Programmanalyse des XRTL Zwischencodes

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes
von

Werner Backes

Saarbrücken
2004

Tag des Kolloquiums:	7. Januar 2005
Dekan:	Prof. Jörg Eschmeier
Prüfungsausschuss :	Prof. Holger Hermanns (Vorsitzender)
	Prof. Andreas Podelski (Berichterstatter)
	Prof. Reinhard Wilhelm (Berichterstatter)
	Dr. Patrick Maier (Akademischer Mitarbeiter)

Kurzzusammenfassung

In dieser Arbeit stellen wir das Analysetool xGCC vor. Dieses Tool analysiert den XRTL Zwischencode, um Sicherheitseigenschaften, z.B. Speicherüberlauf, die Division durch Null und die Verwendung von uninitialisierten Variablen oder Speicherstücken, zu verifizieren. XRTL ist eine von uns entwickelte Erweiterung der Register Transfer Language (RTL). Dieser sprachunabhängige Zwischencode wird von Frontends der GNU Compiler Collection (GCC) für verschiedene Programmiersprachen, wie z.B. C, C++, Java und Fortran 77, erzeugt. xGCC unterstützt diese Programmiersprachen, da sich unsere Compilermodifikationen auf den sprachunabhängigen Teil des Compilers beschränken.

Für die Analyse von XRTL setzen wir die abstrakte Interpretation ein. Wir verwenden Listen von gültigen Intervallen, um Mengen von benötigten Registern und Speicherstücken darzustellen. Gültige Intervalle sind Intervalle für die zusätzlichen Bedingungen gelten, um die Implementierung der XRTL Operationen zu erleichtern. Die Präzision der Approximation ist durch die Listenlänge einstellbar. Wir zeigen, wie einfache und parallele XRTL Anweisungen abgearbeitet werden. Wir leiten Constraints aus der Bedingung von Verzweigungen ab, um den Suchraum einzuschränken. Wir verwenden das Widening/Narrowing, um die Fixpunktberechnung für XRTL Scheifen zu beschleunigen.

Wir stellen die Implementierung von xGCC vor und erläutern die getroffenen Designentscheidungen. Mit Hilfe ausgewählter Beispiele demonstrieren wir verschiedene Klassen von Fehlern, die das Analysetool xGCC entdeckt. Wir untersuchen das Laufzeitverhalten der Analyse mit Beispielen aus den *Numerical Recipes in C* und stellen verschiedene Optimierungen vor.

Abstract

We present the xGCC analysis tool for the verification of safety properties of the XRTL intermediate code. These properties include the absence of buffer overflow, division by zero and the use of uninitialized variables and memory. XRTL is our extension of the Register Transfer Language (RTL). This language independent intermediate code is generated by frontends of the GNU Compiler Collection (GCC) for the programming languages C, C++, Java and Fortran 77. These programming languages are supported by xGCC since we have modified only the language independent part of the compiler.

We apply abstract interpretation for the analysis of XRTL. Lists of valid intervals are used for the abstraction of sets of registers and memory blocks. Valid intervals are intervals with additional contraints that simplify the implementation of the XRTL operations. The precision of the abstraction is parameterized by the list length. We describe the interpretation of sequential and parallel XRTL instructions. We take branching conditions into account for restricting the search space, and apply the Widening/Narrowing techniques to speed up the fixpoint computation for XRTL loops.

We present the implementation of xGCC and explain the tool design. We demonstrate xGCC analysis tool on a collection of examples. We analyse the tool performance on examples from the *Numerical Recipes in C*, and introduce several optimizations.

Danksagung

Ich danke Prof. Dr. Andreas Podelski für die hervorragende Betreuung und das interessante Dissertationsthema. Ich bin dankbar für die Möglichkeit als Doktorand am Max-Planck-Institut für Informatik in der Gruppe „Logik der Programmierung“ von Prof. Dr. Harald Ganzinger zu arbeiten. Für das Korrekturlesen meiner Arbeit möchte ich mich bei Patrick Maier, Andrey Rybalchenko, Stephan Groß, Sandra Steinbrecher und Susanne Wetzel bedanken. Ich möchte mich bei allen Kolleginnen und Kollegen der Arbeitsgruppe „Logik der Programmierung“ für die freundliche Arbeitsumgebung und die vielen hilfreichen Diskussionen bedanken. Besonders danken möchte ich Andrey Rybalchenko für viele gute Ideen und Anmerkungen, Patrick Maier für seine Geduld, Uwe Waldmann für seine Latex Tipps und Hans de Nivelle für Gespräche über das Programmieren. Danken möchte ich auch Dalibor Topic, der die von mir benötigte Software installiert hat.

Zuletzt möchte ich mich vor allem bei meinen Eltern und Geschwistern für die kontinuierliche Unterstützung während meines ganzen Promotionsstudiums bedanken.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Problem von Fehlern in Programmen und das Auffinden von solchen, existiert seit der Einführung des Computers. Die Testphase nimmt in großen Softwareprojekten einen nicht unerheblichen Teil der gesamten Projektzeit ein. Die Methoden, die man üblicherweise zum Debuggen eines Programms verwendet, testen unterschiedliche Eingaben und vergleichen die Ausgabe oder das Verhalten des Programms mit dem erwarteten Ergebnis. Um einen Fehler zu finden, müssen wir die Eingabe finden, die diesen Fehler erzeugt.

Die Abstrakte Interpretation erlaubt uns das Feststellen von Fehlern oder von möglicherweise auftretenden Fehlern, ohne das Programm mit einer vorgegebenen Eingabe auszuführen. Die meisten Tools beschränken sich bei der Analyse auf eine Programmiersprache oder nur auf eine Teilmenge. In vielen Fällen ist Zeigerarithmetik verboten und der Zugriff auf die Bits und Bytes von Datentypen eingeschränkt. Wir wollen jedoch alles erlauben, was der verwendete Compiler übersetzen kann. Unser Ziel ist die Entwicklung eines Analysetools, dass den Programmierer in keiner Weise in seinem Programmierstil einschränkt und dabei verschiedene Programmiersprachen unterstützt. Wir führen zu diesem Zweck die Zwischensprache Extended Register Transfer Language (XRTL) ein. Hierbei handelt es sich um eine erweiterte und modifizierte Version der Zwischensprache, die in der GNU Compiler Collection verwendet wird. Die GNU Compiler Collection ist in der Open Source Bewegung weit verbreitet. Es existieren so genannte Frontends für die Programmiersprachen C, C++, Fortran 77, Objective C und Java. Die Verfügbarkeit der Quellen ermöglicht eine notwendige Veränderung des sprachunabhängigen Teils des Compilers. Es sind keine Modifikationen am eingesetzten Front- oder Backend notwendig. Eine Analyse von Programmen aller unterstützten Hochsprachen ist möglich.

Der konkrete und abstrakte Domain besteht aus einer endlichen Menge von Registern und Speicherstücken, die in der XRTL Zwischensprache, die einer maschinenunabhängigen Assemblersprache ähnelt, verwendet werden. Die Basis für den konkreten und abstrakten Domain sind endliche Teilmengen von ganzen und reellen Zahlen. Die Basisoperationen werden auf Listen von Intervallen, welche eine Abstraktion der Menge von Werten für einen bestimmten Datentyp ist, durchgeführt. Die Länge der Liste ist fest, aber beliebig wählbar um eine Verfeinerung der Abstraktion, durch Erhöhung der maximalen Listenlänge, zu erlauben. Die XRTL Basisoperationen erlaubten einen systemnahen Zugriff auf einzelne Bits und Bytes. Wir werden den Begriff des gültigen Intervalls und der Liste von gültigen In-

tervalle einführen und zeigen, dass wir Intervalle in gültige Intervalle aufteilen können. Diese Vereinfachung hilft uns das Problem der Unterscheidung zwischen vorzeichenlosen und vorzeichenbehafteten, ganzen Zahlen zu lösen. Wir werden Methoden für die Erkennung von Verzweigungen und Schleifen einführen, da Anfang und Ende von einzelnen Teilen einer Verzweigung oder Schleife nicht offensichtlich sind. Wir berechnen Fixpunkte um Schleifen abzuarbeiten. Wir verwenden Constraints für die Ausführung einzelner Zweige einer Verzweigung. Wir führen auftretende Funktionsaufrufe mit Hilfe von Vorberechnungen aus.

Unser Analysetool führt eine konservative Analyse durch. Es sucht nach Speicherzugriffsfehlern, wie z.B. Überlauf oder Unterlauf des Stacks. Wir decken arithmetische Fehler auf, wie z.B. eine Division durch Null, und stellen den Gebrauch von uninitialisierten Variablen oder Speicherstücken fest. Wir zeigen an ausgewählten Beispielen, die Vorteile und Nachteile unseres Ansatzes und demonstrieren mit praktischen Experimenten aus den *Numerical Recipes in C*, wie wir die Laufzeit des Analysetools reduzieren können.

Kapitel 2 gibt eine Einführung in die Register Transfer Language, die Basis für XRTL. Mit ausgewählten C Beispielen werden wir demonstrieren, wie eine Programmiersprache in XRTL übersetzt wird. Wir diskutieren die Vor- und Nachteile der Analyse einer solchen, systemnahen Sprache und stellen die notwendige Erweiterung von RTL zu XRTL vor. Wir beschreiben den Aufbau von Verzweigungen und Schleifen und zeigen, wie wir diese entdecken können. Wir demonstrieren Funktionsaufrufe einschließlich Parameterübergabe und zeigen wie Ergebnisse zurückgeliefert werden.

Kapitel 3 führt den verwendeten abstrakten und konkreten Domain bestehend aus Registern unterschiedlicher Datentypen und Speicherstücken verschiedener Länge und Struktur ein. Wir beschreiben die entsprechenden Abstraktions- und Konkretisierungsfunkionen. Wir führen die Abstraktion mit Hilfe von Listen von gültigen Intervallen ein und beschreiben die, für die Analyse der XRTL Zwischensprache benötigen Operationen. Wir führen Zeiger und Speicherstücke mit den dazugehörigen Operationen ein.

Kapitel 4 beschreibt die für die Implementierung notwendigen Verfeinerungen. Wir stellen die von XRTL unterstützten Grunddatentypen und ihre Darstellung vor. Den Begriff des gültigen Intervalls schränken wir aus technischen Gründen weiter ein und zeigen, dass alle in Kapitel 3 vorgestellten Operationen weiterverwenden werden können. Den Zugriff auf die Darstellung benötigen wir für die auf Speicherstücken definierten Zugriffsfunktionen.

Kapitel 5 beschreibt die Abstrakte Interpretation der XRTL Sprache. Wir zeigen, wie einfache Instruktionen abgearbeitet werden und wie wir mit Verzweigungen umgehen. Für die Interpretation von Schleifen verwenden wir die Fixpunktberechnung. Das Widening/Narrowing nutzen wir für die Beschleunigung dieser Berechnung. Wir machen deutlich, wie Funktionsaufrufe abgearbeitet werden.

Kapitel 6 beschreibt die Vor- und Nachteile unseres Ansatzes. Wir stellen andere statische und nicht-statische Analysetools vor. Wir beschreiben die von uns durchgeführten Compilermodifikationen und die Implementierung des Analysetools xGCC. Die von unserem Analysetool entdeckten Fehler demonstrieren wir anhand ausgewählter Beispiele verschiedener Programmiersprachen. Wir untersuchen das Laufzeitverhalten des Analysetools mit praktischen Tests und stellen einige Verbesserungen vor.

Kapitel 2

Extended Register Transfer Language

In diesem Kapitel geben wir eine kurze Einführung in die Sprache XRTL, einer Erweiterung der Register Transfer Language (RTL), welche innerhalb der GNU Compiler Collection (GCC) als Zwischensprache verwendet wird. Wir beschreiben den Aufbau der GCC, die aus einem Frontend besteht, der die Hochsprache in RTL übersetzt, dem sprachunabhängigen Teil, innerhalb dessen Optimierungen durchgeführt werden und dem so genannten Backend, welches RTL in Maschinensprache übersetzt. Unsere Modifikationen beschränken sich auf den sprachunabhängigen Teil der GCC. Wir stellen die Sprache RTL vor, indem wir Register unterschiedlichen Typs und Größe, sowie einfache Speicher- und Registeroperationen beschreiben. Anhand kleiner Programmfragmente zeigen wir, wie einfache und komplexe Konstrukte übersetzt werden, erklären den Aufbau von Verzweigungen und Schleifen und demonstrieren die Funktionsweise der Parameterübergabe bei Funktions- und Prozeduraufrufen. Wir entwickeln Algorithmen zum Erkennen und Auftrennen von Verzweigungen in Bedingung und die dazugehörigen Zweige. Die verschiedenen Schleifenarten sind durch so genannte Labels gekennzeichnet. Wir zeigen die Notwendigkeit der Erweiterung von RTL zu XRTL und beschreiben diese.

2.1 Der Aufbau der GCC

Die Abbildung 2.1 zeigt sehr vereinfacht den Aufbau der GCC bestehend aus den Teilen Frontend, Sprachunabhängiger Teil und Backend. Das Frontend übersetzt die Programmiersprache wie C, C++, Objective C, Java und Fortran 77 in RTL. Das Programm wird hierbei Zeile für Zeile in RTL übersetzt. Aus diesem RTL Programm wird dann im sprachunabhängigem Teil, mit Hilfe des maschinenabhängigen Backends, ein ausführbares Programm erzeugt. Dieser Aufbau der GCC erleichtert die Portierung auf neue Architekturen und die Unterstützung weiterer Programmiersprachen. Schwieriger ist die Generierung von effizientem Code, da durch die Trennung von Frontend, sprachunabhängigem Teil und Backend alle Optimierungen auf Basis der Zwischensprache durchgeführt werden müssen. Kommerzielle Compiler, die überwiegend für eine Architektur und Programmiersprache optimiert sind, generieren effizientere Programme. Für weitergehende Informationen über den internen Aufbau der GCC verweisen wir auf [32, 33].

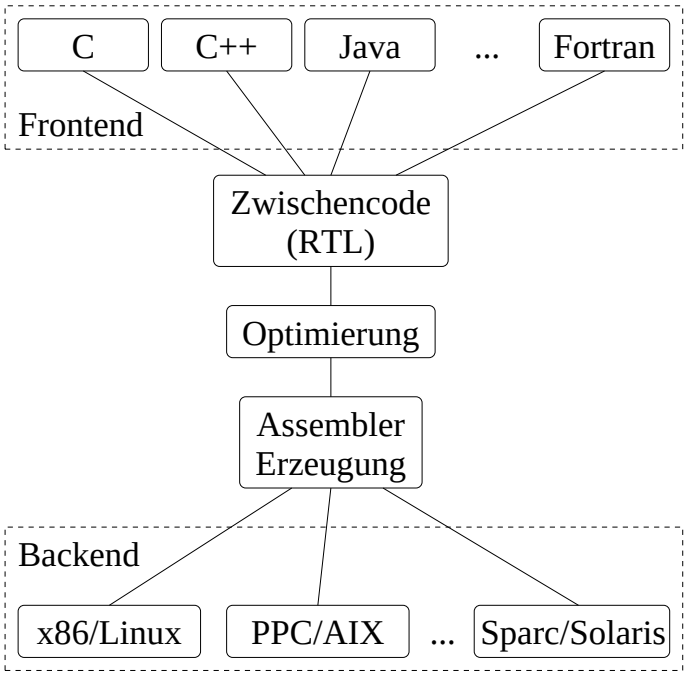


Abbildung 2.1: Aufbau des GCC

2.2 Register Transfer Language

Wir geben in diesem Abschnitt eine Einführung in die Register Transfer Language (RTL). Für weitergehende Informationen verweisen wir auch hier auf [32, 33]. Die Sprache wurde jedoch in einigen Fällen modifiziert und vereinfacht. Wir gehen hier nur auf die Operationen ein, die in der Praxis (auf unseren Testsystemen) vorkommen. Auf die genaue Bedeutung der einzelnen Ausdrücke gehen wir in Kapitel 5 ein. Die von uns verwendeten Testsysteme sind i386/Linux, i386/FreeBSD und Sparc/Solaris. Es kann auf den verschiedenen Systemen zu kleinen Unterschieden in der RTL-Übersetzung kommen.

Die im nachfolgenden beschriebenen Ausdrücke arbeiten in verschiedenen Modi, d.h. das Ergebnis, der durch den Ausdruck beschriebenen Operation, ist ein Wert eines bestimmten Datentyps.

RTL unterstützt die folgenden ganzzahligen Modi bzw. Datentypen:

Modus	QImode	HImode	SImode	DImode	*TImode*	*OImode*
Kürzel	`:QI`	`:HI`	`:SI`	`:DI`	`:TI`	`:OI`
Größe in Byte	1	2	4	8	16	32

und die nachfolgenden Fließkommamodi bzw. Datentypen

Modus	*QFmode*	*HFmode*	SFmode	DFmode	XFmode	TFmode
Kürzel	`:QF`	`:HF`	`:SF`	`:DF`	`:XF`	`:TF`
Größe in Byte	1	2	4	8	12	16

Die hier beschriebenen Modi *TImode, OImode* und *QFmode, HFmode* werden von unserem Analysetool momentan nicht unterstützt. Ein so genannter Blockmodus *BLKmode* mit Kürzel :BLK tritt nur in Verbindung mit speziellen Operationen auf, die in Kapitel 5 genauer beschrieben werden. Wir verwenden die Bezeichnung „Größe eines Ausdrucks x“ für die Anzahl der Bytes, die der Datentyp des aus x resultierenden Wertes benötigt. Mit Hilfe dieser Ordnung können wir Ausdrücke vergleichen.

2.2.1 Instruktionen

In RTL existieren sechs verschiedene Arten von Instruktionen. Die ganzzahligen Werte x, y und z beschreiben die Nummer der Instruktion, sowie die Nummer der Vorgänger- und der Nachfolgerinstruktion.

(insn x y z exp**)**
: Diese Instruktion erlaubt keinerlei Sprünge und Funktionsaufrufe, d.h. exp darf den Programmzähler (pc) nicht verändern.

(jump_insn x y z exp**)**
: Diese Instruktion verändert den Programmzähler. Verzweigungen sind möglich.

(call_insn x y z exp**)**
: Diese Instruktion beschreibt einen Funktionsaufruf.

(code_label x y z no**)**
: Diese Instruktion beschreibt ein Sprungziel. Diese Ziele werden mit Hilfe von no durchnummeriert.

(barrier x y z**)**
: Diese Instruktion darf nicht ausgeführt werden, ist also eine Art Barriere. Mit Sprunginstruktionen kann man Barrieren umgehen.

(note x y z $info$**)**
: Diese Instruktion dient als Platzhalter für zusätzliche Informationen, wird daher nicht ausgeführt.

Die folgenden Ausdrücke werden innerhalb von Instruktionen als exp verwendet.

(set $lval$ x**)**
: Dieser Ausdruck beschreibt die Zuweisung des Ausdrucks x an $lval$. Die Modi von $lval$ und x müssen gleich sein. Für $lval$ sind nur Register und Speicherausdrücke erlaubt. Wird der Programmzähler (pc) als $lval$ verwendet, dann ist diese Zuweisung Teil einer Sprunginstruktion.

(return)
: Dieser Ausdruck entspricht (set (pc) (return)) und beschreibt das Ende einer Funktion.

(call $function\ nargs$**)**
Dieser Ausdruck beschreibt einen Funktionsaufruf. $function$ beschreibt die aufgerufene Funktion und $nargs$ steht entweder für die Anzahl der Registerargumente oder die Anzahl der Bytes auf dem Stack den die Argumente benötigen.

(parallel [$x_0\ x_1\ \ldots x_n$**])**
Dieser Ausdruck beschreibt die parallele Ausführung von $x_0, x_1 \ldots x_n$.

(clobber x**)**
Dieser Ausdruck beschreibt die Veränderung oder die mögliche Veränderung eines Register- oder Speicherausdrucks x. Die genaue Bedeutung ist jedoch kontextabhängig.

(use x**)**
Dieser Ausdruck beschreibt die Verwendung des Wertes des Ausdrucks x. Der Compiler darf diesen Wert während der Optimierungen nicht entfernen, da er vorhergehende oder nachfolgende Instruktionen beeinflusst. Auch hier ist die Bedeutung kontextabhängig.

2.2.2 Register und Speicher

Die Register- und Speicherausdrücke werden als $lval$ in den Zuweisungsausdrücken verwendet. Wir können diese allerdings auch als Teilausdruck in anderen Ausdrücken verwenden.

(reg:m n**)**
Dieser Ausdruck beschreibt das Register mit Nummer n in Modus m. In RTL existiert nur eine endliche Anzahl von Registern eines Typs.

(subreg:m $reg\ bytenum$**)**
Dieser Ausdruck beschreibt den Zugriff auf ein Register reg wobei der Modus von reg und m verschieden sind. m beschreibt den Modus des Ausdrucks und $bytenum$ das Byteoffset des Zugriffs. Die Größe von reg kann größer und kleiner als die Größe von m sein.

(pc)
Dieser Ausdruck beschreibt den Programmzähler. Alle Instruktionen die (pc) verwenden sind Sprunginstruktionen.

(mem:m $addr$**)**
Diese Ausdruck beschreibt den Zugriff auf eine Speicherstück in Modus m auf das $addr$ zeigt.

2.2.3 Konstante Ausdrücke

Diese einfachsten RTL Ausdrücke beschreiben konstante Werte.

(const_int:m i**)**
Dieser Ausdruck beschreibt eine ganze Zahl i im ganzzahligen Modus m.

(const_double:m d**)**
Dieser Ausdruck beschreibt eine Fließkommazahl d im Fließkommamodus m.

(symbol_ref:m $symbol$**)**
Dieser Ausdruck beschreibt einen Zeiger auf einen Wert oder zusammengesetzte Struktur $symbol$. Der Modus m ist maschinenabhängig.

(label_ref $label$**)**
Dieser Ausdruck beschreibt ein Sprungziel, dass durch eine code_label-Instruktion beschrieben wird.

(const:m exp**)**
Dieser Ausdruck beschreibt einen Konstante in Modus m. exp hat Modus m und jeder Teilausdruck ist konstant.

(high:m exp**)**
Dieser Ausdruck in Modus m beschreibt die oberen Bits eines konstanten Teilausdrucks exp in Modus m. Die Anzahl der Bits ist maschinenabhängig.

2.2.4 Arithmetische Ausdrücke

Die nachfolgenden Ausdrücke existieren, sofern nicht anders angegeben, für alle beschriebenen Modi bzw. Datentypen. Die Modi der Argumente x und y müssen m entsprechen. Auf den Unterschied von vorzeichenloser und vorzeichenbehafteter Betrachtungsweise von ganzzahligen Werten gehen wir in Kapitel 3 und 4 ein.

(plus:m x y**)**
Dieser Ausdruck beschreibt die Summe von x und y.

(lo_sum:m x y**)**
Dieser Ausdruck beschreibt die Summe der unteren Bits von x und y. Die Anzahl der Bits ist maschinenabhängig.

(minus:m x y**)**
Dieser Ausdruck beschreibt die Differenz von x und y.

(ss_plus:m x y**)**
(ss_minus:m x y**)**
Diese Ausdrücke beschreiben die Summe bzw. Differenz von x und y. Im Falle eines Überlaufs wird die so genannte Signed Saturation durchgeführt. Diese Ausdrücke sind nur für ganzzahlige Modi definiert.

(us_plus:m x y**)**
(us_minus:m x y**)**
Diese Ausdrücke beschreiben die Summe bzw. Differenz von x und y. Im Falle eines Überlaufs wird die so genannte Unsigned Saturation durchgeführt. Diese Ausdrücke sind nur für ganzzahlige Modi definiert.

(neg:m x**)**
Dieser Ausdruck beschreibt die Negation von x.

(mult:m x y**)**
Dieser Ausdruck beschreibt das Produkt von x und y.

(div:m x y**)**
Dieser Ausdruck beschreibt die Division von x und y. Wir betrachten vorzeichenbehaftete Werte.

(udiv:m x y**)**
Dieser Ausdruck beschreibt die Division von x und y. Wir betrachten vorzeichenlose Werte. Dieser Ausdruck ist nur für ganzzahlige Modi m definiert.

(mod:m x y**)**
Dieser Ausdruck beschreibt den Rest der Division von x und y. x und y werden als vorzeichenbehaftete Werte betrachtet. Dieser Ausdruck ist nur für ganzzahlige Modi m definiert.

(umod:m x y**)**
Dieser Ausdruck beschreibt den Rest der Division von x und y. x und y werden als vorzeichenlose Werte betrachtet. Dieser Ausdruck ist nur für ganzzahlige Modi m definiert.

(smin:m x y**)**
(smax:m x y**)**
Dieser Ausdruck beschreibt das Minimum bzw. Maximum von x und y. Wir betrachten vorzeichenbehaftete Werte.

(umin:m x y**)**
(umax:m x y**)**
Diese Ausdrücke beschreiben das Minimum bzw. Maximum von x und y, wobei wir vorzeichenlose Werte betrachten. Diese Ausdrücke sind nur für ganzzahlige Modi m definiert.

(abs:m x**)**
Dieser Ausdruck beschreibt den Absolutbetrag von x.

(sqrt:m x**)**
Dieser Ausdruck beschreibt die Wurzel von x.

2.2.5 Bitoperationen

Die nachfolgenden Bitoperationen sind nur für ganzzahlige Modi definiert. c beschreibt einen konstanten, ganzzahligen Wert und keinen Teilausdruck. Die Modi von x und y müssen m entsprechen.

(not:m x**)**
Dieser Ausdruck beschreibt das bitweise Komplement von x.

(and:m x y**)**
Dieser Ausdruck beschreibt das bitweise und von x und y.

(ior:m x y**)**
Dieser Ausdruck beschreibt das bitweise oder von x und y.

(xor:m x y**)**
Dieser Ausdruck beschreibt das bitweise exklusiv-oder von x und y.

(ashift:m x c**)**
(ashiftrt:m x c**)**
Dieser Ausdruck beschreibt den arithmetischen links- bzw. rechtshift von x um c Bits.

(lshift:m x c**)**
(lshiftrt:m x c**)**
Dieser Ausdruck beschreibt den bitweisen links- bzw. rechtshift von x um c Bits.

(rotate:m x c**)**
(rotatert:m x c**)**
Dieser Ausdruck beschreibt das links- bzw. rechtsrotieren von x um c Bits.

(ffs:m x**)**
Dieser Ausdruck beschreibt den Index des niedrigsten 1-Bits von x. m beschreibt einen ganzzahligen Datentyp.

2.2.6 Vergleichsausdrücke

Die nachfolgenden Vergleichsausdrücke existieren für alle vorgestellten Modi bzw. Datentypen. Die Modi der Argumente x und y müssen gleich sein, können aber von m verschieden sein.

(compare:m x y**)**
Dieser Ausdruck beschreibt das Vorzeichen der Differenz von x und y. Das Ergebnis ist -1, 0 oder 1.

(eq:m x y**)**
Dieser Ausdruck beschreibt den Vergleich von x und y. Sind diese gleich, so ist das Ergebnis 1, andernfalls 0.

(ne:m x y**)**
Dieser Ausdruck beschreibt den Vergleich von x und y. Sind diese verschieden, so ist das Ergebnis 1, andernfalls 0.

(gt:m x y**)**
(gtu:m x y**)**
Diese Ausdrücke liefern 1 falls x größer als y ist, 0 andernfalls. gtu beschreibt die vorzeichenlose Variante von gt die nur für ganzzahlige Datentypen existiert.

(lt:m x y**)**
(ltu:m x y**)**
Diese Ausdrücke liefern 1 falls x kleiner als y ist, 0 andernfalls. ltu beschreibt die vorzeichenlose Variante von lt die nur für ganzzahlige Datentypen existiert.

(ge:m x y**)**
(geu:m x y**)**
Diese Ausdrücke liefern 1 falls x größer oder gleich als y ist, 0 andernfalls. geu beschreibt die vorzeichenlose Variante von ge die nur für ganzzahlige Datentypen existiert.

(le:m x y**)**
(leu:m x y**)**
Diese Ausdrücke liefern 1 falls x kleiner oder gleich als y ist, 0 andernfalls. leu beschreibt die vorzeichenlose Variante von le die nur für ganzzahlige Datentypen existiert.

Die nachfolgenden Ausdrücke beschreiben keine Vergleichsoperationen sondern verwenden diese als Teilausdrücke.

(if_then_else $cond$ $then$ $else$**)**
Dieser Ausdruck liefert den Teilausdruck $then$ zurück, falls das Ergebnis des Vergleichsausdruck $cond$ ungleich 0 ist, andernfalls wird $else$ zurückgeliefert. Dieser Ausdruck ist Teil einer Sprunginstruktion.

(cond [$test_1$ $value_1$ **...**$test_n$ $value_n$**]** $default$**)**
Dieser Ausdruck liefert $value_i$ zurück, falls $test_i$ einen Wert ungleich 0 zurückliefert, für $1 \leq i \leq n$. Sollte keine der Teilbedingungen erfüllbar sein, so wird $default$ zurückgeliefert.

2.2.7 Bitfelder

Die nachfolgenden Ausdrücke sind nötig um auf einzelne Bits von Bitfeldern lesend und schreibend zuzugreifen.

(sign_extract:m loc $size$ pos**)**
Dieser Ausdruck beschreibt den vorzeichenbehafteten Zugriff auf ein Bitfeld. loc kann ein Speicher- oder Registerausdruck sein. $size$ steht für die Anzahl der Bits und pos gibt die Startposition an. Der Modus von loc entspricht m.

(zero_extract:m loc $size$ pos**)**
Dieser Ausdruck beschreibt den vorzeichenlosen Zugriff auf ein Bitfeld. loc kann ein Speicher- oder Registerausdruck sein. $size$ steht für die Anzahl der Bits und pos gibt die Startposition an. Der Modus von loc entspricht m.

2.2.8 Konvertierung

Wir haben bis jetzt nur Ausdrücke kennen gelernt, deren Ergebnis einen bestimmten festgelegten Datentyp hat. Die direkte Zuweisung von Ausdrücken unterschiedlichen Datentyps ist nicht erlaubt, sondern erfordert eine explizite Konvertierung. Hierbei kann es je nach Datentyp und Konvertierungsroutine zu Präzisionsverlusten kommen. Für einen Ausdruck x und einen Modus m beschreibt $|x|$ die Größe des entsprechenden Datentyps und $|m|$ die Größe des Datentyps, der m entspricht.

`(sign_extend:m` x`)`
: Diese Ausdruck beschreibt die Konvertierung von x zu einem Wert in Modus m unter Beibehaltung des Vorzeichens. Die Modi von x und m sind ganzzahlig mit $|x| < |m|$.

`(zero_extend:m` x`)`
: Diese Ausdruck beschreibt die Konvertierung von x zu einem Wert in Modus m. Das Vorzeichen geht verloren. Die Modi von x und m sind ganzzahlig mit $|x| < |m|$.

`(float_extend:m` x`)`
: Dieser Ausdruck beschreibt die Konvertierung von x zu einem Wert in Modus m. x und m sind Fließkommamodi mit $|x| < |m|$.

`(truncate:m` x`)`
: Diese Ausdruck beschreibt die Konvertierung von x zu einem Wert in Modus m. Die Modi von x und m sind ganzzahlig mit $|x| > |m|$.

`(ss_truncate:m` x`)`
`(us_truncate:m` x`)`
: Diese Ausdrücke entsprechen `(truncate:m` x`)` mit dem Unterschied das im Falle eines Überlaufs eine so genannte Signed bzw. Unsigned Saturation erfolgt.

`(float_truncate:m` x`)`
: Diese Ausdruck beschreibt die Konvertierung von x zu einem Wert in Modus m. x und m sind Fließkommamodi mit $|x| > |m|$.

`(float:m` x`)`
: Dieser Ausdruck beschreibt die Konvertierung eines ganzzahligen x zu einem Wert in Fließkommamodus m unter Berücksichtigung des Vorzeichens.

`(unsigned_float:m` x`)`
: Dieser Ausdruck beschreibt die Konvertierung eines ganzzahligen, vorzeichenlosen x zu einem Wert in Fließkommamodus m.

`(fix:m` x`)`
: Dieser Ausdruck beschreibt die Konvertierung von x in Fließkommamodus in einen ganzzahligen Modus m unter Berücksichtigung des Vorzeichens.

`(unsigned_fix:m` x`)`
: Dieser Ausdruck beschreibt die Konvertierung von x in Fließkommamodus zu einem ganzzahligen, vorzeichenlosen Wert in Modus m.

2.2.9 Ausdrücke mit Seiteneffekten

Die sechs nachfolgenden Ausdrücke treten nur in Verbindung mit Speicherausdrücken auf. Der verwendete Speicherausdruck hat Modus n.

(pre_dec:m x**)**
Dieser Ausdruck liefert x zurück, nachdem x um $|n|$ vermindert wurde.

(pre_inc:m x**)**
Dieser Ausdruck liefert x zurück, nachdem x um $|n|$ erhöht wurde.

(post_dec:m x**)**
Dieser Ausdruck liefert x zurück. Danach wird x um $|n|$ vermindert.

(post_inc:m x**)**
Dieser Ausdruck liefert x zurück. Danach wird x um $|n|$ erhöht.

(post_modify:m x y**)**
Dieser Ausdruck liefert x zurück. Danach wird x durch y verändert.

(pre_modify:m x y**)**
Dieser Ausdruck liefert x zurück nachdem x durch y verändert wurde.

2.3 Beispiele

Nachdem die einzelnen Teilausdrücke vorgestellt und erklärt wurden, geben wir einige Beispiele für die Übersetzung einer Hochsprache in RTL an. Wir beschränken uns hier auf C Programme. Im Abschnitt 6.3.1 finden sich weitere Beispiele für die Übersetzung von C und C++ in RTL.

Das nachfolgende Programmfragment

```
int a, b, c;

c = a + b;
```

wird auf Sparc/Solaris folgendermaßen in RTL übersetzt.

```
(insn 10 8 12  (set (reg:SI 108)
                    (mem:SI (plus:SI (reg:SI 103) (const_int -4)))))
(insn 12 10 14 (set (reg:SI 109)
                    (mem:SI (plus:SI (reg:SI 103) (const_int -8)))))
(insn 14 12 16 (set (reg:SI 110) (plus:SI (reg:SI 108) (reg:SI 109))))
(insn 16 14 17 (set (mem:SI (plus:SI (reg:SI 103) (const_int -12)))
                    (reg:SI 110)))
```

Das hier innerhalb von Speicherausdrücken verwendete Register (reg:SI 103) enthält ein Zeiger auf den Stack der ausgeführten Funktion. Die lokalen Variablen a, b, c sind auf dem Stack, an den Offsets -4, -8 und -12 gespeichert. In diesem Beispiel beschreibt der Ausdruck (mem:SI (plus:SI (reg:SI 103) (const_int -8))) die Variable b. Diese Zusatzinformationen sind jedoch nicht Teil von RTL, sondern gehören zu den im nächsten Abschnitt beschriebenen Erweiterungen. Hierzu zählen auch Werte von temporär verwendeten Fließkommazahlen, wie das nächste Codefragment verdeutlicht.

```
double d, e;

d = 1.25 + e;
```

wird auf i386/Linux folgendermaßen in RTL übersetzt.

```
(insn 18 16 20 (set (reg:DF 61)
                    (mem:DF (plus:SI (reg:SI 54) (const_int -16)))))
(insn 20 18 21 (set (reg:DF 62)
                    (mem:DF (symbol_ref:SI ("LC0")))))
(insn 21 20 23 (set (reg:DF 63) (plus:DF (reg:DF 61) (reg:DF 62))))
(insn 23 21 24 (set (mem:DF (plus:SI (reg:SI 54) (const_int -8)))
                    (reg:DF 63)))
```

Die temporäre Fließkommavariable LC0 hat den Wert 1.25. Das Register (reg:SI 54) enthält für i386/Linux einen Zeiger auf den Stack der ausgeführten Funktion und die lokalen Variablen d und e sind an den Offset -8 und -16 gespeichert.

Spezielle Kommentaranweisungen (note) ermöglichen es uns, den Zusammenhang zwischen Hochsprache und RTL-Anweisungen herzustellen. Die Zeile

```
(note 16 15 18 ("beispiel.c") 5)
```

gibt z.B. an, dass die nachfolgenden RTL-Instruktionen das Resultat der Übersetzung von Zeile 5 aus der Datei beispiel.c sind. Eine spezielle Option des Compilers aktiviert die Generierung dieser Kommentaranweisungen.

Das nachfolgende Codefragment demonstriert die Übersetzung einer komplexen Anweisung in RTL. Wir erhalten eine Vielzahl von „einfachen“ RTL-Anweisungen.

```
int i = (int )&i;
double d = 0.9187;

i = (int )&i;

*((((int *)(i-4))+(int )(((int )d/0.25)+2.9))) = 7;
```

Die letzte, für uns interessante Zeile dieses Programms wird auf i386/Linux übersetzt in

```
(insn 28 26 30 (set (reg:DF 64)
                    (mem:DF (plus:SI (reg:SI 54) (const_int -16)))))
(insn 30 28 31 (set (reg:SI 63) (fix:SI (reg:DF 64))))
(insn 31 30 33 (set (reg:DF 65) (float:DF (reg:SI 63))))
(insn 33 31 34 (set (reg:DF 67)
                    (mem:DF (symbol_ref:SI ("LC0")))))
(insn 34 33 36 (set (reg:DF 66)
                    (div:DF (reg:DF 65) (reg:DF 67))))
(insn 36 34 37 (set (reg:DF 69)
                    (mem:DF (symbol_ref:SI ("LC1")))))
(insn 37 36 39 (set (reg:DF 68)
                    (plus:DF (reg:DF 66) (reg:DF 69))))
(insn 39 37 41 (set (reg:SI 70) (fix:SI (reg:DF 68))))
(insn 41 39 43 (set (reg:SI 71) (reg:SI 70)))
(insn 43 41 45 (set (reg:SI 72) (ashift:SI (reg:SI 71) (const_int 2))))
(insn 45 43 47 (set (reg:SI 73) (plus:SI (reg:SI 72)
                    (mem:SI (plus:SI (reg:SI 54) (const_int -4))))))
(insn 47 45 49 (set (reg:SI 74) (plus:SI (reg:SI 73) (const_int -4))))
(insn 49 47 50 (set (mem:SI (reg:SI 74)) (const_int 7)))
```

Die Variablen LC0 und LC1 haben die Werte 0.25 bzw. 2.9. Eine genaue Überprüfung dieses Programmfragments deckt einen Stackzugriffsfehler (siehe Definition 3.2.35) auf. Das von uns entwickelte Analysetool xGCC entdeckt diesen Fehler (siehe Beispiel 6.3.5).

