KARLSRUHER INSTITUT FÜR TECHNOLOGIE
FAKULTÄT FÜR INFORMATIK

Kevin-Simon Kohlmeyer

Bachelorarbeit

Funktionale Konstruktion und Verifikation von Kontrollflussgraphen

Functional Construction and Verification of Control Flow Graphs

Informatik

Betreuer:
Prof. Dr.-Ing. Gregor Snelting
Andreas Lochbihler

7. August 2012

Abstract

DIESE Arbeit stellt einen Algorithmus vor, der im Quelltext vorliegende While-Programme in Kontrollflussgraphen überführt. Diese Kontrollflussgraphen sind so gestaltet, dass sie im Rahmen des von Daniel Wasserrab in seiner Dissertation vorgestellten Slicing-Frameworks [6] weiter genutzt werden können. Es wurde ein Parser entwickelt, der im Quelltext vorliegende Programme in Syntaxbäume überführt, die dann von einem verifizierten Algorithmus in einen Kontrollflussgraphen umgewandelt werden. Dabei wird eine für diesen Zweck entwickelte, abstrakte Graphenschnittstelle verwendet, die zusammen mit einer dazugehörigen, verifizierten Implentierung, die auch Teil dieser Arbeit ist, eine Ausführung des Algorithmus erlaubt. Die Laufzeiten der Graphenimplementierung und der Konstruktion der Kontrollflussgraphen werden anhand des Quelltextes hergeleitet und anschließend experimentell bestätigt.

Inhaltsverzeichnis

KAPITEL 1

Einführung

DIE moderne Gesellschaft ist in einem hohen Maß abhängig von Computern. Dies betrifft selbst kritische Bereiche wie Kraftwerkssteuerungen oder elektronische Stabilitäts- und Notfallsysteme von Flugzeugen, in denen das Versagen eines Computers fatale Folgen haben kann. Daraus geht ein Bedarf nach gesteigerter Verlässlichkeit dieser Maschinen hervor, was insbesondere die immer komplexer werdende Programmierung betrifft. Da es sich bei Software um formale Systeme handelt, lässt sich diese Verlässlichkeit am besten durch einen formalen Beweis erreichen. Diese Systeme sind jedoch oft sehr komplex und sehen sich laufend ändernden Anforderungen und Rahmenbedingungen gegenübergestellt, was den Aufwand solcher Beweise in die Höhe treibt. Die Möglichkeit, Aussagen über Programme voll- oder zumindest teilautomatisch zu treffen, soll dem entgegenwirken.

Eine Technik, um solche Aussagen herzuleiten, ist Slicing, dessen Hauptaugenmerk darauf liegt, für einzelne Stellen einer Anwendung all jene Anweisungen zu bestimmen, die für sie relevant sind. Einen wichtigen Beitrag zu dieser Technologie stellt das von Daniel Wasserrab in seiner Dissertation „From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security“ [6] vorgestellte verifizierte Slicing-Framework dar. Dieses Framework entstand am Karslruher Institut für Technologie im Rahmen des Projektes „Quis custodiet“[1], dessen Ziel es ist, Werkzeuge zur Programmverifikation bereitzustellen, die wiederum selbst maschinell verifiziert wurden.

[1] „Quis custodied“ ist abgeleitet von dem lateinischen Sprichwort „Quis custodiet ipsos custodes?“, zu deutsch „Wer überwacht die Wächter?“

Das von Wasserrab entwickelte Framwork ist jedoch in seiner jetzigen Form nicht ausführbar, da es auf induktiven Definitionen der zu Grunde liegenden Graphen basiert. Diese Arbeit stellt einen Schritt in Richtung eines ausführbaren Systems dar, indem sie einen ausführbaren und verifizierten Algorithmus zur Konstruktion von Kontrollflussgraphen liefert.

1.1 Slicing

Mark Weiser veröffentlichte 1981 das Paper „Program Slicing“ [7]. Die darin vorgestellten Slices bezeichnen alle Anweisungen eines Programms, die eine bestimmte Stelle beeinflussen, wobei sowohl Daten- als auch Kontrollabhängigkeiten berücksichtigt werden. Eine Datenabhängigkeit zwischen zwei Anweisungen liegt genau dann vor, wenn beide die gleiche Variable benutzen oder verändern. Bestimmt der Ausgang einer Anweisung, ob der Kontrollfluss eine andere Anweisung erreicht, so liegt eine Kontrollabhängigkeit vor. Der Slice bezüglich einer Anweisung ist die Menge der Anweisungen, zu denen direkte oder indirekte Daten- oder Kontrollabhängigkeiten vorliegen.

Slicing-Verfahren lassen sich in statisches Slicing und dynamisches Slicing unterteilen. Während bei der statischen Variante alle möglichen Programmpfade berücksichtigt werden, bezieht dynamisches Slicing einen Programmzustand mit ein, wodurch die möglichen Programmpfade eingeschränkt werden können. Dadurch können kleinere, aussagekräftigere Slices gefunden werden.

Wasserrabs Framework beinhaltet Algorithmen zur Generierung von dynamischen und statischen Slices.

1.2 Zielsetzung dieser Arbeit

Die grundlegende Motivation hinter dieser Arbeit ist eine Erweiterung des von Wasserrab präsentierten Frameworks, um durch einen Export in eine Sprache wie Standard ML eine Sammlung von ausführbaren, verifizierten Algorithmen zu schaffen, die verifiziertes Slicing erlaubt.

Diese Arbeit selbst hat das Ziel, eine performante Implementierung für die Konstruktion von Kontrollflussgraphen zu liefern und eine Struktur zu entwickeln, auf der zukünftige Arbeiten aufbauen können. Es wird nur die Pro-

grammiersprache While betrachtet, um die Komplexität der sprachspezifischen Teile möglichst gering zu halten. Um eine Verwechselung der Sprache While mit einer *while*-Schleife zu vermeiden, wird in dieser Arbeit das Wort *while* typographisch hervorgehoben, wenn die Anweisung gemeint ist.

Weiterhin wird eine Schnittstelle für allgemeine Graphen definiert. Um den Konstruktionsalgorithmus tatsächlich ausführen zu können, wurde auch eine zur Schnittstelle gehörige Implementierung entwickelt. Sowohl die Graphenimplementierung als auch der eigentlich Algorithmus wurden mithilfe des Isabelle-Beweisassistenten [4] verifiziert.

Die Transformation von im Quelltext vorliegenden Whileprogrammen in abstrakte Syntaxbäume wurde nicht verifiziert, da dort zum größten Teil die Syntax von While spezifiziert wurde. Eine Formulierung von Korrektheitsaussagen würde den Code wiederholen. Darüber hinaus kann durch den Verzicht auf eine Verifikation die Entwicklung auf einer niedrigeren Ebene Isabelles stattfinden, wodurch vorhandene Parsing-Bibliotheken verwendet werden können.

Durch die Beschränkung auf While ist die Anwendbarkeit in der Praxis begrenzt. Da jedoch mit dieser Arbeit nur ein Teil eines komplett ausführbaren, verifizierten Frameworks behandelt wird, der als Grundlage für weitere Entwicklungen dienen soll, ist dieser Nachteil hinnehmbar. Die sprachspezifischen Teile können dann in Zukunft für andere Sprachen implementiert werden, ohne eine neue Programmstruktur entwickeln zu müssen.

1.3 Technologischer Rahmen

In diesem Abschnitt werden die technischen Rahmenbedingungen dieser Arbeit vorgestellt. Da das Slicing-Framework in *Isabelle/HOL* [4] implementiert ist, wird es auch hier benutzt. *Isabelle* ist ein generischer Beweisassistent, der mit verschiedenen Objektlogiken instanziierbar ist. Die bekannteste ist *Higher Order Logic (HOL)*, die auch hier zum Einsatz kommt. Mit HOL ist eine Programmierung möglich, die an Standard ML erinnert, auf dem auch Isabelle selbst aufbaut.

Da Aussagen von den bei Programmverifikation auftretenden Komplexitätsgraden selten automatisch bewiesen werden können, gibt bei Isabelle in der Regel der Benutzer die Beweisstruktur vor. Ihm stehen dazu eine Reihe von

Werkzeugen zur Verfügung, mit deren Hilfe er einzelne Beweisabschnitte so weit wie möglich automatisiert ablaufen lassen kann.

Isar erweitert Isabelle/HOL um strukturierte Beweisführung sowie um die Möglichkeit, LaTeX- und Isabelle/HOL-Code zu vermischen und so die tatsächlichen Definitionen in das Dokument einzubinden, anstatt sie in LaTeX nachzubilden. Sowohl für die im Rahmen dieser Arbeit entwickelten Beweise als auch für den Satz aller abgedruckten Aussagen und Codefragmente wurde Isar verwendet.

Detaillierte Informationen zu Isabelle und Isar sind in [4] und [8] zu finden.

KAPITEL 2

Vorüberlegungen

In diesem Kapitel werden einige Überlegungen vorgestellt, die in die Ausgestaltung dieser Arbeit eingeflossen sind.

