verwendet, um eine Mantisse der Länge 64 sowie einen Exponent der Länge 15 zu verwalten.

Die Gleitkommaberechnungen werden auf dem X86 über die X87 Gleitkommaumgebung durchgeführt. Die Bezeichnung X87 deutet hierbei auf die Kompatibilität zu den, für Gleitkommaberechnungen eingeführten, 8087 Coprozessor hin. Mit Erscheinen des Pentium III wurde zusätzlich zur X87 Gleitkommaumgebung eine

zusätzliche "Streaming SIMD Extensions" (SSE) Gleitkommaumgebung eingeführt[1], welche mit dem Pentium 4 auf die SSE2 Gleitkommaumgebung erweitert wurde. Diese zusätzliche Gleitkommaeinheit unterstützt ebenfalls IEEE 754 Gleitkommazahlen im Format `single` und `double`.

Die vier verschiedenen Rundungsmodi des IEEE 754 Standard können auf der X87 sowie der SSE2 Gleitkommaumgebung nur über jeweils ein eigenes globales Steuerwort geregelt werden. Direkt ansprechbare, unterschiedlich gerundete Gleitkommaoperationen stehen nicht zur Verfügung. Eine Änderung des globalen Rundungsmodus ist jedoch anschließend mit einer laufzeitintensiven Leerung der Pipeline der Gleitkommaumgebung verbunden [22].

### 6.1.1 X87

Die X87 Gleitkommaumgebung verwendet für ihre Berechnungen acht 80 Bit Register mit einer Mantissenlänge von 64 Bit, einen 15 Bit Exponenten sowie 1 Bit für das Vorzeichen. Standardmässig werden demnach Berechnungen mit 64 Bit Genauigkeit im IEEE 754 konformen Format `double extended` durchgeführt.

Es werden jedoch auch die Formate `single` und `double` unterstützt, um Programmiersprachen eine IEEE 754 konforme Umgebung zu bieten. Über das globale Steuerwort kann hierbei die Genauigkeit der Mantisse auf 24 oder 53 Bit für die Formate `single` oder `double` begrenzt werden. Die nicht benötigten Bits werden dann beim Runden der Mantisse jeweils mit dem Wert 0 überschrieben. Die Länge des Exponenten bleibt aber bei den verwendeten 15 Bit [32] und entspricht somit nicht den IEEE 754 Spezifikationen für die Formate `single` und `double`.

#### Probleme mit 80 Bit Registern

Bei der Verwendung der Formate `single` und `double` in einer Programmiersprache tritt durch die Verwendung eines 80 Bit Registers sowie des 15 Bit Exponenten der X87 Gleitkommaumgebung oft sogenanntes "double rounding" auf. Hierbei wird während der Berechnung im 80 Bit Register des X87 mit einer Genauigkeit von 64 Bit und einem Exponenten der Länge 15 gerechnet. In diesem Gleitkommaformat nicht darstellbare Zahlen werden auf eine 80 Bit Gleitkommazahl gerundet. Jedoch sind in der Gleitkommaumgebung nur acht Register für die Berechnung vorhanden. Um für weitere Berechnungen Platz zu schaffen, müssen Ergebnisse wieder in den Hauptspeicher zurückgeschrieben werden. Dieses Rückschreiben geschieht jedoch im gewählten Format `single` oder `double` mit den entsprechend kürzeren Mantissen und Exponenten. Für eine Darstellung im Zielformat ist ein zweites Runden der Zahl nötig [17].

Durch diese doppelte Rundung kann die Rechenregel $\bigtriangledown a = - \bigtriangleup (-a)$ aus Satz 8 nicht fehlerfrei verwendet werden [6]. Wird die Rundung der Gleitkom-

[1] SIMD = Single Instruction, Multiple Data

maeinheit global auf $\triangle$ gesetzt und das Ergebnis $a$ im 80 Bit Register berechnet, so ist die angegebene Gleichheit noch gegeben. Beim Zurückschreiben in den Hauptspeicher wird jedoch eine erneute Rundung gegen $+\infty$ durchgeführt. Die Gleichheit aus Satz 8 ist nicht mehr erfüllt und es gilt folgende Beziehung:

$$\bigtriangledown a \leq - \triangle (-a) \tag{6.1}$$

Die Einschließungseigenschaft aus Satz 3 kann demnach nicht garantiert werden.

Aber auch Berechnungen mit Gleitkommazahlen "nahe" dem Wert $\pm\infty$ bereiten, durch den 15 Bit Exponenten, Probleme. Führt man beispielsweise die Multiplikation

$$[x_{max}\,,\, x_{max}] \cdot [2\,,\, 2] \tag{6.2}$$

der beiden Punktintervalle für die größte im Gleitkommaformat `double` darstellbare endliche Zahl $x_{max} < +\infty$ aus, so kann durch den längeren Exponenten das im 80 Bit Register korrekte Ergebnis $[2x_{max}\,,\, 2x_{max}]$ dargestellt werden. Beim Rückschreiben in den Hauptspeicher tritt dann ein Überlauf auf, die beiden Grenzen müssen auf einen im Format `double` darstellbaren Wert gerundet werden. Je nach aktuellem Rundungsmodus der Gleitkommaumgebung wird $[x_{max}\,,\, x_{max}]$ oder $[+\infty\,,\, +\infty]$ als Ergebnis in den Hauptspeicher geschrieben. Der exakte Wert $2x_{max}$ ist somit in beiden Intervallen nicht enthalten und die Einschließungseigenschaft nicht erfüllt. Als korrektes Ergebnis im Format `double` müsste nach den Rechenregeln aus Kapitel 2.6 für die Grenzen eine gerichtete Rundung gegen $-\infty$ und $+\infty$ durchgeführt und als Intervall $[x_{max}\,,\, +\infty)$ geliefert werden.

