TECHNISCHE UNIVERSITÄT
ILMENAU

Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Programmiersprachen und Compilertechnik

Masterarbeit

Synthese von Kontrollfluss für eine synchrone Datenflusssprache

Vorgelegt von:

Andreas Loth

Betreuer: Prof. Dr. Baltasar Trancón Widemann
Zweitgutachter: Dr. Carl Friedrich Bolz (King's College London)

Studiengang: Master Informatik
Matrikel-Nr: 43358

Eingereicht: Ilmenau, 13.03.2015

Kurzfassung

Die synchrone Datenflusssprache SIG unterstützt Fallunterscheidungen. Im bisherigen SIG-Compiler werden alle Fälle berechnet und danach die benötigten Werte ausgewählt. Nicht benötigte Werte werden verworfen. Das Ziel dieser Arbeit ist das Überspringen im konkreten Fall nicht benötigter Berechnungen indem die Bedingungen, die den Fall bestimmen, vor den Berechnungen ausgewertet werden. Neben der Erstellung der Sprünge ist auch die effiziente Prüfung der Sprungbedingungen wichtig. Dazu werden die Bedingungen vereinfacht und die Sprünge so organisiert, dass weniger Bedingungen geprüft werden müssen. Bei den Transformationen auftretende unerwünschte Effekte werden durch Techniken wie Dead Code Elimination und Common Subexpression Elimination entfernt. Weiterhin wird die nächste Compilerphase vorbereitet, indem berechnet wird, unter welchen Bedingungen Variablen definiert sind und Basic Blocks ausgeführt werden. Die Evaluation zeigt, dass der erzeugten Kontrollfluss die Struktur der Programme gut abbildet. In bestimmten Situationen ist das Ergebnis nicht optimal, kann jedoch mit Partial Dead Code Elimination ohne Änderung des Kontrollflusses verbessert werden. Auch die Berechnung der Bedingungen, unter denen die Variablen definiert sind, arbeitet korrekt, jedoch werden logische Zusammenhänge von Variablen nicht erfasst.

Abstract

The synchronous dataflow language SIG supports case differentiation. In the current SIG compiler all cases are computed and the needed values are selected afterwards. Values which are unneeded are discarded. The aim of this thesis is to jump over the computations which are unneeded in the current case by evaluating the conditions of the current case before executing the unneeded computations. Beside the creation of the jumps, the efficient evaluation of the jump conditions is important. Therefore, the conditions are simplified and the jumps are organized in a way which allows to evaluate less conditions. During the transformations, undesired effects can occur. To eliminate this effects techniques like dead code elimination and common subexpression elimination are used. Furthermore, the next compiler phase is prepared by calculating the conditions under which variables are defined and basic blocks are executed. The evaluation shows that the generated controlflow represents the structure of the SIG program well. However, in certain situations the result is not optimal but it can be improved by partial dead code elimination without changing the controlflow. Furthermore, the calculation of the definedness of the variables is correct even though logical connections between variables are not recognized.

Danksagung

Meiner Familie möchte ich mich für die Unterstützung während meines Studiums danken. Ohne euch wäre es so nicht möglich gewesen.

Auch meinen Freunden gilt Dank für ihre Unterstützung. Ohne euch hätte ich im Studium definitiv mehr Probleme gehabt. Danke auch für das Gegenlesen dieser Arbeit!

Danke auch an meinen Betreuer Prof. Widemann für das Thema und die Unterstützung bei dessen Bearbeitung.

Für die nützlichen Anmerkungen zu meiner MA bedanke ich mich auch bei Dr. Bolz.

Schließlich bedanke ich mich auch bei den Entwicklern der Tools, ohne die ich diese Arbeit nicht hätte schreiben können: LibreOffice 4 (Sun / Oracle / The Document Foundation), Eclipse Luna (Eclipse Foundation) und Java (Sun / Oracle).

Werner Seiler (https://commons.wikimedia.org/wiki/User:Centine), gemeinfrei / public domain https://commons.wikimedia.org/wiki/File:AnthocharisCardamines_1606Z1.jpg?uselang=de

Inhaltsverzeichnis

1 Einleitung....1
1.1 Zielstellung....1
1.2 Aufbau der Arbeit....1
1.3 Related Work....2
2 Grundlagen....3
2.1 Zwischenrepräsentation....3
2.2 Kontrollflussgraph....3
2.3 Static Single Assignment Form (SSA)....3
2.4 Datenflusssprachen....4
3 Die Datenflusssprache Sig....7
4 Synthese von Kontrollfluss....11
4.1 Zwischenformate....13
4.1.1 Eingabe- und Ausgabeformat....13
4.1.2 Das Zwischenformat dieser Arbeit....16
4.2 Vereinfachung von Bedingungen....17
4.2.1 Konstante boolesche Variablen....18
4.2.2 Zurückrechnen boolescher Verknüpfungen....18
4.2.3 Exklusive Variablenmengen....19
4.2.4 Idempotenz....21
4.2.5 Neutrales Element....21
4.2.6 Wahre und falsche Variablen....22
4.2.7 Variable und ihre „Negation“ innerhalb eines Konjunktionsterms....22
4.2.8 „Negation“ einer einzigen Variablen....23
4.2.9 Absorptionsgesetz....24
4.3 Def-Use Kette....24
4.4 Notwendigkeit von Variablen und Statements....24
4.4.1 Algorithmus....25
4.4.2 Abhängigkeiten....26
4.5 Synthese von Sprüngen aus Guards....26
4.5.1 Algorithmus....27
4.5.2 Abhängigkeiten....29
4.6 Statements mit mehreren direkten Guards....29
4.6.1 Algorithmus....30
4.6.2 Abhängigkeiten....31
4.7 Verschieben der Berechnungen von Kontrollvariablen....32
4.7.1 Algorithmus....33
4.7.2 Abhängigkeiten....35
4.8 Schachtelung der Sprünge....35
4.8.1 Algorithmus....36
4.8.2 Abhängigkeiten....37
4.9 Sprungbedingungen vereinfachen....37
4.9.1 Algorithmus....39
4.9.2 Abhängigkeiten....41

4.10 Sprünge mit der gleichen Bedingung....41
4.10.1 Algorithmus....44
4.10.2 Abhängigkeiten....45
4.11 Copy Propagation....45
4.11.1 Algorithmus....47
4.11.1.1 Ersetzungstabelle....47
4.11.1.2 Aufbau der Ersetzungstabelle....48
4.11.1.3 Anwenden der Ersetzungen....49
4.11.2 Abhängigkeiten....49
4.12 Sprünge organisieren....49
4.12.1 Algorithmus....51
4.12.2 Abhängigkeiten....52
4.13 Dead Code Elimination (DCE)....53
4.13.1 Algorithmus....53
4.13.2 Abhängigkeiten....54
4.14 Generierung des Codes zur Bedingungsprüfung....55
4.14.1 Algorithmus....56
4.14.2 Abhängigkeiten....56
4.15 Partial Redundancy Elimination (PRE) / Common Subexpression Elimination (CSE)....57
4.15.1 Modifikationen von CSE....59
4.15.2 Algorithmus....60
4.15.3 Abhängigkeiten....61
4.16 Überführung des Zwischenformats ins Ausgabeformat....62
4.16.1 Erzeugung der Basic Blocks....62
4.16.2 Überführung....63
4.16.3 Abhängigkeiten....66
4.17 Aufräumen der lokalen Variablen....66
4.17.1 Algorithmus....67
4.17.2 Abhängigkeiten....67
5 Vorbereitung der nächsten Compilerphase....69
5.1 Algorithmus....69
5.1.1 (Inverser) Kontrollflussgraph....69
5.1.2 Berechnung der Bedingungen....70
6 Evaluation....73
6.1 Synthese von Kontrollfluss (Kapitel 4)....73
6.1.1 sah (Sample & Hold klassisch; 2 Eingänge)....73
6.1.2 sah2 (Sample & Hold algebraisch; 1 Eingang)....74
6.1.3 choose (Auswahl)....76
6.1.4 rle (Run Length Encoding)....79
6.1.5 adsr (Attack/Decay/Sustain/Release Hüllkurvengenerator)....81
6.1.6 triangle (Dreieck Wellenform-Generator)....84
6.1.7 rectangle (Rechteck Wellenform-Generator)....85
6.1.8 Zusammenfassung....87
6.2 Definiertheitsanalyse (Kapitel 5)....87
6.2.1 sah2-1....87
6.2.2 sah2-2....88

7 Zusammenfassung und Ausblick....89
7.1 Zusammenfassung....89
7.2 Ausblick....90
8 Anhang....93
8.1 Ausgangsprogramme der Beispiele....93
8.2 Zwischenergebnisse der Evaluation....107
Literaturverzeichnis....135

Listing-Verzeichnis

Listing 1: Sample & Hold Schaltung....15
Listing 2: Zwischenformat Programmliste von optimierter Sample & Hold Schaltung. 17
Listing 3: Beispiel für Bedingung mit einer Variablen aus dem Kontext....22
Listing 4: Beispiel direkter Guard....25
Listing 5: Alle Anweisungen werden ausgeführt....26
Listing 6: Sprünge überspringen fallweise nicht benötigte Anweisungen....27
Listing 7: Anweisung, die nicht in jedem Fall benötigt wird (mehrere direkte Guards). 29
Listing 8: Teilweise nicht benötigte Anweisung wird übersprungen....30
Listing 9: Sprungbedingung mit undefinierten Kontrollvariablen....32
Listing 10: Berechnung der Kontrollvariablen vor Sprungbedingung....33
Listing 11: Sprünge wurden geschachtelt....36
Listing 12: Sprungbedingungen können vereinfacht werden....38
Listing 13: Vereinfachte Sprungbedingungen....39
Listing 14: Sprünge mit gleichen Bedingungen können zusammengefasst werden....42
Listing 15: Sprünge wurden zusammengefasst....43
Listing 16: Beispiel für Effekte zweiter Ordnung....44
Listing 17: Notwendige und nicht notwendige Kopieranweisungen....46
Listing 18: Ergebnis der Copy Propagation....47
Listing 19: Eine bessere Organisation der Sprünge ist möglich....50
Listing 20: Sprünge mit If-Then-ElseIf-… Semantik....50
Listing 21: DCE entfernte nicht genutzte Variablendefinition....53
Listing 22: Sprungbedingungen werden nicht berechnet....55
Listing 23: Sprungbedingungen werden explizit berechnet....56
Listing 24: Gleiche Ausdrücke auf unterschiedlichen Pfaden....58
Listing 25: Beispiel für lexikalisch unterschiedliche Variablen mit gleichem Wert....59
Listing 26: Gewünschtes Ergebnis nach CSE....60
Listing 27: Aus Beispiel in Listing 2 generierte Basic Blocks....62
Listing 28: Beispiel mit vielen Local-Variablen....66
Listing 29: Das Aufräumen der Local-Variablen hat deren Anzahl reduziert....67
Listing 30: Sig-Programm sah....73
Listing 31: Sig-Programm sah2....74
Listing 32: Sig-Programm choose....77
Listing 33: choose direkt vor der Organisation der Sprünge (Abschnitt 4.12)....78
Listing 34: choose direkt nach der Organisation der Sprünge (Abschnitt 4.12)....78
Listing 35: Sig-Programm rle....79
Listing 36: Sig-Programm adsr....81
Listing 37: Pseudocode Kontrollfluss adsr....83
Listing 38: Sig-Programm triangle....84
Listing 39: Sig-Programm rectangle....86
Listing 40: Ergebnis Definiertheitsanalyse sah2-1....87
Listing 41: Ergebnis Definiertheitsanalyse sah2-2....88
Listing 42: komplexe Bedingungen, flache Struktur....91
Listing 43: verdecktes Optimierungspotenzial....92
Listing 44: Ausgangsprogramm für Beispiele in Listings 5, 6....93
Listing 45: Ausgangsprogramm für Beispiele in Listings 7, 8....94
Listing 46: Ausgangsprogramm für Beispiele in Listings 9, 10, 11....95
Listing 47: Ausgangsprogramm für Beispiele in Listings 12, 13....96
Listing 48: Ausgangsprogramm für Beispiele in Listings 14, 15....97

Listing 49: Ausgangsprogramm für Beispiele in Listings 17, 18.....98
Listing 50: Ausgangsprogramm für Beispiele in Listings 19, 20, 21.....98
Listing 51: Ausgangsprogramm für Beispiele in Listings 22, 23.....99
Listing 52: Ausgangsprogramm für Beispiele in Listings 28, 29, 53, 54.....101
Listing 53: Vollständiges Programm von Listing 28.....103
Listing 54: Vollständiges Programm von Listing 29.....105
Listing 55: Ausgangsprogramm für Beispiel in Listing 42.....106
Listing 56: Zwischenformat sah.....107
Listing 57: optimiertes sah.....108
Listing 58: Zwischenformat sah2.....109
Listing 59: optimiertes sah2-1.....110
Listing 60: optimiertes sah2-2.....111
Listing 61: Zwischenformat choose.....112
Listing 62: optimiertes choose.....113
Listing 63: Zwischenformat rle.....114
Listing 64: optimiertes rle.....115
Listing 65: Zwischenformat adsr.....120
Listing 66: optimiertes adsr.....125
Listing 67: Zwischenformat triangle.....127
Listing 68: optimiertes triangle.....129
Listing 69: Zwischenformat rectangle.....131
Listing 70: optimiertes rectangle.....133

Abbildungsverzeichnis

Abbildung 1: Addierer.....7
Abbildung 2: Integrierer – erlaubte Rückkopplung durch Verzögerung.....7
Abbildung 3: Unerlaubter Zyklus.....7
Abbildung 4: Auflösung der Ein-Schritt-Verzögerung zu Zustand.....8
Abbildung 5: Beispiel für Pattern Matching und partiell undefinierte Ausgänge.....8
Abbildung 6: Konstruktoren für Beispiel in Abbildung 5.....9
Abbildung 7: Sample & Hold Schaltung.....10
Abbildung 8: Abhängigkeiten der Verfahren.....12
Abbildung 9: Beispiel, in dem CSE und PRE unterschiedliche Ergebnisse liefern.....58
Abbildung 10: Kontrollflussgraph sah.....74
Abbildung 11: Kontrollflussgraph sah2-1.....75
Abbildung 12: Kontrollflussgraph sah2-2.....76
Abbildung 13: Kontrollflussgraph choose.....77
Abbildung 14: Kontrollflussgraph rle.....80
Abbildung 15: Kontrollflussgraph triangle.....85
Abbildung 16: Kontrollflussgraph rectangle.....86

1 Einleitung

Programme, die in Hochsprachen geschrieben sind, können von Computern nicht direkt ausgeführt werden. Sie müssen erst in die passende Maschinensprache übersetzt werden. Diese Übersetzung ist die Aufgabe eines Compilers.

Neben der Analyse des Quellcodes und der Erzeugung des Zielcodes können Compiler das Programm auch optimieren. Das Ziel der Optimierung ist üblicherweise die Verbesserung der Laufzeit oder des Speicherverbrauchs. [WG84, Seiten 3, 273 – 298]

SIG ist eine Datenflusssprache, die auch Fallunterscheidungen unterstützt. In der derzeitigen Implementierung des Compilers werden dabei immer alle Daten berechnet und anschließend selektiert anstatt die Bedingungen vor den Berechnungen auszuwerten und nicht benötigte zu überspringen. Für die Ausführung auf massiv parallelen Architekturen, wie zum Beispiel FPGAs, ist diese Vorgehensweise geeignet. Bei der Ausführung auf sequenziellen Architekturen, wie der Von-Neumann-Architektur, wird bei dieser Vorgehensweise je nach Programm viel Zeit und Energie für die Berechnung von Daten verwendet, die später wieder verworfen werden.

1.1 Zielstellung

Der bestehende SIG-Compiler übersetzt die Fallunterscheidungen auf die oben beschriebene naive Weise. Dabei wird als Zwischenrepräsentation die SSA Form verwendet. Ohne Kontrollfluss ist damit jede Funktion ein einziger Basic Block. Diese rein datenflussbasierte Form soll äquivalent in eine kontrollflussbasierte Form überführt werden, in der nicht benötigte Berechnungszweige übersprungen werden.

Dazu müssen die folgenden Teilprobleme gelöst werden: Es muss analysiert werden, welche Berechnungen unter welchen Bedingungen verworfen werden. Weiterhin sollen die Bedingungen effizient geprüft werden, das heißt, dass sie vereinfacht und entsprechend organisiert werden müssen. Außerdem muss das Programm so umgeschrieben werden, dass es bedingte Sprünge verwendet um aktuell nicht benötigte Berechnungen zu überspringen. Schließlich sollen die in der Arbeit entstandenen Techniken mit den üblichen administrativen Analysen und Transformationen auf SSA integriert werden.

Die Lösung soll in den existierenden SIG-Compiler integriert werden. Zudem soll eine Evaluation der Lösung erfolgen.

Die Implementierungssprache des Compilers und der Ziel-Ausführungsumgebung ist Java.

1.2 Aufbau der Arbeit

Das folgende Kapitel 2 erläutert kurz was Zwischenrepräsentationen sind und eine Form der Zwischenreprästation, die SSA Form. Außerdem wird die Datenstruktur Kontrollflussgraph kurz beschrieben. Weiterhin wird ein kurzer Überblick über Datenflusssprachen gegeben.

Danach werden in Kapitel 3 die Datenflusssprache SIG und ihre Besonderheiten beschrieben.

In Kapitel 4 wird die Synthese des Kontrollflusses erläutert. Im ersten Abschnitt wird das Eingabe- und Ausgabeformat beschrieben sowie die Zwischenrepräsentation, die die in dieser Arbeit verwendeten Transformationen benutzen. Als nächstes wird das Programm analysiert. Anschließend werden die Transformationen durchgeführt und unerwünschte Effekte beseitigt. Schließlich wird das transformierte Programm in das Ausgabeformat umgewandelt und ein letzter unerwünschter Effekt entfernt.

Kapitel 5 dient der Vorbereitung der nächsten Compilerphase. Es wird berechnet, unter welcher Bedingung ein Basic Block ausgeführt wird und unter welcher Bedingung eine Variable definiert ist.

Die Evaluation der in dieser Arbeit entwickelten Verfahren wird in Kapitel 6 durchgeführt. Dabei wird die Struktur der optimierten Programme ausgewertet. Außerdem wird auch das Ergebnis der Vorbereitung der nächsten Compilerphase betrachtet.

In Kapitel 7 wird die Arbeit zusammengefasst. Weiterhin wird in diesem Kapitel ein Ausblick auf weitere Fragestellungen gegeben.

1.3 Related Work

Es liegt nahe, den nur Fallweise benötigten Code als partiell toten Code aufzufassen. Bei den Verfahren zur Partial Dead Code Elimination (PDCE) [KRS94, FKCX94, BG97] wird jedoch eine vorhandene Kontrollflussstruktur verwendet, dessen Generierung die Hauptaufgabe dieser Arbeit ist.

PDCE eliminiert Code, der partiell tot ist. Partiell toter Code ist auf einigen Programmpfaden, aber nicht notwendigerweise auf allen, tot, das heißt unnötig. [KRS94, Seite 147]

In [KRS94] wird die Sprungstruktur des Programms nicht geändert, was jedoch in dieser Arbeit notwendig ist, um Sprünge zu erstellen (Abschnitte 4.5 und 4.6), zu verschachteln (4.8) und zu vereinigen (4.10).

Das Verfahren in [FKCX94] kann zwar den Kontrollflussgraphen ändern, aber dabei werden keine neuen Sprünge (Entscheidungsknoten) erstellt, sondern nur vorhandene verschoben oder kopiert. In dieser Arbeit werden in Abschnitt 4.6 neue Sprünge mit den jeweils notwendigen Bedingungen erstellt, um teilweise nicht benötigten Code nur dann auszuführen, wenn er benötigt wird.

In [BG97] werden Prädikate verwendet, um partiell toten Code nur auszuführen, wenn er tatsächlich benötigt wird. Der letzte dort verwendete Schritt („branch deletion step“ [BG97, Seite 162]) kann Code duplizieren. Bei der Erstellung der Sprünge in dieser Arbeit (Abschnitte 4.5 und 4.6) wird kein Code dupliziert. Jedoch kann bei der Generierung des Codes zur Bedingungsprüfung (Abschnitt 4.14) Code mehrfach generiert werden. Da die effiziente Prüfung der Bedingungen in der Aufgabenstellung gefordert wird, wäre die Manipulation der Bedingungen auch bei der Verwendung des in [BG97] beschriebenen Verfahrens notwendig und damit auch die in Abschnitt 4.14 beschriebene Generierung des Codes zur Bedingungsprüfung.

2 Grundlagen

2.1 Zwischenrepräsentation

Zwischenrepräsentationen (englisch: intermediate representations, IRs oder auch intermediate languages) wurden früh in der Entwicklung von Compilern eingeführt, um die Komplexität des Compilierungsprozesses zu handhaben. Sie erlauben es, den Compiler in mehrere Phasen und Module aufzuspalten. [Cho13]

Eine Zwischenrepräsentation ist eine Datenstruktur, die ein Programm so repräsentiert, dass es später ohne Änderung des Verhaltens ausgeführt werden kann. Sie wird intern im Compiler genutzt und dient als Schnittstelle zwischen den einzelnen Komponenten und Phasen. Durch die interne Nutzung kann jeder Compiler eine eigene Zwischenrepräsentation definieren, die außerdem nur dem Compiler bekannt sein muss. Eine IR kann vorübergehend während der Compilierung eines Programms verwendet werden oder aber auch für die Ausgabe von Text bzw. Binärdaten. [Cho13]

2.2 Kontrollflussgraph

Der Kontrollflussgraph eines Programms ist ein gerichteter Graph. Seine Knoten sind Basic Blocks und seine Kanten stellen den Kontrollfluss des Programms (Sprünge) dar. Basic Blocks bestehen aus Anweisungen, die sequentiell abgearbeitet werden. Sie müssen nicht notwendigerweise maximal sein, das heißt, dass es auch legal ist, wenn eine Anweisungssequenz, die zu einem Basic Block zusammengefasst werden kann, in mehrere Basic Blocks aufgeteilt wird. Nachfolger von Knoten X sind alle Knoten Y, die eine Kante von X nach Y besitzen. Dementsprechend sind die Vorgänger von Knoten Y alle Knoten X, die eine Kante von X nach Y haben. [CFRWZ89, Seite 26]

2.3 Static Single Assignment Form (SSA)

In der SSA Form kann jeder Variablen im Quelltext nur einmal ein Wert zugewiesen werden. Für sequenzielle Programme kann diese Eigenschaft leicht hergestellt werden, indem bei jeder Zuweisung eines Wertes zu einer Variablen eine neue Version dieser Variable angelegt wird, das heißt, dass die Variable umbenannt wird. Entsprechend wird sie auch in nachfolgenden Lesezugriffen umbenannt. [WZ91, Seiten 188 – 190]

In Programmen mit nichtlinearem Kontrollfluss, zum Beispiel durch die Verwendung von If-Then-Else-Anweisungen oder von Schleifen, kann der Kontrollfluss verzweigen und sich später wieder vereinigen. An den „join nodes“ [WZ91, Seite 188] (Vereinigungsstellen) können verschiedene Versionen einer Variable zusammentreffen. Die „richtige“ Version wird durch eine φ-Funktion ausgewählt. Die φ-Funktion hat die Syntax

$$v_n = \varphi(v_1, v_2, v_3, \ldots)$$

wobei die Anzahl der Parameter der Anzahl der direkten Vorgänger im Kontrollflussgraph entspricht. Die Reihenfolge der Vorgänger im Kontrollflussgraph ist beliebig, aber fest, und entspricht der Reihenfolge der Parameter der φ-Funktion. Die Funktion wählt

den Parameter v_i aus, der dem Vorgänger entspricht, über den der aktuelle Knoten erreicht wurde, und weist dessen Wert der Variablen v_n zu. [WZ91, Seiten 188 – 190] [CFRWZ89, Seiten 26 – 27]

2.4 Datenflusssprachen

Der Überblick über die Datenflusssprachen, der in diesem Abschnitt gegeben wird, beginnt mit dem allgemeinen asynchronen Fall. SIG ist eine synchrone Datenflusssprache, wodurch Vereinfachungen gegenüber asynchronen Sprachen möglich sind. Der synchrone Fall und die dadurch möglichen Vereinfachungen und Einschränkungen werden am Ende des Abschnitts beschrieben.

Im Datenflussausführungsmodell werden Programme als gerichtete Graphen repräsentiert. Dabei repräsentieren die Knoten des Graphs Anweisungen und die Kanten Datenabhängigkeiten. Die Daten fließen als Token entlang der Kanten. Die Kanten agieren hierbei als FIFO-Warteschlangen mit unbegrenzter Kapazität. Im Vergleich mit Programmen für die Von-Neumann-Architektur können Kanten im Datenflussmodell als Variablen interpretiert werden. Jeder Knoten besitzt lokalen und es gibt keinen globalen Speicher. Die Knoten arbeiten funktional: Daten werden nie verändert, es existieren keine Seiteneffekte und durch die Abwesenheit von globalem Speicher sind die Effekte, die die Knoten verursachen, lokal begrenzt. [JHM04, Seiten 3 – 4]

Ein Knoten ist „fireable“ [JHM04, Seite 3] (kann ausgeführt werden), wenn eine bestimmte Menge der Eingangskanten eines Knotens, die „firing set“ [JHM04, Seite 3] genannt wird, Daten hat. Die genaue Ausführungszeit ist unbestimmt. Bei seiner Ausführung entfernt der Knoten die Datentoken von den Kanten im firing set und schreibt neue Datentoken auf einige oder alle seiner Ausgangskanten. Dadurch können Knoten bzw. Anweisungen ausgeführt werden, sobald alle nötigen Operanden verfügbar sind. Ein weiterer großer Vorteil ist, dass mehrere Knoten parallel ausgeführt werden können. Es ist sogar möglich, dass ein zweiter Datensatz bearbeitet wird, während das Datenflussprogramm noch den ersten Datensatz bearbeitet. Dies ist unter dem Namen „pipelined dataflow“ [JHM04, Seite 4] bekannt. Ein Datenflussprogramm, das für je eine Menge von Eingangstoken genau eine Menge von Ausgangstoken erzeugt, wird „well-behaved“ [JHM04, Seite 4] genannt. [JHM04, Seiten 3 – 4]

Alternativ zum oben beschriebenen Tokenmodell existiert das Strukturmodell („structure model“ [JHM04, Seite 5]). Im Strukturmodell baut jeder Knoten nur eine Datenstruktur auf den Kanten, die dort verbleibt. Die Strukturen können unendlich sein – und damit das Tokenmodell durch ein unendlich großes Array von Token simulieren. Das Strukturmodell ermöglicht dadurch einen wahlfreien Zugriff auf die gesamte Historie der erstellten Daten. Ein bedeutender Nachteil des Strukturmodells ist, dass Daten nicht effizient gespeichert werden können. Es wird in Datenflusssprachen selten verwendet – das Tokenmodell ist weiterhin das fast ausschließlich verwendete Modell. [JHM04, Seite 5]

Für die Realisierung einer Datenflusssprache gibt es zwei Vorschläge: den datengetriebenen („data-driven“ [JHM04, Seite 5]) und den bedarfsgetriebenen („demand-driven“ [JHM04, Seite 5]) Vorschlag. Im datengetriebenen Vorschlag gibt es eine übergeordnete Instanz, die die Knoten informiert, wenn sie fireable sind, und sie feuert. Im bedarfsge-

triebenen Vorschlag dagegen wartet ein Knoten auf eine Anfrage nach Daten von seinen Ausgangskanten. Erreicht eine Anfrage den Knoten, dann stellt er wiederum seinen Eingangskanten Anfragen nach Daten, feuert, wenn er alle Daten erhalten hat, und schreibt die generierten Daten auf seine Ausgangskanten. [JHM04, Seiten 5 – 6]

Die praktische Implementierung des Datenflussmodells ist schwierig. Das Modell geht von FIFO-Warteschlangen mit unbegrenzter Größe aus, was praktisch nicht realisierbar ist. Außerdem nimmt das Modell an, dass beliebig viele Knoten gleichzeitig ausgeführt werden können. In der Praxis ist jedoch die Anzahl der Prozessoren und damit die Anzahl der maximal gleichzeitig ausführbaren Knoten begrenzt. Lösungsmöglichkeiten sind die statische Architektur, die dynamische bzw. Tagged-Token Architektur und der synchrone Datenfluss. Bei der statischen Architektur kann jede Kante maximal ein Token aufnehmen. In der dynamische Architektur können Subgraphen aus konzeptioneller Sicht mehrfach aufgerufen werden. In der Realisierung ist der Graph jedoch nur einmal im Speicher vorhanden und die Zuordnung der Token zu den jeweiligen Aufrufen erfolgt mittels Tags. Der synchrone Datenfluss ist eine Teilmenge des reinen Datenflusses. In ihm ist bereits zur Compilezeit für jeden Knoten bekannt, wie viele Tokens auf welchen Kanten konsumiert bzw. geschrieben werden. Dadurch kann, im Gegensatz zur statischen oder dynamischen Architektur, das Datenflussprogramm statisch geplant und in ein sequenzielles Programm übersetzt werden. Neben dem Vorteil der statischen Planbarkeit existieren auch Einschränkungen. Das heißt, dass nicht alle Programme im synchronen Datenfluss repräsentiert werden können. Ein Beispiel für eine Einschränkung sind Schleifen. Im synchronen Datenfluss sind sie nur möglich, wenn die Anzahl der Durchläufe bereits zur Compilezeit bekannt ist. [JHM04, Seiten 6 – 9]

3 Die Datenflusssprache SIG

Die Sprache SIG arbeitet getaktet synchron, das heißt, dass die Berechnungen mit einer festen und bekannten Taktrate ausgeführt werden. Der Datenfluss ist dabei unverzüglich, mit der Ausnahme expliziter Verzögerung. Unverzüglicher Datenfluss muss zyklenfrei sein [WL14b, Seite 35]. Explizit verzögerter Datenfluss hingegen erlaubt Feedbackschleifen. Keine andere Form der Rekursion ist erlaubt. [WL14a, Seite 147]

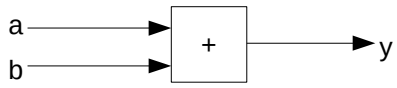

Abbildung 1: Addierer

Die Berechnungen des beispielsweise in Abbildung 1 dargestellten Addierers geschieht konzeptionell ohne Zeitverzögerung. Sobald Daten auf den Kanten a und b vorhanden sind, ist sofort das Ergebnis verfügbar. Die Planung der Berechnung erfolgt zur Compilezeit, da SIG eine synchrone Datenflusssprache ist.

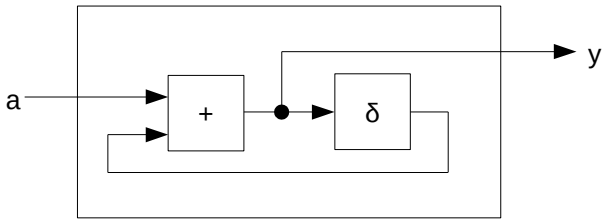

Abbildung 2: Integrierer – erlaubte Rückkopplung durch Verzögerung

Die in Abbildung 2 gezeigte Rückkopplung ist erlaubt, weil der δ-Knoten eine explizite Verzögerung ist. Der Wert des Ausgangs des δ-Knotens ist während eines Verarbeitungstaktes konstant. Damit ist auch die Berechnung eindeutig definiert.

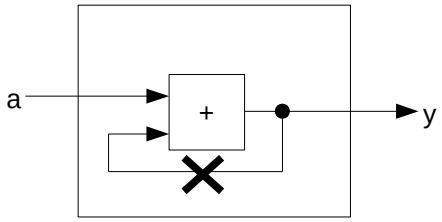

Abbildung 3: Unerlaubter Zyklus

Die Rückkopplung, die in Abbildung 3 dargestellt ist, ist unzulässig, weil zwischen dem Ausgang y und dem Eingang des Addierers keine Verzögerung vorhanden ist.

Die meisten Berechnungen sind zustandslos, wobei die Verzögerung eine bedeutende Ausnahme ist. Aus Gründen der Einfachheit gibt es in SIG nur Ein-Schritt-Verzögerung. Im Frontend existiert Zustand nur implizit durch Verzögerung. In der Zwischenrepräsentation von SIG dagegen ist er explizit. Die Stromsemantik wird durch unendliche Wiederholung eines Schritts erreicht, wobei der Nachzustand eines Schritts der Vorzustand des nächsten ist. [WL14b, Seite 35]

Ein Beispiel für eine zustandslose Berechnung ist der in Abbildung 1 dargestellte Addierer. Sein Ergebnis hängt nur von seinen Eingängen ab. Der δ-Knoten in Abbildung 2 dagegen enthält einen Zustand. Sein Ausgang nimmt den Wert seines Eingangs im vor-

herigen Verarbeitungstakt an. Die Auflösung der Ein-Schritt-Verzögerung eines δ-Knotens zu Zustand ist in Abbildung 4 dargestellt [WL14a, Seite 150].

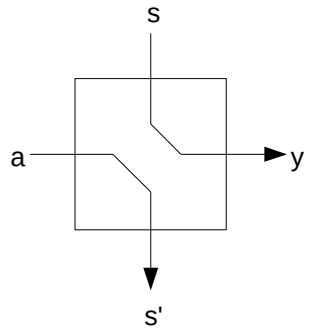

Abbildung 4: Auflösung der Ein-Schritt-Verzögerung zu Zustand

Die Variable `s` ist der Vor- und `s'` der Nachzustand.

Box-ähnliche Komponenten in SIG haben totale Streamfunktionen-Semantik. Gleichzeitig sind Programmfragmente komponierbar, einschließlich Pattern Matching, was von seiner Natur aus partiell ist. Innerhalb einer Box sind ⊥-Werte erlaubt, die anzeigen, dass ein Wert nicht definiert ist – ähnlich dem `null`-Wert in Java, der anzeigt, dass eine Referenz auf kein Objekt zeigt. An den Grenzen der Boxen sind sie dagegen verboten. Kontrollvariablen werden durch die Werte { ⊤ , ⊥ } repräsentiert. [WL14a, Seiten 152 – 154]

Die Komponierbarkeit wurde bereits beim Integrierer in Abbildung 2 verwendet: der Addierer und die Verzögerung wurden zu einer neuen Komponente zusammengesetzt. Durch die totale Streamfunktionen-Semantik ist dabei sichergestellt, dass der Eingang `a` der Komponente Integrierer immer definiert ist, das heiß, dass ⊥-Werte verboten sind. Aus dem gleichen Grund muss auch der Ausgang `y` des Integrierers immer definiert sein. In Abbildung 5 wird ein Beispiel für Pattern Matching und damit verbunden partiell undefinierte Ausgänge dargestellt.

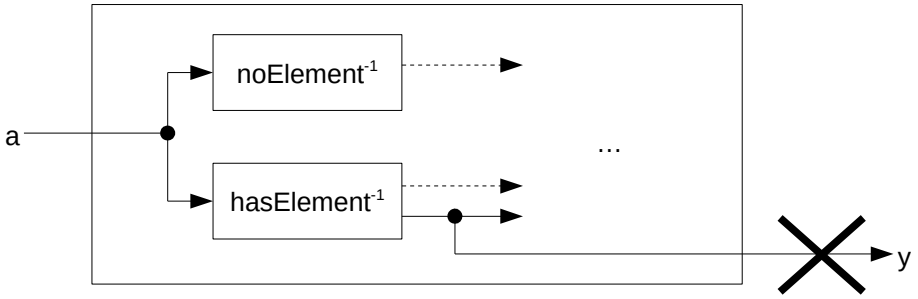

Abbildung 5: Beispiel für Pattern Matching und partiell undefinierte Ausgänge

Der Eingang `a` hat einen algebraischen Datentyp und soll entweder kein Element übertragen oder genau eines. Dazu hat der algebraische Datentyp zwei Konstruktoren: `noElement` , der kein Datenfeld besitzt, und `hasElement` , der genau ein Datenfeld besitzt. In den jeweiligen Boxen in Abbildung 5 wird Pattern Matching verwendet, um zu berechnen, welchen der beiden Fälle das aktuelle Datum von `a` entspricht. Die gestrichelten Pfeile sind Kontrollausgänge, die anzeigen ob der Mustervergleich erfolgreich

war. Der Datenausgang der Box `hasElement`$^{-1}$ ist undefiniert, wenn der entsprechende Mustervergleich fehlschlug. Deshalb ist der Ausgang `y` der Komponente verboten.