Eine allgemeine Vorüberlegung betrifft den Rahmen der Arbeit und die Wahl der betrachteten Programmiersprache. Dabei fiel die Entscheidung auf While. Dies ist motiviert durch den exemplarischen Charakter dieser Arbeit; sie soll in erster Linie die Machbarkeit der effizienten Konstruktion von Kontrollflussgraphen in Isabelle/HOL aufzeigen. Darüber hinaus sollen möglichst flexible Schnittstellen definiert werden, um eine gute Wiederverwendbarkeit zu erreichen.

2.1 Die Sprache While

Die Sprache While wird durch die Grammatik in Abbildung 2.1 beschrieben. Sie ist sehr simpel und besteht nur aus wenigen Anweisungen, jedoch ist ihre Struktur der moderner Sprachen ähnlich. Durch ihre Einfachheit kann der bei üblichen Hochsprachen sehr komplexe Parsing-Teil hier vergleichsweise simpel ausfallen. Da sich das auch im abstrakten Syntaxbaum niederschlägt, vereinfacht es auch andere sprachspezifische Teile des Algorithmus.

Das Slicing-Framework enthält bereits Formalisierungen von Kontrollflussgraphen und abstrakten Syntaxbäumen für While. Die Spezifikation des abstrakten Syntaxbaumes ist Abbildung 2.2 zu entnehmen.

Variable ::=	[a-zA-Z]+
Value ::=	″true″ | ″false″ | [+-]?[0-9]+
Atom ::=	Variable | Value
BinOperator ::=	″=″ | ″&″ | ″<″ | ″+″ | ″-″
BinOpPart ::=	″(″ Expression ″)″ | Atom
BinOp ::=	″(″ BinOp ″)″ | BinOpPart BinOperator BinOpPart
Expression ::=	BinOp | Atom
Block ::=	Statement ″;″ Block | ϵ
Statement ::=	Assignment | If | While
Assignment ::=	Variable ″:=″ Expression
If ::=	IfThen | IfThen IfElse
IfThen ::=	″if(″ Expression ″){″ Block ″}″
IfElse ::=	″else{″ Block ″}″
While ::=	″while(″ Expression ″){″ Block ″}″
WhileProgramm ::=	Block

Abbildung 2.1: Struktur von While

type-synonym *vname* = *string* — Name für Variablen

datatype *val* =
  Bool bool — Boolscher Wert
| *Intg int* — Ganzzahliger Wert

datatype *bop* = *Eq* | *And* | *Less* | *Add* | *Sub* — Namen der binären Operationen

datatype *expr* =
  Val val — Wert
| *Var vname* — Lokale Variable
| *BinOp expr bop expr* (- «-» -) — Binäre Operation

datatype *cmd* =
  Skip
| *LAss vname expr* (- := -) — Lokale Zuweisung
| *Seq cmd cmd* (- ;;/ -)
| *Cond expr cmd cmd* (*if* ′(- ′) -/ *else* -)
| *While expr cmd* (*while* ′(- ′) -)

Abbildung 2.2: Struktur des abstraken Syntaxbaumes von While

Mehrere Anweisungen werden dabei nicht in einer Liste verpackt, sondern mittels des *Seq*-Konstruktors verbunden, was genau wie die üblichen Listen zu einer einfach verketteten Struktur führt.

Die Klammern nach den einzelnen Konstruktoren beinhalten die Definition alternativer Syntax. Dabei haben runde Klammern und der Bindestrich eine besondere Bedeutung, weshalb ihnen ein Apostroph vorangestellt werden muss, wenn die spezielle Bedeutung nicht erwünscht ist. Ein Bindestrich ist ein Platzhalter für die Parameter. Beispielsweise beschreibt „$'- - '-$“ Syntax wie „$- x -$“. Ein Schrägstrich gefolgt von einem Leerzeichen erlaubt es Isabelle, das Leerzeichen bei Bedarf durch einen Zeilenumbruch zu ersetzten.

Mit diesen Definitionen können Ausdrücke wie

$$\mathit{if}\ (\ \mathit{Val}\ (\mathit{Intg}\ 1)\ \text{«}\mathit{Less}\text{»}\ \mathit{Var}\ ''x''\ )\ \mathit{Skip}\ \mathit{else}\ (''x'' := \mathit{Val}\ (\mathit{Intg}\ 1))$$

im Code verwendet werden.

Der Parsing-Teil wurde in dem Isabelle zugrunde liegenden Standard ML implementiert. Dadurch wird eine wesentlich aufwändigere Implementierung in HOL vermieden. Dass deshalb keine verifizierbaren Aussagen über das Parsing getroffen werden können, wird in Kauf genommen, da es sich hierbei nur um ein Randthema dieser Arbeit handelt. Die Implementierung orientiert sich strukturell an der in Abbildung 2.1 vorgestellten Grammatik. Zusätzlich werden zu Beginn alle nicht druckbaren Zeichen wie Tabulatoren, Leerzeichen und Zeilenumbrüche entfernt. Dies erlaubt eine beliebige Quelltextformatierung, wie dies z. B. von Programmiersprachen wie C bekannt ist.

Der Parser konstruiert ohne zusätzliche Zwischenstufen einen abstrakten Syntaxbaum vom Typ *cmd*, der in Abbildung 2.2 zu sehen ist.

2.2 Kontrollflussgraphen

Bei einem Kontrollflussgraphen handelt es sich um einen gerichteten Graphen im informationstechnischen Sinne, also um eine Menge von Knoten und eine Relation auf dieser Menge, welche die Kanten darstellt. Ein Kontrollflussgraph repräsentiert die Struktur eines Programms. Dabei werden

datatype *w-node* = *Node nat* (*'(' - '-'*)
| *Entry* (*'('-Entry'-'*)
| *Exit* (*'('-Exit'-'*)

type-synonym *state* = *vname* ⇀ *val*

datatype *'state edge-kind* =
Update 'state ⇒ *'state* (⇑-)
| *Predicate 'state* ⇒ *bool* (*'(-')*$_{\surd}$)

type-synonym *w-edge* = (*w-node* × *state edge-kind* × *w-node*)

Abbildung 2.3: Definition der Bestandteile des Kontrollflussgraphen

die Effekte einer Anweisung und die Bedingungen einer Verzweigung als Kantenannotationen mit dem Graphen assoziiert.

Wasserrab liefert in seinem Framework eine induktive Definition von Kontrollflussgraphen, dessen Knoten- und Kantentypen in Abbildung 2.3 abgebildet sind. Knoten werden durch natürliche Zahlen identifiziert, wobei zusätzlich noch (*-Entry-*) und (*-Exit-*) vorkommen. Bei einer Kante kann es sich entweder um eine *Update*-Kante wie zum Beispiel $\Uparrow\lambda s.\ s(''x'' \mapsto 3)$ handeln, die der Variablen x den Wert *3* zuweist, oder um eine *Predicate*-Kante wie $(\lambda s.\ interpret\ (Var\ ''x'')\ s = Some\ true)_{\surd}$, der genau dann gefolgt werden soll, wenn die Variable x den Wert *true* hat.

Die Kontrollflussgraphen weisen aus beweistechnischen Gründen in der Regel mehr als einen Knoten pro Anweisung auf[1]. Bei einer Zuweisung werden beispielsweise zwei neue Knoten hinzugefügt, zwischen denen dann die Zustandsänderung stattfindet.

Um die Gestalt der Definition zu illustrieren, werden nun zwei Teile daraus vorgestellt. Der einfachste Teil ist die Aussage

$$prog \vdash (\text{-}Entry\text{-}) -(\lambda s.\ True)_{\surd}\rightarrow (\text{-}\ 0\ \text{-})$$

Sie besagt, dass der Kontrollflussgraph jedes Programms *prog* die Kante

$$((\text{-}Entry\text{-}), (\lambda s.\ True)_{\surd}, (\text{-}\ 0\ \text{-}))$$

beinhaltet.

[1]Eine detaillierte Erklärung der Konstruktion ist in [6] zu finden

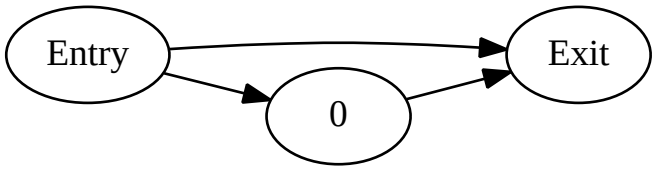

Abbildung 2.4: Kontrollflussgraph von *Skip*

Eine etwas komplexere Aussage ist

$$[\![c_1 \vdash n -et\rightarrow n'; n \neq (\text{-}Entry\text{-})]\!] \Longrightarrow if\ (b)\ c_1\ else\ c_2 \vdash n \oplus 1 -et\rightarrow n' \oplus 1$$

Aus ihr ist zu entnehmen, dass für den *then*-Teil einer *if*-Anweisung alle Kanten, die nicht von (*-Entry-*) ausgehen, übernommen werden. Dabei werden die Knotenindizes inkrementiert, was durch $n \oplus 1$ und $n' \oplus 1$ ausgedrückt wird. Bei Kanten, die von (*-Entry-*) ausgehen, kann es sich stets nur um die bereits erwähnte Kante nach (*- 0 -*) handeln, welche nie übernommen wird, da sie nach der ersten vorgestellten Regel Teil jedes Graphen ist. Kanten zu (*-Exit-*) werden bei einer *if*-Anweisung einfach übernommen (Es gilt (*-Exit-*) $\oplus\ i =$ (*-Exit-*)).

