Karlsruher Institut für Technologie

Fakultät für Informatik
Bachelorarbeit

Verifizierte Summary Kanten Berechnung in System Dependence Graphs

Verified Summary Edge Computation in System Dependence Graphs

Stefan Altmayer

Betreuer:
Prof. Dr.-Ing. Gregor Snelting
Denis Lohner

17. Dezember 2012

Abstract

In dieser Arbeit wird ein verifizierter Algorithmus zum Finden von Summary Kanten in System Dependence Graphs (SDG) vorgestellt. Der Algorithmus ist an das Verfahren von Reps, Horwitz, Sagiv und Rosay aus ihrem Paper "Speeding up Slicing"[8] angelehnt. Die SDGs richten sich nach dem SDG-Framework von Wasserrab[10] und dem Graph-Framework von Kohlmeyer [6]. Die zur Verifikation benötigten Beweise werden erläutert, die wesentlichen Invarianten des Algorithmus werden dazu herausgestellt. Außerdem wird der Algorithmus über einen Code Generator in Haskell exportiert und ist somit ausführbar. Alle Beweise wurden im Theorembeweiser Isabelle[7] geführt.

Inhaltsverzeichnis

Inhaltsverzeichnis **5**

1 Einleitung **7**
1.1 Slicing im Überblick 7
1.2 Zielsetzung dieser Arbeit 8

2 Grundlagen **9**
2.1 Slicing und Summary Kanten 9
2.2 Program Dependence Graphs 9
2.3 System Dependence Graph 10
2.4 Realisierbare Pfade in SDGs 12
2.5 Effiziente P-Summary Kanten Berechnung 13
2.6 Isabelle 17
2.7 Verwendete Frameworks 18
2.8 Formalisierung des Algorithmus 19

3 Beweise zum Algorithmus **23**
3.1 Termination 23
3.1.1 Allgemeines 23
3.1.2 Teil 1 - Eine Größe für *pathEdge* 24
3.1.3 Teil 2 - Eine Größe für *workList* 25
3.1.4 Die finale Größenfunktion 26
3.2 Korrektheit des Algorithmus 27
3.2.1 Lokale Annahmen des Beweises 27
3.2.2 Prädikat isStartState 27
3.2.3 Invariante sndInFormalOuts 28
3.2.4 Bedingungen zum Finden von P-Summary Kanten . . 28
3.2.5 Die finale Invariante 29
3.3 Vollständigkeit des Algorithmus 32
3.3.1 Beeinflussung von Pfaden 32
3.3.2 Induktionsregeln für matched 32
3.3.3 Vollständigkeit bezüglich matched 33
3.3.4 Vollständigkeit aller Summary Kanten 34

4 Fazit und Ausblick **35**
4.1 Fazit 35
4.2 Generierung des Codes 35
4.3 Ausblick 37

5 Anhang **39**
5.1 Namen der Lemmas in Isabelle 39

6 Literaturverzeichnis **41**

Kapitel 1

Einleitung

1.1 Slicing im Überblick

Moderne Computersysteme werden immer mehr in sicherheitskritischen Anwendungen genutzt. In diesen Anwendungen können Fehlfunktion hohe Kosten mit sich tragen, zum Beispiel bei der Programmierung von integrierten Schaltkreisen, Verschlüsselungsprotokollen oder Programmen zur Verwaltung von Finanztransaktionen. Gleichzeitig möchte man nicht auf Optimierungsmöglichkeiten verzichten, die jedoch die Korrektheit beeinträchtigen können. Durch das Konzept des *Slicings* wird eine Möglichkeit geschaffen, Optimierungen anzuwenden, die die Semantik des Programms nicht ändern[2].
Slicing approximiert die Menge aller Programmpunkte, die einen bestimmten Programmpunkt beeinflussen können. Für die Auswertung dieses Programmpunkts sollte es keinen Unterschied machen, wenn in einem Programmlauf nur die durch das Slicing erhaltenen Punkte berücksichtigt werden. Diese These wurde in der Dissertation von Wasserrab[10] formal gezeigt. Der nächste Schritt besteht darin, einen Algorithmus, der Slicing implementiert, zu verifizieren.
Ein bekannter Vertreter solcher Algorithmen ist der Slicer von Horwitz-Reps-Binkley[4] (HRB-Slicer). Dieser Algorithmus besteht dabei aus vier Schritten, die weitgehend unabhängig voneinander verifiziert werden können:

- Transformiere den eingegebenen Programmtext in einen Kontrollflussgraphen.
- Transformiere den Kontrollflussgraphen in einen System Dependence Graph (SDG).
- Füge *Summary Kanten* zu diesem SDG hinzu.
- Berechne den finalen Slice.

Im Rahmen des "Quis custodiet" Projekts am Karlsruher Institut für Technologie wird darauf abgezielt, eine verifizierte Fassung des HRB-Slicers für einige gängige Sprachen zu entwickeln. Neben dem wohl wichtigsten Teil - der Dissertation von Daniel Wasserrab - sind schon einige Fortschritte erreicht worden, beispielsweise konnte Simon Kohlmeyer in seiner Bachelorarbeit einen verifizierten Konverter schreiben, der Programme der Sprache WHILE in Kontrollflussgraphen umformt[6].

1.2 Zielsetzung dieser Arbeit

Diese Arbeit zielt darauf ab, einen Algorithmus zum Finden von Summary Kanten zu implementieren und zu verifizieren, der sich in gängigen funktionalen Sprachen ausführen lässt. Der Algorithmus selbst ist nicht neu, er wurde bereits von Reps, Horwitz, Sagiv und Rosay[8] entwickelt. Die Eingabe ist ein SDG ohne Summary Kanten, die Ausgabe ist eine Liste mit allen Summary Kanten des SDGs. Außerdem soll diese Arbeit vermitteln, warum der verwendete Algorithmus korrekt funktioniert. Dazu werden die zentralen Lemmas des finalen Beweises umrissen. Weiterhin werden einige Vorbereitungen für eine systematische Evaluation getroffen.
Nicht im Fokus dieser Arbeit steht eine effiziente Fassung des Algorithmus.

Kapitel 2

Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe, die zum Verständnis von Slicingalgorithmen, SDGs und Summary Kanten notwendig sind, erläutert. Auf diesen Grundlagen aufbauend wird zum Abschluss schließlich beschrieben, wie der in dieser Arbeit erstellte Algorithmus im Theorembeweiser Isabelle implementiert wurde.

2.1 Slicing und Summary Kanten

Ein *Slice* bezüglich einem Programmpunkt p besteht aus allen Befehlen, die bei einer möglichen Ausführung des Programms bis p die Auswertung von p beeinflussen könnten, ein *Slicingalgorithmus* bezeichnet einen Algorithmus zum Berechnen solcher Slices. Dieses Konzept findet in vielen Bereichen Anwendung, z.B. im Debugging, dem automatischen Parallelisieren von Prozessen oder der Programmoptimierung[2]. Je nach zugrundeliegendem Programm muss ein Slicingalgorithmus verschiedenen Anforderungen genügen. Insbesondere wird zwischen *intra-* und *inter*prozeduralem Slicing unterschieden; während im ersten Fall das Slicing nur in einer einzigen Main-Prozedur ausgeführt wird, kann ein interprozeduraler Slice auch Programmpunkte mehrerer Prozeduren enthalten. Ein bekannter Slicingalgorithmus ist der *HRB-Slicer* von Horwitz, Reps und Binkley. Um den HRB-Slicer anzuwenden, muss das übergebene Programm zuerst in einen System Dependence Graph transformiert werden. Dieser beinhaltet Summary Kanten, die mit einem speziellen Algorithmus berechnet werden. Eine verifizierte, ausführbare Fassung dieses Algorithmus ist das Ergebnis dieser Arbeit.

2.2 Program Dependence Graphs

Ein Program Dependence Graph (PDG) stellt die Abhängigkeiten verschiedener Programmpunkte innerhalb einer Prozedur dar. Zur Erstellung werden in der Regel Kontrollflussgraphen benutzt, eine nähere Beschreibung findet

sich in [2]. Ein PDG zu einer Prozedur m enthält für jeden Befehl in m einen Knoten. Außerdem enthält m noch einen Eingangs- und Ausgangsknoten.

Die Kanten des PDG stellen Abhängigkeiten zwischen diesen Knoten dar. Es wird zwischen Daten- und Kontrollabhängigkeiten unterschieden. Eine *Datenabhängigkeitskante* (DD-Kante) von x zu y drückt aus, dass das Ergebnis von y vom Ergebnis von x abhängt, eine *Kontrollabhängigkeitskante* (CD-Kante) von x zu y liegt vor, wenn von x abhängt, ob y ausgeführt wird oder nicht. Außerdem wird der Eingangsknoten über eine CD-Kante mit dem Ausgangsknoten verbunden. CD- und DD-Kanten werden als *intraprozedurale Kanten* bezeichnet. Ein Slice zu einem Knoten v muss alle Knoten beinhalten, die transitiv über CD- und DD-Kanten erreichbar sind.

2.3 System Dependence Graph

Ein System Dependence Graph (SDG) ergänzt PDGs um *interprozedurale* Kanten und stellt die Auswirkungen von Prozeduraufrufen dar. Abbildung 2.1 auf Seite 11 stellt einen Beispielhaften SDG für den neben der Abbildung angegebenen Quellcode dar. Dieser Abschnitt gibt vor allem einen Überblick über die Struktur von SDGs, die Erstellung sowie die semantische Bedeutung werden nicht behandelt. Es gibt verschiedene Definitionen von SDGs, die hier verwendete richtet sich nach [10]. Eine leicht andere Definition findet sich in [8], dort wird zum Beispiel nicht zwischen Call und Return Knoten unterschieden.

Ein SDG eines Programms P enthält für jede Prozedur m in P den zu m gehörenden PDG als Teilgraphen. In jedem Teilgraphen wird jeder Prozeduraufruf einer Prozedur m zu einer *Call Site* erweitert: dazu wird der Aufruf in einen *Call* und einen *Return Knoten* unterteilt, die mit einer *CR-Summary Kante* verbunden werden. Für jeden Eingangsparameter wird ein *Acutal In* Knoten an den Call Knoten über eine CD-Kante angehängt. Analog wird für jeden Ausgabeparameter von m ein *Formal Out* Knoten über eine CD-Kante an den Return-Knoten angehängt. Abbildung 2.2 stellt eine mögliche Call Site mit allen daran beteiligten Knoten und Kanten dar.

Nach dem gleichen Schema wird für jeden Eingabeparameter einer Prozedur m ein *Formal In* Knoten an den Eingang und für jeden Ausgabeparameter ein *Formal Out* Knoten an den Ausgang über eine CD-Kante angehängt. Actual Out, Actual In, Formal Out und Formal In Knoten werden mit *Parameterknoten* bezeichnet. Sie stellen Anschlussstellen dar, durch die Parameter übergeben und Rückgabewerte erhalten werden können.