Die Probleme mit den 80 Bit Registern kann man über ein explizites Zurückschreiben in den Hauptspeicher lösen. Hierzu werden die Ergebnisse direkt nach der Gleitkommaoperation im noch aktuellen Rundungsmodus wieder in den Hauptspeicher geschrieben. Die Rundung wird demnach noch korrekt ausgeführt. Dies geht jedoch zu Lasten der Performance. Bei vielen C++ Compilern, wie etwa dem `gcc` [16], kann dieses Rückschreiben auch über Compileroptionen erzwungen werden.

### 6.1.2 SSE2

Für Berechnungen mit den IEEE 754 Formaten `single` oder `double` bieten Rechner ab dem Pentium 4 mit der SSE2 Gleitkommaumgebung bessere Möglichkeiten. Die SSE Gleitkommaeinheit verwendet für die Berechnung von Gleitkommazahlen der Formate `single` oder `double` die acht 128 Bit Register aus der MMX Prozessorerweiterung [22]. Auf diesen 128 Bit Registern werden die Datentypen "gepackt" abgelegt. Es können somit vier 32 Bit Gleitkommazahlen im Format `single` oder zwei 64 Bit Gleitkommazahlen im Format `double` gespeichert werden.

Die Gleitkommaoperationen werden dann auch mit der entprechenden Genauigkeit von 32 Bit oder 64 Bit durchgeführt. Probleme wie mit den 80 Bit Registern des X87 treten somit nicht auf.

Bei den Operationen wird in der SSE Gleitkommaumgebung zwischen skalaren und "gepackten" Operationen unterschieden. Skalare Operationen für `single` und `double` entsprechen den Operationen der X87 Gleitkommaeinheit, jedoch wird mit der im IEEE 754 Standard spezifierten Mantissen- und Exponentenlänge gerechnet.

Beim Ausführen einer skalaren Operation für `single` oder `double` findet die Berechnung dabei nur auf dem niederwertigsten Registerbereich für den jeweiligen Datentyp statt. Es werden also nur die niederwertigsten 32 oder 64 Bit der jeweiligen Register für die Berechnung herangezogen. Die restlichen 64 oder 96 Bit werden, wie in Abbildung 6.1 für das Format `double` gezeigt, aus dem höherwertigen Bereich des Registers des ersten Operanden in das Ergebnis kopiert. Für Berechnungen mit Intervallen sind diese Operationen demnach den X87 Operationen durch die IEEE 754 konforme Semantik vorzuziehen[2]. Zusätzlich zu den

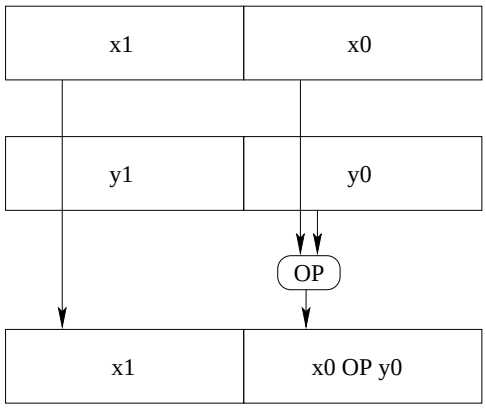



Abbildung 6.1: Skalare Operation

skalaren Operationen stehen noch "gepackte" Operationen bereit. Diese entsprechen dem SIMD (Single Instruction, Multiple Data) Konzept [46], welches eine identische Operation parallel auf mehreren Resourcen ausführt. Somit können auf den 128 Bit Registern, je nach Format, zwei oder vier identische Operationen parallel ausgeführt werden. Abbildung 6.2 zeigt eine parallele Operation für den Typ `double`.

Durch die parallele Ausführung von zwei identischen Operationen im Format `double` eröffnen sich gute Möglichkeiten für die Intervallarithmetik [27]. Die im

[2]Der C++ Compiler `gcc` verwendet für die Rechnerarchitekturen ab Pentium 4 standardmässig SSE2 für die Gleitkommaberechnung der Typen `float` und `double`.

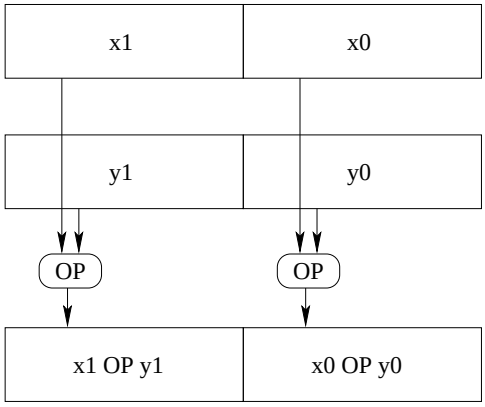



Abbildung 6.2: SIMD Operation

Kapitel 4.2.1 vorgestellte Repräsentation nach Lambov lässt sich somit eins zu eins auf ein 128 Bit Register übertragen. Speichert man beispielsweise die obere Grenze eines Intervalls negiert ab, so können die Addition und Subtraktion mit der Rundung $\bigtriangledown$ jeweils durch die "gepackte" SSE2 Instruktion `ADDPD` ausgeführt werden. Abbildung 6.3 zeigt dies anhand der Addition mit negierter Obergrenze über die SSE2 Intstruktion `ADDPD`[3]