Ein trivialer Kontrollflussgraph ist in Abbildung 2.4 zu sehen. Der dazugehörige Syntaxbaum ist *Skip*. Die beiden Kanten haben den Wert $\Uparrow id$, es handelt sich also um *Update*-Kanten mit der Identitätsfunktion.

2.3 Ein abstraktes Grapheninterface

Performanz und Wiederverwendkarkeit sind Kernanforderungen dieser Arbeit. Um sie im Angesicht unbekannter Verwendung der Kontrollflussgraphen zu ermöglichen, wurde eine abstrakte Graphenschnittstelle definiert, auf der der Konstruktionsalgorithmus arbeitet. Vorbild dafür ist das Isabelle Collections Framework [3], von dem die Struktur der Schnittstellendefinition übernommen wurde.

Die Implementierung basiert auf einem Rot-Schwarz-Baum, der den Knoten jeweils eine Liste ihrer aus- und eingehenden Kanten zuordnet. Dadurch lassen sich die meisten Operationen in logarithmischer Zeit abhängig von der Knotenanzahl verwirklichen. Es wird davon ausgegangen, dass Knotengrade konstant und sehr klein sind, wie es bei Kontrollflussgraphen üblicherweise der Fall ist.

Um eine möglichst vielfältige Nutzung zu ermöglichen, ist der Graph mehrfach parametrisiert. Er unterstützt sowohl beliebige Knotentypen – wobei die Implementierung eine totale Ordnung fordert – als auch einen beliebigen Typ für Kantenannotationen. In den generierten Kontrollflussgraphen werden sie beispielsweise genutzt, um Bedingungen und Zustandsübergänge abzubilden.

KAPITEL 3

Der Graph

DIESES Kapitel dient der Vorstellung der für diese Arbeit entwickelte Graphenschnittstelle und ihrer Implementierung, die auch auf ihre asymptotische Laufzeit hin untersucht wird. Im Anschluss wird noch die Verifikation der implementierten Funktionen erläutert.

Die Schnittstelle orientiert sich strukturell an den im Isabelle Collections Framework [3] vorgestellten abstrakten Datentypen. Dort werden die Datentypen in sogenannten Locales organisiert, einer Technik zur Strukturierung von Isabelle/HOL-Code. Ein Beispiel, in dem die meisten für diese Arbeit relevanten Möglichkeiten von Locales zum Einsatz kommen, ist in Abbildung 3.2a dargestellt. In einer Locale werden mit *fixes* Konstanten oder Funktionen fixiert und mit *assumes* Annahmen über diese getroffen. Durch eine Interpretation der Locale können, nach einem Beweis der Annahmen, die im Rumpf der Locale entwickelten Aussagen auf konkrete Werte übertragen werden. Locales können auch kombiniert und erweitert werden, so ist zum Beispiel die Locale *graph-empty* aus Abbildung 3.2a eine Erweiterung der in Abbildung 3.1 abgebildeten Locale *graph*. Mehr Informationen zu Locales sind in [1] zu finden.

Die Basis-Schnittstelle besteht aus zwei Funktionen. Die Erste ist eine Invariante, die von konkreten Implementierungen definiert und genutzt werden kann. Die zweite Funktion ist eine Abbildung des abstrakten Graphentypes auf einen simplen Isabelle/HOL-Typ, auf dessen Bildmenge dann Aussagen über Operationen getroffen werden können. Dadurch können die Auswirkungen einzelner Operationen unabhängig voneinander spezifiziert werden.

locale *graph* =
fixes *αe* :: *'g* ⇒ (*'node* × *'edgeD* × *'node*) *set*
and *invar* :: *'g* ⇒ *bool*

Abbildung 3.1: Locale mit Definition der grundlegenden Graphenstruktur

Es ist anzumerken, dass keine dieser beiden Funktionen zur Ausführung bestimmt ist. Sie dienen ausschließlich dazu, Aussagen über Operationen zu treffen. Aus diesem Grund sind performante Formulierungen weniger angebracht als klare Definitionen, die spätere Beweise erleichtern.

3.1 Grundlegende Graphenstruktur

In Abbildung 3.1 wird die Grundstruktur mit diesen beiden Funktionen definiert. Dabei ist *αe* die Projektionsfunktion, die einen konkreten Graphen auf ein Menge von Kanten, dargestellt durch den Typ (*'node* × *'edgeD* × *'node*) *set*, abbildet. Das Prädikat *invar* legt die Invariante des Graphen fest.

Für eine grundlegende Graphenstruktur fehlt noch der Basisfall des leeren Graphen *empty*, der in Abbildung 3.2a definiert wird. Die Eigenschaften von *empty* werden in den Annahmen *empty-invar* und *empty-correct* festgelegt, die die Erfüllung der Invariante sowie die Tatsache, dass *empty* keine Knoten oder Kanten enthält, beinhalten.

3.2 Weitere Graphenoperationen

In diesem Abschnitt werden die Operationen erläutert, die zur Kontrollflussgraphenkonstruktion und Darstellung nötig sind.

Die Definition der *addEdge*-Funktion, die dem Hinzufügen neuer Kanten dient, findet sich in Abbildung 3.2b. Ein Aufruf von *addEdge* fügt dabei implizit Start- und Endknoten in den Graphen ein. Wieder werden zwei Zusammenhänge vorausgesetzt, *addEdge-invar* fordert den Erhalt der Invarianten und *addEdge-correct* fordert ein korrektes Hinzufügen.

Die nächste Funktion dient dazu, die eingefügten Kanten effizient zu extrahieren. Durch einen Aufruf von *outEdges graph node* erhält man eine Liste der von *node* ausgehenden Kanten.

locale *graph-empty* = *graph* +
fixes *empty* :: *'g*
assumes *empty-invar*: *invar empty*
and *empty-correct*: *αe empty* = {}

(a) *graph-empty*

locale *graph-addEdge* = *graph* +
fixes *addEdge* :: *'g* ⇒ *'node* ⇒ *'edgeD* ⇒ *'node* ⇒ *'g*
assumes *addEdge-invar*: *invar g* ⟹ *invar* (*addEdge g f d t*)
and *addEdge-correct*: *invar g* ⟹ *e* ∈ *αe* (*addEdge g f d t*) ⟷ *e* = (*f*, *d*, *t*) ∨ *e* ∈ *αe g*

(b) *graph-addEdge*

locale *graph-outEdges* = *graph* +
fixes *outEdges* :: *'g* ⇒ *'node* ⇒ (*'node* × *'edgeD* × *'node*) *list*
assumes *outEdges-correct*: *invar g* ⟹ *set* (*outEdges g n*) = {(*f*, *d*, *t*). *f* = *n*} ∩ *αe g*

(c) *graph-outEdges*

Abbildung 3.2: Graphoperationen

Im Gegensatz zu den bisher vorgestellten Funktionen ist für *outEdges* keine Forderung bezüglich der Invariante nötig, da diese Funktion keine Graphmodifikation darstellt. Die Annahme *outEdges-correct*, die die Korrektheit garantiert, reicht aus.

3.3 Implementierung

Um die Graphenimplementierung möglichst flexibel zu gestalten, werden die Typen für Knoten und Kantendaten nicht festgelegt. Stattdessen sind die Funktionen auf abstrakten Typen definiert. Der Knoten wird mit *'n* bezeichnet, während *'ed* den Typ für Kantenannotationen darstellt. Aufgrund der verwendeten Datenstrukturen muss auf dem Knotentyp *'n* eine totale Ordnung definiert sein. In Isabelle/HOL wird dies dadurch ausgedrückt, dass *'n* als eine Instanz der Typklasse *linorder* definiert ist.

type-synonym (*'n*, *'ed*) *edge* = (*'n* × *'ed* × *'n*)
type-synonym (*'n*, *'ed*) *graphEntry* = ((*'ed* × *'n*) *list* × (*'ed* × *'n*) *list*)
type-synonym (*'n*, *'ed*) *graph* = (*'n*, (*'n*, *'ed*) *graphEntry*) *rbt*

Abbildung 3.3: Typenstruktur der Graphenimplementierung

Der Graph wird durch einen von Isabelle/HOL zur Verfügung gestellten Rot-Schwarz-Baum repräsentiert, der jedem Knoten die dazugehörigen aus- und eingehenden Kanten zuordnet. Diese werden in einfach verketteten Listen gespeichert, wobei der Ausgangsknoten bei ausgehenen Kanten und der Zielknoten bei eingehenden Kanten nicht mit den restlichen Kantendaten zusammen in der Liste gespeichert wird. Da der Index des Rot-Schwarz-Baumes dem Ausgangs-/Eingansknoten entspricht, ist dies nicht nötig und würde sogar eine Invariante erfordern, die sicherstellt, dass in jedem Eintrag nur passende Knoten gespeichert sind.