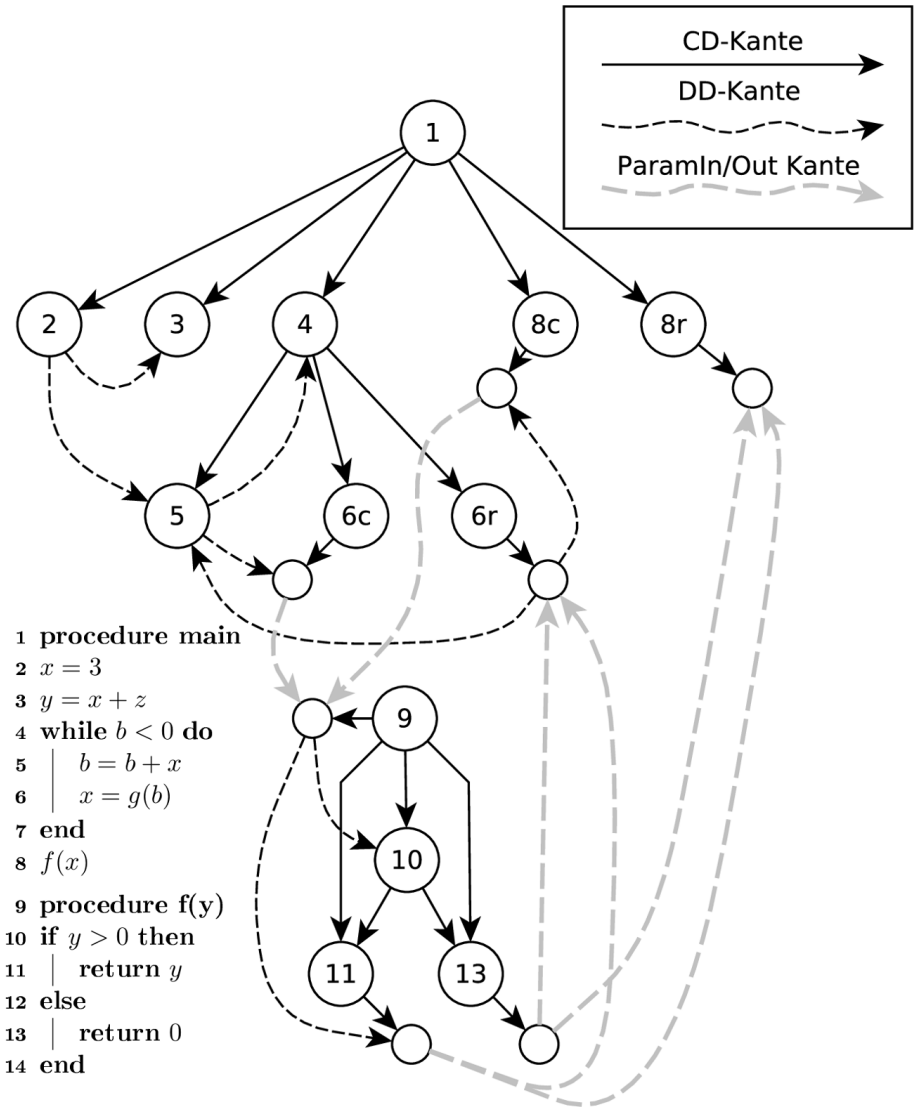

Abbildung 2.1: Beispiel für einen SDG mit dem dazugehörigen Quellcode. Call- und Return-Kanten sind nicht eingezeichnet, Parameterknoten werden als kleine Kreise dargestellt.

Die Call Sites werden durch vier verschiedene Kantentypen verbunden, die die interprozeduralen Abhängigkeiten zwischen Caller und Callee ausdrücken. Sei C eine beliebige Call Site, m bezeichne die durch C aufgerufene Prozedur.

- Für jeden zu C gehörenden Actual In Knoten zu einem Parameter x wird eine *Param In* Kante zu dem zu x gehörenden Formal In Knoten beim Eingangsknoten von m gezogen.
- Für jeden zu C gehörenden Actual Out Knoten zu einem Parameter

x wird eine *Param Out* Kante von dem zu x gehörenden Formal Out Knoten beim Ausgangsknoten von m gezogen.

- Für jeden Call Knoten wird eine *Call-Kante* zum Eingangsknoten von m gezogen.
- Für jeden Return Knoten wird eine *Return-Kante* vom Ausgangsknoten von m gezogen.

In Abbildung 2.1 sind 6c und 8c zwei Call Knoten sowie 6r und 8r zwei Return Knoten. Die bisher noch fehlenden *Parameter-Summary Kanten* (P-Summary Kanten) werden in Abschnitt 2.5 eingeführt. Der in der Literatur gängige Begriff "Summary Kante" bezieht sich auf P-Summary Kanten, CR-Summary Kanten werden für den formalen Beweis benötigt.

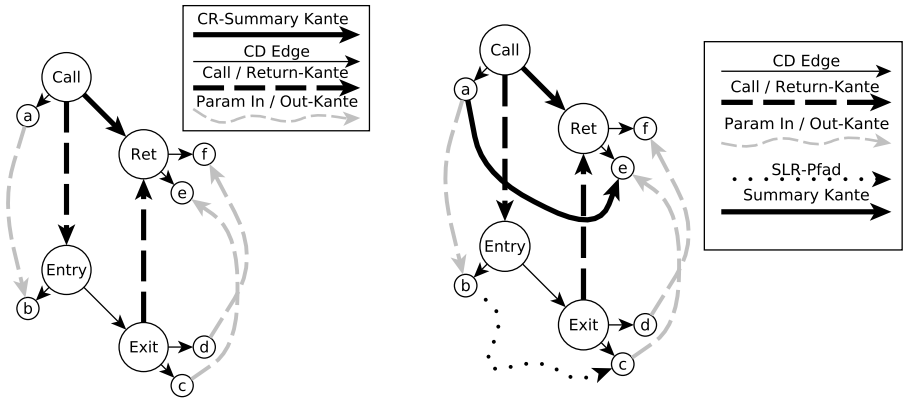

Abbildung 2.2: Beispiel für eine Call Site. In der rechten Abbildung wurde eine Summary Kante eingezeichnet. a, b ... f sind Parameterknoten

2.4 Realisierbare Pfade in SDGs

Für die Definition von P-Summary Kanten sind *realisierbare Pfade* wichtig. Diese werden in [8] formal korrekt definiert, hier soll nur die Idee dahinter vermittelt werden. Ein Pfad in einem SDG kann, solange nicht explizit anders angegeben, beliebige Kantentypen enthalten. Er ist genau dann *realizable*, wenn beim Durchlaufen jedesmal, wenn eine Prozedur verlassen wird, zu der Call Site zurückgekehrt wird, von der die Prozedur aufgerufen wurde. Diese Definition berücksichtigt, dass in einem echten Programmablauf nur Pfade genommen werden können, die realizable sind. Ein Pfad $p = (x_1, x_n)$ über die Knoten $(x_1 \dots x_n)$ ist genau dann *same level realizable* (SLR-Pfad), wenn

- er realizable ist, und

- für jeden Präfix $p' = x_1 \dots x_m$ mit $m < n$ beim Durchlaufen mindestens so viele Call- oder Param In-Kanten wie Return- oder Param Out-Kanten genommen wurden, und
- nach dem Durchlaufen von p genau so viele Call- oder Param In-Kanten wie Return- oder Param Out-Kanten genommen wurden.

Ein SLR-Pfad beginnt und endet immer in der selben Prozedur, außerdem sind einelementige Pfade (v, v) immer same level realizable. Wenn zwei SLR-Pfade aneinander gehängt werden, ist der resultierende Pfad wieder ein SLR-Pfad. Abbildung 2.3 zeigt Beispiele zu SLR-Pfaden.

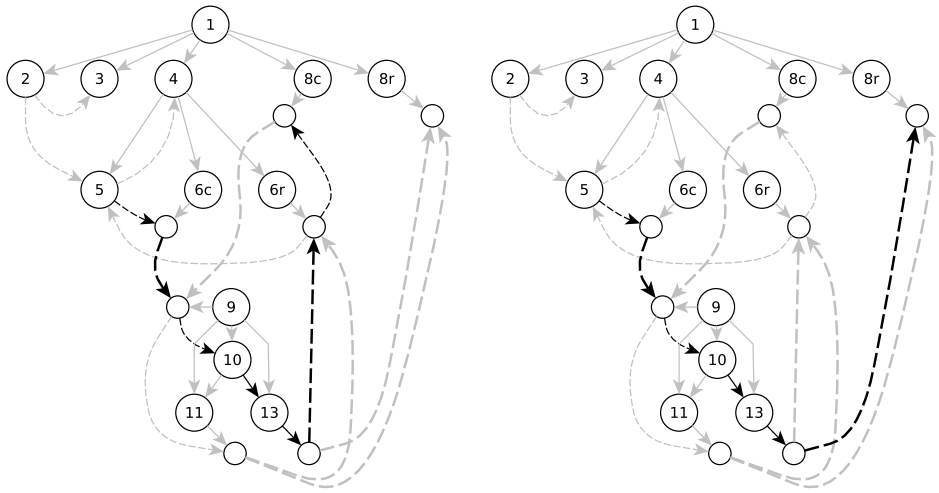

Abbildung 2.3: **links:** Beispiel für einen gültigen SLR-Pfad. **rechts: kein** gültiger SLR-Pfad, da zwar zur richtigen Prozedur, nicht aber zur korrekten Call Site zurückgekehrt wird.

2.5 Effiziente P-Summary Kanten Berechnung

Parameter-Summary Kanten (P-Summary Kanten) sind der Schlüssel zur effizienten Berechnung von Slices. Sie fassen die Auswirkungen eines Prozeduraufrufs zusammen, indem sie Abhängigkeiten zwischen Actual In und Actual Out Knoten erstellen.
Eine P-Summary Kante zwischen einem Actual In Knoten n und einem Actual Out Knoten n' wird genau dann gezogen, wenn ein SLR-Pfad zwischen den korrespondierenden Formal In und Formal Out Knoten existiert. Insbesondere existiert für jede P-Summary Kante (n, n') ein SLR-Pfad von n nach n'. In Abbildung 2.2 wird skizziert, in welchen Fällen eine P-Summary Kante erstellt wird. Wenn nicht weiter spezifiziert, bezieht sich der Begriff *Summary Kante* sowohl auf P- als auch auf CR-Summary Kanten.

```
Data : A SDG G without Summary Edges
Result : A set containing all Summary Edges of G
declare pathEdge, workList, summaryEdge: set of Path
procedure propagate(e: path)
if e ∉ pathEdge then
    insert e into pathEdge
    insert e into workList
end
procedure appendDefaultEdges(v, w: node)
for each x with (x, v) ∈ (cdEdges(G) ∪ ddEdges(G) ∪
summaryEdge) do
    propagate((x, w))
end
procedure findSummaryEdges(v: Formal In, w: Formal Out)
for each Actual Out y with w ⟶_pOut y do
    x = correspondingActualIn(y)
    insert (x, y) into summaryEdge
    for each a with (y, a) ∈ pathEdge do
        propagate((x, a))
    end
end
procedure main
pathEdge = ∅, workList = ∅, summaryEdge = getCREdges(G)
for w ∈ formalOuts(G) do
    insert (w, w) in pathEdge
    insert (w, w) in workList
end
while workList ≠ ∅ do
    select and remove edge (v, w) from workList
    if v ∈ formalIns(G) then
        findSummaryEdges(v, w)
    end
    appendDefaultEdges(v, w)
end
return summaryEdge
```

Algorithmus 1: Pseudocode des in dieser Arbeit verwendeten Algorithmus zum Finden von P-Summary Kanten. Die Notation $w \longrightarrow_{pOut} y$ gibt an, dass eine Param Out-Kante von w nach y existiert

CR-Summary Kanten müssen nicht extra berechnet werden, da sie aus der Konstruktion des SDGs automatisch hervorgehen: Jedes Mal, wenn ein Prozeduraufruf in Call und Return Knoten unterteilt wird, muss lediglich ei-

ne CR-Summary Kante dazwischen hinzugefügt werden. Daher werden CR-Summary Kanten als bereits gegeben vorausgesetzt. Sie werden im Vollständigkeitsbeweis wichtig, siehe Abschnitt 3.3.3.