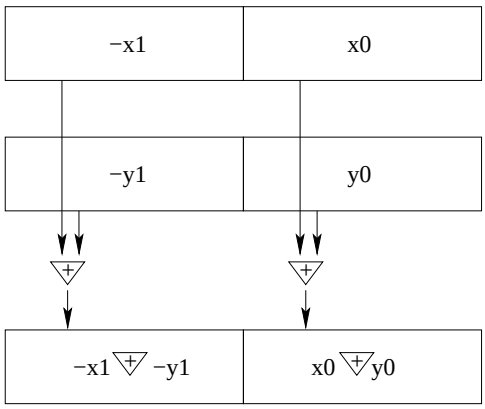



Abbildung 6.3: Intervalladdition mittels `ADDPD`

Für die Subtraktion ist es jedoch vorher nötig die Grenzen eines Intervalls zu drehen, um der Formel

$$X - Y = [x_1 \bigtriangledown x_2 \, , \, y_2 \bigtriangleup y_1] \tag{6.3}$$

gerecht zu werden. In der Darstellung nach Lambov entspricht dies einer Negation des Intervalls. Die SSE2 Instruktion `SHUFPD` kann für diesen Zweck verwendet werden. Hierbei werden die beiden `double` Werte aus dem ersten Operanden miteinander "verodert" und im niederwertigen Bereich des Zielregisters gespeichert.

[3]Die obere Grenze wurde hierbei als höherwertiges (linkes) `double` gespeichert.

Analog werden die beiden Werte des zweiten Operanden "verodert" und im höherwertigen Bereich gespeichert. Verwendet man für beide Operanden das gleiche Register so werden die Werte getauscht und das Intervall negiert. Die Subtraktion kann nun über den Befehl `ADDPD` in der Form $X + (-Y)$ erfolgen.

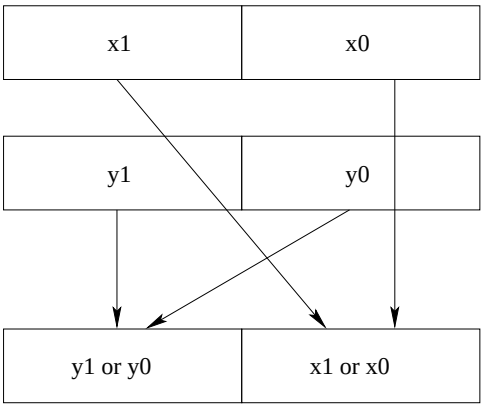



Abbildung 6.4: SSE2 Instruktion `SHUFPD`

Das Konzept der "gepackten" parallelen Operationen der SSE2 Gleitkommaeinheit lässt sich mit gerichteter Rundung somit sehr gut für die beiden Grundrechenarten Addition und Subtraktion verwenden. Hier können alle auftretenden Fälle durch eine oder zwei Instruktionen abgehandelt werden.

Schwieriger gestaltet sich dies bei der Multiplikation und Division. Hier müssen die verschiedenen möglichen Fälle aus den Tabellen 4.7, 4.8 sowie 4.9 über entsprechende Fallunterscheidung behandelt werden. Zusätzlich muss ein Operand $[x_1\,,\,x_2]$ in den vier verschiedenen Formen $\langle x_1\,;\,x_1\rangle$, $\langle x_1\,;\,x_2\rangle$, $\langle x_2\,;\,x_1\rangle$ und $\langle x_2\,;\,x_2\rangle$ für die parallele Multiplikation `MULPD` oder Division `MULPD` bereit stehen. Einfache Fälle, wie etwa die Multiplikation zweier strikt positiver Intervalle, sind aber auch hier leicht über die Instruktion `MULPD` zu handhaben [27].

### 6.1.3 Multicore

Das Erscheinen von neuen Prozessorarchitekturen mit mehreren Prozessorkernen ermöglicht viele neue Möglichkeiten in der Softwareentwicklung. Für Multithreading ist eine echte Parallelität möglich und es können mehrere Aufgaben auf die verschiedenen Kerne der CPU verteilt werden.

Betrachtet man die Intel Core Architektur [23] so existieren je nach Modell zwei oder vier eigenständige Prozessorkerne pro CPU. Die Kerne der Intel Core2 Prozessoren basieren dabei auf der Technologie des Pentium M und besitzen jeweils eigene X87 und SEE Gleitkommaumgebungen. Diese unabhängigen Gleitkommaumgebungen lassen schnell die Hoffnung aufkommen Intervalloperationen auf mehreren Kernen parallel auszuführen. Hierbei könnte ein Kern der CPU mit

Rundung gegen $-\infty$, ein Zweiter mit Rundung gegen $+\infty$ arbeiten. Für die Berechnung der Intervalloperationen sind dann zwei Threads zuständig die jeweils auf einem Kern abgearbeitet werden. Für die Threadverwaltung ist aber in der Regel das Betriebssystem zuständig und liegt somit nicht in der Macht des Benutzers oder eine Softwarebibliothek. Über die Pthreads Bibliothek in C++ kann man jedoch einen Thread an einen Prozessor/Kern binden.

Mit diesen Mitteln könnte somit eine Addition, wie in Abbildung 6.5, von mehreren Intervallen parallel auf zwei Prozessorkernen durchgeführt werden. Die unteren Grenzen werden dabei komplett vom ersten Kern berechnet, die oberen Grenzen komplett vom zweiten Kern.