Die konkrete Typstruktur ist Abbildung 3.3 zu entnehmen. Der Kantentyp $('n, 'ed)$ *edge* wird als Tripel von Ausgangsknoten, Kantendatum und Zielknoten definiert. Zwei Listen von Paaren aus einem Kantendatum und einem Knoten werden zum Typ $('n, 'ed)$ *graphEntry* zusammengefasst, der einen Eintrag im Rot-Schwarz-Baum darstellt. Dabei sind in der ersten Liste die Kantendaten und Zielknoten der ausgehenden Kanten gespeichert. Die zweite Liste enthält die Ausgangsknoten und Kantendaten der eingehenden Kanten. Kombiniert man die Einträge dieser Listen mit dem Index, unter dem sie in dem Rot-Schwarz-Baum zu finden sind, erhält man die ursprünglichen Kanten. Insgesamt ergibt sich mit $'n$ als Index und $('n, 'ed)$ *graphEntry* einen Graph mit Typ $('n, ('n, 'ed)$ *graphEntry*$)$ *rbt*.

Als Präfix für die implementierten Funktionen wurde *mg* gewählt, das für *mapped graph* steht.

Die Funktion *mg-æe* ist in Abbildung 3.4a zu sehen. Sie bildet den Graphen auf eine Kantenmenge $('n, 'ed)$ *edge set* ab. Dabei wird der Rot-Schwarz-Baum zuerst auf eine Menge von Schlüssel-Wert-Paaren abgebildet, also Paare von $'n$ und $('n, 'ed)$ *graphEntry*. Für jedes dieser Paare wird die erste im Eintrag des Rot-Schwarz-Baumes gespeicherte Liste mit dem Index zu einer Menge ausgehender Kanten kombiniert. Die Vereinigung dieser Mengen entspricht dann allen Kanten im Graphen.

Die Funktion *mg-addEdge* (Abbildung 3.4d) arbeitet dreistufig. In *mg-addEdge* selbst werden *mg-addEdge-out* und *mg-addEdge-in* aufgerufen, die die aus- beziehungsweise eingehende Kante einfügen. Da die Funktion *mg-addEdge-out* analog zu *mg-addEdge-in* definiert ist, wurde sie bei der Abbildung ausgelassen.

In *mg-addEdge-out* wird überprüft, ob der Ursprungsknoten bereits als Index im Baum vorhanden ist. Ist dies der Fall, so wird anschließend die Funktion *mg-addEdge-out-map* aufgerufen, die den vorhandenen Eintrag um die

```
definition mg-αe :: ('n::linorder, 'ed) graph ⇒ ('n, 'ed) edge set where
  mg-αe graph = ⋃ ((λ(f, outEs, inEs). Pair f ‘ set outEs) ‘ set (entries graph))
```

(a) Definition von *mg-αe*

```
fun mg-invar :: ('n::linorder, 'ed) graph ⇒ bool where
  mg-invar g = True
```

(b) Definition von *mg-invar*

```
abbreviation mg-empty :: ('n::linorder, 'ed) graph where
  mg-empty ≡ empty
```

(c) Definition von *mg-empty*

```
definition
  mg-addEdge-in-map :: ('n :: linorder, 'ed) graph ⇒ 'n ⇒ 'ed ⇒ 'n ⇒ ('n, 'ed) graph
where mg-addEdge-in-map g f d t = map-entry t (λ(os, is). (os, (d, f) # is)) g

definition
  mg-addEdge-in-create :: ('n :: linorder, 'ed) graph ⇒ 'n ⇒ 'ed ⇒ 'n ⇒ ('n, 'ed) graph
where mg-addEdge-in-create g f d t = insert t ([], [(d, f)]) g

fun mg-addEdge-in :: ('n :: linorder, 'ed) graph ⇒ 'n ⇒ 'ed ⇒ 'n ⇒ ('n, 'ed) graph
where mg-addEdge-in g f d t =
  (if mg-hasNode g t
     then mg-addEdge-in-map g f d t
     else mg-addEdge-in-create g f d t)

fun mg-addEdge :: ('n::linorder, 'ed) graph ⇒ 'n ⇒ 'ed ⇒ 'n ⇒ ('n, 'ed) graph where
  mg-addEdge g f d t = mg-addEdge-out (mg-addEdge-in g f d t) f d t
```

(d) Definition von *mg-addEdge*

```
definition mg-outEdges :: ('n::linorder, 'ed) graph ⇒ 'n ⇒ ('n, 'ed) edge list where
  mg-outEdges g n =
    (let entry = lookup g n;
         implEdges = (if Option.is-none entry then [] else fst (the entry))
     in List.map (Pair n) implEdges)
```

(e) Definition von *mg-outEdges*

Abbildung 3.4: Definitionen der Graphenimplementierung

mg-empty	$= \mathcal{O}(1)$
mg-addEdge	$= \mathcal{O}(\log n)$
mg-outEdges	$= \mathcal{O}(\log n) + \mathcal{O}(e)$ (bei CFGs $\approx \mathcal{O}(\log n)$)

Abbildung 3.5: Laufzeiterwartungen der Graphenimplementierung

übergebene Kante ergänzt. Falls noch kein Eintrag existiert, wird stattdessen *mg-addEdge-out-create* aufgerufen, wodurch ein neuer Eintrag mit der Kante angelegt wird.

Die Funktion *mg-outEdges* (Abbildung 3.4e) selektiert die von einem gegebenen Knoten ausgehenden Kanten. Dabei wird der zum Knoten gehörige Eintrag abgerufen. Falls ein solcher Eintrag existiert, werden die ausgehenden Kanten selektiert und um den Ausgangsknoten ergänzt. Falls nicht, wird die leeren Liste zurückgegeben.

3.4 Überlegungen zur Performanz

Der vorgestellten Graphenimplementierung liegt ein Rot-Schwarz-Baum zugrunde, eine Variation eines binären Suchbäumes, die die drei Basisoperationen *insert*, *lookup* und *delete* in asymptotisch logarithmischer Zeit ausführt. Auch *map-entry* hat als Erweiterung von *lookup* ein logarithmisches Zeitverhalten. Ein leerer Baum lässt sich in konstanter Zeit erzeugen.

In Abbildung 3.5 sind die asymptotischen Laufzeiten der einzelnen Graphenoperationen abgebildet. Die Funktion *mg-empty* ruft nur *empty* auf, hat also konstanten Zeitbedarf. *mg-addEdge-out-map* besteht aus Anwendung der anonymen Funktion $\lambda(os, is).\ ((d, t) \cdot os, is)$, die konstanten Zeitbedarf hat, und einem Aufruf von *map-entry*, liegt also insgesamt in $\mathcal{O}(\log n) + \mathcal{O}(1) = \mathcal{O}(\log n)$, genau wie *mg-addEdge-out-create*, das nur aus einem *insert*-Aufruf besteht.

In *mg-addEdge-out* wird eine dieser beiden Funktionen und *lookup* aufgerufen, die Laufzeit ist $2 * \mathcal{O}(\log n) = \mathcal{O}(\log n)$. Dies ist auch die Laufzeit von *mg-addEdge-in*, das analog definiert ist. Da *mg-addEdge* nur diese Funktionen aufruft gilt auch *mg-addEdge* $= \mathcal{O}(\log n)$.

Bei der Bestimmung der ausgehenden Kanten findet zuerst ein *lookup* statt ($\mathcal{O}(\log n)$), woraufhin eine in Konstantzeit ablaufende Funktion auf eventu-

ell vorhandene Einträge angewandt wird (Mit e als Ausgangsgrad des Knoten: $e * \mathcal{O}(1) = \mathcal{O}(e)$). Insgesamt folgt eine Laufzeit von $\mathcal{O}(\log n + e)$.

Indem die kompletten Kanten in den Einträgen des Rot-Schwarz-Baumes gespeichert werden, könnte man diese Laufzeit auf $\mathcal{O}(\log n)$ reduzieren. Da jedoch bei Kontrollflussgraphen die Ausgangsgrade gewöhnlich konstant oder, wie beispielsweise bei einer *switch*-Anweisung, im Vergleich zur Knotenzahl sehr klein sind, kann für praktische Belange auch bei der vorgestellten Implementierung von einer Laufzeit von $\mathcal{O}(\log n)$ ausgehen.

3.5 Verifikation

Um diese Graphenimplementierung zu verifizieren, ist nur ein Korrektheitsnachweis der einzelnen Graphenoperationen nötig, da die in Abbildung 3.4b dargestellte Invariante trivial ist.