Der in dieser Arbeit verifizierte Algorithmus findet die noch fehlenden P-Summary Kanten. Er richtet sich nach dem Algorithmus von Reps, Horwitz, Sagiv und Rosay, der in ihrem Paper "Speeding up Slicing"[8] beschrieben wird. Um die Korrektheit leichter zeigen zu können, wurde der Algorithmus leicht modifiziert. Die im ursprünglichen Algorithmus verwendeten 3 Unterscheidungen bezüglich des Knotentyps wurden auf eine Fallunterscheidung reduziert. Außerdem wird in im Paper nicht zwischen CR- und P-Summary Kanten unterschieden. Der Pseudocode des angepassten Algorithmus ist auf Seite 14 gegeben.

Die Idee hinter der Berechnung ist, dass *alle* SLR-Pfade, die in einem Formal Out Knoten enden, gefunden werden. Wird dabei ein SLR-Pfad gefunden, der zusätzlich in einem Formal In Knoten beginnt, so können P-Summary Kanten zwischen den korrespondierenden Actual In und Actual Out Knoten gezogen werden. Im Folgenden bezeichnet *Funktion* immer eine Funktion des Algorithmus, *Prozedur* bezieht sich immer auf Prozeduren eines Programms, das als SDG vorliegt.

Für die Berechnung der P-Summary Kanten eines SDGs werden drei Listen geführt:

pathEdge enthält alle bisher gefundenen SLR-Pfade, die in einem Formal Out Knoten enden.

workList enthält alle Pfade aus *pathEdge*, die noch bearbeitet werden müssen.

summaryEdge enthält alle bisher gefundenen Summary Kanten.

Die Listen speichern nur den Anfangs- und Endknoten der Pfade, die dazwischenliegenden Knoten sind nicht relevant.

In jedem Schritt wird ein Pfad (v, w) *selektiert* und aus *workList* entfernt. Falls es sich bei v um einen Formal In Knoten handelt liegt eine Call Site vor, sodass P-Summary Kanten gefunden werden können. Dazu wird findSummaryEdges aufgerufen, findSummaryEdges rekonstruiert die vorliegende Call Site und fügt entsprechend P-Summary Kanten in *summaryEdge* ein. In jedem Fall werden alle nach v führenden Summary- und intraprozeduralen Kanten vor den Pfad (v, w) gehängt (Funktion appendDefaultEdges). Sobald keine Pfade mehr selektiert werden können, wird *summaryEdge* zurückgegeben und gestoppt.

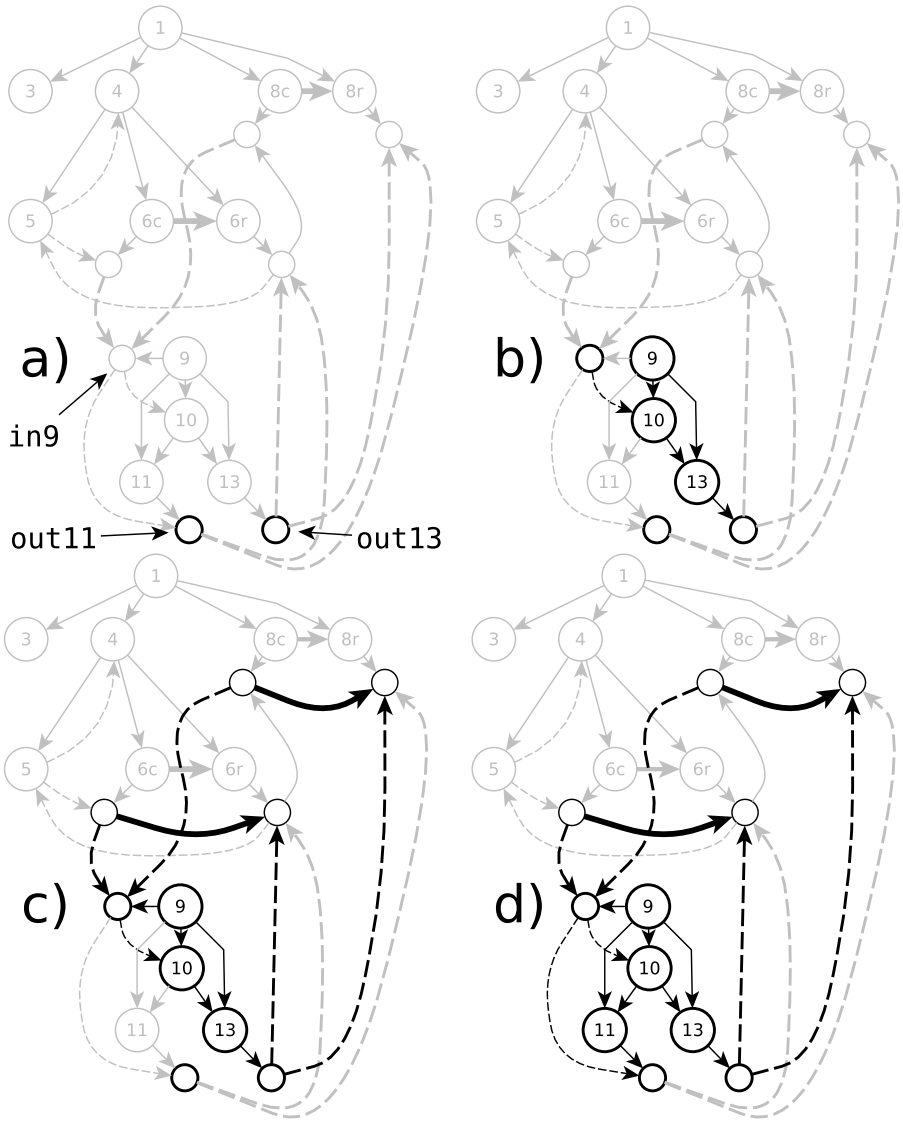

Abbildung 2.4: Beispielhafter Ablauf des Algorithmus. Knoten, von denen ein SLR-Pfad zu einem Formal Out Knoten gefunden wurde, werden fett hervorgehoben. Kanten, die vom Algorithmus benutzt wurden, werden schwarz dargestellt.

Abbildung 2.4 verdeutlicht die Funktionsweise des Algorithmus. Im Startzustand a) sind nur die Pfade (out13, out13) und (out11, out11) in *pathEdge* und *workList*. Nach 3 Iterationen der while-Schleife wird der in b) gezeigte Zustand erreicht, bisher wurde nur appendDefaultEdges aufgerufen. Im Beispiel wird anschließend (in9, out13) selektiert. Da in9 ein Formal In Knoten ist, werden mit einem Aufruf von findSummaryEdges die zwei eingezeichneten P-Summary Kanten hinzugefügt und Zustand c) erreicht. Zuletzt werden

die Pfade (out11, out11), (9, out11), (11, out11), (10, out11), (in9, out11) und (9, out13) selektiert, dabei wird jedoch keine weitere P-Summary Kante gefunden. Anschließend ist *workList* leer und der Algorithmus terminiert im finalen Zustand d).

2.6 Isabelle

Isabelle ist ein interaktiver Theorembeweiser, der auf verschiedenen Logiken aufbaut. Die am weitesten verbreitete Logik ist die *Higher Order Logic* (HOL), die auch in dieser Arbeit verwendet wird. Die Beweise in dieser Arbeit bauen auf der Notation und den Basistypen von HOL auf, die in diesem Abschnitt erläutert werden. Eine weitergehende Einführung findet sich in [7]. HOL enthält vordefinierte Typen für Wahrheitsvariablen, natürliche Zahlen, Listen und Mengen, die mit **bool**, **nat**, $'$**a list** und $'$**a set** bezeichnet werden. $'$**a** ist der Elementtyp der Liste bzw. der Menge. Die Schreibweise $a :: {}'\mathbf{b}$ bedeutet, dass a den Typ **b** hat. Der Operator $\times$ erzeugt Paare, auf die Elemente von Paaren wird über die beiden Projektionen $fst :: {}'\mathbf{a} \times {}'\mathbf{b} \Rightarrow {}'\mathbf{a}$ und $snd :: {}'\mathbf{a} \times {}'\mathbf{b} \Rightarrow {}'\mathbf{b}$ zugegriffen. Für Listen ist die leere Liste [ ] definiert, der Operator # hängt ein neues Element an die Liste an. Zwei Listen l_1, l_2 werden mit $l_1@l_2$ aneinander gehängt. $set\ l$ konvertiert eine Liste l in eine Menge. Für natürliche Zahlen ist die 0 definiert, den Nachfolger einer natürlichen Zahl n liefert $Suc\ n$. Isabelle erlaubt das Definieren von Typ Kurzschreibweisen über das Schlüsselwort **type_synonym**, beispielsweise definiert

$$\textbf{type_synonym Path} = \textbf{Node} \times \textbf{Node}$$

eine Kurzschreibweise für Pfade. Neue Typen können über das Schlüsselwort **datatype** definiert werden, der Typ

$$\textbf{datatype } {}'\textbf{a List} = \text{Nil} \mid \text{Cons } {}'\textbf{a } ({}'\textbf{a List})$$

gibt beispielsweise den Datentyp für Listen mit Elementtyp $'$**a** an.

Lässt sich aus A_1 und A_2 der Ausdruck B ableiten, so schreiben wir

$$\frac{A_1 \qquad A_2}{B}$$

Alternativ wird die Schreibweise $[\![A_1; A_2]\!] \Rightarrow B$ verwendet. Gilt eine Aussage A für beliebige v, so schreiben wir $\bigwedge v.\ A\ v$ (Vergleichbar mit $(\forall)$).