Die Interpretation des Datenflusses ist wie folgt: Verzögerungen werden zu Zustand aufgelöst. Primitive Operationen sind zustandslose Funktionen. Konstruktoren sind injektiv und haben paarweise disjunkte Wertebereiche. Weiterhin haben Konstruktoren partiell inverse Operationen, die Destruktoren. Destruktoren haben paarweise disjunkte Definitionsbereiche. Um die Partialität zu handhaben, besitzen sie neben den Datenausgängen einen Kontrollausgang, der die Definiertheit anzeigt. Kontrollvariablen können nicht im Frontend definiert werden, sondern entstehen ausschließlich durch Destruktoren. [WL14a, Seite 155]

Destruktoren wurden bereits in Abbildung 5 für das Pattern Matching gezeigt. Die zum Beispiel passenden Konstruktoren sind in Abbildung 6 dargestellt.

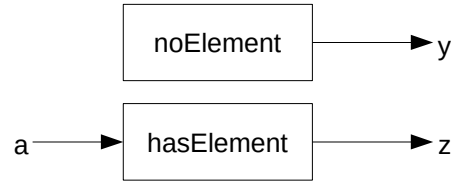

Abbildung 6: Konstruktoren für Beispiel in Abbildung 5

Durch die paarweise disjunkten Werte- bzw. Definitionsbereiche ist sichergestellt, dass für jeden Konstruktor höchstens ein passender Destruktor und für jeden Destruktor höchstens ein passender Konstruktor existiert.

Neben dem Datenfluss gibt es in Sig auch einen Kontrollfluss. Dafür werden die φ-Knoten der SSA Form, die die Zwischenrepräsentation von Sig ist, „missbraucht“. Die φ-Knoten wählen in Sig nichtdeterministisch aus allen definierten Eingängen einen Wert aus, der an den Ausgang weitergeleitet (kopiert) wird. Die Auswahl unterliegt der Bedingung, dass aus den Eingängen ausgewählt werden muss, die nicht den Wert $\bot$ besitzen. Haben alle Eingänge den Wert $\bot$, dann hat auch der Ausgang des φ-Knotens den Wert $\bot$. φ-Knoten führen damit alternative Zweige zusammen. Daneben existieren die Guard (γ) -Knoten. Ein γ-Knoten propagiert ein einzelnes Datum zu seinem Ausgang, genau dann wenn alle seine Kontrolleingänge den Wert $\top$ haben. Ansonsten ist der Ausgang $\bot$. Mit den γ-Knoten werden alternative Zweige bzw. deren Resultate ausgewählt. Für Kontrollwerte sind γ-Knoten Senken, Destruktoren ihre Quellen. Wenn alle Eingänge ausschließlich Kontrollwerte haben, agieren γ-Knoten als Konjunktionen und φ-Knoten als Disjunktionen. [WL14a, Seiten 155 – 156][WL14b, Seite 35]

In Abbildung 7 wird als Beispiel für den Kontrollfluss eine Sample & Hold Schaltung verwendet.[1]

1 Sample & Hold Schaltungen haben zwei Zustände: Im Sample-Zustand wird das Eingabedatum auf den Ausgang geleitet. Im Hold-Zustand wird die letzte Ausgabe wieder auf den Ausgang kopiert (der Ausgang wird gehalten). Das Umschalten zwischen den beiden Zuständen erfolgt über einen zweiten Eingang.

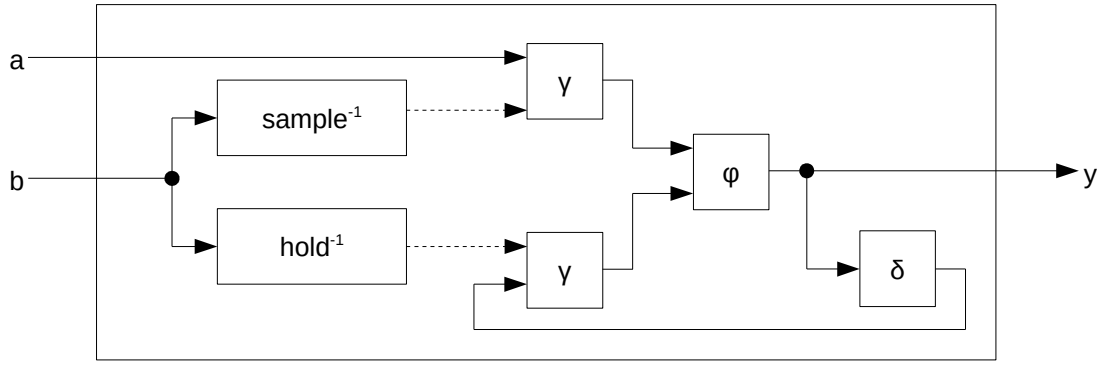

Abbildung 7: Sample & Hold Schaltung

In Abhängigkeit des Eingangs `b` propagieren die beiden γ-Knoten das jeweilige Datum auf ihren Ausgang oder setzten ihn auf ⊥. Der Eingang `b` hat einen algebraischen Datentyp mit genau zwei Konstruktoren: `sample` und `hold`. Durch die mittels des φ-Knotens erfolgte Auswahl ist der Wert des Ausgangs entweder der Wert des Eingangs `a` (sample) oder der vorherige Wert des Ausgangs `y` (hold).

4 Synthese von Kontrollfluss

Das Ziel dieses Kapitels ist die Synthese des Kontrollflusses. Die dazu eingesetzten Analysen und Transformationen werden beschrieben. Dabei wird nicht ein monolithisches Verfahren eingesetzt, sondern die Aufgabe wurde in voneinander abhängige Phasen bzw. Verfahren aufgeteilt. Die Abhängigkeiten der einzelnen Phasen sind in Abbildung 8 dargestellt.

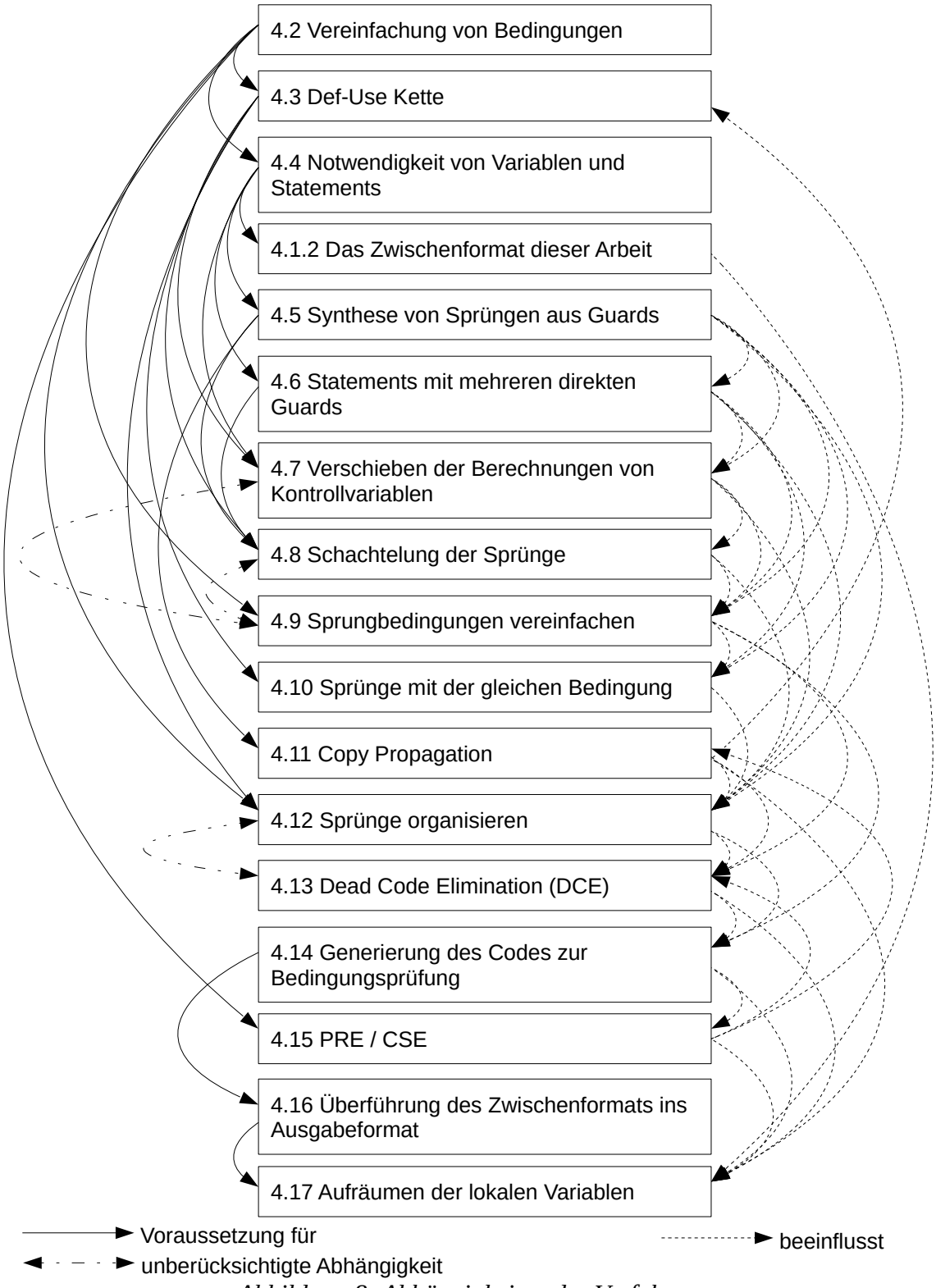

Abbildung 8: Abhängigkeiten der Verfahren

Die Verfahren 4.5 – 4.16 arbeiten auf dem in dieser Arbeit eingeführten Zwischenformat und sind deshalb von 4.1.2 abhängig. Die vielen zusätzlichen Pfeile, die zur Darstellung dieser Abhängigkeiten nötig wären, würden jedoch die Lesbarkeit der Abbildung stark

einschränken. Aus dem gleichen Grund wurde auf die Pfeile verzichtet, die anzeigen, dass die Verfahren 4.5 – 4.15 das Ergebnis der Transformation ins Ausgabeformat und das anschließende Aufräumen der lokalen Variablen (Abschnitte 4.16 und 4.17) beeinflussen.

Eine Phase A ist die Voraussetzung für Phase B, wenn B von A generierte Daten bzw. Strukturen benötigt. Dagegen wird B von A beeinflusst, wenn durch die Ausführung von A das Ergebnis der Berechnung von B geändert wird. Dabei kann B auch ausgeführt werden, ohne dass A vorher ausgeführt wurde, wobei der Sinn der Anwendung von B in diesem Fall fragwürdig sein kann.

Am Ende des jeweiligen Abschnitts werden die Abhängigkeiten kurz erläutert. Auf die unberücksichtigten Abhängigkeiten (4.7 ↔ 4.9, 4.8 ↔ 4.9 und 4.12 ↔ 4.13) wird im Ausblick (Abschnitt 7.2) eingegangen.

4.1 Zwischenformate

Im folgenden Abschnitt wird die Zwischenrepräsentation von SIG beschrieben, die für diese Arbeit das Eingabe- und Ausgabeformat ist. Neben diesem Zwischenformat wird in dieser Arbeit eine weitere Zwischenrepräsentation verwendet, die in Abschnitt 4.1.2 beschrieben wird.

4.1.1 Eingabe- und Ausgabeformat

Die Zwischenrepräsentation von SIG ist das Eingabe- und Ausgabeformat dieser Arbeit. Sie ist in SSA Form.

Jede Komponente in SIG ist ein Block. Jeder Block hat Listen für die verwendeten Variablen und verweist auf seine Basic Blocks, die für die Initialisierung und die Schleife zuständig sind. Ein Durchlauf der Schleife verarbeitet je ein Element der Eingabedatenströme.

Als primitive Datentypen stehen Boolean, Integer, Double und Void zur Verfügung. Daneben gibt es Funktionstypen und algebraische Datentypen. Funktionstypen haben das gleiche Verhalten wie normale Datentypen. Für die rekursive Definition von algebraischen Datentypen kann der Nominaltyp verwendet werden, der ein Synonym für einen algebraischen Datentyp ist. Dadurch wird eine endlose Rekursion vermieden. Als Beispiel soll die Definition einer Liste von Doubles dienen.

$$\texttt{List<Double>} := \begin{cases} \texttt{(Double, List<Double>)} \\ \texttt{nil} \end{cases}$$

Die Liste ist entweder ein Double-Element gefolgt von einer weiteren Liste oder die leere Liste `nil`. Damit wird jedoch die Liste mit sich selbst definiert. Mithilfe des Nominaltyps kann das vermieden werden.

$$\texttt{listOfDoubles} := \texttt{List<Double>}$$

$$\texttt{List<Double>} := \begin{cases} \texttt{(Double, listOfDoubles)} \\ \texttt{nil} \end{cases}$$

Der Typ `listOfDoubles` ist dabei ein Nominaltyp und synonym zu `List<Double>`.

Die Basic Blocks bestehen aus einer Liste von Statements, einer Referenz „next“ auf den nächsten Basic Block und einem optionalen Sprung. Die Referenzen auf Basic Blocks können auch auf `null` zeigen, was bedeutet, dass kein weiterer Basic Block existiert. Soll ein `null`-Basic Block ausgeführt werden, wird die Verarbeitung der aktuellen Elemente der Eingabedatenströme beendet und die Verarbeitung der nächsten Elemente begonnen. Sprünge bestehen aus einer booleschen Variablen und einer Referenz auf den Basic Block, zu dem gesprungen wird, wenn die Variable wahr ist. Ist die Variable falsch oder es existiert kein Sprung, wird der Basic Block abgearbeitet, auf den die next-Referenz zeigt. Sprünge bzw. Referenzen verhalten sich dabei wie GoTo-Befehle, das heißt, dass es keinen definierten Rücksprung gibt.

Die Statements bestehen aus einer Liste von Zielvariablen und einer Instruktion. Instruktionen besitzen eine Liste mit Quellvariablen. Weiterhin können sie, je nach Instruktion, weitere Daten enthalten. Konstrukt- und Destrukt-Instruktionen besitzen zum Beispiel noch einen Konstruktor. Ein weiteres Beispiel sind Konstanten-Instruktionen (laden eine Konstante), die den Wert der Konstanten enthalten.

Das folgende Listing 1 zeigt den Code für eine Sample & Hold Schaltung. Die Java-Datenstruktur wird dabei nicht als Java-Code ausgegeben, sondern in ein für Menschen leichter lesbares Format überführt.

```
1  AlgebraicDataType_0 {
2    hold: A()
3    sample: A()
4  }
5
6  Block_1 {
7    env:
8    in: double, AlgebraicDataType_0
9    out: double
10   state: double
11   locals: boolean, boolean, double, double
12   init: BasicBlock_2
13   loop: BasicBlock_3
14
15   BasicBlock_2 {
16     double Post_0 := DoubleConstant 0.0 ()
17   } next: --
18
19   BasicBlock_3 {
20     boolean Local_0 := Destruct "hold" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
21     double Local_2 := Guard  (double Pre_0, boolean Local_0)
22     boolean Local_1 := Destruct "sample" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
23     double Local_3 := Guard  (double Input_0, boolean Local_1)
24     double Output_0 := Choose  (double Local_2, double Local_3)
25     double Post_0 := Move  (double Output_0)
26   } next: --
27 }
```

Listing 1: Sample & Hold Schaltung

In den Zeilen 1 – 4 ist der verwendete algebraische Datentyp dargestellt. Die Zeilen 2 und 3 zeigen jeweils einen Datenfall. Die Zeichenkette vor dem Doppelpunkt ist der Name des Falls im konkreten algebraischen Datentyp, mit dem Konstruktoren ihn ansprechen können. Nach dem Doppelpunkt folgen der Name des Datenfalls und eine Liste der Typen der Felder, die in diesem Beispiel in beiden Datenfällen leer sind.

Die Zeilen 6 – 27 zeigen den Block, der die Schaltung realisiert. In den Zeilen 7 – 11 werden die verfügbaren Variablen deklariert. Genauer gesagt werden ihre Typen aufgezählt. Die Variablen in den Basic Blocks werden dann von 0 beginnend durchnummeriert. `Env`-Variablen (in den Statements des Programms als `Environment` dargestellt) stammen aus der Umgebung des Blocks. Sie können in dieser Arbeit als normale Eingabevariablen betrachtet werden. `in`-Variablen (`Input`) sind die Eingabevariablen. `out`-Variablen (`Output`) sind die Ausgabevariablen. `state` deklariert die Zustandsvariablen. Sie gibt es jeweils als Eingabe (Vorzustand, `Pre`) und als Ausgabe (Nachzustand, `Post`). Schließlich existieren noch die `local`-Variablen (`Local`), die lokale Variablen sind. In den Zeilen 12 und 13 sind die Referenzen auf den Initialisierungs- bzw. Schleifen-Basic Block dargestellt. Die Zeilen 16 und 20 – 25 zeigen die Statements der Basic Blocks. Statements haben das Format

```
Liste_der_Zielvariablen := Name_der_Instruktion
Daten_der_Instruktion (Liste_der_Quellvariablen)
```

Dabei ist die Anzahl der Quell- und Zielvariablen variabel (es wird kein Drei-Adress-Code verwendet).

Bei Instruktionen, die Konstanten laden (zum Beispiel in der Zeile 16), ist das Datum der Instruktion der Wert der Konstanten. Bei Konstrukt- und Destrukt-Instruktionen (zum Beispiel in den Zeilen 20 und 22) ist das Datum der Konstruktor, der wiederum aus dem Namen des Datenfalls und dem algebraischen Datentyp besteht. Die Instruktion der Guard (γ) -Knoten heißt `Guard`, die der φ-Knoten `Choose`.

Am Ende der Basic Blocks (Zeilen 17 und 26) sind die Referenzen auf die nächsten Basic Blocks dargestellt. Mit `--` wird angezeigt, dass es keine weiteren Basic Blocks gibt. Eine einfache Verkettung, also ohne den optionalen Sprung, wird wie in den Zeilen 12 und 13 dargestellt. Ein Sprung wird mit dem ternären Operator `?:` angezeigt. Das Format ist dabei

```
Bedingungsvariable ? Basic_Block_wenn_wahr :
Basic_Block_wenn_falsch
```

In der Eingabe bestehen der Initialisierungs- und der Schleifen-Teil jeweils nur aus einem einzigen Basic Block. Als Ausgabe können sie aus mehreren Basic Blocks bestehen, die über next-Referenzen und Sprünge miteinander verknüpft sind.

Die Synthese von Kontrollfluss läuft nacheinander auf dem Initialisierungs- und dem Schleifen-Basic Block ab. Deshalb wird im Weiteren nur noch ein Basic Block betrachtet.

4.1.2 Das Zwischenformat dieser Arbeit

Während der Synthese des Kontrollflusses wird das Programm in eine andere Datenstruktur überführt, die es einfach ermöglicht Statements zu verschieben und Sprünge und Labels einzufügen. Die Datenstruktur ist eine doppelt verkettete Liste. Die Listen werden, da sie den Programmcode enthalten, auch Programmlisten genannt. Labels sind die Ziele bei den Rücksprüngen. Sprünge enthalten die Sprungbedingung, das Rücksprunglabel und eine Programmliste (die Zielliste des Sprungs), die wiederum Statements, Sprünge und Labels enthalten kann.

Das folgende Listing 2 zeigt eine optimierte Form der Sample & Hold Schaltung aus Listing 1.

```
1  {
2    boolean Local_0 := Destruct "hold" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
3    label: Label_0
4    label: Label_1
5    { ((Local_0))
6      double Local_2 := Move  (double Pre_0)
7    } goto Label_2
8    { ((true))
9      double Local_3 := Move  (double Input_0)
10   } goto Label_2
11   label: Label_2
12   double Output_0 := Choose  (double Local_2, double Local_3)
13   double Post_0 := Move  (double Output_0)
14 }
```

Listing 2: Zwischenformat Programmliste von optimierter Sample & Hold Schaltung

Jede Programmliste beginnt mit `{` und endet mit `}` (zum Beispiel in den Zeilen 1, 5, 7 und 14). Sprünge haben, wie bereits beschrieben, eine Bedingung und ein Rücksprunglabel. Diese Bedingung wird nach dem Einleitenden `{` notiert (zum Beispiel in Zeile 5). Rücksprunglabel werden nach dem `}` mit dem Schlüsselwort `goto` gekennzeichnet (beispielsweise in Zeile 7). Labels werden, wie zum Beispiel in Zeile 11 dargestellt, mit dem Schlüsselwort `label:` gekennzeichnet.

Die Hauptliste ist die Programmliste, die nicht in einem Sprung gekapselt ist. Sie wird unbedingt ausgeführt und bildet den Einsprungpunkt und das Ende für das Programm. Die Programmlisten, die in Sprünge eingebettet sind, werden Ziellisten der Sprünge genannt, weil sie das Sprungziel sind. Die Ziellisten werden nur ausgeführt, wenn der Sprung erreicht wird und seine Bedingung zur Laufzeit erfüllt ist.

Semantisch wird eine solche Programmliste von oben nach unten abgearbeitet. Ein Sprung wird dabei nur ausgeführt, wenn seine Bedingung erfüllt ist, ansonsten wird mit der Zeile nach dem entsprechenden `}` fortgefahren. Wenn ein Sprung ausgeführt wird, werden die Statements der Zielliste des Sprungs ausgeführt und danach zum angegebenen Rücksprunglabel gesprungen. Die auf das Label folgenden Statements und Sprünge werden dann abgearbeitet.

Zur Überführung der Eingabe in die Programmliste wird jedes Statement des Basic Blocks, dessen Bedingung keine Kontradiktion ist, in die Liste eingetragen. Dazu werden die in Abschnitt 4.4 berechneten Informationen und die Regeln zur Vereinfachung von Bedingungen (4.2) benötigt.

4.2 Vereinfachung von Bedingungen

Neben der Analyse des Programms (Abschnitte 4.2.1 – 4.2.3) werden in diesem Abschnitt auch die Verfahren zur Vereinfachung boolescher Ausdrücke (Abschnitte 4.2.4 – 4.2.9) erläutert, die bei der Vereinfachung der Sprungbedingungen im Rahmen dieser Arbeit eingesetzt werden. Bei der Auswahl der Vereinfachungsverfahren musste berücksichtigt werden, dass Variablen nicht negiert werden können (siehe Abschnitt 4.2.3). Weiterhin erfolgte die Auswahl intuitiv.

Bedingungen sind in dieser Arbeit immer in der Disjunktiven Normalform (DNF). Konjunktionsterme sind dabei Mengen von Kontrollvariablen. Eine DNF ist eine Menge von Konjunktionstermen. Durch diese Darstellung einer DNF steht jede Variable innerhalb eines Konjunktionsterms. Das gilt auch, wenn der dadurch entstehende Konjunktionsterm nur aus einer einzigen Variablen besteht.

Die Reihenfolge der Vereinfachungsverfahren hat, zumindest bei den Verfahren 4.2.8 und 4.2.9, Einfluss auf das Ergebnis. Das kann leicht anhand der DNF

`((a ∧ c) ∨ (a ∧ b ∧ c) ∨ (a ∧ b ∧ ¬c))`

nachvollzogen werden. Je nach Reihenfolge sind folgende Ergebnisse möglich:

- `((a ∧ c) ∨ (a ∧ b))`
- `((a ∧ c) ∨ (a ∧ b ∧ ¬c))`

4.2.1 Konstante boolesche Variablen

Die Zwischenrepräsentation von SIG erlaubt es boolesche Variablen mit Konstanten zu laden. Diese Konstanten können bei der Vereinfachung von Bedingungen verwendet werden. Dazu werden alle booleschen Variablen, die mit booleschen Konstanten geladen werden, gesucht und in zwei Mengen eingeteilt: die Menge der Variablen, die immer wahr sind, und die Menge der Variablen, die immer falsch sind. Dabei werden nicht nur direkt geladene Variablen beachtet, sondern auch Variablen die das Ziel von Kopieranweisungen sind.

Bei der Implementierung werden diese zwei Mengen zusammen mit den exklusiven Variablenmengen (siehe Abschnitt 4.2.3) gespeichert. Jedes DNF-Objekt besitzt eine Referenz auf die exklusiven Variablenmengen. Dadurch können die Informationen bei jeder Entscheidung, ob die DNF eine Tautologie oder eine Kontradiktion ist, und bei der Vereinfachung verwendet werden.

4.2.2 Zurückrechnen boolescher Verknüpfungen

Mittels Guard- und Choose-Instruktionen können, wie bereits am Ende des Kapitels 3 erwähnt, die booleschen Verknüpfungen Konjunktion und Disjunktion realisiert werden. Zur Vereinfachung von Sprungbedingungen werden jedoch die im folgenden Abschnitt eingeführten exklusiven Variablenmengen benötigt. Die mittels Guard- und Choose-Anweisungen erzeugten booleschen Variablen können mit den exklusiven Variablenmengen jedoch nicht analysiert werden. Deshalb werden diese Variablen wieder in je eine DNF zurückgerechnet, die die Ausgangsvariablen enthält und äquivalent zur jeweiligen Variable ist. In einer Tabelle wird jeder booleschen Variable, die zurückgerechnet werden konnte, diese DNF zugeordnet. Variablen, die kein Schlüssel dieser Tabelle sind, werden implizit je eine DNF zugeordnet, die nur aus dieser Variablen besteht (Variable `a` → DNF `((a))`).

Das Zurückrechnen kann in der selben Schleife erfolgen wie die Verarbeitung konstanter boolescher Variablen im vorherigen Abschnitt. Es werden dabei nur Anweisungen betrachtet, die eine boolesche Variable als Ziel haben. Das Zurückrechnen folgt folgenden drei Regeln:

- Die DNF der Zielvariablen einer Move-Anweisungen ist die gleiche DNF wie die der Quellvariablen.
- Die DNF der Zielvariablen einer Guard-Anweisung ist die konjunktive Verknüpfung der DNFs der Quellvariablen.
- Die DNF der Zielvariablen einer Choose-Anweisung ist die disjunktive Verknüpfung der DNFs der Quellvariablen.

Das Zurückrechnen der booleschen Verknüpfungen ist nicht notwendig für die Korrektheit der Vereinfachung der Bedingungen, kann aber Optimierungspotenzial eröffnen. Das ist möglich, weil von (nicht konstanten) Variablen, die in einer exklusiven Variablenmenge enthalten sind, mehr Informationen bekannt sind als von (nicht konstanten) Variablen, die nicht in einer solchen Menge enthalten sind.

4.2.3 Exklusive Variablenmengen

Kontrollvariablen können in SIG nur durch Destruktoren, Konstanten, Guard- und Choose-Instruktionen erstellt werden. Dadurch ist es nicht möglich Variablen zu negieren. Trotzdem können sich Variablen gegenseitig ausschließen. In Listing 1 werden zum Beispiel die Kontrollvariablen `Local_0` und `Local_1` durch Anwendung von Destruktoren aus der Variablen `Input_1` erstellt. `Input_1` kann jedoch nicht gleichzeitig „sample“ und „hold“ sein. Daraus folgt, dass die Variablen `Local_0` und `Local_1` nicht gleichzeitig wahr sein können. Sie schließen sich gegenseitig aus. Mengen von Variablen, in denen höchstens eine Variable wahr sein kann, werden exklusive Variablenmengen genannt.

Exklusive Variablenmengen, deren disjunktive Verknüpfung ihrer Variablen den Wahrheitswert wahr ergeben, sind vollständig. Die exklusive Variablenmenge { `Local_0`, `Local_1` } im Beispiel in Listing 1 ist vollständig, weil beide Variablen durch Dekonstruktion mit verschiedenen Destruktoren aus der selben Variablen erzeugt wurden (Zeilen 20 und 22) und der algebraische Datentyp dieser Variablen genau zwei Fälle enthält. Das ist jedoch noch nicht ausreichend. `Environment`- und `Input`-Variablen sind immer definiert, das heißt, dass ihr Wert niemals $\perp$ ist. `Pre`-Variablen dagegen können auch den Wert $\perp$ besitzen [WL14a, Seite 153]. Nach der Semantik von SIG sind alle Zielvariablen einer Anweisung, die eine Quellvariable mit dem Wert $\perp$ hat, selbst undefiniert, mit zwei Ausnahmen: Choose- und Start-Anweisungen. Angenommen in Listing 1 würden die Destruktoren auf einer `Pre`-Variablen arbeiten, die undefiniert ist. Dann hätten die Variablen `Local_0` und `Local_1` beide den Wahrheitswert falsch und somit wäre die exklusive Variablenmenge nicht mehr vollständig.

Zur Entscheidung, ob eine exklusive Variablenmenge vollständig ist, reicht deshalb die Betrachtung des algebraischen Datentyps und der Destruktoren nicht aus. Zusätzlich muss darauf geachtet werden, dass die Quellvariable, auf der die Destruktoren arbeiten, auch immer definiert ist (niemals den Wert $\perp$ annimmt). Dazu wird für jede Variable die Bedingung gespeichert, unter der sie definiert sein muss. Die Initialisierung der `Environment`-, `Input`- und `Pre`-Variablen folgt direkt aus dem vorherigen Absatz.

Die Berechnung, unter welchen Bedingungen andere Variablen auf jeden Fall definiert sind, ergibt sich aus den folgenden Regeln:

- Bei einem Choose-Statement ist die Zielvariable definiert, wenn mindestens eine der Quellvariablen definiert ist. Deshalb ist die Bedingung für die Zielvariable die disjunktive Verknüpfung der Bedingungen der Quellvariablen.
- Bei einem Destruct-Statement zeigt die Kontrollvariable an, ob die Destruktion der Quellvariablen erfolgreich war. Damit sind die Kontrollvariable[2] und die anderen Zielvariablen, die die Werte der Felder des algebraischen Datentyps enthalten, definiert, wenn die Kontrollvariable definiert ist.
- Eine Start-Instruktion hat aus technischen Gründen die gleichen formalen Eingabeparameter wie die zugehörige Sample-Anweisung. Es wird jedoch nur der erste Eingabeparameter gelesen. Deshalb können alle Eingabeparameter eines Start-Statements außer dem ersten bei der Analyse der Definiertheit der Zielvariablen ignoriert werden. Die Zielvariablen eines Start-Statements sind damit immer definiert, wenn die erste Quellvariable immer definiert ist.
- Wenn das Statement eine andere als die oben genannten Instruktionen besitzt, sind die Zielvariablen definiert, wenn alle Quellvariablen definiert sind. Die Bedingung der Zielvariablen ist demnach die konjunktive Verknüpfung aller Bedingungen der Quellvariablen.

Es ist wichtig, dass bis jetzt die Bedingungen *nicht* vereinfacht wurden. Die Verfahren zur Vereinfachung (genauer gesagt das Verfahren in Abschnitt 4.2.8) benutzen vollständige exklusive Variablenmengen. Die Entscheidung, ob eine exklusive Variablenmenge vollständig ist oder nicht, ist jedoch noch nicht berechnet.

Jede Datenvariable der Destruktoren erzeugt genau eine exklusive Variablenmenge. Damit eine exklusive Variablenmenge vollständig ist, muss die Datenvariable immer definiert und die Anzahl der angewandten Destruktoren gleich der Anzahl der Fälle des algebraischen Datentyps sein. Bei der zweiten Bedingung ist die Gleichheit der Anzahlen der Destruktoren und der Fälle des algebraischen Datentyps ausreichend, weil davon ausgegangen wird, dass bereits CSE bzw. PRE (siehe Abschnitt 4.15) angewendet wurde und damit keine redundanten Berechnungen (insbesondere Destruktoren) auftreten. Zur Berechnung der Vollständigkeit der exklusiven Variablenmengen werden alle Datenvariablen in eine Warteschlange eingetragen. In einer Schleife wird die Warteschlange abgearbeitet. Zuerst wird geprüft, ob für jede Variable in der Bedingung der Definiertheit der Datenvariable bekannt ist, ob ihre exklusive Variablenmenge vollständig ist oder nicht. Wenn es nicht für jede Kontrollvariable bekannt ist, wird die Datenvariable an das Ende der Warteschlange eingefügt und mit dem nächsten Schleifendurchlauf fortgefahren. Ist die Vollständigkeit der exklusiven Variablenmenge jeder Kontrollvariablen bekannt, kann die Bedingung der Definiertheit der Datenvariable vereinfacht werden (siehe dazu die folgenden Abschnitte). Wenn die Bedingung dann eine Tautologie ist und die Anzahl der Destruktoren gleich der Anzahl der Fälle des algebraischen Datentyps der Datenvariable ist, ist die zur Datenvariable gehörende exklusive Variablenmenge vollständig, ansonsten nicht.

2 $\top \triangleq$ wahr; $\bot \triangleq$ falsch (siehe Kapitel 3)

Der gerade beschriebene Algorithmus behandelt den Init- und Loop-Teil eines Blocks getrennt. Es wird immer davon ausgegangen, dass `Pre`-Variablen den Wert ⊥ annehmen können. Im konkreten Programm ist es auch möglich, dass durch die Initialisierung und eine günstige Logik im Loop-Teil `Pre`-Variablen im Loop-Teil niemals den Wert ⊥ annehmen können. Diese Information lässt sich durch eine Abwandlung des beschriebenen Algorithmus berechnen. Bei der Bearbeitung des Init-Teils ändert sich der Algorithmus nicht, es wird nur am Ende gespeichert, welche Zustandsvariablen nach der Initialisierung den Wert ⊥ annehmen können und welche nicht. Bei der Bearbeitung des Loop-Teils wird die Initialisierung der `Pre`-Variablen angepasst. Sie sind immer definiert, wenn sie nicht den Wert ⊥ annehmen können. Anschließend wird der beschriebene Algorithmus ausgeführt. Danach muss die Annahme über die Definiertheit der Zustandsvariablen geprüft werden. Wurde eine `Pre`-Variable als immer definiert angenommen, aber die entsprechende `Post`-Variable ist nicht immer definiert, dann muss auch die `Pre`-Variable als nicht immer definiert angesehen werden. Der Algorithmus wird so lange iteriert, bis die Annahme der Definiertheit der Zustandsvariablen stabil ist. Die exklusiven Variablenmengen hängen nur von der Struktur der Destruktoren ab. Die Definiertheit der Variablen hat nur Einfluss auf die Vollständigkeit der exklusiven Variablenmengen. Weiterhin kann die Vollständigkeit Einfluss auf die Definiertheit von Variablen haben. Deshalb ist die Berechnung der exklusiven Variablenmengen nur einmal notwendig. Dagegen muss die Berechnung der Definiertheit der Variablen und der Vollständigkeit der exklusiven Variablenmengen iteriert werden.

In diesem Kapitel wird nur in diesem Absatz die Information benötigt, ob eine Variable immer definiert ist. In den anderen Abschnitten dieses Kapitels ist es ausreichend zu wissen, ob die Variable geschrieben wurde oder nicht – genau das ist dort gemeint, wenn von definierten Variablen gesprochen wird. Eingabevariablen (`Environment` , `Input` und `Pre`) zählen dabei auch zu den geschriebenen Variablen. Die bei der Vereinfachung von Bedingungen zusätzlich nötigen Informationen über die Definiertheit von Variablen sind bereits in den exklusiven Variablenmengen gespeichert.