Das wichtigste Lemma im Beweis der Graphenimplementierung zeigt den Zusammenhang zwischen dem Rückgabewert von *mg-αe* und dem Zustand der Datenstruktur.

$x \in \textit{mg-αe}\ g \Longrightarrow \exists\, \textit{outEdges}\ \textit{inEdges}.$
$\textit{lookup}\ g\ (\textit{fst}\ x) = \textit{Some}\ (\textit{outEdges}, \textit{inEdges})$
$\wedge\ \textit{snd}\ x \in \textit{set}\ \textit{outEdges}$

Mit diesem Lemma läuft der Beweis zur Locale *graph-empty* dann komplett automatisch ab.

Der Beweis zu *graph-addEdge* ist ein wenig komplexer. Parallel zur Definition von *mg-addEdge* wird bei den Hilfsfunktionen beginnend die Korrektheit bewiesen. Dabei werden jeweils die Bedingungen, unter denen die Funktionen aufgerufen werden, für die Korrektheit vorrausgesetzt. Mit jedem Schritt in Richtung von *addEdge* werden diese Bedingungen dann reduziert, was dann die folgende Aussage liefert:

$e \in \textit{mg-αe}\ (\textit{mg-addEdge}\ g\ f\ d\ t) \longleftrightarrow e = (f\ , d, t) \vee e \in \textit{mg-αe}\ g$

Mit dieser Aussage lässt sich die Interpretation der Locale *graph-addEdge* leicht beweisen.

Der Beweis zu *mg-outEdges* beginnt mit dem Fall, dass der übergebene Knoten nicht Teil des Graphen ist. Der Rot-Schwarz-Baum hat in diesem Fall kei-

nen dazugehörigen Eintrag. Falls der Knoten vorhanden ist, wird zuerst gezeigt, dass die zurückgegebenen Kanten tatsächlich vom angegebenen Knoten ausgehen.

$$(f, d, t) \in \textit{mg-outEdges}\ g\ n \Longrightarrow n = f$$

Darauf aufbauend werden dann die Aussagen

$$e \in \textit{mg-outEdges}\ g\ n \Longrightarrow e \in \textit{mg-}\alpha e\ g$$

und

$$[\![\textit{lookup}\ g\ \textit{from} = \textit{Some}\ (\textit{outEdges}, \textit{inEdges});\ (\textit{data}, \textit{to}) \in \textit{outEdges}]\!]$$
$$\Longrightarrow (\textit{from}, \textit{data}, \textit{to}) \in \textit{mg-outEdges}\ g\ \textit{from}$$

bewiesen. Sie geben den Zusammenhang zwischen dem Zustand des Graphen und des Resultats von *mg-outEdges* an.

Mit diesen Aussagen lässt sich nun die Korrektheit der Interpretation von *graph-outEdges* beweisen, was die Verifikation der Graphenimplementierung abschließt.

Es ist für den Beweis wichtig, den richtigen Eintrag im Rot-Schwarz-Baum zu betrachten. Der automatische Beweisprozess hat jedoch Schwierigkeiten, diesen zu finden. Deshalb musste hier stärker auf den Beweisprozess eingewirkt werden, als dies bei den vorherigen Fällen nötig war.

3.6 Ausblick

Die in diesem Kapitel vorgestellten Operationen stellen offensichtlich keine vollständige, benutzerfreundliche Graphenbibliothek dar. Vielmehr soll hier eine Basis geschaffen werden, die für weitere Anwendungen entsprechend erweitert werden kann. Diese Überlegung ist auch in die in Kapitel 3 vorgestellte Implementierung eingeflossen.

Für eine Weiterverarbeitung der Kontrollflussgaphen wird neben der bereitgestellten Funktion *outEdges* wahrscheinlich noch eine Funktion benötigt, welche die eingehenden Kanten eines Knotens bestimmt. Implementiert man

dann noch eine Tiefen- und Breitensuche in beide Richtungen, so sollten die meisten Anwendungsfälle abgedeckt sein. Darüber hinaus könnte eine ausführbare Projektion auf eine Kantenliste in manchen Situationen hilfreich sein, da sie wesentlich performanter zu implementieren ist als eine Tiefen- oder Breitensuche.

Die vorgestellte Implementierung speichert bereits die Rückwärtskanten, was eine Implementierung von *inEdges* vereinfacht. Dazu muss jedoch noch die Invariante erweitert werden, da sie in der vorliegenden Variante nicht sicherstellt, dass zu jeder ausgehenden Kante auch die entsprechende eingehende Kante im Graphen vorhanden ist. Diese Ergänzung wird keine Änderungen an den vorhandenen Beweisen benötigen. Da eine rückwärtsgerichtete Suche auf *inEdges* aufbaut, ist eine erweiterte Invariante auch für die Implementierung dieser Algorithmen notwendig. Eine vorwärtsgerichtete Tiefen- und Breitensuche und eine Kantenliste sollte jedoch ohne Änderungen an den vorhandenen Definitionen und Aussagen möglich sein.

KAPITEL 4

Der Konstruktionsalgorithmus

Als Nächstes wird der Konstrukionsalgorithmus vorgestellt und seine asymptotische Laufzeit anhand des Codes hergeleitet. Danach wird noch auf den Korrektheitsbeweis eingegangen.

Der Algorithmus baut auf der in Kapitel 3 erläuterten Graphenimplementierung auf. Um sie zu nutzen wird der Algorithmus in einer Locale implementiert, die die Graphenlocales erweitert.

Für die Knoten wird dabei der Typ *w-node*, für die Kanten der Typ *w-edge* verwendet. Dadurch ergibt sich folgende Locale, in der der Algorithmus definiert wird.

locale *While-Graph* = *graph-empty αe invar empty* + *graph-addEdge αe invar addEdge*
for *αe* :: *'graph* ⇒ *w-edge set*
and *invar* :: *'graph* ⇒ *bool*
and *empty* :: *'graph*
and *addEdge* :: *'graph* ⇒ *w-node* ⇒ *state edge-kind* ⇒ *w-node* ⇒ *'graph*

Die **for**-Klauseln werden benötigt, um in den beiden erweiterten Locales *graph-empty* und *graph-addEdge* die selben Graphen-, Knoten- und Kantentypen zu verwenden.

Die Hilfsfunktionen *addWEdge* und *addWEdges* dienen dazu, bequem Kanten hinzuzufügen.

addWEdge g (n, et, n') = *addEdge g n et n'*

addWEdges g es = *foldl addWEdge g es*

```
definition stdEdges :: w-edge set where
 stdEdges ≡ {((-Entry-), (λx. True)√, (- 0 -)), ((-Entry-), (λx. False)√, (-Exit-))}
fun withStdEdges :: 'graph ⇒ 'graph where
 withStdEdges g = addWEdge ( addWEdge g
 ((-Entry-), (λs. True)√, (-0-)))
 ((-Entry-), (λs. False)√, (-Exit-))

definition idTo :: w-node ⇒ w-node list ⇒ w-edge list where
 idTo t = List.map (λn.(n, ⇑id, t))

definition closeExits :: w-node list ⇒ 'graph ⇒ 'graph where
 closeExits preExits g = addWEdges g (idTo (-Exit-) preExits)
```

Abbildung 4.1: Definition von Hilfsfunktionen für *build*

Die Funktion *build* konvertiert einen abstrakten Syntaxbaum vom Typ *cmd* in einen Kontrollflussgraphen des generischen Typs *'graph*. Sie ist in Abbildung 4.3 dargestellt und soll im Folgenden erklärt werden. Als Minimalbeispiel wird dabei der Syntaxbaum *Skip* verwendet, dessen Kontrollflussgraph in Abbildung 2.4 zu sehen ist. Nach der Erklärung wird noch ein umfangreicheres Beispiel präsentiert.

Der Kern von *build* ist die Hilfsfunktion *build'*. Diese ist vom Typ *cmd* ⇒ *nat* ⇒ *'graph* ⇒ *nat* × *w-node list* × *'graph*, wobei der erste Parameter der zu verarbeitenden Syntaxbaum ist. Der zweite Parameter ist die erste Knotennummer, die verwendet werden darf. Falls die Wurzel des Syntaxbaumes umgewandelt werden soll, ist dieser Parameter *0*. Der dritte Parameter ist der Graph, der um die neuen Knoten und Kanten erweitert werden soll. Zu Beginn wird hier der leere Graph *empty* angegeben.