Die Beweise dieser Arbeit benutzen *locales*. locales sind ein Konzept in Isabelle, die das Modularisieren von Beweisen vereinfachen. Sie erlauben das Hinzufügen (fixes) lokaler Definitionen, anschließend werden lokale Annahmen für diese Definitionen getroffen (assumes), die in den Beweisen benutzt werden können. Folgendes locale stellt z.B. eine Halbgruppe dar:

locale semi = **fixes** $\circledcirc$ :: $'\mathbf{a} \Rightarrow '\mathbf{a} \Rightarrow '\mathbf{a}$
assumes $(x \circledcirc y) \circledcirc z = x \circledcirc (y \circledcirc z)$

Außerdem können die Annahmen und Definitionen bestehender locales übernommen und erweitert werden. Das folgende locale erweitert die Annahmen des locales für Halbgruppen um ein neutrales Element:

locale monoid = semi +
fixes e :: $'\mathbf{a}$
asssumes $\bigwedge n.\ e \circledcirc n = n \wedge n \circledcirc e = n$

Alle Aussagen, die im Rahmen dieser Arbeit formal verifiziert wurden, werden mit **lemma:** angekündigt. Ein Verweis auf das Lemma im Quellcode kann Anhang 5.1 entnommen werden. Teilweise wurden die gezeigten Lemmas modifiziert, um Implementierungsdetails, die das Verständnis des Beweises unnötig stören würden, auszublenden.

2.7 Verwendete Frameworks

Der Algorithmus benutzt das *Graph-Framework* von Kohlmeyer[6]. Dieses definiert eine Schnittstelle mit Operationen zum Traversieren eines Graphen. Diese werden in Abschnitt 3.1 erläutert. Außerdem beinhaltet das Framework eine ausführbare und verifizierte Implementierung, die die Funktionen der Graph-Schnittstelle realisiert. Das Graph-Framework erlaubt das Definieren eigener Knoten- und Kantentypen. Als Knotentyp wurde der Typ des SDG-Frameworks (Typ **SDG_node**) übernommen, für Kanten wird dort jedoch kein Datentyp erstellt. Deshalb wurde ein neuer Kantentyp definiert:

datatype SDGEdgeData =
DDEdge **var**
| CDEdge
| CallEdge **pname**
| ReturnEdge **pname**
| InEdge **pname var**
| OutEdge **pname var**

var und **pname** geben die Typen für Variablen und Prozedurbezeichner an. Die Parameter der einzelnen Kantentypen werden in Tabelle 2.1 erläutert, sie richten sich nach den Parametern der entsprechenden Prädikate.

Alle Beweise dieser Arbeit bauen substanziell auf dem SDG-Framework

von Daniel Wasserrab[10] auf. Das SDG-Framework führt formale Regeln für die bereits erläuterten Begriffe ein, beispielsweise werden formale Regeln für Knoten und Kanten in SDGs definiert. Kanten in SDGs werden über Prädikate modelliert. Tabelle 2.1 gibt einen Überblick über einige der definierten Prädikate und ihre Notation.

Tabelle 2.1: Prädikate des SDG-Frameworks

Prädikat	**Beschreibung**
$n \longrightarrow_{\text{cd}} n'$	Gilt, wenn eine CD-Kante zwischen n und n' liegt.
n -$V$$\rightarrow_{\text{dd}}$ n'	Gilt, wenn eine DD-Kante zwischen n und n' liegt. V ist eine Variable, die in n gesetzt und in n' gelesen wird.
n -$p$$\rightarrow_{\text{call}}$ n'	Gilt, falls eine Call-Kante zwischen n und n' liegt. p gibt die Prozedur von n' an.
n -$p$$\rightarrow_{\text{ret}}$ n'	Gilt, falls eine Return-Kante zwischen n und n' liegt. p gibt die Prozedur von n an.
n -p:$V$$\rightarrow_{\text{in}}$ n'	Gilt, falls eine Param In-Kante zwischen n und n' für den Parameter V besteht. p gibt die Prozedur von n' an.
n -p:$V$$\rightarrow_{\text{out}}$ n'	Gilt, falls eine Param Out-Kante zwischen n und n' für den Parameter V besteht. p gibt die Prozedur von n an.
n i-$ns$$\rightarrow_{\text{d}*}$ n'	Gilt, falls es einen intraprozeduralen Pfad im SDG von n nach n' über die in der Liste ns angegebenen Knoten gibt.
call_of_return_node r c	Gilt, falls r ein Return Knoten und c der dazugehörige (eindeutige) Call Knoten ist.
matched n ns n'	Gilt, falls ein SLR-Pfad zwischen n und n' über die in der Liste ns gegebenen Knoten existiert.
n s-$p$$\rightarrow_{\text{sum}}$ n'	Gilt, falls eine Summary Kante zwischen n und n' liegt. p ist der Bezeichner der Prozedur, die durch die Kante "zusammengefasst" wird.

2.8 Formalisierung des Algorithmus

Der auf Seite 14 dargestellte Algorithmus ist zum besseren Verständnis in einer imperativen Variante gegeben, Isabelle benutzt jedoch eine funktionale Sprache zum Definieren von Funktionen. Die für diese Arbeit entworfene

funktionale Fassung wird in diesem Abschnitt erläutert, ein Großteil des resultierenden Quellcodes wird durch Algorithmus 2 auf Seite 22 gezeigt.
Der Algorithmus benutzt die aus dem Graph-Framework gebotene Schnittstelle, um den zugrunde liegenden SDG zu traversieren. Damit der Algorithmus terminiert, muss jedoch sichergestellt werden, dass der Graph endlich ist. Dazu werden die lokalen Definitionen des Graph-Frameworks übernommen und um einige Annahmen erweitert:

locale SummaryEdgeComputation = graph_inEdges + graph_outEdges
for αe :: **Graph** $\Rightarrow$ **SDGEdgeData set**
for inEdges :: **Graph** $\Rightarrow$ **SDG_node** $\Rightarrow$ **SDGEdgeData list**
for outEdges :: **Graph** $\Rightarrow$ **SDG_node** $\Rightarrow$ **SDGEdgeData list**

fixes getCorCallNode :: **SDG_node** $\Rightarrow$ **SDG_node**
fixes getProc :: **SDG_node** $\Rightarrow$ **pname**
fixes S :: **Graph** $\Rightarrow$ **SDGEdgeData set**

finite $finite\ (S\ g)$ **assumes** $\bigwedge g\ n.\ \alpha e\ g \subseteq S\ g$
assumes $\bigwedge g\ n.\ set\ (inEdges\ g\ n) \subseteq\ S\ g$
assumes $\bigwedge g\ n.\ set\ (outEdges\ g\ n) \subseteq S\ g$

Die Schreibweise "**for** <*Funktion*>" wird benutzt, um eine Funktion des Graph-Frameworks in den lokalen Kontext zu übernehmen und passend zu typisieren. $\alpha e\ g$ gibt die Menge an Kanten in g an. inEdges und outEdges geben eine Liste mit eingehenden bzw. ausgehenden Kanten zurück.
Neu gefordert wird die Funktion $getCorCallNode\ n$, die den zu einem Return Knoten n gehörenden Call Knoten zurück gibt. getProc liefert den Bezeichner der Prozedur, in der ein übergebener Knoten liegt. $S\ g$ gibt eine endliche Obermenge an Kanten vor, die in einem Graphen g liegen dürfen ("S" steht für "**S**uperset"). Dadurch wird die Termination des Algorithmus sichergestellt (siehe Abschnitt 3.1). Insbesondere dürfen dazu auch inEdges und outEdges nur Kanten aus dieser Menge liefern.
Da funktionale Sprachen im Gegensatz zu imperativen Sprachen nicht über Statusvariablen verfügen, müssen *pathEdge*, *workList* und *summaryEdge* immer mit übergeben werden. Dazu wurden neue Typen eingeführt, die den aktuellen Status des Algorithmus widerspiegeln:

type_synonyme Path :: **Node** $\times$ **Node**
type_synonyme SummaryPath :: **Node** $\times$ **pname** $\times$ **Node**
type_synonyme Status :: **Path List** $\times$ **Path List**
$\times$ **SummaryPath List**

Der im Typ **SummaryPath** mitgeführte Prozedurbezeichner wird benötigt,

um die durch Summary Kanten zusammengefasste Prozedur zu identifizieren. getPathEdge, getPathSet, getWorkList, getWorkSet, getSummaryEdge und getSummarySet geben den entsprechenden Eintrag eines Status zurück, entweder als Liste oder als Menge. Die Funktion getSummaryPaths liefert das gleiche Ergebnis wie getSummaryEdge, nur wird hier der Prozedurbezeichner ausgelassen und Pfade des Typs **Path** zurückgegeben.

Tabelle 2.2: Rückgabewerte einiger Funktionen der funktionalen Fassung des Pseudocode-Algorithmus auf Seite 14.

Name	**Rückgabewert**
updateState	Durchläuft die **while**-Schleife einmal und gibt den neuen Status zurück.
formalInFilter v	Gibt genau dann **True** zurück, wenn v ein Formal In Knoten ist.
whileLoop	Wiederholt die **while**-Schleife so lange, bis $workList$ leer ist.
prependPaths (v, w)	Gibt eine Liste an Pfaden zurück, für jedes $(u, v) \in$ *set* l ist der Pfad (u, w) in der zurückgegebenen Liste enthalten.
appendPaths (v, w) l	Gibt eine Liste an Pfaden zurück, für jedes $(w, x) \in$ *set* l ist der Pfad (w, x) in der zurückgegebenen Liste enthalten.
appendDefaultEdges	Hängt intraprozedurale Kanten und Summary Kanten vor einen übergebenen Pfad.
findSummaryEdges g s (v, w)	Findet P-Summary Kanten, die sich aus einem SLR-Pfad zwischen Formal In Knoten v und Formal Out Knoten w ableiten lassen.
getCallingActualOuts	Gibt alle Actual Out Knoten an, die über Param Out-Kanten mit einen eingegebenen Formal Out Knoten verbunden sind.
for12	Ermittelt zu übergebenen Formal In, Formal Out und Actual Out Knoten den fehlenden Actual In Knoten und fügt eine Summary Kante hinzu.
getCorActualIn	Gibt zu einem eingegebenen Actual Out Knoten den korrespondierenden Actual In Knoten an.
getCDEdges getDDEdges	Liefern CD- und DD-Kanten des Graphen als Liste.