Verzweigungen, wie in

```
int i, j;

if (i > 0)
  j = 0;
else
  j = 1;
```

werden auf i386/Linux übersetzt in

```
(note 16 15 17 ("beispiel.c") 5)
(insn 17 16 18 (set (reg:SI 626) (compare:SI (mem:SI (plus:SI
                    (reg:SI 54) (const_int -4))) (const_int 0))))
(jump_insn 18 17 19 (set (pc) (if_then_else (le (reg:SI 626)
                         (const_int 0)) (label_ref 24) (pc))))
(note 19 18 21 ("beispiel.c") 6)
(insn 21 19 22 (set (mem:SI (plus:SI (reg:SI 54) (const_int -8)))
                    (const_int 0)))
(jump_insn 22 21 23 (set (pc) (label_ref 28)))
(barrier 23 22 24)
(code_label 24 23 25 2 "" "")
(note 25 24 27 ("beispiel.c") 8)
(insn 27 25 28 (set (mem:SI (plus:SI (reg:SI 54) (const_int -8)))
                    (const_int 1))
(code_label 28 27 29 3 "" "")
```

Wie für alle anderen auf i386/Linux übersetzten Programmfragmente gilt auch hier, dass das Register (reg:SI 54) einen Zeiger auf den Stack der ausgeführten Funktion enthält. Die lokalen Variablen i und j sind an Offset -4 und -8 gespeichert. Den Aufbau einer Schleife demonstrieren wir mit Hilfe dieses Programms.

```
int i, j;

j = 0;
for (i = 0; i < 10; i++)
  j = j + i;
```

Auf Sparc/Solaris wird dieses folgendermaßen in RTL übersetzt. Die for-Schleife besteht aus vier Teilen. Für while- und do-while-Schleifen verschmelzen die Teile Schleifenrumpf und Update und die Initialisierung findet außerhalb der Schleife statt. Je nach Schleifentyp wird die Prüfbedingung vor oder nach dem Schleifenrumpf ausgeführt.

- Initialisierung

```
(note 11 10 13 ("beispiel.c") 6)
(insn 13 11 14 (set (mem:SI (plus:SI (reg:SI 103) (const_int -4)))
                    (const_int 0)))
```

- Prüfbedingung

```
(note 14 13 15 NOTE_INSN_LOOP_BEG)
(code_label 15 14 17 2 "" "")
(insn 17 15 18 (set (reg:SI 108) (mem:SI
                    (plus:SI (reg:SI 103) (const_int -4)))))
(insn 18 17 19 (set (reg:SI 357)
                    (compare:SI (reg:SI 108) (const_int 9))))
(jump_insn 19 18 20 (set (pc) (if_then_else (le (reg:SI 357)
                         (const_int 0)) (label_ref 22) (pc))))
(jump_insn 20 19 21 (set (pc) (label_ref 45)))
(barrier 21 20 22)
(code_label 22 21 23 5 "" "")
(note 23 22 24 NOTE_INSN_LOOP_END_TOP_COND)
```

- Schleifenrumpf

```
(note 24 23 26 ("beispiel.c") 7)
(insn 26 24 28 (set (reg:SI 109) (mem:SI
                    (plus:SI (reg:SI 103) (const_int -8)))))
(insn 28 26 30 (set (reg:SI 110) (mem:SI
                    (plus:SI (reg:SI 103) (const_int -4)))))
(insn 30 28 32 (set (reg:SI 111)
                    (plus:SI (reg:SI 109) (reg:SI 110))))
(insn 32 30 33 (set (mem:SI (plus:SI (reg:SI 103) (const_int -8)))
                    (reg:SI 111)))
(note 33 32 34 ("beispiel.c") 6)
```

- Update

```
(note 34 33 35 NOTE_INSN_LOOP_CONT)
(code_label 35 34 37 4 "" "")
(insn 37 35 39 (set (reg:SI 112)
                    (mem:SI (plus:SI (reg:SI 103) (const_int -4)))))
(insn 39 37 41 (set (reg:SI 113) (plus:SI (reg:SI 112)
                    (const_int 1))))
(insn 41 39 42 (set (mem:SI (plus:SI (reg:SI 103) (const_int -4)))
                    (reg:SI 113)))
(jump_insn 42 41 43 (set (pc) (label_ref 15)))
```

Wir zeigen nun ein Beispiel für den Aufruf einer Funktion mit Parametern und Rückgabewert.

```
int max(int a, int b)
{ return ((a > b) ? a : b); }

int main()
{
  int i, j, k;
  i = max(j, k);
}
```

Auf i386/Linux wird der Funktionsaufruf folgendermaßen in RTL übersetzt.

```
(insn 18 14 19 (set (reg:SI 61) (mem:SI (plus:SI (reg:SI 54)
                    (const_int -8)))))
(insn 19 18 21 (set (mem:SI (reg:SI 56)) (reg:SI 61)))
(insn 21 19 22 (set (reg:SI 62) (mem:SI (plus:SI (reg:SI 54)
                    (const_int -12)))))
(insn 22 21 23 (set (mem:SI (plus:SI (reg:SI 56) (const_int 4)))
                    (reg:SI 62)))
(call_insn 23 22 25 (set (reg:SI 0) (call (mem:QI
                    (symbol_ref:SI ("max"))) (const_int 8))))
(insn 25 23 27 (set (reg:SI 63) (reg:SI 0)))
(insn 27 25 28 (set (mem:SI (plus:SI (reg:SI 54) (const_int -4)))
                    (reg:SI 63)))
```

Das Register (reg:SI 56) enthält einen Zeiger auf ein Speicherstück das hier für die Übergabe von Parametern verwendet wird. Die Werte von j und k werden an Offset 0 und 4 gespeichert. Die Funktion max wird aufgerufen und speichert in Register (reg:SI 0) den Rückgabewert. Die Parameterübergabe und Rückgabe von Werten ist maschinenabhängig. Die durch die Erweiterungen von RTL gewonnen Zusatzinformationen geben hier genauer Auskunft.

2.4 Erweiterungen von RTL

Die in diesem Abschnitt vorgestellte Erweiterung von RTL zu XRTL macht eine sinnvolle Programmanalyse erst möglich. Wir benötigen diese Erweiterung um z.B. Aussagen über

den korrekten bzw. inkorrekten Zugriff auf lokale Variablen (Stack) zu machen. Wir verdeutlichen diesen Sachverhalt anhand zweier Codefragmente, deren Übersetzung in RTL identisch ist. Das korrekte Codefragment sieht folgendermaßen aus:

```
int array[2];

array[0] = 0;
array[1] = 1;
```

Das nachfolgende, inkorrekte Codefragment

```
int array[1];

array[-1] = 0;
array[0] = 1;
```

unterscheidet sich in der RTL-Übersetzung nicht vom korrekten Programm, weil die Anweisungen int array[2] bzw. int array[1] keine RTL-Anweisungen erzeugen.

```
(insn 18 16 19 (set (mem:SI (plus:SI (reg:SI 54) (const_int -8)))
                    (const_int 0)))
(insn 21 19 22 (set (mem:SI (plus:SI (reg:SI 54) (const_int -4)))
                    (const_int 1)))
```

Die im folgenden beschriebenen Erweiterungen ermöglichen uns die Unterscheidung zwischen korrektem und inkorrektem Programmfragment. Wir führen zuerst XRTL-Anweisungen, die es uns erlauben den Stack einer Funktion, also den Teil in dem lokale Variablen gespeichert sind, zu rekonstruieren. Die XRTL-Anweisung

(xinsn (stackvar) (line l) (size:m s) (align a))

bezeichnet eine Stackvariable der Größe s des Typs m, die in Zeile l im Programmtext generiert wurde. Das Alignment gibt an, durch welche Zahl das Offset, an dem die Variable im Stack liegt teilbar sein muss. Das Alignment einer Variablen ist abhängig vom Variablentyp und der Maschine, auf der das Programm ausgeführt wird. Die nachfolgende Anweisung

(xinsn (stack) (line l) (size s))

gibt uns die Größe s des Stacks an. Mit Hilfe dieser Daten können wir nun ein Speicherstück anlegen, welches das Stackframe einer Funktion repräsentiert. Wir müssen hierbei beachten, dass der Stack nach unten wächst, d.h. wir haben negative Offsets und das es, aufgrund des Alignments, zu Speicherlöchern kommen kann, auf die nicht zugegriffen werden darf.

Das nachfolgende Programmfragment

```
char c;
double d;
int i;
```

erzeugt folgende XRTL Stackframe-Anweisungen

```
(xinsn (stackvar) (line 3) (size:QI 1) (align 8))
(xinsn (stackvar) (line 4) (size:DF 8) (align 64))
(xinsn (stackvar) (line 5) (size:SI 4) (align 32))
(xinsn (stack) (line 8) (size 24))}
```

Der rekonstruierte Stack sieht nun folgendermaßen aus:

Offset	Variable
-1	c
-2	
-3	
-4	
-5	$unused$
-6	
-7	
-8	

Offset	Variable
-9	
-10	
-11	
-12	d
-13	
-14	
-15	
-16	

Offset	Variable
-17	
-18	i
-19	
-20	
-21	
-22	$unused$
-23	
-24	

Der Zugriff auf unbenutzte Teile des Speicherstücks (siehe Definition 3.2.34) ist verboten. In XRTL werden spezielle Register für den Zugriff auf den Stack und Funktionsparameter reserviert. Die Anweisung

(xinsn (stack) (incp i) (stackp s) (hstackp h) (outp o))

gibt an, dass wir die Register mit Nummer s und h für den Zugriff auf den Stack einsetzten und das der Zeiger bei Funktionsaufruf implizit gesetzt wurde. Das Register mit Nummer i zeigt auf eine Liste mit eingehenden Funktionsparametern, während das Register mit Nummer o auf eine Liste mit ausgehenden Parametern zeigt. Die nachfolgende Anweisung

(xinsn (function_call) (arguments a))

gibt für jeden getätigten Funktionsaufruf innerhalb einer Funktion, die Anzahl a der Argumente an. Mit Hilfe der Anweisung

(xinsn (function_call) (arg n) (x))

gibt für jedes übergebene Argument n einen Register- oder Speicherausdruck x an, der den Wert des übergebenen Parameters beschreibt.

Das nachfolgende Beispiel

```
(xinsn (function_call) (name "testfunction") (begin))
(xinsn (function_call) (arguments 3))
(xinsn (function_call) (arg 0) (reg:SI 8))
(xinsn (function_call) (arg 1) (reg:DF 9))
(xinsn (function_call) (arg 2) (mem:SI (reg:SI 56)))
(xinsn (function_call) (name "testfunction") (end))
```

gibt die Informationen über den Aufrufs der Funktion $testfunction$ mit drei Parametern an. Die Parameter 0 und 1 werden in Registern übergeben und der Parameter 2 an einer speziellen Speicherstelle. Die Anweisung

```
(xinsn (function) (line l) (name fn))
```

markiert das Ende der XRTL Anweisungen für eine Funktion mit dem Namen fn die in Zeile l beginnt.

Wir benötigen für die XRTL Analyse noch die Werte und Struktur von globalen oder temporär eingeführten Variablen und Konstanten. Diese werden in XRTL für Variablen

```
(xinsn (variable) (line l) (name vn) (begin))

                        ⋮

(xinsn (variable) (line l) (name vn) (end))
```

und Konstanten

```
(xinsn (constant) (line l) (name cn) (begin))

                        ⋮

(xinsn (constant) (line l) (name cn) (end))
```

so ausgedrückt. Die Variable bzw. Konstante hat den Namen vn bzw. cn und befindet sich in Zeile l unseres Ausgangsprogramms. Innerhalb dieser Begrenzungen geben wir die Struktur und die dazugehörigen Initialisierungen an. Die Anweisungen

```
(xinsn (real) (line l) (set (const:m rn) (v)))
```

und

```
(xinsn (integer) (line l) (set (const:m in) (v)))
```

geben an, dass die Variable eine Fließkommazahl oder ganze Zahl in Modus m mit Namen rn bzw. in und initialem Wert v enthält.

Bei Variablen mit zusammengesetztem Datentyp kann es vorkommen, dass mehrere dieser Anweisungen auftreten. Diese werden dann durch

```
(xinsn (record/union) (line l) (begin))
⋮
(xinsn (record/union) (line l) (end))
```

umrahmt. Eine spezielle Form der zusammengesetzten Datentypen sind die Bitfelder. Diese haben die Form

```
(xinsn (bitfield) (line l) (name bn) (begin))
⋮
(xinsn (bitfield) (line l) (name bn) (end))
```

und mit mehreren Anweisungen der Form

```
(xinsn (bitfield) (line l) (first b) (last e) (size s))
(xinsn (bitfield) (line l) (set (const:m bn) (v)))
```

wird die Struktur und der initiale Wert des Bitfeldes angegeben. Die Anweisung

```
(xinsn (zero) (line l) (name zn) (size s) (allocated a))
```

beschreibt einen Bereich der Größe a der mit 0 initialisiert wird. Die Größe und die Anzahl der allokierten Bytes können sich aufgrund des Alignments oder effizienteren Zugriffs unterscheiden. Das nachfolgende Beispiel

```
(xinsn (variable) (line 6) (name "b_a") (begin))
(xinsn (record/union) (line 6) (begin))
(xinsn (bitfield) (line 6) (name "(null)") (begin))
(xinsn (bitfield) (line 6) (first 0) (last 1) (size 2))
(xinsn (bitfield) (line 6) (first 2) (last 5) (size 4))
(xinsn (bitfield) (line 6) (first 6) (last 8) (size 3))
(xinsn (bitfield) (line 6) (set (const:QI "(null)") (201)))
(xinsn (bitfield) (line 6) (first 9) (last 13) (size 5))
(xinsn (bitfield) (line 6) (set (const:QI "(null)") (56)))
(xinsn (bitfield) (line 6) (name "(null)") (end))
(xinsn (zero) (line 6) (name "(null)") (size 2) (allocated 2))
(xinsn (record/union) (line 6) (end))
(xinsn (variable) (line 6) (name "b_a") (end))

(xinsn (constant) (line 22) (name "LC0") (begin))
(xinsn (real) (line 22) (set (const:DF "LC0") (-2.3456)))
(xinsn (constant) (line 22) (name "LC0") (end))
```

beschreiben ein Bitfeld b_a, dass vier Einträge unterschiedlicher Bitgröße hat und aus Effizienzgründen mit zwei Nullbytes zu einer vier Byte großen Struktur erweitert wird und einer Fließkommakonstanten LC0 in Modus DF mit Wert -2.3456. Der Namensstring (null) gibt an, dass hier eine Namensgebung nicht möglich oder nicht sinnvoll ist.

In diesem Kapitel haben wir den Aufbau der GNU Compiler Collection erklärt und in die Register Transfer Language (RTL) eingeführt. Wir haben uns mit unterschiedlichen Modi und Datentypen beschäftigt und anschließend den Aufbau von RTL Instruktionen erklärt. Wir haben RTL Teilausdrücke, wie z.B. Register und Speicherausdrücke, konstante Ausdrücke, arithmetische Ausdrücke und Vergleichsausdrücke vorgestellt. Mit Hilfe kleiner Programmfragmente haben wir die Übersetzung zu RTL demonstriert und so den Aufbau von Verzweigungen und Schleifen erklärt. Wir haben die, für unser Analysetool xGCC notwendige Erweiterung XRTL motiviert und vorgestellt.

Kapitel 3

Abstraktion

In diesem Kapitel geben wir eine Einführung in die Theorie der Abstrakten Interpretation und definieren die Begriffe Verband und Galoisverknüpfung. Wir definieren den abstrakten Domain und stellen die Idee der Approximation vor. Der verwendete abstrakte Domain ist eine endlichen Menge von Registern und Speicherstücken. Die Basis für den konkreten und abstrakten Domain sind die Grunddatentypen, endliche Teilmengen der ganzen und reellen Zahlen. Unser verwendeter Approximationsansatz basiert auf der klassischen Intervall-Approximation. Wir führen den Begriff des gültigen Intervalls ein und approximieren Register und Speicherstücke mit Hilfe von Listen von gültigen Intervallen (Vereinigung von Intervallen) unterschiedlichen Grunddatentyps. Die Länge der Liste bleibt beschränkt, ist aber beliebig wählbar. Dies eröffnet uns die Möglichkeit die Präzision der Approximation zu variieren. Wir führen die von XRTL benötigten Operationen auf dem abstrakten Domain ein. Die Verwendung endlicher Teilmengen von ganzen und reellen Zahlen vereinfacht einige der Operationen, verursacht jedoch auch Probleme, wie z.B. ein möglicher Werteüberlauf bei einer Addition.

3.1 Grundlagen

In [11, 13] wurde die Theorie der Abstrakten Interpretation und die Idee der Approximation eingeführt. Für weitergehende Definitionen, sowie Sätze und Beweise verweisen wir auf diese Artikel und [5, 14, 29, 36]. Diese Theorie der Abstrakten Interpretation enthält zwei Ansätze, die Galoisverknüpfung und den so genannten Widening/Narrowing Ansatz.

Definition 3.1.1 *Wir bezeichnen eine Menge S mit einer reflexiven, transitiven und antisymetrischen Relation $\sqsubseteq$ eine partiell geordnete Menge (partial ordered set).*

Partiell geordnete bedeutet hier, dass wir möglicherweise zwei Elemente $a, b \in S$ finden, die nicht vergleichbar sind, d.h. es gilt weder $a \sqsubseteq b$ noch $b \sqsubseteq a$.

Definition 3.1.2 *Wir bezeichnen $c \in S$ als obere Schranke (upper bound) einer Teilmenge $X \subseteq S$ wenn für alle $c' \in X, c' \sqsubseteq c$ gilt.*

Wir bezeichnen $c \in S$ als untere Schranke (lower bound) von $X \subseteq S$, wenn für alle $c' \in X, c \sqsubseteq c'$ gilt.

Eine obere Schranke c von X ist die kleinste obere Schranke, falls für jede obere Schranke c' gilt $c \sqsubseteq c'$ und eine untere Schranke c von X ist die größte untere Schranke, falls jede untere Schranke $c' \sqsubseteq c$ ist.

Definition 3.1.3 *Wir bezeichnen eine partiell geordnete Menge $L(\sqsubseteq)$ als Verband (lattice), wenn für alle a und b, $a, b \in L$ eine kleinste obere Schranke $a \sqcap b$ und größte untere Schranke $a \sqcup b$ existiert. Für einen Verband (lattice) L verwenden wir die Notation $L(\sqsubseteq, \sqcup, \sqcap)$.*

Definition 3.1.4 *Wir bezeichnen einen Verband als vollständigen Verband (complete lattice), wenn für jede Teilmenge X von L eine größte untere Schranke $\sqcap X$ und eine kleinste obere Schranke $\sqcup X$ existiert. Wir bezeichnen das kleinste Element $\sqcap \emptyset$ mit $\bot$ und das größte Element $\sqcup L$ mit $\top$. Wir verwenden für einen vollständigen Verband (complete lattice) die Notation $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$.*

Jeder Verband auf Basis einer endlichen Menge ist eine vollständiger Verband, da alle endlichen Mengen immer kleinste obere und größte untere Schranken haben. Dies gilt jedoch nicht notwendigerweise für unendliche Mengen.

Definition 3.1.5 *Seien $L_1(\sqsubseteq_1)$ und $L_2(\sqsubseteq_2)$ partiell geordnete Mengen. (α, γ) ist eine Galoisverknüpfung zwischen L_1 und L_2, mit $\alpha \in L_1 \to L_2$ und $\gamma \in L_2 \to L_1$ wenn*

$$\alpha(x) \sqsubseteq_2 y \Leftrightarrow x \sqsubseteq_1 \gamma(y)$$

$\forall x \in L_1$ und $\forall \in L_2$ gilt. Wir verwenden hierfür die folgende Notation:

$$(L_1, \sqsubseteq_1) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} (L_2, \sqsubseteq_2)$$

Wir bezeichnen α als Approximation und γ als Konkretisierung, $(L_1, \sqsubseteq_1)$ als konkreten Verband (concrete lattice), $(L_2, \sqsubseteq_2)$ als abstrakten Verband (abstract lattice).

Für ein $x \in L_1$ ist $\alpha(x)$ die möglichst genaue Approximation in L_2 und für $y \in L_2$ repräsentiert $\gamma(y)$ am wenigsten genaue Element in L_1 welches zu y approximiert werden kann.

Proposition 3.1.6 *(α, γ) ist nur dann eine Galoisverknüpfung, wenn α und γ monoton sind und $(\alpha \circ \gamma)(y) \sqsubseteq_2 y$ und $x \sqsubseteq_1 (\gamma \circ \alpha)(x)$ gilt.*

3.2 Konkreter und abstrakter Domain

Die Basis für den konkreten und abstrakten Domain, eine endlich Menge von Registern und Speicherstücken unterschiedlichen Typs, sind endliche Teilmengen der ganzen und reellen Zahlen. Ein Register entspricht einer Variablen die Werte speichert, während ein Speicherstück aus mehreren, meist unterschiedlichen Datentypen zusammengesetzt sein kann. Obwohl unser konkreter Domain und die Grunddatentypen endlich sind, verwenden wir, aufgrund der hohen Elementanzahl, das Prinzip der Approximation. Wir beschäftigen uns mit der Approximation der Grunddatentypen.

3.2.1 Ganze Zahlen

Wir definieren eine endliche Teilmenge der ganzen Zahlen $\mathbb{Z}$, gehen allerdings erst in Kapitel 4 auf die verwendeten Instanzen dieser Teilmenge ein.

Definition 3.2.1 *Wir bezeichnen mit $\mathbb{Z}_m$ und $m \in \mathbb{N}$ den Restklassenring modulo m, wobei*

$$\mathbb{Z}_m \stackrel{def}{=} \{\bar{a} \mid a \in \mathbb{Z}\} \tag{3.1}$$

und

$$\bar{a} \stackrel{def}{=} \{b \in \mathbb{Z} \mid b \equiv a \bmod m\} \tag{3.2}$$

Es existieren mehrere Möglichkeiten die Elemente von $\mathbb{Z}_m$ darzustellen, z.B. durch die kleinsten, nicht negativen oder die betragsmäßig kleinsten Vertreter der Restklasse.

Definition 3.2.2 *Wir bezeichnen, für $a^+, a^- \in \mathbb{Z}$, mit a^+ den kleinsten nicht negativen Vertreter und mit a^- mit den betragsmäßig kleinsten Vertreter der Restklasse $\bar{a} \in \mathbb{Z}_m$. Existieren bei der Wahl der betragsmäßig kleinsten Vertreter zwei Vertreter $a', a'' \in \bar{a}$ mit minimalem Betrag und $a' < a''$ so definieren wir a^- als a'.*

Beispiel 3.2.3 Für $\mathbb{Z}_8$ ist $\{0, 1, \ldots, 7\}$ die Menge der kleinsten nicht negativen Vertreter der Restklassen. $\{-4, \ldots, -1, 0, \ldots, 3\}$ ist die Menge der betragsmäßig kleinsten Vertreter der Restklassen. Nach Definition 3.2.2 wählen wir hierbei -4 als betragsmäßig kleinsten Vertreter und nicht 4. Es gilt $\mathbb{Z}_8 = \{\bar{0}, \bar{1}, \ldots, \bar{7}\}$ und $\mathbb{Z}_8 = \{-\bar{4}, \ldots, -\bar{1}, \bar{0}, \ldots, \bar{3}\}$.

Definition 3.2.4 *Mit dem Intervall $\left[\bar{a}, \bar{b}\right]$, wobei $\bar{a}, \bar{b} \in \mathbb{Z}_m$ und $a^+ \leq b^+$ bezeichnen wir die Menge aller $\bar{x} \in \mathbb{Z}_m$, für die $a^+ \leq x^+$ und $x^+ \leq b^+$ gilt.*

Mit $I(\mathbb{Z}_m)$ bezeichnen wir die Menge aller Intervalle über $\mathbb{Z}_m$ mit

$$I(\mathbb{Z}_m) \stackrel{def}{=} \{\bot\} \cup \left\{ \left[\bar{a}, \bar{b}\right] \mid \left[\bar{a}, \bar{b}\right] \textit{ ist Intervall, } \bar{a}, \bar{b} \in \mathbb{Z}_m \right\}$$

Mit $\bot$ bezeichnen wir hier das leere Intervall.

Die Operationen Addition, Subtraktion und Multiplikation auf $\mathbb{Z}_m$ sind unabhängig von der Wahl der Vertreter der Restklasse. Dies folgt aus der Ringeigenschaft von $\mathbb{Z}_m$. Es spielt also keine Rolle ob wir für ein $\bar{a} \in \mathbb{Z}_m$ den Vertreter a^+ oder a^- der Restklasse verwenden. Wir führen zwei Pseudodivisionen ein, die jedoch abhängig von der Wahl der Vertreter der Restklasse sind. Diese Abhängigkeit erfordert es die Definition von Intervallen anzupassen, da diese in einigen Fällen unzureichend ist, da Intervalle $\left[\, \bar{a}, \bar{b} \,\right]$ existieren, mit $\bar{a}, \bar{b} \in \mathbb{Z}_m$ und $a^+ < b^+$ existieren, für die $a^- > b^-$ gilt.

Beispiel 3.2.5 Gegeben sei das Intervall $\left[\, \bar{a}, \bar{b} \,\right]$, für $\bar{a}, \bar{b} \in \mathbb{Z}_8$, mit $\bar{a} = \bar{2}$ und $\bar{b} = \bar{6}$. Für dieses Intervall gilt $a^+ < b^+ \;\Leftrightarrow\; 2 < 6$, aber auch $a^- > b^- \;\Leftrightarrow\; 2 > -2$.

Definition 3.2.6 *Wir bezeichnen ein Intervall $\left[\, \bar{a}, \bar{b} \,\right]$ und $\bar{a}, \bar{b} \in \mathbb{Z}_m$ mit $a^+ \leq b^+$ als gültig, wenn $a^+ \geq \frac{m}{2}$ oder $b^+ < \frac{m}{2}$ gilt.*
Mit $I^*(\mathbb{Z}_m)$ bezeichnen wir die Menge aller gültigen Intervalle über $\mathbb{Z}_m$ mit

$$I^*(\mathbb{Z}_m) \stackrel{def}{=} \{\bot\} \cup \left\{ \left[\, \bar{a}, \bar{b} \,\right] \mid \left[\, \bar{a}, \bar{b} \,\right] \textit{ ist gültiges Intervall } \right\}$$

Bemerkung 3.2.7 Ein Intervall $\left[\, \bar{a}, \bar{b} \,\right]$ ist dann gültig, wenn $a^+ \leq b^+$ und $a^- \leq b^-$ gilt.

Gültige Intervalle führen zu Einschränkungen und machen das Einführen von Listen von gültigen Intervallen erforderlich da z.B. kein gültiges Intervall existiert, in dem alle Elemente aus $\mathbb{Z}_m$ enthalten sind. Eine Beziehung zwischen Intervallen und gültigen Intervallen ermöglicht die Umwandlung.

Lemma 3.2.8 *Jedes Intervall $\left[\, \bar{a}, \bar{b} \,\right]$ ist gültig, oder kann in zwei gültige Intervalle aufgeteilt werden, die alle Elemente aus $\left[\, \bar{a}, \bar{b} \,\right]$ enthalten.*

Beweis: Durch Konstruktion:

$$\left[\, \bar{a}, \bar{b} \,\right] \Rightarrow \begin{cases} \left[\, \overline{a_1}, \overline{b_1} \,\right] \stackrel{\text{def}}{=} \left[\, \bar{a}, \overline{\frac{m}{2} - 1} \,\right] \\ \left[\, \overline{a_2}, \overline{b_2} \,\right] \stackrel{\text{def}}{=} \left[\, \overline{\frac{m}{2}}, \bar{b} \,\right] \end{cases}$$

Die Intervalle $\left[\, \overline{a_1}, \overline{b_1} \,\right]$ und $\left[\, \overline{a_2}, \overline{b_2} \,\right]$ enthalten alle Elemente aus $\left[\, \bar{a}, \bar{b} \,\right]$ und sind offensichtlich gültig. ■

Wir verwenden für die Approximation also keine einfachen Intervalle, sondern eine Liste von gültigen Intervallen. Die Länge dieser Liste ist durch einen Parameter beschränkt.

Definition 3.2.9 *Wir bezeichnen mit $\left(\left[\, \overline{a_i}, \overline{b_i} \,\right]\right)_{i=1}^{n}$ die Liste von gültigen Intervallen $\left[\, \overline{a_i}, \overline{b_i} \,\right]$ mit maximaler Länge n. Für ein $\overline{x} \in \mathbb{Z}_m$ gilt $\overline{x} \in \left(\left[\, \overline{a_i}, \overline{b_i} \,\right]\right)_{i=1}^{n}$, falls $\exists i$ mit $\overline{x} \in \left[\, \overline{a_i}, \overline{b_i} \,\right]$.*
Wir bezeichnen mit $I_n^*(\mathbb{Z}_m)$ die Menge alle Listen von gültigen Intervallen über $\mathbb{Z}_m$ mit maximaler Länge n.

Bemerkung 3.2.10 Wir verwenden hier den Begriff der Listen von gültigen Intervallen, da in der Praxis die gültigen Intervalle sortiert sind, kein Intervall mehrfach auftritt und zwei beliebige gültige Intervalle aus der Liste kein Element gemeinsam haben.

Die nachfolgend definierten Funktionen α und γ bilden eine Galoisverknüpfung zwischen $P(\mathbb{Z}_m)$ und $I_n^*(\mathbb{Z}_m)$ gemäß Definition 3.1.5. Wir verwenden hierfür den Algorithmus `split`, der ein $X \subseteq \mathbb{Z}_m$ in eine Liste mit maximal n gültigen Intervallen aufspaltet.

$$\begin{aligned} \alpha &: P(\mathbb{Z}_m) \longrightarrow I_n^*(\mathbb{Z}_m) \\ \gamma &: I_n^*(\mathbb{Z}_m) \longrightarrow P(\mathbb{Z}_m) \end{aligned}$$

wobei

$$\begin{aligned} \alpha(X) &\stackrel{\text{def}}{=} \texttt{split}(X) \\ \gamma\left(\left(\left[\,\overline{a_i}, \overline{b_i}\,\right]\right)_{i=1}^{n}\right) &\stackrel{\text{def}}{=} \left\{\, \bar{x} \in \mathbb{Z}_m \mid \exists i \text{ mit } \bar{x} \in \left[\,\overline{a_i}, \overline{b_i}\,\right] \,\right\} \end{aligned}$$

Der Algorithmus `split` teilt die Menge $X \in P(\mathbb{Z}_m)$ in $\frac{n}{2}$ Mengen auf. Diese werden in Intervalle und dann in gültige Intervalle umgewandelt. Zwei Elemente $\bar{x}, \bar{y} \in X$ sind benachbart in X, wenn $\nexists\, \bar{z} \in X$ mit $x^+ < z^+ < y^+$.

Algorithmus 3.2.11

Algorithmus `split`

EINGABE: $X \in P(\mathbb{Z}_m)$
AUSGABE: $\left(\left[\,\overline{a_i}, \overline{b_i}\,\right]\right)_{i=1}^{n}$ – Liste gültiger Intervalle

(1) $L := \{X\}$
(2) **while** ($|L| \leq \frac{n}{2}$) **do**
(3) finde $l \in L$ mit $\bar{x}, \bar{y} \in l$, $|x^+ - y^+|$ maximal, $\bar{x}, \bar{y}$ benachbart
(4) $l_x := \{\bar{z} \in l \mid z^+ \leq x^+\}$
(5) $l_y := \{\bar{z} \in l \mid z^+ \geq y^+\}$
(6) $L := L \setminus \{l\}$
(7) $L := L \cup \{l_x, l_y\}$
(8) **od**
(9) **foreach** ($l \in L$) **do**
(10) $x_{min} := \min\{x^+ \mid \bar{x} \in l\}$
(11) $x_{max} := \max\{x^+ \mid \bar{x} \in l\}$
(12) $\left[\,\bar{a}, \bar{b}\,\right] := \left[\,\overline{x_{min}}, \overline{x_{max}}\,\right]$
(13) Teile $\left[\,\bar{a}, \bar{b}\,\right]$ in gültige Intervalle auf
(14) Füge gültige Intervalle zur Liste hinzu
(15) **od**

Gibt es in Zeile `(3)` von Algorithmus `split` mehrere Auswahlmöglichkeiten, dann wählen wir das kleinste x, das die Bedingungen erfüllt.

Die abstrakte Interpretation erlaubt es uns mit Hilfe der konkreten Semantik und der Galoisverknüpfung die abstrakte Semantik zu definieren. Es ergibt sich die folgende Beziehung:

$$\begin{array}{ccc} P(\mathbb{Z}_m) & \xrightarrow{post} & P(\mathbb{Z}_m) \\ \alpha \downarrow & & \uparrow \underset{\gamma}{\subseteq} \\ I_n^*(\mathbb{Z}_m) & \xrightarrow{post^\#} & I_n^*(\mathbb{Z}_m) \end{array}$$

Eine Definition der Addition $a^\# +^\# b^\# \stackrel{\text{def}}{=} \alpha(\gamma(a^\#) + \gamma(b^\#))$ ist in der Praxis wenig sinnvoll. Probleme, wie der Unterlauf und Überlauf von Operationen in $\mathbb{Z}_m$ sind zu beachten, was in $\mathbb{Z}_m$ das Zusammenfassen von Werten in Intervalle erschwert.

Für die Implementierung von $post^\#$ auf $I_n^*(\mathbb{Z}_m)$ setzen wir Operationen wie Addition, Subtraktion, Multiplikation und zwei Arten von Pseudodivisionen und Modulo auf $I_n^*(\mathbb{Z}_m)$ ein. Wir verwenden weiterhin einen so genannten Verschmelzungsalgorithmus `merge`, der eine zu hohe Anzahl von Einzelintervallen reduziert. Wir gehen später genauer auf diesen Algorithmus ein.

Für $op \in \{add, sub, mul, div, udiv, mod, umod\}$:

$$op^\# \ : \ I_n^*(\mathbb{Z}_m) \times I_n^*(\mathbb{Z}_m) \longrightarrow I_n^*(\mathbb{Z}_m)$$

$$\begin{array}{c} op^\# \left(\left(\left[\, \overline{a_i}, \overline{b_i} \, \right] \right)_{i=1}^n, \left(\left[\, \overline{c_j}, \overline{d_j} \, \right] \right)_{j=1}^n \right) \stackrel{\text{def}}{=} \\ \texttt{merge} \left(\left(op^\# \left(\left[\, \overline{a_i}, \overline{b_i} \, \right], \left[\, \overline{c_j}, \overline{d_j} \, \right] \right) \right)_{i=1,j=1}^n \right) \end{array}$$

Um die Funktionsweise des Verschmelzungsalgorithmus `merge` zu verstehen, müssen wir zuerst den Begriff der Distanz zweier Intervalle einführen. Diesen benötigen wir innerhalb des Algorithmus.

Definition 3.2.12 *Für zwei Intervalle* $\left[\, \bar{a}, \bar{b} \, \right]$ *und* $\left[\, \bar{c}, \bar{d} \, \right]$ *mit* $\bar{a}, \bar{b}, \bar{c}, \bar{d} \in \mathbb{Z}_m$ *und* $a^+ \leq c^+$ *definieren wir die Distanz von* $\left[\, \bar{a}, \bar{b} \, \right]$ *und* $\left[\, \bar{c}, \bar{d} \, \right]$ *folgendermaßen:*

$$\begin{array}{l} distanz \ : \ I(\mathbb{Z}_m) \times I(\mathbb{Z}_m) \longrightarrow \mathbb{Z} \\ distanz \ \left(\left[\, \bar{a}, \bar{b} \, \right], \left[\, \bar{c}, \bar{d} \, \right] \right) \stackrel{def}{=} \begin{cases} 0, \textit{ falls } c^+ \leq b^+ \\ c^+ - b^+ \textit{ sonst} \end{cases} \end{array}$$

Der Verschmelzungsalgorithmus `merge` arbeitet folgendermaßen.

Algorithmus 3.2.13

Algorithmus `merge`

EINGABE: L : Liste von Intervallen
AUSGABE: L : Liste von gültigen Intervallen, mit maximaler Länge n

(1) Sortiere L aufsteigend, nach unterer Schranke
(2) **foreach** ($k \in L$) **do**
(3) **foreach** ($l \in L$) **do**
(4) **if** ($k \cap l \neq \emptyset$) **then**
(5) $j := k \cup l$
(6) $L := L \setminus \{k, l\}$
(7) $L := L \cup \{j\}$
(8) **fi**
(9) **od**
(10) **od**
(11) **while** ($|L| > \frac{n}{2}$) **do**
(12) finde i mit $distanz\left(\left[\overline{a_i}, \overline{b_i}\right], \left[\overline{a_{i+1}}, \overline{b_{i+1}}\right]\right)$ minimal
(13) $L := L \setminus \left\{\left[\overline{a_i}, \overline{b_i}\right], \left[\overline{a_{i+1}}, \overline{b_{i+1}}\right]\right\}$
(14) $L := L \cup \left\{\left[\overline{a_i}, \overline{b_{i+1}}\right]\right\}$
(15) **od**
(16) **foreach** ($l \in L$) **do**
(17) Ersetze l durch gültige Intervalle
(18) **od**

Gemäß Konstruktion erzeugt der Verschmelzungsalgorithmus eine Liste von gültigen Intervallen mit maximaler Länge n. Es gehen offensichtlich Informationen verloren. Die Schritte `(2)` – `(10)` ersetzen zwei sich überlappende Intervalle durch ein Intervall, das die beiden umfasst. In `(11)` – `(15)` werden zwei Intervalle mit minimaler Distanz durch ein Intervall ersetzt, dass beide enthält. In diesem Abschnitt verlieren wir an Genauigkeit, da Werte aus $\mathbb{Z}_m$ hinzukommen, die in keinem der Ausgangsintervalle waren. Durch Auswahl von Intervallen mit minimaler Distanz halten wir den Verlust an Genauigkeit gering. Zum Abschluss werden in Schritt `(16)` – `(18)` die Intervalle durch gültige Intervalle ersetzt.