Der Rückgabewert ist ein Tripel vom Typ *nat* × *w-node list* × *'graph*. Das erste Element des Tupels die Anzahl der durch die Operation hinzugefügten Knoten. Dabei werden sowohl Knoten berücksichtig, die durch neue Kanten hinzugefügt wurden, als auch Knoten, die im zweiten Teil des Tupels enthalten sind. Das zweite Element ist eine Liste der Knoten, die mit der nächsten Anweisung oder (*-Exit-*) verbunden werden sollen. Da diese Verbindungen immer den Kantenwert ⇑*id* haben, der einer Null-Operation entspricht, braucht dieser nicht explizit angegeben werden. Bei Skip ist dies nur (*- 0 -*). Als drittes enthält das Tupel noch den modifizierte Graph, der im Fall von Skip dem ursprünglichen Graphen entspricht.

```
fun build' :: cmd ⇒ nat ⇒ 'graph ⇒ nat × w-node list × 'graph where

 build' Skip count g = (1, [(- count -)], g)

| build' (V:=e) count g =
 (2, [(- count + 1 -)], addEdge g (- count -) ⇑(λs. s(V:=(interpret e s))) (- count + 1 -))

| build' (c1;;c2) count g =
 (let (count1, nodes1, g) = build' c1 count g;
    g = addWEdges g (idTo (- count + count1 -) nodes1);
    (count2, nodes2, g) = build' c2 (count + count1) g
  in (count1 + count2, nodes2, g))

| build' (if (b) c1 else c2) count g =
   (let g = addWEdge g ((- count -), (λs. interpret b s = Some true)√, (- count + 1 -));
      (count1, nodes1, g) = build' c1 (count + 1) g;
      g = addWEdge g
        ((- count -), (λs. interpret b s = Some false)√, (- count + count1 + 1 -));
      (count2, nodes2, g) = build' c2 (count + count1 + 1) g
    in (count1 + count2 + 1, nodes1 @ nodes2, g))

| build' (while (b) c) count g =
   (let g = addWEdge g ((- count -), (λs. interpret b s = Some true)√, (- count + 2 -));
      g = addWEdge g ((- count -), (λs. interpret b s = Some false)√, (- count + 1 -));
      (count1, nodes, g) = build' c (count + 2) g;
      g = addWEdges g (idTo (- count -) nodes)
    in (count1 + 2, [(- count + 1 -)], g))

abbreviation getCount :: nat × w-node list × 'graph ⇒ nat where getCount ≡ fst
abbreviation getNodes :: nat × w-node list × 'graph ⇒ w-node list where getNodes g ≡
fst (snd g)
abbreviation getG :: nat × w-node list × 'graph ⇒ 'graph where getG g ≡ snd (snd g)
```

Abbildung 4.2: Definition von *build'* und Hilfsfunktionen

```
fun build :: cmd ⇒ 'graph where
 build cmd =
   (let (ig, nodes, g) = build' cmd 0 empty
    in withStdEdges (closeExits nodes g))
```

Abbildung 4.3: Definition von *build*

```
x := 5;
run := true;
while (run) {
  if (x < 10) {
    x := x + 2;
  } else {
    run := false;
  }
}
```

(a) Code

```
''x'':=Val (Intg 5);;
''run'':=Val true;;
while (Var ''run'') (
  if (Var ''x'' «Less» Val (Intg 10)) (
    ''x'':= Var ''x'' «Add» Val (Intg 2)
  ) else (
    ''run'':=Val false
  )
)
```

(b) Syntaxbaum

Abbildung 4.4: Beispielprogramm

Um den Graphen zu vervollständigen, müssen die als drittes Tupel-Element zurückgegebenen Kanten mit (*-Exit-*) verbunden werden und die zwei Standardkanten

$((\text{-}Entry\text{-}), (\lambda x.\ True)_{\surd}, (\text{-}\ 0\ \text{-}))$ und $((\text{-}Entry\text{-}), (\lambda x.\ False)_{\surd}, (\text{-}Exit\text{-}))$

hinzugefügt werden. Dies übernehmen die Funktion *closeExits* und *withStdEdges*. Ihre Definition ist Abbildung 4.1 zu entnehmen.

4.1 Ein umfangreicheres Beispiel

An dieser Stelle wird der Algorithmus an einem Beispiel mit sechs Anweisungen genauer erklärt. Der betrachtete Code ist zusammen mit dem dazugehörigen Syntaxbaum in Abbildung 4.4 abgebildet. Dabei ist ;; rechtsassoziativ, so dass jede Zeile mit dem nachfolgenden Rest ihres Blocks verbunden

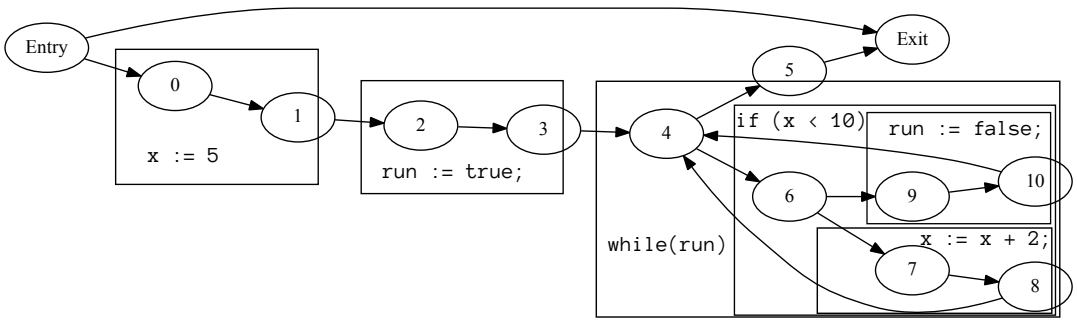

Abbildung 4.5: Kontrollflussgraph des Beispielprogramms

ist. In Abbildung 4.5 findet sich eine grafische Repräsentation des erzeugten Kontrollflussgraphen, auf dessen Knotennummern im folgenden Bezug genommen wird. Die Kästen zeigen, zu welcher Anweisung die Knoten gehören. Liegen Knoten nur teilweise innerhalb eines Kastens, wurden sie von *build′* zurückgegeben um zum nächsten Programmteil verbunden zu werden.

In *build* wird zuerst *build′* aufgerufen. Für die Anweisung „`x := 5;`“ wird dabei die Kante

$$((\text{-}\ 0\ \text{-}), \Uparrow\lambda s.\ s(V := \mathit{interpret}\ (\mathit{Val}\ (\mathit{Intg}\ 5))\ s), (\text{-}\ 1\ \text{-}))$$

eingefügt und der Knoten (- *1* -) zurückgegeben, um ihn weiterzuverbinden. Die Anweisung „`run := true;`“ wird genauso verarbeitet, für sie werden (- *2* -) und (- *3* -) hinzugefügt. Dabei wurden (- *1* -) und (- *2* -) in der zweiten Zeile der *build′*-Klausel für ;; verbunden. Die *while*-Anweisung beginnt mit Knoten (- *4* -) und fügt die Kanten

$$((\text{-}\ 4\ \text{-}), (\lambda s.\ \mathit{interpret}\ (\mathit{Var}\ ''run'')\ s = \mathit{Some}\ \mathit{true})_{\surd}, (\text{-}\ 6\ \text{-}))$$

ein, die verfolgt wird, falls die Bedingung erfüllt ist. Für den Fall dass sie das nicht ist, wird die Kante

$$((\text{-}\ 4\ \text{-}), (\lambda s.\ \mathit{interpret}\ (\mathit{Var}\ ''run'')\ s = \mathit{Some}\ \mathit{false})_{\surd}, (\text{-}\ 6\ \text{-}))$$

hinzugefügt. Der Knoten (- *5* -) wird zur späteren Verbindung zurückgegeben.

Der Rumpf der *while*-Schleife wird beginnend mit (- *6* -) verarbeitet. Für die *if*-Anweisung wird eine Kanten zu dem Knoten (- *7* -) eingefügt, der gefolgt wird, falls *Var ''x''* «*Less*» *Val* (*Intg 10*) wahr ist. Die Kante von (- *7* -) nach (- *8* -) wird nach den bekannten Regeln für Zuweisungen eingefügt, genau wie die Kante von (- *9* -) nach (- *10* -), die über den von *if* hinzugefügten Knoten (- *9* -) erreichbar ist. Die *if*-Anweisung reicht die zu verbindenden Knoten beider Rümpfe weiter zur *while*-Schleife, die sie mit ihrem Eingangsknoten (- *4* -) verbindet.

Für ;; wird nun auch Knoten (- *3* -) mit (- *4* -) verbunden und der von der *while*-Schleife zurückgegebene Knoten (- *5* -) an *build* weitergereicht. Wieder dort angekommen wird nun (- *5* -) zu (-*Exit*-) verbunden. Nachdem dann noch die Kanten $((\text{-}\mathit{Entry}\text{-}), (\lambda x.\ \mathit{True})_{\surd}, (\text{-}\ 0\ \text{-}))$ und $((\text{-}\mathit{Entry}\text{-}), (\lambda x.\ \mathit{False})_{\surd}, (\text{-}\mathit{Exit}\text{-}))$ hinzugefügt wurden, ist die Konstruktion abgeschlossen.

4.2 Laufzeitabschätzungen

Nun zur asymptotischen Laufzeit von *build*. Dabei steht l (abgeleitet von *length*) für die Anzahl der Anweisungen im While-Programm und n für die Anzahl der Knoten im Syntaxbaum. Dabei gilt $n = \mathcal{O}(l)$, da jede Anweisung[1] nur eine konstante Zahl von Knoten einführt.