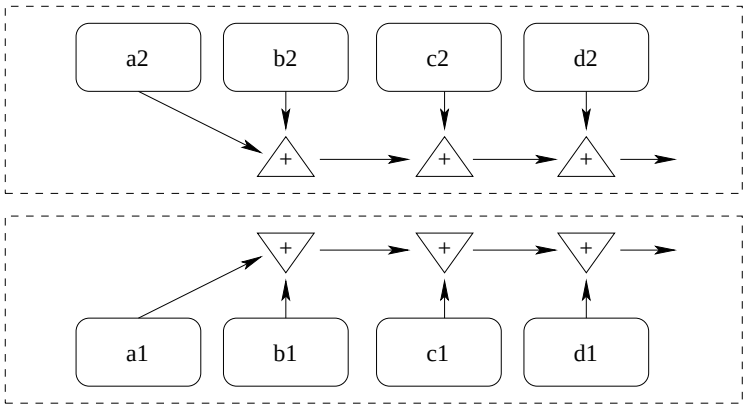



Abbildung 6.5: Parallele Intervalladdition $a + b + c + d$ auf zwei Kernen

Wird jedoch die Multiplikation in Tabelle 4.8 betrachted, so erkennt man schnell, dass eine strikte Trennung von Unter- und Obergrenze auf zwei verschiedene Threads nicht möglich ist. Es müssen Werte zwischen den Threads ausgetauscht werden. Dies bringt die bei der Thread-Programmierung notwendige Synchronisation mit sich. Die Threads müssen jeweils auf die berechneten Ergebnisse des Anderen warten und Daten müssen zwischen den Threads ausgetauscht werden.

Der Datenaustausch zwischen Threads wird über globale Variablen mit Sperrmechanismen gehandhabt [46]. Bei schnell aufeinander folgende Synchronisationen bietet sich, um die Latenzzeiten so gering wie möglich zu halten, der Spin Lock Mechanismus [21] an. Für eine Laufzeitabschätzung der benötigten Sperrmechanismen wurde auf einem Intel Core2 Quad Q9450 mit 2,66 GHz ein Testlauf durchgeführt. Hierbei konnte festgestellt werden, dass ein Wechsel des Rundungsmodus nur etwa die doppelte Zeit wie das Sperren und wieder Freigeben einer globalen Variable über Spin Locks benötigt. Jedoch müssen für eine Datenübergabe zwischen Threads mehrere Sperren über Spin Locks durchgeführt werden. Eine Laufzeitverbesserung ist hierbei nicht zu erwarten. Vor allem die Intervallauswertungen über Expression Templates mit einseitiger Rundung haben hier weit mehr Vorteile, da die Rundung nur einmal für den ganzen Ausdruck gesetzt werden muss.

Dennoch hat die Multicore Architektur, wie bei vielen numerischen Berechnungen, auch in der Intervallarithmetik seine Vorteile. Man denke nur an die Multiplikation von Intervallmatrizen oder andere gut verteilbare Probleme. Diese können, wie in Kapitel 5.2.1 angesprochen, über die Auswertung mit Expression Templates und ausdrucksbezogener Flagverwaltung sehr gut parallel ausgeführt werden.

## 6.2 Interessante Architekturen

Im Bereich des numerischen Rechnens erregen im Moment vor allem zwei Architekturen die Aufmerksamkeit der Entwickler. Zum einen die Prozessoren auf Grafikkarten und zum anderen der ursprünglich für Spielekonsolen entwickelte Cell Prozessor [12].

Die auf Grafikkarten eingesetzte "Graphics Processing Unit" (GPU) ist im Gegensatz zu einer CPU ein, für einen bestimmten Anwendungsfall bestimmter Mikroprozessor. Durch die spezielle Auslegung auf Grafikverarbeitung und 3D Rendering haben sie bei der numerischen Berechnung gegenüber einer CPU einen enormen Performancevorteil. Sylvain Collange, Jorge Flórez und David Defour haben die Möglichkeiten der Intervallarithmetik auf Grafikkarten untersucht [11] und eine auf der Boost Intervallbiliothek aufbauende Implementierung für die Berechnung bei "Interval Ray Tracing" Algorithmen [14] verwendet.

Das Hauptproblem bei der Intervallrechnung auf der GPU ist die mangelnde Unterstützung des IEEE 754 Standard für Gleitkommazahlen. Bei DirectX und OpenGL kompatiblen Grafikarten sind zwar Gleitkommazahlen dem IEEE 754 Format `single` entsprechend implementiert, jedoch sind die Rundungsmodi $\bigtriangledown$ und $\bigtriangleup$ nicht vorhanden. Um die bei der Intervallrechnung jeweils benötigten Rundungen auf Grafikkarten zu simulieren, kann die Rundung gegen Null verwendet werden. Je nach Vorzeichen entspricht dies dem Modus $\bigtriangledown$ oder $\bigtriangleup$. Die jeweils andere Rundung kann dann über den Vorgänger $pred(x)$ oder den Nachfolger $succ(x)$ der berechneten Zahl $x$ erfolgen. Dies kann jedoch bei genau berechneten Ergebnissen zu einer Überschätzung führen [11].

Bei Laufzeitvergleichen konnten die drei Entwickler auf einem System mit einem 3GHz Xeon Prozessor, 3 Gigabyte Hauptspeicher sowie einer NVIDIA GeForce 8800 GTX Grafikkarte eine um den Faktor 100 bis 300 bessere Performance ihrer GPU Implementierung gegenüber einer CPU Version messen [11].