Wir führen nun die Operationen auf gültigen Intervallen ein. Die Ergebnisintervalle sind nicht notwendigerweise gültig. Der Unterlauf bzw. Überlauf in $\mathbb{Z}_m$ führt dazu, dass wir mehr als ein Ergebnisintervall erhalten.

Definition 3.2.14 *Mit einem arithmetischen Fehler bezeichnen wir den Aufruf einer arithmetischen Operation mit Parametern für die das Ergebnis nicht definiert ist.*

Bemerkung 3.2.15 Ein Überlauf bzw. Unterlauf einer Operation führt nicht zu einem arithmetischen Fehler. Die Division durch ein Intervall das die Null enthält ist nicht definiert.

Bemerkung 3.2.16 Die Ausgabe des Algorithmus `merge` ist undefiniert, wenn eines der Eingabeintervalle undefiniert ist.

Wir müssen bei der Defintion der Addition und Subtraktion einen möglicherweise auftretenden Überlauf bzw. Unterlauf betrachten. Wir führen deshalb eine Fallunterscheidung durch. Die Addition und Subtraktion sieht folgendermaßen aus:

$$add^{\#} \ : \ I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \times I(\mathbb{Z}_m)$$
$$sub^{\#} \ : \ I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \times I(\mathbb{Z}_m)$$

wobei

$$add^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \begin{cases} \left(\left[\,\overline{a^+ + c^+},\overline{b^+ + d^+}\,\right],\bot\right) \\ \qquad \text{falls } (a^+ + c^+ \bmod m) \leq (b^+ + d^+ \bmod m) \\ \left(\left[\,\overline{a^+ + c^+},\overline{m-1}\,\right],\left[\,\bar{0},\overline{b^+ + d^+}\,\right]\right) \text{ sonst} \end{cases}$$

$$sub^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \begin{cases} \left(\left[\,\overline{a^+ - d^+},\overline{b^+ - c^+}\,\right],\bot\right) \\ \qquad \text{falls } (a^+ - d^+ \bmod m) \leq (b^+ - c^+ \bmod m) \\ \left(\left[\,\overline{a^+ - d^+},\overline{m-1}\,\right],\left[\,\bar{0},\overline{b^+ - c^+}\,\right]\right) \text{ sonst} \end{cases}$$

Wir sprechen von einem doppelten Überlauf der Addition, wenn $a^+ + c^+ \geq m$. Aus der Definition von gültigen Intervallen folgt dann $b^+ + d^+ \geq m$. Bei der Subtraktion tritt ein doppelter Unterlauf auf, wenn $b^+ - c^+ < 0$ ist. Hier folgt aus den Eigenschaften gültiger Intervalle $a^+ - d^+ < 0$. Kein oder ein doppelter Überlauf tritt bei der Addition auf, wenn $(a^+ + c^+ \bmod m) \leq (b^+ + d^+ \bmod m)$ gilt. Wenn $(a^+ - d^+ \bmod m) \leq (b^+ - c^+ \bmod m)$ für die Subtraktion gilt, dann tritt hier kein oder ein doppelter Unterlauf auf. Die Ergebnisintervalle der Addition und Subtraktion sind nicht notwendigerweise gültig. Aus diesem Grund ist ein Aufruf des Algorithmus `merge` notwendig.

Beispiel 3.2.17 Wir veranschaulichen die in der Definition der Addition und Subtraktion auftretenden Fälle mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{Z}_{16}$.

$$\begin{aligned}
add^{\#}\left(\left[\,\bar{0},\bar{2}\,\right],\left[\,\bar{1},\bar{3}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\bar{1},\bar{5}\,\right],\bot\right) && (0+1 \bmod 16) \leq (2+3 \bmod 16) \\
add^{\#}\left(\left[\,\bar{9},\overline{12}\,\right],\left[\,\bar{2},\bar{6}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\overline{11},\overline{15}\,\right],\left[\,\bar{0},\bar{2}\,\right]\right) && (9+2 \bmod 16) > (12+6 \bmod 16) \\
sub^{\#}\left(\left[\,\overline{13},\overline{15}\,\right],\left[\,\bar{9},\overline{11}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\bar{2},\bar{6}\,\right],\bot\right) && (13-11 \bmod 16) \leq (15-9 \bmod 16) \\
sub^{\#}\left(\left[\,\bar{9},\overline{12}\,\right],\left[\,\overline{10},\overline{13}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\overline{12},\overline{15}\,\right],\left[\,\bar{0},\bar{4}\,\right]\right) && (9-13 \bmod 16) > (12-10 \bmod 16)
\end{aligned}$$

Beispiele für doppelten Überlauf bzw. Unterlauf

$$\begin{aligned}
add^{\#}\left(\left[\,\overline{11},\overline{14}\,\right],\left[\,\bar{6},\bar{7}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\bar{1},\bar{5}\,\right],\bot\right) && (11+6 \bmod 16) \leq (14+7 \bmod 16) \\
sub^{\#}\left(\left[\,\bar{1},\bar{2}\,\right],\left[\,\bar{9},\overline{11}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\bar{6},\bar{9}\,\right],\bot\right) && (1-11 \bmod 16) \leq (2-9 \bmod 16)
\end{aligned}$$

Das Ergebnisintervall $\left[\,\bar{6},\bar{9}\,\right]$ ist hier nicht gültig.

Die Multiplikation auf $I^*(\mathbb{Z}_m)$ ist schwieriger, da wir die möglichen Überläufe genauer betrachen müssen. Wir verwenden hier, im Gegensatz zur Addition und Subtraktion, den Grad des Überlaufs um in der Definition der Multiplikation zwischen zwei Arten von Überlauf zu unterscheiden. Die Definition der Multiplikation sieht folgendermaßen aus.

$$mul^{\#} \; : \; I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \times I(\mathbb{Z}_m)$$

wobei

$$mul^{\#} \; \left(\left[\,\bar{a}, \bar{b}\,\right], \left[\,\bar{c}, \bar{d}\,\right]\right) \; \stackrel{\text{def}}{=} \; \begin{cases} \left(\left[\,\bar{0}, \overline{m-1}\,\right], \bot\right) \\ \qquad \text{falls } \lfloor \frac{b^+ \cdot d^+}{m} \rfloor - \lfloor \frac{a^+ \cdot c^+}{m} \rfloor > 1 \\ \left(\left[\,\bar{0}, \overline{b^+ \cdot d^+}\,\right], \left[\,\overline{a^+ \cdot c^+}, \overline{m-1}\,\right]\right) \\ \qquad \text{falls } \lfloor \frac{b^+ \cdot d^+}{m} \rfloor - \lfloor \frac{a^+ \cdot c^+}{m} \rfloor = 1 \\ \left(\left[\,\overline{a^+ \cdot c^+}, \overline{b^+ \cdot d^+}\,\right], \bot\right) \text{ sonst} \end{cases}$$

Wir bezeichnen mit $\lfloor \frac{b^+ \cdot d^+}{m} \rfloor - \lfloor \frac{a^+ \cdot c^+}{m} \rfloor$ den Grad des Überlaufs der Multiplikation. Dieser gibt an in wieweit sich der mögliche Überlauf von $a^+ \cdot c^+$ und $b^+ \cdot d^+$ unterscheidet. Ist der Grad des Überlaufs 1 dann können wir, wie für die Addition und Subtraktion, zwei Ergebnisintervalle angeben.

Beispiel 3.2.18 Wir veranschaulichen die in der Definition der Multiplikation auftretenden Fälle mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{Z}_{16}$.

$$mul^{\#} \left(\left[\,\bar{0}, \bar{4}\,\right], \left[\,\overline{10}, \overline{14}\,\right]\right) \; \stackrel{\text{def}}{=} \; \left(\left[\,\bar{0}, \overline{15}\,\right], \bot\right) \qquad \left\lfloor \frac{4 \cdot 14}{16} \right\rfloor - \left\lfloor \frac{0 \cdot 10}{16} \right\rfloor = 3$$

$$mul^{\#} \left(\left[\,\bar{2}, \bar{5}\,\right], \left[\,\bar{3}, \bar{4}\,\right]\right) \; \stackrel{\text{def}}{=} \; \left(\left[\,\bar{0}, \bar{4}\,\right], \left[\,\bar{6}, \overline{15}\,\right]\right) \qquad \left\lfloor \frac{5 \cdot 4}{16} \right\rfloor - \left\lfloor \frac{2 \cdot 3}{16} \right\rfloor = 1$$

$$mul^{\#} \left(\left[\,\bar{1}, \bar{3}\,\right], \left[\,\bar{2}, \bar{4}\,\right]\right) \; \stackrel{\text{def}}{=} \; \left(\left[\,\bar{2}, \overline{12}\,\right], \bot\right) \qquad \left\lfloor \frac{3 \cdot 4}{16} \right\rfloor - \left\lfloor \frac{1 \cdot 2}{16} \right\rfloor = 0$$

Wir definieren zwei verschiedene Arten von Pseudodivisionen, die Methode $udiv$, die als Vertreter der Restklasse die kleinsten nicht negativen Elemente verwendet und die Methode div, wo wir die betragsmäßig kleinsten Vertreter der Restklasse einsetzen. Bei keiner dieser Funktionen tritt eine Unter- bzw. Überlauf auf. Für die Methode div müssen wir die Vorzeichen der Vertreter der Restklasse betrachten.

$$udiv^{\#} \; : \; I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m)$$
$$div^{\#} \; : \; I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m)$$

wobei

$$udiv^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \begin{cases} \textit{undefiniert}\text{ , falls } c^{+}=0 \\ \left[\,\overline{\lfloor\frac{a^{+}}{d^{+}}\rfloor},\overline{\lfloor\frac{b^{+}}{c^{+}}\rfloor}\,\right] \text{ sonst} \end{cases}$$

$$div^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \begin{cases} \textit{undefiniert}\text{ , falls } c^{-}\leq 0\leq d^{-} \\ \left[\,\overline{\lfloor\frac{a^{-}}{d^{-}}\rfloor},\overline{\lfloor\frac{b^{-}}{c^{-}}\rfloor}\,\right] \text{ falls } (a^{-}\geq 0)\wedge(c^{-}\geq 0) \\ \left[\,\overline{\lfloor\frac{b^{-}}{c^{-}}\rfloor},\overline{\lfloor\frac{a^{-}}{d^{-}}\rfloor}\,\right] \text{ falls } (a^{-}<0)\wedge(c^{-}<0) \\ \left[\,\overline{\lfloor\frac{b^{-}}{d^{-}}\rfloor},\overline{\lfloor\frac{a^{-}}{c^{-}}\rfloor}\,\right] \text{ falls } (a^{-}\geq 0)\wedge(c^{-}<0) \\ \left[\,\overline{\lfloor\frac{a^{-}}{c^{-}}\rfloor},\overline{\lfloor\frac{b^{-}}{d^{-}}\rfloor}\,\right] \text{ falls } (a^{-}<0)\wedge(c^{-}\geq 0) \end{cases}$$

Wir müssen für $div^{\#}$ nur die Werte a^{-} und c^{-} vergleichen, da $\left[\,\bar{a},\bar{b}\,\right]$ und $\left[\,\bar{c},\bar{d}\,\right]$ gültige Intervalle sind. Aus der Definition von gültigen Intervallen folgt $(b^{-}\geq 0)$ aus $(a^{-}\geq 0)$ und $(b^{-}<0)$ aus $(a^{-}<0)$.

Beispiel 3.2.19 Wir veranschaulichen die in der Definition der Operationen $udiv^{\#}$ und $div^{\#}$ auftretenden Fälle mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{Z}_{16}$.

$$udiv^{\#}\ \left(\left[\,\bar{8},\overline{13}\,\right],\left[\,\bar{2},\bar{5}\,\right]\right) \stackrel{\text{def}}{=} \left[\,\overline{\left\lfloor\frac{8}{5}\right\rfloor},\overline{\left\lfloor\frac{13}{2}\right\rfloor}\,\right] = \left[\,\bar{1},\bar{6}\,\right]$$

$$div^{\#}\ \left(\left[\,\bar{2},\bar{5}\,\right],\left[\,\bar{3},\bar{4}\,\right]\right) \stackrel{\text{def}}{=} \left[\,\overline{\left\lfloor\frac{2}{4}\right\rfloor},\overline{\left\lfloor\frac{5}{3}\right\rfloor}\,\right] = \left[\,\bar{0},\bar{1}\,\right]$$

$$div^{\#}\ \left(\left[\,\overline{-5},\overline{-3}\,\right],\left[\,\overline{-2},\overline{-1}\,\right]\right) \stackrel{\text{def}}{=} \left[\,\overline{\left\lfloor\frac{-3}{-2}\right\rfloor},\overline{\left\lfloor\frac{-5}{-1}\right\rfloor}\,\right] = \left[\,\bar{1},\bar{5}\,\right]$$

$$div^{\#}\ \left(\left[\,\overline{-7},\overline{-4}\,\right],\left[\,\bar{3},\bar{4}\,\right]\right) \stackrel{\text{def}}{=} \left[\,\overline{\left\lfloor\frac{-7}{3}\right\rfloor},\overline{\left\lfloor\frac{-4}{4}\right\rfloor}\,\right] = \left[\,\overline{-3},\overline{-1}\,\right]$$

$$div^{\#}\ \left(\left[\,\bar{4},\bar{6}\,\right],\left[\,\overline{-7},\overline{-3}\,\right]\right) \stackrel{\text{def}}{=} \left[\,\overline{\left\lfloor\frac{6}{-3}\right\rfloor},\overline{\left\lfloor\frac{4}{-7}\right\rfloor}\,\right] = \left[\,\overline{-2},\overline{-1}\,\right]$$

Eine weitere für ganze Zahlen unverzichtbare Operation ist modulo. Wir definieren zwei Arten von modulo die Methode $umod^{\#}$ und $mod^{\#}$, wo analog zu $udiv^{\#}$ und $div^{\#}$ unterschiedliche Vertreter der Restklasse verwenden werden. Wir müssen für die Operation $mod^{\#}$ die Vorzeichen der Vertreter der Restklasse beachten. Das Problem eines Unter- bzw. Überlaufs tritt hier nicht auf.

$$umod^{\#}\ :\ I^{*}(\mathbb{Z}_m)\times I^{*}(\mathbb{Z}_m)\longrightarrow I(\mathbb{Z}_m)$$
$$mod^{\#}\ :\ I^{*}(\mathbb{Z}_m)\times I^{*}(\mathbb{Z}_m)\longrightarrow I(\mathbb{Z}_m)$$

wobei

$$umod^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \begin{cases} \textit{undefiniert} \text{ falls } c^+ = 0 \\ \left[\,\bar{a},\bar{b}\,\right] \text{ falls } b^+ < c^+ \\ \left[\,\bar{0},\overline{b^+}\,\right] \text{ falls } (b^+ \geq c^+) \wedge (b^+ < d^+) \\ \left[\,\bar{0},\overline{d^+ - 1}\,\right] \text{ falls } (b^+ \geq c^+) \wedge (b^+ \geq d^+) \end{cases}$$

$$mod^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \begin{cases} \textit{undefiniert} \text{ falls } c^- \leq 0 \leq d^- \\ \left[\,\bar{a},\bar{b}\,\right] \text{ falls } (b^- < c^-) \wedge (a^- \geq 0) \wedge (c^- \geq 0) \\ \qquad \text{oder } (a^- < d^-) \wedge (a^- < 0) \wedge (c^- < 0) \\ \left[\,\bar{0},\overline{d^- - 1}\,\right] \text{ falls } (a^- \geq 0) \wedge (c^- \geq 0) \\ \left[\,\bar{0},\overline{|c^-| - 1}\,\right] \text{ falls } (a^- \geq 0) \wedge (c^- < 0) \\ \left[\,\overline{-(d^-) + 1},\bar{0}\,\right] \text{ falls } (a^- < 0) \wedge (c^- \geq 0) \\ \left[\,\overline{c^- + 1},\bar{0}\,\right] \text{ falls } (a^- < 0) \wedge (c^- < 0) \end{cases}$$

Wir schätzen die Ergebnisintervalle durch Betrachten und Vergleichen der entsprechenden Vertreter der Restklassen noch oben ab. Nach einer Konvention für modulo Operationen ist das Ergebnis von $x \bmod y$ in $\mathbb{Z}$ nur dann negativ, wenn $x < 0$ und y kein Teiler von x ist.

Beispiel 3.2.20 Wir veranschaulichen die in der Definition der Operationen $umod^{\#}$ und $mod^{\#}$ auftretenden Fälle mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{Z}_{16}$.

$$\begin{aligned}
umod^{\#}\left(\left[\,\bar{1},\bar{7}\,\right],\left[\,\bar{8},\bar{9}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{1},\bar{7}\,\right] \\
umod^{\#}\left(\left[\,\bar{0},\bar{7}\,\right],\left[\,\bar{3},\bar{5}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{0},\bar{4}\,\right] \\
umod^{\#}\left(\left[\,\bar{1},\bar{5}\,\right],\left[\,\bar{2},\bar{6}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{0},\bar{5}\,\right] \\
mod^{\#}\left(\left[\,\overline{-3},\overline{-2}\,\right],\left[\,\overline{-6},\overline{-4}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\overline{-3},\overline{-2}\,\right] \\
mod^{\#}\left(\left[\,\bar{1},\bar{4}\,\right],\left[\,\bar{5},\bar{7}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{1},\bar{4}\,\right] \\
mod^{\#}\left(\left[\,\bar{4},\bar{7}\,\right],\left[\,\bar{3},\bar{5}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{0},\bar{4}\,\right] \\
mod^{\#}\left(\left[\,\bar{1},\bar{5}\,\right],\left[\,\overline{-4},\overline{-2}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{0},\bar{3}\,\right] \\
mod^{\#}\left(\left[\,\overline{-6},\overline{-3}\,\right],\left[\,\bar{2},\bar{4}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\overline{-3},\bar{0}\,\right] \\
mod^{\#}\left(\left[\,\overline{-5},\overline{-2}\,\right],\left[\,\overline{-3},\overline{-1}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\overline{-2},\bar{0}\,\right]
\end{aligned}$$

Für Funktionen mit nur einem Argument verwenden wir das folgende, leicht modifizierte Schema:

$$\begin{aligned}
&op^{\#} \ :\ I_n^*(\mathbb{Z}_m) \longrightarrow I_n^*(\mathbb{Z}_m) \\
&op^{\#}\ \left(\left(\left[\,\overline{a_i},\overline{b_i}\,\right]\right)_{i=1}^{n}\right) \stackrel{\text{def}}{=} \texttt{merge}\left(\left(op^{\#}\left(\left[\,\overline{a_i},\overline{b_i}\,\right]\right)\right)_{i=1}^{n}\right)
\end{aligned}$$

Es stehen Funktionen zur Negation, zur Wurzelberechnung auf ganzen Zahlen und zur Berechnung des Betrags zur Verfügung. Wir definieren analog zu den arithmetischen Operationen mit zwei Argumenten die Funktionen $neg^{\#}$, $sqrt^{\#}$ und $abs^{\#}$. Für diese Funktionen verwenden wir die Darstellung mit den betragsmäßig kleinsten Vertretern der Restklasse.

$$\begin{aligned} neg^{\#} &: I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \\ sqrt^{\#} &: I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \\ abs^{\#} &: I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \end{aligned}$$

wobei

$$\begin{aligned} neg^{\#}\left(\left[\,\bar{a},\bar{b}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\overline{-(b^-)},\overline{-(a^-)}\,\right] \\ sqrt^{\#}\left(\left[\,\bar{a},\bar{b}\,\right]\right) &\stackrel{\text{def}}{=} \begin{cases} \textit{undefiniert}\text{ , falls } a^- < 0 \\ \left[\,\overline{\lfloor\sqrt{a^-}\rfloor},\overline{\lfloor\sqrt{b^-}\rfloor}\,\right] \text{ sonst} \end{cases} \\ abs^{\#}\left(\left[\,\bar{a},\bar{b}\,\right]\right) &\stackrel{\text{def}}{=} \begin{cases} \left[\,\bar{a},\bar{b}\,\right] \text{ falls } a^- \geq 0 \\ \left[\,\overline{-(b^-)},\overline{-(a^-)}\,\right] \text{ sonst} \end{cases} \end{aligned}$$

Beispiel 3.2.21 Wir veranschaulichen die Definition der Operationen neg, $sqrt$ und abs mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{Z}_{16}$.

$$\begin{aligned} neg^{\#}\left(\left[\,\bar{2},\bar{6}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\overline{-6},\overline{-2}\,\right] \\ sqrt^{\#}\left(\left[\,\bar{1},\bar{8}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{1},\bar{2}\,\right] \\ abs^{\#}\left(\left[\,\overline{-4},\overline{-3}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{3},\bar{4}\,\right] \end{aligned}$$

Wir interessieren uns nicht nur für arithmetische Operationen auf $\mathbb{Z}_m$, sondern auch für Bitoperationen wie and, or, xor und not. Diese Operationen auf Intervallen zu definieren ist schwierig, weshalb wir zur Vereinfachung einen weiteren Approximationsschritt einführen, der ein Intervall in einen Vektor möglicher Bits umwandelt.

$$\begin{aligned} \alpha &: I(\mathbb{Z}_m) \longrightarrow P\left(\{0,1\}^n\right) \\ \gamma &: P\left(\{0,1\}^n\right) \longrightarrow I(\mathbb{Z}_m) \end{aligned}$$

Die dazugehörigen Funktionen α und γ sind maschinenabhängig und werden in Kapitel 4 genauer erläutert. Wir erhalten die folgende Beziehung:

$$\begin{array}{ccc} I(\mathbb{Z}_m) & \xrightarrow{post^{\#}} & I(\mathbb{Z}_m) \\ \alpha\downarrow & & \uparrow\stackrel{\subseteq}{\gamma} \\ P\left(\{0,1\}^n\right) & \xrightarrow{post^{\natural}} & P\left(\{0,1\}^n\right) \end{array}$$

Die Bitoperationen $and^\natural$, $or^\natural$, $xor^\natural$ und $not^\natural$ werden folgendermaßen definiert.

$$\begin{aligned} and^\natural &: P(\{0,1\}^n) \times P(\{0,1\}^n) \longrightarrow P(\{0,1\}^n) \\ or^\natural &: P(\{0,1\}^n) \times P(\{0,1\}^n) \longrightarrow P(\{0,1\}^n) \\ xor^\natural &: P(\{0,1\}^n) \times P(\{0,1\}^n) \longrightarrow P(\{0,1\}^n) \\ not^\natural &: P(\{0,1\}^n) \longrightarrow P(\{0,1\}^n) \end{aligned}$$

wobei

$$and^\natural\left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}\right) \stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ mit } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\}, \text{ falls } a_i = \{0\} \text{ oder } b_i = \{0\} \\ \{1\}, \text{ falls } a_i = \{1\} \text{ und } b_i = \{1\} \\ \{0,1\}, \text{ sonst} \end{cases}$$

$$or^\natural\left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}\right) \stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ mit } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\}, \text{ falls } a_i = \{0\} \text{ und } b_i = \{0\} \\ \{1\}, \text{ falls } a_i = \{1\} \text{ oder } b_i = \{1\} \\ \{0,1\}, \text{ sonst} \end{cases}$$

$$xor^\natural\left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}\right) \stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ mit } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\} \text{ falls } a_i = \{0\} \text{ und } b_i = \{0\} \\ \quad \text{oder } a_i = \{1\} \text{ und } b_i = \{1\} \\ \{1\} \text{ falls } a_i = \{0\} \text{ und } b_i = \{1\} \\ \quad \text{oder } a_i = \{1\} \text{ und } b_i = \{0\} \\ \{0,1\} \text{ sonst} \end{cases}$$

$$not^\natural\left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}\right) \stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ mit } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\}, \text{ falls } a_i = \{1\} \\ \{1\}, \text{ falls } a_i = \{0\} \\ \{0,1\}, \text{ sonst} \end{cases}$$

Wir führen die Bitoperation and, or, xor und not für jedes Element des Bitvektors durch. Hierzu haben wir die Operationen um die Menge $\{0,1\}$ als Eingabeargument erweitert. Das Ergebnis dieser erweiterten Bitoperation kann die Menge $\{0,1\}$ sein.

Interessant sind auch die so genannten Shift- und Rotatefunktionen auf $\mathbb{Z}_m$. Die Operation $shift\_left^\#$ ist äquivalent zu einer Multiplikation mit der entsprechenden Zweierpotenz bei der das Ergebnis eines Überlaufs 0 ist und nicht modulo m berechnet wird. Eine zur Division äquivalente Operation $shift\_right^\#$ existiert ebenfalls. Wir verwenden aus diesem Grund das ursprüngliche Approximationsschema.

$$\begin{aligned} shift\_left^\# &: I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \times I(\mathbb{Z}_m) \\ shift\_right^\# &: I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \end{aligned}$$

wobei

$$shift\_left^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \begin{cases} \left(\left[\,\bar{0},\bar{0}\,\right],\bot\right) \text{ falls } a^+ \cdot 2^{c^+} \geq m \\ \left(\left[\,\overline{a^+ \cdot 2^{c^+}},\overline{m-1}\,\right],\left[\,\bar{0},\bar{0}\,\right]\right) \\ \qquad \text{falls } (a^+ \cdot 2^{c^+} < m) \wedge (b^+ \cdot 2^{d^+} \geq m) \\ \left(\left[\,\overline{a^+ \cdot 2^{c^+}},\overline{b^+ \cdot 2^{d^+}}\,\right],\bot\right) \text{ sonst} \end{cases}$$

$$shift\_right^{\#}\ \left(\left[\,\bar{a},\bar{b}\,\right],\left[\,\bar{c},\bar{d}\,\right]\right) \stackrel{\text{def}}{=} \left[\,\overline{\left\lfloor \frac{a^+}{2^{d^+}} \right\rfloor},\overline{\left\lfloor \frac{b^+}{2^{c^+}} \right\rfloor}\,\right]$$

Beispiel 3.2.22 Wir veranschaulichen die in der Definition der Operationen $shift\_left^{\#}$ und $shift\_right^{\#}$ auftretenden Fälle mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{Z}_{16}$.

$$\begin{aligned} shift\_left^{\#}\left(\left[\,\bar{5},\bar{7}\,\right],\left[\,\bar{3},\bar{4}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\bar{0},\bar{0}\,\right],\bot\right) \\ shift\_left^{\#}\left(\left[\,\bar{1},\bar{3}\,\right],\left[\,\bar{2},\bar{5}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\bar{4},\overline{15}\,\right],\left[\,\bar{0},\bar{0}\,\right]\right) \\ shift\_left^{\#}\left(\left[\,\bar{2},\bar{3}\,\right],\left[\,\bar{1},\bar{2}\,\right]\right) &\stackrel{\text{def}}{=} \left(\left[\,\bar{4},\overline{12}\,\right],\bot\right) \\ shift\_right^{\#}\left(\left[\,\bar{2},\bar{3}\,\right],\left[\,\bar{3},\bar{5}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{0},\bar{0}\,\right] \\ shift\_right^{\#}\left(\left[\,\bar{1},\bar{7}\,\right],\left[\,\bar{1},\bar{2}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{0},\bar{3}\,\right] \\ shift\_right^{\#}\left(\left[\,\bar{8},\overline{11}\,\right],\left[\,\bar{0},\bar{2}\,\right]\right) &\stackrel{\text{def}}{=} \left[\,\bar{2},\overline{11}\,\right] \end{aligned}$$

Die Funktionen $shift\_left^{\#}$ und $shift\_right^{\#}$ implementieren einen bitweisen Shift für die Wahl der kleinsten, nicht negativen Vertreter der Restklasse. Die, den arithmetischen Shift implementierenden, Funktionen $ashift\_left^{\#}$ und $ashift\_right^{\#}$ werden analog definiert. Wir verwenden hier nur die betragsmäßig kleinsten Vertreter der Restklasse. Wir verwenden hier a^- anstatt a^+ für eine Restklasse $\bar{a} \in \mathbb{Z}_m$.

Wir unterstützen die Funktionen $rotate\_left^{\natural}$ und $rotate\_right^{\natural}$ ebenfalls. Diese rotieren, analog zu den Shiftfunktionen, die Bitvektoren nach links, bzw. rechts. Wir rotieren diese Bits allerdings nur für feste Werte. Andernfalls schätzen wir den resultierenden Wert mit $\top$ ab.

$$\begin{aligned} rotate\_left^{\natural} &: P\left(\{0,1\}^n\right) \times I^*(\mathbb{Z}_m) \longrightarrow P\left(\{0,1\}^n\right) \\ rotate\_right^{\natural} &: P\left(\{0,1\}^n\right) \times I^*(\mathbb{Z}_m) \longrightarrow P\left(\{0,1\}^n\right) \end{aligned}$$

wobei

$$\begin{aligned} rotate\_left^{\natural}\left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix},\left[\,\bar{b},\bar{b}\,\right]\right) &\stackrel{\text{def}}{=} \begin{pmatrix} d_1 \\ \vdots \\ d_n \end{pmatrix} \\ rotate\_right^{\natural}\left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix},\left[\,\bar{b},\bar{b}\,\right]\right) &\stackrel{\text{def}}{=} \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix} \end{aligned}$$

mit $d_i = a_{((i+b^+-1) \bmod n)+1}$ und $e_i = a_{((i-b^+-1) \bmod n)+1}$ und $a_i \neq \{0,1\}$ für $1 \leq i \leq n$.

3.2.2 Reelle Zahlen

Wir betrachten für die XRTL Analyse nicht nur ganze Zahlen, sondern auch reelle Zahlen, genauer gesagt endliche Teilmengen von $\mathbb{R}$, da XRTL auch Fließkommazahlen unterstützt. Auf konkret verwendete Instanzen dieser Teilmengen gehen wir in Kapitel 4 genauer ein.

Definition 3.2.23 *Wir bezeichnen mit $\mathbb{R}_m$, wobei $m \in \mathbb{R}$, eine endliche Teilmenge von $\mathbb{R}$ mit gleich verteilten Elementen $x \in \mathbb{R}$ für die $x \leq m$ und $x \geq -m$ gilt. Zusätzlich sind $0 \in \mathbb{R}_m$ und $1 \in \mathbb{R}_m$.*

Wir führen für $\mathbb{R}_m$ zwei zusätzliche Elemente $\bot$ und $\top$ ein. Für diese verwenden wir die folgenden Rechenregeln:

$$
\begin{aligned}
\forall x \in \mathbb{R}_m \setminus \{\bot, \top\} \text{ gilt:} \quad & \bot < x \text{ und } x < \top \\
& \bot < \top \\
& \bot \pm x \stackrel{\text{def}}{=} \bot \text{ und } \top \pm x \stackrel{\text{def}}{=} \top \\
& x + \bot \stackrel{\text{def}}{=} \bot \text{ und } x + \top \stackrel{\text{def}}{=} \top \\
& x - \bot \stackrel{\text{def}}{=} \top \text{ und } x - \top \stackrel{\text{def}}{=} \bot \\
& \frac{x}{\bot} \stackrel{\text{def}}{=} 0 \text{ und } \frac{x}{\top} \stackrel{\text{def}}{=} 0 \\
& x \cdot \bot \stackrel{\text{def}}{=} \begin{cases} \bot, \text{falls } x > 0 \\ \top, \text{falls } x < 0 \\ 0, \text{sonst} \end{cases} \\
& x \cdot \top \stackrel{\text{def}}{=} \begin{cases} \top, \text{falls } x > 0 \\ \bot, \text{falls } x < 0 \\ 0, \text{sonst} \end{cases} \\
& \frac{\bot}{x} \stackrel{\text{def}}{=} \begin{cases} \bot, \text{falls } x > 0 \\ \top, \text{falls } x < 0 \\ 0, \text{sonst} \end{cases} \\
& \frac{\top}{x} \stackrel{\text{def}}{=} \begin{cases} \top, \text{falls } x > 0 \\ \bot, \text{falls } x < 0 \\ 0, \text{sonst} \end{cases}
\end{aligned}
$$

Definition 3.2.24 *Mit dem Intervall $\big[a, b\big]$, wobei $a, b \in \mathbb{R}_m$ und $a \leq b$, bezeichnen wir die Menge der reellen Zahlen $x \in \mathbb{R}_m$, für die $a \leq x$ und $x \leq b$ gilt.*

Wir bezeichnen mit $I(\mathbb{R}_m)$ die Menge aller Intervalle über $\mathbb{R}_m$ wobei

$$I(\mathbb{R}_m) \stackrel{def}{=} \{\bot\} \cup \{\big[a, b\big] \mid \big[a, b\big] \textit{ Intervall über } \mathbb{R}_m\}$$

Mit $\bot$ bezeichnen wir hier das leere Intervall.

Wir verwenden für arithmetische Operationen auf $\mathbb{R}_m$ die korrespondierenden Operationen auf $\mathbb{R}$. Da das Ergebnis einer Berechnung in $\mathbb{R}$, aber möglicherweise nicht in $\mathbb{R}_m$ liegt, benötigen wir die folgenden Funktionen.

$$\begin{aligned} inf &: \mathbb{R} \longrightarrow \mathbb{R}_m \\ sup &: \mathbb{R} \longrightarrow \mathbb{R}_m \end{aligned}$$

wobei

$$inf(a) \stackrel{\text{def}}{=} \begin{cases} a, \text{ falls } a \in \mathbb{R}_m \\ \top, \text{ falls } a > m \\ \bot, \text{ falls } a < -m \\ max\{x \mid x \in \mathbb{R}_m, x < a\}, \text{ sonst} \end{cases}$$

$$sup(a) \stackrel{\text{def}}{=} \begin{cases} a, \text{ falls } a \in \mathbb{R}_m \\ \top, \text{ falls } a > m \\ \bot, \text{ falls } a < -m \\ min\{x \mid x \in \mathbb{R}_m, x > a\}, \text{ sonst} \end{cases}$$

Mit dem Überlauf und Unterlauf von Operationen in $\mathbb{R}_m$ müssen wir uns, aufgrund der auf $\mathbb{R}_m$ eingeführten Elemente $\bot$ und $\top$, nicht beschäftigen. Dieses Verhalten entspricht dem der Fließkommazahlen in der Praxis.

Wir führen den Begriff der gültigen Intervalle über $\mathbb{R}_m$ aus rein praktischen Gründen ein. Es erleichtert uns später die Implementierung einzelner Operationen.

Definition 3.2.25 *Wir bezeichnen ein Intervall $[a, b]$, mit $a, b \in \mathbb{R}_m$ und $a \leq b$ als gültig, wenn $a \geq 0$ oder $b < 0$ gilt.*

Mit $I^*(\mathbb{R}_m)$ bezeichnen wir die Menge aller gültigen Intervalle über $\mathbb{R}_m$ wobei

$$I^*(\mathbb{R}_m) \stackrel{def}{=} \{\bot\} \cup \{[a, b] \mid [a, b] \textit{ gültiges Intervall über } \mathbb{R}_m\}$$

Analog zur Beziehung von Intervallen und gültigen Intervallen über $\mathbb{Z}_m$ können wir für Intervalle über $\mathbb{R}_m$ das nachfolgende Lemma angeben.

Lemma 3.2.26 *Jedes Intervall $[a, b]$ über $\mathbb{R}_m$ ist gültig, oder kann in zwei gültige Intervalle aufgeteilt werden, welche alle Elemente aus $[a, b]$ enthalten.*

Beweis: Konstruktion:

$$[a, b] \Rightarrow \begin{cases} [a_1, b_1] \stackrel{\text{def}}{=} [0, b] \\ [a_2, b_2] \stackrel{\text{def}}{=} [a, max\{x \mid x \in \mathbb{R}_m, x < 0\}] \end{cases}$$

■

Wir verwenden auch hier, analog zu den ganzen Zahlen, Listen von gültigen Intervallen mit beschränkter maximaler Länge. Dies erlaubt es uns die Präzision der Approximation den Bedürfnissen anzupassen.

Definition 3.2.27 *Wir bezeichnen mit* $\left(\left[a_i, b_i\right]\right)_{i=1}^{n}$ *die Liste von gültigen Intervallen* $\left[a_i, b_i\right]$ *mit maximaler Länge* n*. Für ein* $x \in \mathbb{R}_m$ *gilt* $x \in \left(\left[a_i, b_i\right]\right)_{i=1}^{n}$*, falls* $\exists i$ *mit* $x \in \left[a_i, b_i\right]$*.*

Mit $I_n^*(\mathbb{R}_m)$ *bezeichnen wir die Menge aller Listen von gültigen Intervallen über* $\mathbb{R}_m$ *mit maximaler Länge* n*.*

Die im folgenden definierten Funktionen α und γ bilden eine Galoisverknüpfung zwischen $P(\mathbb{R}_m)$ und $I_n^*(\mathbb{R}_m)$. Wir verwenden hierfür einen leicht modifizierten `split` - Algorithmus.

$$\begin{aligned} \alpha &: P(\mathbb{R}_m) \longrightarrow I_n^*(\mathbb{R}_m) \\ \gamma &: I_n^*(\mathbb{R}_m) \longrightarrow P(\mathbb{R}_m) \end{aligned}$$

wobei

$$\begin{aligned} \alpha(X) &\stackrel{\text{def}}{=} \texttt{split}(X) \\ \gamma(\left(\left[a_i, b_i\right]\right)_{i=1}^{n}) &\stackrel{\text{def}}{=} \{x \in \mathbb{R}_m \mid \exists i \text{ mit } x \in \left[a_i, b_i\right]\} \end{aligned}$$

Der für $\mathbb{R}_m$ modifizierte Algorithmus `split` entspricht dem für ganze Zahlen, wobei nur die entsprechenden Verfahren für Intervalle, bzw. gültige Intervalle über $\mathbb{R}_m$ verwendet werden.