Alle **for**-Schleifen des Algorithmus iterieren durch Elemente aus endlichen Mengen (diese werden als Listen dargestellt) und können somit durch foldl ersetzt werden. Tabelle 2.2 erläutert die verwendeten Funktionen. Eine wichtige Eigenschaft des Algorithmus ist, dass *pathEdge* nur durch propagate, *workList* nur durch propagate und das Entfernen des ersten Elements in updateState geändert wird. Die Beweise in Kapitel 3 bauen darauf auf.

```
1  fun propagate :: State ⇒ Path ⇒ State where
2  propagate s e = if e ∈ (getPathSet s) then s
3    else (e#getPathSet s, e#getWorkList s, getSummaryEdge s)
4  fun appendDefaultEdges :: Graph ⇒ State ⇒ Path ⇒ State
   where
5  appendDefaultEdges g s (v, w) =
6      let l = getSummaryPaths s @ getCDEdges g @ getDDEdges g
7      in foldl propagate s (prependPaths (v, w) l)
8  fun for12 :: Graph ⇒ Node ⇒ Node ⇒ State ⇒ Node ⇒ State
   where
9  for12 g v w s y = (let x = getCorActualIn g v y)
10     s' = (getPathEdge s, getWorkList s,
11     (x, getProc v, y)#getSummaryEdge s)
12   in foldl propagate s' (appendPaths (x, y) (getPathEdge s))
13 fun findSummaryEdges :: Graph ⇒ State ⇒ Path ⇒ State where
14 findSummaryEdges g s (v, w) = foldl (for12 g v w) s
   (getCallingActualOuts g w)
15 fun updateState :: Graph ⇒ State ⇒ State   where
16 updateState g (pathEdge, ((v, w)#xs), summaryEdge) =
17   let s' = (pathEdge, xs, summaryEdge) in
18     let s29 = if (formalInFilter v) then
19       findSummaryEdges g s' (v, w)
20     else
21       s'
22     in appendDefaultEdges g s29 (v, w)
23 fun whileLoop :: Graph ⇒ State ⇒ State    where
24 whileLoop g s =
25   if getWorkList s ≠ [ ] then
26     whileLoop g (updateState g s)
27   else s
```

Algorithmus 2: Realisierung des Algorithmus in Isabelle

Kapitel 3

Beweise zum Algorithmus

Im Rahmen dieser Arbeit wurden 3 Beweise geführt:

1. Der Terminationsbeweis, d.h. es wurde gezeigt, dass der Algorithmus für beliebige Eingaben terminiert
2. Beweis der Korrektheit, d.h. alle gefunden Kanten sind auch wirklich Summary Kanten.
3. Beweis der Vollständigkeit, d.h. alle Summary Kanten werden auch gefunden.

Die Beweise werden in diesem Kapitel genauer erläutert. Sie beziehen sich nur auf die funktionale Fassung auf Seite 22.

3.1 Termination

In diesem Abschnitt wird gezeigt, dass die auf Seite 22 gegebene Funktion whileLoop für beliebige Eingaben terminiert.

3.1.1 Allgemeines

HOL ist eine Logik totaler Funktionen, andernfalls könnte es zu Inkonsistenzen kommen [1]. Um eine Funktionsdefinition abzuschließen, muss unter anderem gezeigt werden, dass sie für *jede* Eingabe (und nicht nur praktisch sinnvolle Eingaben) terminiert. Daher wurde auch in dieser Arbeit die Termination für beliebige Eingaben gezeigt.
Terminationsbeweise werden in Isabelle meistens automatisch geführt. Dazu wird bei einem rekursiven Aufruf ermittelt, ob die "Größe" eines der in der Rekursion wiederkehrenden Argumente bei jedem rekursiven Aufruf verringert wird. Da die Größe nicht negativ sein kann, muss in diesem Fall nach endlich vielen Aufrufen der Basisfall eintreten und die Rekursion stoppen. Als Beispiel wird hier die Summenfunktion betrachtet:

```
fun sum :: (nat ⇒ nat list ⇒ nat) where
  sum n [] = n
  | sum n (x#xs) = sum (n + x) xs
```

Die Summenfunktion terminiert, da in jedem Aufruf die Größe der übergebenen Liste echt kleiner wird. Da die Liste keine "negative Größe" haben darf, muss die Funktion nach endlich vielen Rekursionen terminieren. Der gleiche Mechanismus wird in dem in dieser Arbeit geführten Terminationsbeweis für whileLoop angewandt. Da der übergebene Graph unverändert bleibt, kann dieser für den Terminationsbeweis ignoriert werden, lediglich der zweite Statusparameter ist relevant. Die Schwierigkeit besteht in zwei Punkten:

1. Eine geeignete Größenfunktion f :: **Status** ⇒ **M** zu finden. Über **M** muss eine totale Halbordnung $\leq_M$ mit minimalen Element definiert sein (d.h. die Größe muss nicht eine natürliche Zahl sein).

2. Beweisen, dass diese Funktion in jedem Schritt (d.h. bei jedem rekursiven Aufruf von updateState kleiner wird. Formal muss

$$f\ (updateState\ g\ s) <_M f\ s \tag{3.1}$$

gelten.

Die Größenfunktion wird Schritt für Schritt in den nächsten Abschnitten aufgebaut. Sie besteht aus zwei verschiedenen Teilen: einem Teil, der die Veränderung von *pathEdge* darstellt, und einem anderen Teil, der die Veränderung von *workList* berücksichtigt. Keiner dieser Teile alleine ist geeignet als Größenfunktion, erst wenn sie lexikographisch kombiniert werden kann die Termination beweisen werden. Dieses Verfahren wird in Abschnitt 3.1.4 beschrieben.

3.1.2 Teil 1 - Eine Größe für *pathEdge*

Sei $SNodes\ g$ die Menge an Knoten, die am Anfang oder Ende einer Kante in $S\ g$ liegen. $S\ g$ ist die in den lokalen Annahmen geforderte Obermenge an Kanten eines Graphen g (siehe Abschnitt 2.8), $SNodes\ g$ definiert eine geeignete Obermenge an Knoten. Wir definieren

$$stateClosure\ s = getWorkSet\ s \cup getPathSet\ s \cup getSummarySet\ s$$

stateClosure enthält alle Pfade, die in einer der drei Variablen von s vorkommen. Damit definieren wir

$$\begin{aligned} &closure\ g\ s := SNodes\ g \cup \\ &\{x \mid \exists y.\ (x, y) \in stateClosure\ s \vee (y, x) \in stateClosure\ s\} \end{aligned}$$

In $closure\ g\ s$ sind alle Knoten, die entweder im Graph g oder in irgendeinem Pfad von s vorkommen. Mit dieser Definition wird berücksichtigt, dass der Startzustand auch ungültige, außerhalb von g liegende Pfade enthalten kann. Da wir die Termination für beliebige, auch nicht vorgesehene Eingaben zeigen wollen, müssen auch diese Eingaben mit berücksichtigt werden. Mit den in 2.8 gegebenen lokalen Annahmen folgt, dass closure immer endlich ist:

$$\textbf{lemma: } finite\ (closure\ g\ s) \tag{3.2}$$

Nach Definition von closure folgt direkt

$$\textbf{lemma: } getPathSet\ g\ s \subseteq closure\ g\ s \times closure\ g\ s \tag{3.3}$$

Wir definieren die Größenfunktion:

$$peCard_dec\ g\ s = |closure\ g\ s|^2 - |getPathSet\ s|$$

Nach Lemma 3.3 ist diese Funktion immer positiv.

Es lässt sich zeigen, dass durch einen Aufruf von updateState $|closure\ g\ s|$ monoton kleiner und $|getPathSet\ s|$ monoton größer wird:

$$\textbf{lemma: } closure\ g\ (updateState\ g\ s) \subseteq closure\ g\ s \tag{3.4}$$

$$\textbf{lemma: } |getPathSet\ s| \leq |getPathSet\ (updateState\ g\ s)| \tag{3.5}$$

Daraus folgt, dass peCard_dec monoton fallend ist (das "dec" steht für "decreasing"):

$$\textbf{lemma: } peCard_dec\ g\ (updateState\ g\ s) \leq peCard_dec\ g\ s \tag{3.6}$$

Damit wurde der erste Teil der Größenfunktion definiert. Die Funktion alleine kann jedoch nicht als Größenfunktion verwendet werden, da diese streng monoton fallend sein müssen (siehe Gleichung 3.1 auf Seite 24). Für den Fall, dass $peCard_dec$ gleich bleibt, wird eine zweite Größenfunktion benutzt.

3.1.3 Teil 2 - Eine Größe für *workList*

Wir definieren

$$wlSize\ s := |getWorkSet\ s|$$

wlSize ist bereits der zweite Teil der Größenfunktion.
Mit Lemma 3.4 und 3.5 gilt nach Definition von peCard_dec

$$\frac{peCard_dec\ g\ s = peCard_dec\ (updateState\ g\ s)}{|getPathSet\ s| = |getPathSet\ (updateState\ g\ s)|}$$

updateState hat in diesem Fall keine Kanten in *pathEdge* eingefügt. Da *pathEdge* nur *gleichzeitig* mit *workList* durch propagate geändert wird, wurde auch keine Kante in *workList* eingefügt. Es wird allerdings immer eine Kante aus *workList* entfernt, wlSize wird damit monoton kleiner. Es folgt:

$$\frac{\begin{array}{c} getWorkList\ s \neq [\,] \\ peCard_dec\ g\ s = peCard_dec\ (updateState\ g\ s) \end{array}}{wlSize\ s < wlSize\ (updateState\ g\ s)} \tag{3.7}$$

Nun können wir die finale Größenfunktion definieren.

3.1.4 Die finale Größenfunktion

In diesem Abschnitt wird mit den in Abschnitten 3.1.2 und 3.1.3 definierten Größenfunktionen über eine lexikographische Kombination die finale Größenfunktion zusammengesetzt.

Seien M und N zwei totale Halbordnungen, notiert durch $\leq_M$ und $\leq_N$. Das *lexikographische Produkt* $M *_{lex} N$, notiert als $\leq_{M\times N}$ ist definiert durch

$$(a, b) \leq_{M\times N} (c, d) :\Longleftrightarrow a \leq_M c \vee (a = c \wedge b \leq_N d)$$

Das lexikographische Produkt ist eine totale Halbordnung. Besitzen $\leq_M$ und $\leq_N$ die minimalen Elemente $\perp_M$ und $\perp_N$, dann ist $(\perp_M, \perp_N)$ das minimale Element von $M *_{lex} N$.
Wir stellen *peCard_dec* und *wlSize* als Relationen dar:

$$\begin{array}{l} (s_1, s_2) \in P :\Longleftrightarrow peCard_dec\ g\ s_1 \leq peCard_dec\ g\ s_2 \\ (s_1, s_2) \in W :\Longleftrightarrow wlSize\ s_1 \leq wlSize\ s_2 \end{array}$$

$P *_{lex} W$ besitzt nun $(0, 0)$ als minimales Element.

Setze

$$s' := updateState\ g\ s$$

für einen Status s mit $getWorkList\ s \neq [\,]$. Mit Lemma 3.6 auf Seite 25 und 3.7 auf Seite 26 gilt nun

$$\begin{array}{l} peCard_dec\ g\ s' < peCard_dec\ g\ s \vee \\ peCard_dec\ g\ s' = peCard_dec\ g\ s \overset{3.7}{\Longrightarrow} \\ peCard_dec\ g\ s' < peCard_dec\ g\ s \vee \\ (peCard_dec\ g\ s' = peCard_dec\ g\ s \wedge wlSize\ s' < wlSize\ s) \end{array}$$

Also gilt nach Definition von $\leq_{P\times W}$ und s'

$$\textbf{lemma: } updateState\ g\ s <_{P\times W} s \tag{3.8}$$

Mit diesem Lemma erfüllt Abbildung $f(x) := (peCard_dec\ g\ s, wlSize\ s)$ mit der Halbordnung $P *_{lex} W$ die in Gleichung 3.1 auf Seite 24 gestellten Bedingungen an eine gültige Größenfunktion. Damit kann Isabelle nun die Termination von whileLoop für beliebige Eingaben zeigen.