Der Konstruktionsalgorithmus fügt für jeden Eintrag im Syntaxbaum eine konstante Anzahl Knoten in den Graphen ein. Dazu kommen dann noch (*-Entry-*) und (*-Exit-*). Der Ausgangsgrad im Kontrollflussgraphen ist immer kleiner oder gleich zwei, also wird auch nur eine lineare Zahl von Kanten eingefügt. Da keine Kanten entfernt werden, ergibt sich daraus eine Laufzeit von $\mathcal{O}(l * addEdge)$.

Zusätzlich werden jedoch im Falle einer *if*-Anweisung zwei Listen von zu verbindenden Knoten konkateniert, was in Linearzeit bezüglich der Länge der ersten Liste abläuft. Die Zahl der zum Ausgang des Teilbaumes zu verbindenenden Knoten kann beliebig wachsen, da im Falle einer *if*-Anweisung jeweils die Knoten beider Blöcke mit der nächsten Anweisung verknüpft werden müssen. Betrachtet man ein Programm mit einer *if*-Anweisung, die sowohl im *then*- als auch im *else*-Teil wiederum mit einer *if*-Anweisung endet, so finden sich vier Pfade aus der äußeren *if*-Anweisung. Erhöht man nach diesem Muster die Schachtelungstiefe weiter, so lässt sich die Zahl der zu verbindenden Knoten beliebig erhöhen. Hierbei handelt es sich jedoch um ein konstruiertes Problem, das kaum Relevanz für die Praxis hat. Dort sind Schachtelungstiefen von vielen Ebenen die Ausnahme, und selbst diese sind im Vergleich zur Programmlänge winzig.

Für die Praxis ergibt sich also eine Laufzeit von $\mathcal{O}(l * addEdge)$.

Betrachtet man nun die in Kapitel 3.3 vorgestellte Graphenimplementierung mit den in Abbildung 3.5 vorgestellten Laufzeiten, ergibt sich eine Laufzeit von $\mathcal{O}(l * addEdge) = \mathcal{O}(l * \log l)$.

4.3 Verifikation

Um die Korrektheit von *build* zu beweisen, ist zu zeigen, dass die Konstruktionsfunktion die gleichen Kontrollflussgraphen wie die induktive Definition

[1] „Anweisung“ bezeichnet die While-Anweisungen *if*, *while* und Zuweisungen.

build′-count-correct: $getCount\ (build'\ cmd\ count\ g) = \#{:}cmd$

build′-nodes: $[\![invar\ g; unfinished\ g]\!]$
$\Longrightarrow cmd \vdash n\ -\Uparrow id\rightarrow\ (\text{-}Exit\text{-}) \longleftrightarrow n \oplus count \in set\ (getNodes\ (build'\ cmd\ count\ g))$

build′-graph: $[\![n' \neq (\text{-}Exit\text{-}); (n, et, n') \notin stdEdges; invar\ g; unfinished\ g]\!]$
$\Longrightarrow (n \oplus count, et, n' \oplus count) \in \alpha e\ (getG\ (build'\ cmd\ count\ g))$
$\longleftrightarrow cmd \vdash n\ -et\rightarrow\ n' \vee (n \oplus count, et, n' \oplus count) \in \alpha e\ g$

Abbildung 4.6: Lemmata für *build′*

aus dem Slicing-Frameworks generiert und dass die erstellten Graphen die Invariante erfüllen. Der Beweis der Invariante läuft komplett automatisch ab.

lemma *invar* (*build cmd*) **by** *simp*

Um die Korrektheit zu beweisen ist die Mengengleichheit der induktiven Definition mit der Bildmenge von *build* zu zeigen. Formal ausgedrückt:

$cmd \vdash n\ -et\rightarrow\ n' = ((n, et, n') \in \alpha e\ (build\ cmd))$

Auf die Beweise der einzelnen Aussagen wurde in diesem Kapitel nicht genauer eingegangen, da sie zum größten Teil aus Induktion über den Typ des Kontrollflussgraphens mit einer Fallunterscheidung über dessen Konstruktionsregeln besteht. Dadurch sind die Beweise lang und tragen nicht nennenswert zum Verständnis bei.

Als Startpunkt wurde der Invariantenerhalt und die Korrektheit der Hilfsfunktionen nachgewiesen. Darüber hinaus wurde eine weitere Einschränkung definiert, die besagt, dass ein Graph keine Kanten von oder zu (*-Entry-*) oder (*-Exit-*) enthält. Formal ist sie definiert durch das Prädikat *unfinished*.

unfinished g =
$(\forall n\ et\ n'.\ (n, et, n') \in \alpha e\ g \longrightarrow n \neq (\text{-}Entry\text{-}) \wedge n' \neq (\text{-}Exit\text{-}))$

Diese Einschränkung muss von *build′* erhalten werden, damit nachgewiesen werden kann, dass keine falschen Kanten hinzugefügt wurden.

Der Kern des Beweises sind die Lemmata *build′-count-correct*, *build′-nodes* und *build′-graph*, die in Abbildung 4.6 dargestellt sind. Sie zeigen, dass die

von *build* zurückgegebenen Werte der dokumentierten Schnittstelle entsprechen. Dabei erforderten die Beweise von *build′-unfinished*, *build′-nodes* und *build′-graph* jeweils eine Induktion über den Datentyp *cmd*.

In bestimmten Fällen muss ein hinzugefügter Knoten hergenommen werden, um den Beweis führen zu können. Dies kann ein Problem darstellen, beispielsweise bei der Betrachtung von

$$[\![c_2 \vdash n\ -et\rightarrow\ n';\ n \neq (\text{-}Entry\text{-})]\!] \Longrightarrow c_1;; c_2 \vdash n \oplus \#{:}c_1\ -et\rightarrow\ n' \oplus \#{:}c_1$$

Hier muss aus dem Knoten $n \oplus \#{:}c1$ der Knoten n rekonstruiert werden. Jedoch liegt der Knoten nicht in der Additionsform vor, sondern als Knoten $(\text{-}\ m\ \text{-})$. Da die Knotenindizes natürliche Zahlen sind, kann nicht ohne weiteres $n = (\text{-}\ m - \#{:}c1\ \text{-})$ vorausgesetzt werden, da $m - \#{:}c1$ nicht definiert sein könnte.

Ein Beweis, dass *build′* den zweiten Parameter *count* korrekt berücksichtigt und nur Knoten mit Nummern größer oder gleich diesem hinzufügt, löst dieses Problem.

$$[\![((\text{-}\ n\ \text{-}), et, n') \in \alpha e\ (getG\ (build'\ cmd\ count\ g));\ ((\text{-}\ n\ \text{-}), et, n') \notin \alpha e\ g;\ invar\ g]\!] \Longrightarrow count \leq n$$

Daraus lässt sich folgern, dass, falls Kanten hinzugefügt wurden, mindestens eine davon von einem Knoten $(\text{-}\ c\ \text{-})$ mit $count \leq c$ ausgeht und mindestens eine zu einem solchen Knoten hinführt. Ersterer Fall ist ausgedrückt durch

lemma
$[\![(n, et, n') \in \alpha e\ (getG\ (build'\ cmd\ count\ g));\ (n, et, n') \notin \alpha e\ g;\ unfinished\ g;\ invar\ g]\!] \Longrightarrow \exists c.\ n = (\text{-}\ c\ \text{-}) \wedge count \leq c$
by *auto*

Für den zweiten Fall ist $n = (\text{-}\ c\ \text{-})$ durch $n' = (\text{-}\ c\ \text{-})$ ersetzt.

Mit der Nebenbedingung $count \leq c$ lassen sich dann die oben erwähnten Fälle verarbeiten und der Beweis von *build′-graph* vollenden.

Mit *build′-nodes*, *build′-graph* und der Aussage, dass die Knoten (*-Entry-*) und (*-Exit-*) nicht Teil des Graphen sind, lässt sich dann das eingangs beschriebene Äquivalenztheorem beweisen.

KAPITEL 5

Laufzeitmessungen

In diesem Abschnitt werden die in Kapitel 4.2 vorgenommenen Überlegungen zur Laufzeit experimentell belegt. Die Versuche fanden dabei auf einem System mit einer Intel Core i5-2380P CPU mit vier Kernen und 3.1 GHz statt. Es waren 8 Gigabyte Arbeitsspeicher im Dual-Channel-Modus verbaut, der Speichertakt betrug 1333 MHz. Das verwendete Betriebssystem ist Gentoo Linux, das zum Zeitpunkt der Messungen auf dem Linux-Kernel in der Version 3.4.2 basiert. Es wurde die Isabelle/HOL-Distribution in der Version 2012 verwendet.