Zwischen der konkreten und abstrakten Semantik ergibt sich unter Verwendung der Galoisverknüpfung die folgende Beziehung.

$$\begin{array}{ccc} P(\mathbb{R}_m) & \xrightarrow{post} & P(\mathbb{R}_m) \\ {\scriptstyle\alpha}\downarrow & & \uparrow{\scriptstyle\subseteq\atop\gamma} \\ I_n^*(\mathbb{R}_m) & \xrightarrow{post^\#} & I_n^*(\mathbb{R}_m) \end{array}$$

Für die Implementierung der Operationen auf $I_n^*(\mathbb{R}_m)$ verwenden wir ein ähnliches Schema wie für $I_n^*(\mathbb{Z}_m)$.

$$op^\# : I_n^*(\mathbb{R}_m) \times I_n^*(\mathbb{R}_m) \longrightarrow I_n^*(\mathbb{R}_m)$$

$$\begin{aligned} op^\#\left(\left(\left[a_i, b_i\right]\right)_{i=1}^{n}, \left(\left[c_j, d_j\right]\right)_{j=1}^{n}\right) &\stackrel{\text{def}}{=} \\ \texttt{merge}\left(\left(op^\#\left(\left[a_i, b_i\right], \left[c_j, d_j\right]\right)\right)_{i=1,j=1}^{n}\right) & \end{aligned}$$

Wir verwenden den für $I_n^*(\mathbb{R}_m)$ modifizierten Algorithmus `merge` (Algorithmus 3.2.13) für $I_n^*(\mathbb{Z}_m)$. Den Begriff der Distanz zweier Intervalle über $\mathbb{R}_m$ definieren wir folgendermaßen.

Definition 3.2.28 *Für zwei Intervalle* $\big[a,b\big]$ *und* $\big[c,d\big]$ *mit* $a,b,c,d \in \mathbb{R}_m$ *und* $a \leq c$ *definieren wir die Distanz von* $\big[a,b\big]$ *und* $\big[c,d\big]$ *als*

$$\begin{aligned} &distanz \ : \ I(\mathbb{R}_m) \times I(\mathbb{R}_m) \longrightarrow \mathbb{R}_m \\ &distanz \ \ \big(\big[a,b\big],\big[c,d\big]\big) \stackrel{def}{=} \begin{cases} 0, \textit{ falls } c \leq b \\ sup(c-b) \textit{ sonst} \end{cases} \end{aligned}$$

Wir führen nun, analog zu den ganzen Zahlen, die Operationen auf gültigen Intervallen über $\mathbb{R}_m$ ein. Durch die Einführung der Symbole $\perp$ und $\top$ auf $\mathbb{R}_m$ kommt es hier zu keinem Unterlauf oder Überlauf. Die arithmetischen Grundoperationen sehen folgendermaßen aus.

$$\begin{aligned} add^{\#} \ &: \ I^*(\mathbb{R}_m) \times I^*(\mathbb{R}_m) \longrightarrow I(\mathbb{R}_m) \\ sub^{\#} \ &: \ I^*(\mathbb{R}_m) \times I^*(\mathbb{R}_m) \longrightarrow I(\mathbb{R}_m) \\ mul^{\#} \ &: \ I^*(\mathbb{R}_m) \times I^*(\mathbb{R}_m) \longrightarrow I(\mathbb{R}_m) \end{aligned}$$

wobei

$$\begin{aligned} add^{\#} \ \big(\big[a,b\big],\big[c,d\big]\big) &\stackrel{\text{def}}{=} \big[inf(a+c), sup(b+d)\big] \\ sub^{\#} \ \big(\big[a,b\big],\big[c,d\big]\big) &\stackrel{\text{def}}{=} \big[inf(a-d), sup(b-c)\big] \\ mul^{\#} \ \big(\big[a,b\big],\big[c,d\big]\big) &\stackrel{\text{def}}{=} \begin{cases} \big[inf(a\cdot c), sup(b\cdot d)\big], \text{falls } a \geq 0 \text{ und } c \geq 0 \\ \big[inf(b\cdot d), sup(a\cdot c)\big], \text{falls } a < 0 \text{ und } c < 0 \\ \big[inf(a\cdot d), sup(b\cdot c)\big], \text{falls } a < 0 \text{ und } c \geq 0 \\ \big[inf(b\cdot c), sup(a\cdot d)\big], \text{falls } a \geq 0 \text{ und } c < 0 \end{cases} \end{aligned}$$

Wir müssen weniger Fälle betrachten, da für jedes Element x eines gültigen Eingabeintervalls über $\mathbb{R}_m$ nach Definition 3.2.25 entweder $x \geq 0$ oder $x < 0$ gilt.

Beispiel 3.2.29 Wir veranschaulichen die Definition der Addition, Subtraktion und Multiplikation mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{R}_{100}$.

$$\begin{aligned} add^{\#} \ \big(\big[13,27\big],\big[7,35\big]\big) &\stackrel{\text{def}}{=} \big[20,62\big] \\ add^{\#} \ \big(\big[-65,-10\big],\big[9,12\big]\big) &\stackrel{\text{def}}{=} \big[-56,2\big] \\ sub^{\#} \ \big(\big[27,56\big],\big[12,29\big]\big) &\stackrel{\text{def}}{=} \big[-2,44\big] \\ sub^{\#} \ \big(\big[23,51\big],\big[-38,-31\big]\big) &\stackrel{\text{def}}{=} \big[8,89\big] \\ mul^{\#} \ \big(\big[5,9\big],\big[6,8\big]\big) &\stackrel{\text{def}}{=} \big[30,72\big] \\ mul^{\#} \ \big(\big[-7,-3\big],\big[-8,-2\big]\big) &\stackrel{\text{def}}{=} \big[6,56\big] \\ mul^{\#} \ \big(\big[-3,-1\big],\big[10,12\big]\big) &\stackrel{\text{def}}{=} \big[-36,-10\big] \\ mul^{\#} \ \big(\big[5,6\big],\big[-13,-4\big]\big) &\stackrel{\text{def}}{=} \big[-78,-20\big] \end{aligned}$$

Die Ergebnisintervalle $\big[-56,2\big]$ und $\big[-2,44\big]$ sind nicht gültig.

Für Intervalle über $\mathbb{R}_m$ benötigen wir nur eine Art der Division.

$$div^{\#} \ : \ I^*(\mathbb{R}_m) \times I^*(\mathbb{R}_m) \longrightarrow I(\mathbb{R}_m)$$

wobei

$$div^{\#} \ \big([a,b],[c,d]\big) \ \stackrel{\text{def}}{=} \ \begin{cases} \textit{undefiniert}, \text{ falls c = 0} \\ \big[inf(\frac{a}{d}), sup(\frac{b}{c})\big], \text{ falls } a \geq 0 \text{ und } c \geq 0 \\ \big[inf(\frac{b}{c}), sup(\frac{a}{d})\big], \text{ falls } a < 0 \text{ und } c < 0 \\ \big[inf(\frac{a}{c}), sup(\frac{b}{d})\big], \text{ falls } a < 0 \text{ und } c \geq 0 \\ \big[inf(\frac{b}{d}), sup(\frac{a}{c})\big], \text{ falls } a \geq 0 \text{ und } c < 0 \end{cases}$$

Beispiel 3.2.30 Wir veranschaulichen die Definition der Division mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{R}_{100}$.

$$\begin{aligned} div^{\#} \big([45,72],[6,9]\big) &\stackrel{\text{def}}{=} [5,12] \\ div^{\#} \big([-42,-36],[-18,-2]\big) &\stackrel{\text{def}}{=} [2,21] \\ div^{\#} \big([-27,-22],[9,11]\big) &\stackrel{\text{def}}{=} [-3,-2] \\ div^{\#} \big([39,75],[-13,-5]\big) &\stackrel{\text{def}}{=} [-15,-3] \end{aligned}$$

Wir verwenden für Funktionen mit nur einem Argument das folgende modifizierte Schema.

$$\begin{aligned} &op^{\#} \ : \ I_n^*(\mathbb{R}_m) \longrightarrow I_n^*(\mathbb{R}_m) \\ &op^{\#} \ \big(([a_i,b_i])_{i=1}^n\big) \stackrel{\text{def}}{=} \texttt{merge}\left(\left(op^{\#}\big([a_i,b_i]\big)\right)_{i=1}^n\right) \end{aligned}$$

Wir definieren, wie für die ganzen Zahlen, Funktionen zur Negation, zur Wurzelberechnung und zur Berechnung des Betrags.

$$\begin{aligned} neg^{\#} \ &: \ I^*(\mathbb{R}_m) \longrightarrow I(\mathbb{R}_m) \\ sqrt^{\#} \ &: \ I^*(\mathbb{R}_m) \longrightarrow I(\mathbb{R}_m) \\ abs^{\#} \ &: \ I^*(\mathbb{R}_m) \longrightarrow I(\mathbb{R}_m) \end{aligned}$$

wobei

$$\begin{aligned} neg^{\#} \ \big([a,b]\big) \ &\stackrel{\text{def}}{=} \ \big[inf(-b), sup(-a)\big] \\ sqrt^{\#} \ \big([a,b]\big) \ &\stackrel{\text{def}}{=} \ \begin{cases} \textit{undefiniert}, \text{ falls } a < 0 \\ \big[inf(\sqrt{a}), sup(\sqrt{b})\big], \text{ sonst} \end{cases} \\ abs^{\#} \ \big([a,b]\big) \ &\stackrel{\text{def}}{=} \ \begin{cases} [a,b] \text{ falls } a \geq 0 \\ \big[inf(-b), sup(-a)\big], \text{ sonst} \end{cases} \end{aligned}$$

Beispiel 3.2.31 Wir veranschaulichen die Definition der Operationen $neg^{\#}$, $sqrt^{\#}$ und $abs^{\#}$ mit Hilfe von Beispielen für gültige Intervalle über $\mathbb{R}_{100}$.

$$\begin{aligned} neg^{\#}\left(\left[51,63\right]\right) &\stackrel{\text{def}}{=} \left[-63,-51\right] \\ neg^{\#}\left(\left[-47,-31\right]\right) &\stackrel{\text{def}}{=} \left[31,47\right] \\ sqrt^{\#}\left(\left[1,81\right]\right) &\stackrel{\text{def}}{=} \left[1,9\right] \\ abs^{\#}\left(\left[-91,-72\right]\right) &\stackrel{\text{def}}{=} \left[72,91\right] \\ abs^{\#}\left(\left[9,23\right]\right) &\stackrel{\text{def}}{=} \left[9,23\right] \end{aligned}$$

Für gültige Intervalle über $\mathbb{R}_m$ existieren keine Bitoperationen wie $and, or, xor, \ldots$ oder Shift- und Rotatefunktionen. Diese machen aufgrund der konkreten Darstellung, die wir in Kapitel 4 vorstellen, keinen Sinn.

3.2.3 Speicherstücke und Zeiger

Aus den vorgestellten endlichen Mengen von ganzen und reellen Zahlen konstruieren wir Speicherstücke. Diese können im Gegensatz zu Registern einen oder mehrere Werte unterschiedlichen Typs speichern. In der Praxis werden $\mathbb{Z}_m$ und $\mathbb{R}_{m'}$ mit unterschiedlichen m bzw. m' verwendet. Für $\mathbb{R}_{m'}$ ist die Anzahl der Elemente und für $\mathbb{Z}_m$ die Anzahl der kleinsten nicht negativen oder betragsmäßig kleinsten Vertreter der Restklasse endlich und abhängig von m bzw. m'. Wir führen eine symbolische Größe ein, der dem Bedarf an Speicherplatz dieser Mengen, abhängig von m bzw. m' entspricht. Auf den Speicherplatz der verwendeten Instanzen gehen wir in Kapitel 4 genauer ein.

Definition 3.2.32 *Wir bezeichnen mit $|\mathbb{Z}_m|$ und $|\mathbb{R}_{m'}|$, wobei $m \in \mathbb{N}$ und $m' \in \mathbb{R}$ die Menge an Speicherplatz, die $\mathbb{Z}_m$ bzw. $\mathbb{R}_{m'}$ benötigt.*

Die Elemente $x \in P(\mathbb{Z}_m)$ und $y \in I_n^*(\mathbb{Z}_m)$ benötigen Speicherplatz $|\mathbb{Z}_m|$. Wir definieren den Speicherbedarf von x und y als $|x| = |\mathbb{Z}_m|$ und $|y| = |\mathbb{Z}_m|$. Dies gilt analog für jedes Element aus $P(\mathbb{R}_{m'})$ und $I_n^*(\mathbb{R}_{m'})$. Speicherstücke bestehen aus einer Liste von Paaren $(\mathbb{Z}, P(\mathbb{Z}_m) \cup P(\mathbb{R}_{m'}))$ von Offset und dem gespeicherten Datentyp. Diese Paare sind nach der ersten Komponente (Offset) aufsteigend geordnet, wobei zu beachten ist, dass es zu keinen Überlappungen kommen darf.

Definition 3.2.33 *Wir bezeichnen mit $\mathcal{M}_{i=1}^{k}(o_i, d_i)$ ein Speicherstück mit k Komponenten, wobei $o_i \in \mathbb{Z}$ und $d_i \in I_n^*(\mathbb{Z}_m) \cup I_n^*(\mathbb{R}_m)$.*

Mit $\mathcal{M}_k$ bezeichnen wir die Menge aller Speicherstücke mit maximal k Komponenten.

Die Größe des verwendeten Speicherplatz für ein $x \in \mathcal{M}_k$ mit $k' \leq k$ Komponenten ist folgendermaßen definiert.

$$|x| \stackrel{\text{def}}{=} o_{k'} + |d_{k'}|$$

Es darf innerhalb der Komponenten von x zu keinerlei Überlappungen kommen. Es gilt:

$$o_i + |d_i| \leq o_{i+1} \qquad \forall\, 1 \leq i < k'$$

Gilt für eine Komponente i sogar $o_i + |d_i| < o_{i+1}$ dann sprechen wir von einer so genannten Speicherlücke innerhalb eines Speicherstücks. Dies kommt in der Praxis aufgrund des so genannten „alignments“ vor, d.h. um den Zugriff zu beschleunigen muss das Offset, abhängig vom Maschinentyp, unterschiedlichen Bedingungen genügen.

Die Approximation von Speicherstücken geschieht komponentenweise. Wir verwenden hierfür die bereits vorgestellten Approximationen. Zwischen der konkreten und der abstrakten Semantik ergibt sich mit Hilfe der Galoisverknüpfung folgende Beziehung.

$$\begin{array}{ccc} P\left(\mathcal{M}^k_{i=1}(\mathbb{Z}, P(\mathbb{Z}_m) \cup P(\mathbb{R}_{m'})\right) & \xrightarrow{post} & P\left(\mathcal{M}^k_{i=1}(\mathbb{Z}, P(\mathbb{Z}_m) \cup P(\mathbb{R}_{m'})\right) \\ \alpha \downarrow & & \uparrow \subseteq_\gamma \\ \mathcal{M}_k & \xrightarrow{post^\#} & \mathcal{M}_k \end{array}$$

Wir verwenden für α und γ die auf $I^*_n(\mathbb{Z}_m)$ und $I^*_n(\mathbb{R}_{m'})$ eingeführten α und γ.

$$\begin{aligned} \alpha &: P\left(\mathcal{M}^k_{i=1}(\mathbb{N}, P(\mathbb{Z}_m) \cup P(\mathbb{R}_{m'}))\right) \longrightarrow \mathcal{M}_k \\ \gamma &: \mathcal{M}_k \longrightarrow P\left(\mathcal{M}^k_{i=1}(\mathbb{N}, P(\mathbb{Z}_m) \cup P(\mathbb{R}_{m'}))\right) \end{aligned}$$

wobei

$$\begin{aligned} \alpha\ \left(\mathcal{M}^k_{i=1}(o_i, pd_i)\right) &\stackrel{\text{def}}{=} \mathcal{M}^k_{i=1}(o_i, \alpha(pd_i)) \\ \gamma\ \left(\mathcal{M}^k_{i=1}(o_i, d_i)\right) &\stackrel{\text{def}}{=} \mathcal{M}^k_{i=1}(o_i, \gamma(d_i)) \end{aligned}$$

Es existiert nur die Operation $store^\#$, die Speicherstücke verändert.

$$\begin{aligned} store^\# &: \mathcal{M}_k \times \mathbb{Z} \times (I^*_n(\mathbb{Z}_m) \cup I^*_n(\mathbb{R}_{m'})) \longrightarrow \mathcal{M}_k \\ store^\# &\left(\mathcal{M}^k_{i=1}(o_i, d_i), p, a\right) \stackrel{\text{def}}{=} \mathcal{M}^k_{i=1}(o_i, d^*_i) \end{aligned}$$

Das Offset und der zu speichernde Wert können unabhängig von der Struktur des Speicherstücks verwendet werden. Es darf zu keinem Unterschreiten, bzw. Überschreiten, was durch ein zu kleines bzw. zu großes Offset ausgelöst wird, kommen und der Zugriff auf ein Speicherloch ist undefiniert. Die Implementierung dieser Funktionen ist maschinenabhängig und wird in Kapitel 4 erläutert.

Wir verwenden auch verschiedene Arten von Zugriffsfunktionen, die das Speicherstück nicht verändern. Auf diese maschinenabhängige Funktionen werden wir in Kapitel 4 genauer eingehen.

$$
\begin{aligned}
load_{\mathbb{Z}_m} &: \mathcal{M}_k \times \mathbb{Z} \longrightarrow I_n^*(\mathbb{Z}_m) \\
load_{\mathbb{R}_{m'}} &: \mathcal{M}_k \times \mathbb{Z} \longrightarrow I_n^*(\mathbb{R}_{m'})
\end{aligned}
$$

wobei

$$
\begin{aligned}
load_{\mathbb{Z}_m}\left(\mathcal{M}_{i=1}^k(o_i, d_i)\right) &\stackrel{\text{def}}{=} \left(\left[a_i, b_i\right]\right)_{i=1}^n \\
load_{\mathbb{R}_{m'}}\left(\mathcal{M}_{i=1}^k(o_i, d_i)\right) &\stackrel{\text{def}}{=} \left(\left[a_i, b_i\right]\right)_{i=1}^n
\end{aligned}
$$

Auch hier ist der Zugriff auf Speicherlücken und das Unterschreiten bzw. Überschreiten nicht definiert.

Definition 3.2.34 *Mit einem Speicherzugriffsfehler bezeichnen wir den Zugriff mit $x \in I_n^*(\mathbb{Z}_m) \cup I_n^*(\mathbb{R}_{m'})$ auf ein Speicherstück m mit einem negativen Offset (Underrun) oder zu großem Offset (Overrun) oder den Zugriff auf ein Speicherloch. Das Offset o ist zu groß, wenn $o + |x| \geq |m|$ wobei $|x|$ der Speicherbedarf von x ist.*

Eine spezielle Form des Speicherstücks ist der Stack (Stapel). Dieser wächst nicht nach oben, sondern nach unten, d.h. wir benötigen negative Offsets um auf eine Stack zuzugreifen.

Definition 3.2.35 *Mit einem Stackzugriffsfehler bezeichnen wir den Zugriff mit $x \in I_n^*(\mathbb{Z}_m) \cup I_n^*(\mathbb{R}_{m'})$ auf einen Stack m mit einem zu großem Offset (Overrun) oder zu kleinem Offset (Underrun) oder den Zugriff auf ein Speicherloch. Das Offset o ist zu klein, wenn $o < -|m|$ und zu groß, wenn $o + |x| \geq 0$ wobei $|x|$ der Speicherbedarf von x ist.*

Für den Zugriff auf Speicherstücke benötigen wir Zeiger. Ein Zeiger besteht aus einem Paar $(\mathcal{M}_k, \mathbb{Z}_m)$, einem symbolischen Namen, der für jedes Speicherstück eindeutig ist und einem Offset. Die Approximation geschieht auch hier komponentenweise.

Definition 3.2.36 *Wir bezeichnen mit $\left((m_i)_{i=1}^l, I_n^*(\mathbb{Z}_m)\right)$ mit $m_i \in M_k \cup \{\bot, \top\}$ einen Zeiger auf Speicherstücke aus $(m_i)_{i=1}^l$ mit Offsets aus $I_n^*(\mathbb{Z}_m)$. $\mathcal{Z}$ bezeichnet die Menge aller Zeiger.*

Mit Hilfe der Galoisverknüpfung ergibt sich zwischen der konkreten und der abstrakten Semantik die nachfolgende Beziehung.

$$
\begin{array}{ccc}
P\left(\mathcal{M}_k, P(\mathbb{Z}_m)\right) & \xrightarrow{post} & P\left(\mathcal{M}_k, P(\mathbb{Z}_m)\right) \\
\alpha \downarrow & & \uparrow \underset{\gamma}{\subseteq} \\
\mathcal{Z} & \xrightarrow{post^\#} & \mathcal{Z}
\end{array}
$$

Für α und γ verwenden wir auch hier die bereits eingeführten α und γ auf $I_n^*(\mathbb{Z}_m)$.

$$\alpha \ : \ P\left(\mathcal{M}_k, P(\mathbb{Z}_m)\right) \longrightarrow \mathcal{Z}$$
$$\gamma \ : \ \mathcal{Z} \longrightarrow P\left(\mathcal{M}_k, P(\mathbb{Z}_m)\right)$$

wobei

$$\begin{aligned} \alpha \ (M, O) &\stackrel{\text{def}}{=} \big(\alpha(M), \alpha(O)\big) \\ \gamma \ \Big(\Big((m_i)_{i=1}^{l}, o\Big)\Big) &\stackrel{\text{def}}{=} \Big(\gamma\big((m_i)_{i=1}^{l}\big), \gamma(o)\Big) \end{aligned}$$

mit

$$\alpha\left(M\right) \stackrel{\text{def}}{=} \begin{cases} \bot, \text{ falls } \exists m \in M \text{ mit } m = \bot \\ \top, \text{ falls } |M| > l \\ (m_i)_{i=1}^{l}, \text{ mit } m_i \in M \text{ sonst} \end{cases}$$

$$\gamma\ \Big((m_i)_{i=1}^{l}\Big) \stackrel{\text{def}}{=} \begin{cases} \emptyset, \text{ falls } \exists i \text{ mit } m_i = \bot \\ \mathcal{M}_k, \text{ falls } \exists i \text{ mit } m_i = \top \\ \{m_1, \ldots, m_l\}, \text{ sonst} \end{cases}$$

Wir erlauben auf Zeigern nur solche Operationen, die das Offset verändern.

$$add^{\#} \ : \ \mathcal{Z} \times I_n^*(\mathbb{Z}_m) \longrightarrow \mathcal{Z}$$
$$sub^{\#} \ : \ \mathcal{Z} \times I_n^*(\mathbb{Z}_m) \longrightarrow \mathcal{Z}$$

wobei

$$\begin{aligned} add^{\#}\ \Big(\Big((m_i)_{i=1}^{l}, o\Big), a\Big) &\stackrel{\text{def}}{=} \Big((m_i)_{i=1}^{l}, add^{\#}(o, a)\Big) \\ sub^{\#}\ \Big(\Big((m_i)_{i=1}^{l}, o\Big), a\Big) &\stackrel{\text{def}}{=} \Big((m_i)_{i=1}^{l}, sub^{\#}(o, a)\Big) \end{aligned}$$

Eine weitere, spezielle Operation, berechnet die Differenz zweier Zeiger. Diese verändert die Zeiger nicht und ist nur definiert für den Fall, dass die Zeiger auf dasselbe Speicherstück zeigen. Wir erhalten als Ergebnis die Differenz der Offsets.

$$sub \ : \ \mathcal{Z} \times \mathcal{Z} \longrightarrow I_n^*(\mathbb{Z}_m)$$
$$sub\ \Big(\Big((m_i)_{i=1}^{l}, o\Big), \Big((p_i)_{i=1}^{k}, q\Big)\Big)$$
$$\stackrel{\text{def}}{=} \begin{cases} sub^{\#}(o, q), \text{ falls } l = 1, k = 1 \text{ und } m_1 = p_1 \\ \bot, \text{ sonst} \end{cases}$$

Eine Multiplikation und Division von Zeigern macht keinen Sinn und ist hier deshalb nicht definiert. Wir erweitern nun die Speicherstücke derart, dass wir Zeiger als Speicherkomponenten verwenden können. Wir passen die Definition 3.2.33 entsprechend an.

Definition 3.2.37 *Wir bezeichnen mit $\mathcal{M}_{i=1}^{k}(o_i, d_i)$ ein Speicherstück mit k Komponenten, wobei $o_i \in \mathbb{N}$ und $d_i \in I_n^*(\mathbb{Z}_m) \cup I_n^*(\mathbb{R}_m) \cup \mathcal{Z}$.*

Mit $\mathcal{M}_k$ bezeichnen wir die Menge aller Speicherstücke mit maximal k Komponenten.

Die für Speicherstücke eingeführten Funktionen α und γ müssen entsprechend angepasst werden. Die modifizierte Operation $store^{\#}$ ist folgendermaßen definiert:

$$\begin{aligned} store^{\#} &: \mathcal{M}_k \times \mathbb{N} \times (I_n^*(\mathbb{Z}_m) \cup I_n^*(\mathbb{R}_{m'}) \cup \mathcal{Z}) \longrightarrow \mathcal{M}_k \\ store^{\#} &\left(\mathcal{M}_{i=1}^{k}(o_i, d_i), p, a\right) \stackrel{\text{def}}{=} \mathcal{M}_{i=1}^{k}(o_i, d_i^*) \end{aligned}$$

Eine entsprechende $load_{\mathcal{Z}}$ Funktion existiert ebenfalls.

$$load_{\mathcal{Z}} : \mathcal{M}_k \times \mathbb{N} \longrightarrow \mathcal{Z}$$

wobei

$$load_{\mathcal{Z}} \left(\mathcal{M}_{i=1}^{k}(o_i, d_i)\right) \stackrel{\text{def}}{=} \left((m_i)_{i=1}^{l}, o\right)$$

Wir werden auf diese Funktion in Kapitel 4 genauer eingehen.

In diesem Kapitel haben wir eine Einführung in die Theorie der abstrakten Interpretation gegeben. Wir haben die Idee der Approximation und den für XRTL verwendeten abstrakten und konkreten Domain vorgestellt. Wir haben endliche Teilmengen der ganzen Zahlen $\mathbb{Z}_m$ und reellen Zahlen $\mathbb{R}_{m'}$ und deren Approximation mit Listen von gültigen Intervallen eingeführt. Die für die XRTL Analyse notwendigen Operationen wurden definiert. Anschließend haben wir Speicherstücke mit den dazugehörigen Zugriffsoperationen und Zeiger mit entsprechender Zeigerarithmetik vorgestellt.

Kapitel 4

Implementierung

In diesem Kapitel gehen wir genauer auf die, die Implementierung betreffenden, technischen Details ein. Wir geben die konkreten Instanzen der Grunddatentypen, die endliche Teilmenge der ganzen Zahlen $\mathbb{Z}_m$ und der reellen Zahlen $\mathbb{R}_{m'}$ an. Die entsprechenden, in der Praxis verwendeten Mengen bezeichnen wir mit int_m bzw. $float_{m'}$ und bezeichnen diese als Datentypen. Wir führen die Bit- und Bytedarstellung von Zahlen aus int_m und $float_{m'}$ ein und zeigen, wie diese berechnet werden. Wir benötigen die Bytedarstellung für die Implementierung der Operationen auf Speicherstücken. Wir schränken den Begriff des gültigen Intervalls weiter ein, um Problemen mit der Bytedarstellung zu begegnen. Wir zeigen, wie wir für Listen von Intervallen über int_m oder $float_{m'}$ Listen von gültigen Intervallen des jeweiligen Datentyps nach der eingeschränkten Definition erzeugen und wie die in Kapitel 3 eingeführten Algorithmen `split` und `merge` angepasst werden müssen. Wir geben die Funktionen rep und rep^{-1} an, die für eine Listen von gültigen Intervallen, die die Bytedarstellung berechnet, bzw. aus einer gegebenen Bytedarstellung eine Liste von gültigen Intervallen erzeugt. Diese Funktionen verwenden wir für den Zugriff Speicherstücke. Wir unterscheiden zwischen Zeiger auf Speicherstücke und Zeiger auf Funktionen. Für Funktionszeiger erlauben wir keine Zeigerarithmetik.

4.1 Datentypen und ihre Darstellung

Definition 4.1.1 *Wir bezeichnen mit int_m den Datentyp mit Bytegröße m der die kleinsten nicht negativen Vertreter $a^+ \in \mathbb{Z}$ der Restklasse $\bar{a} \in \mathbb{Z}_{m'}$ mit $m' = 2^{8m}$ repräsentiert.*

Ein Wert vom Datentyp int_m besteht aus $8 \cdot m$ Bits, da ein Byte acht Bit umfasst. Ein Byte ist die kleinste Einheit, auf die wir mit XRTL Sprachkonstrukten direkt zugreifen können.

Bemerkung 4.1.2 Unser Analysetool xGCC unterstützt momentan die ganzzahligen Datentypen int_1, int_2, int_4 und int_8. Das Design des Tools ist so ausgelegt, dass eine Erweiterung einfach möglich ist.

Definition 4.1.3 *Wir bezeichnen, für ein $x \in int_m$, mit x^- den betragsmäßig kleinsten Vertreter der Restklasse $\bar{a} \in \mathbb{Z}_{m'}$ für den $x \in \bar{a}$ gilt.*

Beispiel 4.1.4 Gegeben seien int_1 und $\mathbb{Z}_{256}$. Für $x = 254$ ist $x^- \stackrel{\text{def}}{=} (\bar{x})^- = \left(\overline{254}\right)^- = -2$.

Für die bei ganzen Zahlen auftretende Problematik der Auswahl der Vertreter der Restklasse finden wir in den ganzzahligen Datentypen eine Entsprechung. Wir betrachten die Werte $x \in int_m$ als kleinste nicht negative Vertreter von Restklassen aus $\mathbb{Z}_{m'}$. Wir sprechen hier von einer vorzeichenlosen Betrachtungsweise der Werte aus int_m. Wenn wir anstatt x die dazugehörigen Werte x^- betrachten, so sprechen wir von einer vorzeichenbehafteten Betrachtungsweise der Werte aus int_m.

Bemerkung 4.1.5 Für ein $x \in int_m$ und $x \in \bar{a}$ mit $\bar{a} \in \mathbb{Z}_{m'}$ ist also $x = a^+$ und $x^- = a^-$.

Betrachten wir die Zahl $x \in int_m$ als vorzeichenlose Zahl so können wir deren Darstellung als Bitvektor folgendermaßen definieren.

$$\begin{aligned} rep_n &: int_n \longrightarrow \{0,1\}^{8n} \\ rep_n(x) &\stackrel{\text{def}}{=} x_0, \ldots, x_{8n-1} \end{aligned}$$

wobei

$$x_i = \left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2$$

Wir bezeichnen x_0 als niedrigstwertiges und x_{8n-1} als höchstwertiges Bit. Eine gegebene Bitdarstellung $x_0, \ldots, x_{8n-1}$ können wir als vorzeichenlose und als vorzeichenbehaftete Zahl interpretieren. Die Zahl x und das dazugehörige x^- haben dieselbe Bitdarstellung.

$x_0 \ldots x_{8n-1}$	x *vorzeichenlos*	x^- *vorzeichenbehaftet*
$00\ldots00$	0	0
$10\ldots00$	1	1
$\vdots$	$\vdots$	$\vdots$
$11\ldots10$	$2^{8n-1}-1$	$2^{8n-1}-1$
$00\ldots01$	2^{8n-1}	-2^{8n-1}
$10\ldots01$	$2^{8n-1}+1$	$-2^{8n-1}+1$
$\vdots$	$\vdots$	$\vdots$
$01\ldots11$	$2^{8n}-2$	-2
$11\ldots11$	$2^{8n}-1$	-1

Wir verwenden aus praktischen Gründen die einfacher zu handhabende Bytedarstellung anstelle der rechenintensiveren Bitdarstellung, da wir direkt auf ein Byte eines Wertes aus int_m zugreifen können, ohne die einzelnen Bits vorher zu berechnen. Wir können aber aus der Bitdarstellung $x_0, \ldots, x_{8m-1}$ von x sehr einfach die dazugehörige Bytedarstellung berechnen. Wir verwenden $int_1 = \{0, \ldots, 255\}$ für die Bytedarstellung.

$$
\begin{aligned}
rep_m &: int_m \longrightarrow \left(int_1\right)^m \\
rep_m\,(x) &\stackrel{\text{def}}{=} b_0, \ldots, b_{m-1}
\end{aligned}
$$

mit

$$
b_i = \sum_{j=8i}^{8i+7} x_i 2^{j-8i}
$$

Unter Verwendung der Funktion rep für den Datentyp int_m können wir die Bytedarstellung von gültigen Intervallen nach Definition 3.2.6 für $I^*(int_m)$ beschreiben.

$$
\begin{aligned}
rep_m &: I^*(int_m) \longrightarrow \left(I(int_1)\right)^m \\
rep_m\,\left(\left[a,b\right]\right) &\stackrel{\text{def}}{=} \left[c_0,d_0\right], \ldots, \left[c_{m-1},d_{m-1}\right]
\end{aligned}
$$

mit

$$
\begin{aligned}
rep_m\,(a) &\stackrel{\text{def}}{=} a_0, \ldots, a_{m-1} \\
rep_m\,(b) &\stackrel{\text{def}}{=} b_0, \ldots, b_{m-1}
\end{aligned}
$$

wobei

$$
\begin{aligned}
c_i &\stackrel{\text{def}}{=} \begin{cases} 0, \text{ falls } b-a \geq 2^{8(i+1)} - 2^{8i} \\ a_i, \text{ sonst} \end{cases} \\
d_i &\stackrel{\text{def}}{=} \begin{cases} 255, \text{ falls } b-a \geq 2^{8(i+1)} - 2^{8i} \\ b_i, \text{ sonst} \end{cases}
\end{aligned}
$$

Es kann hierbei vorkommen, dass ein i existiert, für das $d_i < c_i$ gilt, obwohl $\left[a,b\right]$ ein gültiges Intervall nach Definition 3.2.6 ist und somit $a \leq b$ gilt. Die Intervalldefinition ist für $\left[c_i,d_i\right]$ verletzt.

Beispiel 4.1.6 Für ein Intervall $\left[10,256\right] \in I^*(int_2)$ das gültig nach Definition 3.2.6 ist, sieht die Bytedarstellung folgendermaßen aus: $\left[10,0\right], \left[0,1\right]$. Es gilt $c_1 \leq d_1$ aber auch $d_0 < c_0$.

Dieser Umstand zwingt uns den Begriff des gültigen Intervalls zu überdenken und seine Definition weiter einzuschränken.

Definition 4.1.7 *Wir bezeichnen ein Intervall* $\left[a,b\right]$ *mit* $a,b \in int_m$ *als gültig, wenn es gültig nach Definition 3.2.6 ist und für* $rep\,\left(\left[a,b\right]\right) \stackrel{def}{=} \left[c_0,d_0\right], \ldots, \left[c_{m-1},d_{m-1}\right]$ *gilt, dass* $c_i \leq d_i$ *für alle* $0 \leq i < m$.

Diese zusätzliche Bedingung führt zu Vereinfachungen in der Implementierung, da einige Spezialfälle beim Zugriff auf die Bytedarstellung wegfallen. Wir ersparen uns die entsprechenden Überprüfungen, da diese nun Teil der Bedingung sind. Wir können jedes beliebige Intervall in gültige Intervalle nach Definition 4.1.7 aufteilen.

Lemma 4.1.8 *Ein Intervall* $\left[a,b\right]$ *mit* $a,b \in int_m$ *ist gültig oder kann in maximal vier gültige Intervalle aufgeteilt werden, die alle Elemente aus* $\left[a,b\right]$ *enthalten.*

Beweis: Konstruktion: Sei $\left[a,b\right]$ gültig nach Definition 3.2.6, aber nicht nach Definition 4.1.7. Sei $rep\left(\left[a,b\right]\right) \stackrel{\text{def}}{=} \left[c_0,d_0\right],\ldots,\left[c_{m-1},d_{m-1}\right]$.

$$
\begin{aligned}
p &\stackrel{\text{def}}{=} max\{i \mid c_i > d_i\} \\
a' &\stackrel{\text{def}}{=} \sum_{j=0}^{i} 255 \cdot 2^{8j} + \sum_{j=i+1}^{m-1} c_j \cdot 2^{8j} \\
b' &\stackrel{\text{def}}{=} \sum_{j=0}^{i} 0 \cdot 2^{8j} + (c_{i+1}+1) \cdot 2^{(i+1)8} + \sum_{j=i+2}^{m-1} d_j \cdot 2^{8j}
\end{aligned}
$$

Gemäß Konstruktion von a' und b' gilt $\left[a,a'\right]$ und $\left[b',b\right]$ gültig nach Definition 4.1.7. Für $j \leq i$ gilt für die Bytedarstellung von a,a',b',b offensichtlich $a_j \leq a'_j$ und $b'_j \leq b_j$. Diese Bedingungen gelten auch für $j > i$, da hier $c_j \leq d_j$ und $c \leq d$ gilt. Da gemäß Konstruktion auch $b' = a' + 1$ gilt enthalten die Intervalle $\left[a,a'\right]$ uns $\left[b',b\right]$ alle Werte aus $\left[a,b\right]$. ■

Die neue, eingeschränkte Definition von gültigen Intervallen wirkt sich nur auf die in Kapitel 3 eingeführten Algorithmen `merge` und `split` aus. Diese müssen gemäß Lemma 4.1.8 und dazugehörigem Beweis überarbeitet werden.