3.2 Korrektheit des Algorithmus

Dieser Abschnitt zeigt, dass alle Kanten, die durch den Algorithmus (bei Eingabe eines gültigen Startzustands) gefunden werden, auch wirklich Summary Kanten sind. Um den Beweis zu führen wird zuerst die Einbettung in das SDG-Framework erläutert und die Menge an gültigen Startzuständen definiert, anschließend wird über einige Invarianten der Beweis erbracht.

3.2.1 Lokale Annahmen des Beweises

Um den Algorithmus zu definieren, musste der Zugriff auf eine Graph-Schnittstelle vorausgesetzt werden (siehe Abschnitt 2.8). Es wurde - bis auf die Vorgabe der Knoten- und Kantentypen - keine strukturelle Voraussetzung an den Graphen gestellt. Der Beweis kann jedoch nur korrekt funktionieren, wenn der eingegebene Graph ein gültiger SDG ist. Für den Beweis der Korrektheit (und der Vollständigkeit) wurden die lokalen Annahmen erweitert, sodass ein Bezug des Graphen zum SDG-Framework (das korrekte SDGs bschreibt) besteht:

locale graph_SDG = SummaryEdgeComputation + SDG
fixes G :: **Graph**
assumes $call_of_return_node\ n\ n' \Longrightarrow getCorCallNode\ n = n'$
assumes $(n, DDEdge\ V, n') \in \alpha e\ G \Longleftrightarrow n\ \text{-}V\rightarrow_{\text{dd}}\ n'$
assumes $(n, CDEdge, n') \in \alpha e\ G \Longleftrightarrow n \longrightarrow_{\text{cd}} n'$
assumes $(n, CallEdge\ p, n') \in \alpha e\ G \Longleftrightarrow n\ \text{-}p\rightarrow_{\text{call}}\ n'$
assumes $(n, ReturnEdge\ p, n') \in \alpha e\ G \Longleftrightarrow n\ \text{-}p\rightarrow_{\text{ret}}\ n'$
assumes $(n, InEdge\ p\ V, n') \in \alpha e\ G \Longleftrightarrow n\ \text{-}p{:}V\rightarrow_{\text{in}}\ n'$
assumes $(n, OutEdge\ p\ V, n') \in \alpha e\ G \Longleftrightarrow n\ \text{-}p{:}V\rightarrow_{\text{out}}\ n'$

Die fixierte Variable G bezeichnet einen beliebigen Graphen, für den die Korrektheit gezeigt werden soll. Die hinzugefügten Annahmen stellen sicher, dass die Kanten in G die Prädikate des SDG-Frameworks (siehe 2.7) widerspiegeln.

3.2.2 Prädikat isStartState

Im Pseudocode des Algorithmus (Seite 14) werden in Zeile 21 - 24 zuerst alle Formal Out Knoten in *pathEdge* und *workList* eingefügt, außerdem

wird vorausgesetzt, dass *summaryEdge* alle CR-Summary Kanten des Graphen enthält. Der daraus resultierende Zustand ist der Erste, der von der **while**-Schleife verarbeitet wird. Deswegen muss dieser *Startzustand* bei der funktionalen Implementierung (Seite 22) in whileLoop als Argument übergeben werden. Die Korrektheit kann nur für gültige Startzustände gezeigt werden, die über das folgende Prädikat formal definiert werden:

$$\frac{\begin{array}{l} getPathSet\ s = \{(a,a) | a \in formalOuts\ G\} \\ getWorkSet\ s = \{(a,a) | a \in formalOuts\ G\} \\ getSummarySet\ s = crSummaryEdges\ G \end{array}}{isStartState\ s}$$

Es werden nur gültige Startzustände bezüglich der fixierten Variable G betrachtet.

3.2.3 Invariante sndInFormalOuts

Wir definieren ein neues Prädikat:

$$\frac{snd`\ A \subseteq formalOuts\ g}{sndInFormalOuts\ g\ A}$$

$$\frac{\begin{array}{l} sndInFormalOuts\ g\ (getPathSet\ s) \\ sndInFormalOuts\ g\ (getWorkSet\ s) \end{array}}{sndInFormalOuts_State\ g\ s}$$

snd` A bezeichnet das Bild von A unter snd. sndInFormalOuts_State gibt an, ob alle Pfade in *pathEdge* und *workList* in einem Formal Out Knoten enden. Es lässt sich zeigen, dass sndInFormalOuts unter Aufrufen von appendDefaultEdges und findSummaryEdges invariant ist. Daraus folgt die Invarianz unter updateState:

$$\textbf{lemma:}\ \frac{sndInFormalOuts_State\ g\ s}{sndInFormalOuts_State\ g\ (updateState\ g\ s)} \tag{3.9}$$

Die Invariante ist auch für Startzustände gültig, wie Lemma

$$\textbf{lemma:}\ \frac{isStartState\ s}{sndInFormalOuts_State\ G\ s} \tag{3.10}$$

zeigt. Der Beweis folgt direkt aus der Definition von isStartState.

3.2.4 Bedingungen zum Finden von P-Summary Kanten

In diesem Abschnitt wird beschrieben, unter welchen Umständen der Algorithmus P-Summary Kanten findet. Für die Existenz einer P-Summary Kante müssen alle folgenden Bedingungen erfüllt sein:

1. Es muss ein gültiger SDG vorliegen. Im Folgenden wird daher nur noch die lokal fixierte Variable G betrachtet (siehe Abschnitt 3.2.1).

2. Es gibt einen SLR-Pfad zwischen einem Formal In Knoten v und einem Formal Out Knoten w, formal muss

$$\exists xs.\ matched\ v\ xs\ w \wedge w \in formalOuts\ G \wedge v \in formalIns\ G \quad (3.11)$$

gelten.

3. Es gibt einen Actual In Knoten x und einen Actual Out Knoten y, x ist über eine Param In-Kante mit v verbunden, w ist über eine Param Out-Kante mit y verbunden. Formalisiert muss

$$x\ \text{-}p{:}V{\rightarrow}_{\text{in}}\ v \wedge w\ \text{-}p{:}V'{\rightarrow}_{\text{out}}\ y \wedge p = getProc\ v \quad (3.12)$$

für Variablen V, V' gelten.

4. x und y gehören zur selben Call Site.

Falls y der Actual Out und v der Formal In Knoten der Call Site sind, wird für

$$x := getCorActualIn\ G\ v\ y$$

sichergestellt, dass x und y zur selben Call Site gehören (Bedingung 4). Weiter wird dadurch sichergestellt, dass x -$getProc\ v{:}V{\rightarrow}_{\text{in}}\ v$ gilt (Bedingung 3).
Für

$$y \in set\ (getCallingActualOuts\ G\ w)$$

gilt, dass w über eine Param Out mit y verbunden ist (Bedingung 3). Bedingung 2 muss vorausgesetzt werden, es folgt:

$$\textbf{lemma:}\ \frac{\begin{array}{ll}\exists xs.\ matched\ v\ xs\ w & x = getCorActualIn\ G\ v\ y \\ v \in formalIns\ g & w \in formalOuts\ G \\ y \in set\ (getCallingActualOuts\ G\ w) & \end{array}}{x\ \text{s-getProc}\ v{\rightarrow}_{\text{sum}}\ y} \quad (3.13)$$

3.2.5 Die finale Invariante

Dieser Abschnitt definiert eine letzte Invariante, aus der direkt hervorgeht, dass alle gefundenen Kanten Summary Kanten sind. Wir definieren dafür

einige Prädikate:

$$\frac{\forall (n, n') \in S.\ \exists xs.\ matched\ n\ xs\ n'}{matchedSet\ S} \tag{3.14}$$

$$\frac{\forall (n, p, n') \in S.\ n\ \text{s-}p\rightarrow_{\text{sum}}\ n'}{summarySet\ S} \tag{3.15}$$

$$\frac{\begin{array}{l} matchedSet\ (set\ pathEdge) \qquad matchedSet\ (set\ workList) \\ summarySet\ (set\ summaryEdge) \end{array}}{matchedState\ (pathEdge, workList, summaryEdge)} \tag{3.16}$$

matchedState ist genau dann erfüllt, wenn *workList* und *pathEdge* nur SLR-Pfade und *summaryEdge* nur Summary Kanten enthält. Hier soll gezeigt werden, dass matchedState invariant unter Aufruf von updateState ist.

Aus der Definition des matched-Prädikats im SDG-Framework folgen drei Eigenschaften:

P1: Das Anhängen von zwei SLR-Pfaden ergibt einen neuen SLR-Pfad:

$$\textbf{lemma:}\ \frac{matched\ a\ ns\ b \qquad matched\ b\ ms\ c}{matched\ a\ (ns@ms)\ c} \tag{3.17}$$

P2: Eine einzelne DD- oder CD Kante ist ein SLR-Pfad.

P3 : Eine einzelne Summary Kante u s-$p\rightarrow_{\text{sum}}$ v ist ein SLR-Pfad (u, v).

appendDefaultEdges hängt passende DD-, Summary- und CD- Kanten vor einen übergebenen Pfad (v, w). Nach P2 und P3 handelt es sich bei den angehängten Pfaden immer um SLR-Pfade. Die resultierenden Pfade sind nach Lemma 3.17 wieder ein SLR-Pfad, wenn (v, w) ein SLR-Pfad ist. Somit werden nur SLR-Pfade propagiert. Es folgt, dass matchedState unter appendDefaultEdges invariant ist:

$$\textbf{lemma:}\ \frac{\exists ns.\ matched\ v\ ns\ w \qquad matchedState\ s}{matchedState\ (appendDefaultEdges\ G\ s\ (v, w))} \tag{3.18}$$

Als nächstes betrachten wir for12. Wir setzen die Invariante voraus:

$$matchedState\ s \tag{3.19}$$

$for12\ G\ v\ w\ s\ y$ fügt die Kante $(getCorActualIn\ G\ v\ y, getProc\ v, y)$ als neue P-Summary Kante hinzu. Damit matchedState durch den Aufruf erhalten bleibt, muss diese Kante eine gültige Summary Kante sein. Dazu müssen

die in Lemma 3.13 identifizierten Bedingungen erfüllt sein. Wir setzen diese Bedingungen voraus, es müssen

$$\begin{array}{l} v \in formalIns\ G \\ w \in formalOuts\ G \\ y \in set\ (getCallingActualOuts\ G\ w) \\ \exists xs.\ matched\ v\ xs\ w \end{array} \tag{3.20}$$

gelten. for12 wird nur von findSummaryEdges aufgerufen, nach Definition wird dadurch die Bedingung

$$y \in set\ (getCallingActualOuts\ G\ w)$$

sichergestellt. Daraus folgt

$$\textbf{lemma:}\ \frac{\begin{array}{ll} matchedState\ s & \exists xs.\ matched\ v\ xs\ w \\ v \in formalIns\ G & w \in formalOuts\ G \end{array}}{matchedState\ (findSummaryEdges\ G\ s\ (v,w))} \tag{3.21}$$

Nun können wir die Invarianz von matchedState unter updateState zeigen. Wir setzen die Invarianten

$$matchedState\ s$$

und

$$sndInFormalOuts\ G\ s$$

voraus. Der Aufruf $updateState\ G\ s$ entfernt (v,w) aus $getWorkList\ s$, dadurch werden die Invarianten nicht verletzt. Aus den Invarianten folgt, dass w ein Formal Out Knoten und (v,w) ein SLR-Pfad ist. Für den Fall, dass v ein Formal In Knoten ist, sind alle in Lemma 3.21 geforderten Bedingungen erfüllt und der Aufruf von findSummaryEdges verletzt nicht die Invarianten. Auch die Prämissen von Lemma 3.18 sind immer erfüllt, ein Aufruf von appendDefaultEdges erhält damit alle Invarianten. Mit Lemma 3.9 folgt

$$\textbf{lemma:}\ \frac{matchedState\ s \qquad sndInFormalOuts\ G\ s}{matchedState\ (updateState\ G\ s)} \tag{3.22}$$

Die matchedState-Invariante gilt auch für Startzustände, da einelementige Pfade (n,n) immer SLR-Pfade sind. Mit Lemma 3.10 auf Seite 28 folgt durch Induktion schließlich

$$\textbf{lemma:}\ \frac{isStartState\ s}{matchedState(whileLoop\ G\ s)} \tag{3.23}$$

Mit der Definition von matchedState folgt

$$\textbf{lemma:}\ \frac{isStartState\ s}{\forall (n,p,n') \in (getSummarySet(whileLoop\ G\ s)).\ n\ \text{s-}p\rightarrow_{\text{sum}}\ n'} \tag{3.24}$$

was zu zeigen war.