Die Testdaten wurden von einem Skript generiert, das, ausgehend von einem leeren Programm, so lange Anweisungen hinzufügt, bis die gewünschte Programmgröße erreicht ist. Dabei wird zu 60 % eine Zuweisung, zu jeweils 10 % eine *if*-Anweisung mit und ohne *else*-Teil und zu 20 % eine *while*-Anweisung angehängt. Jeder Block innerhalb einer *if*- oder *while*-Anweisung enthält zwischen zwei und einhundert (oder der maximal verfügbare Anzahl) Anweisungen. Auch sie sind im genannten Verhältnis verteilt, wobei ab einer Schachtelungstiefe von acht nur noch Zuweisungen eingefügt werden. Die Gesamtzahl an Anweisungen wird nie überschritten.

Die so generierten Theory-Dateien rufen zuerst *parseWhileString* auf, was einen Syntaxbaum generiert. Dann wird über einen ML-Block der Code für *build* generiert, dem der Code für den von *parseWhileString* erstellten Syntaxbaum übergeben wird. Gemessen wird dann die Zeit, die der Aufruf von *build* benötigt.

Mit dieser Methode wurden zwei Datensätze generiert, die jeweils zwei mal gemessen wurden. Die Ergebnisse sind in Abbildung 5.1 zu sehen, die drei

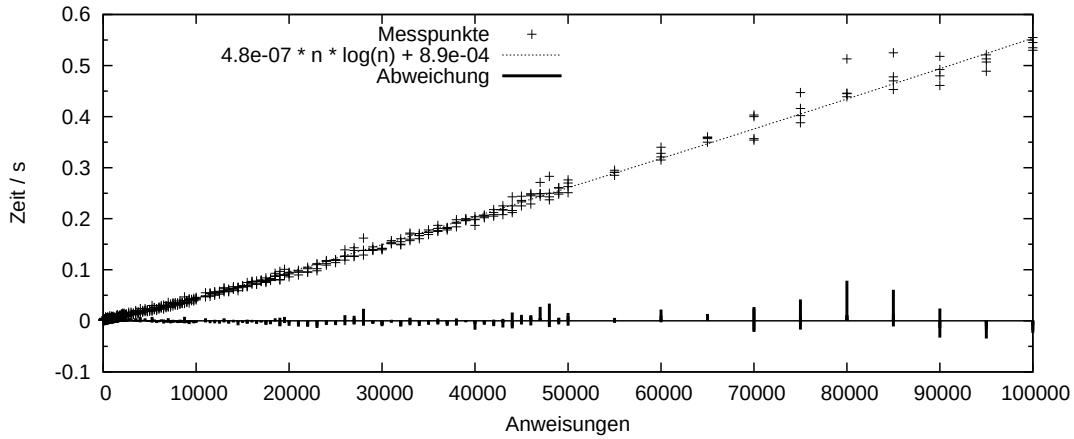

Abbildung 5.1: Zeitmessung von *build*

Graphen enthält. Als Erstes sind die einzelnen Messpunkte eingezeichnet. Um einen Vergleich mit der Vermutung zu ermöglichen wurde eine parametrische Regression basierend auf der Funktion

$$a * n * \log n + b$$

durchgeführt. Die entsprechenden Werte für a und b sind den Abbildungen zu entnehmen. Durch ein Stabdiagramm des arithmetischen Mittels der Abweichung von dieser Funktion soll der Vergleich weiter unterstützt werden.

Insgesamt lässt sich aus der Abbildung ablesen, dass die aus der Codeanalyse hergeleitete Laufzeit erreicht wurde.

Obwohl eine Laufzeit von etwas über einer halben Sekunde sicherlich für die meisten Anwendungsfälle ausreicht, könnte man an einigen Stellen Verbesserungen vornehmen. So könnte zum Beispiel die *addEdge*-Funktion so verändert werden, dass kein separater *lookup* und *insert* beziehungsweise *map-entry* nötig ist. Durch eine geschicktere Speichernutzung könnte man die Zeit reduzieren, die der Garbage Collector laufen muss. Drei Testläufe mit 100 000 Anweisungen zeigen, dass diese Zeit in etwa vier mal so lang wie die produktiv genutzte Zeit ist.

Jedoch wäre eine Zeitersparnis an dieser Stelle wenig zielführend. Bei einem Testprogramm mit 100 000 Anweisungen dauert das Parsen eine Minute und 35, 7 Sekunden; dazu kommen 48, 8 Sekunden, die im Garbage Collector verbracht wurden.

Das Generieren des ML-Codes dauerte 4 Minuten und 4,3 Sekunden zuzüglich 3 Minuten und 52,9 Sekunden für Garbage Collection. Im Vergleich zu den 0,5 Sekunden produktive Arbeit und 2,3 Sekunden Garbage Collection für die Konstruktion ist dieser Aufwand enorm. Obwohl das Parsen im Produktiveinsatz oft in anderer Form stattfindet, so muss um den Inhalt des Kontrollflussgraphens zu erhalten der Codegenerator eingesetzt werden. Eine weitere Optimierung von *build* ist also wahrscheinlich nicht sehr sinnvoll.

KAPITEL 6

Fazit

Im Rahmen dieser Arbeit konnte ich eine Graphenschnittstelle zusammen mit einer effizienten Implementierung entwickeln. Darauf aufbauend implementierte ich einen Algorithmus, der abstrakte Syntaxbäume in Kontrollflussgraphen überführt und belegte dessen Korrektheit. Nach Betrachtung des Codes konnte ich eine Zeitkomplexität von $\mathcal{O}(n \log n)$ feststellen, die sich experimentell bestätigen ließ.

Die Verifikation der implementierten Algorithmen erlaubt ein tiefes Vertrauen in deren Korrektheit und ermöglicht auf ihnen aufbauende Arbeiten im Rahmen von verifiziertem Programmslicing.

Diese Arbeit alleine reicht natürlich nicht aus, um verifiziertes Slicing praktikabel zu machen. In der Zukunft werden verifizierte Algorithmen zum Erkennen von Daten- und Kontrollabhängigkeit und zur Weiterverarbeitung der um diese Abhängigkeiten erweiterten Kontrollflussgraphen implementiert werden müssen. Dazu wird auch eine Ergänzung der Graphenschnittstelle um zusätzliche Operationen nötig sein. Ein anderer nächster Schritt könnte es sein, die Konstruktionsroutinen auf andere Programmiersprachen auszuweiten. Mögliche Sprachen wären zum Beispiel CoreC++ [5] oder Jinja [2], da sie bereits vom Slicing-Framework unterstützt werden.

Zusammenfassend konnte ich im Rahmen dieser Arbeit einen Algorithmus zur Generierung von Kontrollflussgraphen entwickeln, der den Anforderungen im vollen Umfang gerecht wird und damit einen Beitrag zu verifiziertem Slicing leisten.

Literaturverzeichnis

[1] BALLARIN, Clemens: Interpretation of locales in isabelle: theories and proof contexts. In: *Proceedings of the 5th international conference on Mathematical Knowledge Management*. Berlin, Heidelberg : Springer-Verlag, 2006 (MKM'06). – ISBN 3–540–37104–4, 978–3–540–37104–5, 31–43

[2] KLEIN, Gerwin ; NIPKOW, Tobias: A machine-checked model for a Java-like language, virtual machine, and compiler. In: *ACM Trans. Program. Lang. Syst.* 28 (2006), Juli, Nr. 4, 619–695. http://dx.doi.org/10.1145/1146809.1146811. – DOI 10.1145/1146809.1146811. – ISSN 0164–0925

[3] LAMMICH, Peter ; LOCHBIHLER, Andreas: The isabelle collections framework. In: *Proceedings of the First international conference on Interactive Theorem Proving*. Berlin, Heidelberg : Springer-Verlag, 2010 (ITP'10). – ISBN 3–642–14051–3, 978–3–642–14051–8, 339–354

[4] NIPKOW, Tobias ; WENZEL, Markus ; PAULSON, Lawrence C.: *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg : Springer-Verlag, 2002. – ISBN 3–540–43376–7

[5] WASSERRAB, Daniel: CoreC++. In: KLEIN, Gerwin (Hrsg.) ; NIPKOW, Tobias (Hrsg.) ; PAULSON, Lawrence (Hrsg.): *The Archive of Formal Proofs*. http://afp.sf.net/entries/CoreC++.shtml, Mai 2006. – Formal proof development

[6] WASSERRAB, Daniel: *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security*, Karlsruher Institut für Technologie, Fakultät für Informatik, Diss., Oktober 2010. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000020678

[7] WEISER, Mark: Program slicing. In: *Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA : IEEE Press, 1981 (ICSE '81). – ISBN 0–89791–146–6, 439–449

[8] WENZEL, Makarius: *The Isabelle/Isar Reference Manual*. http://www.cl.cam.ac.uk/research/hvg/isabelle/dist/Isabelle2012/doc/isar-ref.pdf. Version: Mai 2012

Abbildungsverzeichnis

Erklärung

HIERMIT erkläre ich, Kevin-Simon Kohlmeyer, dass ich die vorliegende Bachelorarbeit selbstständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Ort, Datum

Unterschrift