Der von IBM, Toshiba und Sony gemeinsam entwickelte Cell Prozessor [12] bietet eine für die Intervallarithmetik interessante Möglichkeit der Rundungssteuerung. Für den IEEE 754 konformen 64 Bit Datentyp `double`[4] können bei SIMD Opera-

[4]Der Cell Prozessor verwendet ein eigenes vom IEEE 754 Standard abweichendes Format für 32 Bit Gleitkommazahlen.

tionen auf 128 Bit Registern die Rundungsmodi für die niederwertigere sowie die höherwertigere Gleitkommazahl getrennt gesteuert werden. Intervalloperationen können somit intuitiv mittels SIMD Operationen mit den beiden Rundungsmodi $\bigtriangledown$ und $\bigtriangleup$ auf den 128 Bit Registern der SPU abgebildet werden. Weiter ist die Operation "fused multiply add" auf Cell Prozessoren implementiert. Viele der in Kapitel 4.4 angesprochenen Punkte könnten somit ohne große Probleme auf dieser Hardware umgesetzt werden.

## 6.3 Anmerkungen

Durch den IEEE 754 Standard [37] ist eine gute Unterstützung der Gleitkommaberechnung auf vielen Hardwareplattformen gegeben. Jedoch sind direkt ausführbare Operationen mit gerichteter Rundung gegen $\pm\infty$ sehr selten umgesetzt. Und auch im 2008 neu überarbeiteten IEEE 754 Standard [38] wurden, trotz Anregungen durch Forscher im Bereich der Intervallarithmetik [25], die Operationen mit gerichteter Rundung nicht gefordert. Diese Operationen könnten jedoch eine Implementierung der Intervallarithmetik sowohl in Soft- als auch in Hardware erleichtern und zu einer größeren Verbreitung der Intervallrechung beitragen.

# Kapitel 7

# Zusammenfassung und Diskussion

Das Ziel dieser Diplomarbeit war die Untersuchung effizienter Implementierungsmöglichkeiten der Intervallarithmetik mittels der Programmiersprache C++. Besonderes Augenmerk wurde dabei auf die Repräsentation der Intervalle als Datentyp, der Implementierung der Intervalloperationen mittels der zur Verfügung stehenden Gleitkommaarithmetik, der Verwendung von Flags und ihre Verwaltung sowie auf effizienten Rundungsstrategien geworfen. Zusätzlich wurden die existierenden Ansätze der Boost Intervallbibliothek [6], filib++ [29] sowie der Vorschlag für Intervallarithmetik in der C++ Standardbibliothek [41] analysiert.

## 7.1 Vergleich bestehender Systeme

Bei den drei miteinander verglichenen Implementierungskonzepten konnten viele Gemeinsamkeiten festgestellt werden. So wird bei allen die Repräsentation der Intervalle über Unter- und Obergrenze gehandhabt. Auch die implementierte Funktionsvielfalt ist auf ähnlichem Niveau.

Jedoch gehen die drei Konzepte bei der Konfigurierbarkeit für verschiedene Datentypen der Grenzwerte, der verwendeten Rundungsstrategien oder dem gewählten Berechnungsmodell auseinander. Das für Änderungen und Anpassungen mit Abstand offenste System ist die Boost Implementierung. Hier wird ein modernes Konzept zur Konfiguration der Intervalldatentypen über Policy Klassen [4] geboten. Der Benutzer kann aus einem großen Angebot an vorgefertigen Policies für die Rundungsstrategie sowie dem Berechnungsmodell wählen. Auch ist es durch eigene Policies möglich die Intervallrechnung an beliebige Anforderungen anzupassen. Ein gutes Beispiel ist hier eine auf Boost basierende Implementierung der Intervallarithmetik für Grafikkarten [11]. Auch konnte das Konzept der Vergleichsoperatoren über verschiedene Namensräume überzeugen.

filib++ geht einen etwas anderen, aber dennoch eleganten Weg. Verschiedene Berechnungsmodelle und Rundungsstrategien wurden über Traits Klassen [33] sehr effizient implementiert. Hier konnte vor allem die im Vergleich zur Boost

Implementierung gute Performance überzeugen. Jedoch sind einige Designentscheidungen, wie etwa die Verwaltung eines globalen Fehlerflags oder die strikte Rücksetzung des Rundungsmodus auf eine Rundung zur nächsten Zahl, für einige Anwendungen zu einschränkend.

Das Ziel des Vorschlags für eine Intervallarithmetik in der C++ Standardbibliothek ist weniger eine frei konfigurierbare Bibliothek anzubieten, sondern vielmehr einen einheitlichen Datentyp mit einer genau spezifizierten Semantik bereitzustellen. Durch den Ursprung in der Entwicklergemeinde der Boost Intervallbibliothek sind einige gute Konzepte der Boost Implementierung mit in den Vorschlag eingeflossen. Vor allem die Wahl der Vergleichsoperationen über verschiedene Namensbereiche sowie die auf Bool Sets [42] basierenden Vergleiche stechen hier positiv hervor.

## 7.2 Repräsentation eines Intervalls

Für die Realisierung eines Datentyps `Interval` wurden in dieser Arbeit mehrere verschiedene Ansätze zur Repräsentation von Intervallen untersucht. Hierbei sprechen jedoch einge Argumente für eine Darstellung über Unter- und Obergrenze. Es konnte zwar eine modifizierte Mittelpunkt Radius Darstellung gefunden werden welche das Abspeichern einer zusätzlichen Information, wie etwa einem Fehlerflag, erlaubt. Jedoch werden durch diese Modifikation die benötigten Intervalloperationen so sehr erschwert, dass sich diese Darstellungsform nicht für eine Implementierung empfiehlt.