4.2.4 Idempotenz

Die Idempotenz erlaubt folgende Vereinfachungen [K3BA, Seite 11]:

```
a ∧ a = a
a ∨ a = a
```

Durch die Verwendung von Mengen zur Speicherung von Konjunktionstermen und Disjunktionen wird diese Vereinfachung automatisch durchgeführt.

4.2.5 Neutrales Element

Wahr ist das neutrale Element der Konjunktion [K3BA, Seite 7]. Deshalb sind leere Konjunktionsterme Tautologien. Aus dem gleichen Grund können immer wahre Variablen aus einem Konjunktionsterm entfernt werden.

Das neutrale Element der Disjunktion ist falsch [K3BA, Seite 7]. Deswegen sind leere Disjunktionen Kontradiktionen. Außerdem können aus diesem Grund Konjunktionsterme, die Kontradiktionen sind, aus einer Disjunktion entfernt werden.

4.2.6 Wahre und falsche Variablen

Konjunktionsterme, die ausschließlich aus Variablen bestehen, die immer den Wahrheitswert wahr besitzen, sind Tautologien. Dagegen sind Konjunktionsterme, die mindestens eine Variable enthalten, die immer den Wahrheitswert falsch hat, Kontradiktionen.

4.2.7 Variable und ihre „Negation“ innerhalb eines Konjunktionsterms

Ein Konjunktionsterm ist immer falsch, wenn er mindestens zwei Variablen enthält, die in der gleichen exklusiven Variablenmenge enthalten sind. Sei zum Beispiel die exklusive Variablenmenge { `a` , `b` } gegeben. Der Konjunktionsterm `(a ∧ b)` ist dann eine Kontradiktion, weil niemals `a` und `b` gleichzeitig wahr werden können.

Da wahr das neutrale Element der Konjunktion ist, ist es ohne Veränderung des Wertes möglich, immer wahre Variablen zu einem Konjunktionsterm hinzuzufügen. Damit kann auch erkannt werden, dass ein Konjunktionsterm eine Kontradiktion ist, wenn die zweite Variable einer exklusiven Variablenmenge nicht explizit im Konjunktionsterm vorkommt, sondern sich implizit aus dem Kontext ergibt. Zur Veranschaulichung dient das Beispiel in Listing 3.

```
1 ...
2 { ((Local_0 AND Local_2) OR (Local_0 AND Local_3))
3   ...
4   { ((Local_1 AND Local_4))
5     ...
6   } goto Label_X
7   ...
8 } goto Label_Y
9 ...
```

Listing 3: Beispiel für Bedingung mit einer Variablen aus dem Kontext

Der Sprung in Zeile 2 wird nur ausgeführt, wenn die Variable `Local_0` wahr ist. Deshalb kann die Sprungbedingung in Zeile 4 ohne Änderung ihres Wertes zu

`((Local_1 AND Local_4 AND Local_0))`

erweitert werden. Unter der Annahme, dass die exklusive Variablenmenge { `Local_0` , `Local_1` } existiert, ist die erweiterte Bedingung eine Kontradiktion. Da diese Bedingung und die Sprungbedingung in Zeile 4 äquivalent sind, ist auch die Sprungbedingung eine Kontradiktion.

4.2.8 „Negation“ einer einzigen Variablen

Wenn sich zwei Konjunktionsterme, die durch Disjunktion verknüpft sind, nur durch die Negation einer einzigen Variablen unterscheiden, kann diese eine Variable aus den beiden Konjunktionstermen entfernt werden. Aus den zwei gleich großen Konjunktionstermen wird dabei ein einziger Konjunktionsterm, der um eine Variable kürzer ist. [McC56, Seite 1418]

```
(a ∧ b) ∨ (a ∧ ¬b) = a
```

Da es keine negierten Variablen gibt, sondern nur exklusive Variablenmengen, wird von ihnen die Aufgabe der negierten Variablen übernommen. Das heißt auch, dass im Allgemeinen nicht nur zwei Konjunktionsterme benötigt werden, sondern so viele, wie die verwendete (vollständige) exklusive Variablenmenge groß ist.

Zur Veranschaulichung des Algorithmus wird als Beispiel die DNF

```
((a ∧ b) ∨ (a ∧ c) ∨ (d ∧ e ∧ f) ∨ (d ∧ e ∧ g) ∨ (h ∧ i)
∨ (h ∧ j))
```

mit den vollständigen exklusiven Variablenmengen { `b` , `c` } und { `f` , `g` } verwendet.

Der Algorithmus, der diese Vereinfachung berechnet, arbeitet wie folgt: Zuerst werden alle Konjunktionsterme in Buckets einsortiert, die der Anzahl der Variablen jedes Konjunktionsterms entsprechen.

```
2: {a, b}, {a, c}, {h, i}, {h, j}
3: {d, e, f}, {d, e, g}
```

Als nächstes wird für jedes Paar von Konjunktionstermen in einem Bucket die Schnittmenge berechnet. Wenn die Schnittmenge genau ein Element kleiner ist, als die beiden Ausgangskonjunktionsterme, wurden potenzielle Kandidaten für die Vereinfachung gefunden. Die Schnittmenge ist der Schlüssel für zwei weitere Buckets: in den einen werden die potenziell „negierten“ Variablen (Variablen aus einer exklusiven Variablenmenge) einsortiert, in den anderen die verwendeten Konjunktionsterme.

Schlüssel	potenziell negierte Variablen	verwendete Konjunktionsterme
`{a}`	`{b, c}`	`{a, b}, {a, c}`
`{h}`	`{i, j}`	`{h, i}, {h, j}`
`{d, e}`	`{f, g}`	`{d, e, f}, {d, e, g}`

Als letztes wird für jeden Schlüssel geprüft, ob die potenziell „negierten“ Variablen eine Obermenge einer vollständigen exklusiven Variablenmenge sind. Wenn sie es sind, werden die zum Schlüssel gehörenden verwendeten Konjunktionsterme aus der DNF entfernt und der Schlüssel eingetragen. Es ist unschädlich, wenn in der Menge der potenziell „negierten“ Variablen mehr Variablen als in der entsprechenden exklusiven Variablenmenge sind. Die zusätzlich entfernten Konjunktionsterme wären mit dem Absorptionsgesetz (siehe folgenden Abschnitt) auch entfernt worden.

```
resultierende DNF: ((a) ∨ (d ∧ e) ∨ (h ∧ i) ∨ (h ∧ j))
```

Dieses Verfahren wird so lange wiederholt, bis die DNF nicht mehr geändert wird.

4.2.9 Absorptionsgesetz

Das Absorptionsgesetz erlaubt es Konjunktionsterme zu entfernen, die eine Obermenge eines anderen Konjunktionsterms sind [K3BA, Seite 11].

`(a) ∨ (a ∧ b) = a`

4.3 Def-Use Kette

Eine Definition (kurz Def) einer Variablen ist eine Anweisung, die dieser Variablen einen Wert zuweist (sie schreibt). Das Lesen einer Variablen wird Use (Benutzung) genannt. Use-Def und Def-Use Ketten verbinden die Definitionen der Variablen mit den entsprechenden Uses. [KSK09, Seiten 25, 29 – 30]

Die Def-Use Ketten, die im Rahmen dieser Arbeit aufgebaut werden, enthalten für jede Variable ihre durch die SSA Eigenschaft eindeutige Definition und alle Uses. Sie können damit als kombinierte Def-Use und Use-Def Ketten angesehen werden.

Dadurch, dass in der SSA Form jede Variable nur einmal definiert wird, kann die Def-Use Kette jeder Variablen leicht aus ihr ausgelesen werden. Für jede Zielvariable eines Statements ist das Statement die Definition der jeweiligen Variablen. Die Eingabevariablen haben dabei als Definition den Wert `null`. Weiterhin ist das Statement für jede seiner Quellvariablen ein Use. Eine Ausnahme existiert dabei im Kontext dieser Arbeit bei den Guards. Sie werden in Abschnitt 4.5 umgeschrieben. Deshalb ist das Guard-Statement kein Use für seine Kontrollvariablen. Stattdessen ist es ein Use für alle Variablen, die in den DNFs der Kontrollvariablen (siehe Abschnitt 4.2.2) vorkommen. Um effizient auf die Definitionen und Uses zugreifen zu können, werden sie in Hashtabellen gespeichert.

Aus diesen Informationen lassen sich die Vorgänger und Nachfolger eines Statements berechnen. Die Vorgänger eines Statements sind alle Definitionen, ausgenommen `null`-Werte, der Quellvariablen des jeweiligen Statements. Analog dazu sind die Nachfolger eines Statements alle Uses der Zielvariablen des jeweiligen Statements.

4.4 Notwendigkeit von Variablen und Statements

Variablen und Statements werden nicht immer benötigt. Guards können den Wert einer Variablen an ihren Ausgang weiterleiten oder auch nicht. Wenn der Wert nicht weitergeleitet wird, muss er (wenn er nicht woanders benötigt wird) nicht berechnet werden. Damit ist es auch nicht notwendig, die Statements, die zur Berechnung des Werts dienen, auszuführen. Beispielsweise muss das Statement in Zeile 1 im folgenden Listing 4 nur ausgeführt werden, wenn die durch … angedeuteten Bedingungen des Guards in Zeile 2 wahr sind. In diesem Schritt soll berechnet werden, unter welchen Bedingungen welche Variablen (bzw. dessen Werte) und Statements notwendig sind. Dabei werden auch die Guards gespeichert, die die Berechnung der Anweisung unmittelbar durchlassen oder sperren.

Im folgenden Beispiel (Listing 4) beeinflusst der Guard in Zeile 2 das Statement in Zeile 1 unmittelbar: der Guard ist ein direkter Guard für das Statement. Der Guard in Zeile

3 beeinflusst das Statement nur indirekt dadurch, dass er den Guard in Zeile 2 beeinflusst, der wiederum das Statement beeinflusst.

```
1 double Local_0 := Call "f" (double Input_3)
2 double Local_1 := Guard  (double Local_0, …)
3 double Local_2 := Guard  (double Local_1, …)
```

Listing 4: Beispiel direkter Guard

Alle Ausgabevariablen werden immer benötigt. Das heißt, dass ihre Bedingung `true` ist und sie nur einen direkten Guard, `null` , besitzen.

4.4.1 Algorithmus

Die Statements werden mit dem letzten beginnend rückwärts durchlaufen. Ein Statement, das keine Guard-Instruktion enthält, wird dann benötigt, wenn eine der Zielvariablen benötigt wird. Die Bedingung des Statements ist deshalb die disjunktive Verknüpfung aller Bedingungen der Zielvariablen. Die direkten Guards des Statements sind alle direkten Guards der Zielvariablen. Die Quellvariablen werden benötigt, wenn sie von diesem Statement oder von einem anderen benötigt werden. Deshalb werden ihre Bedingungen disjunktiv mit der Bedingung dieses Statements verknüpft. Weiterhin wird zur Menge der direkten Guards jeder einzelnen Quellvariablen alle direkten Guards des Statements hinzugefügt.

Ein Statement, das eine Guard-Instruktion enthält, muss anders behandelt werden. Später werden Guard-Statements in einen Sprung und eine Move-Anweisung (die Weiterleitung des einzelnen Datums auf den Ausgang) aufgespalten. Die Guard-Bedingung ist die Bedingung, unter der der Wert der Datenvariable zur Zielvariablen kopiert wird. Sie wird durch die konjunktive Verknüpfung der DNFs der Kontrollvariablen[3] des Guards berechnet. Die Variablen in der Guard-Bedingung werden immer benötigt, wenn das Guard-Statement im Programm erreicht wird, also getestet wird, ob der Sprung ausgeführt werden muss. Die Bedingung der Variablen ist daher, wie bei den anderen Statements, die disjunktive Verknüpfung der bisherigen Bedingungen mit der disjunktiven Verknüpfung der Bedingungen der Zielvariablen. Die direkten Guards der Variablen der Guard-Bedingung und des Guard-Statements werden ebenso wie bei den anderen Statements berechnet: die Mengen der direkten Guards der Zielvariablen werden vereinigt und zu den jeweiligen Mengen der Variablen hinzugefügt. Hinsichtlich seiner Bedingung wird das Guard-Statement selbst wie der Move-Teil, in den es zerlegt wird, behandelt. Die Bedingung ist die disjunktive Verknüpfung aller Bedingungen der Zielvariablen, die zusätzlich konjunktiv mit der Guard-Bedingung verknüpft wird. Die Datenvariable hat die gleiche Bedingung wie das Guard-Statement. Ihr direkter Guard ist (abgesehen von bereits vorhandenen direkten Guards) nur das Guard-Statement selbst.

Wenn ein Statement als direkten Guard den Pseudo-Guard `null` hat, kann die Menge der direkten Guards durch die Menge { `null` } ersetzt werden, weil das Statement in jedem Fall ausgeführt werden muss. Der Pseudo-Guard `null` wird nur bei der Initialisierung der Ausgabevariablen eingeführt. Die Ausgabevariablen müssen immer berechnet werden. Weiterhin muss es zwischen der entsprechenden Ausgabevariablen und dem be-

3 Zur Erläuterung von DNFs einer booleschen Variable siehe Abschnitt 4.2.2.

trachteten Statement bzw. der betrachteten Variablen einen Pfad ohne Guard geben, weil sonst der Pseudo-Guard `null` durch den Guard ersetzt worden wäre. Dementsprechend muss die Bedingung des Statements bzw. der Variablen eine Tautologie sein.

4.4.2 Abhängigkeiten

Für die Vereinfachungen der Bedingungen müssen die in Abschnitt 4.2 beschriebenen Verfahren verwendet werden. Es ist jedoch nicht zwingend Notwendig, die Bedingungen bereits in diesem Schritt zu vereinfachen. Die Abhängigkeit ist deshalb von der Implementierung abhängig.

Im Rahmen dieser Arbeit werden die Bedingungen erst später vereinfacht. Trotzdem werden in diesem Schritt die in den Abschnitten 4.2.1 – 4.2.3 berechneten Informationen benötigt, weil sie in den DNFs gespeichert und damit bei dessen Erstellung benötigt werden.

4.5 Synthese von Sprüngen aus Guards

Die Ausgangssituation für diesen Schritt ist, dass die Werte der Variablen immer berechnet werden – auch wenn ein Guard sie später verwirft. In diesem Abschnitt werden Sprünge aus Guards synthetisiert, so dass unnötige Berechnungen (zum Teil, weitere notwendige Schritte folgen in späteren Abschnitten) vermieden werden.

Beispielhaft ist eine Ausgangssituation in Listing 5 dargestellt. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 44 gezeigt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
5    boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
6    double Local_4 := Call "f1" (double Input_2)
7    double Local_5 := Call "f2" (double Local_4)
8    double Local_6 := Call "f3" (double Local_4)
9    double Local_7 := Guard  (double Local_5, boolean Local_0)
10   double Local_8 := Guard  (double Local_6, boolean Local_1)
11   double Local_9 := Choose  (double Local_7, double Local_8)
12   double Local_10 := Call "f4" (double Input_2)
13   double Local_11 := Guard  (double Local_9, boolean Local_2)
14   double Local_12 := Guard  (double Local_10, boolean Local_3)
15   double Output_0 := Choose  (double Local_11, double Local_12)
16 }
```

Listing 5: Alle Anweisungen werden ausgeführt

Alle Anweisungen werden ausgeführt, obwohl nur eine der Berechnungen `f2(f1(Input_2))`, `f3(f1(Input_2))` oder `f4(Input_2)` verwendet wird. Auch die Berechnungen von `Local_0` und `Local_1` sind nicht in jedem Fall notwendig.

Listing 5 zeigt dasselbe Programm nach der Anwendung des in diesem Abschnitt beschriebenen Verfahrens.

```
1  {
2    boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
3    boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
4    { ((Local_2))
5      boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
6      boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
7      double Local_4 := Call "f1" (double Input_2)
8      { ((Local_2 AND Local_0))
9        double Local_5 := Call "f2" (double Local_4)
10       double Local_7 := Move  (double Local_5)
11     } goto Label_0
12     label: Label_0
13     { ((Local_2 AND Local_1))
14       double Local_6 := Call "f3" (double Local_4)
15       double Local_8 := Move  (double Local_6)
16     } goto Label_1
17     label: Label_1
18     double Local_9 := Choose  (double Local_7, double Local_8)
19     double Local_11 := Move  (double Local_9)
20   } goto Label_2
21   label: Label_2
22   { ((Local_3))
23     double Local_10 := Call "f4" (double Input_2)
24     double Local_12 := Move  (double Local_10)
25   } goto Label_3
26   label: Label_3
27   double Output_0 := Choose  (double Local_11, double Local_12)
28 }
```

Listing 6: Sprünge überspringen fallweise nicht benötigte Anweisungen

Durch die Einführung von Sprüngen werden die Bedingungen für die Notwendigkeit der Aufrufe von `f2`, `f3` und `f4` vor deren Ausführung geprüft. Auch die Berechnungen der Variablen `Local_0` und `Local_1` geschehen nur noch, wenn sie benötigt werden. Die Anweisung in Zeile 7 ist zwar von mehreren direkten Guards abhängig, aber durch die im folgenden Abschnitt beschriebene Manipulation der direkten Guards kann auch sie in den Sprung in Zeile 4 geschachtelt werden.

4.5.1 Algorithmus

Die Programmliste wird von vorne beginnend durchlaufen. Sobald ein Statement mit einer Guard-Instruktion gefunden wird, wird die Guard-Instruktion durch eine Move-Instruktion ersetzt, so dass das Statement jetzt die Datenvariable des Guards zur Zielvariablen kopiert. Weiterhin wird gespeichert, dass dieses Statement ein Guard-Statement war. Das so geänderte Statement wird nun vor die Position des gefunden Statements in die Liste kopiert. Die Bedingung der Kopie ist die Bedingung des Guards. Der einzige direkte Guard des kopierten Statements ist das gefundene Statement selbst. Damit exis-

tiert das Statement doppelt in der Liste. Da später das Statement an der aktuellen Position gelöscht wird, ist die Doppelung unschädlich und sogar notwendig. Später werden die bereits errechneten Def-Use Ketten verwendet. Ein neue erstelltes Statement würde darin nicht vorkommen – im Gegensatz zum vorhandenen (und kopierten) Statement. Allerdings muss bei der späteren Verwendung der Def-Use Ketten darauf geachtet werden, dass das Statement die Kontrollvariablen nicht mehr verwendet.

Als nächstes wird die Programmliste beginnend mit dem gerade kopierten Statement (also mit dem Element genau vor der aktuellen Position) rückwärts durchlaufen. Dabei werden alle Elemente (Statements, Sprünge und Labels) aus der Liste entfernt und in eine neue Liste gespeichert, deren Menge direkter Guards nur aus dem gefundenen Statement besteht. Dabei stimmt die Reihenfolge der Statements in der neuen Liste mit der Reihenfolge in der durchlaufenen Liste überein.

Beim rückwärts Durchlaufen der Programmliste können auch Elemente gefunden werden, die das gefundene Statement als direkten Guard besitzen, aber auch noch andere Guard-Statements. Dann wird das gefundene Statement aus der Menge der direkten Guards entfernt und die Menge der direkten Guards des gefundenen Statements hinzugefügt.

Nach dem Erreichen des Anfangs der Programmliste und damit dem Fertigstellen der neuen Liste wird ein Sprung erstellt, dessen Zielliste die neu erstellte Liste ist. Die Bedingungen und die direkten Guards sind die gleichen wie die des gefundenen Statements. Das Label, das das Rücksprungziel des Sprungs kennzeichnet, erhält die gleichen direkten Guards wie der Sprung. Zusätzlich werden für jeden Sprung seine „natürlichen Guards“, die Menge der Guards, die seine Erstellung veranlasst haben, gespeichert. Das ist in diesem Fall das gefundene Statement. Der Sprung wird direkt über dem gefundenen Statement eingefügt. Das Label wird danach auch direkt über dem gefundenen Statement eingefügt. Damit folgt das Label direkt auf den Sprung.

Als letztes wird das gefundene Statement entfernt und beginnend mit dem nächsten nach weiteren Guard-Statements gesucht. Das wird so lange fortgesetzt, bis das Ende der Liste erreicht wurde.

Die Elemente, deren direkter Guard nur aus dem gefundenen Guard-Statement besteht, können eingesammelt werden, weil ihre Berechnungen nur vom gefundenen Guard-Statement benötigt werden. Würden auch andere Berechnungspfade sie benötigen, würden diese Pfade durch entsprechende Guards in der Menge der direkten Guards repräsentiert werden. Es kann also keine Statements außerhalb der eingesammelten Elemente geben, die die Berechnungen bzw. die definierten Variablen der eingesammelten Statements benötigen – abgesehen vom gefundenen Guard-Statement. Da jedoch der Sprung und das Label direkt vor dem Guard-Statement platziert werden, sind alle nötigen Daten dort verfügbar. Da die relative Reihenfolge der Elemente beim Einsammeln nicht verändert wird und sie nur weiter nach unten verschoben werden, sind auch alle für die Statements nötigen Daten bzw. Variablen verfügbar – durch die SSA Eigenschaft können einmal definierte Variablen innerhalb des Basic Blocks nicht überschrieben werden.

Dadurch, dass der Sprung und das Label die direkten Guards des gefundenen Guard-Statements erhalten, können sie bei einem weiteren Schleifendurchlauf, der das entsprechende Guard-Statement findet, weiter verschachtelt werden.

Bei mehr als einem direktem Guard werden die Berechnungen bzw. die durch sie definierten Variablen in mehreren Berechnungspfaden benötigt. Damit muss auch die Position der Sprünge und der Labels so bestimmt werden, dass die definierten Variablen für alle Berechnungspfade rechtzeitig zur Verfügung stehen. Mit dem oben beschriebenen Verfahren ist dies nicht möglich, weshalb diese Elemente vorerst nicht weiter beachtet werden. Allerdings führt die beschriebene Manipulation der Menge der direkten Guards dazu, dass die Elemente doch noch bearbeitet werden können, wenn sich die Berechnungspfade später vereinigen.

4.5.2 Abhängigkeiten

Der Algorithmus benutzt die direkten Guards der Statements, die in Abschnitt 4.4 berechnet wurden.

4.6 Statements mit mehreren direkten Guards

Statements mit mehreren direkten Guards werden benötigt, wenn die Bedingung mindestens einer ihrer direkten Guards wahr ist. Das heißt, dass sie nicht in jedem Fall, aber auch nicht niemals, benötigt werden.

In Listing 7 ist ein Beispiel dargestellt, in dem eine nicht in jedem Fall benötigte Anweisung unbedingt ausgeführt wird. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 45 gezeigt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_6 := Destruct "C" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
5    double Local_8 := Call "f4" (double Input_1)
6    { ((Local_0))
7      double Local_2 := Call "f1" (double Local_8)
8      double Local_4 := Move  (double Local_2)
9    } goto Label_0
10   label: Label_0
11   { ((Local_1))
12     double Local_3 := Call "f2" (double Local_8)
13     double Local_5 := Move  (double Local_3)
14   } goto Label_1
15   label: Label_1
16   { ((Local_6))
17     double Local_7 := Call "f3" (double Input_1)
18     double Local_9 := Move  (double Local_7)
19   } goto Label_2
20   label: Label_2
21   double Output_0 := Choose  (double Local_4, double Local_5, double
   Local_9)
22 }
```

Listing 7: Anweisung, die nicht in jedem Fall benötigt wird (mehrere direkte Guards)

Das Ergebnis der Anweisung in Zeile 5 wird nur von den Anweisungen in den Zeilen 7 und 12 verwendet. Mit dem Sprung in Zeile 16 existiert ein Kontrollflusspfad, in dem die Anweisung in Zeile 5 nicht benötigt wird. Trotzdem wird die Anweisung unbedingt ausgeführt.

In Listing 8 ist die Situation nach der Anwendung des in diesem Abschnitt beschriebenen Verfahrens dargestellt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_6 := Destruct "C" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
5    { ((Local_1) OR (Local_0))
6      double Local_8 := Call "f4" (double Input_1)
7    } goto Label_3
8    label: Label_3
9    { ((Local_0))
10     double Local_2 := Call "f1" (double Local_8)
11     double Local_4 := Move  (double Local_2)
12   } goto Label_0
13   label: Label_0
14   { ((Local_1))
15     double Local_3 := Call "f2" (double Local_8)
16     double Local_5 := Move  (double Local_3)
17   } goto Label_1
18   label: Label_1
19   { ((Local_6))
20     double Local_7 := Call "f3" (double Input_1)
21     double Local_9 := Move  (double Local_7)
22   } goto Label_2
23   label: Label_2
24   double Output_0 := Choose  (double Local_4, double Local_5,
   double Local_9)
25 }
```

Listing 8: Teilweise nicht benötigte Anweisung wird übersprungen

Der Aufruf von `f4` wird nur dann ausgeführt, wenn das Ergebnis in den Zeilen 10 oder 15 verwendet wird. Das wird durch den Sprung in Zeile 5 erreicht.

4.6.1 Algorithmus

Im vorherigen Abschnitt wurden die Mengen der direkten Guards manipuliert. In diesem Abschnitt werden sie jedoch in ihrer Originalform benötigt. Deshalb müssen zuerst die Mengen der direkten Guards wiederhergestellt werden. Dabei ist zu beachten, dass der direkte Guard eines Statements, das vorher ein Guard-Statement war, das Statement selbst ist.

Damit die partiell toten Statements nicht unnötig ausgeführt werden, wird ihre Ausführung durch bedingte Sprünge geschützt. Dazu werden alle Elemente, die in einer Programmliste stehen und die gleiche Menge direkter Guards haben, gesucht, in einer neu-

en Liste zusammengestellt, ein Sprung erstellt und dieser Sprung an die Stelle des letzten gefundenen Statements platziert. Die natürlichen Guards der erstellten Sprünge sind die direkten Guards der Elemente, die in die jeweilige Zielliste eingetragen werden. Mit diesem Vorgehen werden Elemente ausschließlich weiter nach hinten in der Liste verschoben. Deshalb sind alle Daten, die sie für ihre Ausführung benötigen, vorhanden. Es bleibt noch zu argumentieren, dass kein anderes Element Daten der verschobenen Elemente `E*` benötigt, bevor sie an ihrer neuen Position ausgeführt werden. Alle Elemente `E*` werden in der gleichen relativen Reihenfolge in die neue Liste eingefügt, mit der sie in der verarbeiteten Programmliste vorkamen. Innerhalb der verschobenen Anweisungen bleiben die Datenabhängigkeiten somit unberührt. Wenn ein Element `e1` mit einer anderen Menge direkter Guards Daten der verschobenen Elemente `E*` benötigt (also ein Nachfolger dieser Elemente ist), muss die Menge direkter Guards dieses Elements `e1` eine Teilmenge der Menge der direkten Guards der verschobenen Elemente `E*` sein. Das folgt aus der Tatsache, dass wenn ein Statement benötigt wird, auch alle seine Vorgänger benötigt werden. Da die Menge der direkten Guards eines Statements aus der Vereinigung der Mengen direkter Guards der Nachfolger gebildet wird, muss das letzte Vorkommen eines Elementes mit der gleichen Menge direkter Guards wie Element `e1` nach dem letzten Vorkommen der Elemente von `E*` auftreten. Damit wird das Element `e1` mit diesem Verfahren hinter das letzte Vorkommen eines Elements aus `E*` geschoben und kann damit auf die in `E*` berechneten Daten zugreifen.

In jeder Liste kann es Statements bzw. Elemente geben, die in jedem Fall ausgeführt werden müssen. Dies sind die Statements, die beim Erstellen der Liste auch ohne Manipulation der direkten Guards in die Liste aufgenommen wurden. Bei einem Sprung sind das die Elemente, deren einziger direkter Guard der Guard ist, der die Sprungbedingung bestimmt. Außerhalb von Sprüngen sind das die Elemente, die den Pseudo-Guard `null` besitzen. Diese Guards wurden im vorherigen Abschnitt als natürliche Guards zu jedem Sprung gespeichert. Ein Sprung zu diesen Elementen würde zur Laufzeit immer ausgeführt werden und ist damit unnötig. Um ihn zu vermeiden werden Elemente, die alle natürlichen Guards des Sprunges, der die aktuelle Liste als Ziel hat, enthalten nicht verschoben. Die nicht verschobenen Elemente können keines der verschobenen Elemente als Vorgänger besitzen, weil sonst der Vorgänger ebenfalls alle natürlichen Guards als direkte Guards besitzen und somit auch nicht verschoben werden würde.

4.6.2 Abhängigkeiten

Für das Wiederherstellen der direkten Guards werden die in Abschnitt 4.4 berechneten Informationen benötigt.

Außerdem wird dieser Algorithmus vom Algorithmus in Abschnitt 4.5 beeinflusst, weil der Algorithmus aus Abschnitt 4.5 bereits Sprünge erzeugt und damit den Kontrollfluss ändert. Dieser Algorithmus wird erst nach 4.5 ausgeführt, weil 4.5 Sprünge verschachteln kann, dieser Algorithmus jedoch nicht.

4.7 Verschieben der Berechnungen von Kontrollvariablen

Durch das bisherige Vorgehen kann es vorkommen, dass in den Bedingungen der Sprünge Kontrollvariablen vorkommen, die noch nicht definiert sind. Eine solche Situation ist beispielhaft in Listing 9 dargestellt. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 46 gezeigt.

```
1  {
2    { ((Local_1) OR (Local_0))
3      double Local_8 := Call "f4" (double Input_1)
4    } goto Label_3
5    label: Label_3
6    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
7    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
8    boolean Local_6 := Destruct "C" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
9    { ((Local_0))
10     double Local_2 := Call "f1" (double Local_8)
11     double Local_4 := Move  (double Local_2)
12   } goto Label_0
13   label: Label_0
14   { ((Local_1))
15     double Local_3 := Call "f2" (double Local_8)
16     double Local_5 := Move  (double Local_3)
17   } goto Label_1
18   label: Label_1
19   { ((Local_6))
20     double Local_7 := Call "f3" (double Input_1)
21     double Local_9 := Move  (double Local_7)
22   } goto Label_2
23   label: Label_2
24   double Output_0 := Choose  (double Local_4, double Local_5, double
   Local_9)
25 }
```

Listing 9: Sprungbedingung mit undefinierten Kontrollvariablen

In Listing 9 wird die gleiche Funktion wie in Listing 8 berechnet – nur die Reihenfolge der Anweisungen ist unterschiedlich. Zu beachten ist, dass beide Reihenfolgen legal sind. Der Sprung in Zeile 2 benutzt die Kontrollvariablen `Local_0` und `Local_1`, die erst später in den Zeilen 6 und 7 definiert werden. Für die Korrektheit ist das unschädlich, weil später in Abschnitt 4.9 auf diese Situation getestet und sie entsprechend behandelt wird.[4] Allerdings können die Bedingungen besser behandelt werden, wenn alle Kontrollvariablen definiert sind. Das Ziel dieses Abschnitts ist deshalb, die Berechnung der Kontrollvariablen soweit möglich vor die erste Bedingung, die die Kontrollvariable benutzt, zu verschieben. Die Anweisungen werden nur innerhalb einer Programmliste verschoben.

4 Nicht definierte Kontrollvariablen werden durch wahr ersetzt, damit benötigter Code in jedem Fall, in dem er benötigt wird, auch ausgeführt wird.

In Listing 10 ist die Situation nach der Anwendung des in diesem Abschnitt beschriebenen Verfahrens dargestellt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    { ((Local_1) OR (Local_0))
5      double Local_8 := Call "f4" (double Input_1)
6    } goto Label_3
7    label: Label_3
8    boolean Local_6 := Destruct "C" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
9    { ((Local_0))
10     double Local_2 := Call "f1" (double Local_8)
11     double Local_4 := Move  (double Local_2)
12   } goto Label_0
13   label: Label_0
14   { ((Local_1))
15     double Local_3 := Call "f2" (double Local_8)
16     double Local_5 := Move  (double Local_3)
17   } goto Label_1
18   label: Label_1
19   { ((Local_6))
20     double Local_7 := Call "f3" (double Input_1)
21     double Local_9 := Move  (double Local_7)
22   } goto Label_2
23   label: Label_2
24   double Output_0 := Choose  (double Local_4, double Local_5, double
   Local_9)
25 }
```

Listing 10: Berechnung der Kontrollvariablen vor Sprungbedingung

Die Berechnungen der Variablen `Local_0` und `Local_1` wurden vor den Sprung, der sich jetzt in Zeile 4 befindet, verschoben. Dadurch sind alle in der Sprungbedingung auftretenden Kontrollvariablen definiert.

4.7.1 Algorithmus

Anders als bisher kann nicht allgemein argumentiert werden, dass die Codeverschiebung sicher ist, das heißt, dass alle Datenabhängigkeiten eingehalten werden. Deshalb werden die Def-Use Ketten verwendet, um beim Verschieben einer Anweisung nach vorne in der Programmliste alle Vorgänger mit zu verschieben. Die Unstimmigkeit in den Def-Use Ketten, dass die Move-Statements, die vorher Guards waren, auch die Variablen in den DNFs ihre Kontrollvariablen noch benötigen, ist hier von Vorteil. Vorkommen von Variablen in Bedingungen werden nicht als Uses gezählt – noch nicht definierte Kontrollvariablen führen nicht zu Fehlern. Dadurch, dass die Def-Use Ketten anzeigen, dass die ehemaligen Guard-Statements immer noch die Kontrollvariablen benötigen, kann es nicht passieren, dass ein Sprung, der aufgrund eines Guard-Statements erstellt wurde (siehe Abschnitt 4.5), vor die Definition einer seiner Kontrollvariablen ge-

schoben wird. Weiterhin werden die vor der Abarbeitung der Liste bereits definierten Variablen benötigt. Für die Hauptliste sind das die Eingabevariablen.

Der Algorithmus bearbeitet zuerst die Hauptliste, danach rekursiv die Ziellisten der Sprünge. Zuerst werden die in der Liste auftretenden Kontrollvariablen gesucht. Das sind die Variablen, die in den Bedingungen der Sprünge in dieser Liste auftreten, aber auch in den Sprüngen innerhalb der Sprünge. Das heißt, dass die auftretenden Kontrollvariablen rekursiv ohne Beschränkung der Rekursionstiefe gesucht werden. Danach werden alle bereits definierten Variablen aus der Menge der Kontrollvariablen entfernt. Da die Variablen bereits definiert sind, können alle Sprungbedingungen in der Liste auf die Variablen zugreifen, was das Ziel dieses Abschnitts ist.

Als nächstes folgt eine Schleife, die über die verbleibenden Kontrollvariablen iteriert. In dieser Schleife werden zuerst die Definition der Variablen und ihre erste Verwendung in einer Sprungbedingung gesucht. Wenn die Variable definiert wurde, bevor sie zum ersten Mal in einer Sprungbedingung vorkommt, oder sie in dieser Liste nicht definiert wird, ist für diese Kontrollvariable nichts mehr zu tun – es kann mit der nächsten fortgefahren werden. Das Vorkommen in einer Sprungbedingung wird wieder rekursiv berechnet, es werden also auch die Sprünge innerhalb des Sprungs betrachtet. Ansonsten wurde eine Kontrollvariable gefunden, die definiert wird, nachdem ein Sprung sie benutzt.