Eine zur Funktion rep inverse Funktion rep^{-1} existiert ebenfalls und erzeugt aus einer gegebenen Bytedarstellung ein Intervall, welches nicht notwendigerweise gültig sein muss.

$$
\begin{aligned}
rep_m^{-1} &: \left(I^*(int_1)\right)^m \longrightarrow I^*(int_m) \\
rep_m^{-1} &\left(\left[a_0,b_0\right],\ldots,\left[a_{m-1},b_{m-1}\right]\right) \stackrel{\text{def}}{=} \left[a,b\right]
\end{aligned}
$$

wobei

$$
\begin{aligned}
a &\stackrel{\text{def}}{=} \sum_{i=1}^{m-1} a_i \cdot 2^{8i} \\
b &\stackrel{\text{def}}{=} \sum_{i=1}^{m-1} b_i \cdot 2^{8i}
\end{aligned}
$$

Wenn für alle $0 \leq i < m$, $a_i \leq b_i$ gilt, dann ist auch $a \leq b$. Somit beschreibt $\left[a, b\right]$ ein Intervall. Die Umkehrung gilt allerdings nicht.

Wir können die Funktionen rep und rep^{-1} auf Listen von gültigen Intervallen nach Definition 4.1.7 erweitern. Wir verwenden hierzu den modifizieren `merge` Algorithmus.

$$\begin{aligned} rep_m &: I_n^*(int_m) \longrightarrow \left(\left(I^*(int_1)\right)^m\right)_{i=1}^n \\ rep_m^{-1} &: \left(\left(I^*(int_1)\right)^m\right)_{i=1}^n \longrightarrow I_n^*(int_m) \end{aligned}$$

wobei

$$\begin{aligned} rep_m\ &\left(\left(\left[a_i, b_i\right]\right)_{i=1}^n\right) \stackrel{\text{def}}{=} \left(rep_m\left(\left[a_i, b_i\right]\right)\right)_{i=1}^n \\ rep_m^{-1}\ &\left(\left[a_0, b_0\right] \ldots, \left[a_{m-1}, b_{m-1}\right]\right)_{i=1}^n \stackrel{\text{def}}{=} \\ &\texttt{merge}\Big(\Big(rep_m^{-1}\left(\left[a_{i_0}, b_{i_0}\right], \ldots, \left[a_{i_{m-1}}, b_{i_{m-1}}\right]\right)\Big)_{i=1}^n\Big) \end{aligned}$$

Es gilt:

$$\left(\left[a_i, b_i\right]\right)_{i=1}^n \subseteq rep_n^{-1}\Big(rep_n(\left(\left[a_i, b_i\right]\right)_{i=1}^n)\Big)$$

Wir führen jetzt, analog zu den ganzen Zahlen $\mathbb{Z}_m$, die in der Implementierung verwendeten Instanzen der reellen Zahlen $\mathbb{R}_m$ ein.

Definition 4.1.9 *Wir bezeichnen mit $float_m$ den Datentyp der Bytegröße m der eine endliche Teilmenge der reellen Zahlen $\mathbb{R}_{m'}$ repräsentiert.*

Das zu $float_m$ entsprechende $\mathbb{R}_{m'}$ ist durch die Standards IEEE 754/854 [20, 21] definiert.

Bemerkung 4.1.10 Unser Analysetool xGCC unterstützt die Datentypen $float_4$, $float_8$, $float_{12}$ und $float_{16}$. Die Datentypen $float_{12}$ und $float_{16}$ treten in der Praxis nie gemeinsam auf.

Wir erweitern die Definition von rep auf den Datentyp $float_m$ um die Bitdarstellung gemäß IEEE 754 [20, 35] zu erhalten.

$$\begin{aligned} rep_m &: float_m \longrightarrow \{0,1\}^{8m} \\ rep_m\,(a) &\stackrel{\text{def}}{=} a_0, \ldots, a_{8m-1} \end{aligned}$$

Die Bitdarstellung eines Wertes aus $float_m$ ist aufgeteilt in Bits für das Vorzeichen, die Mantisse, den Exponenten und ungenutzte Bits.

	$Vorzeichen\ (s)$	$Mantisse\ (a)$	$Exponent\ (e)$	$ungenutzt\ (u)$
$float_4$	1	23	8	0
$float_8$	1	52	11	0
$float_{12}$	1	64	15	16
$float_{16}$	1	112	15	0

Für die Bitdarstellung $f = f_0, \ldots f_{8m-1}$ eines Wertes aus $float_m$ sieht die Aufteilung von f in Vorzeichen, Mantisse, Exponent und ungenutzte Bits folgendermaßen aus. Die Mantisse besteht aus $|a|$ Bits, der Exponent aus $|e|$ Bits und $|u|$ ist die Anzahl der ungenutzten Bits.

$$f = f_0, \ldots, f_{m-1} = a_0, \ldots, a_{|a|-1}\ u_0, \ldots, u_{|u|-1}\ e_0, \ldots, e_{|e|-1}\ sign$$

Definition 4.1.11 *Aus einer Bitrepräsentation mit Vorzeichen $sign$, der Mantisse $mantissa = a_0, \ldots, a_{|a|-1}$ und dem Exponenten $exponent = e_0, \ldots, e_{|e|-1}$ können wir den dazugehörigen Fließkommawert f folgendermaßen berechnen, wobei $bias = 2^{(|e|-2)} - 1$:*

$$\begin{aligned}
e = 0 \ldots 0,\ a = 0 \ldots 0 : &\quad f = (-1)^{sign} \cdot 0 \\
e = 0 \ldots 0,\ a \neq 0 \ldots 0 : &\quad f = (-1)^{sign} \cdot 2^{-bias+1} \cdot (0.a) \\
e \neq 0 \ldots 0,\ e \neq 1 \ldots 1 : &\quad f = (-1)^{sign} \cdot 2^{e-bias} \cdot (1.a) \\
e = 1 \ldots 1,\ a = 0 \ldots 0 : &\quad f = (-1)^{sign} \cdot \infty \\
e = 1 \ldots 1,\ a \neq 0 \ldots 0 : &\quad f = \textit{NaN (Not a Number)}
\end{aligned}$$

Bemerkung 4.1.12 Es gibt spezielle Bitdarstellungen für $+\infty$ und $-\infty$ sowie NaN. Außerdem existieren auch unterschiedliche Bitdarstellungen für $+0.0$ und -0.0. Diese verhalten sich in arithmetischen Operationen jedoch gleich.

Bemerkung 4.1.13 Der Datentyp $float_{12}$ weicht gemäß [35] in seinem Aufbau leicht von den anderen Datentypen ab. Das führende Bit in der Mantisse a muss explizit gesetzt werden.

Für die Definition der Repräsentation eines Intervalls über $float_m$ verwenden wir zwei, die Behandlung der Mantisse eines Fließkommawertes vereinfachende, Funktionen rep_m^{inf} und rep_m^{sup} mit

$$\begin{aligned}
rep_m^{inf} &: float_m \longrightarrow \{0,1\}^{8m} \\
rep_m^{inf}\ (f) &\stackrel{\text{def}}{=} f_0, \ldots, f_{8m-1}
\end{aligned}$$

wobei $f_0, \ldots, f_{8m-1}$ Bitdarstellung von f mit Mantisse $a_0^f, \ldots, a_{8m-1}^f = 0, \ldots, 0$, und

$$\begin{aligned} rep_m^{sup} &: \mathit{float}_m \longrightarrow \{0,1\}^{8m} \\ rep_m^{sup}(f) &\stackrel{\text{def}}{=} f_0, \ldots, f_{8m-1} \end{aligned}$$

wobei $f_0, \ldots, f_{8m-1}$ Bitdarstellung von f mit Mantisse $a_0^f, \ldots, a_{8m-1}^f = 1, \ldots, 1$.

Es gilt für ein $f \in \mathit{float}_m$:

$$rep_m^{-1}(rep_m^{inf}(f)) \leq f \leq rep_m^{-1}(rep_m^{sup}(f))$$

Wir passen die Funktionen rep_m und rep_m^{-1} an die von uns verwendete Bytedarstellung an.

$$\begin{aligned} rep_m &: \mathit{float}_m \longrightarrow \big(\mathit{int}_1\big)^m \\ rep_m^{-1} &: \big(\mathit{int}_1\big)^m \longrightarrow \mathit{float}_m \end{aligned}$$

mit

$$\begin{aligned} rep_m(f) &\stackrel{\text{def}}{=} b_0, \ldots, b_{m-1} \\ rep_m^{-1}(b_0, \ldots, b_{m-1}) &\stackrel{\text{def}}{=} f \end{aligned}$$

wobei

$$b_i = \sum_{j=8i}^{8i+7} a_i 2^{j-8i}$$

und $a_0, \ldots, a_{8m-1}$ die Bitdarstellung von f gemäß IEEE 754 [20, 35] ist.

Die Funktionen rep_m^{inf} und rep_m^{sup} werden analog modifiziert. Wir müssen nur beachten, dass wir eine, in der Mantisse veränderte Darstellung verwenden.
Wir erweitern die Funktionen rep_m und die inverse rep_m^{-1} auf Intervalle aus $I^*(\mathit{float}_m)$.

$$\begin{aligned} rep_m &: I^*(\mathit{float}_m) \longrightarrow \big(I(\mathit{int}_1)\big)^m \\ rep_m([a,b]) &\stackrel{\text{def}}{=} ([c_0,d_0], \ldots, [c_{m-1},d_{m-1}]) \end{aligned}$$

wobei $rep_m^{inf}(a) = c_0, \ldots, c_{m-1}$ und $rep_m^{sup}(b) = d_0, \ldots, d_{m-1}$ und

$$\begin{aligned} rep_m^{-1} &: \big(I(\mathit{int}_1)\big)^m \longrightarrow I(\mathit{float}_m) \\ rep_m^{-1}([a_0,b_0], \ldots, [a_{m-1},b_{m-1}]) &\stackrel{\text{def}}{=} [c,d] \end{aligned}$$

wobei

$$\begin{aligned} rep_m(c) &= rep_m^{inf}\,(rep(a_0,\ldots,a_{m-1})) \\ rep_m(d) &= rep_m^{sup}\,(rep(a_0,\ldots,a_{m-1})) \end{aligned}$$

Wir rechnen auch für die inverse Funktion mit einer vereinfachten Mantisse um die konkrete Implementierung dieser Funktion zu erleichtern.

Wir schränken die Definition 3.2.25 für gültige Intervalle über $float_m$ ein, da es wie für ganze Zahlen, vorkommen kann, dass für $a, b \in float_m$ mit $a \leq b$ und $rep_m(a) = c_0, \ldots, c_{m-1}$ sowie $rep_m(b) = d_0, \ldots, d_{m-1}$ ein i geben kann mit $c_i > d_i$. Die Intervalldefinition für $\left[c_i, d_i\right]$ ist damit verletzt. Dieses kann allerdings, aufgrund der Konstruktion von rep_m und rep_m^{-1}, nur durch unterschiedliche Exponenten oder Vorzeichen ausgelöst werden.

Definition 4.1.14 *Wir bezeichnen ein Intervall $\left[a, b\right]$ mit $a, b \in float_m$ gültig, wenn es gültig nach Definition 3.2.25 ist und für $rep\left(\left[a, b\right]\right) \stackrel{def}{=} \left[c_0, d_0\right], \ldots, \left[c_{m-1}, d_{m-1}\right]$ gilt, dass $c_i \leq d_i$ für alle $0 \leq i < m$.*

Zwischen Intervallen und gültigen Intervallen nach Definition 4.1.14 existiert auch hier eine Verbindung.

Lemma 4.1.15 *Ein Intervall $\left[a, b\right]$ mit $a, b \in float_m$ ist gültig oder kann in maximal vier gültige Intervalle aufgeteilt werden, die alle Elemente aus $\left[a, b\right]$ enthalten.*

Beweis: Konstruktion: Sei $\left[a, b\right]$ mit $rep_m(\left[a, b\right]) \stackrel{\text{def}}{=} \left[c_0, d_0\right], \ldots, \left[c_{m-1}, d_{m-1}\right]$ gültig nach Definition 3.2.25, aber nicht nach Definition 4.1.14.

Idee: Wir finden zwei benachbarte Elemente a_u und b_l, mit $a_u < b_l$, die sich im Exponenten um eins unterscheiden und deren Mantisse einmal $1, \ldots, 1$ und im anderen Fall $0, \ldots, 0$ ist. Es existiert offensichtlich kein weiteres darstellbares $f \in float_m$ mit $a_u < f < b_l$. Wir wählen a_u und b_l derart aus, dass $\left[a, a_u\right]$ und $\left[b_l, b\right]$ gültige Intervalle gemäß Definition 4.1.14 sind. Für weitergehende Details verweisen wir auf die Implementierung. ■

Wir passen die Algorithmen `merge` und `split` aus Kapitel 3 an die neue, eingeschränkte Definition von gültigen Intervallen an. Diese müssen gemäß Lemma 4.1.15 und dazugehörigem Beweis überarbeitet werden.

Wir erweitern nun die Funktionen rep und rep^{-1} auf Listen von gültigen Intervallen nach Definition 4.1.14. Wir verwenden hierzu den modifizieren `merge` Algorithmus.

$$\begin{aligned} rep_m &: I_n^*(float_m) \longrightarrow \left(\left(I^*(int_1)\right)^m\right)_{i=1}^n \\ rep_m^{-1} &: \left(\left(I^*(int_1)\right)^m\right)_{i=1}^n \longrightarrow I_n^*(float_m) \end{aligned}$$

wobei

$$
\begin{aligned}
rep_m\ \left(\left(\left[a_i, b_i\right]\right)_{i=1}^{n}\right) &\stackrel{\text{def}}{=} \left(rep_m\left(\left[a_i, b_i\right]\right)\right)_{i=1}^{n} \\
rep_m^{-1}\ \left(\left(\left[a_{i_0}, b_{i_0}\right], \ldots, \left[a_{i_{m-1}}, b_{i_{m-1}}\right]\right)_{i=1}^{n}\right) &\stackrel{\text{def}}{=} \\
&\texttt{merge}\left(\left(rep_m^{-1}\left(\left[a_{i_0}, b_{i_0}\right], \ldots, \left[a_{i_{m-1}}, b_{i_{m-1}}\right]\right)\right)_{i=1}^{n}\right)
\end{aligned}
$$

Es gilt:

$$
\left(\left[a_i, b_i\right]\right)_{i=0}^{m} \subseteq rep_n^{-1}\left(rep_n\left(\left(\left[a_i, b_i\right]\right)_{i=0}^{m}\right)\right)
$$

Die in Kapitel 3 eingeführten arithmetischen Operationen für ganze und reelle, sowie die Bitoperationen für ganze Zahlen können unverändert übernommen werden, da gültige Intervalle nach der neuen und eingeschränkten Definition auch gültig nach der alten sind.

Die Funktionen rep und rep^{-1} für Zeiger zu implementieren ist schwierig, da für uns ein Zeiger aus einem Paar von einer Liste von symbolischen Namen von Speicherstücken und einem Offsetintervall besteht. Da die Anfangsadressen der einzelnen Speicherstücke erst zu Ausführungszeit des Programms bestimmt werden, können wir hier keine genau Bit- bzw. Bytedarstellung angeben. Wir vereinfachen rep und rep^{-1} deshalb entsprechend:

$$
\begin{aligned}
rep\ &:\ Z \longrightarrow \left(\left(I^*(int_1)\right)^{|Z|}\right)_{i=1}^{n} \\
rep^{-1}\ &:\ \left(\left(I^*(int_1)\right)^{|Z|}\right)_{i=1}^{n} \longrightarrow Z
\end{aligned}
$$

wobei

$$
\begin{aligned}
rep\ \left(\left((m_i)_{i=1}^{l}, o\right)\right) &\stackrel{\text{def}}{=} \left\{\left[0, 255\right], \ldots, \left[0, 255\right]\right\} \setminus \left\{\left[0, 0\right], \ldots, \left[0, 0\right]\right\} \\
rep^{-1}\ \left(\left(\left[a_{i_0}, b_{i_0}\right], \ldots, \left[a_{i_{m-1}}, b_{i_{m-1}}\right]\right)_{i=1}^{n}\right) &\stackrel{\text{def}}{=} \begin{cases} \bot, \text{ falls } a_{i_j} = b_{i_j} = 0,\ \forall i, j \\ \top, \text{ sonst} \end{cases}
\end{aligned}
$$

Durch eine Zuordnung von echten Adressen an symbolische Namen könnten wir diese Funktionen präzisieren, würden aber auch die Gültigkeit unserer Aussagen einschränken.

4.2 Speicherstücke

Die für $I_n^*(int_m)$, $I_n^*(float_m)$ und Zeiger Z eingeführten Funktionen rep und rep^{-1} sind ein wichtiger Baustein für die Implementierung von Speicherstücken und deren Zugriffsfunktionen. Wir können die Bytedarstellung eines Speicherstücks Hilfe des nachfolgenden Algorithmus `memory_rep` berechnen.

Algorithmus 4.2.1

Algorithmus `memory_rep`

EINGABE: $\mathcal{M}_{i=1}^{n}(o_i, d_i) \in \mathcal{M}_k$

AUSGABE: $\Big(\big[a_{i_0}, b_{i_0}\big], \ldots, \big[a_{i_{|\mathcal{M}_k|-1}}, b_{i_{|\mathcal{M}_k|-1}}\big]\Big)_{i=1}^{n}$

(1) $p = 1$
(2) $i = 0$
(3) **while** $(i < |\mathcal{M}_k|)$ **do**
(4) **while** $(i < o_p)$ **do**
(5) **for** $(j := 1; j \leq n; j := j + 1)$ **do**
(6) $a_{i_j} := \perp$
(7) $b_{i_j} := \perp$
(8) **od**
(9) $i := i + 1$
(10) **od**
(11) $\Big(\big[c_{i_0}, d_{i_0}\big], \ldots, \big[c_{i_{|d_p|-1}}, d_{i_{|d_p|-1}}\big]\Big)_{i=1}^{l} := rep_{|d_p|}(d_p)$
(12) **while** $(i < o_p + |d_p|)$ **do**
(13) $k := i - o_p$
(14) **for** $(j := 1; j \leq l; j := j + 1)$ **do**
(15) $a_{i_j} := c_{k_j}$
(16) $b_{i_j} := d_{k_j}$
(17) **od**
(18) **for** $(j := l + 1; j \leq n; j := j + 1)$ **do**
(19) $a_{i_j} := c_{k_l}$
(20) $b_{i_j} := d_{k_l}$
(21) **od**
(22) $i := i + 1$
(23) **od**
(24) $p := p + 1$
(25) **od**

Wir können nun, mit Hilfe der Bytedarstellung eines Speicherstücks, die nachfolgenden Zugriffsfunktionen angeben.

$$load_{int_m} \; : \; \mathcal{M}_k \times \mathbb{N} \longrightarrow I_n^*(int_m)$$
$$load_{float_m} \; : \; \mathcal{M}_k \times \mathbb{N} \longrightarrow I_n^*(float_m)$$

mit

$$load_{int_m}\ \left(\mathcal{M}_{i=1}^{k}(o_i,d_i),o\right) \stackrel{\text{def}}{=} \begin{cases} d_l, \text{ falls } o_l = o \text{ und } d_l \text{ ein } int_m \text{ ist} \\ rep_m^{-1}\left(\left(\left[a_{i_o},b_{i_o}\right],\ldots,\left[a_{i_{o+m-1}},b_{i_{o+m-1}}\right]\right)_{i=1}^{n}\right) \end{cases}$$

$$load_{float_m}\ \left(\mathcal{M}_{i=1}^{k}(o_i,d_i),o\right) \stackrel{\text{def}}{=} \begin{cases} d_l, \text{ falls } o_l = o \text{ und } d_l \text{ ein } float_m \text{ ist} \\ rep_m^{-1}\left(\left(\left[a_{i_o},b_{i_o}\right],\ldots,\left[a_{i_{o+m-1}},b_{i_{o+m-1}}\right]\right)_{i=1}\ n\right) \end{cases}$$

wobei

$$\left(\left[a_{i_0},b_{i_0}\right],\ldots,\left[a_{i_{|\mathcal{M}_k|-1}},b_{i_{|\mathcal{M}_k|-1}}\right]\right)_{i=1}^{n}$$

die Bytedarstellung von $\mathcal{M}_{i=1}^{k}(o_i,d_i)$ ist.

Da wir für Zeiger keine genaue Bytedarstellung angeben können, definieren wir $load_Z$ folgendermaßen.

$$load_Z\ :\ \mathcal{M}_k \times \mathbb{N} \longrightarrow Z$$

mit

$$load_Z\ \left(\mathcal{M}_{i=1}^{k}(o_i,d_i),o\right) \stackrel{\text{def}}{=} \begin{cases} d_l, \text{ falls } o_l = o \text{ und } d_l \text{ ein Zeiger ist} \\ \top, \text{ sonst} \end{cases}$$

Um den schreibenden Zugriff auf ein Speicherstück zu implementieren benötigen wir den Algorithmus `rep_memory`, der aus einer gegeben Bytedarstellung die einzelnen Komponenten eines Speicherstücks aktualisiert. Die Speicherfunktionen sehen folgendermaßen aus. Die Bytedarstellung von $\mathcal{M}_{i=1}^{k}(o_i,d_i)$ sei $\left(\left[a_{i_0},b_{i_0}\right],\ldots,\left[a_{i_{|\mathcal{M}_k|-1}},b_{i_{|\mathcal{M}_k|-1}}\right]\right)_{i=1}^{n}$.

$$store_{int_m}\ :\ \mathcal{M}_k \times \mathbb{N} \times I_n^*(int_m) \longrightarrow \mathcal{M}_k$$

$$store_{int_m}\ \left(\mathcal{M}_{i=1}^{k}(o_i,d_i),o,x\right) \stackrel{\text{def}}{=}$$

$$\begin{cases} \mathcal{M}_{i=1}^{k}(o_i,d_i'), \text{ falls } o_l = o \text{ und } d_l \text{ ein } int_m \text{ ist} \\ \texttt{rep\_memory}\left(\left(\left[a'_{i_0},b'_{i_0}\right],\ldots,\left[a'_{i_{|\mathcal{M}_k|-1}},b'_{i_{|\mathcal{M}_k|-1}}\right]\right)_{i=1}^{n}\right) \end{cases}$$

mit

$$d_i' \stackrel{\text{def}}{=} \begin{cases} x, \text{ falls } i = l \\ d_i, \text{ sonst} \end{cases}$$

$$\left[a'_{i_j},b'_{i_j}\right] \stackrel{\text{def}}{=} \begin{cases} \left[a_{i_j},b_{i_j}\right], \text{ falls } j < o \text{ oder } j \geq o + |x| \\ \left[c_{i_{j-o}},d_{i_{j-o}}\right], \text{ sonst, wobei} \\ \qquad rep_m(x) = \left(\left[c_{i_0},d_{i_0}\right],\ldots,\left[c_{i_{m-1}},d_{i_{m-1}}\right]\right)_{i=1}^{n} \end{cases}$$

und

$$
\begin{aligned}
&store_{float_m} \ : \ \mathcal{M}_k \times \mathbb{N} \times I_n^*(float_m) \longrightarrow \mathcal{M}_k \\
&store_{float_m} \ \left(\mathcal{M}_{i=1}^k(o_i, d_i'), o, f\right) \ \stackrel{\text{def}}{=} \\
&\qquad \begin{cases} \mathcal{M}_{i=1}^k(o_i, d_i'), \text{falls } o_l = o \text{ und } d_l \text{ ein } float_m \text{ ist} \\ \texttt{rep\_memory}\left(\left(\left[a_{i_0}', b_{i_0}'\right], \ldots, \left[a_{i_{|\mathcal{M}_k|-1}}', b_{i_{|\mathcal{M}_k|-1}}'\right]\right)_{i=1}^n\right) \end{cases}
\end{aligned}
$$

mit

$$
\begin{aligned}
d_i' \ &\stackrel{\text{def}}{=} \begin{cases} f, \text{falls } i = l \\ d_i, \text{sonst} \end{cases} \\
\left[a_{i_j}', b_{i_j}'\right] \ &\stackrel{\text{def}}{=} \begin{cases} \left[a_{i_j}, b_{i_j}\right], \text{falls } j < o \text{ oder } j \geq o + |f| \\ \left[f_{i_{j-o}}, g_{i_{j-o}}\right], \text{sonst, wobei} \\ \qquad rep_m(f) = \left(\left[f_{i_0}, g_{i_0}\right], \ldots, \left[f_{i_{m-1}}, g_{i_{m-1}}\right]\right)_{i=1}^n \end{cases}
\end{aligned}
$$

Das Speichern von Zeigern innerhalb eines Speicherstücks wir hier aufgrund der ungenauen Bytedarstellung folgendermaßen definiert.

$$
\begin{aligned}
&store_Z \ : \ \mathcal{M}_k \times \mathbb{N} \times Z \longrightarrow \mathcal{M}_k \\
&store_Z \ \left(\mathcal{M}_{i=1}^k(o_i, d_i), o, z\right) \ \stackrel{\text{def}}{=} \\
&\qquad \begin{cases} \mathcal{M}_{i=1}^k(o_i, d_i'), \text{falls } o_l = o \text{ und } d_l \text{ ein Zeiger ist} \\ \texttt{rep\_memory}\left(\left(\left[a_{i_0}', b_{i_0}'\right], \ldots, \left[a_{i_{|\mathcal{M}_k|-1}}', b_{i_{|\mathcal{M}_k|-1}}'\right]\right)_{i=1}^n\right) \end{cases}
\end{aligned}
$$

mit

$$
\begin{aligned}
d_i' \ &\stackrel{\text{def}}{=} \begin{cases} z, \text{falls } i = l \\ d_i, \text{sonst} \end{cases} \\
\left[a_{i_j}', b_{i_j}'\right] \ &\stackrel{\text{def}}{=} \begin{cases} \left[a_{i_j}, b_{i_j}\right], \text{falls } j < o \text{ oder } j \geq o + |z| \\ \left[0, 255\right], \text{sonst} \end{cases}
\end{aligned}
$$

Der verwendete Algorithmus `rep_memory` sieht folgendermaßen aus.

Algorithmus 4.2.2

Algorithmus `rep_memory`

EINGABE: $\left(\left[a_{i_0}, b_{i_0}\right], \ldots, \left[a_{i_{|\mathcal{M}_k|-1}}, b_{i_{|\mathcal{M}_k|-1}}\right]\right)_{i=1}^{n}$
AUSGABE: $\mathcal{M}_{i=1}^{n}(o_i, d_i) \in \mathcal{M}_k$

(1) $p = 1$
(2) $i = 0$
(3) **while** $(i < |\mathcal{M}_k|)$ **do**
(4) **while** $(i < o_p)$ **do**
(5) $i := i + 1$
(6) **od**
(7) $d_p := rep_m^{-1}\left(\left(\left[a_{j_i}, b_{j_i}\right], \ldots, \left[a_{j_{i+|d_p|-1}}, b_{j_{i+|d_p|-1}}\right]\right)_{j=1}^{n}\right)$
(8) $i := i + |d_p|$
(9) $p := p + 1$
(10) **od**

In der Praxis sind Speicherstücke meistens unstrukturiert. Wir erhalten für ein Speicherstück der Größe k eine Liste mit k Komponenten, mit Offset von 0 bis $k - 1$ und einem $I_n^*(int_1)$ als gespeicherten Wert. Wir erlauben für diesen Fall eine dynamische Strukturierung, die sich durch die Benutzung der Speicherfunktionen ergibt. Für weitergehende Informationen verweisen wir auf die Implementierung der `xmemory` Klasse.

In diesem Kapitel sind wir auf die Grunddatentypen int_1, int_2, int_4, int_8 und $float_4$, $float_8$, $float_{12}$, $float_{16}$, d.h. auf die in der Praxis konkret verwendeten Instanzen der endlichen Mengen von ganzen und reellen Zahlen eingegangen. Wir haben deren Bit- und Bytedarstellung vorgestellt und entsprechende Zugriffsfunktionen definiert. Die Definition des gültigen Intervalls wurde weiter eingeschränkt um Probleme beim Zugriff auf die Bytedarstellung zu vermeiden. Die vorgestellten Algorithmen wurden entsprechend angepasst. Wir haben Zeiger und Speicherstücke weiter präzisiert und gezeigt, wie wir unabhängig von der Struktur eines Speicherstücks den Zugriff implementieren, mit Hilfe der Zugriffsfunktionen auf Grunddatentypen.

Kapitel 5

Abstrakte Interpretation von XRTL

In diesem Kapitel zeigen wir wie mit Hilfe der Ergebnisse aus Kapitel 3 und 4 die Sprache XRTL analysiert werden kann. Wir verwenden die in Kapitel 2 eingeführten Erweiterungen, um den Stackrahmen für Funktionen zu rekonstruieren. Wir zeigen, wie die Abstrakte Interpretation für einfache XRTL Anweisungen durchführt wird und erklären, wie Verzweigungen und Schleifen ausgeführt werden. Der Bedingungsteil entscheidet bei Verzweigungen, welcher der Zweige ausgeführt werden muss. Wir verwenden die Fixpunktberechnung zur Analyse von Schleifen. Wir verwenden, um diese Berechnung zu beschleunigen, das so genannte Widening/Narrowing. Wir erlauben auch die Iteration der Schleife, da der Abstrakte Domain endlich ist und endlich viele Schritte zu einem Fixpunkt führen. Wir beschreiben unseren Ansatz zu Ausführen von Funktionsaufrufen.

Wir verfolgen mit der Abstrakten Interpretation das Ziel, Informationen für jedes mögliche Laufzeitverhalten zu erhalten ohne hierzu das Programm konkret auszuführen. Wir verwenden hierzu die in Kapitel 3 vorgestellte Approximation und berechnen mit Hilfe der dazugehörigen Operationen, wie wir von einer Menge von Zuständen unter Ausführung einzelner Anweisungen zu einer anderen Menge von Zuständen kommen. Im nachfolgenden Ausführungsschema (siehe Bild 5.1) verwenden wir eine Prioritätswarteschlange PQ_{pc} die Paare von PC (program counter) und eine Menge von Zuständen enthält. Wir benötigen PQ_{pc} da Sprünge innerhalb von XRTL dazu führen können, dass Instruktionen übersprungen werden.

5.1 Anfangszustand

Der Anfangszustand (initial state) mit

$$state \stackrel{\text{def}}{=} PC \times Env_R \times Env_M$$

besteht aus einem Programmzähler PC, einer Umgebung Env_R für Variablen (Register) und Env_M für Speicherstücke, wobei Env_R Variablen an Werte aus $I_n^*(t_m)$, mit $t \in \{int_1, \ldots, int_8, float_4, \ldots, float_{16}\}$ den in Kapitel 4 eingeführten Grunddatentypen bindet und Env_M eindeutige Namen an Speicherstücke bindet.

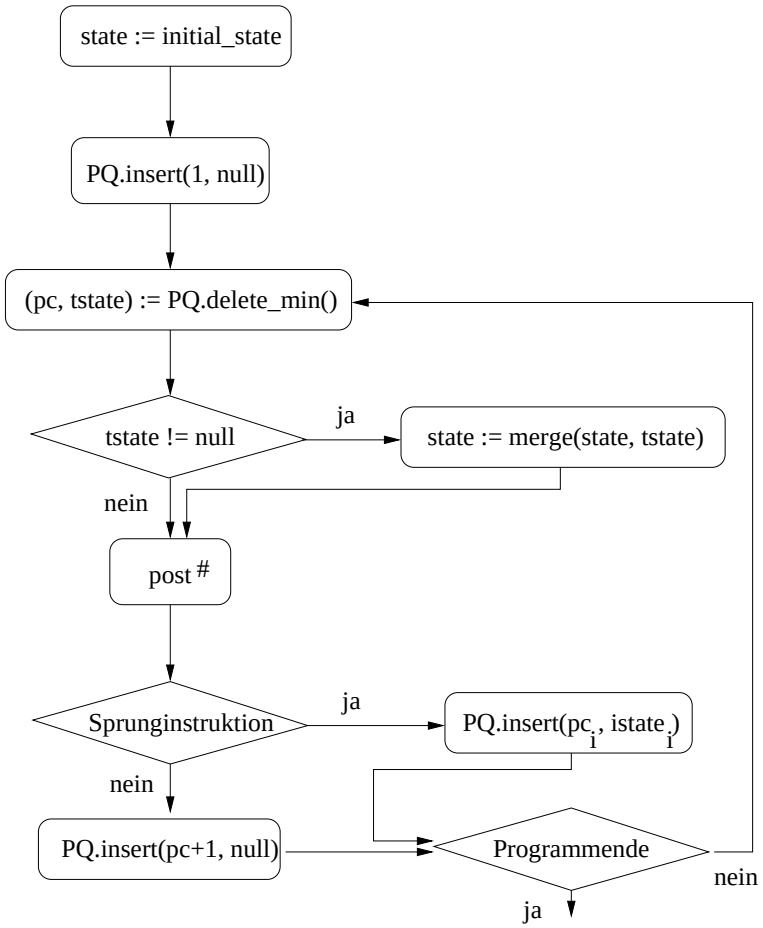


Abbildung 5.1: Ausführungsschema

$$
\begin{aligned}
Env_R &\stackrel{\text{def}}{=} Variable \longrightarrow I_n^*(int_1) \cup I_n^*(int_2) \cup I_n^*(int_4) \cup I_n^*(int_8) \cup \\
&\qquad I_n^*(float_4) \cup I_n^*(float_8) \cup I_n^*(float_{12}) \cup I_n^*(float_{16}) \\
Env_M &\stackrel{\text{def}}{=} Mem \longrightarrow M_n
\end{aligned}
$$

Wir erlauben eine endliche Anzahl n_{reg} von Variablen jedes Typs. Diese entsprechen den in XRTL verwendeten Registern. Es werden während der Analyse eines Programms keine neuen Variablen mehr hinzugefügt. Wir bezeichnen mit

$$R_l^t \longrightarrow I_m^*(t)$$

mit $t \in \{int_1, \ldots, int_8, float_4, \ldots, float_{16}\}$ und $l \leq n_{reg} \in \mathbb{N}$ ein Register eines bestimmten Typs t. Abhängig vom Maschinentyp auf dem das XRTL Programm ausgeführt werden soll, können in Registern vom Typ int_4 oder int_8 auch Zeiger abgespeichert werden. Wir gehen

im folgenden davon aus, dass Register vom Typ int_4 Zeiger speichern können. In XRTL Anweisungen kann ein Zeiger als ganzzahliger Wert und jede ganze Zahl kann auch als Zeiger betrachtet werden. Es existiert deshalb eine implizite Umwandlung zwischen diesen beiden Datentypen, d.h. für Register vom Typ int_4 mit

$$R_l^{int_4} \longrightarrow I_4^*(int_m) \cup Z$$

gilt

$$\begin{aligned} conv_Z &: I_4^*(int_m) \longrightarrow Z \\ conv_Z\ (a) &\stackrel{\text{def}}{=} \top \end{aligned}$$

und

$$\begin{aligned} conv_{int_4} &: Z \longrightarrow I_4^*(int_m) \\ conv_{int_4}\ (a) &\stackrel{\text{def}}{=} \top \end{aligned}$$

Diese Art der Konvertierung ist sehr unpräzise, lässt sich aber aufgrund der fehlenden Startadresse eines Speicherstücks, die erst zur Laufzeit bekannt ist, nicht weiter verfeinern.

Im Startzustand sind alle Register uninitialisiert, d.h.

$$R_l^t \longrightarrow \bot \qquad \forall t \in \{int_1, \ldots, int_8, float_4, \ldots, float_{16}\}, 0 \leq l \leq n_{reg}$$

und zu Beginn existiert kein Speicherstück, außer in Form von globalen Variablen. Bevor wir die einzelnen Instruktionen einer Funktion abarbeiten können, müssen wir mit Hilfe der XRTL Zusatzinformationen aus Abschnitt 2.4 den Stackrahmen rekonstruieren und die Parameter der aufgerufenen Funktionen zugänglich machen. Wir erhalten für den Stackrahmen eine Liste von Paaren aus Variablentyp und dem so genannten Bytealignment.

$$S \stackrel{\text{def}}{=} \left(t_i, a_i\right)_{i=1}^{l}$$

Der Begriff Bytealignment bezeichnet die Teilbarkeit des dazugehörigen Offset durch eben dieses Bytealignment. Aus der gegebenen Liste S können wir für jede vorkommende Funktion ein Speicherstück konstruieren, welches wir $stack$ nennen und dessen Komponenten uninitialisiert sind. Wir müssen beachten, dass es zu so genannten Lücken im Stack kommen kann, auf die wir nicht zugreifen dürfen. Das Speicherstück $stack$ sieht folgendermaßen aus:

$$stack \longrightarrow \left(o_i, v_i\right)_{i=1}^{l}$$

wobei

$$v_i \in I_i^*(t_m), v_i = \perp$$

und

$$\begin{aligned} o_0 &= 0 \\ o_{i+1} &= \left\lceil \frac{(o_i + |v_i|)}{a_i} \right\rceil a_i \quad \text{für } 1 < i \leq l \end{aligned}$$

Nachdem der Stack erzeugt wurde können wir die im XRTL Code aufgeführten Register an einen Zeiger auf das Speicherstück $stack$ binden, mit

$$\begin{aligned} R_{sr}^{int_4} &\longrightarrow (stack, 0) \in Z \\ R_{hr}^{int_4} &\longrightarrow (stack, 0) \in Z \end{aligned}$$

wobei sr die Nummer des Stackregisters und hr die Nummer des Hardwarestackregisters ist. Wir müssen nun das Register, welches für die Parameterübergabe verantwortlich ist initialisieren. Dieses Register $R_{in}^{int_4}$ zeigt auf ein Speicherstück $param$, das eine Liste möglicher Parameter enthält.

$$R_{in}^{int_4} \longrightarrow (param, 0) \in Z$$

Die XRTL Sprache erlaubt auch das Übergeben von Parametern mit Hilfe von Registern. Diese werden, für diesen Fall, entsprechend initialisiert. Informationen über Anzahl und Aussehen der Parameter finden sich in den XRTL Zusatzinformationen. Diese können jedoch durch Benutzerinteraktion sinnvoll präzisiert werden. In einigen Fällen ist eine Unterscheidung zwischen einem Zeiger und dem Datentyp int_4 nicht möglich.