In dem Bereich der Intervallrepräsentation konnte das von Branimir Lambov vorgestellte Verfahren mit negierter Obergrenze [27] überzeugen. Druch diese Darstellung können Intervalloperationen mit einer einseitigen Rundung durchgeführt werden und resultieren durch eine Optimierte Rundungsstrategie in einer guten Performance. Durch die spezielle Darstellung ist dieses Verfahren sowohl auf Hard- als auch in Software eine gute Lösung. In Software können ungewollte und fehlerhafte Compileroptimierungen verhindert werden. In Hardware kann man diese Repräsentation vor allem gut bei SIMD Operationen wie etwa auf der SSE2 Gleitkommaeinheit verwenden.

Weiter konnte für die Repräsentation eines Datentyps `Interval` über zwei IEEE 754 Gleitkommazahlen gezeigt werden, dass Intervalloperationen leicht über Gleitkommaoperationen abgebildet werden können. Wichtig ist hierbei die Darstellung der leeren Menge in der Form $\langle$`NaN`; `NaN`$\rangle$. Hierdurch können viele Eigenschaft der IEEE 754 Gleitkommzahlen ausgenutzt werden, um eine ausnahmefrei erweiterte Intervallarithmetik umzusetzen.

## 7.3 Problemlösungen

Für drei in der Intervallrechnung auftretende Probleme konnte in dieser Arbeit durch Ausnutzen einiger C++ Techniken eine effiziente, sichere und auch komfortable Lösung erarbeitetet werden.

Durch Anwenden von Expression Templates [50] zur Auswertung von Intervallausdrücken können die in der Intervallrechnung notwendigen Rundungswechsel optimiert werden. Hierbei werden über Expression Templates zur Übersetzungszeit Ausdrucksbäume für die Intervallausdrücke generieren. Bei der Auswertung kann man dann über das Besucher Muster [15] spezielle Rundungsstrategien auf dem Baum anwenden. Diese Technik konnte bei Laufzeitvergleichen mit filib++ oder der Boost Intervallbibliothek vor allem bei längeren Intervallausdrücken durch eine gute Performance überzeugen.

Durch die Anwendung des Besucher Musters bei der Baumauswertung konnte auch für die Verwaltung von Fehlerflags eine neue Strategie erarbeitet werden. Der Gültigkeitsbereich der Flags wurde im Gegensatz zu filib++ oder dem Vorschlag für Intervallrechnung in der C++ Standardbibliothek auf einen Intervallausdruck beschränkt. Die Fehlerflags sind demnach Teil eines Intervallausdrucks und können über das Ergebnis der Auswertung ausgelesen werden. Dies hat vor allem gegenüber den globalen Flags der Bibliothek filib++ den Vorteil einer möglichen parallelen Auswertung mehrere Intervallausdrücke in einer Multithreadingumgebung. Weiter wird die Programmierung mit Fehlerflags für den Anwender erleichtert. Dies konnte durch einen speziellen Rückgabetyp `ExprResult` gelöst werden. Der Typ `ExprResult` stellt hierbei eine Wrapperklasse für alle Ergebnisse der Intervallauswertung des Ausdrucks dar. Er enthält sowohl das Intervall als auch Daten wie etwa ein Fehlerflag. Durch Zuweisen eines Ausdrucks an diesen Typ werden dann die entsprechenden Ergebnisse gespeichert und der Benutzer kann über die Schnittstelle des Typs `ExprResult` auf das Intervall sowie die Flags zugreifen. Durch die Zuweisung eines `ExprResult` an den Datentyp `Interval` wird in diesem jedoch nur das Intervall gespeichert, das Flag wird verworfen. Diese Technik bietet den Anwender eine einfach Möglichkeit mit Flags zu programmieren ohne dabei die Algorithmen komplett neu zu überarbeiten.

Das Konzept des `ExprResult` als Wrapperklasse konnte auch für die Division durch Null verwendet werden. In `ExprResult` können die bei einer Division durch Null auftretenden zwei disjunkte Intervalle als Ergebnis gespeichert werden. Bei einer Zuweisung an den Typ `Interval` wird für diese beiden Mengen dann das einschließende Intervall erzeugt, und man erhält das gewohnte (und überschätzte) Ergebnis. Jedoch kann der Anwender über `ExprResult` auch auf die beiden Teilmengen zugreifen.

Die Implementierung des Datentyps `ExprResult` verbessert somit die Benutzbarkeit bei Anwendungen die zum Beispiel auf die Stetigkeit bei der Intervallauswertung angewiesen sind. Hier müssen nicht mühsam Flags verwaltet oder Intervalle händisch durch Bisektion aufbereitet werden. Zusätzlich profitiert man

von den Performancevorteilen durch die Auswertung mit Expression Templates.

## 7.4 Hardware

Für die X86 Architektur wurden einige Probleme aufgezeigt, die vor allem mit dem 80 Bit Registern der X87 Gleitkommaeinheit zusammen hängen. In bestimmten Fällen kann die Einschließungseigenschaft bei der Intervallrechnung nicht garantiert werden. Hierfür ist die SSE2 Gleitkommaeinheit besser geeignet. Deshalb ist es auf X86 Systemen wichtig die Gleitkommaoperationen der Intervallgrenzen des Typs `single` oder `double` auf der SEE2 Umgebung abzuarbeiten. Weiter können durch SIMD Operationen einige Intervalloperationen relativ einfach auf die Hardware abgebildet werden.