Die Definition dieser Variablen wird jetzt nach vorne geschoben. Dazu wird die Menge der benötigten Anweisungen erstellt. Sie wird mit der Definition der Kontrollvariablen initialisiert. Die Liste wird beginnend an der Stelle der Definition der Variablen rückwärts durchlaufen. Jedes dabei gefundene Statement, das in der Menge der benötigten Anweisungen ist, und jeder Sprung, inklusive dem dazugehörigen Label, der ein solches Statement (wieder rekursiv) enthält, wird in einer zweiten Liste zwischengespeichert und aus der Programmliste gelöscht. Dabei bleibt die relative Reihenfolge der Anweisungen erhalten. Bei jedem Hinzufügen eines Statements bzw. eines Sprungs zur zweiten Liste werden die Vorgänger zur Menge der benötigten Anweisungen hinzugefügt. Die Vorgänger eines Sprungs sind alle Vorgänger der Statements und Sprünge in der Liste des Sprungs (ohne die Definitionen der Kontrollvariablen der Sprungbedingungen). Das Durchlaufen der Liste endet, wenn der erste Sprung, der die Kontrollvariable benötigt, erreicht und verarbeitet wurde. Die in der zweiten Liste eingesammelten Anweisungen werden direkt nach dem Element in der Liste eingefügt, das vor dem ersten Sprung steht bzw. stand. Wenn kein solches Element existiert, stand der erste Sprung am Anfang der Liste und die Elemente werden am Anfang der Liste eingefügt.

Nachdem über alle verbleibenden Kontrollvariablen iteriert wurde, muss der Algorithmus noch auf die Ziellisten der Sprünge in der Programmliste angewendet werden. Dazu wird die Programmliste vom Anfang beginnend durchlaufen. Bei jedem gefundenen Statement werden die Zielvariablen zu den definierten Variablen hinzugefügt. Wenn ein Sprung gefunden wird, wird der Algorithmus auf die Zielliste des Sprungs angewendet, wobei als definierte Variablen die aktuelle Menge der definierten Variablen übergeben wird. Danach werden zur aktuellen Menge der definierten Variablen die im Sprung definierten Variablen hinzugefügt.

Nach der Anwendung dieses Verfahrens ist es sinnvoll, die Bedingungen der Sprünge auch als Use der Variablen anzusehen, damit bei der Vereinfachung oder Auswertung

der Bedingungen möglichst viele ihrer Kontrollvariablen verfügbar sind. Falls die Bedingungen als Use angesehen werden, können an dieser Stelle die Def-Use Ketten korrigiert werden, das heißt, dass die Kontrollvariablen, die bei der Umwandlung der Guards in Move-Anweisungen (Abschnitt 4.5) nicht mehr als von der Move-Anweisung verwendet angesehen werden. Das hat den Vorteil, dass in nachfolgenden Verfahren eventuell durch Vereinfachung der Sprungbedingungen entfallene Variablen Codeverschiebungen nicht mehr blockieren können. In dieser Arbeit werden in den folgenden Verfahren die Sprungbedingungen immer als Use der Kontrollvariablen gezählt und die Def-Use Ketten an dieser Stelle korrigiert.

4.7.2 Abhängigkeiten

Die Berechnung der Vorgänger eines Statements wird mithilfe der Def-Use Ketten (Abschnitt 4.3) durchgeführt. Da die Variablen in den Sprungbedingungen eine zentrale Rolle in diesem Algorithmus spielen, beeinflussen alle Verfahren, die Sprünge erstellen, diesen Algorithmus. Das sind die in den Abschnitten 4.5 und 4.6 beschriebenen Verfahren.

Für die Korrektur der Def-Use Ketten wird weiterhin die Menge der ehemaligen Guard-Statements, die in Move-Statements umgewandelt wurden (Abschnitt 4.5), benötigt. Nur bei diesen Statements wurden Quellvariablen entfernt. Alle anderen Statements besitzen noch ihre ursprünglichen Quell- und Zielvariablen.

4.8 Schachtelung der Sprünge

In Abschnitt 4.6 wurden Sprünge erstellt ohne zu prüfen, ob sich die Sprünge verschachteln lassen.

In Listing 10, dem Ergebnis des Beispiels des letzten Abschnitts, ist eine Situation zu sehen, in der eine Schachtelung möglich ist und die Anzahl der auszuwertenden Bedingungen reduzieren kann. Die beiden Sprünge in den Zeilen 9 und 14 können in den Sprung in Zeile 4 geschachtelt werden. Das ist möglich, weil die Bedingung von `4` erfüllt sein muss, damit die Bedingungen von `9` oder `14` erfüllt sein können.

In Listing 11 ist die Situation nach der Anwendung des in diesem Abschnitt beschriebenen Verfahrens dargestellt.

```
1  {
2    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    { ((Local_1) OR (Local_0))
5      double Local_8 := Call "f4" (double Input_1)
6      { ((Local_0))
7        double Local_2 := Call "f1" (double Local_8)
8        double Local_4 := Move  (double Local_2)
9      } goto Label_0
10     label: Label_0
11     { ((Local_1))
12       double Local_3 := Call "f2" (double Local_8)
13       double Local_5 := Move  (double Local_3)
14     } goto Label_1
15     label: Label_1
16   } goto Label_3
17   label: Label_3
18   boolean Local_6 := Destruct "C" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
19   { ((Local_6))
20     double Local_7 := Call "f3" (double Input_1)
21     double Local_9 := Move  (double Local_7)
22   } goto Label_2
23   label: Label_2
24   double Output_0 := Choose  (double Local_4, double Local_5, double
   Local_9)
25 }
```

Listing 11: Sprünge wurden geschachtelt

4.8.1 Algorithmus

Der in diesem Abschnitt beschriebene Algorithmus benutzt dabei jedoch nicht die Bedingungen, sondern wie auch Abschnitt 4.6 die Guards. Im Gegensatz zu 4.6 werden hier jedoch die natürlichen Guards verwendet. Sie bieten den Vorteil, dass zum Treffen von Entscheidungen nur geprüft werden muss, ob eine Menge eine Teilmenge einer anderen ist anstatt Implikationen von booleschen Ausdrücken auswerten zu müssen. Eine Menge von Guards entspricht dabei der disjunktiven Verknüpfung der Bedingungen, unter der die Guards das Eingabedatum auf den Ausgang kopieren. Wie bereits in Abschnitt 4.6 argumentiert, befinden sich die Sprünge mit den allgemeineren Bedingungen immer oberhalb der Sprünge mit den spezielleren. Ziel des Algorithmus ist es, die Sprünge mit den spezielleren Bedingungen in die mit den allgemeineren zu schachteln. Da dabei Code nach vorne verschoben wird, muss darauf geachtet werden, dass keine Datenabhängigkeit die Codeverschiebung blockiert.

Zur Kontrolle der Datenabhängigkeiten werden die Statements und die Vorgänger der Statements bzw. die Definitionen der gelesenen Variablen innerhalb der Sprünge benötigt. Zu den gelesenen Variablen zählen hier auch die Variablen innerhalb der Sprungbedingungen. Die Berechnung der Statements und der Vorgänger der Sprünge muss rekursiv erfolgen. Der hier beschriebene Algorithmus arbeitet mit Breitensuche. Dadurch werden die Statements und Vorgänger der inneren Sprünge mehrfach benötigt. Die Da-

ten lassen sich jedoch zwischenspeichern. Zwar werden die Sprünge verändert, durch die Traversierungsreihenfolge werden jedoch die (veralteten) Daten der veränderten Sprünge nicht mehr benötigt, wenn ein Sprung verändert wurde.

Der Algorithmus läuft wie folgt ab. Die Hauptliste wird in eine Warteschlange eingetragen. Darauf folgt eine Schleife, die so lange eine Programmliste aus der Warteschlange entnimmt, wie diese nicht leer ist.

Alle Elemente der Programmliste werden in einer weiteren Schleife durchlaufen. Dabei sind nur Sprünge interessant, alle anderen Elemente werden übersprungen. Beim finden eines Sprungs `S` wird seine Zielliste in die Warteschlange eingetragen. Weiterhin werden in einer weiteren Schleife alle Elemente der Programmliste beginnend mit dem auf dem gefunden Sprung `S` folgenden Element durchlaufen.

In dieser inneren Schleife werden die Sprünge gesucht, die in den gefundenen Sprung `S` geschachtelt werden können. Zur effizienten Prüfung der bereits erwähnten Datenflussabhängigkeiten wird eine Menge der Statements verwaltet, die auf dem Weg von `S` (exklusive) bis zur aktuellen Position in der inneren Schleife aufgetreten sind. Beim finden eines Statements wird dieses zur Menge hinzugefügt. Statements von Sprüngen werden nur hinzugefügt, wenn der entsprechende Sprung nicht in `S` geschachtelt wird, weil sie mit dem Schachteln in `S` nicht mehr auf dem Weg von `S` (exklusive) zu einem weiter hinten in der Programmliste stehenden Element liegen können. Labels werden bei der Berechnung der auftretenden Statements ignoriert, weil zur Zeit noch jedes Label genau einem Sprung zugeordnet ist und das Label direkt auf dem Sprung folgt. Ein gefundener Sprung `S'` wird in `S` geschachtelt, wenn die natürlichen Guards von `S'` eine Teilmenge der natürlichen Guards von `S` sind und keiner der Vorgänger bzw. der Definitionen der gelesenen Variablen von `S'` in der Menge der auftretenden Statements von `S` zu `S'` (jeweils exklusive) vorkommen. `S'` wird in `S` geschachtelt heißt, dass `S'` und sein Label an das Ende der Zielliste von `S` angehangen werden.

4.8.2 Abhängigkeiten

Die natürlichen Guards werden bei der Erstellung der Sprünge berechnet. Deshalb sind die Abschnitte 4.5 und 4.6 Voraussetzungen für diesen Algorithmus. Außerdem werden die Definitionen von Variablen und die Vorgänger von Statements benutzt. Sie werden mittels der in Abschnitt 4.3 beschriebenen Def-Use Ketten berechnet.

Weiterhin beeinflusst die Verschiebung der Definitionen der Kontrollvariablen (Abschnitt 4.7) diesen Algorithmus, weil dadurch das Verschieben von Sprüngen ermöglicht oder verhindert werden kann. Dieser Algorithmus wird nach 4.7 ausgeführt, damit vermieden wird, dass durch die Verschiebung eines Sprungs weniger Kontrollvariablen in den Sprungbedingungen definiert sind.

4.9 Sprungbedingungen vereinfachen

Die Informationen, welche Variablen definiert sind und welche Sprünge zum Erreichen eines bestimmten Pfades bereits genommen wurden, können genutzt werden, um die Sprungbedingungen zu vereinfachen.

In Listing 12 sind Sprünge mit Bedingungen dargestellt, die vereinfacht werden können. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 47 gezeigt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_6 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_2)
5    boolean Local_10 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_2)
6    { ((Local_0 AND Local_6) OR (Local_0 AND Local_10))
7      double Local_8 := Call "f4" (double Input_1)
8      { ((Local_0 AND Local_6))
9        double Local_2 := Call "f1" (double Local_8)
10       double Local_4 := Move  (double Local_2)
11     } goto Label_0
12     label: Label_0
13     { ((Local_0 AND Local_10))
14       double Local_3 := Call "f2" (double Local_8)
15       double Local_5 := Move  (double Local_3)
16     } goto Label_1
17     label: Label_1
18   } goto Label_3
19   label: Label_3
20   { ((Local_1))
21     double Local_7 := Call "f3" (double Input_1)
22     double Local_9 := Move  (double Local_7)
23   } goto Label_2
24   label: Label_2
25   double Output_0 := Choose  (double Local_4, double Local_5, double
   Local_9)
26 }
```

Listing 12: Sprungbedingungen können vereinfacht werden

Die Variablen `Local_6` und `Local_10` bilden eine vollständige exklusive Variablenmenge. Deshalb kann die Bedingung in Zeile 6 vereinfacht werden. Weiterhin können die Sprünge in den Zeilen 8 und 13 nur erreicht werden, wenn der Sprung in Zeile 6 ausgeführt wurde. Aus diesem Grund ist die Variable `Local_0` immer wahr, wenn die Sprünge in den Zeilen 8 und 13 erreicht werden. Das erlaubt die Vereinfachung der Bedingungen dieser Sprünge.

In Listing 13 ist die Situation nach der Anwendung des in diesem Abschnitt beschriebenen Verfahrens dargestellt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_6 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_2)
5    boolean Local_10 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_2)
6    { ((Local_0))
7      double Local_8 := Call "f4" (double Input_1)
8      { ((Local_6))
9        double Local_2 := Call "f1" (double Local_8)
10       double Local_4 := Move  (double Local_2)
11     } goto Label_0
12     label: Label_0
13     { ((Local_10))
14       double Local_3 := Call "f2" (double Local_8)
15       double Local_5 := Move  (double Local_3)
16     } goto Label_1
17     label: Label_1
18   } goto Label_3
19   label: Label_3
20   { ((Local_1))
21     double Local_7 := Call "f3" (double Input_1)
22     double Local_9 := Move  (double Local_7)
23   } goto Label_2
24   label: Label_2
25   double Output_0 := Choose  (double Local_4, double Local_5, double
   Local_9)
26 }
```

Listing 13: Vereinfachte Sprungbedingungen

4.9.1 Algorithmus

Zur Vereinfachung der Bedingungen werden die Programmliste, die exklusiven Variablenmengen, die bisher definierten Variablen und die an diesem Punkt in jedem Fall wahren Kontrollvariablen benötigt. Die innerhalb eines Sprungs definierten Variablen werden nach dem Rücksprung nicht mehr verwendet, weil die Anweisungen innerhalb eines Sprungs übersprungen werden können und die damit definierten Variablen nicht in jedem Fall definiert sind. Generell sind für diesen Algorithmus nur die Kontollvariablen interessant, weil durch die vorhergehenden Schritte nicht sichergestellt ist, dass alle Kontrollvariablen in den Bedingungen definiert sind. Für alle anderen Variablen und Anweisungen wird davon ausgegangen, dass der Kontrollfluss und die Def-Use Ketten sich nicht widersprechen. Für jeden Sprung wird ein Selbstaufruf getätigt, der die Zielliste des Sprungs bearbeitet.

Die hier verwendeten wahren Kontrollvariablen sind nicht die gleichen wie die, von denen in Abschnitt 4.2.1 gesprochen wurde. Die Variablen in Abschnitt 4.2.1 ergeben sich durch das Laden von Konstanten. Die wahren Kontrollvariablen in diesem Abschnitt ergeben sich durch die Bedingungen der Sprünge und dem Kontrollfluss.

Die Programmliste wird von vorne beginnend durchlaufen. Die Menge der definierten Variablen wird mit den Eingabevariablen initialisiert. Wenn ein Statement gefunden wird, werden seine Zielvariablen zu den definierten Variablen hinzugefügt.

Bei einem Sprung werden zuerst alle bisher nicht definierten Kontrollvariablen aus der Sprungbedingung entfernt. Dadurch werden sie implizit als wahr angenommen und es kann nicht vorkommen, dass benötigter Code nicht ausgeführt wird. Anschließend werden auch die an diesem Punkt auf jeden Fall wahren Variablen aus der Bedingung entfernt. Da sie bekanntermaßen wahr sind, müssen sie nicht zur Laufzeit überprüft werden. Darauffolgend wird die Bedingung mit den in Abschnitt 4.2 beschriebenen Verfahren vereinfacht. Das weitere Vorgehen hängt von der Bedingung des Sprungs ab. Wenn die Bedingung eine Tautologie ist, wird die Zielliste des Sprungs in die aktuelle Programmliste hinter dem Rücksprunglabel des Sprungs eingefügt. Der Sprung und das zugehörige Rücksprunglabel werden aus der aktuellen Liste gelöscht. Das ist möglich, weil der Sprung zur Laufzeit immer ausgeführt werden würde. Das Einfügen der Zielliste an der Position des Labels ohne weitere Prüfungen ist möglich, weil die Sprünge bisher nur eine If-Then Semantik haben. Ist die Bedingung dagegen eine Kontradiktion, wird der Sprung und das dazugehörige Rücksprunglabel aus der aktuellen Liste gelöscht. Die Begründung dieses Vorgehens ist analog der Begründung des Vorgehens bei einer Tautologie: der Sprung würde zur Laufzeit niemals ausgeführt werden. Als dritte und letzte Möglichkeit kann die Bedingung weder eine Tautologie noch eine Kontradiktion sein, das heißt, dass sie zur Laufzeit ausgewertet werden muss. Dann werden die Variablen bestimmt, die in jedem Fall wahr sein müssen, wenn die Bedingung wahr ist. Das sind alle Variablen, die in allen Konjunktionstermen der DNF vorkommen. Anschließend folgt ein Selbstaufruf mit der Liste des Sprungs und den gerade berechneten wahren Variablen vereinigt mit den bereits bekannten wahren Variablen. Die übergebenen definierten Variablen sind alle bis zu diesem Punkt definierten Variablen.

Labels müssen nicht weiter bearbeitet werden. Zu jedem Label gehört zu diesem Zeitpunkt genau ein Sprung und der wurde im vorherigen Absatz bearbeitet.

Sprünge mit Bedingungen, die Tautologien sind, werden in die aufrufende Programmliste integriert, weil nachfolgende Verfahren Code innerhalb einer Programmliste bearbeiten. Wenn der Sprung als Sprung bestehen bleibt, kann es vorkommen, dass mögliche Optimierungen nicht genutzt werden können. Später in Abschnitt 4.12 können Bedingungen teilweise zu einer Tautologie geändert werden. Wenn dieser letzte Sprung einer vollständigen exklusiven Sprungmenge jedoch eine Kontradiktion als Bedingung besitzt, ist das nicht möglich. Auch da würde demzufolge Optimierungspotenzial ungenutzt bleiben, wenn die Sprünge nicht entfernt werden würden.

Dieses Verfahren benutzt Informationen von übergeordneten Sprüngen, um die Bedingung eines Sprungs zu vereinfachen (die wahren Variablen). Das ist möglich, weil zum erreichen eines Sprungs die übergeordneten Sprünge ausgeführt worden sein müssen. Das heißt, dass ein Sprung nur ausgeführt wird, wenn seine Bedingung und die Bedingungen aller übergeordneten Sprünge erfüllt sind. Deshalb dürfen nach Anwendung dieses Verfahrens keine Sprünge in der Sprunghierarchie weiter nach oben verschoben werden. Damit würden potenziell notwendige Informationen verloren gehen. Dabei gibt es eine Ausnahme: Sprünge mit einer Tautologie als Bedingung können wie in diesem

Abschnitt beschrieben in die Liste des übergeordneten Sprungs eingebunden werden, weil wahr das neutrale Element der Konjunktion ist. Außerdem kann eine Tautologie keine neuen in jedem Fall wahren Variablen (wenn die Bedingung wahr ist) einführen.[5]

4.9.2 Abhängigkeiten

Zur Vereinfachung der Bedingungen werden die in Abschnitt 4.2 berechneten Informationen und beschriebenen Verfahren eingesetzt. Außerdem wird dieses Verfahren vom Kontrollfluss und damit allen Verfahren, die Sprünge erstellen und verschachteln, beeinflusst (Abschnitte 4.5, 4.6 und 4.8). Weiterhin sind die (nicht) definierten Kontrollvariablen wichtig. Damit beeinflusst auch das Verfahren in Abschnitt 4.7 diesen Algorithmus.

4.10 Sprünge mit der gleichen Bedingung

Trotz unterschiedlicher Guards können mehrere Sprünge gleiche Bedingungen besitzen. Das kann zum Beispiel dann vorkommen, wenn nicht nur ein Wert unter einer bestimmten Bedingung benötigt wird und unter anderen nicht, sondern mehrere. Dann müssen mehrere Guards eingesetzt werden, weil ein Guard immer nur den Wert einer Variablen auf die Zielvariable kopieren kann. Dabei sind jedoch die Kontrollvariablen der Guards identisch.

In Listing 14 ist eine solche Situation dargestellt. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 48 gezeigt.

5 Eine Tautologie hat einen leeren Konjunktionsterm, einen Konjunktionsterm, der nur aus (bereits vorhandenen) wahren Variablen besteht, oder die Form `(a) ∨ (¬a)`. Da es jedoch keine Negation gibt, müssen im letzten Fall die beiden Konjunktionsterme verschiedene Variablen enthalten.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    { ((Local_0))
5      double Local_2 := Call "f1" (double Input_1)
6      double Local_4 := Move  (double Local_2)
7    } goto Label_0
8    label: Label_0
9    { ((Local_1))
10     double Local_3 := Call "f2" (double Input_1)
11     double Local_5 := Move  (double Local_3)
12   } goto Label_1
13   label: Label_1
14   { ((Local_0))
15     double Local_6 := Call "f3" (double Input_2)
16     double Local_8 := Move  (double Local_6)
17   } goto Label_2
18   label: Label_2
19   { ((Local_1))
20     double Local_7 := Call "f4" (double Input_2)
21     double Local_9 := Move  (double Local_7)
22   } goto Label_3
23   label: Label_3
24   double Output_0 := Choose  (double Local_4, double Local_5)
25   double Output_1 := Choose  (double Local_8, double Local_9)
26 }
```

Listing 14: Sprünge mit gleichen Bedingungen können zusammengefasst werden

Die Sprünge in den Zeilen 4 und 14 bzw. 9 und 19 haben die gleichen Bedingungen. Sie können deshalb zusammengefasst werden, so dass zur Laufzeit weniger Bedingungen geprüft und weniger Sprünge ausgeführt werden müssen.

Listing 15 zeigt das Ergebnis des in diesem Abschnitt vorgestellten Verfahrens.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    { ((Local_0))
5      double Local_2 := Call "f1" (double Input_1)
6      double Local_4 := Move  (double Local_2)
7      double Local_6 := Call "f3" (double Input_2)
8      double Local_8 := Move  (double Local_6)
9    } goto Label_2
10   label: Label_2
11   { ((Local_1))
12     double Local_3 := Call "f2" (double Input_1)
13     double Local_5 := Move  (double Local_3)
14     double Local_7 := Call "f4" (double Input_2)
15     double Local_9 := Move  (double Local_7)
16   } goto Label_3
17   label: Label_3
18   double Output_0 := Choose  (double Local_4, double Local_5)
19   double Output_1 := Choose  (double Local_8, double Local_9)
20 }
```

Listing 15: Sprünge wurden zusammengefasst

Das Ziel dieses Abschnitts ist das Zusammenführen mehrerer Sprünge mit der gleichen Bedingung, die sich in einer Programmliste befinden, sodass nur noch ein Sprung vorhanden ist. Dabei muss darauf geachtet werden, dass beim Verschieben des Codes keine Datenabhängigkeiten verletzt werden.

Zur Kontrolle der Datenabhängigkeiten werden die auftretenden Statements, die Nachfolger der auftretenden Statements und die Vorgänger der Statements bzw. die Definitionen der gelesenen Variablen innerhalb der Sprünge benötigt. Wie in Abschnitt 4.8 zählen auch die Variablen der Sprungbedingungen als gelesene Variablen. Als Traversierungsreihenfolge wird wieder die Reihenfolge der Breitensuche verwendet. Zum Vermeiden von Mehrfachberechnungen der Statements, Vorgänger und Nachfolger werden diese Informationen wieder wie in Abschnitt 4.8 zwischengespeichert. Als zusätzliche Schwierigkeit erweisen sich in diesem Abschnitt jedoch Effekte zweiter Ordnung[6], durch die eine Cacheinvalidierung notwendig ist. Durch die Traversierungsreihenfolge ist diese Invalidierung jedoch nicht rekursiv nötig. Damit bleibt der Aufwand für die Zwischenspeicherung überschaubar.

Effekte zweiter Ordnung können auftreten, wenn Code das Verschieben anderen Codes verhindert, aber der blockierende Code später selbst verschoben wird. Ein Beispiel dafür wird in Listing 16 gezeigt.

6 englisch: second order effects, hier: Die Verschiebung von Code A ermöglicht erst die Verschiebung von Code B.

```
1 Jump with condition 1
2 Statement
3 Jump with condition 2
4 Jump with condition 1
5 …
```

Listing 16: Beispiel für Effekte zweiter Ordnung

Angenommen, der Sprung in Zeile 3 verhindert, dass der Sprung in Zeile 1 nach unten verschoben werden kann. Weiterhin verhindert das Statement in Zeile 2, dass der Sprung in Zeile 4 nach oben verschoben werden kann. Dann können die beiden Sprünge in den Zeilen 1 und 4 nicht zusammengefasst werden. Wenn jedoch der Sprung in Zeile 3 weiter nach unten verschoben wird, kann auch der Sprung in Zeile 1 weiter nach unten verschoben werden (unter der Annahme, dass dies nicht vom Statement in Zeile 2 verhindert wird). Damit ist die Zusammenfassung der Sprünge in den Zeilen 1 und 4 möglich. Das Verschieben des Sprungs in Zeile 3 hat erst das Verschieben des Sprungs in Zeile 1 und damit auch die Zusammenfassung der Sprünge ermöglicht.

4.10.1 Algorithmus

Das in diesem Abschnitt beschriebene Verfahren ermöglicht Effekte zweiter Ordnung durch eine iterative Vorgehensweise. Die Bearbeitung einer Programmliste wird wiederholt, bis keine Änderungen mehr auftreten.

Der Algorithmus arbeitet mit einer äußeren und einer inneren Schleife. Sprünge, die von der äußeren Schleife gefunden werden, werden im folgenden erste Sprünge genannt. Sprünge, die von der inneren Schleife gefunden werden, werden dementsprechend zweite Sprünge genannt.

Die äußere Schleife läuft von vorne beginnend durch die Programmliste und sucht Sprünge. Wenn ein Sprung gefunden wurde und es der erste Schleifendurchlauf mit dieser Liste ist, wird die Zielliste des Sprungs an das Ende der Warteschlange der Breitensuche angefügt. Weiterhin wird die innere Schleife gestartet. Sie läuft beginnend mit dem nächsten Element in der Programmliste die Programmliste durch, aber nur solange der von der äußeren Schleife gefundene Sprung zur Programmliste gehört.

Die innere Schleife verwaltet eine Menge der aufgetretenen Statements. Ist das Element an der aktuellen Position ein Statement, wird es zu dieser Menge hinzugefügt. Bei Sprüngen dagegen wird zuerst geprüft, ob eine Zusammenfassung mit dem in der äußeren Schleife gefundenen Sprung möglich ist. Grundvoraussetzung dafür ist, dass die Bedingungen der beiden Sprünge gleich sind. Wenn die Menge der Nachfolger des ersten Sprungs und die Menge der aufgetretenen Statements disjunkt sind, können die Sprünge zusammengeführt werden. Dazu wird die Zielliste des ersten Sprungs an den Anfang der Zielliste des zweiten eingefügt. Weiterhin werden die Elemente der direkten Guards und der natürlichen Guards des ersten Sprungs zu den entsprechenden Mengen des zweiten Sprungs hinzugefügt. Schließlich wird der erste Sprung inklusive seinem korrespondierenden Label aus der Programmliste gelöscht. Wenn die Mengen nicht disjunkt waren, wird als nächstes geprüft, ob die Menge der aufgetretenen Statements disjunkt ist mit der Menge der Vorgänger (inklusive der Definitionen der Variablen der Sprungbedingungen) des zweiten Sprungs. Wenn das so ist, kann der zweite Sprung an das Ende des

ersten Sprungs angehangen werden. Dazu wird die Zielliste des zweiten Sprungs an das Ende der Zielliste des ersten Sprungs angehangen und die direkten und natürlichen Guards des zweiten Sprungs in die Mengen des ersten eingefügt. Danach wird der zweite Sprung inklusive dem zugehörigen Label aus der Programmliste gelöscht. Wenn die beiden Sprünge zusammengeführt wurden, haben sich ihre Ziellisten geändert bzw. der Sprung existiert nicht mehr. Deshalb wird der Zwischenspeicher für diese beiden Sprünge invalidiert. Ansonsten werden die Statements, die in der Zielliste des zweiten Sprungs (auch rekursiv in den Sprüngen des Sprungs) zu der Menge der aufgetretenen Statements hinzugefügt.

Wenn nach dem vollständigen Durchlaufen der äußeren Schleife Änderungen (das heißt, dass Sprünge zusammengefügt worden) durchgeführt worden, wird die Programmliste wieder an den Anfang der Warteschlange der Breitensuche eingefügt, um Effekte zweiter Ordnung ausnutzen zu können.

4.10.2 Abhängigkeiten

Um zu vermeiden, dass Datenabhängigkeiten verletzt werden, werden Def-Use Ketten (Abschnitt 4.3) eingesetzt.

Beeinflusst wird dieses Verfahren durch die Erstellung von Sprüngen (Abschnitte 4.5 und 4.6). Außerdem werden die Bedingungen verglichen, womit auch Abschnitt 4.9 Einfluss auf das Ergebnis dieses Algorithmus hat.

4.11 Copy Propagation

Eine Anweisung der Form

```
a = b
```

bzw. in der in dieser Arbeit verwendeten Notation

```
type Variable_1 := Move  (type Variable_2)
```

wird Kopieranweisung genannt. Bei der Copy Propagation werden Vorkommen des Ziels einer Kopieranweisung (im obigen Beispiel `a` und `Variable_1`) durch ihre Werte (`b` und `Variable_2`) ersetzt, unter der Bedingung, dass die Werte der beiden Variablen zwischenzeitlich nicht geändert wurden. [KSK09, Seiten 32 – 33]

Durch die Verwendung der SSA Form können die Werte der Variablen nach ihrer Definition nicht geändert werden. Deshalb ist keine Prüfung auf geänderte Werte notwendig.

Die Umwandlung von Guards in Move-Anweisungen erzeugt unter Umständen nicht notwendige Kopieranweisungen. In Listing 17 sind notwendige und nicht notwendige Kopieranweisungen vorhanden. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 49 gezeigt.

```
1  {
2    boolean Local_0 = Destruct A AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 = Destruct B AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    { ((Local_0))
5      double Local_2 = Call f (double Input_1)
6      double Local_3 = Move  (double Local_2)
7    } goto Label_0
8    label Label_0
9    { ((Local_1))
10     double Local_4 = Move  (double Input_1)
11   } goto Label_1
12   label Label_1
13   double Output_0 = Choose  (double Local_3, double Local_4)
14   double Output_1 = Move  (double Input_2)
15 }
```

Listing 17: Notwendige und nicht notwendige Kopieranweisungen

An dieser Stelle sei in Erinnerung gerufen, dass Choose-Anweisungen nichtdeterministisch einen Wert ihrer Quellvariablen auswählen, die nicht $\perp$ sind, und zur Zielvariable kopieren. Nur wenn alle Quellvariablen den Wert $\perp$ besitzen, ist auch die Zielvariable $\perp$. Im Gegensatz zu den φ-Knoten der SSA Form hängt die Auswahl der Choose-Anweisungen nur von den Werten der Quellvariablen ab, nicht vom konkret durchlaufenen Kontrollflusspfad. (siehe Kapitel 3, [WL14a, Seiten 155 – 156])

Die Anweisung in Zeile 10 ist notwendig, weil sie ursprünglich ein Guard war (und damit nur bedingt ausgeführt werden soll, Listing 49 Zeile 20) und die Variable `Input_1` auch außerhalb des Sprungs definiert ist (Wert $\neq \perp$). Die Anwendung von Copy Propagation an dieser Stelle würde dazu führen, dass im Fall von `Local_1 = false` in der Choose-Anweisung in Zeile 13 statt der undefinierten Variablen `Local_4` die definierte Variable `Input_1` als Argument übergeben würde, wodurch sich das Verhalten des Programms ändern würde. Anstatt aus der Menge { `Local_3` } könnte die Choose-Anweisung in Zeile 13 dann aus der Menge { `Local_3` , `Input_1` } auswählen.

Die Move-Anweisung in Zeile 6 dagegen ist nicht notwendig, weil die Variable `Local_2` nur definiert ist, wenn der Sprung ausgeführt wird. Damit ist `Local_2` genau dann definiert, wenn `Local_3` definiert ist. Die Move-Anweisung in Zeile 14 ist notwendig, weil sie sowohl als Quelle als auch als Ziel nicht-`Local`-Variablen besitzt. Eingabe- und Ausgabevariablen können durch Copy Propagation nicht eliminiert werden, weil sie wesentlich für den Datenfluss sind und damit das Programmverhalten beeinflussen. Neben den Move-Anweisungen werden auch Guard- und Choose-Anweisungen, die nur eine Quellvariable besitzen, als direkte Zuweisungen behandelt. Choose-Anweisungen mit nur einer Quellvariablen können bei der Anwendung von CSE (Abschnitt 4.15) entstehen.