5.2 Instruktionen

Nachdem diese Initialisierungsschritte abgeschlossen sind, können wir uns mit der Interpretation einzelner Instruktionen befassen. Wir unterscheiden hier einfache Instruktionen, Parallel-Instruktionen, Verzweigungen, Schleifen und Sprünge.

5.2.1 Einfache Instruktionen

Wir befassen uns mit der Interpretation von einfachen XRTL Anweisungen und wie sich diese auf die Menge von Zuständen auswirkt. Wir definieren hierzu die Bedeutung der innerhalb einfacher XRTL Anweisungen auftretenden Teilausdrücke.

Eine einfache XRTL Anweisung sieht folgendermaßen aus:

$$\texttt{(insn x y z (set } (lvalue)^t \ (rvalue)^t\texttt{))}$$

Die ganzzahligen Werte x, y und z werden zur Numerierung der Anweisungen verwendet, wobei y und z die Nummern für die Vorgänger- bzw. Nachfolgeranweisungen sind. Der Ausdruck

$$\texttt{(set } (lvalue)^t \ (rvalue)^t\texttt{)}$$

entspricht einer einfachen Zuweisung $(lvalue)^t := (rvalue)^t$ wobei die beiden Teilausdrücke denselben Typ haben. Mit Ausnahme von Zeigern und int_4 findet hier keine implizite Konvertierung statt, wie sie aus anderen Programmiersprachen bekannt ist. Wir stellen später allerdings Anweisungen vor, die uns eine Typkonvertierung ermöglichen. Es gibt drei verschiedene Arten von $(lvalue)$ Ausdrücken:

- `(reg:MODE` (no)`)`
- `(mem:MODE` $(pointer)$`)`
- `(subreg:MODE` $(register)$`)`

Der erste dieser Ausdruck entspricht einer existierenden Variablen aus der Umgebung. Zu jedem angegebenen Modus gibt es einen korrespondierenden Datentyp.

Modus	:QI	:HI	:SI	:DI	:SF	:DF	:XF	:TF
Datentyp	int_1	int_2	int_4	int_8	$float_4$	$float_8$	$float_{12}$	$float_{16}$

Die Variable R^t_{no} entspricht dem im Ausdruck (reg:MODE$_t$ (no)) bezeichneten Register.

Wir bezeichnen mit $\rho \in Env_R \times Env_M$ eine bestimmte Umgebung ρ und die Bedeutung des Ausdrucks $(rvalue)^t$ unter Berücksichtigung von ρ mit $[\![(rvalue)^t]\!]\rho$. Unter diesen Voraussetzungen können wir jetzt für einfache XRTL Anweisungen ein $post^\#$ angeben. Diese Funktion geht ausgehend von einer Menge möglicher Zustände unter Anwendung einer XRTL Instruktion in eine andere Menge möglicher Zustände über. Für eine Instruktion c mit

$$c = \texttt{(insn x y z (set } (R^t_{no}) \ (rvalue)^t\texttt{)}$$

ergibt sich

$$post^\#_c\ (pc, \rho) \ \stackrel{\text{def}}{=} \ \left(pc + 1, \rho\left[R^t_{no} \longrightarrow [\![(rvalue)^t]\!]\rho\right]\right)$$

wobei die Registernummer no eindeutig ist. Dies trifft für Zeiger (siehe zweiter Ausdruck) nicht zu. Zeigern können auf verschiedene Objekte zeigen und unterschiedliche Offsets haben. Wir müssen die Instruktion c mit

$$c = \texttt{(insn x y z (set (mem:MODE } (pointer)\texttt{)) } (rvalue)^t\texttt{)}$$

für jede Kombination von Offset und Speicherstück durchführen. Die Anzahl der möglichen Kombinationen ist beschränkt durch das Produkt aus der Länge des größten beteiligten Speicherstücks und der Anzahl der Speicherstücke, ist also endlich, da nur eine beschränkte Menge an Speicher zur Verfügung steht. Wir erhalten nun

$$post_c^{\#}\,(pc,\rho) \;\stackrel{\text{def}}{=}\; \left(pc+1, \rho\left[m_i \;\longrightarrow\; store_t^{\#}\left(m_i, o_j, [\![(rvalue)^t]\!]\rho\right),\; \forall\, i,j\right]\right)$$

wobei m_i mit $1 \leq i \leq l$ die beteiligten Speicherstücke sind und o_j, mit $1 \leq j \leq n$ die möglichen Offsets. Der Ausdruck $[\![(rvalue)^t]\!]\rho$ muss vorberechnet werden, da er möglicherweise abhängig von den modifizierten Speicherstücken ist. In der Praxis verwenden wir Techniken um den Anzahl der Zugriffsoperationen zu reduzieren ohne dabei an Präzision zu verlieren.

Die so genannten Sub-Register (siehe dritter Ausdruck) erlauben den Zugriff auf ein Register mit einem XRTL Ausdruck dessen Typ nicht dem Typ des Registers entspricht. Um diese implizite Konvertierung zu beschreiben verwenden wir dir nachfolgende Funktion

$$sr_{t_{out}}^{t_{in}} \;:\; I_m^*(t_{in}) \times I_m^*(t_{out}) \times \mathbb{N} \longrightarrow I_m^*(t_{out})$$

die allerdings den vorherigen Wert des Registers vom Typ t_{out} berücksichtigt. Der dritte ganzzahlige Parameter der Funktion gibt an, welcher Teil des Registers betrachtet werden soll. Dieses Offset ist eindeutig. Für die Instruktion c mit

$$c = \texttt{(insn x y z (set (subreg:MODE}_{in}\ (R_{no}^{t_{out}})\ \texttt{(offset))}\ (rvalue)^{t_{in}}\texttt{)}$$

ergibt sich

$$post_c^{\#}\,(pc,\rho) \;\stackrel{\text{def}}{=}\; \left(pc+1, \rho\left[R_{no}^{t_{out}} \;\longrightarrow\; sr_{t_{out}}^{t_{in}}\left([\![(rvalue)^{t_{in}}]\!]\rho, R_{no}^{t_{out}}, offset\right)\right]\right)$$

Nachdem wir uns mit den drei Möglichkeiten eines $(lvalue)^t$ Ausdruck für einfache Instruktionen beschäftigt haben, geben wir nun die Bedeutung der $(rvalue)^t$ Ausdrücke an. Wir gehen zunächst auf die Konstanten ein.

$$\begin{aligned}
[\![\texttt{(const_int:MODE}_t\ (cvalue)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} \texttt{merge}([(cvalue)^t, (cvalue)^t]) \\
[\![\texttt{(const_double:MODE}_t\ (cvalue)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} \texttt{merge}([(cvalue)^t, (cvalue)^t]) \\
[\![\texttt{(symbol_ref:MODE}_t\ (string)\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} ((string), 0) \in Z
\end{aligned}$$

Der Ausdruck $(cvalue)^t$ beschreibt einen eindeutigen Wert des Typs t und $(string)$ beschreibt den Namen einer existierenden, möglicherweise globalen Variablen. Wir gehen nun auf die arithmetischen Grundoperationen ein.

$$
\begin{aligned}
[\![\texttt{(plus:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} add^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![\texttt{(minus:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} sub^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![\texttt{(mult:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} mul^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![\texttt{(div:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} div^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![\texttt{(neg:MODE}_t\ \ (E)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} neg^{\#}\left([\![(E)^t]\!]\rho\right)
\end{aligned}
$$

Die nachfolgenden Ausdrücke sind nur für $t \in \{int_1, int_2, int_4, int_8\}$ definiert.

$$
\begin{aligned}
[\![\texttt{(udiv:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} udiv^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![\texttt{(mod:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} mod^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![\texttt{(umod:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} umod^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)
\end{aligned}
$$

Es existiert eine spezielle Form der Addition, die allerdings nur in einer festgelegten Kombination von Instruktionen auftaucht. Diese haben das folgende Aussehen

$$
\begin{aligned}
&\texttt{(insn x y z (set } (R^t_{no}) \texttt{ (high:MODE } (rvalue)^t\texttt{)))}\\
&\texttt{(insn z x u (set } (R^t_{no'}) \texttt{ (lo\_sum:MODE } (R^t_{no}) \ (rvalue)^t\texttt{)))}
\end{aligned}
$$

und entsprechen einer Zuweisung

$$
\texttt{(insn x y z (set } (R^t_{no'}) \ (rvalue)^t\texttt{))}
$$

wobei das Register R^t_{no} verändert wird.

Die Bedeutung der nachfolgenden Operationen wurde nicht implementiert, da sie in den getesteten Programmen, auf den von uns verwendeten Systemen, nicht auftraten.

$$
\begin{aligned}
&[\![\texttt{(ss\_plus:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho\\
&[\![\texttt{(ss\_minus:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho\\
&[\![\texttt{(us\_plus:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho\\
&[\![\texttt{(us\_minus:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho
\end{aligned}
$$

Diese Ausdrücke haben eine ähnlich Bedeutung wie die bereits definierte Addition und Subtraktion unterscheiden sich jedoch im Verhalten beim Überlauf des Ergebnisses und sind deshalb nur für ganze Zahlen interessant.

Die folgenden Ausdrücke werden verwendet um das Minimum bzw. Maximum von zwei Ausdrücken zu bestimmen. Wir benötigen hierzu die Ergebnisintervalle der Teilausdrücke (E^t_1) und (E^t_2) mit

$$\begin{aligned}\left(\left[a_i, b_i\right]\right)_{i=1}^{n} &= [\![(E_1)^t]\!]\rho \\ \left(\left[c_j, d_j\right]\right)_{j=1}^{n} &= [\![(E_2)^t]\!]\rho\end{aligned}$$

und erhalten nun

$$\begin{aligned}[\![\texttt{(smin:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} \texttt{merge}\left(\left(\left[a'_i, b'_i\right]\right)_{i=1}^{n} \cup \left(\left[c'_i, d'_i\right]\right)_{i=1}^{n}\right) \\ [\![\texttt{(smax:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} \texttt{merge}\left(\left(\left[a''_i, b''_i\right]\right)_{i=1}^{n} \cup \left(\left[c''_i, d''_i\right]\right)_{i=1}^{n}\right)\end{aligned}$$

mit

$$x_{cd} \stackrel{\text{def}}{=} \max\{y | y \in \left(\left[c_j, d_j\right]\right)_{j=1}^{n}, y \leq b_i\}\,,\ \ x_{ab} \stackrel{\text{def}}{=} \max\{y | y \in \left(\left[a_j, b_j\right]\right)_{j=1}^{n}, y \leq d_i\}$$

$$[a'_i, b'_i] \stackrel{\text{def}}{=} \begin{cases}[a_i, x_{cd}], \text{falls } x_{cd} \geq a_i \\ \bot, \text{sonst}\end{cases} \qquad [c'_i, d'_i] \stackrel{\text{def}}{=} \begin{cases}[c_i, x_{ab}], \text{falls } x_{ab} \geq c_i \\ \bot, \text{sonst}\end{cases}$$

und

$$x^{cd} \stackrel{\text{def}}{=} \min\{y | y \in \left(\left[c_j, d_j\right]\right)_{j=1}^{n}, y \geq a_i\}\,,\ \ x^{ab} \stackrel{\text{def}}{=} \min\{y | y \in \left(\left[a_j, b_j\right]\right)_{j=1}^{n}, y \geq c_i\}$$

$$[a''_i, b''_i] \stackrel{\text{def}}{=} \begin{cases}[x^{cd}, b_i], \text{falls } x^{cd} \leq b_i \\ \bot, \text{sonst}\end{cases} \qquad [c''_i, d''_i] \stackrel{\text{def}}{=} \begin{cases}[x^{ab}, d_i], \text{falls } x^{ab} \leq d_i \\ \bot, \text{sonst}\end{cases}$$

Die für ganzzahlige Datentypen zur Verfügung stehenden

$$\begin{aligned}[\![\texttt{(umin:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \\ [\![\texttt{(umax:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho\end{aligned}$$

entsprechen den Bedeutungen von $smin$ und $smax$ mit der Einschränkung, dass alle Zahlen vorzeichenlos betrachtet werden.

Die von XRTL unterstützten Bitoperationen sind nur für ganzzahlige Datentypen definiert und haben die folgende Bedeutung.

$$\begin{aligned}[\![\texttt{(and:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} and^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right) \\ [\![\texttt{(ior:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} ior^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right) \\ [\![\texttt{(xor:MODE}_t\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} xor^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right) \\ [\![\texttt{(not:MODE}_t\ \ (E)^t\texttt{)}]\!]\,\rho &\stackrel{\text{def}}{=} not^{\#}\left([\![(E)^t]\!]\rho\right)\end{aligned}$$

und die Shift- und Rotateausdrücke

$$\begin{aligned}
[\![(\texttt{lshift:MODE}_t\ \ (E_1)^t\ \ (E_2)^t)]\!]\,\rho &\stackrel{\text{def}}{=} shift_left^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![(\texttt{lshiftrt:MODE}_t\ \ (E_1)^t\ \ (E_2)^t)]\!]\,\rho &\stackrel{\text{def}}{=} shift_right^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![(\texttt{ashift:MODE}_t\ \ (E_1)^t\ \ (E_2)^t)]\!]\,\rho &\stackrel{\text{def}}{=} ashift_left^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![(\texttt{ashiftrt:MODE}_t\ \ (E_1)^t\ \ (E_2)^t)]\!]\,\rho &\stackrel{\text{def}}{=} ashift_right^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![(\texttt{rotate:MODE}_t\ \ (E_1)^t\ \ (E_2)^t)]\!]\,\rho &\stackrel{\text{def}}{=} rotate_left^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\\
[\![(\texttt{rotatert:MODE}_t\ \ (E_1)^t\ \ (E_2)^t)]\!]\,\rho &\stackrel{\text{def}}{=} rotate_right^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)
\end{aligned}$$

Wir haben bis jetzt nur Ausdrücke kennengelernt, die auf einen Datentyp beschränkt sind. Es ist jedoch oft erforderlich zwischen verschiedenen Typen zu konvertieren. Hierfür stehen die nachfolgenden Ausdrücke zur Verfügung:

$$\begin{aligned}
[\![(\texttt{sign_extend:MODE}_{t'}\ \ (E)^t)]\!]\,\rho &\stackrel{\text{def}}{=} sign_extend_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)\\
[\![(\texttt{zero_extend:MODE}_{t'}\ \ (E)^t)]\!]\,\rho &\stackrel{\text{def}}{=} zero_extend_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)
\end{aligned}$$

mit $t, t' \in \{int_1, int_2, int_4, int_8\}$ und $|t| < |t'|$

$$[\![(\texttt{float\_extend:MODE}_{t'}\ \ (E)^t)]\!]\,\rho \stackrel{\text{def}}{=} float\_extend_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)$$

mit $t, t' \in \{float_4, float_8, float_{12}, float_{16}\}$ und $|t| < |t'|$

$$[\![(\texttt{truncate:MODE}_{t'}\ \ (E)^t)]\!]\,\rho \stackrel{\text{def}}{=} truncate_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)$$

mit $t, t' \in \{int_1, int_2, int_4, int_8\}$ und $|t| > |t'|$

$$[\![(\texttt{float\_truncate:MODE}_{t'}\ \ (E)^t)]\!]\,\rho \stackrel{\text{def}}{=} float\_truncate_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)$$

mit $t, t' \in \{float_4, float_8, float_{12}, float_{16}\}$ und $|t| > |t'|$

$$\begin{aligned}
[\![(\texttt{float:MODE}_{t'}\ \ (E)^t)]\!]\,\rho &\stackrel{\text{def}}{=} float_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)\\
[\![(\texttt{unsigned_float:MODE}_{t'}\ \ (E)^t)]\!]\,\rho &\stackrel{\text{def}}{=} unsigned_float_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)
\end{aligned}$$

mit $t \in \{int_1, int_2, int_4, int_8\}$ und $t' \in \{float_4, float_8, float_{12}, float_{16}\}$

$$[\![(\texttt{fix:MODE}_{t'}\ \ (E)^t)]\!]\,\rho \stackrel{\text{def}}{=} fix_{t'}^{\#}\left([\![(E)^t]\!]\rho\right)$$

mit $t \in \{float_4, float_8, float_{12}, float_{16}\}$ und $t' \in \{int_1, int_2, int_4, int_8\}$

Für $op \in \{sign\_extend, zero\_extend, truncate\}$:

$$op^{\#} \ : \ I_n^*(int_t) \longrightarrow I_n^*(int_{t'})$$

$$op^{\#}\left(\left(\left[a_i, b_i\right]\right)_{i=1}^{n}\right) \stackrel{\text{def}}{=} \texttt{merge}\left(\bigcup_{i=1}^{\leq n} op^{\#}\left([a_i, b_i]\right)\right)$$

wobei

$$\begin{aligned} sign\_extend_{t'}^{\#} &: I^*(int_t) \longrightarrow I^*(int_{t'}) \\ zero\_extend_{t'}^{\#} &: I^*(int_t) \longrightarrow I^*(int_{t'}) \\ truncate_{t'}^{\#} &: I^*(int_t) \longrightarrow I^*(int_{t'}) \end{aligned}$$

mit

$$\begin{aligned} sign\_extend_{t'}^{\#}\ ([a,b]) &\stackrel{\text{def}}{=} \begin{cases} [a,b], \text{ falls } b < 2^{8|t'|-1} \\ \left[2^{8|t|}-a, 2^{8|t|}-b\right], \text{ falls } a \geq 2^{8|t'|-1} \end{cases} \\ zero\_extend_{t'}^{\#}\ ([a,b]) &\stackrel{\text{def}}{=} [a,b] \\ truncate_{t'}^{\#}\ ([a,b]) &\stackrel{\text{def}}{=} \Big[\min\{a \bmod 2^{8|t'|}, b \bmod 2^{8|t'|}\}\ , \\ &\qquad \max\{a \bmod 2^{8|t'|}, b \bmod 2^{8|t'|}\}\Big] \end{aligned}$$

Für $op \in \{float\_extend, float\_truncate\}$:

$$op^{\#} \ : \ I_n^*(float_t) \longrightarrow I_n^*(float_{t'})$$

$$op^{\#}\left(\left(\left[a_i, b_i\right]\right)_{i=1}^{n}\right) \stackrel{\text{def}}{=} \texttt{merge}\left(\bigcup_{i=1}^{\leq n} op^{\#}\left([a_i, b_i]\right)\right)$$

wobei

$$\begin{aligned} float\_extend_{t'}^{\#} &: I^*(float_t) \longrightarrow I^*(float_{t'}) \\ float\_truncate_{t'}^{\#} &: I^*(float_t) \longrightarrow I^*(float_{t'}) \end{aligned}$$

mit

$$\begin{aligned} float\_extend_{t'}^{\#}\ ([a,b]) &\stackrel{\text{def}}{=} [inf(a), sup(b)] \\ float\_truncate_{t'}^{\#}\ ([a,b]) &\stackrel{\text{def}}{=} [inf(a), sup(b)] \end{aligned}$$

Für $op \in \{float, unsigned\_float\}$:

$$op^{\#} \ : \ I_n^*(int_t) \longrightarrow I_n^*(float_{t'})$$

$$op^{\#}\left(\left(\left[a_i, b_i\right]\right)_{i=1}^{n}\right) \stackrel{\text{def}}{=} \texttt{merge}\left(\bigcup_{i=1}^{\leq n} op^{\#}\left([a_i, b_i]\right)\right)$$

wobei

$$\begin{aligned} float_{t'}^{\#} \ &: I^*(int_t) \longrightarrow I^*(float_{t'}) \\ unsigned\_float_{t'}^{\#} \ &: I^*(int_t) \longrightarrow I^*(float_{t'}) \end{aligned}$$

mit

$$\begin{aligned} float_{t'}^{\#}\ ([a,b]) \ &\stackrel{\text{def}}{=} \ \begin{cases} [inf(a), sup(b)], \text{ falls } b < 2^{8|t'|-1} \\ \left[inf(a - 2^{8|t|}), sup(b - 2^{8|t|})\right], \text{ falls } a \geq 2^{8|t'|-1} \end{cases} \\ unsigned\_float_{t'}^{\#}\ ([a,b]) \ &\stackrel{\text{def}}{=} \ [inf(a), sup(b)] \end{aligned}$$

Für $op \in \{fix\}$:

$$op^{\#} \ : \ I_n^*(float_t) \longrightarrow I_n^*(int_{t'})$$

$$op^{\#}\left(\left(\left[a_i, b_i\right]\right)_{i=1}^{n}\right) \stackrel{\text{def}}{=} \texttt{merge}\left(\bigcup_{i=1}^{\leq n} op^{\#}\left([a_i, b_i]\right)\right)$$

wobei

$$\begin{aligned} fix_{t'}^{\#} \ &: I^*(float_t) \longrightarrow I^*(int_{t'}) \\ fix_{t'}^{\#}\ ([a,b]) \ &\stackrel{\text{def}}{=} \ \left[\lfloor a \rfloor \bmod 2^{8|t'|}, \lfloor b \rfloor \bmod 2^{8|t'|}\right] \end{aligned}$$

Ein notwendiger Baustein, der sinnvolle Verzweigungen und Schleifen erst möglich macht sind die so genannten Vergleichsausdrücke. Die Ergebnisintervalle der Teilausdrücke (E_1^t) und (E_2^t) sehen folgendermaßen aus.

$$\begin{aligned} A \ &\stackrel{\text{def}}{=} \ \left(\left[a_i, b_i\right]\right)_{i=1}^{n} \ = \ [\![(E_1)^t]\!]\rho \\ C \ &\stackrel{\text{def}}{=} \ \left(\left[c_j, d_j\right]\right)_{j=1}^{n} \ = \ [\![(E_2)^t]\!]\rho \end{aligned}$$

Die Bedeutung der Vergleichsausdrücke, die nur für Verzweigungen und Schleifen verwendet werden, ist nun folgendermaßen definiert. Wir erhalten als Ergebnis eines solchen Ausdrucks eine Teilmenge von $\{0, 1\}$ bzw. $\{-1, 0, 1\}$ für compare.

$$[\![\texttt{(compare:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} sign^{\#}\left(sub^{\#}\left([\![(E_1)^t]\!]\rho, [\![(E_2)^t]\!]\rho\right)\right)$$

und

$$[\![\texttt{(eq:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} [1,1], \text{falls } \nexists x \in A, y \in C \text{ mit } x \neq y \\ [0,0], \text{falls } \forall x \in A, y \in C \text{ gilt } x \neq y \\ [0,0] \cup [1,1], \text{sonst} \end{cases}$$

$$[\![\texttt{(ne:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} [1,1], \text{falls } \forall x \in A, y \in C \text{ gilt } x \neq y \\ [0,0], \text{falls } \nexists x \in A, y \in C \text{ mit } x \neq y \\ [0,0] \cup [1,1], \text{sonst} \end{cases}$$

$$[\![\texttt{(gt:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} [1,1], \text{falls } \forall x \in A, y \in C \text{ gilt } x > y \\ [0,0], \text{falls } \nexists x \in A, y \in C \text{ mit } x \leq y \\ [0,0] \cup [1,1], \text{sonst} \end{cases}$$

$$[\![\texttt{(lt:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} [1,1], \text{falls } \forall x \in A, y \in C \text{ gilt } x < y \\ [0,0], \text{falls } \nexists x \in A, y \in C \text{ mit } x \geq y \\ [0,0] \cup [1,1], \text{sonst} \end{cases}$$

$$[\![\texttt{(ge:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} [1,1], \text{falls } \forall x \in A, y \in C \text{ gilt } x \geq y \\ [0,0], \text{falls } \nexists x \in A, y \in C \text{ mit } x < y \\ [0,0] \cup [1,1], \text{sonst} \end{cases}$$

$$[\![\texttt{(le:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} [1,1], \text{falls } \forall x \in A, y \in C \text{ gilt } x \leq y \\ [0,0], \text{falls } \nexists x \in A, y \in C \text{ mit } x > y \\ [0,0] \cup [1,1], \text{sonst} \end{cases}$$

Die Bedeutung, der für ganzzahlige Datentypen interessanten, Vergleichsausdrücke

$$[\![\texttt{(gtu:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho$$
$$[\![\texttt{(ltu:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho$$
$$[\![\texttt{(geu:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho$$
$$[\![\texttt{(leu:MODE}_{t'}\ \ (E_1)^t\ \ (E_2)^t\texttt{)}]\!]\,\rho$$

entsprechen der Bedeutung von gt, lt, ge bzw. le. Wir betrachten hier allerdings vorzeichenlose Zahlen.

Nachdem wir nun auf die Bedeutung der einzelnen Teilausdrücke eingegangen sind geben wir weitere einfache Instruktionen an. Die Instruktionen

$$\texttt{(insn x y z (clobber } (lvalue)^t\texttt{))}$$

wobei $(lvalue)^t$ ein Register, Subregister oder eine Speicherstelle sein kann und

$$\texttt{(insn x y z (use } (R^t_{no})\texttt{))}$$

werden meistens innerhalb der so genannten Parallel-Instruktionen verwendet. Die erste der beiden Instruktionen gibt an, welcher Teilausdruck zurückgesetzt wird, während die andere Anweisung angibt welches Register für eine vorherige oder nachfolgende Anweisung verwendet wird. Die exakten Auswirkungen dieser Ausdrücke sind kontext- und maschinenabhängig.

5.2.2 Parallel-Instruktionen

Eine Instruktion mit parallel auszuführenden Einzelinstruktionen hat das folgende Aussehen

$$\texttt{(insn x y z (parallel [(instr)}_1\ \texttt{(instr)}_2\ \ldots\ \texttt{(instr)}_n\ \texttt{]))}$$

wobei eine Teilinstruktion $\texttt{(instr)}_i$ aus einer der nachfolgenden Instruktionen

- $\texttt{(set } (lvalue)^t_i\ \ (rvalue)^t_i\texttt{)}$
- $\texttt{(clobber } (lvalue)^t_i\texttt{)}$
- $\texttt{(use } (lvalue)^t_i\texttt{)}$
- $\texttt{(use } (rvalue)^t_i\texttt{)}$

besteht. Wir sequentialisieren die einzelnen Teilinstruktionen dieser Parallel-Instruktion, weil eine simultane Ausführung die Analyse erschwert. Die Sequentialisierung ist möglich, weil für zwei Teilinstruktionen i und j gilt, dass die entsprechenden Ausdrücke $(lvalue)^t_i$ und $(lvalue)^t_j$ paarweise verschieden sind.

Wir erzeugen nun eine Kopie unserer Umgebung und führen die Teilinstruktionen sequentiell aus. Die lesenden Zugriffe auf Register und Speicherstücke werden auf der Kopie und schreibende Zugriffe auf dem Original durchführt. Nach Beendigung der Parallel-Instruktion können wir die Kopie unserer Umgebung verwerfen. Wir haben die implementierten Datenstrukturen entsprechend angepasst.

5.2.3 Verzweigungen

Eine der wichtigsten Strukturen sind die so genannten Verzweigungen. Diese bestehen aus einer Bedingung, die aus mehreren Teilbedingungen bestehen kann und den Zweigen, von denen es mindestens einen geben muss. Wir erhalten die folgende Struktur:

```
    condition;
br1: branch1;
      ⋮
brn: branchn;
end:
```

Diese Struktur wird mit Hilfe des Algorithmus `branch_detect` rekonstruiert, der anhand der Sprungziele die einzelnen Teile erkennen und auftrennen kann.

Wir prüfen in der Bedingung (*condition*) unter welchen Voraussetzungen wir welchen Zweig ausführen und schränken so die Menge der möglichen Zustände für jeden Zweig ein. Dabei müssen wir beachten, dass jeder Zweig durchlässig sein kann, d.h. nach dem abarbeiten eines Zweiges wird auch der nächste ausgeführt, d.h. es gibt keinen Sprung an das Ende der Verzweigung. Wir berechnen für jeden Zweig br_i mit $1 \leq i \leq n$ ein $\rho_{br_i} \in PC \times Env_R \times Env_M$.

Zum Auswerten der Bedingung benötigen wir zwei XRTL Konstrukte, die bedingten Ausdrücke (siehe Kapitel 2).

$$\texttt{(if\_then\_else } bed\ then\ else\texttt{)}$$

$$\texttt{(cond [}bed_1\ value_1\ \texttt{...}bed_n\ value_n\texttt{]}\ default\texttt{)}$$

In der Praxis (siehe Beispiele, Kapitel 2) treten diese meist in Kombination mit anderen Anweisungen auf, wie

$$\texttt{(insn x u y (set }(R_n)\texttt{ (compare }(E_1)^t\ (E_2)^t\texttt{)))}$$

$$\texttt{(jump\_insn y x z (set (pc) (if\_then\_else (}op\ (R_n)\ (c)\texttt{))}\ (l_1)\ (l_2)\texttt{)))}$$

wobei l_1 und l_2 Sprungziele sind, op ein Vergleichsausdruck und (c) ein ganzzahlige Konstante, die den Wert -1, 0 oder 1 hat. Wir werden diese zu einer Instruktion verschmelzen, für die wir das Register (R_n) nicht mehr benötigen. Wir erhalten dann folgende Instruktion

$$\texttt{(jump\_insn y x z (set (pc) (if\_then\_else (}op'\ (E_1)^t\ (E_2)^t\texttt{)}\ (l_1)\ (l_2)\texttt{)))}$$

mit einem modifizierten Vergleichsausdruck op' der von op und (c) abhängt. Wir verwenden folgende Übersetzungstabelle für op'.

(c)	op									
	eq	ne	gt	gtu	lt	ltu	ge	geu	le	leu
-1	–	–	ge	geu	–	–	–	–	lt	ltu
0	eq	ne	gt	gtu	lt	ltu	ge	geu	le	leu
1	–	–	–	–	le	leu	gt	gtu	–	–

Wir markieren mit „–“ Einträge die keinen Sinn machen und in der Praxis auch nicht vorkommen. Wir betrachten nun die Bedingung

$$(bed) \stackrel{\text{def}}{=} (\, op' \;\; (E_1)^t \;\; (E_2)^t \,)$$

als Ungleichung. Sei nun $\bigcup_i A_i$ die Menge aller Register- und Speicherausdrücke aus (bed) für die sich diese Ungleichung derart umformen lässt, dass

$$[\![A_i]\!]^t \rho \;\; op_i \;\; X_i$$

mit $X_i \in I_n^*(t)$ und $op_i \in \{<, >, \leq, \geq, =\neq\}$. X_i und op_i sind abhängig vom gewählten Teilausdruck A_i, der Bedingung (bed) und der notwendigen Umformung.

Beispiel 5.2.1 Die Bedingung $i + j < z + 10$ lässt sich folgendermaßen umformen:

- $i < z + 10 - j$
- $j < z + 10 - i$
- $z > i + j - 10$

Wir können nun feststellen, für welche Elemente x aus dem Intervall für z ein Element y aus $(i + j - 10)$ existiert mit $x > y$.

Wir verwenden die nachfolgende Funktion g_{op} um festzustellen, welches der Elemente eines Intervalls eine gegebene Bedingung erfüllt.

$$g_{op}^t \;:\; t \times I_n^*(t) \longrightarrow \{0, 1\}$$

mit $op \in \{<, >, \leq, \geq, =, \neq\}$ und

$$g_{op}^t \; (a, \underbrace{([x_i, y_i])_{i=1}^n}_{X_i}) \stackrel{\text{def}}{=} \begin{cases} 1, \text{ falls } \exists\, x \in X_i \text{ mit } a \; op \; x \\ 0, \text{ sonst} \end{cases}$$

Wir können nun die Funktionen $restrict$ definieren, die den Wertebereich einer Variablen oder eines Speicherstücks gemäß der erhaltenen Ungleichungen einschränkt.

$$
\begin{aligned}
&restrict^t \ : \ I_n^*(t) \times g^{op} \times I_n^*(t) \longrightarrow I_n^*(t) \\
&restrict^t \ \left(\left(\left[a_i, b_i\right]\right)_{i=1}^n, g^{op_i}, X_i\right) \ \stackrel{\text{def}}{=} \ \left(\left[a_i', b_i'\right]\right)_{i=1}^n
\end{aligned}
$$

wobei

$$
\forall x \in \left(\left[a_i', b_i'\right]\right)_{i=1}^n \text{ gilt } x \in \left(\left[a_i, b_i\right]\right)_{i=1}^n \ \wedge \ g^{op_i}\left(x, X_i\right) = 1
$$

Wir können nun ρ einschränken für den Fall, dass die Vergleichsbedingung erfüllt ist oder nicht. Ist diese Bedingung immer erfüllt oder nicht erfüllbar, so verhält sich die Verzweigung wie ein unbedingter Sprung zu (l_1) bzw. (l_2). Wir gehen im folgenden davon aus, dass beide Zweige erreichbar sind. Für die Instruktion c mit

$$
c = \texttt{(jump\_insn y x z (set (pc) (if\_then\_else } (bed) \ (l_1) \ (l_2)\texttt{)))}
$$

und der Menge aller Ungleichungen $\bigcup_i (A_i \ op_i \ X_i)$, die sich aus der Bedingung (bed) ableiten lassen, ergibt sich

$$
post_c^{\#}\left(pc, \rho\right) \ \stackrel{\text{def}}{=} \ \left(l_1, \rho^{true}\right) \cup \left(l_2, \rho^{false}\right)
$$

wobei

$$
\begin{aligned}
\rho^{true} \ &\stackrel{\text{def}}{=} \ \rho\left[\forall i \ \left(A_i \ \longrightarrow \ restrict^t\left(\left[\!\left[A_i\right]\!\right]^t \rho, g^{op_i}, X_i\right)\right)\right] \\
\rho^{false} \ &\stackrel{\text{def}}{=} \ \rho\left[\forall i \ \left(A_i \ \longrightarrow \ restrict^t\left(\left[\!\left[A_i\right]\!\right]^t \rho, g^{\overline{op}_i}, X_i\right)\right)\right]
\end{aligned}
$$

und $\overline{op}_i$ die zu op_i inverse Vergleichsoperation ist. Wir fügen nun die Paare (l_1, ρ^{true}) und (l_2, ρ^{false}) in die Prioritätswarteschlange (siehe Abbildung 5.1) ein. Die Einschränkungen lassen sich durch weiteres Ersetzen von Teilausdrücken der Vergleichsbedingung (bed) weiter präzisieren.

Die Instruktion c mit

$$
c = \texttt{(jump\_insn y x z (set (pc) (cond [}bed_1 \ \ l_1 \ \ \texttt{...}bed_n \ \ l_n\texttt{]} \ \ l_{def}\texttt{)}
$$

entspricht einer verallgemeinerten Einfachverzweigung, wird aber in der Praxis kaum verwendet. Die Bedingungen (bed_1) bis (bed_n) werden nacheinander geprüft. Falls die Bedingung (bed_i) erfüllt ist, wird (l_i) zurückgeliefert. Falls keine der Bedingungen erfüllt ist, so liefert sie (l_{def}) zurück. Jede der Bedingungen wird so umgeformt, wie wir es für einfache Verzweigungen vorgeführt haben. Wir erhalten nun

$$
post_c^{\#}\left(pc, \rho\right) \ \stackrel{\text{def}}{=} \ \left(l_1, \rho_1^{true}\right) \cup \cdots \cup \left(l_n, \rho_n^{true}\right) \cup \left(l_{def}, \rho_{def}\right)
$$

mit

$$
\begin{array}{rcl}
\rho_1^{true} & \stackrel{\text{def}}{=} & \rho\left[\forall i_1 \ \left(A_{i_1} \ \longrightarrow \ restrict^t\left(\llbracket A_{i_1} \rrbracket^t \rho, g^{op_{i_1}}, X_{i_1}\right)\right)\right] \\
\rho_1^{false} & \stackrel{\text{def}}{=} & \rho\left[\forall i_1 \ \left(A_{i_1} \ \longrightarrow \ restrict^t\left(\llbracket A_{i_1} \rrbracket^t \rho, g^{\overline{op}_{i_1}}, X_{i_1}\right)\right)\right] \\
 & \vdots & \\
\rho_n^{true} & \stackrel{\text{def}}{=} & \rho_{n-1}^{false}\left[\forall i_n \ \left(A_{i_n} \ \longrightarrow \ restrict^t\left(\llbracket A_{i_n} \rrbracket^t \rho, g^{op_{i_n}}, X_{i_n}\right)\right)\right] \\
\rho_n^{false} & \stackrel{\text{def}}{=} & \rho_{n-1}^{false}\left[\forall i_n \ \left(A_{i_n} \ \longrightarrow \ restrict^t\left(\llbracket A_{i_n} \rrbracket^t \rho, g^{\overline{op}_{i_n}}, X_{i_n}\right)\right)\right] \\
\rho_{def} & \stackrel{\text{def}}{=} & \rho_n^{false}
\end{array}
$$

wobei $\bigcup_{i_j} \left(A_{i_j} \ op_{i_j} \ X_{i_j}\right)$ die Menge alle Ungleichungen sei, die sich aus der Vergleichsbedingung (bed_j) ergeben. Wir fügen nun für jede erfüllbare Bedingung (bed_i) das Paar (l_i, ρ_i^{true}) in die Prioritätswarteschlange ein. Wenn alle Bedingungen auch nicht erfüllbar sind, dann fügen wir (l_{def}, ρ_{def}) ein.

Wir können nun mit Hilfe der vorgestellten $post^{\#}$ die Bedingung einer Verzweigung abarbeiten. Hier können drei verschiedenen Sprungarten auftreten, der Sprung zu einer anderen Teilbedingung, der zu einem bestimmten Zweig und der ans Ende der Verzweigung. Nachdem die Bedingung ausgewertet ist, existiert in der Prioritätswarteschlange für jeden erreichbaren Zweig br_i mit $1 \leq i \leq n$ ein Paar (br_i, ρ_{br_i}) und (end, ρ_{end}). Wir behandeln Verzweigungen in der konkreten Implementierung wie einfache Instruktion und erreichen dadurch eine vereinfachte Handhabung von verschachtelten Verzweigungen. Diese ist dann nur noch eine Instruktion innerhalb eines Zweigs einer Verzweigung.