Im Bereich der Hardware für die Intervallrechnung stechen jedoch zwei andere Architekturen heraus. Zum einem die Grafikprozessoren und zum anderen der Cell Prozessor. Grafikprozessoren moderner Grafikkarten profitieren stark von den Rechenanforderungen einiger 3D Anwendungen. Diese Prozessoren sind für numerische Berechnungen spezialisiert und können einen enormen Performancegewinn für die Intervallrechnung darstellen [11]. Jedoch sind durch die eingeschränkte Unterstützung des IEEE 754 Standards und vor allem durch die fehlenden Rundungsmodi $\nabla$ und $\triangle$ andere Einschränkungen hinzunehmen.

Die zweite interessante Architektur ist der Cell Prozessor, welcher sich unter anderem durch die Möglichkeit zweier verschiedener Rundungsmodi bei parallelen SIMD Operationen für die Intervallrechnung anbietet.

## 7.5 Fazit

In dieser Arbeit wurden einige Implementierungsmöglichkeit mit der Sprache C++ untersucht. Vor allem die Verwendung von Expression Templates hat sich für die Intervallauswertung als geeignet herausgestellt. Hier konnte eine gute Optimierungen für die Rundungsstrategie erarbeitet werden. Jedoch bietet die Möglichkeit der Template Metaprogrammierung [49] noch viele andere mächtige und effiziente Techniken. Hier besteht weiteres Potential für zukünftige Forschungen.

Weitere für die Zukunft interessante Möglichkeiten eröffnet der Cell Prozessor. Hier können vor allem die flexiblen SIMD Operationen sowie “fused multiply add” für effiziente Implementierungen herangezogen werden.

# Listings

# Tabellenverzeichnis

# Abbildungsverzeichnis

# Literaturverzeichnis


[1] Pascal-XSC Language Reference with Examples, March 1999. http://www.math.uni-wuppertal.de/~xsc/literatur/PXSCENGL.pdf.

[2] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 1: Application Programming, Revision 3.14, September 2007. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf.

[3] Götz Alefeld and Jürgen Herzberger. Einführung in die Intervallrechnung. Bibliographisches Institut AG, Zürich, 1974.

[4] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001.

[5] Boost C++ Libraries, February 2009. http://www.boost.org/.

[6] Boost Interval Arithmetic Library, February 2009. http://www.boost.org/doc/libs/1_37_0/libs/numeric/interval/doc/interval.htm.

[7] Boost.Tribool, February 2009. http://www.boost.org/doc/libs/1_37_0/doc/html/tribool.html.

[8] Axel Böttcher. Rechneraufbau und Rechnerarchitektur (eXamen.press). Springer-Verlag, 2006.

[9] I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. Taschenbuch der Mathematik, überarbeitete und erweiterte 5. Auflage. Verlag Harri Deutsch, Frankfurt am Main, 2000.

[10] International Standard ISO/IEC 9899: Programming languages - C, September 2007. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[11] Sylvain Collange, Jorge Flórez, and David Defour. A GPU interval library based on Boost interval. funded by ARN eva-FLO.

[12] IBM Corporation, Sony Computer Entertainment Corporation, and Toshiba Corporation. Synergistic Processor Unit Instruction Set Architecture, Version 1.2, January 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44/$file/SPU_ISA_v1.2_27Jan2007_pub.pdf.

[13] filib++ Interval Library, July 2008. http://www.math.uni-wuppertal.de/org/WRST/software/filib.html.

[14] Jorge Flórez, Mateu Sbert, Miguel A. Sainz, and Josep Vehí. Improving the Interval Ray Tracing of Implicit Surfaces. In Computer Graphics International 2006, pages 655–664, June 2006.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Entwurfsmuster. Addison-Wesley, 1996.

[16] GCC, the GNU Compiler Collection, February 2009. http://gcc.gnu.org/.

[17] David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv., 23(1):5–48, 1991.

[18] Bruce Greer, John Harrison, Greg Henry, Wei Li, and Peter Tang. Scientific computing on the Itanium$^{TM}$ processor. In Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), pages 41–41, New York, NY, USA, 2001. ACM.

[19] IEEE Interval Standard Working Group - P1788, January 2009. http://grouper.ieee.org/groups/1788/.

[20] ANSI/IEEE. ANSI/IEEE Std 854-1987, IEEE Standard for Radix-Independent Floating-Point Arithmetic. IEEE, New York, October 1987.

[21] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, December 2008. http://download.intel.com/design/processor/manuals/248966.pdf.

[22] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture, November 2008. http://download.intel.com/design/processor/manuals/253665.pdf.

[23] Intel Core Microarchitecture, February 2009. http://www.intel.com/technology/architecture-silicon/core/index.htm.

[24] Ulrich Kulisch. Advanced arithmetic for the digital computer. Springer-Verlag, Wien, 2002.

[25] Ulrich Kulisch. 13 Letters to the IEEE Computer Arithmetic Standards Revision Group, 2005 - 2008. `http://www.mathematik.uni-karlsruhe.de/ianm2/~kulisch/media/lettersieee754r12.pdf`.

[26] Ulrich Kulisch. Complete Interval Arithmetic and its Implementation on the Computer, 2008. `http://www.mathematik.uni-karlsruhe.de/ianm2/~kulisch/media/arjpkx.pdf`.

[27] Branimir Lambov. Interval Arithmetic Using SSE-2. Lecture Notes in Computer Science, 5045:102–113, 2008.

[28] Michael Lerch, German Tischler, and Jürgen Wolff von Gudenberg. filib++ Interval Library Specification and Reference Manual. Technical Report 279, Institut für Informatik, Universität Würzburg, August 2001.