In Listing 18 ist das Ergebnis der Copy Propagation angewendet auf das obige Beispiel dargestellt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    { ((Local_0))
5      double Local_2 := Call "f" (double Input_1)
6    } goto Label_0
7    label: Label_0
8    { ((Local_1))
9      double Local_4 := Move  (double Input_1)
10   } goto Label_1
11   label: Label_1
12   double Output_0 := Choose  (double Local_2, double Local_4)
13   double Output_1 := Move  (double Input_2)
14 }
```

Listing 18: Ergebnis der Copy Propagation

4.11.1 Algorithmus

Der Algorithmus zur Copy Propagation arbeitet in zwei Phasen. Zuerst wird eine Ersetzungstabelle aufgebaut. Danach werden in der zweiten Phase die Ersetzungen durchgeführt.

Durch das Ersetzen von Variablen werden die Def-Use Beziehungen geändert. Da der Algorithmus im folgenden Abschnitt 4.12 die Def-Use Ketten verwendet, müssen sie im Anschluss an diesen Algorithmus neu berechnet werden.

4.11.1.1 Ersetzungstabelle

Die Variablenersetzungen werden in einer Tabelle gespeichert, die einer zu ersetzenden Variablen ihre Ersetzung zuordnet. Diese Ersetzungstabelle wird iterativ aufgebaut, das heißt, dass nacheinander Variablenpaare `(search, replace)` eingefügt werden. Dabei soll bei der Ersetzung von Variablen die Variable `search` durch die Variable `replace` ersetzt werden. Die Variable `search` ist der Schlüssel in der Tabelle.

Beim Ersetzen einer Variablen wird zuerst geprüft, ob diese Variable als Schlüssel in der Tabelle vorkommt. Wenn das nicht der Fall ist, wird die Variable selbst zurückgegeben. Ansonsten ist die Rückgabe die entsprechende Variable `replace`.

Für das Einfügen eines neuen Ersetzungspaares ist es nicht ausreichend nur das Ersetzungspaar in die Tabelle einzutragen. Es können Ersetzungsketten auftreten, zum Beispiel

`Local_3` → `Local_5` → `Local_2`

die bei diesem naiven Ansatz durch mehrfache Anwendung der Ersetzung aufgelöst werden müssten. Damit später eine einzige Variablenersetzung ausreicht, wird bei jedem Einfügen eines Ersetzungspaares die ganze Tabelle entsprechend angepasst. Beim Einfügen eines Ersetzungspaares `(search, replace)` wird dazu zuerst die Ersetzung auf `replace` angewendet, damit nach Anwendung dieses Ersetzungspaares keine weitere Ersetzung notwendig ist. Im Weiteren ist mit `replace*` diese ersetzte Variable ge-

meint, die ursprüngliche Version wird nicht mehr benötigt. Als nächstes werden alle in der Ersetzungstabelle vorhandenen Einträge `(search', replace')` betrachtet und alle `replace'` durch `replace*` ersetzt, für die gilt, dass `replace'` die gleiche Variable wie `search` ist. Damit wird erreicht, dass nach Anwendung eines Ersetzungspaares `(search'', replace'')` die Anwendung des Ersetzungspaares `(search, replace*)` nicht notwendig ist. Schließlich wird das Ersetzungspaar `(search, replace*)` in die Ersetzungstabelle eingetragen.

4.11.1.2 Aufbau der Ersetzungstabelle

Da in der SSA Form jede Variable nur einmal geschrieben werden kann, kann eine globale Ersetzungstabelle verwendet werden. Neben der Ersetzungstabelle werden noch die Informationen benötigt, welche Statements ursprünglich Guards waren und welche Variablen lokal definiert wurden. Lokal bedeutet dabei innerhalb der aktuellen Programmliste. Eine Ausnahme von dieser Regel wird später in diesem Abschnitt besprochen.

Die Programmliste wird von vorne beginnend durchlaufen. Bei jedem gefundenen Statement werden die Zielvariablen zu den lokal definierten Variablen hinzugefügt. Wenn das Statement ein Move-Statement oder ein Guard- oder Choose-Statement mit nur einer Quellvariablen ist, kann unter folgenden Bedingungen Copy Propagation eingesetzt werden:

- Das Statement war kein Guard oder die Ersetzung der Quellvariable gehört zu den lokal definierten Variablen.

  und

- Die Ersetzung der Quell- oder die Ersetzung der Zielvariablen ist eine `Local`-Variable.

Die Ersetzungen der jeweiligen Variablen müssen verwendet werden, weil durch die Aufnahme in die Ersetzungstabelle quasi der Code schon geändert wurde – durch die Verwendung der Ersetzungen arbeitet der Algorithmus immer mit dem aktuellen Stand. Es wäre zum Beispiel möglich, dass eine Kopieranweisung als Quellvariable eine `Local`-Variable besitzt, als Zielvariable jedoch eine `Output`-Variable. Damit wäre eine Anwendung von Copy Propagation möglich. Wenn die Quellvariable jedoch beispielsweise durch eine bereits in die Ersetzungstabelle eingetragene Ersetzung durch eine `Input`-Variable ersetzt werden würde, wäre die Anwendung von Copy Propagation nicht mehr möglich. Wenn die ersetzte Zielvariable eine `Local`-Variable ist, wird das Ersetzungspaar (Zielvariable, ersetzte Quellvariable) in die Ersetzungstabelle eingetragen, ansonsten (Quellvariable, ersetzte Zielvariable).

Bei einem Sprung wird das in diesem Abschnitt beschriebene Verfahren auf die Zielliste des Sprungs angewendet. Eine Besonderheit existiert für Sprünge, deren Bedingung eine Tautologie ist. Wenn zusätzlich alle anderen Sprünge der exklusiven Sprungmenge Kontradiktionen als Bedingungen haben, wird der Sprung immer aufgeführt. Dann kann die Zielliste des Sprungs so behandelt werden, als ob sich die Elemente in der aktuellen Programmliste befinden. Das heißt, dass bei der Bearbeitung der Zielliste des Sprungs nicht nur die innerhalb der Zielliste des Sprungs definierten Variablen als lokale Varia-

blen angesehen werden, sondern auch die bisherigen in der aktuellen Programmliste berechneten lokalen Variablen. Ansonsten wird für die Bearbeitung der Zielliste des Sprungs eine neue leere Menge von lokalen Variablen erstellt. Die bei der Bearbeitung des Sprungs berechnete Menge lokaler Variablen wird mit dem zugehörigen Sprung gespeichert.

Wenn ein Label erreicht wird, werden die Elemente der Mengen lokaler Variablen, die bei der Bearbeitung der Ziellisten der Sprünge berechnet wurden, zur aktuellen Menge lokaler Variablen hinzugefügt. Das ist möglich, weil die lokalen Variablen eines Sprungs in der aktuellen Programmliste verfügbar werden, wenn der Sprung zurückkehrt, und nur dann definiert sind, wenn die aktuelle Programmliste ausgeführt wird (sonst könnte der Sprung nicht ausgeführt werden).

4.11.1.3 Anwenden der Ersetzungen

Nachdem die Ersetzungstabelle vollständig aufgebaut wurde, müssen die Ersetzungen angewendet werden. Dazu werden alle Programmlisten durchlaufen und die Quell- und Zielvariablen der Statements sowie die Variablen in den Sprungbedingungen gemäß der Ersetzungstabelle ersetzt.

4.11.2 Abhängigkeiten

Die notwendige Information, welches Statement früher ein Guard war, wird in Abschnitt 4.5 erstellt. Im selben Abschnitt werden auch Kopieranweisungen generiert, die von der Copy Propagation möglicherweise eliminiert werden können. Außerdem können bei der Anwendung der CSE (Abschnitt 4.15) Anweisungen erstellt werden, die wie Kopieranweisungen behandelt werden können.

Durch die Ersetzung von Variablen werden die Def-Use Beziehungen geändert. Deswegen ist das Neuberechnen der Def-Use Ketten (Abschnitt 4.3) notwendig, wenn sie nach Einsatz dieses Algorithmus nochmals benötigt werden.

4.12 Sprünge organisieren

Bisher gehört zu jedem Label genau ein Sprung. Die Semantik ist ähnlich der einer If-Then Anweisung. In diesem Schritt sollen die Sprünge so organisiert werden, dass ihre Semantik einer If-Then-ElseIf-… Anweisung entspricht. Dabei haben alle Sprünge, die einer If-Then-ElseIf-… Anweisung entsprechen, ein gemeinsames Rücksprunglabel und ihre Bedingungen schließen sich paarweise aus. Das heißt, dass sich jeweils eine Kontradiktion ergibt, wenn die Bedingungen paarweise konjunktiv miteinander verknüpft werden. Eine solche Menge von Sprüngen wird exklusive Sprungmenge genannt. Zusätzlich sollen die exklusiven Sprungmengen keine weiteren Elemente überspringen. Das heißt, dass in der Liste alle Sprünge und ihr Label direkt aufeinanderfolgen.

Die in den vorherigen Abschnitten gebildeten Labels und ihre Sprünge sind auch exklusive Sprungmengen, da jedes Label genau einen Sprung besitzt und Sprung und Label direkt aufeinanderfolgen.

In Listing 19 ist eine mögliche Ausgangssituation für das in diesem Abschnitt beschriebene Verfahren dargestellt. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 50 gezeigt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    { ((Local_0))
5      double Local_2 := Move  (double Input_1)
6    } goto Label_0
7    label: Label_0
8    { ((Local_1))
9      double Local_3 := Move  (double Input_2)
10   } goto Label_1
11   label: Label_1
12   double Output_0 := Choose  (double Local_2, double Local_3)
13 }
```

Listing 19: Eine bessere Organisation der Sprünge ist möglich

Die Bedingungen der beiden Sprünge in den Zeilen 4 und 8 schließen sich gegenseitig aus. Dadurch ist es möglich, sie zu einer exklusiven Sprungmenge mit If-Then-ElseIf-… Semantik zusammenzufassen. Da die Bedingungen auch vollständig sind (alle Bedingungen disjunktiv verknüpft ergeben eine Tautologie), kann auf die Prüfung der letzten Bedingung verzichtet werden, weil sie immer wahr sein muss, wenn alle anderen Bedingungen falsch sind.

In Listing 20 ist das Ergebnis des in diesem Abschnitt beschriebenen Algorithmus angewandt auf das obige Beispiel dargestellt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    label: Label_0
5    label: Label_1
6    { ((Local_0))
7      double Local_2 := Move  (double Input_1)
8    } goto Label_2
9    { ((true))
10     double Local_3 := Move  (double Input_2)
11   } goto Label_2
12   label: Label_2
13   double Output_0 := Choose  (double Local_2, double Local_3)
14 }
```

Listing 20: Sprünge mit If-Then-ElseIf-… Semantik

Der in diesem Abschnitt beschriebene Algorithmus setzt voraus, dass in den Bedingungen nur definierte Variablen vorkommen. Wenn eine undefinierte Variable später entfernt werden würde, ist nicht mehr sichergestellt, dass die exklusiven Sprungmengen

wirklich exklusiv sind. Ein einfaches Beispiel dafür sind folgende Bedingungen, die jeweils nur aus einer Variablen bestehen:

- `((a))`
- `((b))`

wobei `a` und `b` zur selben exklusiven Variablenmenge gehören. Es ist leicht zu erkennen, dass sich die beiden Bedingungen gegenseitig ausschließen. Wenn jedoch `a` noch nicht definiert wurde und daher aus der Bedingung entfernt wird, ergeben sich die Bedingungen

- `((true))`
- `((b))`

die sich nicht mehr gegenseitig ausschließen. Der in Abschnitt 4.9 beschriebene Algorithmus entfernt noch nicht definierte Variablen aus den Bedingungen. Außerdem rechnet der dortige Algorithmus mit einer If-Then Semantik, die nach Anwendung des Algorithmus in diesem Abschnitt nicht mehr gegeben ist. Daher muss der Algorithmus in Abschnitt 4.9 vor dem Algorithmus dieses Abschnitts ausgeführt werden.

4.12.1 Algorithmus

Das hier beschriebene Verfahren benötigt die Menge aller Statements der Sprünge, aller Nachfolger der Statements in den Sprüngen und die Menge aller Vorgänger (inklusive der Definitionen der Kontrollvariablen in den Bedingungen). Diese Mengen müssen rekursiv berechnet werden. Die Sprünge werden verändert, das heißt, dass das Speichern von Zwischenergebnissen nicht einfach möglich ist. Durch eine passende Traversierungsreihenfolge ist es jedoch möglich, dass die gespeicherten Zwischenergebnisse erst dann veraltet sind, wenn sie nicht mehr benötigt werden. Diese Traversierungsreihenfolge wird durch eine Breitensuche erreicht.

Bei der Bearbeitung einer Programmliste wird sie von vorne beginnend durchlaufen und eine Menge der gesehenen Sprünge verwaltet. Bei der Verarbeitung eines Elements werden zuerst die gesehenen Sprünge gesucht, die Vorgänger des aktuellen Elements sind. Ein Sprung ist Vorgänger eines Statements, wenn das Statement ein Nachfolger des Sprungs ist (im Sprung wird eine Variable definiert, die eine Quellvariable des Statements ist). Weiterhin ist der gesehene Sprung Vorgänger eines anderen Sprungs, wenn mindestens ein Statement des gesehenen Sprungs innerhalb der Menge der Vorgänger (inklusive der Definitionen der Kontrollvariablen der Bedingungen) des anderen Sprungs vorkommt. Die gesehenen Sprüngen, die Vorgänger des gefundenen Elements in der Programmliste sind, werden aus der Menge der gesehenen Sprünge entfernt.

Mit den gesehenen Sprüngen, die Vorgänger des gefundenen Elements in der Programmliste sind, werden exklusive Sprungmengen erstellt und vor dem gefundenen Element in die Liste eingefügt. Dazu werden Buckets verwendet, die jeweils eine exklusive Sprungmenge repräsentieren. Die Sprünge werden nacheinander betrachtet und in die Buckets einsortiert. Dafür wird die Bedingung des Sprungs nacheinander mit jeder Bedingung der Sprünge im Bucket konjunktiv verknüpft. Wenn die verknüpfte Bedingung eine Kontradiktion ist, wird der Sprung in den Bucket einsortiert. Ansonsten wird das nächste Bucket getestet oder, wenn kein weiteres Bucket vorhanden ist, ein neues er-

stellt. Nachdem alle Sprünge in Buckets einsortiert wurden, wird für jedes Bucket ein neues Label erstellt. Die Sprünge eines Buckets und ihr neues Label werden dann zusammenhängend vor dem gefundenen Element in die Programmliste eingefügt. Dabei wird gespeichert, welcher Sprung als letzter in die Liste eingetragen wurde. Anschließend wird geprüft, ob die Bedingungen der Sprünge der soeben erstellten exklusiven Sprungmenge vollständig sind, das heißt, dass ihre Bedingungen disjunktiv verknüpft eine Tautologie ergeben. Dann kann die Bedingung des letzten Sprungs auf wahr gesetzt werden, sofern sie keine Kontradiktion ist. Eine Bedingung auf wahr setzen heißt, dass die Bedingung nur noch aus einem leeren Konjunktionsterm besteht. Wenn die Bedingung nicht auf wahr gesetzt wurde, wird das Label gekennzeichnet, dass es keinen letzten Sprung hat (der letzte Sprung des Labels ist `null`). Ansonsten wird der letzte Sprung im Label gespeichert.

Nachdem die Menge der gesehenen Sprünge abgearbeitet und die notwendigen exklusiven Sprungmengen erstellt wurden, muss noch das gefundene Element verarbeitet werden. Wenn es ein Sprung ist, wird es in die Menge der gesehenen Sprünge eingefügt und aus der Liste gelöscht. Außerdem wird die Zielliste des Sprungs in die Warteschlange der Breitensuche eingetragen. Andere Elementarten müssen an dieser Stelle nicht beachtet werden.

Wenn alle Elemente der Programmliste abgearbeitet wurden und die Menge der gesehenen Sprünge nicht leer ist, werden schließlich mit den verbleibenden Sprüngen exklusive Sprungmengen erstellt und an das Ende der Liste angehangen.

Bei diesem Verfahren sind Effekte zweiter Ordnung möglich. Angenommen Sprung `B` enthält einen Use für eine Variable, die in Sprung `A` definiert wurde. Sprung `A` wird damit vor Sprung `B` gesetzt und Sprung `B` kann weiter nach hinten in der Programmliste verschoben werden. Dadurch könnte eventuell doch Sprung `A` weiter nach hinten verschoben werden und mit einem anderen Sprung eine exklusive Sprungmenge bilden. Eine solche Situation wird vom vorgestellten Algorithmus nicht abgedeckt, damit beim Bearbeiten der gesehenen Sprünge keine Abhängigkeiten der gesehenen Sprünge untereinander beachtet werden müssen. Ein iteratives Ausführen des Algorithmus ist nicht möglich, weil er die Sprungbedingungen manipuliert und davon ausgeht, dass bei Beginn seiner Ausführung Sprünge ausschließlich If-Then Semantik besitzen.

4.12.2 Abhängigkeiten

Zur Berechnung der Vorgänger und Nachfolger werden Def-Use Ketten (Abschnitte 4.3 und 4.11) verwendet. Außerdem werden die Informationen und Verfahren aus Abschnitt 4.2 benötigt, um zu testen ob sich Bedingungen gegenseitig ausschließen oder vollständig sind.

Beeinflusst wird dieses Verfahren von allen Verfahren, die Sprünge erstellen (Abschnitte 4.5 und 4.6) oder Manipulieren (Abschnitte 4.8 und 4.10). Weiterhin ist die Definiertheit der Kontrollvariablen der Sprungbedingungen wichtig. Damit beeinflusst auch Abschnitt 4.7 das Ergebnis dieses Algorithmus. Da Code verschoben wird und damit auch Datenflussabhängigkeiten für diesen Algorithmus wichtig sind, beeinflusst auch die

Copy Propagation (Abschnitt 4.11) mit dem Löschen von Kopieranweisungen diesen Algorithmus.

4.13 Dead Code Elimination (DCE)

Durch die Organisation der Sprünge kann es auftreten, dass nicht mehr alle Kontrollvariablen zur Laufzeit geprüft werden. Diese Situation tritt beispielsweise im Beispiel des letzten Verfahrens in Listing 20 auf. Die Variable `Local_1` wird zwar definiert, aber nirgends benutzt. In Listing 21 ist das Programm nach der DCE dargestellt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    label: Label_0
4    label: Label_1
5    { ((Local_0))
6      double Local_2 := Move  (double Input_1)
7    } goto Label_2
8    { ((true))
9      double Local_3 := Move  (double Input_2)
10   } goto Label_2
11   label: Label_2
12   double Output_0 := Choose  (double Local_2, double Local_3)
13 }
```

Listing 21: DCE entfernte nicht genutzte Variablendefinition

Die Definition der nicht benutzten Variablen `Local_1` wurde entfernt.

Code, dessen linke Seite (die Zielvariablen) im restlichen Programm nicht benutzt wird, ist (vollständig) tot [KRS94, Seite 147]. Variablen, die auf einigen Pfaden von ihrer Definition zum Ende des Programms tot sind, sind partiell tot [BG97, Seite 159].

In SSA kann eine Variable nur einmal geschrieben werden, wodurch nur eine globale Menge lebender Variablen benötigt wird [BGG05, Seite 5]. Der in [BGG05, Seiten 4 – 5] vorgeschlagene Algorithmus initialisiert alle Programmpunkte als unbesichtigt und alle interessanten Variablen, Kontrollvariablen und Eingabe- und Ausgabevariablen, als lebend. In jedem Schritt des Algorithmus wird ein bisher unbesichtigter Programmpunkt bearbeitet, der eine Zuweisung an eine lebende Variable ist. Alle Variablen auf der rechten Seite der Zuweisung (Quellvariablen) sind lebende Variablen. Das wird so lange wiederholt, bis keine unbesichtigten Programmpunkte mehr existieren, die Zuweisungen an lebende Variablen sind.

4.13.1 Algorithmus

In dieser Arbeit wird ein etwas abweichender Algorithmus benutzt. Die Menge lebender Variablen ist global. Initial sind nur Ausgabevariablen lebend. Eine Programmliste wird vom Ende beginnend rückwärts durchlaufen. Wenn ein Statement gefunden wird, wird geprüft, ob mindestens eine Zielvariable lebt. Wenn das der Fall ist, werden alle Quellvariablen zu den lebenden Variablen hinzugefügt. Ansonsten wird das Statement gelöscht.

Wenn ein Label gefunden wird, werden die Ziellisten der Sprünge, die das Label als Rücksprunglabel besitzen, mit dem hier beschriebenen Algorithmus rekursiv bearbeitet, wenn ihre Bedingung keine Kontradiktion ist. Ein Sprung ist ungenutzt, wenn seine Bedingung eine Kontradiktion ist oder in seiner Zielliste keine lebenden Variablen definiert werden. Sind alle Sprünge des Labels ungenutzt, es hat keinen letzten Sprung (das heißt, dass die Bedingungen der Sprünge nicht vollständig sind) oder der letzte Sprung ist ungenutzt, können die Bedingungen der ungenutzten Sprünge auf falsch gesetzt werden, das heißt, dass alle Konjunktionsterme aus der DNF entfernt werden. Schließlich werden die Variablen, die in den Bedingungen der Sprünge vorkommen, zu den lebenden Variablen hinzugefügt.

Sprünge müssen nicht mehr bearbeitet werden, wenn sie beim Durchlaufen der Liste gefunden werden, weil sie bereits beim Bearbeiten des zugehörigen Labels bearbeitet wurden.

Die Bedingungen von ungenutzten Sprüngen können nicht in jedem Fall auf falsch gesetzt werden, weil die Sprünge Teil einer If-Then-ElseIf-… Anweisung sind. Wenn die Bedingungen der Sprünge einer exklusiven Sprungmenge vollständig sind, wurde in Abschnitt 4.12 die Bedingung des letzten Sprungs auf wahr gesetzt. Anstatt nur zum angegebenen Rücksprunglabel zu springen, würde der letzte Sprung ausgeführt werden. Sofern der letzte Sprung nicht ungenutzt ist, wäre das Ausführen dieses Sprungs falsch. Ist der Sprung jedoch ungenutzt, gibt es keinen Unterschied zwischen der Ausführung oder dem Überspringen des letzten Sprungs, weil nur Sprünge einer exklusiven Sprungmenge betrachtet werden und diese alle das gleiche Rücksprunglabel besitzen. Ungenutzte Sprünge haben eine Kontradiktion als Bedingung oder enthalten keine Statements. Wenn die Bedingung des letzten Sprungs eine Kontradiktion ist, wird er niemals ausgeführt und damit ist es nicht wichtig, ob er erreicht wird oder nicht. Wenn der Sprung keine Statements hat, kann die Zielliste leer sein oder ausschließlich weitere Sprünge und Rücksprunglabels enthalten. Da auch diese Sprünge alle ungenutzt sein müssen (ansonsten würde der Sprung Statements enthalten, die lebende Variablen definieren), werden die Bedingungen dieser Sprünge auf falsch gesetzt. Bei der Überführung des Zwischenformats ins Ausgabeformat (Abschnitt 4.16) werden solche Sprünge eliminiert.

Im in [BGG05] vorgeschlagenen Algorithmus werden auch die Kontrollvariabeln als lebend initialisiert. Der Algorithmus dieser Arbeit dagegen behandelt Kontrollvariablen so wie andere Variablen auch: sie werden erst dann zu lebenden Variablen, wenn sie genutzt werden. Damit können Sprungbedingungen flexibler behandelt werden. Weiterhin werden Eingabevariablen nicht als lebend initialisiert. Da Eingabevariablen nicht aus der Variablendeklaration entfernt werden, ist diese Initialisierung hier nicht erforderlich.

4.13.2 Abhängigkeiten

Durch die Vereinfachung der Sprungbedingungen und der Organisation der Sprünge können Kontrollvariablen aus den Sprungbedingungen entfernt werden. Wenn sie nicht an anderen Punkten im Programm benötigt werden, sind die Definitionen der entfernten Variablen tot. Damit beeinflussen die Algorithmen der Abschnitte 4.9 und 4.12 den hier beschriebenen Algorithmus. Weiterhin sind die Definitionen innerhalb von Sprüngen

wichtig, um zu entscheiden ob ein Sprung benötigt wird. Da mit Copy Propagation (Abschnitt 4.11) Definitionen aus Sprüngen entfernt werden können, kann auch sie Einfluss auf das Ergebnis dieses Algorithmus haben. Schließlich kann auch durch die Anwendung von CSE (Abschnitt 4.15) toter Code entstehen.

4.14 Generierung des Codes zur Bedingungsprüfung

Das Ausgabeformat erwartet für jede Sprungbedingung eine einzige Kontrollvariable – im Zwischenformat ist es jedoch eine DNF. Deshalb muss Code erstellt werden, der die DNF auswertet und in einer Variablen `c` speichert. Da es im Zwischenformat nicht möglich ist, eine einzige Variable als Bedingung für einen Sprung zu speichern, wird stattdessen eine DNF erzeugt, die ausschließlich die Variable `c` enthält.

Zur Berechnung von Konjunktionen und Disjunktionen können Guard- und Choose-Statements eingesetzt werden [WL14a, Seite 156]. Eine Konjunktion wird nur mittels eines Guard-Statements berechnet, wenn sie aus mehr als einer Variablen besteht. Ansonsten ist die einzige Variable bereits das Ergebnis der Konjunktion. Analoges gilt für die Berechnung der Disjunktion mittels Choose-Statements.

In Listing 22 ist beispielhaft eine Ausgangssituation für dieses Verfahren dargestellt. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 51 gezeigt.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
5    boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
6    label: Label_0
7    label: Label_1
8    { ((Local_3) OR (Local_1))
9      double Local_5 := Call "f2" (double Input_2)
10   } goto Label_2
11   { ((Local_2 AND Local_0))
12     double Local_4 := Call "f1" (double Input_2)
13   } goto Label_2
14   label: Label_2
15   double Output_0 := Choose  (double Local_4, double Local_5)
16 }
```

Listing 22: Sprungbedingungen werden nicht berechnet

Die Sprungbedingungen sind, wie in den Zeilen 8 und 11 dargestellt, mit den Sprüngen gespeichert. Sie werden jedoch im Programm nicht berechnet.

Listing 23 zeigt das Beispiel mit generiertem Code zur Berechnung der Sprungbedingungen.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
5    boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
6    label: Label_0
7    label: Label_1
8    boolean Local_9 := Choose  (boolean Local_3, boolean Local_1)
9    { ((Local_9))
10     double Local_5 := Call "f2" (double Input_2)
11   } goto Label_2
12   boolean Local_10 := Guard  (boolean Local_2, boolean Local_0)
13   { ((Local_10))
14     double Local_4 := Call "f1" (double Input_2)
15   } goto Label_2
16   label: Label_2
17   double Output_0 := Choose  (double Local_4, double Local_5)
18 }
```

Listing 23: Sprungbedingungen werden explizit berechnet

Die Bedingung des Sprungs in Zeile 9 wird mithilfe eines Choose-Statements in Zeile 8 berechnet. Weiterhin wird die Bedingung des Sprungs in Zeile 13 in Zeile 12 mittels eines Guard-Statements berechnet. Dabei ist zu beachten, dass die beiden Sprünge keine exklusive Sprungmenge mehr bilden, weil zwischen ihnen und ihrem Rücksprunglabel andere Anweisungen stehen.

4.14.1 Algorithmus

Der Code wird erstellt, indem jede Programmliste durchlaufen wird und die erstellten Guard- und Choose-Anweisungen direkt vor dem jeweiligen Sprung eingefügt werden. Dabei werden die exklusiven Sprungmengen zerstört, weil eine ihrer Eigenschaften ist, dass zwischen den Sprüngen und dem zugehörigen Label kein weiterer Code stehen darf.

Bei der Erzeugung des Codes wird nicht darauf geachtet, ob eine gleiche Berechnung bereits ausgeführt wurde und damit redundanter Code eingefügt wird. Zur Elimination dieser Redundanzen wird später CSE (Abschnitt 4.15) eingesetzt.

4.14.2 Abhängigkeiten

Die Verfahren in den Abschnitten 4.9, 4.12 und 4.13 können die Sprungbedingungen ändern und haben damit Einfluss auf das Ergebnis des Algorithmus dieses Abschnitts.

Zu beachten ist auch, dass durch die Zerstörung der exklusiven Sprungmengen keine Verfahren mehr angewendet werden können, die diese voraussetzen.

4.15 Partial Redundancy Elimination (PRE) / Common Subexpression Elimination (CSE)

PRE transformiert ein Programm so, dass es normalerweise nach der Transformation weniger Berechnungen enthält, die auf bestimmten Programmpfaden redundant sind. CSE ist ein Spezialfall von PRE. [PSS03, Seite 1]

Ein Ausdruck an einem bestimmten Punkt ist genau dann redundant, wenn alle Pfade, die diesen Punkt erreichen, den Ausdruck bereits berechnen und keiner der Teilausdrücke des Ausdrucks neu definiert wurden. Dann kann der Ausdruck durch eine Referenz ersetzt werden. Weiterhin ist ein Ausdruck an einem Punkt partiell redundant, wenn er auf einigen, aber nicht allen, Pfaden zu diesem Punkt bereits berechnet wird. [BC94, Seiten 159 – 160]

PRE hat zwei schwerwiegende Einschränkungen: es erkennt nur lexikalisch gleiche Ausdrücke und es kann Teilausdrücke nicht umordnen [BC94, Seite 159]. Das Verfahren in [BC94] unterscheidet zwischen Variablen- und Ausdrucksnamen. Variablennamen sind das Ziel von Kopieranweisungen (Zuweisungen). Ausdrucksnamen dagegen bezeichnen das Ergebnis von Berechnungen. Damit erhält jeder (Teil-)Ausdruck einen Namen. Innerhalb einer einzigen Routine erhalten lexikalisch identische Ausdrücke einen gemeinsamen Namen. Dadurch ist die Effektivität des Verfahrens von der Wahl der Namen abhängig. Der in [BC94] benutzte Compiler verwendet eine Zwischenrepräsentation mit Drei-Adress-Code, das heißt, dass die Anweisungen zwei Quelloperanden und einen Zieloperanden haben. Ausdrücke der Form

```
o4 = o1 + o2 + o3
```

müssen deshalb in Teilausdrücke zerlegt werden. Bei assoziativen Operationen existieren verschiedene Möglichkeiten dafür. Die Effektivität von PRE ist auch von der Zerlegung in die Teilausdrücke abhängig. [BC94, Seiten 159 – 161]

An dieser Stelle sei in Erinnerung gerufen, dass die Zwischenrepräsentation von SIG im Gegensatz zur Zwischenrepräsentation in [BC94] keinen Drei-Adress-Code benutzt. Die Zwischenrepräsentation von SIG ist in Abschnitt 4.1.1 genauer beschrieben.

Weiterhin ist nach [BC94, Seite 160] eine Schlüsselfunktion von PRE, dass Code-Pfade nie verlängert werden.

Wenn `A1` und `A2` Auftreten des selben Ausdrucks `A` sind und es einen Pfad von `A1` zu `A2` gibt, auf dem nichts passiert, das den Wert des Ausdrucks `A` verändern würde, dann ist `A2` redundant bezüglich `A1` . [KCLLTC99, Definition 2, Seite 632]

Der Unterschied zwischen CSE und PRE ist, dass CSE nur vollständige Redundanzen eliminiert, PRE dagegen auch partielle. PRE fügt dafür jedoch zusätzlichen Code ein. Das bedeutet, dass PRE auf der einen Seite die mächtigere Optimierung ist und das Laufzeitverhalten stärker optimiert als CSE, das aber auf der anderen Seite zulasten der Codegröße geschieht. In Abbildung 9 ist ein Beispiel dargestellt, an dem der Unterschied gezeigt werden kann. [GCC:GCSE]

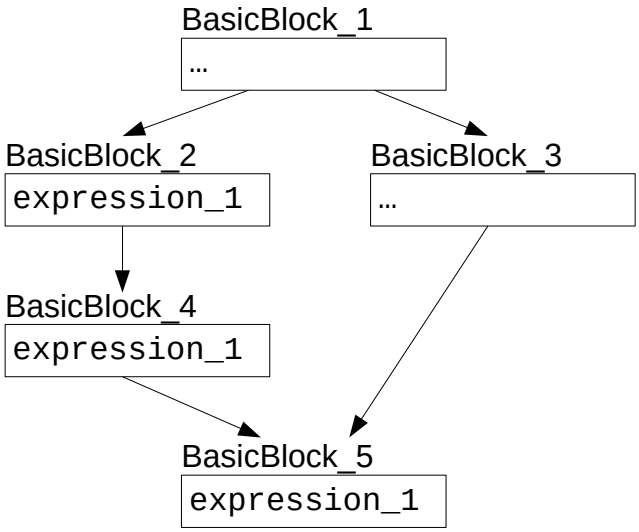

Abbildung 9: Beispiel, in dem CSE und PRE unterschiedliche Ergebnisse liefern

CSE würde nur die `expression_1` in `BasicBlock_4` entfernen, da sie dort vollständig redundant ist. Zu `BasicBlock_5` existiert jedoch einen Pfad, in dem der Ausdruck nicht definiert wird – er ist nur partiell redundant, weswegen CSE ihn nicht entfernen kann. PRE dagegen entfernt den Ausdruck `expression_1` in `BasicBlock_4` und `BasicBlock_5`. Zur Erhaltung der Semantik muss jedoch die `expression_1` auf allen Pfaden zu `BasicBlock_5` eingefügt werden, in denen der Ausdruck (noch) nicht vorkommt. [GCC:GCSE]

Es ist noch eine weitere Form der Redundanz möglich, die jedoch nicht durch die Definition 2 aus [KCLLTC99] abgedeckt wird und auch keine Pfade verlängert. Auf verschiedenen Pfaden können die gleichen (Teil-)Bedingungen berechnet werden. Ein Beispiel dazu ist Listing 24.

```
1    BasicBlock_i {
2      …
3    } next: boolean Local_m ? BasicBlock_j : BasicBlock_k
4
5    BasicBlock_j {
6      boolean Local_n := Guard  (boolean Local_p, boolean Local_q)
7    } next: …
8
9    BasicBlock_k {
10     boolean Local_o := Guard  (boolean Local_p, boolean Local_q)
11   } next: …
```

Listing 24: Gleiche Ausdrücke auf unterschiedlichen Pfaden

Unter der Annahme, dass kein Pfad von `BasicBlock_j` zu `BasicBlock_k` und umgekehrt existiert, sind die Ausdrücke in den Zeilen 6 und 10 nicht Redundant im Sinne von Definition 2 aus [KCLLTC99]. Trotzdem könnte eine der Anweisungen – und damit zumindest etwas Speicherplatz für den Code – eingespart werden, wenn die zwei Anweisungen durch eine Anweisung am Ende von `BasicBlock_i` ersetzt werden würden.

Bei der Erzeugung des Codes zur Berechnung der Sprungbedingungen (Abschnitt 4.14) wurde nicht auf die oben beschriebenen Redundanzen geachtet. Außerdem kann durch das Zurückrechnen boolescher Verknüpfungen (Abschnitt 4.2.2) bereits Code vorhanden sein, der nochmals für die Berechnung der Bedingungen eingefügt wird. Zur Elimination der Redundanzen wird CSE eingesetzt, weil es einfacher zu implementieren ist.

4.15.1 Modifikationen von CSE

Bei der Synthese des Kontrollflusses wird nur neuer Code erzeugt, der boolesche Bedingungen berechnet. Dazu werden Guard- und Choose-Statements verwendet, die eine boolesche Variable als Zielvariable besitzen. Deshalb werden nur solche Statements und Move-Statements betrachtet. Die Move-Statements werden betrachtet, weil sie für das Zurückrechnen boolescher Verknüpfungen nützlich sind. Die booleschen Verknüpfungen werden wie in Abschnitt 4.2.2 beschrieben zurückgerechnet. Das Zurückrechnen hat den Vorteil, dass auch unterschiedliche Variablen, deren Werte durch logisch äquivalente Verknüpfungen boolescher Variablen berechnet wurden und damit gleich sind, erkannt werden.

In Listing 25 ist ein Beispiel dargestellt, in dem CSE Variablen mit gleichen Werten nicht erkennt.

```
1  BasicBlock_u {
2    boolean Local_e := Guard  (boolean Local_a, boolean Local_b)
3    boolean Local_f := Guard  (boolean Local_c, boolean Local_d)
4    boolean Local_g := Choose  (boolean Local_e, boolean Local_f)
5  } next: BasicBlock_z
6
7  BasicBlock_v {
8    boolean Local_h := Guard  (boolean Local_a, boolean Local_b)
9    boolean Local_i := Guard  (boolean Local_c, boolean Local_d)
10   boolean Local_j := Choose  (boolean Local_h, boolean Local_i)
11 } next: BasicBlock_z
12
13 BasicBlock_w {
14   boolean Local_k := Guard  (boolean Local_a, boolean Local_b)
15   boolean Local_l := Guard  (boolean Local_c, boolean Local_d)
16   boolean Local_m := Choose  (boolean Local_k, boolean Local_l)
17 } next: …
```

Listing 25: Beispiel für lexikalisch unterschiedliche Variablen mit gleichem Wert

CSE erkennt, dass die Variablen `Local_k` und `Lockal_l` den gleichen Wert haben wie die Variablen `Local_e` und `Local_h` bzw. `Local_f` und `Local_i`, und fügt entsprechende φ-Knoten (Choose-Anweisungen) in den Zeilen 14 und 15 ein. In Zeile 16 haben die Argumente der Choose-Instruktion andere Namen als in den Zeilen 4 und 10. Deshalb geht CSE von unterschiedlichen Werten aus. Durch das Zurückrechnen der booleschen Verknüpfungen kann erkannt werden, dass `Local_g`, `Local_j` und auch `Local_m` die DNF

`((Local_a ∧ Local_b) ∨ (Local_c ∧ Local_d))`

repräsentieren. Damit kann auch die redundante Berechnung in Zeile 16 durch einen entsprechenden φ-Knoten ersetzt werden. Alternativ dazu sollte auch Global Value Numbering (GVN) [Cli95] das gewünschte Ergebnis berechnen, was aber in dieser Arbeit nicht untersucht wurde.

Das gewünschte Ergebnis des Beispiels in Listing 25 nach der CSE ist in Listing 26 dargestellt.

```
1  BasicBlock_u {
2    boolean Local_e := Guard  (boolean Local_a, boolean Local_b)
3    boolean Local_f := Guard  (boolean Local_c, boolean Local_d)
4    boolean Local_g := Choose  (boolean Local_e, boolean Local_f)
5  } next: BasicBlock_z
6
7  BasicBlock_v {
8    boolean Local_h := Guard  (boolean Local_a, boolean Local_b)
9    boolean Local_i := Guard  (boolean Local_c, boolean Local_d)
10   boolean Local_j := Choose  (boolean Local_h, boolean Local_i)
11 } next: BasicBlock_z
12
13 BasicBlock_w {
14   boolean Local_k := Choose  (boolean Local_e, boolean Local_h)
15   boolean Local_l := Choose  (boolean Local_f, boolean Local_i)
16   boolean Local_m := Choose  (boolean Local_g, boolean Local_j)
17 } next: …
```

Listing 26: Gewünschtes Ergebnis nach CSE

In den Beispielen in den Listings und Abbildungen in diesem Abschnitt wurden wegen der einfacheren Darstellung Basic Blocks verwendet. Der hier beschriebene Algorithmus arbeitet jedoch auf dem in dieser Arbeit beschriebenen Zwischenformat, den Programmlisten (siehe Abschnitt 4.1.2). Für die Idee des Algorithmus ist dies unerheblich.