5.2.4 Schleifen

Wir versuchen zuerst mit Hilfe einer einfachen Methode Schleifen zu analysieren, indem wir die Schleife durchiterieren. Die maximal erlaubte Anzahl von Iterationen ist beschränkt. Wir erhalten so für Schleifen mit kleiner Anzahl von Durchläufen präzise Ergebnisse. Um jedoch Schleifen zu analysieren, für die die iterative Methode keine Ergebnisse liefert, müssen wir den Begriff der Fixpunkte einführen.

Ein Fixpunkt einer Funktion f beschreibt ein Element x für das $f(x) = x$ gilt. Wir bezeichnen mit lfp (f) den kleinsten und mit gfp (f) den größten Fixpunkt.

Satz 5.2.2 *(Tarski) Eine Menge von Fixpunkten einer monotonen Funktion f eines vollständigen Verbandes $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ ist ein nichtleerer, vollständiger Verband der durch $\sqsubseteq$ geordnet ist. Der kleinste und größte Fixpunkt von f sieht folgendermaßen aus*

$$
\begin{array}{rcl}
\mathit{lfp}\,(f) & = & \sqcup\{x \in L \mid f(x) \sqsubseteq x\} \\
\mathit{gfp}\,(f) & = & \sqcap\{x \in L \mid x \sqsubseteq f(x)\}
\end{array}
$$

Beweis: siehe [36] ■

Wir finden bei der Analyse von Schleifen nach endlich vielen Schritten einen Fixpunkt, da der verwendete abstrakte Domain endlich ist. Um diese Berechnungen zu beschleunigen verwenden wir den Widening/Narrowing Ansatz aus [11, 12, 13] mit Operatoren, die speziell für unsere Implementierung angepasst wurden.

Definition 5.2.3 *(Widening) Ein Widening Operator ist eine Funktion* $\triangledown : L \times L \longrightarrow L$*, so dass für alle* $x, y \in L$ *gilt*

$$x \sqsubseteq x \triangledown y \qquad\qquad y \sqsubseteq x \triangledown y$$

Für alle monoton aufsteigenden Ketten $x_0 \sqsubseteq x_1 \sqsubseteq, \ldots$ *gilt, dass die entsprechende Kette* $y_0, y_1, \ldots$ *mit* $y_0 = x_0$ *und* $y_{i+1} = y_i \triangledown x_{i+1}$ *nicht streng monoton ansteigend ist.*

Proposition 5.2.4 *Für eine monotone Funktion* f *und einen Widening Operator* $\triangledown$ *ist die Sequenz*

$$\begin{aligned} x_0 &= \bot \\ x_{i+1} &= \begin{cases} x_i, \textit{falls } f(x_i) \sqsubseteq x_i \\ x_i \triangledown f(x_i), \textit{sonst} \end{cases} \end{aligned}$$

stabil und ist größer oder gleich lfp (f)*.*

Beweis: Siehe [12, 13] ■

In der Praxis führt das Widening zu einem zu hohen Präzisionsverlust. Die Verwendung eines Narrowing Operators erlaubt uns diesen Verlust zu begrenzen.

Definition 5.2.5 *(Narrowing) Ein Narrowing Operator ist eine Funktion* $\triangle : L \times L \longrightarrow L$*, so dass für alle* $x, y \in L$*,* $x \sqsubseteq y$ *impliziert, das* $y \sqsubseteq x \triangle y \sqsubseteq y$ *ist.*

Für alle monoton fallenden Ketten $x_0 \sqsupseteq x_1 \sqsupseteq \ldots$ *ist die entsprechende Kette* $y_0, y_1, \ldots$ *mit* $y_0 = x_0$ *und* $y_{i+1} = y_i \triangle x_{i+1}$ *nicht streng monoton fallend.*

Proposition 5.2.6 *Sei* f *eine monotone Funktion und* $\triangle$ *ein Narrowing Operator. Falls* $x_0 \sqsupseteq$ *lfp* (f) *und* $f(x_0) \sqsubseteq x_0$ *so ist die Sequenz* $x_{i+1} = x_i \triangle f(x_i)$ *monoton fallend, stabil und größer oder gleich lfp* (f)

Beweis: Siehe [12, 13] ■

In [5, 11, 12, 13] finden sich Beispiele für konkrete Widening und Narrowing Operatoren. Wir verwenden ein Widening mit direkt nachfolgendem Narrowing. Wir sprechen von einem eingeschränkten Widening.

Wir vereinfachen die Analyse der drei auftretenden Schleifentypen (siehe Abschnitt 2.3) indem wir die for und do-while Schleife in eine while Schleife überführen.

Die nachfolgende **for Schleife**

```
for (start; condition; update)
  { body }
```

wird folgendermaßen in eine while Schleife übersetzt.

```
start;
while (condition)
  { body; update }
```

Eine **do-while Schleife**

```
do {
  body
} while (condition)
```

wird folgendermaßen in eine while Schleife übersetzt.

```
body;
while (condition)
  { body }
```

Wir müssen uns nur noch mit der Analyse von while Schleifen beschäftigen. Diese haben die folgende Form:

```
while (condition)
  { body }
```

5.2.4.1 Widening Kandidaten

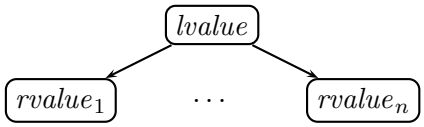
Nachdem die unterschiedlichen Schleifentypen auf einen zurückgeführt wurden, müssen wir uns mit den Registern oder Speicherausdrücken befassen, die für ein Widening in Frage kommen. Wir versuchen die Anzahl der Kandidaten zu reduzieren, indem wir uns die Abhängigkeiten in einfachen XRTL Instruktionen, sowie Verzweigungen und Schleifen anschauen. Wir unterscheiden zwischen veränderten und nicht veränderten Teilausdrücken.

Bemerkung 5.2.7 Wir bezeichnen im folgenden den Register oder Speicherausdruck, der in einer Instruktion verändert wird als $lvalue$-Ausdruck und jeden der Teilausdrücke (Register und Speicher) die zur Veränderung beitragen als $rvalue$-Ausdruck.

Alle Register und Speicherausdrücke, die in den Instruktionen des Schleifenrumpfes verändert werden, sind Kandidaten für das Widening. Wir schränken diese Liste mit Hilfe von gerichteten Abhängigkeitsgraphen von $lvalue$-Teilausdrücken ein. Für einfache Instruktionen der Form

$$\texttt{(insn x y z (set } (lvalue)\ (rvalue)\texttt{))}$$

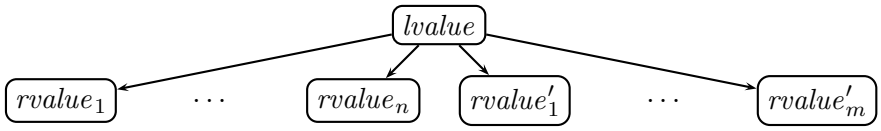
sieht ein Abhängigkeitsgraph folgendermaßen aus:



Die Ausdrücke $(rvalue)_1, \ldots, (rvalue)_n$ sind Teilausdrücke von $(rvalue)$ die zu der Menge der Widening Kandidaten gehören. Wir erzeugen für jede Instruktion des Schleifenrumpfs einen Abhängigkeitsgraphen unter Berücksichtigung des vorhergehenden Graphen. Wir fügen so einzelne Graphen zusammen, müssen dabei allerdings beachten, dass es zu gegenseitigen Abhängigkeiten (Zyklen im Graph) kommen kann.

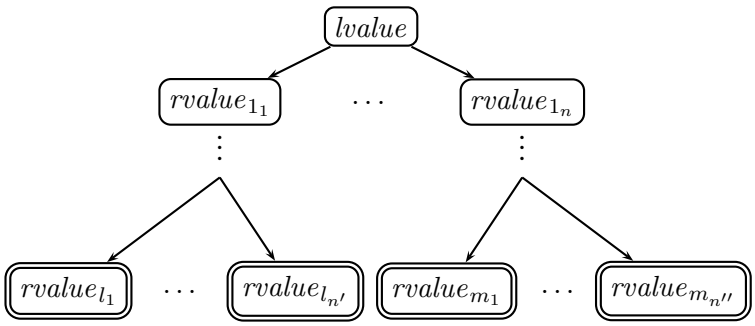
Für Verzweigungen innerhalb eines Schleifenrumpfes erstellen wir für jeden Zweig Abhängigkeitsgraphen. Sollten innerhalb verschiedener Zweige Graphen mit gleicher Wurzel auftreten,

so werden diese zu einem Graphen verschmolzen.



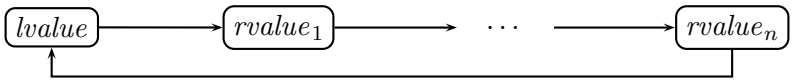
Bei Funktionsaufrufen gibt es einen Zusammenhang zwischen den Parametern und dem Rückgabewert. Dieser ist jedoch abhängig von der aufgerufenen Funktion, kann also vorberechnet werden. Wir müssen hierbei allerdings beachten, das Parameter, die von der Funktion verändert werden, den Graphen beeinflussen. Wir wenden diese Methoden rekursiv für verschachtelte Verzweigungen und Schleifen an.

Für Abhängigkeitsgraphen, die keine Zyklen enthalten, wie



sind nur die Teilausdrücke mit Ausgangsgrad 0, d.h. die Blätter interessant, weil alle anderen Teilausdrücke (Knoten) von einem oder mehreren anderen abhängig sind. Wir können nun die Menge der Kandidaten für das Widening auf alle Teilausdrücke einschränken die als Blatt in einem Abhängigkeitsgraphen auftreten.

Existiert jedoch ein zyklischer Teilgraph der Form

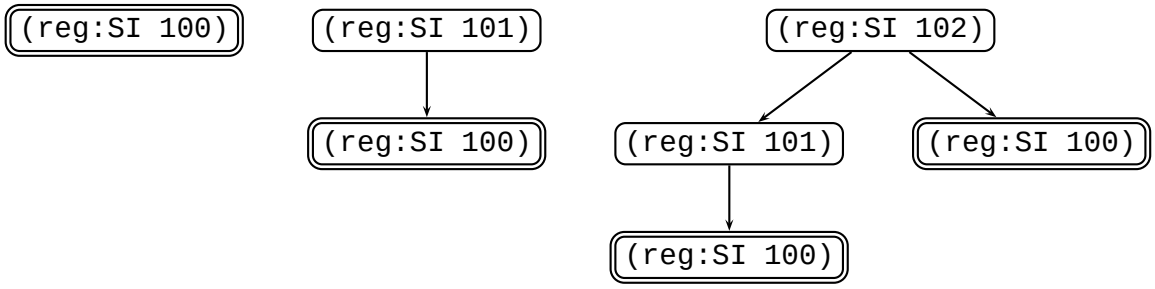


auf, können wir uns einen der am Zyklus beteiligten Teilausdrücke aussuchen und alle anderen aus der Menge der Widening Kandidaten entfernen. Wir verwenden vorzugsweise einen Ausdruck der innerhalb der Schleifenbedingung verwendet wird.

Beispiel 5.2.8 Der folgende XRTL Schleifenrumpf

```
(set (reg:SI 100) (const_int:SI 10))
(set (reg:SI 101) (add:SI (reg:SI 100) (const_int:SI 2)))
(set (reg:SI 102) (add:SI (reg:SI 100) (reg:SI 101)))
```

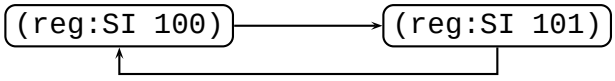
erzeugt die Abhängigkeitsgraphen



Wir können aus den Graphen sehr leicht ablesen, dass in diesem Beispiel nur der Ausdruck `(reg:SI 100)` für das Widening interessant ist. Dieser XRTL Code

```
(set (reg:SI 100) (reg:SI 101))
(set (reg:SI 101) (reg:SI 100))
```

erzeugt einen zyklischen Graphen



Wir können uns nun für einen der beiden Teilausdrücke entscheiden.

5.2.4.2 Widening Operator

Der Widening Operator wird auf die Register und Speicherausdrücke angewendet, die wir durch die vorhergehende Abhängigkeitsanalyse erhalten haben. Wir verwenden für die Implementierung des Widening Operators die Einschränkungen, die sich aus dem Bedingungsteil der while Schleife ergeben. Die Schleifenbedingung besteht aus einer oder mehreren bedingten Anweisungen. Wir teilen diese mit ihren Bedingungen in zwei Gruppen Bed_1 und Bed_2 ein, in die, die auf jeden Fall erfüllt sein müssen damit die Schleife fortgeführt werden kann und die anderen.

Beispiel 5.2.9 Die Teilbedingungen der folgenden Bedingungen werden in die beiden vorgestellten Gruppen eingeteilt.

1. `((x < 10) and (y < 20))`
   $\Rightarrow$ `(x < 10)` und `(y < 20)` gehören zur ersten Gruppe.
2. `((x < 10) or (y < 20))`
   $\Rightarrow$ `(x < 10)` und `(y < 20)` gehören zur zweiten Gruppe.
3. `((x < 10) and ((y < 10) or (y > 20)))`
   $\Rightarrow$ `(x < 10)` gehört zur ersten Gruppe, `(y < 10)`, `(y > 20)` zur zweiten Gruppe.

Wir unterscheiden die entsprechenden bedingten XRTL-Ausdrücke durch eine Analyse der Sprungziele. Alle Teilbedingungen gehören zur ersten Gruppe, wenn eines der Sprungziele außerhalb der Schleife liegt. Erfolgt hingegen ein Sprung zu einer anderen Teilbedingung, so ist diese in Gruppe zwei. Ist deren Sprungziel außerhalb der Schleife, so kann deren nachfolgende Teilbedingung in der ersten oder zweiten Gruppe sein. Mit Hilfe dieser einfachen Regeln können wir die bedingten XRTL-Ausdrücke in die vorgestellten Gruppen aufteilen.

Wir erzeugen für jeden Ausdruck eine Ungleichung wie in Abschnitt 5.2.4 für Verzweigungen und definieren die entsprechenden Funktionen g^t_{op} und $restrict^t$. Für zwei aufeinanderfolgende Umgebungen ρ' und ρ sei

$$
\begin{aligned}
[\![A]\!]\rho' &\stackrel{\text{def}}{=} \left(\left[a_i, b_i\right]\right)_{i=1}^{n} \in I_n^*(t) \\
[\![A]\!]\rho &\stackrel{\text{def}}{=} \left(\left[c_j, d_j\right]\right)_{j=1}^{n} \in I_n^*(t)
\end{aligned}
$$

und

$$
\begin{aligned}
a_{min} &\stackrel{\text{def}}{=} \min\left\{x | x \in \left(\left[a_i, b_i\right]\right)_{i=1}^{n}\right\} \\
a_{max} &\stackrel{\text{def}}{=} \max\left\{x | x \in \left(\left[a_i, b_i\right]\right)_{i=1}^{n}\right\} \\
c_{min} &\stackrel{\text{def}}{=} \min\left\{x | x \in \left(\left[c_j, d_j\right]\right)_{j=1}^{n}\right\} \\
c_{max} &\stackrel{\text{def}}{=} \max\left\{x | x \in \left(\left[c_j, d_j\right]\right)_{j=1}^{n}\right\}
\end{aligned}
$$

Wir definieren den Widening Operator wie folgt:

$$\nabla \; : \; I_n^*(t) \times I_n^*(t) \longrightarrow I_n^*(t)$$

$$\nabla \left(\left(\left[a_i, b_i \right] \right)_{i=1}^n, \left(\left[c_j, d_j \right] \right)_{j=1}^n \right) \stackrel{\text{def}}{=}$$
$$\begin{cases} \left(\left[a_i, b_i \right] \right)_{i=1}^n \bigsqcup \left(\left[c_j, d_j \right] \right)_{j=1}^n \\ \qquad \text{falls } giant_l = false \text{ und } giant_r = false \\ restrict^t \left([a', b'], A \right) \bigsqcup \left(\left[a_i, b_i \right] \right)_{i=1}^n \bigsqcup \left(\left[c_j, d_j \right] \right)_{j=1}^n, \text{ sonst} \end{cases}$$

wobei

$$a' \stackrel{\text{def}}{=} \begin{cases} -\infty, \text{ falls } c_{min} < a_{min} \text{ und } giant_l = true, giant_l := false \\ c_{min}, \text{ falls } c_{min} < a_{min} \text{ und } giant_l = false \\ a_{min}, \text{ sonst} \end{cases}$$

$$b' \stackrel{\text{def}}{=} \begin{cases} +\infty, \text{ falls } c_{max} > a_{max} \text{ und } giant_r = true, giant_r := false \\ c_{max}, \text{ falls } c_{max} > a_{max} \text{ und } giant_r = false \\ a_{max}, \text{ sonst} \end{cases}$$

Die Funktion $restrict^t$ mit

$$restrict^t \; : I_n(t) \times (Env_R \cup Env_M) \longrightarrow I_n^*(t)$$
$$restrict^t \; \left([a', b'], A \right) \stackrel{\text{def}}{=} \bigsqcap_{(A \; op \; X) \; \in \; Bed_1} restrict^t([a', b'], g^{op}, X) \sqcup \bigsqcup_{(A \; op' \; X') \; \in \; Bed_2} restrict^t([a', b'], g^{op'}, X')$$

schränkt das Intervall $[a', b']$ ein, mit Hilfe der aus den bedingten XRTL Anweisungen abgeleiteten Ungleichungen, die wir in zwei Gruppen Bed_1 und Bed_2 eingeteilt haben.

Der definierte Widening Operator ∇ kann in jede Richtung ($\pm\infty$) einen großen Schritt (giant step) ausführen, der allerdings durch die Funktion $restrict^t$ eingeschränkt wird. Danach werden nur noch kleine Schritte (baby step) ausgeführt. Wir verwenden einen Widening Operator, der die Idee des Narrowing umsetzt, indem wir ein beschränktes Widening für diejenigen Register- und Speicherausdrücke durchführen, für die wir aus der Schleifenbedingung entsprechende Restriktionen ableiten können. Für die Widening/Narrowing Kandidaten, für die wir diese Einschränkungen nicht haben führt auch ein Narrowing Operator zu keinerlei Verbesserungen. Die Abbildung 5.2 zeigt, wie wir den Fixpunkt einer while Schleife mit Hilfe des eingeführten Widening Operators berechnen. Das Widening führen wir erst für $i > 1$ durch. Die Analyse der Schleifenbedingung liefert uns auch die Umgebung ρ_{end} mit der wir nach der Analyse der Schleife, die mit der Berechnung eines Fixpunktes beendet ist, fortfahren.

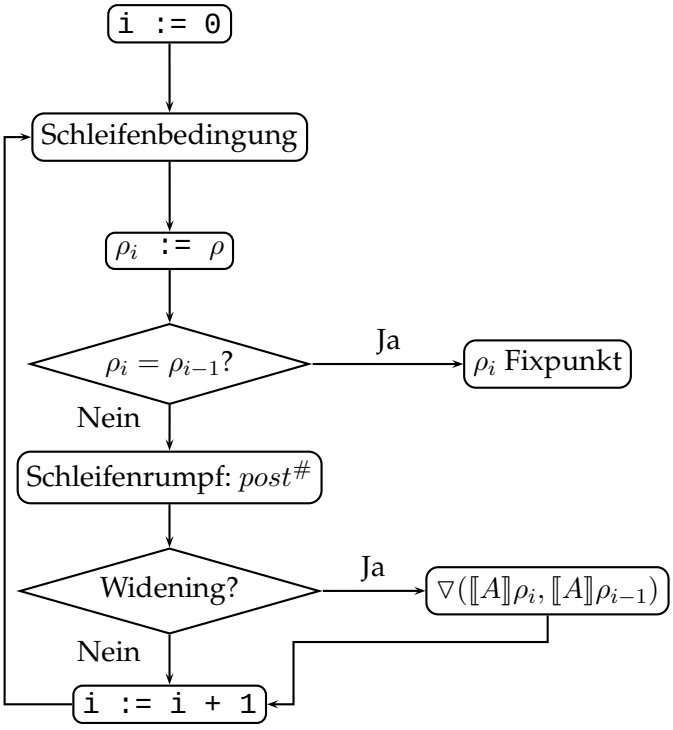


Abbildung 5.2: Fixpunkt Berechnung

5.2.5 Funktionsaufrufe

Wir führen keine klassische interprozedurale Analyse durch, wie sie in [29] beschrieben wird, sondern führen die Analyse von Funktionsaufrufen mit Hilfe von Vorberechnungen der Veränderungen an der Umgebung ρ durch die aufgerufene Funktion unter Berücksichtigung der Parameter durch. Wir können für Funktionen mit gegebenem XRTL Code das so genannte Inlining anwenden. Wir verwenden hierzu die Funktion $fcall$:

$$fcall \ : \ state \times Fn \times Args \longrightarrow state$$
$$fcall\,(\rho, name, args) \ \stackrel{\text{def}}{=} \ \rho'$$

Fn bezeichnet hier den Funktionsnamen und $Args$ die Liste der Funktionsparameter. In Abschnitt 2.4 ist der Funktionsaufruf und die Übergabe der Parameter in XRTL genauer beschrieben. Für Funktionen mit Rückgabewert benötigen wir $freturn$ die abhängig von Parametern und Umgebung den Rückgabewert berechnet.

$$freturn \ : \ state \times Fn \times Args \longrightarrow I_n^*(t)$$
$$freturn\,(\rho, name, args) \ \stackrel{\text{def}}{=} \ r$$

Wir sind nun in der Lage einen Funktionsaufruf abzuarbeiten. Wir erhalten

$$[\![\texttt{(call } (fn) \ \ldots\texttt{)}]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} \bot, \text{ falls kein Rückgabewert existiert} \\ freturn(\rho, fn, args), \text{ sonst} \end{cases}$$

und

$$\rho \stackrel{\text{def}}{=} fcall(\rho, fn, args)$$

Die hier für den Funktionsaufruf durch „. . . “ angegebenen zusätzlichen Argumente sindmaschinenabhängigg, aber für unsere Analyse nicht notwendig, da es sich hierbei nicht um die Funktionsparameter handelt. Wir müssen beachten, dass bei Funktionen, die komplexe Datenstrukturen zurückliefern, der Rückgabewert ein Parameter der Funktion ist.

In diesem Kapitel haben wir die Abstrakte Interpretation von XRTL Instruktionen beschrieben. Um einfache Instruktionen abzuarbeiten haben wir die Bedeutung der einzelnen RTL Teilausdrücke angegeben. Wir haben gezeigt, wie Verzweigungen abgearbeitet werden und wie die Menge von Zuständen für jeden Zweig eingeschränkt werden kann. Schleifen analysieren wir durch Iteration oder die Berechnung von Fixpunkten. Obwohl der abstrakte Domain endlich ist haben wir das Widening/Narrowing eingeführt um die Berechnung von Fixpunkten zu beschleunigen. Wir haben entsprechende, an XRTL angepasste, Widening Operatoren eingeführt und beschäftigten uns anschließend mit der Analyse von Funktionsaufrufen.

Kapitel 6

Das Analysetool xGCC

In diesem Kapitel werden wir andere statische und nicht-statische Analysetools kurz vorstellen. Wir beschreiben unseren Ansatz, der aus einer modifizierten GNU Compiler Collection (GCC) und einem separaten Analysetool xGCC besteht, welches die vom Compiler erzeugte XRTL Zwischensprache analysiert. Wir beschreiben die auftretenden Schwierigkeiten und begründen die getroffenen Designentscheidungen. Wir gehen kurz auf die, für das Analysetool xGCC zentralen Datenstrukturen ein. Wir veranschaulichen mit Hilfe ausgewählter C, C++ und Fortran 77 Programmbeispiele, welche Klassen von Fehlern das Analysetool xGCC entdeckt. Wir führen Experimente mit Beispielprogrammen aus den *Numerical Recipes in C* um die Laufzeit der Analyse zu testen. Wir stellen einige Verbesserungen und alternative Implementierungen vor, mit denen wir die Laufzeit der Analyse deutlich reduzieren können.

6.1 Andere Analysetools

Klassische, nicht-statische Analysetools, wie z.B. valgrind [52] oder das kommerzielle purify [30] eigenen sich hervorragen zum Auffinden von Fehlern in sehr großen Programmen unter der Voraussetzung, dass der aktuelle Programmablauf einen Fehler erzeugt. Mit valgrind, welches leider nur für x86/Linux verfügbar ist, werden unter anderem viele bekannte KDE und Gnome Applikationen, Mozilla, OpenOffice und Opera getestet. Das kommerzielle purify wurde für die Tests im LiDIA-Projekt [46] verwendet. Diese Programme haben ihre Vorzüge bei bekannten und leicht reproduzierbaren Fehlern. Sie erlauben die Nutzung aller Sprachkonstrukte der unterstützten Programmiersprachen und Compiler. Der so genannte GCC-Stackprotector [51] soll ein Ausnutzen eines Stackzugriffsfehlers verhindern. Hier werden die Symptome behandelt, nicht das eigentliche Problem.

Analysetools wie SLAM [4, 50], BLAST [19, 43] und CBMC [10, 44] sind auf die Programmiersprache C beschränkt und verwenden einen Model Checking Ansatz. SLAM verwendet Predicate Abstraction. Aus einem C Programm wird ein abstraktes boolsches Programm erzeugt mit mit Hilfe einer festen Anzahl von Prädikaten und unter Verwendung von boolschen Variablen für jedes Prädikat. BLAST verwendet das Lazy Abstraction. Das abstrakte

Modell wird nach Bedarf des Model Checkers konstruiert und verfeinert. CBMC implementiert das Bounded Model Checking und verwendet einen SAT-Solver um die Erfüllbarkeit der resultierenden boolschen Formeln zu überprüfen. Diese Analysetools beschränken sich auf eine Teilmenge der Programmiersprache C, d.h. einige der vom Compiler erlaubten Konstrukte sind verboten.

Das Analysetool CMC [18, 25, 45] verwendet ebenfalls einen Model Checking Ansatz und durchsucht große Zustandsräume effizient, indem es Zustände speichert. Es wird keine separate Modellbeschreibung benötigt, da CMC direkt mit dem C oder C++ Code arbeitet. Die Probleme, die durch eine fehlerhafte Modellbeschreibung entstehen, werden so vermieden. Ein weiterer Vorteil ist, dass sich Änderungen in der Implementierung automatisch auf das Modell auswirken. CMC wurde u.a. erfolgreich bei der Suche nach Fehlern in der Implementierung von Netzwerkprotokollen eingesetzt.

Das Sable/Soot System [38, 49] beschränkt sich auf Java. Hier werden verschiedene Analysemethoden und Repräsentationen verwendet um den Java Bytecode zu optimieren. Im Gegensatz dazu arbeitet das Bandera System [41] mit dem Java Quelltext. Aus diesem wird ein Modell des Programms erzeugt in der Sprache eines der unterstützten Verifikationstools. Die Ausgabe des Tools kann wieder mit dem entsprechenden Java Quelltext verknüpft werden. Syntox [8, 42] ist kein Model Checker, sondern implementiert die abstrakte Interpretation von Pascal Programmen mit Hilfe der Intervallapproximation. Ein Widening/Narrowing Ansatz wird verwendet.

Einen anderen Ansatz verwendet das CCured/CIL System [26, 27, 47]. Ein eingeschränktes C wird als Zwischensprache CIL verwendet. Diese Zwischensprache wird für die Programmanalyse verwendet. CCured fügt notwendige Laufzeitchecks in das Programm ein, welche sich aus der dazugehörigen Programmanalyse ergeben. Tritt beim Ablauf des Programms ein Fehler auf, dann stoppt es, womit das Ausnutzen solcher, möglicherweise sicherheitskritischer Fehler so verhindert wird. Dieser Overhead kostet aber etwa $10 - 60\%$ an Geschwindigkeit, was in einigen Bereichen inakzeptabel ist. Eine Anpassung des Quelltextes ist in vielen Fällen nötig.

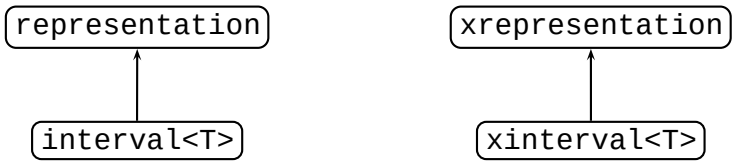
Das PAG System [1, 24, 37, 48] erlaubt die automatische Generierung von Programmanalysatoren aus einer Spezifikationen. Eine eigens entwickelte funktionale Sprache wird verwendet um die Transferfunktion zu spezifizieren. Diese Transferfunktion modelliert die Effekte der Programmanweisungen für die Analyse. Mit rekursiven Definitionen werden die abstrakten Wertebereiche der Analyse beschrieben. PAG verwendet abstrakte Interpretation und Datenflussanalyse und ist für imperative und logische Programmiersprachen geeignet. Die generierten Analysatoren, die in ANSI C implementiert sind, benötigen eine Reihe von Funktionen für den Zugriff auf den Kontrollflussgraphen und den abstrakten Syntaxbaum. Eine Erweiterung mit eigenen Datentypen und Funktionen ist möglich.

6.2 Compiler und Analysetool xGCC

Unser Analysetool xGCC verwendet die leicht modifizierte und erweiterte Zwischensprache XRTL, für die Übersetzer aus den Programmiersprachen C, C++, Java, Fortran 77 existieren. Die von uns durchgeführten Veränderungen beschränken sich auf wenige Dateien, die

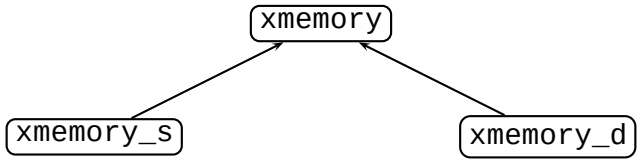
alle zum sprachunabhängigen Teil des Compilers gehören. Damit ist gewährleistet, dass das Analysetool nicht angepasst werden muss, wenn Programmiersprachen, bzw. Frontends für diese Programmiersprachen, neu hinzukommen oder verändert werden. Die aktuelle, von uns modifizierte Version der GCC ist 3.1.1. Begonnen haben wir mit Version 2.95.1, weshalb die in neueren Versionen hinzugekommenen Analysemethoden und Optimierungen, wie z.B. das *Static Single Assignment (SSA)* [9, 33], von uns nicht berücksichtigt werden. Die durchgeführten Modifikationen können wir in zwei Teile aufteilen. Im ersten Teil beschäftigen wir uns mit der Ausgabe der RTL Instruktionen. Hier haben wir Informationen, wie z.B. die eigentlichen Registernamen weggelassen, da diese für unsere Analyse nicht von Bedeutung sind. Wir beschränken uns auf die Registernummern. Für Fließkommazahlen, wurde ursprünglich ein maschinenunabhängiges Format, bestehend aus mehreren ganzen Zahlen angegeben. Diese können dann mit Hilfe einer Funktion wieder in eine Fließkommazahl umgewandelt werden. Wir haben diese Funktion verwendet um den Fließkommawert auszugeben. Die RTL Anweisungen wurden für eine Einlesen und Parsen mit Bison/Flex optimiert. Wir arbeiten nicht auf den internen Strukturen, da dieses Vorgehen ein tieferes Verständnis der Arbeitsweise der GCC voraussetzt. Dies war aufgrund des höheren Zeitaufwands und der unzureichenden Dokumentation nicht möglich. Der zweite Teil beschäftigt sich mit den für unser Analysetool xGCC benötigten Zusatzinformationen, die der Einfachheit halber in einer separate Datei speichern. Diese umfassen u.a. Informationen über globale Variablen und Konstanten sowie deren Startwerte. Hierzu zählen auch temporär verwendete Konstanten und Strings. Für jede Funktion sind Informationen über die Größe und Belegung des Stacks enthalten. Den Stack einer Funktionen können wir so rekonstruieren.

Das Analysetool xGCC liest die Zusatzinformationen vor den eigentlichen RTL Anweisungen ein. Dies hat den Vorteil, dass wir beim Einlesen die RTL Anweisungen direkt der entsprechenden Funktion zuordnen können. Um Akzeptanzprobleme bei Programmierern zu vermeiden, versuchen wir alle Programme, die der Compiler fehlerfrei übersetzt zu analysieren. Da wir aus diesem Grund alle RTL Anweisungen erlauben, muss die von uns verwendete Intervallarithmetik in der Lage sein, die entsprechenden Operationen auszuführen. Neben den einfachen arithmetischen Grundoperationen und Vergleichen gehören hierzu auch Bitoperationen und der Zugriff auf die Bytedarstellung. Nach einer ersten fehlerhaften Implementierung der Intervallarithmetik, haben wir die Begriffe des gültigen Intervalls und der Liste von gültigen Intervallen eingeführt. Erst diese eingeschränkten Definitionen ermöglichten es uns mit dem Problem der vorzeichenlosen und vorzeichenbehafteten Betrachtungsweise umzugehen und den Überlauf bzw. Unterlauf von arithmetischen Operationen korrekt einzuordnen. Der Zugriff auf die Bytedarstellung wird vereinfacht, weil wir durch einfache Überprüfungen der zu verändernden Bytes dafür sorgen, dass die Intervalldefinition nicht verletzt wird.



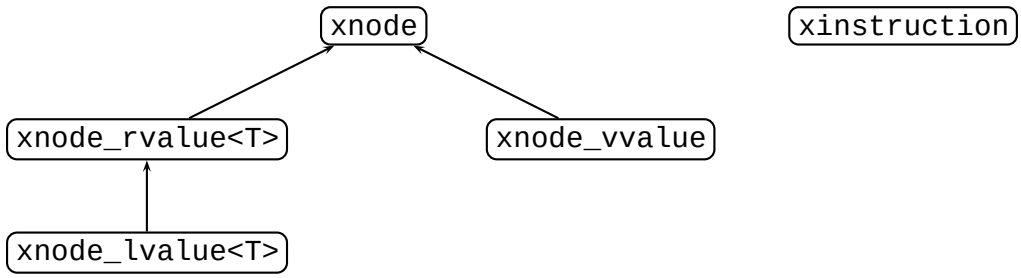
Die Template-Klasse interval<T> implementiert ein gültiges Intervall vom Typ T und wird

von der Klasse representation abgeleitet, die die dazugehörige Bytedarstellung repräsentiert. Die Bytedarstellung wird aus Effizienzgründen nur bei Bedarf berechnet. Die voneinander abgeleiteten Klassen xinterval<T> und xrepresentation implementieren eine Liste von gültigen Intervallen und die dazugehörige Liste mit Bytedarstellungen. Wir verwenden hierzu die entsprechenden Klassen interval<T> und representation. Die Implementierung der gültigen Intervalle, der Listen von gültigen Intervallen und der, für die XRTL Analyse notwendigen Operationen macht zusammengenommen ungefähr 30% der gesamten Implementierung aus, die mehr als 35000 Zeilen C++ Code umfasst. Ein weiterer wichtiger Teil der Implementierung ist die Struktur der Speicherstücke.



Wir verwenden eine virtuelle Basisklasse xmemory, die das für die Analyse benötigte Interface definiert. Davon abgeleitet ist die Klasse xmemory_s, die jedes Byte eines Speicherstücks und seinen möglichen Zustand abbildet. Die im Interface angegebenen Funktionen werden für diese interne Struktur implementiert. Die Klasse xmemory_d, die auch von xmemory abgeleitet ist, verwendet intern eine alternative Darstellung, welche eine Art Intervallbaum einsetzt um die Zugriffe auf das Speicherstück zu realisieren. Diese noch im Experimentierstadium befindliche Klasse eröffnet uns die Möglichkeit den Speicherbedarf zu beschränken, unabhängig von der eigentlichen Größe des zu approximierenden Speicherstücks. Wir arbeiten momentan an einer Beschränkung der Anzahl von Knoten im Intervallbaum, um so die Laufzeit für einen Zugriff auf ein Speicherstück zu beschränken.

Den größten Teil der Implementierung benötigen wir für das Einlesen der XRTL Instruktionen und die Analyse der XRTL Zwischensprache.



Wir erzeugen beim Einlesen für jeden RTL-Teilausdruck einen Knoten. Dabei unterscheiden wir drei Arten von Knoten, die *rvalue*-Ausdrücke, *lvalue*-Ausdrücke und die restlichen Teilausdrücke, die keinerlei Rückgabewert haben. Da jeder *lvalue*-Ausdruck auch ein *rvalue*-Ausdruck ist wird die dazugehörige Template-Klasse xnode_lvalue<T> von xnode_rvalue<T> abgeleitet. Für jeden auftretenden RTL Ausdruck existiert eine Klasse, die von xnode_rvalue<T>, xnode_lvalue<T> oder xnode_vvalue abgeleitet ist. Für jede RTL-Instruktion

leiten wir eine Klasse von xinstruction ab. Eine Instruktion besteht entweder aus einem Knoten, der von der Klasse xnode_vvalue abgeleitet ist, oder aus mehreren Teilinstruktionen, wie z.B. für Schleifen oder Verzweigungen. Die Berechnung von Fixpunkten und das Widening wird für Schleifen innerhalb der Klasse xinstruction_loop durchgeführt. Für Verzweigungen ist die Klasse xinstruction_branch verantwortlich. Wir haben den Programmanalysatorengenerator PAG nicht verwendet, weil XRTL nicht kompatibel mit der verwendeten Beschreibungssprache DATLA [37] ist. In XRTL wird nicht zwischen vorzeichenbehafteten und vorzeichenlosen ganzen Zahlen unterschieden. Außerdem ist der Zugriff auf die Bytedarstellungen von ganzen Zahlen und Fließkommazahlen erlaubt. Eine Verwendung von PAG mit den vorgestellten Datentypen für Listen von gültigen Intervallen und Speicherstücken ist möglich aber aufwendig.