[29] Michael Lerch, German Tischler, Jürgen Wolff von Gudenberg, Werner Hofschuster, and Walter Krämer. filib++, a fast interval library supporting containment computations. ACM Trans. Math. Softw., 32(2):299–324, 2006.

[30] Michael Lerch and Jürgen Wolff von Gudenberg. Expression Templates for Dot Product Expressions. Reliable Computing, 5(1):69–80, 1999.

[31] Svetoslav Markov and Dalcidio Claudio. On the Interval Arithmetic in Midpoint-Radius Form. In Mathematics and Education in Mathematics, volume 33, pages 434–439. Inst. Math. and Informatics, BAS, 2004.

[32] David Monniaux. The pitfalls of verifying floating-point computations. ACM Trans. Program. Lang. Syst., 30(3):1–41, 2008.

[33] Nathan C. Myers. Traits: a new and useful template technique. C++ Report, June 1995.

[34] Arnold Neumaier. Vienna proposal for interval standardization, Version 3.0, November 2008. in [19].

[35] Arnold Neumaier and Svetoslav Markov. Diskussion über Mittelpunkt-Radius Darstellung. Mailingliste `STDS-1788@LISTSERV.IEEE.ORG`, November 2008. `http://grouper.ieee.org/groups/1788/email/msg00050.html`.

[36] Archiv: Mailingliste `STDS-1788@LISTSERV.IEEE.ORG`, February 2009. `http://grouper.ieee.org/groups/1788/email/`.

[37] IEEE Task P754. ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic. IEEE, New York, August 1985.

[38] IEEE Task P754. IEEE 754-2008, Standard for Floating-Point Arithmetic. IEEE, New York, August 2008.

[39] Sylvain Pion. C++ standardization work, February 2009. http://www-sop.inria.fr/members/Sylvain.Pion/cxx/.

[40] Sylvain Pion, Hervé Brönnimann, and Guillaume Melquiond. The Boost Interval Arithmetic Library. In Proc. 5th Conference on Real Numbers and Computers, pages 65–80, 2003.

[41] Sylvain Pion, Hervé Brönnimann, and Guillaume Melquiond. A Proposal to add Interval Arithmetic to the C++ Standard Library, November 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2137.pdf.

[42] Sylvain Pion, Hervé Brönnimann, and Guillaume Melquiond. Bool_set: multi-valued logic, June 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2046.pdf.

[43] Sylvain Pion, Hervé Brönnimann, and Guillaume Melquiond. The design of the Boost interval arithmetic library. Theoretical Computer Science, 351(1):111–118, 2006.

[44] J. D. Pryce and G. F. Corliss. Interval Arithmetic with Containment Sets. Computing, 78(3):251–276, 2006.

[45] T. Rauber and G. Rünger. Parallele Programmierung (eXamen.press). Springer Verlag, 2007.

[46] T. Rauber and G. Rünger. Multicore: Parallele Programmierung (Informatik im Fokus). Springer Verlag, 2008.

[47] The RealLib Project, February 2009. http://www.brics.dk/~barnie/RealLib/.

[48] Bjarne Stroustrup. Die C++ Programmiersprache. Addison-Wesley Verlag, München, 4. edition, 2000.

[49] David Vandevoorde and Nicolai M. Josuttis. C++ Templates: The Complete Guide. Addison-Wesley Professional, November 2002.

[50] Todd L. Veldhuizen. Expression templates. C++ Report, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[51] Todd L. Veldhuizen. Techniques for Scientific C++. Technical Report 542, Indiana University Computer Science, August 2000.

[52] Jürgen Wolff von Gudenberg. Algorithmen, Datenstrukturen, Funktionale Programmierung. Addison-Wesley, 1996.

[53] Jürgen Wolff von Gudenberg. Rechnerarithmetik, Skript zur Vorlesung. Universität Würzburg, January 2007.

[54] Jürgen Wolff von Gudenberg. Interval Arithmetic and Standardization. In Annie Cuyt, Walter Krämer, Wolfram Luther, and Peter Markstein, editors, Numerical Validation in Current Hardware Architectures, number 08021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[55] G. William Walster, E. R. Hansen, and J. D. Pryce. Extended Real Intervals and the Topological Closure of Extended Real Relations. Technical report, Sun Microsystems, 2000.

**Danksagungen**

Auf dieser Seite möchte ich mich bei all jenen bedanken, welche durch ihre Unterstützung diese Diplomarbeit ermöglicht haben. Besonderer Danke gebührt hierbei folgenden Personen:

Meinen Eltern **Edeltraud** und **Ernst Nehmeier**, welche mir durch ihre Unterstützung ein Studium überhaupt erst ermöglicht haben.

Meiner Freundin **Ursula Schmidt** für ihre Unterstützung und Motivation. Vor allem in den letztend Wochen dieser Arbeit.

**Professor Dr. Jürgen Wolff von Gudenberg** für die gute Betreuung während meines Studiums und dieser Diplomarbeit.

**Dr. German Tischler** für die Tipps zu C++ und filib++.

**Fabian Haupt** und **Benjamin Ruderisch** für das Korrketurlesen sowie für fachliche Ratschläge.

**Joachim Lusiardi** für Tipps zum Schreiben der Diplomarbeit sowie die Zurverfügungstellung der Formatvorlage seiner Diplomarbeit.

**Erklärung**

Ich erkläre hiermit, dass ich die vorliegende Diplomarbeit selbständig und ohne unzulässige, fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die bildlichen Darstellungen, Tabellen, Quelltexte und Graphen habe ich selbst angefertigt.

Würzburg, den 16. Februar 2009