4.15.2 Algorithmus

Der Algorithmus benutzt eine Tabelle, die jeder Kontrollvariablen eine DNF zuordnet, die sie repräsentiert. Das ist die gleiche Datenstruktur, wie sie auch in Abschnitt 4.2.2 verwendet wurde. In diesem Algorithmus wird sie jedoch nicht verschränkt mit den exklusiven Variablenmengen aufgebaut, da die exklusiven Variablenmengen bereits bekannt sind. Weiterhin wird eine Tabelle der verfügbaren Ausdrücke verwendet. Sie ordnet jedem an der aktuellen Position in der Programmliste verfügbaren Ausdruck (DNF) die Menge der Variablen zu, die den Wert des Ausdrucks speichern. Schließlich wird noch eine Tabelle benötigt, die jedem Label die Tabellen der verfügbaren Ausdrücke zuordnet, die die Sprünge, die das Label als Rücksprungziel besitzen, nach der Abarbeitung der jeweiligen Programmliste aufgebaut haben (Label-Tabelle). Bis auf die Tabelle der verfügbaren Ausdrücke sind die Tabellen global für diesen Algorithmus, werden bei einem Selbstaufruf also unverändert übergeben.

Eine Programmliste wird von vorne beginnend durchlaufen. Bei einem gefundenen Statement wird zuerst geprüft, ob es ein Move-, Guard- oder Choose-Statement mit einer Kontrollvariable als Ziel ist. Wenn die Prüfung erfolgreich ist, wird der Wert der

Zielvariablen wie in Abschnitt 4.2.2 beschrieben zurückgerechnet. Ist die dabei resultierende DNF ein Schlüssel in der Tabelle der verfügbaren Ausdrücke, wird das Statement durch ein Choose-Statement ersetzt, dessen Quellvariablen die Variablen sind, die den Wert der DNF speichern. Ansonsten wird die DNF mit der Zielvariablen des Statements in die Tabelle eingefügt. Wenn das Statement kein Move-, Guard- oder Choose-Statement mit einer Kontrollvariablen als Ziel ist, wird es nicht geändert.

Als nächstes wird die Verarbeitung eines Sprungs beschrieben. Die Zielliste eines Sprungs hat nur einen Vorgänger. Deshalb sind alle aktuell verfügbaren Ausdrücke auch in der Zielliste des Sprungs verfügbar. Zur Bearbeitung der Zielliste erfolgt ein Selbstaufruf mit der Zielliste des Sprungs und einer Kopie der Tabelle der verfügbaren Ausdrücke. Nach der Rückkehr vom Selbstaufruf wird die Tabelle der verfügbaren Ausdrücke der Zielliste des Sprungs mit dem Rücksprunglabel als Schlüssel in die Label-Tabelle eingetragen.

Schließlich müssen noch die Labels verarbeitet werden. Sie stehen an den Stellen, an denen sich mehrere Kontrollflusspfade vereinigen. CSE verwendet jedoch nur Ausdrücke, die auf allen Pfaden vom Start bis zum aktuellen Punkt verfügbar sind. Deshalb muss an dieser Stelle der Schnitt der verfügbaren Ausdrücke der Vorgänger berechnet werden. Die verfügbaren Ausdrücke der Vorgänger sind die aktuell verfügbaren Ausdrücke und die dem Label zugeordneten verfügbaren Ausdrücke aus der Label-Tabelle. Jede DNF in diesem Schnitt ist ein Schlüssel in der zukünftigen Tabelle der verfügbaren Ausdrücke. Die Menge der Variablen ist die Vereinigung der entsprechenden Mengen der verfügbaren Ausdrücke.

4.15.3 Abhängigkeiten

Für die Vereinfachung der aus den Kontrollvariablen zurückgerechneten DNFs werden die Informationen und Verfahren aus Abschnitt 4.2 verwendet. Außerdem beeinflusst der Algorithmus aus Abschnitt 4.14 diesen Algorithmus, weil erst er Anweisungen einfügt, die redundant sein können.

Der hier vorgestellte Algorithmus kann Choose-Statements einfügen, die nur eine Quellvariable besitzen. Das geschieht dann, wenn ein Wert redundant ist und nur einmal vorher berechnet wurde. Solche Choose-Statements können wie Move-Statements betrachtet werden. Deshalb wird nach CSE nochmals Copy Propagation (Abschnitt 4.11) ausgeführt.

Außerdem kann neuer toter Code durch die Anwendung der in dieser Arbeit beschriebenen CSE entstehen. Die Variablen `Local_k` und `Local_l` in Listing 26 sind zum Beispiel tot, wenn sie ausschließlich für die Berechnung der Variablen `Local_m` eingesetzt werden. Deshalb ist an dieser Stelle der Einsatz von DCE sinnvoll. Dazu wird das Verfahren aus Abschnitt 4.13 eingesetzt. Da jedoch durch die Generierung des Codes zur Bedingungsprüfung (Abschnitt 4.14) die exklusiven Sprungmengen zerstört wurden, werden bei dieser Ausführung der DCE die Sprungbedingungen nicht modifiziert.

CSE kann mit in die Überführung des Zwischenformats ins Ausgabeformat (Abschnitt 4.16) integriert werden. Es wurde trotzdem separat implementiert, damit es bei Bedarf

später leichter durch ein anderes Verfahren wie PRE oder das bereits erwähnte Global Value Numbering ersetzt werden kann.

4.16 Überführung des Zwischenformats ins Ausgabeformat

Die bisher verwendeten und manipulierten Programmlisten müssen ins Ausgabeformat überführt werden, bevor andere Komponenten des SIG-Compilers sie verwenden können. Dabei wird darauf geachtet, dass möglichst viele Sprünge bereits zur Compilezeit ausgewertet und möglichst wenige Basic Blocks erstellt werden.

In Listing 27 sind die aus dem Beispiel in Listing 2 generierten Basic Blocks dargestellt.

```
1  BasicBlock_5 {
2    boolean Local_0 := Destruct "hold" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
3  } next: boolean Local_0 ? BasicBlock_6 : BasicBlock_7
4
5  BasicBlock_6 {
6    double Local_2 := Move  (double Pre_0)
7  } next: BasicBlock_8
8
9  BasicBlock_7 {
10   double Local_3 := Move  (double Input_0)
11 } next: BasicBlock_8
12
13 BasicBlock_8 {
14   double Output_0 := Choose  (double Local_2, double Local_3)
15   double Post_0 := Move  (double Output_0)
16 } next: --
```

Listing 27: Aus Beispiel in Listing 2 generierte Basic Blocks

Es ist zu erkennen, dass für den Sprung in Zeile 8 in Listing 2 keine weitere Bedingungsprüfung generiert wurde, sondern seine Zielliste den Else-Zweig des Sprungs in Zeile 5 bildet.

Im Folgenden ist mit Vorgänger eines Basic Blocks ein Basic Block mit dem Ausgangspunkt (next-Referenz bzw. Sprung) gemeint.

4.16.1 Erzeugung der Basic Blocks

Mit dem Erstellen eines Basic Blocks werden gleichzeitig die Referenzen der Vorgänger auf den neuen Basic Block gesetzt. Dazu wird die Menge der Vorgänger des neuen Basic Blocks verwendet. Anhand dieser Menge können die notwendigen Referenzen der Vorgänger auf diesen Basic Block gesetzt werden.

Die übergebene Menge der Vorgänger wird am Ende geleert. Sie war für das Setzen der Referenzen notwendig. Da dies geschehen ist, werden ihre Elemente nicht mehr benötigt. Außerdem wird später davon ausgegangen, dass die Menge der Vorgänger nach der Erstellung eines neuen Basic Blocks leer ist.

4.16.2 Überführung

Bei der Überführung des Zwischenformats ins Ausgabeformat wird darauf geachtet keine leeren Basic Blocks (Basic Blocks ohne Statements und ohne Sprung) zu erzeugen (Ausnahme: der End-Basic Block). Deshalb werden nicht sofort neue Basic Blocks erzeugt, wenn Referenzen gesetzt werden müssen, sondern die Basic Blocks mit dem jeweiligen Ausgangspunkt gespeichert. Erst wenn der Basic Block benötigt wird, wird er erzeugt und die Referenzen entsprechend gesetzt.

Der Algorithmus benötigt folgende Daten: die Programmliste, die in das Ausgabeformat umgewandelt werden soll, das Rücksprunglabel, die Label-Tabelle, die jedem Label eine Menge von Vorgängern zuordnet, und den Vorgänger, der zur zu bearbeitenden Liste gesprungen ist. Die Label-Tabelle ist für den Algorithmus global, wird also bei einem Selbstaufruf unverändert als Referenz übergeben.

Da der eigentliche Algorithmus den Startknoten nicht speichert, wird zur Initialisierung ein neuer Basic Block erzeugt und als Vorgänger übergeben. Als Ausgangspunkt wird die next-Referenz gewählt. Weiterhin wird bei der Initialisierung das Rücksprunglabel auf `null` gesetzt und die nötige Label-Tabelle angelegt. Nach Ablauf des Algorithmus kann aus der next-Referenz der Start-Basic Block ausgelesen werden. Sollte die next-Referenz auf keinen Basic Block zeigen, kann ein neuer, leerer Basic Block angelegt werden. Das Programm ist dann das leere Programm.

Der Algorithmus arbeitet rekursiv. Am Anfang eines Rekursionsschritts wird der aktuelle Basic Block auf `null` gesetzt (Basic Blocks werden erst erstellt, wenn sie benötigt werden) und die Menge der Vorgänger (des nächsten erstellten Basic Blocks) enthält ausschließlich den übergebenen Vorgänger.

Als nächstes wird eine Schleife ausgeführt, die über alle Elemente der Programmliste läuft. Für die Schleife gilt die Invariante, dass es keinen aktuellen Basic Block gibt (der Basic Block ist `null`) oder die Menge der Vorgänger leer ist. Je nach Typ des Elements (Statement, Sprung, Label oder GoTo) müssen bestimmte Aktionen durchgeführt werden. GoTos existieren nur innerhalb dieses Algorithmus und werden an der entsprechenden Stelle erklärt. Zu Beginn der Schleife ist die Schleifeninvariante erfüllt, weil kein aktueller Basic Block existiert.

Wenn das aktuell bearbeitete Element ein Statement ist, wird es in den aktuellen Basic Block eingetragen. Existiert kein aktueller Basic Block, wird ein neuer erstellt, wie in Abschnitt 4.16.1 beschrieben. Die Schleifeninvariante ist erfüllt: Wenn bereits ein Basic Block existierte, war die Menge der Vorgänger leer und es wurde kein Vorgänger hinzugefügt. Sollte kein Basic Block existiert haben, wurde bei der Erstellung des Basic Blocks die Menge der Vorgänger geleert.

Als nächstes werden Labels betrachtet. Dabei ist zu beachten, dass nur an den Anfang von Basic Blocks gesprungen werden kann. Ein Label muss nur beachtet werden, wenn Basic Blocks es anspringen, das heißt, dass in der Label-Tabelle Vorgänger für dieses Label eingetragen sind. Existieren Vorgänger für dieses Label, wird der aktuelle Basic Block – falls einer existiert – geschlossen. Das Schließen eines Basic Blocks bedeutet dabei, dass er in die Vorgängermenge mit der next-Referenz als Ausgangspunkt eingetragen und der aktuelle Basic Block auf `null` gesetzt wird. Damit können keine weite-

ren Statements eingefügt und auch kein Sprung erstellt werden. Durch den Eintrag in die Vorgängermenge können jedoch noch Referenzen gesetzt werden. Weiterhin werden alle Vorgänger des Labels in die Menge der Vorgänger eingefügt und das Label aus der Label-Tabelle gelöscht. Die Schleifeninvariante ist erfüllt, weil kein aktueller Basic Block existiert.

Wenn das Element ein Sprung ist, wird als erstes überprüft, ob seine Bedingung eine Tautologie oder Kontradiktion ist. Bei einer Tautologie wird ein GoTo erstellt. Im Gegensatz zu Sprüngen, die nur ausgeführt werden, wenn ihre Bedingung wahr ist, wird bei einem GoTo immer zum angegebenen Label gesprungen. Das Ziellabel des GoTos ist das Rücksprunglabel des Sprungs. Das erstellte Element wird direkt nach dem Sprung in die Programmliste eingefügt. Weiterhin wird die Zielliste des Sprungs direkt nach dem Sprung in die aktuelle Programmliste eingefügt. Damit werden als nächstes die Elemente, die in der Zielliste des Sprungs sind, bearbeitet und danach zum Rücksprunglabel des Sprungs gesprungen. Das ist das gewünschte Verhalten, wenn der Sprung immer ausgeführt wird, aber ohne zur Laufzeit die Bedingung prüfen und einen Sprung ausführen zu müssen. Ein Sprung mit einer Kontradiktion als Bedingung kann ignoriert werden. Es muss kein Code dafür erzeugt werden, weil er zur Laufzeit nie ausgeführt werden würde. Wenn die Bedingung weder eine Tautologie noch eine Kontradiktion ist, wird zuerst ein neuer Basic Block erstellt, wenn kein aktueller Basic Block existiert. Anschließend wird der Sprung des Basic Blocks erstellt und ein Selbstaufruf getätigt. Die übergebene Programmliste ist die Zielliste des Sprungs, das Rücksprunglabel ist das Rücksprunglabel des Sprungs. Als Vorgänger wird der aktuelle Basic Block mit dem Sprung als Ausgangspunkt übergeben. Alle Referenzen werden innerhalb des Selbstaufrufs richtig gesetzt oder in die Label-Tabelle eingetragen. Nach der Rückkehr vom Selbstaufruf bleibt nur noch das Schließen des aktuellen Basic Blocks, weil nur am Ende eines Basic Blocks Verzweigungen des Kontrollflusses bzw. Sprünge möglich sind.

Es ist zu zeigen, dass die Schleifeninvariante auch bei Sprüngen erfüllt ist. Wenn die Sprungbedingung eine Tautologie oder eine Kontradiktion ist, werden keine Änderungen am aktuellen Basic Block oder der Menge der Vorgänger vorgenommen. Deshalb ist die Invariante nach dem Schleifendurchlauf erfüllt, wenn sie vor dem Durchlauf erfüllt war, was vorausgesetzt wird. Ansonsten (die Sprungbedingung ist weder Tautologie noch Kontradiktion) wird als erstes ein neuer Basic Block erstellt, wenn kein aktueller existiert. Wenn ein aktueller Basic Block existiert, ist nach der Schleifeninvariante die Menge der Vorgänger leer. Ansonsten wird ein neuer Basic Block erstellt und dabei die Menge der Vorgänger geleert. Zu diesem Zeitpunkt ist die Schleifeninvariante erfüllt. Bei der Berechnung der Bedingungen und dem Selbstaufruf wird die Menge der Vorgänger nicht verändert, womit die Schleifeninvariante erfüllt bleibt, weil die Menge der Vorgänger leer ist. Schließlich wird der aktuelle Basic Block geschlossen. Dadurch ist die Menge der Vorgänger nicht mehr leer, aber es existiert kein aktueller Basic Block mehr. Die Schleifeninvariante ist damit erfüllt.

Schließlich müssen noch die neu eingeführten GoTos bearbeitet werden. Die GoTos dienen dazu andere Elemente zu überspringen. Deshalb wird als erstes das erste auf das GoTo folgende Label gesucht, das angesprungen wird. Ein Label wird angesprungen, wenn es das Ziel des GoTos oder als Schlüssel in der Label-Tabelle vorhanden ist. Die

aktuelle Position in der Programmliste wird auf das Element direkt vor dem gefundenen Label gesetzt, damit im nächsten Schleifendurchlauf das Label verarbeitet wird. Wenn das gefundene Label das Ziel des GoTos ist, ist die Bearbeitung des Elements an dieser Stelle abgeschlossen. Die Verarbeitung des Labels im nächsten Schritt wird, falls erforderlich, einen neuen Basic Block erstellen und alle nötigen Referenzen setzen. Ansonsten (das Label ist nicht das Ziel des GoTos) wird der aktuelle Basic Block – falls einer existiert – geschlossen. Danach werden die Vorgänger zu den eventuell bereits dem Ziellabel des GoTos zugeordneten Vorgängern hinzugefügt bzw. ein entsprechender Eintrag erstellt. Im nächsten Schleifendurchlauf wird das gefundene Label verarbeitet. Bei der Konstruktion von Sprüngen und exklusiven Sprungmengen in dieser Arbeit kann der letztgenannte Fall nicht vorkommen. Auch durch das Einfügen des Codes zur Berechnung der Sprungbedingungen (Abschnitt 4.14) und dem damit verbundenen Zerstören der exklusiven Sprungmengen kann kein Code zwischen dem letzten Sprung und dem zugehörigen Label eingefügt werden. Nach einem Sprung mit einer Tautologie als Bedingung folgt immer das Rücksprunglabel. Dieser Fall wird trotzdem behandelt, damit auch bei einer späteren Änderung der vorhergehenden Verfahren die Überführung des Zwischenformats ins Ausgabeformat korrekt durchgeführt wird. Für die Argumentation der Erfüllung der Schleifeninvariante müssen zwei Fälle bei der Bearbeitung der GoTos unterscheiden werden. Im ersten Fall wird weder der aktuelle Basic Block noch die Menge der Vorgänger geändert. Damit ist die Schleifeninvariante erfüllt (unter der Voraussetzung, dass sie zu Beginn des Schleifendurchlaufs erfüllt war). Im zweiten Fall wird ein eventuell vorhandener aktueller Basic Block geschlossen. Dadurch ist die Schleifeninvariante erfüllt, weil kein aktueller Basic Block existiert.

Die Schleife ist beendet, wenn das Ende der Programmliste erreicht wurde. Die Schleifeninvariante ist notwendig für die Korrektheit des Algorithmus. Wenn sie nicht erfüllt wäre, würde ein aktueller Basic Block existieren und es gäbe gleichzeitig Basic Blocks, die zum aktuellen Basic Block springen. Das heißt, dass in die Mitte eines Basic Blocks gesprungen werden müsste, was nicht möglich ist.

Nach dem Abarbeiten der Schleife kann noch ein offener aktueller Basic Block oder eine nicht-leere Menge von Vorgängern existieren. Wenn das Rücksprunglabel `null` ist, wurde die Hauptliste bearbeitet und die Label-Tabelle muss leer sein, das heißt, dass zu jedem Sprung ein passendes Rücksprunglabel gefunden wurde. Die Vorgängermenge kann, muss aber nicht zwingend, nicht-leer sein (zum Beispiel wenn das letzte Element ein Label war). Wenn es gewünscht ist, dass es genau einen End-Basic Block gibt, kann in diesem Fall ein neuer Basic Block wie in Abschnitt 4.16.1 beschrieben erstellt werden. Ansonsten, das heißt, dass das Rücksprunglabel nicht null ist, muss ein neuer Eintrag in der Label-Tabelle erstellt werden, falls das Rücksprunglabel noch nicht eingetragen ist. Wenn es einen aktuellen Basic Block gibt, wird dieser als Vorgänger (mit der next-Referenz als Ausgangspunkt) des Rücksprunglabels zu den vorhandenen Vorgängern in der Label-Tabelle hinzugefügt. Ansonsten werden die Elemente der Vorgängermenge zu den bereits vorhandenen Vorgängern in der Label-Tabelle hinzugefügt.

In dieser Arbeit wurde die Variante gewählt, in der genau ein End-Basic Block existiert.

Nicht berücksichtigt wurde im vorgestellten Algorithmus die Situation, dass sowohl im wahr- als auch im falsch-Fall (Sprung und next-Referenz) zum selben Ziel gesprungen

wird. Diese Situation kann bei der verwendeten If-Then-ElseIf-… Semantik und der Dead Code Elimination (Abschnitt 4.13) nicht auftreten. Wenn ein solcher Fall eintritt, kann der Sprung entfernt werden. Sollte dadurch nur noch ein einziger Vorgänger existieren, muss kein neuer Basic Block erstellt, sondern der vorhandene kann wieder geöffnet werden.

4.16.3 Abhängigkeiten

Es wird erwartet, dass die Sprungbedingungen, wenn sie keine Tautologien oder Kontradiktionen sind, aus genau einer Variablen bestehen. Deshalb ist dieses Verfahren vom Verfahren in Abschnitt 4.14, das den Code zur Berechnung der Bedingungen erstellt, abhängig.

4.17 Aufräumen der lokalen Variablen

Im Eingabe- und Ausgabeformat werden die Variablen explizit deklariert, wie in Listing 1 in den Zeilen 9 – 11 zu sehen ist. Bei der Dead Code Elimination (Abschnitt 4.13) können lokale Variablen aus dem Code entfernt werden. Weiterhin können bei einer naiven Implementierung der Generierung des Codes zur Auswertung der Sprungbedingungen (Abschnitt 4.14) mehr Variablen neu erstellt werden als es notwendig ist.[7] Für diesen Schritt müssen die bisherigen Optimierungen für den Init- und Loop-Teil bereits abgeschlossen sein. Deshalb ist er am einfachsten auf dem Ausgabeformat zu realisieren, indem ihm der gesamte erstellte Block übergeben wird.

In Listing 28 ist verkürzt ein Beispiel einer Ausgangssituation dargestellt. Listing 53 stellt das vollständige Beispiel dar. Das Ausgangsprogramm, aus dem dieses Beispiel generiert wurde, wird in Listing 52 gezeigt.

```
1 Block_20 {
2   env: AlgebraicDataType_0, AlgebraicDataType_0
3   in: double
4   out: double
5   state: double
6   locals: boolean, boolean, boolean, boolean, double, double,
  double, double, double, double, double, double, boolean, boolean,
  boolean, boolean, boolean, boolean
7   init: BasicBlock_4
8   loop: BasicBlock_12
9 …
```

Listing 28: Beispiel mit vielen Local-Variablen

Es werden insgesamt 18 Local-Variablen deklariert, davon sind 10 Variablen Kontrollvariablen.

In Listing 29 ist die Situation nach dem Aufräumen der lokalen Variablen verkürzt dargestellt. In Listing 54 wird das vollständige Beispiel gezeigt.

7 Wenn Init- und Loop-Teil unabhängig voneinander betrachtet werden, können Kontrollvariablen eingefügt werden, die nur in jeweils einem Teil verwendet werden. Bei der Verwendung der Variablen in beiden Teilen kann die Anzahl der lokalen Variablen reduziert werden. Dabei wird jedoch die Anzahl der jeweils lebenden Variablen nicht verändert.

```
1 Block_20 {
2   env: AlgebraicDataType_0, AlgebraicDataType_0
3   in: double
4   out: double
5   state: double
6   locals: double, double, double, double, boolean, boolean, boolean,
  boolean, boolean, boolean, boolean
7   init: BasicBlock_4
8   loop: BasicBlock_12
9 …
```

Listing 29: Das Aufräumen der Local-Variablen hat deren Anzahl reduziert

Es werden insgesamt nur noch 11 Variablen deklariert, davon sind 7 Variablen Kontrollvariablen.

4.17.1 Algorithmus

Zur Reduzierung der Anzahl der lokalen Variablen muss zuerst bestimmt werden, wie viele lokale Variablen welchen Typs jeweils im Init- und Loop-Teil (der jeweils referenzierte Basic Block und rekursiv die per next-Referenz bzw. Sprung verketteten Basic Blocks) benötigt werden. Dazu werden die Blöcke einmal durchlaufen. Da jede verwendete `Local`-Variable definiert (geschrieben) werden muss, ist die Betrachtung der Zielvariablen der Statements ausreichend.

Danach werden für jeden Datentyp so viele `Local`-Variablen angelegt, wie maximal im Init- bzw. Loop-Teil verwendet wurden. Dabei wird auch die Liste, in der die lokalen Variablen deklariert werden (siehe Listing 1, Zeile 11), aktualisiert.

Als nächstes werden für den Init- bzw. Loop-Teil jeweils eine Ersetzungstabelle angelegt (eine einfache Tabelle, nicht die Datenstruktur aus Abschnitt 4.11.1.1), die jeweils einer vorkommenden `Local`-Variablen eine der neuen Variablen mit entsprechendem Typ zuordnet. Für die Ersetzung der Variablen werden wiederum die Basic Blocks einmal durchlaufen und die `Local`-Variablen in den Quell- und Zielvariablen der Statements und in den Bedingungen der Sprünge durch die neuen Variablen ersetzt. Andere Variablen bleiben unverändert.

4.17.2 Abhängigkeiten

Da dieses Verfahren auf dem Ausgabeformat arbeitet, ist es vom vorherigen Abschnitt (4.16) abhängig.

Alle Verfahren, die Variablen erstellen oder deren Definition löschen, haben Einfluss auf diesen Algorithmus. Das sind die in den Abschnitten 4.1.2, 4.11 und 4.13 – 4.15 vorgestellten Verfahren.

5 Vorbereitung der nächsten Compilerphase

Wie bereits in Kapitel 3 erwähnt, können in SIG Variablen den Wert ⊥ haben, der Anzeigt, dass sie nicht definiert sind. Jedoch unterstützen nicht alle Operatoren diesen Wert als Eingabe. Deshalb muss dieser Wert als Eingabe bei diesen Operatoren abgefangen werden. Eine Prüfung zur Laufzeit verlangsamt das Programm. Aus diesem Grund ist es wünschenswert, dass sie bereits zur Compilezeit durchgeführt wird. Wenn bekannt ist, unter welchen Bedingungen ein Basic Block ausgeführt wird und die Quellvariablen definiert sind, kann durch eine Implikation berechnet werden, ob eine Laufzeitprüfung auf den Wert ⊥ notwendig ist.

Angenommen die Quellvariablen eines Statements sind unter der Bedingung `(a ∨ ¬b)` definiert. Weiterhin wird der Basic Block, in dem das Statement steht, unter der Bedingung `(a ∧ ¬b)` ausgeführt. Die Implikation

`(a ∧ ¬b) → (a ∨ ¬b) = true`

lässt erkennen, dass bei der Ausführung des Statements niemals eine Quellvariable den Wert ⊥ besitzen kann. Dadurch ist eine Laufzeitprüfung unnötig. Anders ist der Fall, wenn der Basic Block unter der Bedingung `(a ∨ c)` ausgeführt wird. Die Implikation

`(a ∨ c) → (a ∨ ¬b) = (a ∨ ¬b ∨ ¬c)`

ist keine Tautologie. Damit kann zur Compilezeit nicht garantiert werden, dass bei der Ausführung des Statements keine Quellvariable den Wert ⊥ hat. Deswegen ist eine Laufzeitprüfung notwendig.

5.1 Algorithmus

Der Algorithmus benutzt wieder die DNF zur Verarbeitung der Bedingungen. Im Unterschied zu Kapitel 4 werden jedoch Negationen benötigt. Deshalb werden Literale eingeführt. Ein Literal besteht aus einer Variablen und einem Flag das angibt, ob die Variable positiv ist (Flag = falsch heißt, dass die Variable negiert ist). Ein Konjunktionsterm ist eine Menge von Literalen und eine DNF eine Menge von Konjunktionstermen. Wenn der Wert des Flags nicht explizit angegeben wird, ist es wahr und das Literal damit positiv.

Wenn eine Variable zur Laufzeit den Wert ⊥ annehmen kann, wird sie in der DNF mit dem Wert falsch interpretiert. Andererseits kann eine Variable, die in der DNF als wahr interpretiert wird, niemals zur Laufzeit den Wert ⊥ annehmen.

5.1.1 (Inverser) Kontrollflussgraph

Im folgenden Abschnitt werden die Vorgänger und Nachfolger eines Basic Blocks benötigt. Die Nachfolger sind ohne weitere Datenstrukturen effizient berechenbar, da ein Basic Block sie direkt referenziert (siehe Abschnitt 4.1.1: next-Referenz und Sprung). Der Kontrollflussgraph ist damit in den Basic Blocks gespeichert.

Die Vorgänger jedes Basic Blocks lassen sich durch eine Breitensuche über den Kontrollflussgraphen berechnen. Der Start-Basic Block, mit dem die Breitensuche startet,

hat keine Vorgänger. Beim Bearbeiten des Basic Block `bb` wird für jeden Nachfolger `succ(bb)` gespeichert, dass `bb` der Vorgänger von `succ(bb)` ist. Dazu kann eine Tabelle verwendet werden, die jedem Basic Block die Menge seiner Vorgänger zuordnet. Der End-Basic Block ist der Block, der keine Nachfolger besitzt.

5.1.2 Berechnung der Bedingungen

Neben dem im vorherigen Abschnitt beschriebenen Kontrollflussgraphen benötigt der Algorithmus zwei weitere Tabellen. Eine Tabelle speichert für jede Variable unter welcher Bedingung sie definiert ist. Die zweite Tabelle speichert für jeden Basic Block unter welcher Bedingung er ausgeführt wird. Alle `Input`- und `Environment`-Variablen sind immer definiert, das heißt, dass sie eine Tautologie als Bedingung erhalten. `Pre`-Variablen dagegen können auch undefiniert sein. Ihre Bedingung enthält ausschließlich die Variable selbst (eine `Pre`-Variable ist definiert, wenn sie definiert ist). Die Initialisierungen von `Block_1` in Listing 1 ist zum Beispiel:

```
double Input_0 --> (())
AlgebraicDataType_0 Input_1 --> (())
double Pre_0 --> ((Pre_0))
```

Außerdem wird eine Warteschlange verwendet, die die zu verarbeitenden Basic Blocks verwaltet und eine Menge der bereits verarbeiteten Basic Blocks. Am Anfang des Algorithmus wird der Start-Basic Block in die Warteschlange eingetragen.

Eine Schleife entnimmt so lange den ersten Basic Block aus der Warteschlange wie sie nicht leer ist. Ein Block wird dabei erst verarbeitet, wenn alle seine Vorgänger bereits verarbeitet wurden, ansonsten wird er wieder an das Ende der Warteschlange angehangen. Als erstes wird bei der Verarbeitung die Bedingung berechnet, unter der der Basic Block ausgeführt wird. Beim Start-Basic Block ist das eine Tautologie. Bei anderen Basic Blocks hängt die Bedingung von den Vorgängern ab. Die Teilbedingungen werden wie folgt aus den Vorgängern berechnet:

- Wenn der Vorgänger keinen Sprung besitzt oder die Ziele des wahr- und falsch-Zweigs beide der aktuelle Basic Block sind, dann ist die Teilbedingung die Bedingung, unter der der Vorgänger ausgeführt wird.
- Wenn der Vorgänger einen Sprung besitzt und nur der wahr-Zweig den aktuellen Basic Block als Ziel hat, dann ist die Teilbedingung die Bedingung unter der der Vorgänger ausgeführt wird konjunktiv verknüpft mit der Bedingung unter der die Sprungbedingung (-variable) definiert ist.[8]
- Wenn der Vorgänger einen Sprung besitzt und nur der falsch-Zweig den aktuellen Basic Block als Ziel hat, dann ist die Teilbedingung die Bedingung unter der der Vorgänger ausgeführt wird konjunktiv verknüpft mit der Negation der Bedingung unter der die Sprungbedingung (-variable) definiert ist.[9]

Nach der Berechnung der Bedingung unter der der Basic Block ausgeführt wird, werden die Bedingungen berechnet, unter der die Zielvariablen der Statements berechnet wer-

8 $\top \triangleq$ wahr (siehe Kapitel 3)

9 $\bot \triangleq$ falsch (siehe Kapitel 3)

den. Dazu läuft eine Schleife über alle Statements des Basic Blocks. Bei einem Choose-Statement ist die Zielvariable definiert, wenn eine der Quellvariablen definiert ist und das Statement ausgeführt wird. Die Bedingung der Zielvariablen ist deshalb die disjunktive Verknüpfung der Definiertheitsbedingungen der Quellvariablen, die zusätzlich konjunktiv mit der Ausführungsbedingung des Basic Blocks verknüpft wird. Destruct-Statements besitzen eine Kontrollvariable, die anzeigt, ob die Datenausgänge definiert sind oder nicht. Da die Kontrollvariable nur den Wert $\top$ haben kann, wenn das Statement auch ausgeführt wird, ist die Kontrollvariable die einzige Bedingung für alle Zielvariablen einschließlich der Kontrollvariablen selbst. Bei einem Start-Statement wird nur der erste Eingabeparameter gelesen. Deshalb sind die Zielvariablen definiert, wenn das Statement ausgeführt wird und die erste Eingabevariable definiert ist. Alle anderen Statements können mit der selben Regel verarbeitet werden: Ihre Zielvariablen sind nur dann definiert, wenn das Statement ausgeführt wird und alle Quellvariablen definiert sind. Die Definiertheitsbedingung der Variablen ist deshalb die konjunktive Verknüpfung der Ausführungsbedingung des Basic Blocks und der Definiertheitsbedingungen der Zielvariablen.

Wie auch in Abschnitt 4.2.3 behandelt der beschriebene Algorithmus den Init- und Loop-Teil zunächst getrennt voneinander. Auch hier lassen sich einfach die zusätzlichen Informationen über die Definiertheit der Zustandsvariablen, die in Abschnitt 4.2.3 gewonnen wurden, verwenden. Es wird eine Menge der wahren Literale eingeführt. Die wahren Literale enthalten genau die `Pre`-Variablen, die immer definiert sind, und sind positiv. Nach jeder Bedingungsberechnung kann die neu entstandene DNF vereinfacht und dabei alle wahren Literale aus den Konjunktionstermen entfernt werden. Das hat den gleichen Effekt wie die Initialisierung der entsprechenden Variablen als immer definiert. Bei der Bearbeitung des Init-Teils ist die Menge der wahren Literale leer. Für die Bearbeitung des Loop-Teils wird die Menge entsprechend der in Abschnitt 4.2.3 berechneten Definiertheit der Zustandsvariablen gefüllt.

6 Evaluation

Bei der Evaluation wird die Struktur des Kontrollflusses untersucht. Außerdem wird auch die Definiertheitsanalyse aus Kapitel 5 betrachtet.

Die Beispiele sah, sah2, rle, adsr, sine, triangle und rectangle wurden vorgegeben.

6.1 Synthese von Kontrollfluss (Kapitel 4)

Bei der Evaluation wird der Teil des jeweiligen Programms, für den Kontrollfluss erstellt werden konnte, als Kontrollflussgraph dargestellt. Anschließend wird der Graph mit dem SIG-Programm verglichen.

6.1.1 sah (Sample & Hold klassisch; 2 Eingänge)

Das Beispiel in Listing 30 zeigt eine klassische Sample & Hold Schaltung, die in SIG geschrieben ist.

```
1 sah := [
2   t : { sample | hold }, x : real -> y : real
3   where
4   y := case t of {
5     .sample -> x
6     .hold   -> 0.0 ; y
7   }
8 ]
```

Listing 30: SIG-Programm sah

Das Programm hat zwei Eingaben: der Parameter `t` bestimmt, ob der Parameter `x` zum Ausgang (`y`) durchgeleitet (sample) oder der letzte Wert wiederholt (hold) wird. Der Ausdruck `0.0 ; y` hat im ersten Schritt den Wert 0.0 (Initialisierung des Zustands) und in jedem weiteren Schritt den Ausgabewert des letzten Schritts.

In Listing 56 ist das Programm im Eingabeformat (Zwischenformat von SIG) dargestellt. Listing 57 zeigt es im Ausgabeformat nach der Anwendung der in dieser Arbeit vorgestellten Optimierung.

In Abbildung 10 ist der Kontrollflussgraph des relevanten Teils nach der Optimierung dargestellt.