3.3 Vollständigkeit des Algorithmus

In diesem Abschnitt wird gezeigt, dass alle Summary Kanten des SDGs gefunden werden. Dazu wird gezeigt, dass *pathEdge* nach Ablauf des Algorithmus alle SLR-Pfade enthält, die in einem Formal Out Knoten enden. Der Beweis wird wieder nur für den fixierten, korrekten SDG G geführt, das in Abschnitt 3.2 verwendete locale wird exakt übernommen.

3.3.1 Beeinflussung von Pfaden

Wir definieren die n-te Rekursion von updateState:

$$\begin{aligned}
&\textbf{fun } \mathit{whileLoopN} :: \textbf{nat} \Rightarrow \textbf{Graph} \Rightarrow \textbf{State} \Rightarrow \textbf{State where}\\
&\mathit{whileLoopN}\ 0\ g\ s = s\\
&\mid \mathit{whileLoopN}\ (\mathit{Suc}\ n)\ g\ s = \mathit{whileLoopN}\ n\ g\ (\mathit{updateState}\ g\ s)
\end{aligned}$$

Ein Pfad p *beeinflusst das Endergebnis*, wenn p während der Abarbeitung von whileLoop am Anfang der *workList* stand und selektiert wurde. Ein wichtiges Lemma zeigt nun, dass alle Pfade in der finalen *pathEdge* das Endergebnis beeinflussen:

$$\textbf{lemma: } \frac{\mathit{isStartState}\ s \qquad p \in \mathit{getPathSet}\ (\mathit{whileLoop}\ G\ s)}{\exists n\ xs.\ \mathit{getWorkList}\ (\mathit{whileLoopN}\ n\ g\ s) = p\#xs} \tag{3.25}$$

Der Beweis nutzt aus, dass ein Pfad (u, v), der in *pathEdge* eingefügt wird, gleichzeitig auch in *workList* eingefügt wird und damit - da *workList* nach Durchlauf des Algorithmus komplett abgearbeitet wurde - auch in einem dazwischenliegenden Schritt selektiert werden muss.

3.3.2 Induktionsregeln für matched

In Abschnitt 3.3.3 wird eine Induktionsregel benutzt, die hier erläutert werden soll. Die im SDG-Framework vorhandenen Regeln für *matched* zeigen, dass das Anhängen von intraprozeduralen und Summary Kanten an das Ende eines SLR-Pfads erneut einen SLR-Pfad ergibt. Der Algorithmus dieser Arbeit hängt jedoch neue Kanten *vor* bereits gefundene SLR-Pfade. Aus diesem Grund musste die bereits vorhandene Induktionsregel für matched so modifiziert werden, dass sie das Anhängen von Pfaden an den Anfang eines SLR-Pfades berücksichtigt. Die resultierende Regel ist Lemma 3.26.

Die Regel ist in 4 Fälle aufgeteilt. Der erste Fall ist der Induktionsanfang. In den drei letzten Induktionsregeln wird vorausgesetzt, dass ein zu zeigendes Prädikat P für das Ende eines Pfades (n_4, n_5) gilt. Wenn nachgewiesen wird, dass

- P erhalten bleibt, wenn ein intraprozeduraler Pfad der Länge 1 vor (n_4, n_5) angehängt wird, und dass
- P erhalten bleibt, wenn n_4 das Ende einer CR-Summary Kante ist und diese CR-Summary Kante vor n_4 angehängt wird, und dass
- P erhalten bleibt, wenn n_4 das Ende einer P-Summary Kante ist und diese P-Summary Kante vor n_4 angehängt wird.

Wenn auch der Induktionsanfang gezeigt werden kann, folgt

$$matched\ n\ ns\ n' \Longrightarrow P\ n\ ns\ n'$$

Lemma :

$$\frac{\begin{array}{l} matched\ n\ ns\ n' \\ \bigwedge n.\ P\ n\ [\,]\ n \\ \bigwedge n_3\ ns\ n_4\ n_5.\ [\![n_3\ \text{i-}[n_3]\rightarrow_{\text{d*}}\ n_4;\ matched\ n_4\ ns\ n_5;\ P\ n_4\ ns\ n_5]\!] \\ \quad \Longrightarrow P\ n_3\ (n_3\#ns)\ n_5 \\ \bigwedge n_1\ n_2\ n_3\ n_4\ n_5\ ns\ p.\ [\![matched\ n_4\ ns\ n_5;\ P\ n_4\ ns\ n_5;\ n_1\ \text{-}p\rightarrow_{\text{call}}\ n_2 \\ matched\ n_2\ ns'\ n_3\ ; P\ n_2\ ns'\ n_3;\ n_3\ \text{-}p\rightarrow_{\text{ret}}\ n_4;\ \text{call_of_return_node}\ n_4\ n_1]\!] \\ \quad \Longrightarrow P\ n_1\ (n_1\#ns'@[n_3]@ns)\ n_5 \\ \bigwedge n_1\ n_2\ n_3\ n_4\ n_5\ ns\ ns'\ p\ V\ V'.\ [\![n_1\ \text{-}p{:}V\rightarrow_{\text{in}}\ n_2;\ matched\ n_2\ ns'; n_3; \\ P\ n_2\ ns'\ n_3;\ n_3\ \text{-}p{:}V'\rightarrow_{\text{out}}\ n_4;\ matched\ n_4; ns\ n_5\ P\ n_4\ ns\ n_5;]\!] \\ \quad \Longrightarrow\ P\ n_1\ (n_1\#ns'@[n_3]@ns)\ n_5 \end{array}}{P\ n\ ns\ n'} \tag{3.26}$$

3.3.3 Vollständigkeit bezüglich matched

Es lässt sich zeigen, dass jeder SLR-Pfad, der in einem Formal Out Knoten endet, vom Algorithmus gefunden wird:

$$\textbf{lemma:}\ \frac{matched\ n\ ns\ n' \qquad n' \in formalOuts\ G \qquad isStartState\ s}{(n,\ n') \in getPathSet\ (whileLoop\ G\ s)} \tag{3.27}$$

Zum Beweis benutzten wir die in Abschnitt 3.3.2 eingeführte Induktionsregel. Wir müssen zeigen, dass ein SLR-Pfad (n_1, n_5) in der finalen *pathEdge* liegt. Nach der Induktionsregel dürfen wir voraussetzen, dass (n_1, n_5) aus den Teilpfaden (n_1, n_4) und (n_4, n_5) besteht, (n_4, n_5) in der finalen *pathEdge* liegt und (n_1, n_4) - je nach Fall - eine CD-, DD- oder Summary Kante ist. Mit Lemma 3.25 erhalten wir nun ein $n \in \mathbb{N}_0$, sodass (n_4, n_5) nach n Rekursionen von whileLoop selektiert wird.
Für jeden Fall der Induktionsregel folgern wir, dass durch die Selektion von

(n_4, n_5) Pfad (n_1, n_4) gefunden und vor (n_4, n_5) gehängt wird. Somit liegt (n_1, n_5) nach $n + 1$ Durchläufen in *pathEdge* und wird dort bis zum Ende der Rekursion nicht mehr entfernt.

3.3.4 Vollständigkeit aller Summary Kanten

Wir haben nun alle Mittel, um den Beweis zu Ende zu führen. Wir möchten zeigen, dass jede Summary Kante n_1 s-$p\rightarrow_{\text{sum}}$ n_4 durch den Beweis gefunden wird.

Falls (n_1, n_4) eine CR-Summary Kante ist, so liegt (n_1, n_4) nach Definition in der *summaryEdge* des Startzustands und damit auch im finalen Zustand.
Sei $(n1, n4)$ nun eine P-Summary Kante. n_2 und n_3 seien die daran beteiligten Formal In- und Formal Out Knoten, (n_2, n_3) ist ein SLR-Pfad. Nach Lemma 3.27 liegt (n_2, n_3) auch in der finalen *pathEdge*. Mit Lemma 3.25 erhalten wir ein n, sodass (n_2, n_3) in der n-ten Rekursion selektiert wird:

$$\exists xs.\ getWorkList\ (whileLoopN\ n\ G\ s) = (n_3, n_4)\#xs$$

Es lässt sich zeigen, dass (n_1, n_4) in dieser Rekursion gefunden wird (hier nicht weiter erläutert). Daraus folgt das finale Lemma:

$$\textbf{lemma:}\ \frac{n\ \text{s-}p\rightarrow_{\text{sum}}\ n' \qquad isStartState\ s}{(n, p, n') \in getSummaryEdge\ (whileLoop\ G\ s)} \tag{3.28}$$

was zu zeigen war.

Kapitel 4

Fazit und Ausblick

4.1 Fazit

Ziel dieser Arbeit war das Erstellen eines ausführbaren, verifizierten Algorithmus zum Finden von Summary Kanten in SDGs, die sich nach den Konventionen des SDG-Frameworks von Wasserrab[10] richten.
Dazu wurden in insgesamt 2796 Zeilen Beweiscode (663 Zeilen für Definition und Termination, 806 Zeilen für den Beweis der Korrektheit und 1327 Zeilen für den Beweis der Vollständigkeit) die geforderten Eigenschaften des Algorithmus verifiziert.

Erreicht wurde dieses Ziel insofern, als dass der Code eines ausführbaren, verifizierten Algorithmus exportiert werden kann, der bei Eingabe eines gültigen Startzustands alle Summary Kanten des SDGs findet. Der Algorithmus konnte erfolgreich in Haskell exportiert werden. Die in den Beweisen identifizierten Invarianten liefern außerdem einen wertvollen Einblick in die Funktionsweise des in "Speeding up Slicing"[8] beschriebenen Algorithmus.
Nicht erreicht wurde das Ziel insofern, als dass das Berechnen von Startzuständen nur auf Basis eines SDG noch nicht möglich ist. Eine aussagekräftige Evaluation, die das Laufzeitverhalten unter realistischen Bedingungen bewertet, wurde auch nicht vorgenommen.