6.3 Praktische Auswertung

6.3.1 Beispiele

Wir veranschaulichen den so genannten „array out of bound“ Fehler und die Anwendung des xGCC Analysetools. Dieser Fehler wird verursacht durch einen Zugriff auf ein Speicherstück mit zu kleinem oder zu großem Offset. Man bezeichnet diesen Fehler deshalb auch als „buffer overflow“ oder Speicherüberlauf. Ist das Speicherstück Teil des Stacks einer Funktion, dann kann diese Art von Fehlern von potentiellen Angreifern ausgenutzt werden und somit zu Sicherheitsproblemen führen.

Das Beispiel 6.3.1 zeigt eine fehlerhafte C++ Template Funktion. Dieses Fragment war Teil der LLL Implementierung der LiDIA Klassenbibliothek [46] und wurde von einer Vielzahl von Anwendern eingesetzt. Der Fehler ist mit üblichen Testverfahren für mehr als zwei Jahre unentdeckt geblieben. Mehr als 500000 Testläufe mit verschiedenen Eingabegittern unterschiedlicher Dimension wurden alleine im Rahmen von Dissertationen und Diplomarbeiten durchgeführt. Der Fehler wurde nicht entdeckt, weil keine der verwendeten Eingabegitter mehr als doppelt soviele Spalten wie Zeilen hatte. Die Teile der Funktion, die für den Fehler verantwortlich sind hervorgehoben.

Beispiel 6.3.1 Template C++ Funktion:

```
(1) template <class T>
(2) void lll(int rows, int columns)
(3) {
(4)   T* v_sp;
(5)
(6)   v_sp = new T[2*rows];
(7)
(8)   for (int i = 0; i < columns; i++)
(9)     v_sp[i] = 0;
(10) }
```

Unser Analysetool ist nicht in der Lage diese Funktion für einen abstrakten Datentyp T zu analysieren, da erst bei der Verwendung eines konkreten Datentyps XRTL Code erzeugt wird. Durch das Hinzufügen der nachfolgenden Anweisung erzeugt der Compiler für dieses Beispiel und den Datentyp double den entsprechenden XRTL Code.

```
template <double> lll(int, int);
```

Die Argumente rows und columns stehen für die Anzahl der Zeilen bzw. Spalten eines Eingabegitters und nehmen typischerweise Werte aus dem Intervall $[1, 500]$ an. Die minimale Anzahl von Zeilen bzw. Spalten ist also 1, die maximale Anzahl 500. Der Aufruf unseres Analysetools xGCC liefert bei der Analyse dieser Funktion die folgende Fehlermeldung.

```
(xmemory :: set(value, offset) :: too many bytes or offset too large!)
```

Die in den Zeilen (8)-(9) angegebene for-Schleife ist hierfür verantwortlich. Der Compiler erzeugt für die Zeile v_sp = new T[**2*rows**] den folgenden XRTL Code. Wir geben der Einfachheit halber nur den auszuführenden XRTL Ausdruck an.

```
(set (reg:SI 59) (mem:SI (reg:SI 53)))
(set (reg:SI 60) (reg:SI 59))
(parallel[
  (set (reg:SI 61) (ashift:SI (reg:SI 60) (const_int 4)))
  (clobber (reg:SI 274))
])
(set (reg:SI 58) (reg:SI 61))
(set (mem:SI (reg:SI 56)) (reg:SI 58))
(set (reg:SI 0) (call (mem:QI (symbol_ref:SI ("new"))) (const_int 4)))
(set (mem:SI (plus:SI (reg:SI 54) (const_int -4))) (reg:SI 0))
```

Ausgehend vom Speicherbedarf eines double, der 8 Byte beträgt, und dem Ausgangintervall für rows, erzeugt die Anweisung new ein Speicherstück mit Bytegröße im Intervall $[16, 8000]$. Das Array v_sp hat bei minimaler Anzahl von Zeilen eine Größe von 16 und bei maximaler Anzahl von Zeilen von 8000 Bytes. Der Speicherbedarf von v_sp ist unabhängig von der Anzahl der Spalten. Die Anzahl der Iterationen der for-Schleife, die dieses Array initialisiert ist allerdings abhängig von columns, der Anzahl der Spalten.

Der folgende XRTL Code wird durch die Zuweisung v_sp[i] = 0 erzeugt:

```
(set (reg:SI 63) (mem:SI (plus:SI (reg:SI 54) (const_int -8))))
(set (reg:SI 64) (reg:SI 63))
(parallel[
  (set (reg:SI 65) (ashift:SI (reg:SI 64) (const_int 3)))
  (clobber (reg:SI 274))
])
(set (reg:SI 66) (mem:SI (plus:SI (reg:SI 54) (const_int -4))))
(set (mem:DF (plus:SI (reg:SI 65) (reg:SI 66))) (const_double:DF 0))
```

In Register (reg:SI 66) ist die Basisadresse des Speicherstücks des Arrays v_sp gespeichert und in Register (reg:SI 65) das Offset abhängig vom Datentyp double und Index i. Wir erhalten durch die Fixpunktberechnung und das Widening mit anschließendem Narrowing als Offset die möglichen Werte $[0, 3992]$. Der Index i ist durch die Anzahl der Spalten beschränkt und das Speicherstück v_sp hängt von der Anzahl der Zeilen ab. Der Zugriff auf das Speicherstück mit einem Offset aus dem Intervall $[0, 3992]$ führt zu einem Fehler, da v_sp für die minimal Anzahl von Zeilen die Speichergröße 16 hat. Die entsprechenden XRTL Teilausdrücke die diese Abhängigkeiten beschreiben können sehr leicht aus dem Parameter für new und der Schleifenbedingung abgeleitet werden.

Das nachfolgende Beispiel 6.3.2 veranschaulicht die so genannten arithmetischen Fehler. Diese treten auf, wenn arithmetische Operationen für undefinierte Werte ausgeführt werden, wie z.B. eine Division durch 0 oder die Wurzelberechnung für eine negative Zahl. Die Folge sind unerwartete Abstürze und Sicherheitsprobleme. Die nachfolgende, fehlerhafte Funktion war Teil einer traditionellen Testumgebung und sollte Zufallszahlen aus dem Intervall $[min, max]$ liefern.

Beispiel 6.3.2 Zufallszahlen aus $[min, max]$:

```
(1) unsigned xrand32m(unsigned min, unsigned max)
(2) {
(3)   if (min < max)
(4)    {
(5)      unsigned diff;
(6)
(7)      diff = max - min + 1;
(8)      return ((xrand32() % diff) + min);
(9)    }
(10)  else
(11)    return xrand32();
(12) }
```

Die hier verwendete Funktion xrand32 liefert einen zufälligen 32-Bit Wert zurück. Die relevanten Codezeilen (5)-(8) werden folgendermaßen in XRTL Code übersetzt.

```
(set (reg:SI 61) (mem:SI (reg:SI 53)))
(set (reg:SI 62) (mem:SI (plus:SI (reg:SI 53) (const_int 4))))
(parallel[
  (set (reg:SI 60) (minus:SI (reg:SI 62) (reg:SI 61)))
  (clobber (reg:SI 274))
])
(parallel[
  (set (reg:SI 63) (plus:SI (reg:SI 60) (const_int 1)))
  (clobber (reg:SI 274))
])
(set (mem:SI (plus:SI (reg:SI 54) (const_int -4))) (reg:SI 63))
(set (reg:SI 0) (call (mem:QI (symbol_ref:SI ("xrand32"))) (const_int 0)))
(set (reg:SI 64) (reg:SI 0))
(parallel[
  (set (reg:SI 67) (udiv:SI (reg:SI 64)
       (mem:SI (plus:SI (reg:SI 54) (const_int -4)))))
  (set (reg:SI 66) (umod:SI (reg:SI 64)
       (mem:SI (plus:SI (reg:SI 54) (const_int -4)))))
  (clobber (reg:SI 274))
])
(parallel[
  (set (reg:SI 64) (plus:SI (reg:SI 66) (mem:SI (reg:SI 53))))
  (clobber (reg:SI 274))
])
(set (reg:SI 58) (reg:SI 64))
```

Wenn max und min Werte aus dem Intervall $\left[0, 2^{32} - 1\right]$ annehmen können, kann es bei der Berechnung von $max - min + 1$ kann es zu einem Überlauf kommen. Das Ergebnisintervall enthält die Null, denn das Ergebnis von $(2^{32} - 1) - 0 + 1$ ist 2^{32} und das ist äquivalent zu $0 \bmod 2^{32}$, dem kleinsten nicht negativen Vertreter der entsprechenden Restklasse. Bei nachfolgenden modulo-Operation kann es somit zu einer Division durch Null kommen.

Eine weitere häufig auftretende Klasse von Fehlern ist die Benutzung von uninitialisierten Variablen oder Speicherstücken. Diese Art von Fehler führt sehr oft zu einem nicht nachvollziehbarem Programmverhalten, da der Inhalt der verwendeten Variable oder Speicherstücks variieren kann. Das nachfolgende Fortran 77 Programm (Beispiel 6.3.3) demonstriert diesen häufig auftretenden Fehler. Im Ingenieurbereich werden aus historischen Gründen noch sehr viele Fortran Programme eingesetzt, weshalb deren Analyse auch heute noch interessant ist.

Beispiel 6.3.3 Quadratsumme:

```
 (1)       program sum_of_squares
 (2)       real sum(100)
 (3)       real r
 (4)       integer i
 (5)
 (6)       do 10 i = 2,100
 (7)         r = i;
 (8)         sum(i) = sum(i-1) + r * r;
 (9)    10   continue
(10)
(11)       print *, "result: ", sum(100)
(12)       end
```

Die relevante Codezeile (8) wird folgendermaßen in XRTL Code übersetzt.

```
(set (reg:SI 59) (mem:SI (plus:SI (reg:SI 54) (const_int -408))))
(set (reg:SI 60) (mem:SI (plus:SI (reg:SI 54) (const_int -408))))
(set (reg:SF 62) (mem:SF (plus:SI (reg:SI 54) (const_int -404))))
(set (reg:SF 61) (mult:SF (reg:SF 62)
                 (mem:SF (plus:SI (reg:SI 54) (const_int -404)))))
(set (reg:SF 63) (mem:SF (plus:SI (plus:SI (mult:SI (reg:SI 60)
                 (const_int 4)) (reg:SI 54)) (const_int -408))))
(set (reg:SF 64) (plus:SF (reg:SF 63) (reg:SF 61)))
(set (mem:SF (plus:SI (plus:SI (mult:SI (reg:SI 59) (const_int 4))
             (reg:SI 54)) (const_int -404))) (reg:SF 64))
```

Unser Analysetool xGCC liefert für dieses Programm die folgende Fehlermeldung:

```
("xmemory :: get(value, offset) :: parts are not initialized!")
```

Es kommt zu diesem Fehler, da für $i = 2$ innerhalb der Schleife versucht wird auf das nicht initialisierte Arrayelement sum(i-1) zuzugreifen.

Ein praktisches Beispiel für einen „buffer overflow“ der von potentiellen Angreifern missbraucht werden kann, ist die Implementierung der realpath Funktion des FreeBSD Betriebsystems. Diese Funktion wird verwendet um absolute Pfadnamen zu ermitteln. Das Sicherheitsloch wird durch eine fehlerhafte Abfrage der Länge der Pfadangabe verursacht. Eine zu lange Eingabe führt zu einem Speicherüberlauf, der den Stack überschreibt und potenziell zum Einschleusen und Ausführen von beliebigem Code genutzt werden kann. Besonders kritisch ist dieser Fehler für Programme (z.B. sftp-server), die diese Funktion verwenden und mit Administratorrechten laufen. Die fehlerhafte Funktion wurde sofort nach Bekanntwerden der Sicherheitslücke korrigiert. Beispiel 6.3.4 zeigt die fehlerhafte und die korrigierte Version der Implementierungen der realpath Funktion. Wir konzentrieren uns hier auf die unterschiedlichen Codefragmente.

Beispiel 6.3.4 FreeBSD Implementierung der Funktion realpath

Fehlerhafte Implementierung:

```
(1) if (symlink[slen - 1] != '/' && p != NULL) {
(2)         if (slen >= PATH_MAX) {
(3)                 errno = ENAMETOOLONG;
(4)                 return NULL;
(5)         }
(6)
(7)         symlink[slen] = '/';
(8)         symlink[slen + 1] = 0;
(9) }
(10) if (p != NULL)
(11)         left_len = strlcat(symlink, left, PATH_MAX);
(12) if (left_len > PATH_MAX) {
(13)         errno = ENAMETOOLONG;
(14)         return NULL;
(15) }
(16) left_len = strlcpy(left, symlink, PATH_MAX);
```

Die korrigierte Fassung:

```
(1) if (p != NULL) {
(2)         if (symlink[slen - 1] != '/') {
(3)                 if (slen + 1 >= PATH_MAX) {
(4)                         errno = ENAMETOOLONG;
(5)                         return NULL;
(6)                 }
(7)                 symlink[slen] = '/';
(8)                 symlink[slen + 1] = 0;
(9)         }
(10)        left_len = strlcat(symlink, left, PATH_MAX);
(11)        if (left_len >= PATH_MAX) {
(12)                errno = ENAMETOOLONG;
(13)                return NULL;
(14)        }
(15) }
(16) left_len = strlcpy(left, symlink, PATH_MAX);
```

Das Array symlink hat die Größe PATH_MAX und ist ein Puffer für Zeichen. Die ganzzahlige Variable slen kann Werte aus dem Intervall $[0, \texttt{PATH\_MAX}]$ annehmen. Bei der Betrachtung der Zeilen (2) und (8) in der fehlerhaften Implementierung fällt auf, dass die Funktion abgebrochen wird, wenn die Variable slen den Wert PATH_MAX hat. Zeile (8) wird ausgeführt für Werte von slen+1 aus dem Intervall $[1, \texttt{PATH\_MAX}]$. Hier tritt möglicherweise ein Speicherüberlauf auf, da das Array symlink die Größe PATH_MAX hat. In der korrigierten Fassung kann slen+1 in Zeile (8) nur Werte aus dem Intervall $[1, \texttt{PATH\_MAX} - 1]$ annehmen, da die Bedingung in Zeile (3) entsprechend modifiziert wurde. Ein Speicherüberlauf tritt hier in korrigierten Version von realpath nicht auf.

Das Analysetool xGCC liefert für das fehlerhafte Programmfragment

```
(xmemory :: set(value, offset) :: too many bytes or offset too large!)
```

und läuft für die korrigierte Fassung ohne Fehlermeldung durch.

Wir gehen nun auf die in Kapitel 2 vorgestellten Programmfragmente ein.

Beispiel 6.3.5 Ein C Programm mit Pointerarithmetik (siehe Seite 13)

```
(1) int main()
(2) {
(3)   int i = (int )&i;
(4)   double d = 0.9187;
(5)
(6)   i = (int )&i;
(7)
(8)   *((((int *)(i-4))+(int )(((int )d/0.25)+2.9))) = 7;
(9)
(10)  return 0;
(11) }
```

Die Analyse der in Kapitel 2 vorgestellten XRTL Übersetzung liefert einen Stackzugriffsfehler, da das berechnete Offset einen positiven Wert hat. Pointerarithmetik und das Casten von Werten und Pointern wird allerdings auch sinnvoll eingesetzt.

Beispiel 6.3.6 Zwei Programmfragmente (siehe Seite 17) mit gleicher RTL-Übersetzung und unterschiedlicher Erweiterung. Für das korrekte Programmfragment

```
(1) int array[2];
(2)
(3) array[0] = 0;
(4) array[1] = 1;
```

mit XRTL Erweiterung

```
(xinsn (stackvar) (line 3) (size:DI 8) (align 64))
(xinsn (stack) (line 9) (size 8))
```

läuft das Analysetool fehlerfrei durch, während für das offensichtlich inkorrekte Programmfragment

```
(1) int array[1];
(2)
(3) array[-1] = 0;
(4) array[0] = 1;
```

mit XRTL Erweiterung

```
(xinsn (stackvar) (line 3) (size:SI 4) (align 32))
(xinsn (stack) (line 9) (size 4))
```

der auftretende Stackzugriffsfehler entdeckt wird.

6.3.2 Benchmarks

Wir verwenden für unsere Experimente Programme der *Numerical Recipes in C* [31]. Der verwendete Testrechner hat einen Pentium IV Prozessor mit 2.6 Ghz Taktfrequenz und verfügt über 2 GB Hauptspeicher. Als Betriebsystem kommt ein *Debian GNU/Linux* zum Einsatz.

Datei	Laufzeit (in Sek)	Zeilen C	Zeilen XRTL	Schleifen Anzahl	Schleifen Tiefe	Verzweigungenn Anzahl	Verzweigungenn Tiefe
speicherschonend							
ran0.c	0.01	24	117	–	–	1	1
ran1.c	0.19	45	283	1	1	6	2
ran2.c	0.19	63	350	1	1	8	2
bessj0.c	0.01	25	176	–	–	1	1
bessj1.c	0.02	26	195	–	–	2	2
bessj.c	0.25	53	340	2	1	6	3
rc.c	0.10	52	271	1	1	1	1
rd.c	0.98	56	312	1	1	2	2
rf.c	0.40	46	248	1	1	2	2
rj.c	1.69	83	632	1	1	9	3
ei.c	0.15	43	272	2	1	7	2
dawson.c	0.20	42	333	2	1	2	1
gcf.c	0.19	33	199	1	1	4	1
gammln.c	0.05	15	94	1	1	–	–
probks.c	0.13	19	134	1	1	1	1
speicherintensiv							
avevar.c	0.26	14	139	2	1	–	–
fleg.c	0.66	17	129	1	1	1	1
gauher.c	12.66	45	401	3	3	6	4
rebin.c	2.49	15	181	3	2	1	1
vander.c	22.16	32	341	5	2	1	1
lop.c	20.75	13	309	3	2	–	–
matadd.c matsub.c	12.98	7	105	2	2	–	–
gaussj.c	158.63	59	988	13	3	7	3

Tabelle 6.1: Numerical Recipes in C

Die in Tabelle 6.1 aufgeführten Programme wurden mit dem von uns modifizierten Compilers in XRTL übersetzt und dann mit dem Analysetools xGCC untersucht. Da unsere Compilererweitungen keinen messbaren Einfluss auf die Compilezeit haben beziehen sich die angegebenen Laufzeiten beziehen sich nur auf die für die Analyse benötigte Laufzeit. Wir verwenden in unseren Tests eine maximale Länge von 10 für die Liste von Intervallen. Diese Zahl kann sich allerdings, aufgrund von Aufteilungsoperationen von Intervallen in gültige Intervalle, bis zu 40 erhöhen. Die verwendeten Beispielprogramme teilen wir in zwei Gruppen auf. Die Programme die Vektoren und/oder Matrizen verwenden bezeichnen wir als speicherintensiv. Alle anderen Programme bezeichnen wir als speicherschonend. In Tabelle 6.1 verwenden wir für alle speicherintensiven Programme eine maximale Vektorgröße von 200 und eine maximal Matrixgröße von 20×20. In weiteren Tests (siehe Tabelle 6.4 und 6.3) geben wir die Laufzeit des Analysetools in Abhängigkeit von Vektordimension und Matrix-

größe an. Tabelle 6.1 gibt auch Auskunft über die Anzahl der Verzweigungen und Schleifen, sowie die Tiefe der Verschachtelung.

Beim Vergleich der Laufzeiten von speicherschonenden und speicherintensiven Programmen fällt auf, dass die Verwendung von Speicherstücken in Form von Vektoren und Matrizen die Laufzeit sehr stark beeinflusst. Am Beispiel von *rj.c* (83 Zeilen C, 632 Zeilen XRTL) und *matadd.c* (7 Zeilen C, 105 Zeilen XRTL) kann man zeigen, dass die Laufzeit nicht nur von der Programmgröße bestimmt wird. Konzentrieren wir uns allerdings auf die speicherschonenden Programme, so fällt auf, das die Programmgröße in Verbindung mit der Anzahl der Schleifen und der Schleifentiefe die Laufzeit größtenteils bestimmt. Die Schleifentiefe ist hierbei meist entscheidender als die Anzahl der Schleifen. Vergleichen wir die Laufzeit der speicherintensiven Programmen, so können wir am Beispiel von *rebin.c* wo Vektoren und *lop.c* wo Matrizen verwendet werden sehen, dass bei gleicher Schleifenanzahl und -tiefe der höhere Speicherbedarf von *lop.c* entscheidend ist.

Datei	Laufzeit (in Sek) vorher	nachher	Parallele Anweisungen	Reduzierung (in %)
speicherschonend				
ran0.c	0.01	0.01	14	0.00
ran1.c	0.19	0.13	28	31.58
ran2.c	0.19	0.13	41	31.58
bessj0.c	0.01	0.01	4	0.00
bessj1.c	0.02	0.01	5	50.00
bessj.c	0.25	0.22	21	12.00
rc.c	0.10	0.08	8	20.00
rd.c	0.98	0.71	16	27.55
rf.c	0.40	0.30	14	25.00
rj.c	1.69	1.28	23	24.26
ei.c	0.15	0.14	6	6.67
dawson.c	0.20	0.15	23	25.00
gcf.c	0.19	0.16	11	15.79
gammln.c	0.05	0.05	2	0.00
probks.c	0.13	0.10	8	23.08
speicherintensiv				
avevar.c	0.26	0.20	8	23.08
fleg.c	0.66	0.46	12	30.30
gauher.c	12.66	10.57	34	16.51
rebin.c	2.49	1.96	13	21.29
vander.c	22.16	17.33	30	21.80
lop.c	20.75	8.67	36	58.22
matadd.c matsub.c	12.98	8.51	8	34.44
gaussj.c	158.63	102.35	88	35.48

Tabelle 6.2: Parallele Instruktionen

Wir benötigen Kopien von Zuständen für die Fixpunktberechnung und das Abarbeiten von parallelen XRTL Anweisungen. Da beim Kopieren von Zuständen von Registern und Speicherstücken Kopien anlegen werden, ist die Laufzeit abhängig vom Speicherbedarf. Diese Operation bei der Analyse von Programmen mit hoher Schleifenanzahl und -tiefe häufig

verwendet. Die Anzahl der Verzweigungen und die Tiefe der Verschachtelung hat offensichtlich keinen großen Einfluss auf die Laufzeit. Um die Laufzeit der Analyse zu verbessern müssen wir das Kopieren von Zuständen, wie es z.B. bei parallelen Anweisungen verwendet wird, einschränken. Der Großteil der in XRTL Programmen auftretenden parallelen Anweisungen bestehen aus einer set und mehreren clobber Anweisungen (siehe Abschnitt 5.2.2). Diese Form der Anweisungen können wir sequentiell ausführen, wenn keiner der $lvalue$-Ausdrücke in einem $rvalue$-Teilausdruck vorkommt. Das Kopieren von Zuständen entfällt dann. In Tabelle 6.2 geben wir die Anzahl der parallelen Anweisungen, die jetzt sequentiell ausgeführt werden können, und die Laufzeiten der Programme vor und nach der beschriebenen Optimierung an. Wenn diese Parallelanweisungen innerhalb einer Schleife auftreten, dann können wir eine massive Verbesserung der Laufzeit erreichen, wie z.B. für *matadd.c*, wo wir eine Reduzierung um 34.44% bei nur 8 Parallelanweisungen haben. Die durchschnittliche Laufzeitverbesserung für speicherintensive Programme liegt bei 30.14% und bei speicherschonenden Programmen beträgt diese immer noch 19.50%.

Eine Verdopplung der maximalen Listenlänge für Intervalle auf 20 (bzw. 80) bewirkt bei unseren Tests keine messbare Veränderung der Laufzeit, da die Implementierung der Liste von Intervallen nur Teilintervalle verwendet, die sich nicht überschneiden.

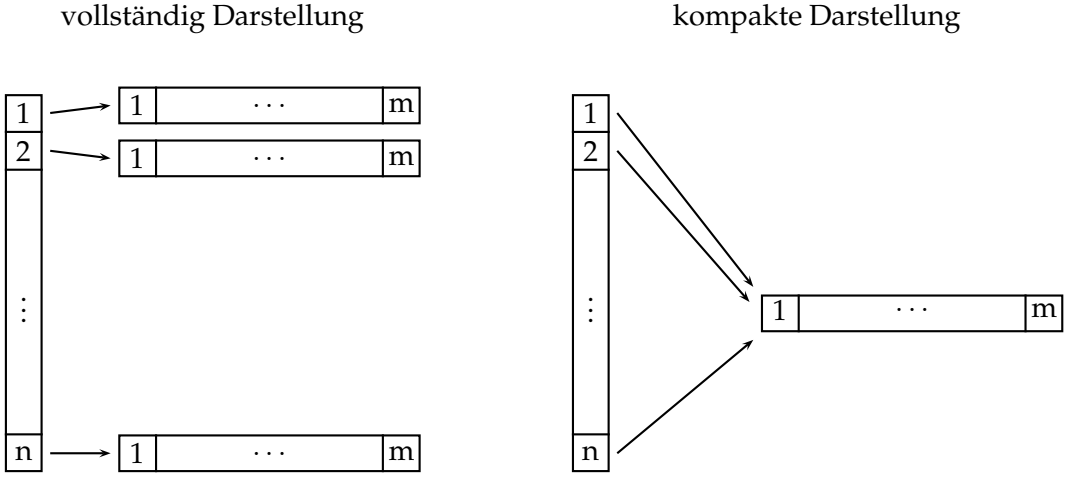


Abbildung 6.1: Unterschiedliche Matrixdarstellungen

Wir versuchen die Laufzeit für speicherintensive Programme weiter zu verbessern, indem wir den Speicherbedarf während der Analyse reduzieren durch eine geeignete Approximation. Für unsere bisherigen Tests haben wir die vollständige Darstellung einer Matrix (siehe Abbildung 6.1) verwendet. Sei n die Anzahl der Zeilen und m die der Spalten einer Matrix. Dann hat die vollständige Darstellung einen Speicherbedarf von $O(nm)$. Verwenden wir jedoch eine kompaktere, aber ungenauere Darstellung (siehe Abbildung 6.1), wo unabhängig von der Anzahl der Zeilen nur ein Vektor existiert, so reduziert sich der Speicherbedarf auf $O(n + m)$.

Tabelle 6.3 zeigt die Laufzeit der Matrixaddition *matadd.c* in Abhängigkeit von n. Wir haben dabei die standardmäßige mit einer alternativen Implementierung, die sich noch in der

matadd.c	Laufzeit (in Sek)			
	Standard		Alternativ	
n	vollst.	komp.	vollst.	komp.
20	8.51	0.75	1.06	0.37
50	48.37	1.54	3.59	0.57
100	253.43	2.82	11.04	0.89
200	–	5.32	38.57	1.54
500	–	13.34	220.43	3.65
1000	–	35.25	–	8.15
2000	–	114.49	–	29.25
5000	–	542.51	–	183.44

Tabelle 6.3: Laufzeit von matadd.c abhängig von n

Entwicklungsphase befindet, verglichen. Für beide Implementierungen haben wir die vollständige und die kompakte Darstellung verwendet. Die konventionelle Speicherimplementierung benötigt Vielfaches der eigentlichen Speichergröße an Platz, weil jedes Byte eines Speicherstücks nachgebildet wird. Die alternative Implementierung verwendet einen balancierten Intervallbaum, der die Offsets der Speicherzugriffe und den dazugehörigen Wert speichert. Die Menge des benötigten Speichers kann, unabhängig von der Größe des zu approximierden Speicherstücks, beschränkt werden. In Tabelle 6.4 finden wir den Vergleich der beiden Implementierungen für das Programm *avevar.c*, abhängig von der maximalen Vektordimension n.

avevar.c	Laufzeit (in Sek)	
n	Standard	Alternativ
1000	0.72	0.17
2000	1.36	0.26
5000	3.26	0.55
10000	6.42	1.02
20000	12.77	1.96
50000	31.87	4.79
100000	63.35	9.49
200000	–	18.92
500000	–	47.29
1000000	–	94.30

Tabelle 6.4: Laufzeit von avevar.c abhängig von n

Die Tabellen 6.3 und 6.4 verdeutlichen das Potenzial der alternativen Implementierung. Für Matrizen lohnt sich der Einsatz der kompakten Darstellung, wenn wir auf die Präzision der vollständigen Darstellung verzichten können.

In diesem Kapitel haben wir unser Analysetool xGCC und andere statische und nichtstatische Analysetools vorgestellt. Wir haben unsere Compilermodifikationen und die zentralen Datenstrukturen für das Analysetool xGCC vorgestellt. Mit Hilfe ausgewählter Beispiele haben wir gezeigt welche Arten von Fehlern unser Tool entdeckt und in Experimenten mit Programmen aus den *Numerical Recipes in C* die Laufzeit der Analyse untersucht. Wir haben gezeigt, dass durch geeignete Approximationen und Optimierungen die Laufzeit der Analyse deutlich reduziert werden kann.

Kapitel 7

Zusammenfassung

In dieser Arbeit haben wir das Analysetool xGCC vorgestellt, das den XRTL Code untersucht, den wir mit einem modifizierten Compilers erzeugen. Wir haben die Register Transfer Language (RTL) vorgestellt und motiviert, warum diese Erweiterung zu XRTL notwendig war. Anschließend wurde der konkrete und abstrakte Domain für ganze und reelle Zahlen, sowie Speicherstücke und Zeiger vorgestellt. Wir haben das Konzept der Liste von gültigen Intervallen erklärt, welches ein Anpassen der Genauigkeit der Approximation ermöglicht. Entsprechende Abstraktions- und Konkretisierungsfunktionen, sowie Operationen auf dem abstrakten Domain wurden definiert. Anschließend haben wir implementierungsspezifische Details genauer beschrieben und sind auf die Darstellung der in der Praxis verwendeten Datentypen genauer eingegangen. Wir haben die auf den Grunddatentypen definierten Zugriffsfunktionen verwendet um Speicherstücke zu implementieren. Wir haben die abstrakte Interpretation der XRTL Sprache beschrieben wie z.B. einfache und parallele Instruktionen ausgeführt werden. Wir sind genauer auf Verzweigungen eingegangen und haben gezeigt wie wir aus der Bedingung Constraints ableiten können, um die Menge von Zuständen für die einzelnen Zweige einzuschränken. Anschließend stellten wir die drei verschiedenen Schleifentypen vor und zeigten, wie wir mit Hilfe des Widening/Narrowing einen Fixpunkt für Schleifen berechnen. Wir erklärten, wie wir Funktionsaufrufe abarbeiten. Abschließend haben wir unseren Ansatz mit anderen statischen und nicht-statischen Analysetools verglichen und demonstrierten anhand ausgewählter Beispiele die Stärken und Schwächen unseres Tools. Wir verwendeten Beispielprogramme der *Numerical Recipes in C* um das Laufzeitverhalten des Analysetools xGCC zu untersuchen. Wir haben einige Verbesserungen und eine alternative, noch im Experimentierstadium befindliche, Implementierung von Speicherstücken vorgestellt.

Für die zukünftige Weiterentwicklung des Analysetools xGCC sind zwei Bereiche interessant, die Verfeinerung der Analyse und die Reduzierung der Laufzeit. Um die Analyse zu verfeinern ist es sinnvoll Ideen anderer Analysetechniken zu integrieren, z.B. eine Aliasanalyse, oder Prädikate zu integrieren, wie sie in der Predicate Abstraction verwendet werden. Weitere Erweiterungen von XRTL, durch Extraktion von zusätzlichen Compilerinformationen, könnten die Analyse weiter vereinfachen oder auch verfeinern. Die in neueren Compilerversionen hinzugekommenen Optimierungs- und Analysetechniken könnten einige diese Daten liefern. Um das Abarbeiten von Verzweigungen zu verbessern wäre die

Generierung weiterer Constraints sinnvoll. Dies würde es uns erlauben die Auswirkungen, die sich aus der Bedingung einer Verzweigung ergeben, besser auszunutzen. Um das Laufzeitverhalten von xGCC weiter zu verbessern ist eine Optimierung der Fixpunktberechnung notwendig. Das Kopieren von Zuständen muss reduziert werden, oder es dürfen nur noch die Teile kopiert werden, die sich verändern. Für verschachtelte Schleifen müssen wir überprüfen, in wieweit vorher berechnete Fixpunkte wiederverwendet werden können. Um die Praxistauglichkeit weiter zu erhöhen können die verwendeten Datenstrukturen (Klassen) durch geeignete Heuristiken weiter verbessert werden. Wir können zusätzliche Operationen auf Zeigern erlauben, durch deren Erweiterung um virtuelle Startadressen. An einer weiteren Verbesserung der optimierten Speicherimplementierung wird gearbeitet. Eine Beschränkung der Knoten des Intervallbaumes, in Kombination mit der Beschränkung des Platzbedarfes, erlauben es jeden Zugriff auf ein Speicherstück, unabhängig von dessen Größe, in konstanter Zeit durchzuführen. Die Arbeiten an einer graphischen Benutzeroberfläche haben begonnen. Ein Langzeitziel ist die Integration von Teilen der Implementierung in den offiziellen Compiler.

Literaturverzeichnis

[1] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In Alan Mycroft, editor, *SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50. Springer, September 1995.

[2] Thomas Ball, Andreas Podelski, and Sriram K Rajamani. Boolean and cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer (STTT), Special Section on TACAS'01*, 2001.

[3] Thomas Ball, Andreas Podelski, and Sriram K Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Kaoen and Perdita Stevens, editors, *Proceedings of TACAS02: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172. Springer-Verlag, 2002.

[4] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Program Analysis*, pages 1–3. ACM Press, 2002.

[5] Bruno Blanchet. Introduction to abstract interpretation. *Vorlesungskript*, 2003.

[6] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 1993.

[7] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*. Springer-Verlag, Berlin, Germany, 1993.

[8] François Bourdoncle. Abstract debugging of higher-order imperative languages. In Robert Cartwright, editor, *Proceedings of the Conference on Programming Language Design and Implementation*, pages 46–55. ACM Press, 1993.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *acm Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.

[10] Edmund Clarke and Daniel Kroening. ANSI-C bounded model checker - user manual. Technical report, Carnegie Mellon University, 2003.

[11] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[13] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[15] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[16] P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*. Scuola Superiore G. Reiss Romoli, Compact disk paper 224 and electronic proceedings, L'Aquila, Italy, July 31 – August 6 2000.

[17] Chris Drake and Kimberley A.D. Brown. *Panic! : UNIX Crash Dumps analysieren*. Prentice Hall, 1996.

[18] Dawson R. Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI-2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 191–210. Springer, 2004.

[19] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Program Analysis*, pages 58–70, 2002.

[20] IEEE. *IEEE standard for the Binary Floating-Point Arithmetics*, volume Std. 754-1985 of *IEEE standard*. IEEE, 1985.

[21] IEEE. *IEEE standard for the Radix-Independent Floating-Point Arithmetic*, volume Std. 854-1987 of *IEEE standard*. IEEE, 1987.

[22] Neil D. Jones and Flemming Nielson. Abstract interpretation : a semantics-based tool for program analysis. Kobenhavns universitet, Kobenhavns Universitet/Datalogisk Institut, 1994.

[23] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language : ANSI C*. Prentice Hall, 2nd ed. edition, 1988.

[24] Florian Martin. *Generation of Program Analyzers*. PhD thesis, Universität des Saarlandes, 1999.

[25] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, pages 75–88. ACM Press, 2002.

[26] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Program Analysis*, pages 128–139, 2002.

[27] George Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *In Proceedings of Conference on Compiler Construction*, 2002.

[28] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In Jr. James B. Fenwick and Cindy Norris, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, volume 38, 5 of *ACM SIGPLAN Notices*, pages 232–244. ACM Press, 2003.

[29] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[30] Pure Software. *Purify : user's guide*. Pure Atria Corporation, version 4.0 edition, 1996.

[31] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C : the art of scientific computing*. Cambridge Univ. Press, 2nd ed. edition, 1994.

[32] Richard M. Stallman. *GNU Compiler Collection*. Free Software Foundation, Inc., 2002.

[33] Richard M. Stallman. *GNU Compiler Collection Internals*. Free Software Foundation, Inc., 2002.

[34] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 4. special edition edition, 2000.

[35] Sun Microsystems. *Numerical Computation Guide*, 2000.

[36] Alfred Tarski. A lattice theoretical fixpoint theorem and it's applications. *Pacific Journal of Math. 5*, 1955.

[37] Stephan Thesing, Florian Martin, Martin Alt, and Oliver Lauer. *PAG User's Manual*, 1998. Version 1.0.

[38] R. Vallée-Rai. Soot: A java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montreal, July 2000.

[39] André Willms. *C++ Standard Template Library*. Galileo-Press, 2000.

[40] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau : Theorie, Konstruktion, Generierung*. Springer-Lehrbuch. Springer, 2., überarb. und erweiterte auflage edition, 1996.

[41] Bandera - Homepage, August 2004. `http://bandera.projects.cis.ksu.edu/`.

[42] François Bourdoncle, August 2004. `http://www.exalead.com/Francois.Bourdoncle/`.

[43] BLAST - Homepage, August 2004. `http://www-cad.eecs.berkeley.edu/~rupak/blast/`.

[44] CBMC - Homepage, August 2004. `http://www-2.cs.cmu.edu/~modelcheck/cbmc/`.

[45] Dawson Engler. MC - Homepage, August 2004. `http://www.stanford.edu/~engler/`.

[46] LiDIA - Homepage, August 2004. `http://www.informatik.tu-darmstadt.de/TI/LiDIA`.

[47] George Necula. CIL - Homepage, August 2004. `http://www.cs.berkeley.edu/~necula/`.

[48] PAG - Homepage, August 2004. `http://pag.cs.uni-sb.de/`.

[49] Sable/Soot - Homepage, August 2004. `http://www.sable.mcgill.ca/`.

[50] SLAM Project - Homepage, August 2004. `http://research.microsort.com/slam/`.

[51] GCC extension for protecting applications from stack-smashing attacks, August 2004. `http://www.research.ibm.com/trl/projects/security/ssp/`.

[52] Valgrind - Homepage, August 2004. `http://valgrind.kde.org`.