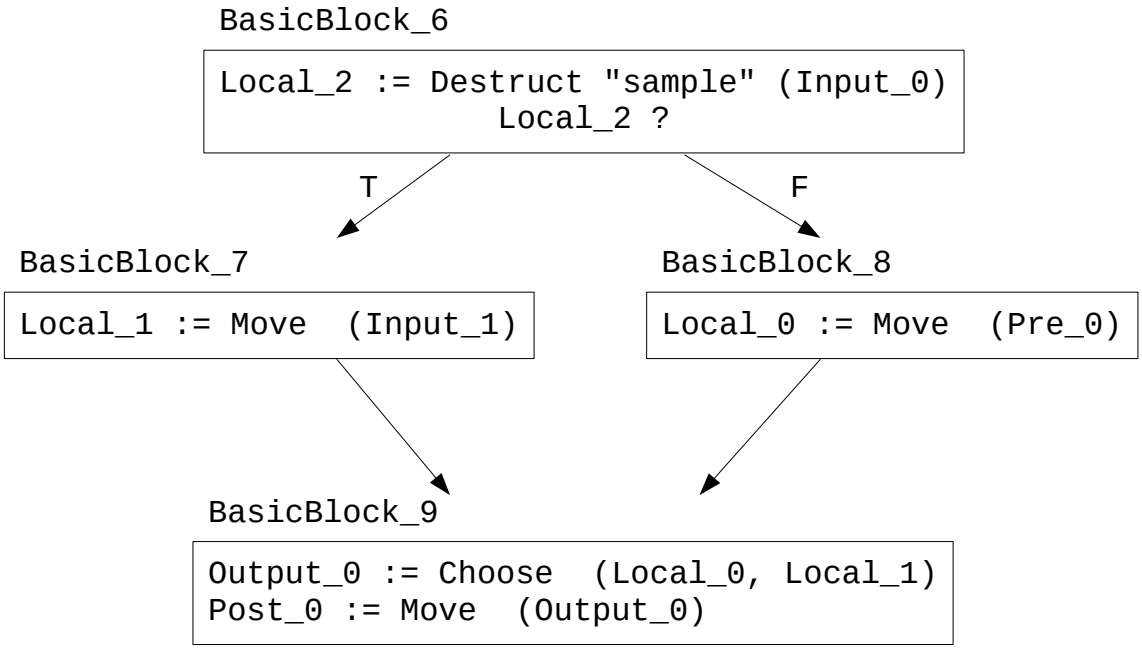

Abbildung 10: Kontrollflussgraph sah

Die zwei Fälle sample und hold sind in den Basic Blocks 7 und 8 zu erkennen. Für die Auswahl des Falls wird nur eine Bedingung geprüft – wenn im aktuellen Schritt nicht sample ausgeführt wird, muss es hold sein, eine weitere Bedingungsprüfung ist nicht notwendig.

6.1.2 sah2 (Sample & Hold algebraisch; 1 Eingang)

In Listing 31 ist eine weitere Variante der Sample & Hold Schaltung dargestellt.

```
1 sah2 := [
2   x : { sample(real) | hold } -> y : real
3   where
4   y := case x of {
5     .sample(z) -> z
6     .hold      -> 0.0 ; y
7   }
8 ]
```

Listing 31: SIG-Programm sah2

Bei dieser Variante wird der Eingabewert als Feld des sample-Falls des algebraischen Datentyps übergeben, der die Fälle sample bzw. hold auswählt.

Das Programm wird im Eingabeformat in Listing 58 und nach der Optimierung in Listing 59 gezeigt.

Der Kontrollflussgraph des optimierten Programms ist in Abbildung 11 dargestellt.

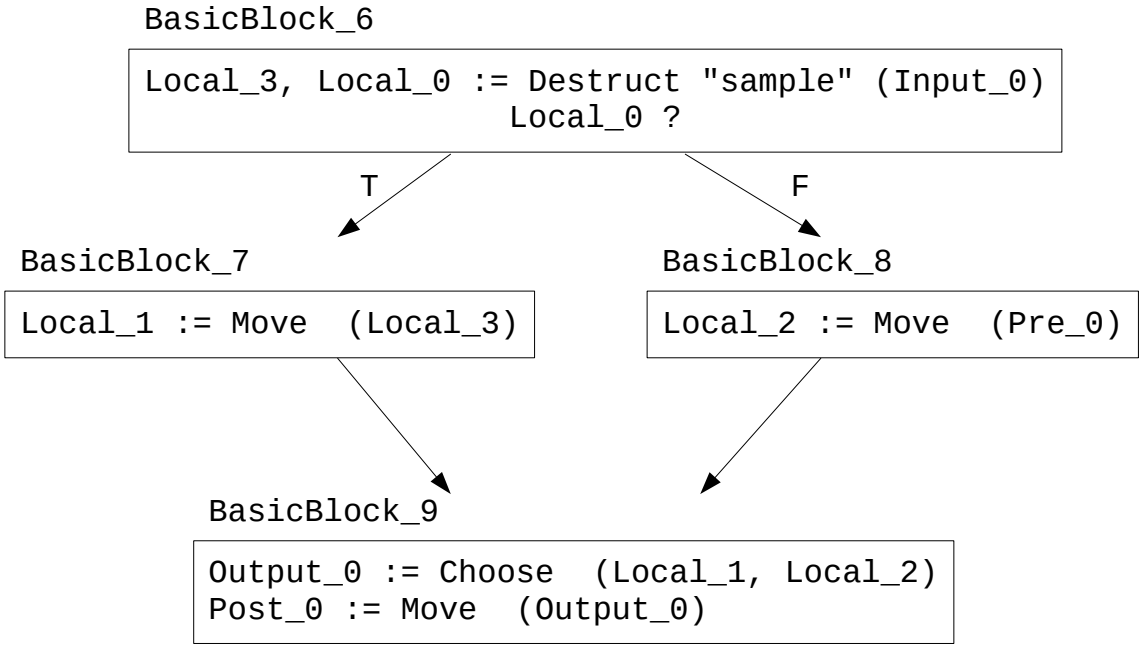

Abbildung 11: Kontrollflussgraph sah2-1

Auch in diesem Beispiel ist die Struktur des Programms gut erkennbar. Jede Anweisung wird nur ausgeführt, wenn sie auch benötigt wird. Die erste Anweisung in Basic Block 6 erzeugt nicht nur eine Kontrollvariable (`Local_0`), sondern packt auch das mit sample übergebene Datum aus.

Die Reihenfolge der Sprünge hängt in der aktuellen Implementierung von den Hash Codes der Sprungobjekte ab (siehe Abschnitt 4.12). Die Ausführlichkeit der während der Optimierung erstellten Debug-Ausgaben ist einstellbar. Die Reihenfolge der Sprünge ändert sich teilweise mit der Ausführlichkeit der Debug-Ausgaben – vermutlich weil die Anzahl der erzeugten String-Objekte Einfluss auf die von der JVM vergebenen Hash Codes der Sprungobjekte hat (die Sprungobjekte erben die Methode `hashCode()` von `Object` ohne sie zu überschreiben).

In Listing 60 und Abbildung 12 wird ein weiteres Ergebnis der Optimierung gezeigt, bei dem aufgrund anderer Ausführlichkeit der Debug-Ausgaben die Reihenfolge der Sprünge eine andere ist.

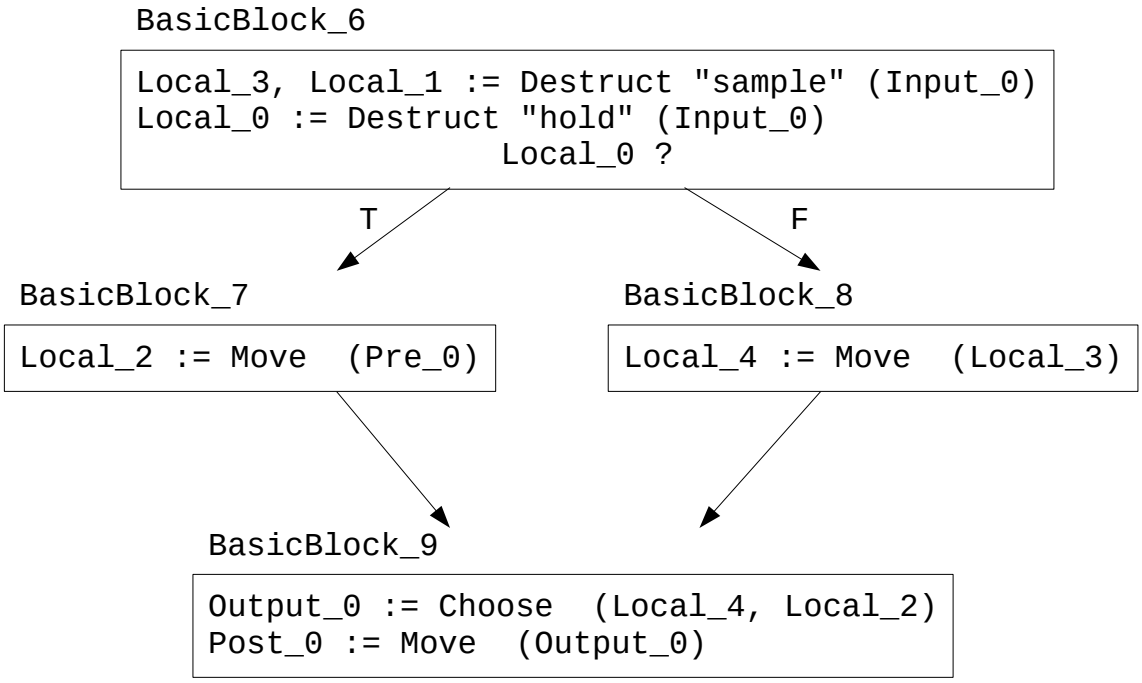

Abbildung 12: Kontrollflussgraph sah2-2

Die Struktur des Ausgangsprogramms ist wieder gut im Kontrollfluss abgebildet. Es fällt auf, dass die erste Anweisung in Basic Block 6 immer ausgeführt wird, obwohl sie nur im sample-Fall benötigt wird. Das liegt an der Berechnung der Kontrollvariablen `Local_1`. Wegen dieser Berechnung wird die Anweisung als immer benötigt bei der Analyse der Notwendigkeiten in Abschnitt 4.4 eingestuft. Da dieser Fall nur aufgetreten ist, wenn die Ausführlichkeit der Debug-Ausgaben niedrig ist, kann leider kein Zwischenergebnis zur Erklärung gezeigt werden. Vor Abschnitt 4.12 haben die Sprünge noch eine If-Then Semantik. Damit muss die Bedingung jedes Sprungs geprüft werden. Daraus folgt, dass zu diesem Zeitpunkt die Anweisung, die `Local_1` berechnet, immer ausgeführt werden muss. Erst in Abschnitt 4.12 werden die Sprünge so organisiert, dass der letzte Fall bei vollständigen Fallunterscheidungen implizit erfolgt. Da die Sprünge jedoch vorher erstellt werden, muss die Analyse der Notwendigkeit der Anweisungen auch vorher durchgeführt werden.

Eine mögliche Lösung betroffene Anweisungen trotzdem nur dann auszuführen, wenn sie notwendig sind, ist Partial Dead Code Elimination (PDCE). Da in dieser Arbeit bereits die Struktur des Kontrollflusses erstellt wurde, ist eine Änderung des Kontrollflusses durch PDCE nicht notwendig.

6.1.3 choose (Auswahl)

Ein weiteres Beispiel, in dem partiell toter Code auftritt, ist in Listing 32 dargestellt.

```
1 choose := [
2   t : { caseA | caseB | caseC }, x : real -> y : real
3   where
4   y := case t of {
5     .caseA -> -x
6     .caseB -> x + x
7     .caseC -> x * x
8   }
9 ]
```

Listing 32: SIG-Programm choose

Der Parameter `t` bestimmt, welche der drei Funktionen auf `x` angewendet wird.

Das Programm wird im Eingabeformat in Listing 61 und nach der Optimierung in Listing 62 gezeigt.

Der Kontrollflussgraph des optimierten Programms ist in Abbildung 13 dargestellt.

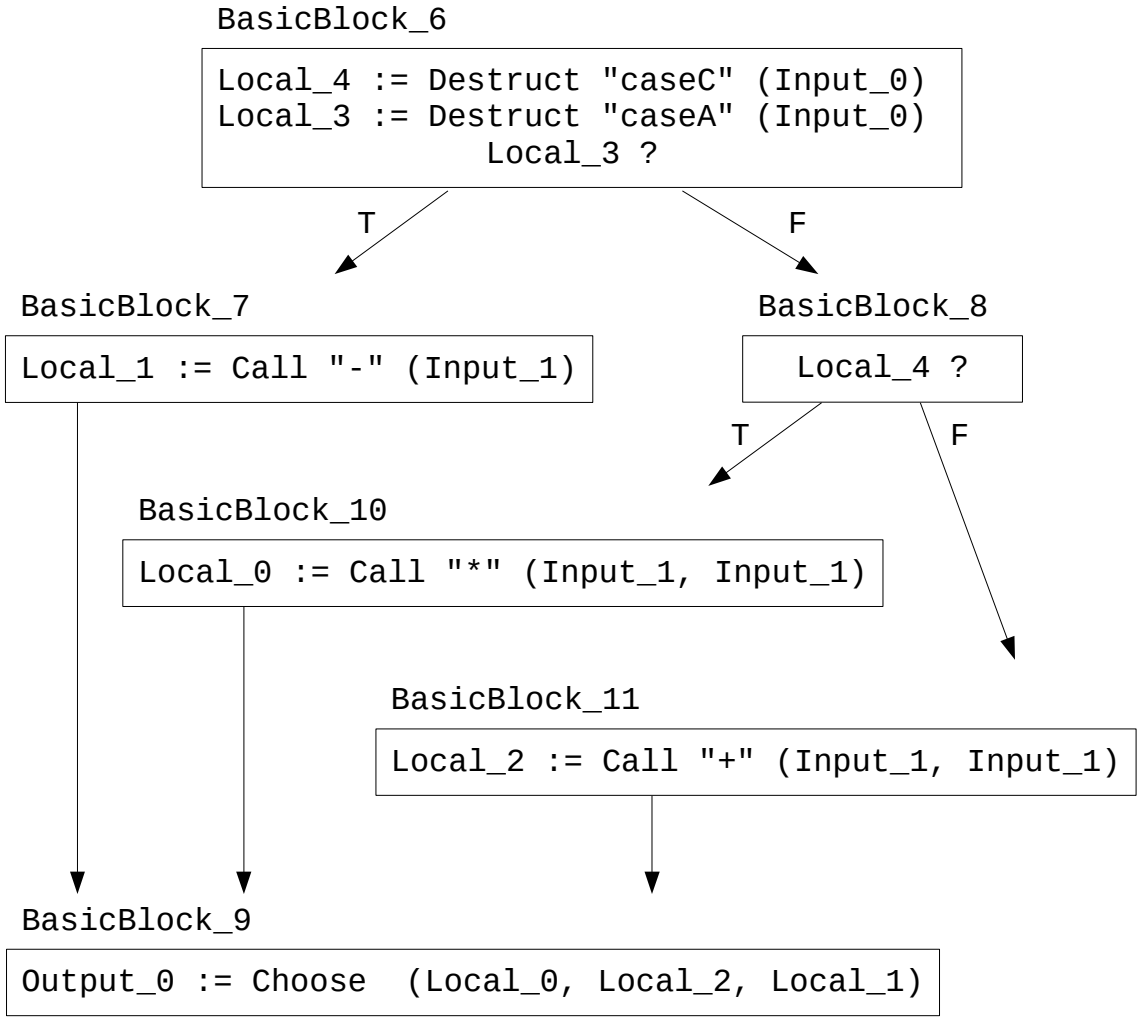

Abbildung 13: Kontrollflussgraph choose

Die erste Anweisung in Basic Block 6 wird nur in Basic Block 8 benötigt. In Listing 33 wird ein Zwischenergebnis der Optimierung gezeigt.

```
1  {
2    boolean Local_4 := Destruct "caseC" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_0 := Destruct "caseA" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_2 := Destruct "caseB" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
5    { ((Local_2))
6      double Local_3 := Call "+" (double Input_1, double Input_1)
7    } goto Label_0
8    label: Label_0
9    { ((Local_0))
10     double Local_1 := Call "-" (double Input_1)
11   } goto Label_1
12   label: Label_1
13   { ((Local_4))
14     double Local_5 := Call "*" (double Input_1, double Input_1)
15   } goto Label_2
16   label: Label_2
17   double Output_0 := Choose  (double Local_5, double Local_3, double
   Local_1)
18 }
```

Listing 33: choose direkt vor der Organisation der Sprünge (Abschnitt 4.12)

Dass der Destruktor von caseC nicht in Basic Block 8 ausgeführt wird, hat zwei Gründe. Zum einen wird wie in sah2-2 die Kontrollvariable `Local_4` zum Zeitpunkt der Analyse der Notwendigkeit der Variablen und Statements (Abschnitt 4.4) immer benötigt. Zum anderen haben die Sprünge nach der Organisation (Listing 34) eine If-Then-ElseIf-… Semantik.

```
1  {
2    boolean Local_4 := Destruct "caseC" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_0 := Destruct "caseA" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_2 := Destruct "caseB" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
5    label: Label_0
6    label: Label_1
7    label: Label_2
8    { ((Local_0))
9      double Local_1 := Call "-" (double Input_1)
10   } goto Label_3
11   { ((Local_4))
12     double Local_5 := Call "*" (double Input_1, double Input_1)
13   } goto Label_3
14   { ((true))
15     double Local_3 := Call "+" (double Input_1, double Input_1)
16   } goto Label_3
17   label: Label_3
18   double Output_0 := Choose  (double Local_5, double Local_3, double
   Local_1)
19 }
```

Listing 34: choose direkt nach der Organisation der Sprünge (Abschnitt 4.12)

Die If-Then-ElseIf-… Semantik der zusammengehörenden Sprünge (exklusive Sprungmengen) verbietet, dass zwischen den Sprüngen weitere Anweisungen stehen. Selbst wenn vor 4.12 die Destruktoren direkt vor den jeweiligen Sprüngen stehen würden, würde 4.12 die Sprünge so verschieben, dass zwischen ihnen keine anderen Anweisungen stehen. Damit wären die Destruktoren wieder vor den Sprüngen, wie es in Listing 34 dargestellt ist. Das Verschieben der Sprünge ist notwendig, damit keine Anweisungen übersprungen werden, die nicht übersprungen werden dürfen.

Als Lösung ist wie bei sah2-2 wieder PDCE geeignet, wobei auch hier die Kontrollflussstruktur nicht geändert werden muss.

6.1.4 rle (Run Length Encoding)

Ein weiteres Beispielprogramm ist in Listing 35 dargestellt.

```
1  rle := [
2    x : int -> y, n : int
3    -- Verzögerung x -> y um 1 Schritt
4    -- n gibt Länge des eben abgeschlossenen Runs an;
5    --   n=0 falls noch laufend
6    -- Beispiel:
7    --  x = 6 6 6 5 4 4 3 ...
8    --  y = 0 6 6 6 5 4 4 ...
9    --  n = 0 0 0 3 1 0 2 ...
10 where
11   k : int               -- aktueller run
12   y := 0 ; x
13   k := 0 ; if x = y then k + 1 else 1
14   n :=     if x = y then 0     else k
15 ]
```

Listing 35: SIG-Programm rle

Der Fallweise (nicht) benötigte Code wird in diesem Beispiel nicht durch eine Fallauswahl mit einem algebraischen Datentyp bestimmte, sondern durch zwei If-Anweisungen in den Zeilen 13 und 14. Das Ergebnis des Vergleichs `x = y` hat den Datentyp

```
bool = { T | F }
```

Das Programm wird im Eingabeformat in Listing 63 und nach der Optimierung in Listing 64 gezeigt.

Der Kontrollflussgraph des optimierten Programms ist in Abbildung 14 dargestellt.

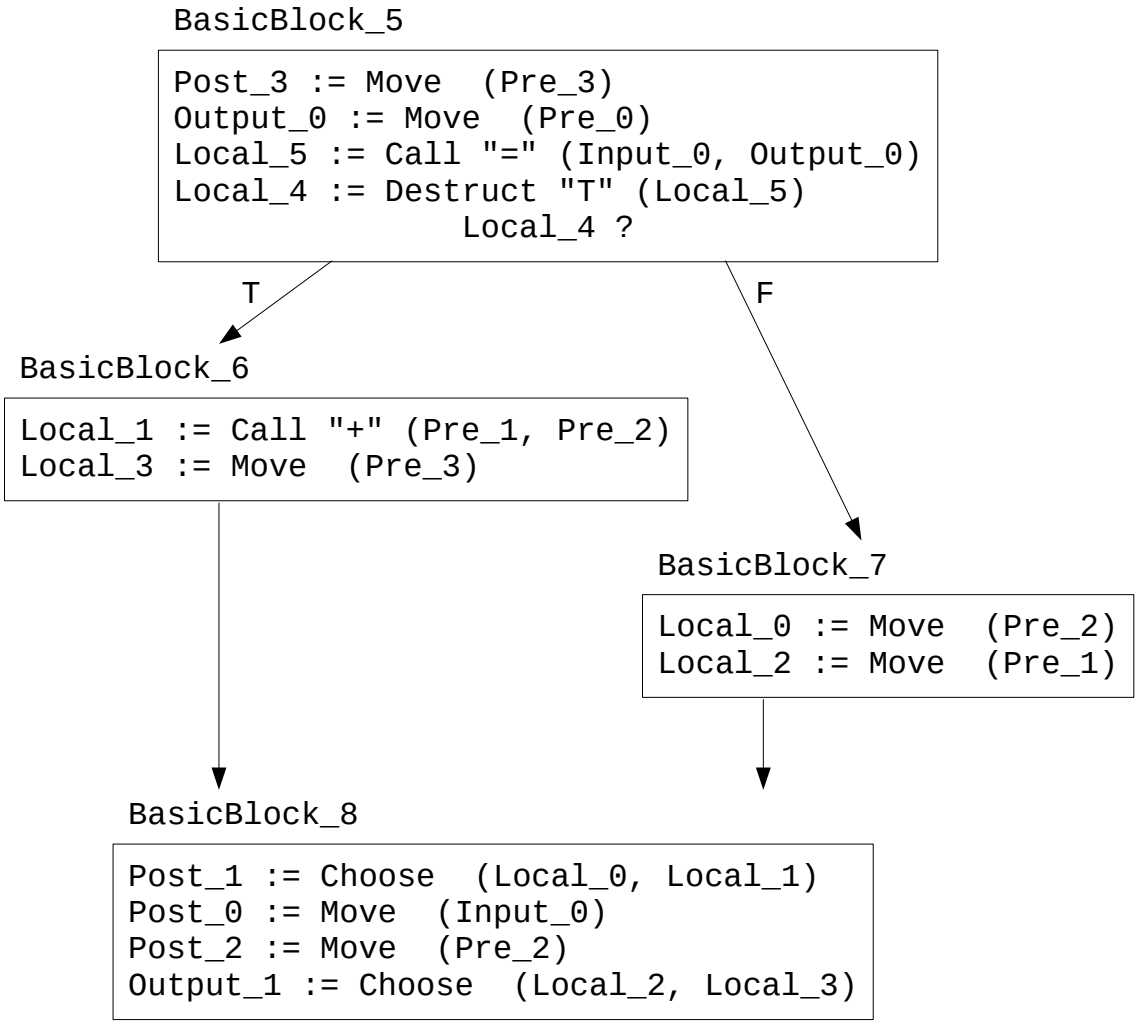

Abbildung 14: Kontrollflussgraph rle

Im Gegensatz zum Quellcode ist im Kontrollfluss nur eine Fallunterscheidung zu sehen. Im Quellcode hatten beide If-Anweisungen die gleiche Bedingung. Deshalb konnten sie zusammengefasst werden. Das ist auch im Kontrollflussgraphen in den Basic Blocks 6 und 7 erkennbar – in den Basic Blocks werden jeweils zwei Anweisungen ausgeführt: eine Anweisung je If-Anweisung im SIG-Programm.

Die Variable `Local_5` wird indirekt auch aus `Pre_0` berechnet. Da Zustandsvariablen auch den Wert ⊥ annehmen können [WL14a, Seite 153], kann im Allgemeinen auch `Local_5` ⊥ sein. Damit wäre die Fallunterscheidung nicht vollständig, weil alle durch Destruktoren entstandenen Kontrollvariablen ⊥ sein könnten, und der letzte Fall könnte nicht implizit angenommen werden. Da die Definiertheitsanalyse in Abschnitt 4.2.3 jedoch die Informationen aus Init- und Loop-Teil kombiniert, wurde erkannt, dass im Loop-Teil `Pre_0` immer definiert ist und damit auch `Local_5` niemals den Wert ⊥ haben kann. Damit ist die Fallunterscheidung vollständig.

6.1.5 adsr (Attack/Decay/Sustain/Release Hüllkurvengenerator)

Das in Listing 36 dargestellte SIG-Programm ist ein Attack-Decay-Sustain-Release Modell. Es wird bei der Synthese von Musik eingesetzt, um die Hüllkurve der Amplitude für virtuelle Musikinstrumente zu erstellen [WL14a, Seite 145].

```
1  adsr := {
2    -- Äusserer Block ist Konfigurator;
3    -- innerer Block ist eigentlicher Automat
4    a, d : #[real -> real], s : #[-> real], r : #[real -> real] ->
5    #[ gate : bool -> out' : real
6       where
7       out : real
8       out := 0.0 ; out'
9       state, state' : { Attack | Decay | Sustain | Release }
10      state  := .Release ; state'
11      state' := case state of {
12        .Release if  gate                -> .Attack
13        .Attack  if  gate && out' >= 1.0 -> .Decay
14        .Decay   if  gate && out' <= $s  -> .Sustain
15      else
16        _        if !gate                -> .Release
17      else
18        _                                -> state
19      }
20      out' := case state of {
21        .Attack  -> min(1.0, out + $a(out))
22        .Decay   -> max($s,  out - $d(out))
23        .Sustain ->          out
24        .Release -> max(0.0, out - $r(out))
25      }
26   ]
27 }
```

Listing 36: SIG-Programm adsr

Das Programm wird im Eingabeformat in Listing 65 und nach der Optimierung in Listing 66 gezeigt.

Das optimierte Programm ist zu groß, um seinen Kontrollflussgraph hier graphisch darzustellen. Deswegen wird es als Pseudocode mit If-Then-ElseIf-… Anweisungen zur Verdeutlichung der Struktur in Listing 37 dargestellt. Darin sind als Kommentare die Basic Blocks angegeben, die den jeweiligen Code enthalten, und Informationen die es erleichtern die Verbindung zum SIG Quelltext zu sehen.

```
1  /* BasicBlock_17 */
2  Local_27, Post_3 := Sample  (Pre_14, Pre_3, Pre_0)
3  Local_29, Post_4 := Sample  (Pre_7, Pre_4, Pre_0)
4  Local_17 := Destruct "Release" (Pre_1)  // Pre_1 : state,
                                              Local_17 : isRelease
5  Local_15 := Destruct "Decay" (Pre_1)  // Pre_1 : state,
                                            Local_15 : isDecay
6  Local_18 := Destruct "Attack" (Pre_1)  // Pre_1 : state,
                                              Local_18 : isAttack
7  Local_21 := Destruct "T" (Input_0)  // Input_0 : gate,
                                          Local_21 : isGate
8  Local_22 := Destruct "F" (Input_0)  // Input_0 : gate,
                                          Local_22 : isNotGate
9  IF (Local_22) {  // isNotGate
10   /* BasicBlock_18 */
11   Local_9 := Move  (Pre_19) // Pre_19 = Release
12 }
13 /* BasicBlock_19 */
14 Local_33, Post_5 := Sample  (Pre_16, Pre_5)
15 Local_23, Post_6 := Sample  (Pre_12, Pre_6, Pre_0)
16 Local_19 := Destruct "Sustain" (Pre_1)  // Pre_1 : state,
                                              Local_19 : isSustain
17 IF (Local_19) {  // isSustain
18   /* BasicBlock_20 */
19   Local_31 := Move  (Pre_0)
20   IF (Local_21) {  // isGate
21     /* BasicBlock_22 */
22     Local_11 := Move  (Pre_1)  // Pre_1 : state
23   }
24   /* BasicBlock_23 */
25   Local_4 := Choose  (Local_9, Local_11)  // Release or state
26 } ELSE /* BasicBlock_21 */ IF (Local_17) {  // isRelease
27   /* BasicBlock_24 */
28   IF (Local_21) {  // isGate
29     /* BasicBlock_27 */
30     Local_2 := Move  (Pre_10)  // Pre_10 : Attack
31   }
32   /* BasicBlock_28 */
33   Local_5 := Choose  (Local_9, Local_2)  // Release or Attack
34   Local_28 := Call "-" (Pre_0, Local_29)
35   Local_34 := Call "max" (Pre_20, Local_28)
36 } ELSE /* BasicBlock_25 */ IF (Local_15) {  // isDecay
37   /* BasicBlock_29 */
38   Local_32 := Call "-" (Pre_0, Local_23)
39   Local_25 := Call "max" (Local_33, Local_32)
40 } ELSE {
41   /* BasicBlock_30*/
42   Local_26 := Call "+" (Pre_0, Local_27)
43   Local_24 := Call "min" (Pre_18, Local_26)
44 }
45 /* BasicBlock_26 */
46 Output_0 := Choose  (Local_24, Local_34, Local_25, Local_31)
                                                    // Output_0 : out'
47 Local_30, Post_2 := Sample  (Pre_16, Pre_2)
48 Post_15 := Move  (Pre_15)
49 Post_17 := Move  (Pre_17)
```

```
50  Post_10 := Move  (Pre_10)  // Pre_10 = Attack
51  Post_9 := Move  (Pre_9)  // Pre_9 = Decay
52  Post_18 := Move  (Pre_18)
53  IF (Local_15) {  // isDecay
54    /* BasicBlock_31 */
55    IF (Local_21) {  // isGate
56      /* BasicBlock_33 */
57      Local_0 := Call "<=" (Output_0, Local_30)
58      Local_20 := Destruct "F" (Local_0)
59      IF (Local_20) {
60        /* BasicBlock_37 */
61        Local_7 := Move  (Pre_1)  // Pre_1 : state
62      } ELSE {
63        /* BasicBlock_38 */
64        Local_6 := Move  (Pre_11)  // Pre_11 = Sustain
65      }
66      /* BAsicBlock_41 */
67      Local_14 := Choose  (Local_7, Local_6)  // state or Sustain
68    }
69    /* BasicBlock_34 */
70    Local_3 := Choose  (Local_9, Local_14)  // Release or state
                                                 or Sustain
71  } ELSE /* BasicBlock_32 */ IF (Local_18) {  // isAttack
72    /* BasicBlock_35 */
73    IF (Local_21) {  // isGate
74      /* BasicBlock_39 */
75      Local_1 := Call ">=" (Output_0, Pre_18)
76      Local_16 := Destruct "T" (Local_1)
77      IF (Local_16) {
78        /* BasicBlock_42 */
79        Local_13 := Move  (Pre_9)  // Pre_9 = Decay
80      } ELSE {
81        /* BasicBlock_43 */
82        Local_10 := Move  (Pre_1)  // Pre_1 : state
83      }
84      /* BasicBlock_44 */
85      Local_12 := Choose  (Local_10, Local_13)  // state or Decay
86    }
87    /* BasicBlock_40 */
88    Local_8 := Choose  (Local_9, Local_12)  // Release or state
                                                 or Decay
89  }
90  /* BasicBlock_36 */
91  Post_1 := Choose  (Local_8, Local_5, Local_3, Local_4)  // state'
92  Post_14 := Move  (Pre_14)
93  Post_20 := Move  (Pre_20)
94  Post_19 := Move  (Pre_19)  // Pre_19 = Release
95  Post_16 := Move  (Pre_16)
96  Post_12 := Move  (Pre_12)
97  Post_7 := Move  (Pre_7)
98  Post_8 := Move  (Pre_8)
99  Post_0 := Move  (Output_0) // Output_0 : out'
100 Post_11 := Move  (Pre_11)  // Pre_11 = Sustain
101 Post_13 := Move  (Pre_13)
```

Listing 37: Pseudocode Kontrollfluss adsr

Es fällt auf, dass die `Sample`-Anweisungen immer ausgeführt werden. Das liegt daran, dass sie `Post`-Variablen schreiben. Dieses Verhalten ist notwendig, wenn die Unterfunktionen einen Zustand haben, um diesen Zustand zu aktualisieren, auch wenn der zurückgegebene Wert aktuell nicht benötigt wird.

Wenn `gate` deaktiviert (falsch) ist, ist der neue Zustand `state'` immer Release (Listing 36 Zeile 16). Dementsprechend wird der Code in Listing 37 Zeile 11 immer benötigt, wenn `gate` deaktiviert ist. Im ersten großen If-Then-ElseIf-… Block in Listing 37 Zeilen 17 – 44 wird der Ausgabewert `out'` berechnet, was den Zeilen 20 – 25 in Listing 36 entspricht. Außerdem wird im selben Block der neue Zustand `state'` berechnet, wenn der aktuelle Zustand Sustain oder Release ist, was den Zeilen 12, 16 und 18 in Listing 36 entspricht. Der neue Zustand wird in Listing 37 Zeilen 53 – 89 berechnet, wenn der aktuelle Zustand Decay oder Attack ist, weil dafür der Ausgabewert `out'` benötigt wird, der erst in Listing 37 Zeile 46 verfügbar ist. Das entspricht den Zeilen 13, 14, 16 und 18 in Listing 36.

Weiterhin ist zu sehen, dass bei vollständigen Fallunterscheidungen (alle Fälle des algebraischen Datentyps werden in einem If-Then-ElseIf-… Block abgefragt) der letzte Fall nur implizit in einem Else-Zweig auftritt, anstatt die Bedingung, die an dieser Stelle nur wahr sein kann, zur Laufzeit zu prüfen.

6.1.6 triangle (Dreieck Wellenform-Generator)

Als nächstes Beispiel ist in Listing 38 ein Wellenform-Generator dargestellt.

```
1  triangle := {
2    eps : real ->
3    #[ -> out : real
4    where
5      out', step, step', tmp : real
6      over : bool
7      out   := 0.0 ; out'
8      step  := 1.0 ; step'                  -- immer +1.0 (auf) oder
                                                -1.0 (ab)
9      tmp   := out + step * eps / 1.570796  -- neuer Wert bis auf
                                                Überlauf
10     over  := step * tmp >= 1.0            -- Überlauf?
11     out'  := if over then 2.0 * step - tmp else tmp
12     step' := if over then -step            else step
13   ]
14 }
```

Listing 38: SIG-Programm triangle

Das Programm wird im Eingabeformat in Listing 67 und nach der Optimierung in Listing 68 gezeigt.

Der Kontrollflussgraph des optimierten Programms ist in Abbildung 15 dargestellt.