4.2 Generierung des Codes

In Isabelle definierte Funktionen können über den *Code Generator* in Haskell, SML, OCaml und Scala exportiert werden. Das grundlegende Prinzip ist das *shallow embedding*[3]: Grundlegende Terme in Isabelle, zum Beispiel Konstanten und Typen, werden mit den korrespondierenden Termen in der Zielsprache über *Code Gleichungen* in Bezug gesetzt. Damit wird sichergestellt, dass die Semantik erhalten bleibt. Einige Ausdrücke, wie beispielsweise der Existenzquantor $\exists$ oder der Allquantor $\forall$, haben jedoch kein Gegenstück

in der Zielsprache. Daher wurden bereits bei der Erstellung des Algorithmus nur ausführbare Terme verwendet.
Für die Code-Generierung müssen die im locale fixierten Funktionen getProc und getCorCallNode implementiert werden. In der Testimplementierung wurde dazu der Typ **SDG_node** um einen Eintrag für den Prozedurbezeichner und für (falls vorhanden) den korrespondierenden Call Knoten erweitert, sodass diese Funktionen nur den entsprechenden Parameter des übergebenen Knotens zurückgeben müssen. Außerdem müssen die Funktionen αe, inEdges und outEdges implementiert werden. Dazu wurde die *mapped graph* Implementierung verwendet, die im Graph-Framework enthalten ist und die benötigten Funkionen liefert. Die Implementierung verwaltet alle Knoten in einem Suchbaum. Daher muss auf dem Knotentyp **SDG_node** eine lineare Ordnung definiert sein. Für das Definieren dieser Ordnung wurde der *Order Generator* von René Thiemann[1] verwendet, der lineare Ordnungen für die meisten Datentypen einfach generieren kann.

Im Rahmen dieser Arbeit wurde manuell ein Testgraph erstellt, mit dem der exportierte Haskellcode getestet wurde. Der Testgraph besteht jedoch aus nur einem Prozeduraufruf und nur wenigen Knoten, da das händische Erstellen sehr aufwendig ist. Der Graph kann jedoch als Konstruktionsbeispiel für größere Graphen dienen. Für den Test wurden die Knoten und Kanten des Graphen direkt in Haskell erstellt, die nötigen Funktionen zum Hinzufügen von Knoten und Kanten stammen aus der mapped graph Implementierung. Der fertige Graph wird in Abbildung 4.1 dargestellt.

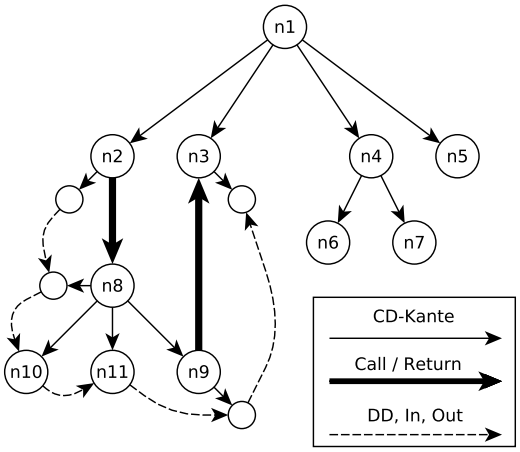

Abbildung 4.1: Beispielgraph, mit dem der Algorithmus getestet wurde

[1] http://afp.sourceforge.net/entries/Datatype_Order_Generator.shtml

$p2$ bezeichne die Prozedur, in der Knoten $n8$ liegt. Der Graph enthält nur einen einzigen Formal Out Knoten $fOut$ und eine einzige CR-Summary Kante $(n2, p2, n3)$. Gemäß Definition von isStartState ergibt sich der Startzustand

$$start := ([(fOut, fOut)], [(fOut, fOut)], [(n2, p2, n3)])$$

Ein Aufruf von whileLoop mit dem Beispielgraphen und dem Startzustand erkennt nun korrekterweise die einzige Summary Kante zwischen dem einzigen Actual In und dem einzigen Actual Out Knoten.

4.3 Ausblick

Der Algorithmus kann momentan nicht genutzt werden, da keine Startzustände berechnet werden können. Diese Berechnung sollte daher in jedem Fall nachgeholt werden. Denkbar sind zwei Ansätze:

- Zuerst wird der SDG erstellt, anschließend wird über alle Knoten iteriert. Während der Iteration werden alle Formal Out Knoten und CR-Summary Kanten identifiziert.
- Während dem Erstellen des SDGs wird eine Liste mit Formal Out Knoten mitgeführt, beim Erstellen einer Call Site wird darauf geachtet, dass der neue Call Knoten mit dem neuen Return Knoten über eine CR-Summary Kante verbunden wird. Somit kann eine Liste an CR-Kanten mitgeführt werden, aus der sich mit den gesammelten Formal Out Knoten ein gültiger Startzustand ergibt.

Weiterhin sollte eine Evaluation des Algorithmus mit einigen SDGs aus echten Programmen vorgenommen werden, um die Laufzeiteigenschaften besser abschätzen zu können. Dazu kann auf bestehende Software zum Erstellen von SDGs zurückgegriffen werden (zum Beispiel Joana[2]), die resultierenden SDGs müssen anschließend an den Algorithmus weitergereicht werden. Dazu muss der SDG der Software zuerst auf das Eingabeformat des Algorithmus angepasst und in ein für Haskell lesbares Austauschformat konvertiert werden. Der in dieser Arbeit erstellte Beispielgraph vermittelt, wie ein solches Eingabeformat aussehen könnte und wie sich geeignete Graphen in Haskell erstellen lassen.
Je nach Ergebnis der Laufzeiteigenschaften könnte eine Optimierung des Algorithmus von Interesse sein. Die Laufzeit sollte sich durch geschicktere Wahl der Datenstrukturen verringern lassen, beispielsweise könnte $pathEdge$ als Suchbaum modelliert werden. Dadurch würde der Aufwand von propagate von $O(|pathEdge|)$ auf $O(log(|pathEdge|))$ verringert werden.

[2] `http://pp.ipd.kit.edu/projects/joana/`

Der Algorithmus kann in einigen Hinsichten noch vereinfacht werden. Durch eine Vereinfachung ließe sich das locale des Algorithmus (Abschnitt 2.8) leichter instantiieren. Denkbar sind hierbei folgende Ansätze:

- Auf Funktion getProc kann verzichtet werden. Sie wird nur benutzt, um den Prozedurbezeichner für eine Summary Kante zu ermitteln. Dieser könnte allerdings auch aus den Parameterkanten erhalten werden, an die dieser Bezeichner angehängt ist.
- Der Datentyp **SDGEdgeData** könnte vereinfacht werden, indem die an die Kanten angehängten Parameter weggelassen werden. Diese werden vom Algorithmus nicht benutzt.
- Auf Funktion getCorCallNode kann verzichtet werden. Der korrespondierende Call Knoten n kann zu einem Return Knoten n' erhalten werden, indem eine in n' endende CR-Summary Kante n s-$p\rightarrow_{\text{sum}}$ n' gefunden wird (Es existiert mindestens eine dieser Kanten). Die Regeln des SDG-Frameworks erzwingen, dass diese im korrespondierenden Call Knoten beginnt.

Das größere Ziel hinter dieser Arbeit ist ein ausführbarer, verifizierter HRB-Slicer für gängige Hochsprachen. Der HRB-Slicer arbeitet in vier Schritten:

1. Transformiere die Prozeduren des Eingangsprogramm in Kontrollflussgraphen.
2. Transformiere die Kontrollflussgraphen in einen SDG ohne Summary Kanten.
3. Erweitere diesen SDG um Summary Kanten.
4. Berechne mithilfe des erweiterten SDGs den finalen Slice.

Schritt 1 wird mit einem Algorithmus aus der Arbeit von Simon Kohlmeyer für die Sprache WHILE vollbracht. Hier fehlen noch verifizierte Algorithmen, die andere, mächtigere Sprachen, z.B. Jinja[5] oder CoreC++[9] transformieren können. Für die Transformation im zweiten Schritt existieren zwar einige bekannte und in der Praxis genutzte Algorithmen[4], eine formal verifizierte Variante fehlt jedoch noch. Der dritte Schritt wird durch den Algorithmus in dieser Arbeit, wie im Fazit geschildert, vollführt. Ein verifizierter Algorithmus, der den finalen Slice aus dem SDG berechnet, muss noch erstellt werden.
Bis ein voll funktionsfähiger, verifizierter Slicer vorhanden ist, ist noch viel Arbeit zu erledigen. Mit dem hier vorgestellten Algorithmus ist das Ziel jedoch wieder ein kleines Stück näher gekommen.

Kapitel 5

Anhang

5.1 Namen der Lemmas in Isabelle

Die folgende Tabelle ordnet die Nummer eines Lemmas dem Namen des Lemmas im Quellcode zu. Die zweite Spalte gibt an, in welcher Datei das Lemma gezeigt wird.

Alg bezieht sich auf *SummaryEdgeComputationAlgorithm.thy*.

1 bezieht sich auf *SummaryEdgeComputationProof1*.

2 bezieht sich auf *SummaryEdgeComputationProof2*.

Nummer	**Datei**	**Name in Isabelle**
3.2	Alg	closureFinite
3.3	Alg	pathEdgeInClosure
3.4	Alg	getClosure_mono
3.5	Alg	peCard_mono
3.6	Alg	peCard_dec_mono
3.9	1	sifo_updateState
3.10	1	sifo_startState
3.13	1	sumEdge_between_cor_pair
3.18	1	ms_appendDefaultEdges
3.21	1	ms_findSummaryEdges
3.22	1	ms_updateState
3.23	1	ms_whileLoop
3.24	1	SEC_proof1
3.25	2	infl_whileLoopN
3.26	2	matched_rev_induct
3.27	2	matchedInWlPathEdge
3.28	2	SEC_proof2

Kapitel 6

Literaturverzeichnis

[1] Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. Finding lexicographic orders for termination proofs in isabelle/hol. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732, pages 38–53, 2007.

[2] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[3] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009 of *Lecture Notes in Computer Science*. Springer, 2010.

[4] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[5] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[6] Kevin-Simon Kohlmeyer. Funktionale konstruktion und verifikation von kontrollflussgraphen, August 2012.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[8] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, December 1994.

[9] Daniel Wasserrab. Formale semantik einer c++-ähnlichen sprache mit fokussierung auf mehrfachvererbung. Master's thesis, Technische Universität München, 2004.

[10] Daniel Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security.* PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.

Erklärung

Hiermit erkläre ich, Stefan Altmayer, dass ich die vorliegende Bachelorarbeit selbstständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Ich habe mich über die Regeln zu Sicherung guter wissenschaftlicher Praxis informiert und sie in in dieser Arbeit beachtet.

Ort, Datum	Unterzeichner