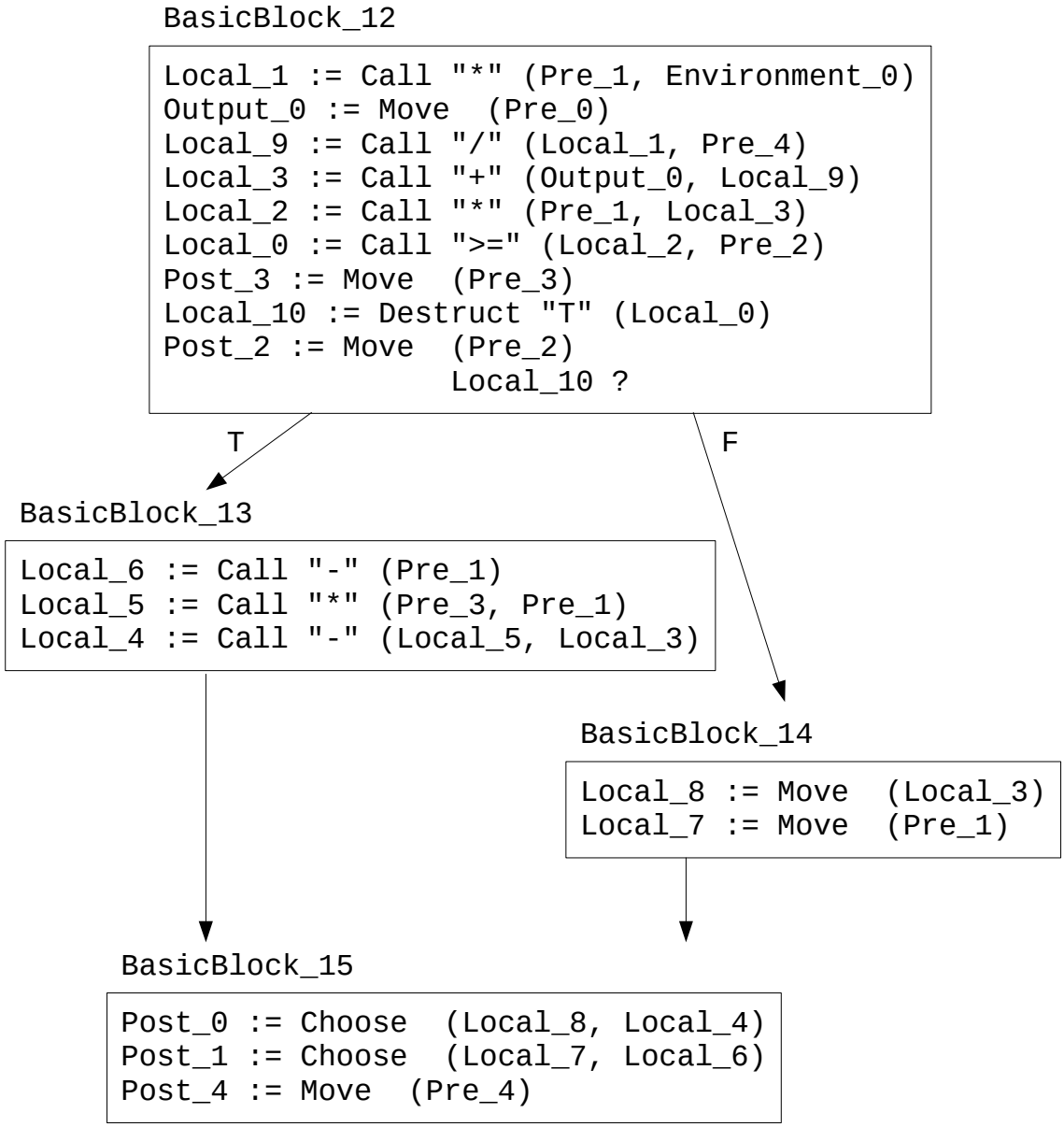

Abbildung 15: Kontrollflussgraph triangle

Wie auch beim Programm rle wurden die beiden If-Anweisungen mit der gleichen Bedingung zusammengefasst und auch erkannt, dass die `Pre`-Variablen, die in die Berechnung von `Local_10` eingehen, immer definiert sind.

6.1.7 rectangle (Rechteck Wellenform-Generator)

Das letzte Beispiel ist auch ein Wellenform-Generator und wird in Listing 39 gezeigt.

```
1  rectangle := {
2    eps : real ->
3    #[ -> out : real
4    where
5      out', phase, phase', tmp : real
6      over : bool
7      out    := 1.0 ; out'              -- immer +1.0 oder -1.0
8      phase  := 0.0 ; phase'
9      tmp    := phase + eps / 1.570796  -- neuer Wert bis auf Überlauf
10     over   := tmp >= 1.0              -- Überlauf?
11     out'   := if over then -out      else out
12     phase' := if over then tmp - 1.0 else tmp
13   ]
14 }
```

Listing 39: SIG-Programm rectangle

Das Programm wird im Eingabeformat in Listing 69 und nach der Optimierung in Listing 70 gezeigt.

Der Kontrollflussgraph des optimierten Programms ist in Abbildung 16 dargestellt.

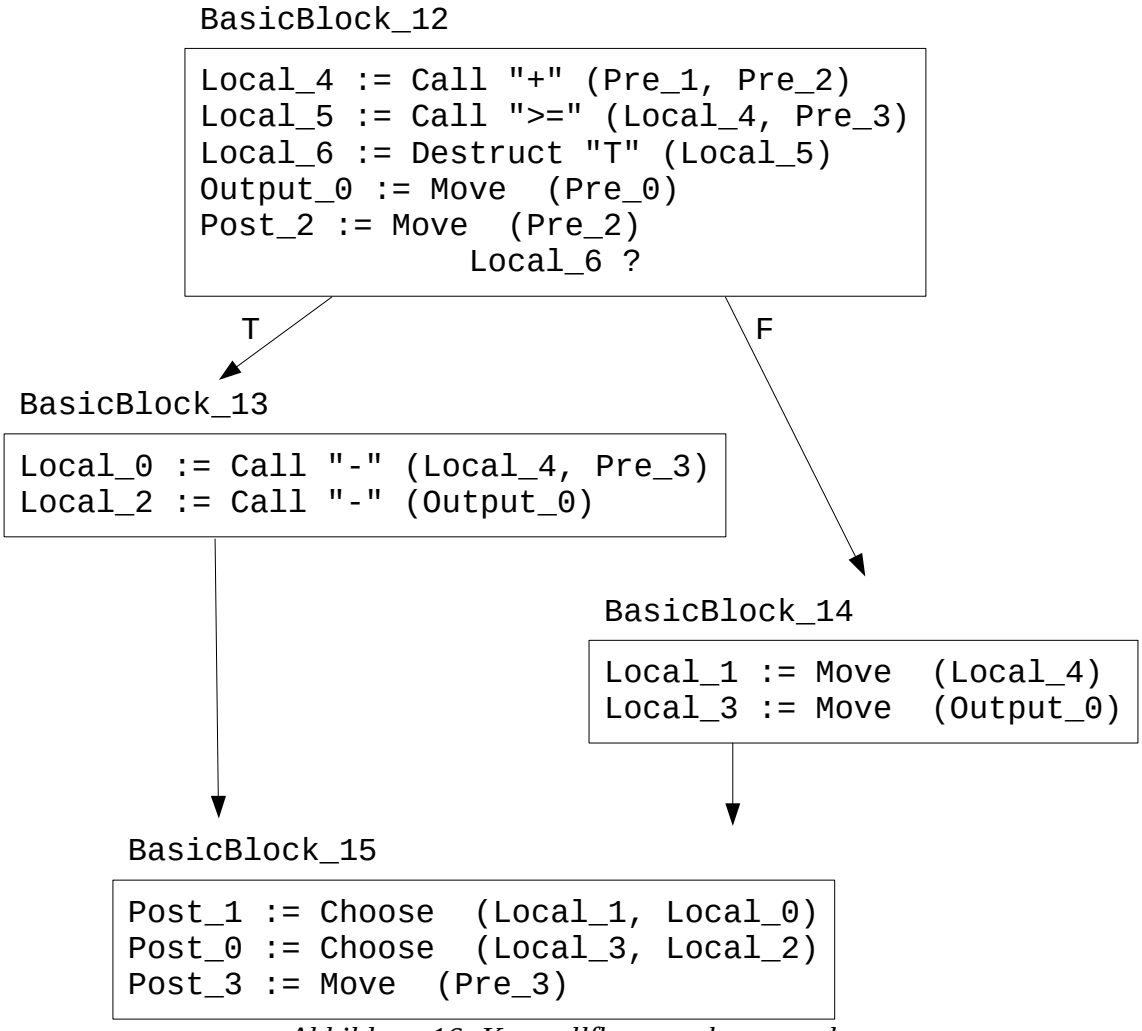

Abbildung 16: Kontrollflussgraph rectangle

Wie auch bei den Programmen rle und triangle wurden die beiden If-Anweisungen mit der gleichen Bedingung zusammengefasst und auch erkannt, dass die `Pre`-Variablen, die in die Berechnung von `Local_6` eingehen, immer definiert sind.

6.1.8 Zusammenfassung

Der generierte Kontrollfluss bildet die Struktur der Beispiele gut ab. Bei mehreren Fallunterscheidungen mit der gleichen Bedingung wurde im Kontrollfluss nur eine Fallunterscheidung generiert, wenn der Datenfluss es zugelassen hat. Außerdem erfolgt die letzte Bedingungsprüfung bei vollständigen Fallunterscheidungen implizit durch Verwendung des Else-Zweigs.

In den Beispielen sind auch nicht optimale Ergebnisse zu finden. Ein Grund dafür ist die frühe Berechnung unter welchen Bedingungen eine Anweisung ausgeführt wird (Beispiel sah2-2). Ein weiterer Grund ist die If-Then-ElseIf-… Semantik, die den angewendeten Transformationen zugrunde liegt (Beispiel choose). Mittels Partial Dead Code Elimination (ohne Änderung des Kontrollflusses) können jedoch die negativen Effekte entfernt werden.

6.2 Definiertheitsanalyse (Kapitel 5)

Für die Evaluation der Definiertheitsanalyse wird wieder das Beispiel sah2 (6.1.2) verwendet. Dabei werden beide Ergebnisse (sah2-1 und sah2-2) betrachtet.

6.2.1 sah2-1

Die Bedingungen, unter denen die Variablen immer definiert sind bzw. die Basic Blocks ausgeführt werden, werden in Listing 40 gezeigt.

```
1  variables {
2    double Output_0 --> (())          isTautology: true
3    AlgebraicDataType_0 Input_0 --> (())  isTautology: true
4    double Post_0   --> (())          isTautology: true
5    double Local_1  --> ((Local_0))   isTautology: false
6    boolean Local_0 --> ((Local_0))   isTautology: false
7    double Local_3  --> ((Local_0))   isTautology: false
8    double Pre_0    --> (())          isTautology: true
9    double Local_2  --> ((!Local_0))  isTautology: false
10 }
11 basic blocks {
12   BasicBlock_7 --> ((Local_0))   isTautology: false
13   BasicBlock_9 --> (())          isTautology: true
14   BasicBlock_8 --> ((!Local_0))  isTautology: false
15   BasicBlock_6 --> (())          isTautology: true
16 }
```

Listing 40: Ergebnis Definiertheitsanalyse sah2-1

Die Ergebnisse sind wie erwartet. Durch die Verwendung von Informationen über die Definiertheit der Zustandsvariablen aus Abschnitt 4.2.3 wurde auch erkannt, dass die Variable `Pre_0` immer definiert ist.

6.2.2 sah2-2

Die Bedingungen, unter denen die Variablen immer definiert sind bzw. die Basic Blocks ausgeführt werden, werden in Listing 41 gezeigt.

```
1  variables {
2    boolean Local_1 --> ((Local_1))  isTautology: false
3    double Pre_0    --> (())         isTautology: true
4    AlgebraicDataType_0 Input_0  --> (())         isTautology: true
5    double Local_4  --> ((!Local_0 AND Local_1))  isTautology: false
6    double Post_0   --> ((Local_0) OR (!Local_0 AND Local_1))
                                      isTautology: false
7    double Output_0 --> ((Local_0) OR (!Local_0 AND Local_1))
                                      isTautology: false
8    double Local_3  --> ((Local_1))  isTautology: false
9    double Local_2  --> ((Local_0))  isTautology: false
10   boolean Local_0 --> ((Local_0))  isTautology: false
11 }
12 basic blocks {
13   BasicBlock_8 --> ((!Local_0))  isTautology: false
14   BasicBlock_9 --> (())          isTautology: true
15   BasicBlock_6 --> (())          isTautology: true
16   BasicBlock_7 --> ((Local_0))   isTautology: false
17 }
```

Listing 41: Ergebnis Definiertheitsanalyse sah2-2

Das Ergebnis ist nicht ganz das erwartete. Wieder wird erkannt, dass die Variable `Pre_0` immer definiert ist, weil die Information aus 4.2.3 übernommen wurde. Dagegen wird angezeigt, dass die Variable `Post_0` nicht immer definiert ist. Die Definiertheitsbedingung von `Post_0` wird neu berechnet und nicht aus 4.2.3 übernommen. Deswegen ist der (scheinbare) Widerspruch zwischen den Definiertheitsbedingungen von `Pre_0` und `Post_0` möglich.

`Local_1` zeigt an, dass der Destruktor von sample erfolgreich war. Daraus folgt auch, dass `Local_3` nur definiert ist, wenn `Local_1` wahr ist. `Local_0` zeigt an, dass der Destruktor von hold erfolgreich war. Daraus folgt `Local_1 = !Local_0`. Dieser Zusammenhang wird bei der Definiertheitsanalyse jedoch weder berechnet noch genutzt. Wenn er berücksichtigt werden würde, ergäbe sich

```
((!Local_0 AND Local_1))
= ((!Local_0))
```

und

```
((Local_0) OR (!Local_0 AND Local_1))
= ((Local_0) OR (!Local_0))
= (true)
```

was in das Ergebnis der Definiertheitsanalyse eingesetzt das erwartete Ergebnis ergibt.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die synchrone Datenflusssprache SIG unterstützt über spezielle Konstrukte (γ- und φ-Knoten) Fallunterscheidungen. Durch die in dieser Arbeit durchgeführten Optimierungen können Berechnungen, deren Ergebnisse im konkreten Fall nicht benötigt werden, übersprungen werden.

Die notwendigen Analysen und Transformationen, die in Kapitel 4 beschrieben wurden, werden nicht in einem monolithischen Algorithmus bzw. Verfahren durchgeführt, sondern laufen in mehreren voneinander abhängigen Phasen ab. Neben der Erstellung der Sprünge werden diese auch verschachtelt. Die Verarbeitung der Bedingungen ist ein weiterer wichtiger Teil dieser Arbeit. Dabei wird der Code, der die Kontrollvariablen der Bedingungen erzeugt, so verschoben, dass möglichst viele Variablen für die Auswertung der Bedingungen verfügbar sind. Außerdem werden die Bedingungen vereinfacht, wobei bei den Vereinfachungsregeln darauf geachtet werden musste, dass eine Negation der Variablen nicht möglich ist. Sprünge mit den gleichen Sprungbedingungen werden, sofern der Daten- und der Kontrollfluss es erlauben, zusammengefasst. Weiterhin werden die Sprünge, die eine If-Then Semantik haben, so organisiert, dass sie einer If-Then-ElseIf-… Semantik entsprechen. Dabei wird, sofern möglich, auch die Laufzeitprüfung der Bedingung des letzten Sprungs ausgelassen, vergleichbar mit einem Else-Zweig, der ausgeführt wird, wenn alle anderen Bedingungen nicht zugetroffen haben. Bei den Transformationen und der Generierung des Codes zur Bedingungsprüfung können unerwünschte Effekte entstehen. Um diese zu eliminieren werden Copy Propagation, Dead Code Elimination und eine modifizierte Variante von Common Subexpression Elimination eingesetzt. Bei der Erstellung der Basic Blocks wird darauf geachtet, dass möglichst wenige erstellt werden. Als letzter Schritt in Kapitel 4 wird die Anzahl der deklarierten lokalen Variablen auf das Notwendige reduziert.

Nicht alle Operationen in SIG erlauben den speziellen Wert $\perp$, der anzeigt, dass eine Variable nicht definiert ist, als Eingabe. Deshalb muss zur Laufzeit sichergestellt werden, dass die Operanden der Operationen, die $\perp$-Werte nicht als Eingabe unterstützen, nicht mit diesen Werten aufgerufen werden. In Kapitel 5 wird die Prüfung auf den Wert $\perp$ vorbereitet. Das Ziel ist dabei die Berechnung, ob eine Variable an einem bestimmten Programmpunkt den Wert $\perp$ annehmen kann und somit eine Laufzeitprüfung erforderlich ist oder nicht.

Die Evaluation zeigt, dass der generierte Kontrollfluss die Struktur des SIG-Programms gut abbildet und damit das Ausführen von fallweise nicht benötigtem Code zum großen Teil verhindert wird. Die Ergebnisse sind jedoch teilweise nicht optimal, was dazu führt, dass Code ausgeführt wird, der im konkreten Fall nicht benötigt wird. Durch die Anwendung von Partial Dead Code Elimination (ohne Änderung des Kontrollflusses) kann auch diese fallweise unnötige Codeausführung verhindert werden. Weiterhin zeigt die Evaluation, dass die Definiertheitsanalyse grundsätzlich funktioniert und korrekte Ergebnisse liefert. Dabei werden jedoch logische Zusammenhänge mehrerer Variablen und damit bestimmte Tautologien nicht erkannt.

7.2 Ausblick

Bei der Berechnung der exklusiven Variablenmengen (Abschnitt 4.2.3) wäre es wünschenswert, wenn für `Pre`-Variablen, die Kontrollvariablen sind, die Zugehörigkeit zu exklusiven Variablenmengen berechnet werden würde. Dabei muss bei der Verarbeitung des Loop-Teils der Init-Teil berücksichtigt werden.

In den Abschnitten 4.7 und 4.8 werden Def-Use Ketten verwendet, damit beim Verschieben von Code keine Datenflussabhängigkeiten verletzt werden. Später werden in Abschnitt 4.9 die Sprungbedingungen vereinfacht und damit eventuell Variablen aus den Bedingungen entfernt. Wenn die Definition einer solchen entfernten Variablen eine Codeverschiebung blockiert hat, könnte die Verschiebung nach dem Entfernen der Variablen möglich sein. Da bei der Vereinfachung der Bedingung (noch) nicht definierte Variablen aus den Bedingungen entfernt werden, ist der Einsatz von 4.9 vor 4.7 oder 4.8 nicht sinnvoll. Fraglich ist, ob in der Praxis solche Probleme auftreten und ob eine Vereinfachung der Bedingungen ohne Entfernung (noch) nicht definierter Variablen vor 4.7 das Problem lösen kann. Dabei müsste auch darauf geachtet werden, dass durch das Zurückrechnen der Kontrollvariablen auch die Def-Use Ketten manipuliert wurden.

Bei der Organisation der Sprünge (Abschnitt 4.12) werden unter Umständen Bedingungen zu Tautologien geändert. Damit werden die in der Bedingung vorkommenden Variablen an dieser Stelle nicht mehr benutzt und es kann toter Code entstehen. Toter Code wird in Abschnitt 4.13 entfernt. Die Reihenfolge der Sprünge bei der Organisation hängt in der aktuellen Implementierung vom Hash Code der Sprungobjekte ab und ist deshalb für den Programmierer unvorhersehbar. Damit ist auch unkontrollierbar, welche Variablen nicht mehr benutzt werden. Durch eine koordinierte Vorgehensweise und einer geschickten Auswahl, welche Bedingung zu einer Tautologie geändert wird, kann mehr Code zu totem Code werden.

Die Verfahren zur Vereinfachung boolescher Ausdrücke sind nicht vollständig. Es gibt Ausdrücke, die weiter vereinfacht werden können, was jedoch von den ausgewählten und implementierten Verfahren nicht erreicht wird. Ein Beispiel dafür ist in Listing 22 zu sehen. Die Bedingungen der beiden Sprünge sind vollständig. Damit kann die Bedingung des zweiten Sprungs zu einer Tautologie geändert werden. Dadurch, dass die Verfahren zur Vereinfachung jedoch nicht vollständig sind, wurde bei der Organisation der Sprünge die Bedingung

```
((Local_2 AND Local_0) OR (Local_3) OR (Local_1))
```

nicht zu einer Tautologie vereinfacht und damit nicht erkannt, dass die beiden Sprungbedingungen vollständig sind. Weitere Verfahren zur Vereinfachung boolescher Ausdrücke könnten eventuell auch diesen und weitere möglicherweise existierende noch nicht abgedeckte Fälle abdecken.

In Abschnitt 4.9 werden die wahren Variablen aus dem Kontext verwendet, um die Sprungbedingungen zu vereinfachen. Mit weiteren Verfahren zur Vereinfachung könnten eventuell nicht nur die wahren Variablen, sondern ganze Sprungbedingungen aus dem Kontext zur Vereinfachung verwendet werden.

Zur Eliminierung von Redundanzen wird eine modifizierte CSE eingesetzt. Mächtigere Verfahren, wie zum Beispiel PRE, könnten bessere Ergebnisse berechnen.

Sprungbedingungen sind DNFs, die beliebig komplex werden können. In den Abschnitten 4.12 und 4.14 werden die Bedingungen jeweils als eine Einheit betrachtet. Dadurch entstehen flache Strukturen. Durch die Zerlegung der Bedingungen in Teilbedingungen, die nacheinander Ausgewertet werden, könnten tiefere Strukturen entstehen, die den Aufwand für die Auswertung der Bedingungen verringern und teilweise Code nicht mehr benötigen. Das folgende Beispiel (Listing 42) zeigt das Programm in Listing 55 nach der Anwendung von 4.11.

```
1  {
2    boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
3    boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
4    boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
5    boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
6    { ((Local_2 AND Local_0))
7      double Local_4 := Call "f1" (double Input_2)
8    } goto Label_0
9    label: Label_0
10   { ((Local_3 AND Local_0))
11     double Local_5 := Call "f2" (double Input_2)
12   } goto Label_1
13   label: Label_1
14   { ((Local_2 AND Local_1))
15     double Local_6 := Call "f3" (double Input_2)
16   } goto Label_2
17   label: Label_2
18   { ((Local_3 AND Local_1))
19     double Local_7 := Call "f4" (double Input_2)
20   } goto Label_3
21   label: Label_3
22   double Output_0 := Choose  (double Local_4, double Local_5, double
   Local_6, double Local_7)
23 }
```

Listing 42: komplexe Bedingungen, flache Struktur

Durch die Organisation der Sprünge (Abschnitt 4.12) wird die Bedingung des letzten Sprungs (Zeile 18) zu einer Tautologie geändert. Wenn die Sprungbedingungen in Teilbedingungen aufgelöst werden würden (zuerst `Local_0` bzw. `Local_1` und in einer weiteren Verschachtelungsebene `Local_2` bzw. `Local_3`), würde zum einen der Aufwand für die Berechnung der Konjunktion entfallen. Zum anderen könnte bei der richtigen Reihenfolge der Sprünge eine der Variablen `Local_2` oder `Local_3` und eine der Variablen `Local_0` oder `Local_1` nicht mehr benötigt und damit ihre Definitionen zu totem Code werden.

Die verwendete Zwischenrepräsentation ist kein 3-Adress-Code. Damit entfällt die Zerlegung von Ausdrücken mit mehr als zwei Operanden in Teilausdrücke. Diese Zerle-

gung kann jedoch Optimierungspotenzial bei der Anwendung von CSE (Abschnitt 4.15) eröffnen. Ein Beispiel dafür ist in Listing 43 dargestellt.

```
1 boolean Local_4 := Guard  (boolean Local_0, boolean Local_1)
2 boolean Local_5 := Guard  (boolean Local_2, boolean Local_3)
3 boolean Local_6 := Guard  (boolean Local_0, boolean Local_1, boolean
  Local_2, boolean Local_3)
```

Listing 43: verdecktes Optimierungspotenzial

Durch eine geschickte Zerlegung des Ausdrucks in Zeile 3 und die Verwendung der bereits berechneten Werte aus den Zeilen 1 und 2 könnte Zeile 3 durch die Anweisung

```
boolean Local_6 := Guard  (boolean Local_4, boolean
Local_5)
```

ersetzt werden. Statt vier Variablen müssen nur noch zwei konjunktiv verknüpft werden.

8 Anhang

Die in Abschnitt 8.1 gezeigten Beispiele vor der Optimierung wurden manuell im Eingabeformat (im Zwischenformat von SIG) geschrieben.

Die in Abschnitt 8.2 gezeigten Beispiele vor der Optimierung wurden durch das Frontend des SIG-Compilers von SIG ins Eingabeformat übersetzt.

8.1 Ausgangsprogramme der Beispiele

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, AlgebraicDataType_0, double
9    out: double
10   state:
11   locals: boolean, boolean, boolean, boolean, double, double,
   double, double, double, double, double, double, double
12   init: --
13   loop: BasicBlock_2
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
19     boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
20     double Local_4 := Call "f1" (double Input_2)
21     double Local_5 := Call "f2" (double Local_4)
22     double Local_6 := Call "f3" (double Local_4)
23     double Local_7 := Guard  (double Local_5, boolean Local_0)
24     double Local_8 := Guard  (double Local_6, boolean Local_1)
25     double Local_9 := Choose  (double Local_7, double Local_8)
26     double Local_10 := Call "f4" (double Input_2)
27     double Local_11 := Guard  (double Local_9, boolean Local_2)
28     double Local_12 := Guard  (double Local_10, boolean Local_3)
29     double Output_0 := Choose  (double Local_11, double Local_12)
30   } next: --
31 }
```

Listing 44: Ausgangsprogramm für Beispiele in Listings 5, 6

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4    C: A()
5  }
6
7  Block_1 {
8    env:
9    in: AlgebraicDataType_0, double
10   out: double
11   state:
12   locals: boolean, boolean, double, double, double, double, boolean,
   double, double, double
13   init: --
14   loop: BasicBlock_2
15
16   BasicBlock_2 {
17     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
19     boolean Local_6 := Destruct "C" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
20     double Local_8 := Call "f4" (double Input_1)
21     double Local_2 := Call "f1" (double Local_8)
22     double Local_3 := Call "f2" (double Local_8)
23     double Local_7 := Call "f3" (double Input_1)
24     double Local_4 := Guard  (double Local_2, boolean Local_0)
25     double Local_5 := Guard  (double Local_3, boolean Local_1)
26     double Local_9 := Guard  (double Local_7, boolean Local_6)
27     double Output_0 := Choose  (double Local_4, double Local_5,
   double Local_9)
28   } next: --
29 }
```

Listing 45: Ausgangsprogramm für Beispiele in Listings 7, 8

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4    C: A()
5  }
6
7  Block_1 {
8    env:
9    in: AlgebraicDataType_0, double
10   out: double
11   state:
12   locals: boolean, boolean, double, double, double, double, boolean,
   double, double, double
13   init: --
14   loop: BasicBlock_2
15
16   BasicBlock_2 {
17     double Local_8 := Call "f4" (double Input_1)
18     double Local_2 := Call "f1" (double Local_8)
19     double Local_3 := Call "f2" (double Local_8)
20     double Local_7 := Call "f3" (double Input_1)
21     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
22     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
23     boolean Local_6 := Destruct "C" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
24     double Local_4 := Guard  (double Local_2, boolean Local_0)
25     double Local_5 := Guard  (double Local_3, boolean Local_1)
26     double Local_9 := Guard  (double Local_7, boolean Local_6)
27     double Output_0 := Choose  (double Local_4, double Local_5,
   double Local_9)
28   } next: --
29 }
```

Listing 46: Ausgangsprogramm für Beispiele in Listings 9, 10, 11

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, double, AlgebraicDataType_0
9    out: double
10   state:
11   locals: boolean, boolean, double, double, double, double, boolean,
   double, double, double, boolean
12   init: --
13   loop: BasicBlock_2
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     boolean Local_6 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_2)
19     boolean Local_10 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_2)
20     double Local_8 := Call "f4" (double Input_1)
21     double Local_2 := Call "f1" (double Local_8)
22     double Local_3 := Call "f2" (double Local_8)
23     double Local_7 := Call "f3" (double Input_1)
24     double Local_4 := Guard  (double Local_2, boolean Local_0,
   boolean Local_6)
25     double Local_5 := Guard  (double Local_3, boolean Local_0,
   boolean Local_10)
26     double Local_9 := Guard  (double Local_7, boolean Local_1)
27     double Output_0 := Choose  (double Local_4, double Local_5,
   double Local_9)
28   } next: --
29 }
```

Listing 47: Ausgangsprogramm für Beispiele in Listings 12, 13

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, double, double
9    out: double, double
10   state:
11   locals: boolean, boolean, double, double, double, double, double,
   double, double, double
12   init: --
13   loop: BasicBlock_2
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     double Local_2 := Call "f1" (double Input_1)
19     double Local_3 := Call "f2" (double Input_1)
20     double Local_6 := Call "f3" (double Input_2)
21     double Local_7 := Call "f4" (double Input_2)
22     double Local_4 := Guard  (double Local_2, boolean Local_0)
23     double Local_5 := Guard  (double Local_3, boolean Local_1)
24     double Local_8 := Guard  (double Local_6, boolean Local_0)
25     double Local_9 := Guard  (double Local_7, boolean Local_1)
26     double Output_0 := Choose  (double Local_4, double Local_5)
27     double Output_1 := Choose  (double Local_8, double Local_9)
28   } next: --
29 }
```

Listing 48: Ausgangsprogramm für Beispiele in Listings 14, 15

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, double, double
9    out: double, double
10   state:
11   locals: boolean, boolean, double, double, double
12   init: --
13   loop: BasicBlock_2
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     double Local_2 := Call "f" (double Input_1)
19     double Local_3 := Guard  (double Local_2, boolean Local_0)
20     double Local_4 := Guard  (double Input_1, boolean Local_1)
21     double Output_0 := Choose  (double Local_3, double Local_4)
22     double Output_1 := Move  (double Input_2)
23   } next: --
24 }
```

Listing 49: Ausgangsprogramm für Beispiele in Listings 17, 18

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, double, double
9    out: double
10   state:
11   locals: boolean, boolean, double, double
12   init: --
13   loop: BasicBlock_2
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     double Local_2 := Guard  (double Input_1, boolean Local_0)
19     double Local_3 := Guard  (double Input_2, boolean Local_1)
20     double Output_0 := Choose  (double Local_2, double Local_3)
21   } next: --
22 }
```

Listing 50: Ausgangsprogramm für Beispiele in Listings 19, 20, 21

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, AlgebraicDataType_0, double
9    out: double
10   state:
11   locals: boolean, boolean, boolean, boolean, double, double,
   double, double, boolean
12   init: --
13   loop: BasicBlock_2
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
19     boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
20     double Local_4 := Call "f1" (double Input_2)
21     double Local_5 := Call "f2" (double Input_2)
22     double Local_6 := Guard  (double Local_4, boolean Local_0,
   boolean Local_2)
23     boolean Local_8 := Choose  (boolean Local_1, boolean Local_3)
24     double Local_7 := Guard  (double Local_5, boolean Local_8)
25     double Output_0 := Choose  (double Local_6, double Local_7)
26   } next: --
27 }
```

Listing 51: Ausgangsprogramm für Beispiele in Listings 22, 23

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env: AlgebraicDataType_0, AlgebraicDataType_0
8    in: double
9    out: double
10   state: double
11   locals: boolean, boolean, boolean, boolean, double, double,
   double, double, double, double, double, double
12   init: BasicBlock_2
13   loop: BasicBlock_3
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
18     boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
19     boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
20     double Local_4 := DoubleConstant 1.0 ()
21     double Local_5 := DoubleConstant 2.0 ()
22     double Local_6 := DoubleConstant 3.0 ()
23     double Local_7 := DoubleConstant 4.0 ()
24     double Local_8 := Guard  (double Local_4, boolean Local_0,
   boolean Local_2)
25     double Local_9 := Guard  (double Local_5, boolean Local_0,
   boolean Local_3)
26     double Local_10 := Guard  (double Local_6, boolean Local_1,
   boolean Local_2)
27     double Local_11 := Guard  (double Local_7, boolean Local_1,
   boolean Local_3)
28     double Post_0 := Choose  (double Local_8, double Local_9, double
   Local_10, double Local_11)
29   } next: --
30
31   BasicBlock_3 {
32     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
33     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
34     boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
35     boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
36     double Local_4 := Call "f1" (double Input_0, double Pre_0)
37     double Local_5 := Call "f2" (double Input_0, double Pre_0)
38     double Local_6 := Call "f3" (double Input_0, double Pre_0)
39     double Local_7 := Call "f4" (double Input_0, double Pre_0)
40     double Local_8 := Guard  (double Local_4, boolean Local_0,
   boolean Local_2)
```

```
41     double Local_9 := Guard  (double Local_5, boolean Local_0,
   boolean Local_3)
42     double Local_10 := Guard  (double Local_6, boolean Local_1,
   boolean Local_2)
43     double Local_11 := Guard  (double Local_7, boolean Local_1,
   boolean Local_3)
44     double Post_0 := Choose  (double Local_8, double Local_9, double
   Local_10, double Local_11)
45     double Output_0 := Move  (double Post_0)
46   } next: --
47 }
```

Listing 52: Ausgangsprogramm für Beispiele in Listings 28, 29, 53, 54

```
1  Block_20 {
2    env: AlgebraicDataType_0, AlgebraicDataType_0
3    in: double
4    out: double
5    state: double
6    locals: boolean, boolean, boolean, boolean, double, double,
   double, double, double, double, double, double, boolean, boolean,
   boolean, boolean, boolean, boolean
7    init: BasicBlock_4
8    loop: BasicBlock_12
9
10   BasicBlock_4 {
11     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
12     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
13     boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
14     boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
15     boolean Local_12 := Guard  (boolean Local_2, boolean Local_0)
16   } next: boolean Local_12 ? BasicBlock_5 : BasicBlock_6
17
18   BasicBlock_12 {
19     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
20     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
21     boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
22     boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
23     boolean Local_15 := Guard  (boolean Local_3, boolean Local_1)
24   } next: boolean Local_15 ? BasicBlock_13 : BasicBlock_14
25
26   BasicBlock_5 {
27     double Local_4 := DoubleConstant 1.0 ()
28   } next: BasicBlock_7
29
30   BasicBlock_6 {
31     boolean Local_13 := Guard  (boolean Local_3, boolean Local_0)
32   } next: boolean Local_13 ? BasicBlock_8 : BasicBlock_9
33
34   BasicBlock_13 {
35     double Local_7 := Call "f4" (double Input_0, double Pre_0)
36   } next: BasicBlock_15
37
38   BasicBlock_14 {
39     boolean Local_16 := Guard  (boolean Local_2, boolean Local_0)
40   } next: boolean Local_16 ? BasicBlock_16 : BasicBlock_17
41
42   BasicBlock_7 {
43     double Post_0 := Choose  (double Local_4, double Local_5, double
   Local_6, double Local_7)
44   } next: --
45
```

```
46   BasicBlock_8 {
47     double Local_5 := DoubleConstant 2.0 ()
48   } next: BasicBlock_7
49
50   BasicBlock_9 {
51     boolean Local_14 := Guard  (boolean Local_3, boolean Local_1)
52   } next: boolean Local_14 ? BasicBlock_10 : BasicBlock_11
53
54   BasicBlock_15 {
55     double Post_0 := Choose  (double Local_4, double Local_5, double
   Local_6, double Local_7)
56     double Output_0 := Move  (double Post_0)
57   } next: --
58
59   BasicBlock_16 {
60     double Local_4 := Call "f1" (double Input_0, double Pre_0)
61   } next: BasicBlock_15
62
63   BasicBlock_17 {
64     boolean Local_17 := Guard  (boolean Local_3, boolean Local_0)
65   } next: boolean Local_17 ? BasicBlock_18 : BasicBlock_19
66
67   BasicBlock_10 {
68     double Local_7 := DoubleConstant 4.0 ()
69   } next: BasicBlock_7
70
71   BasicBlock_11 {
72     double Local_6 := DoubleConstant 3.0 ()
73   } next: BasicBlock_7
74
75   BasicBlock_18 {
76     double Local_5 := Call "f2" (double Input_0, double Pre_0)
77   } next: BasicBlock_15
78
79   BasicBlock_19 {
80     double Local_6 := Call "f3" (double Input_0, double Pre_0)
81   } next: BasicBlock_15
82 }
```

Listing 53: Vollständiges Programm von Listing 28

```
1  Block_20 {
2    env: AlgebraicDataType_0, AlgebraicDataType_0
3    in: double
4    out: double
5    state: double
6    locals: double, double, double, double, boolean, boolean, boolean,
   boolean, boolean, boolean, boolean
7    init: BasicBlock_4
8    loop: BasicBlock_12
9
10   BasicBlock_4 {
11     boolean Local_4 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
12     boolean Local_5 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
13     boolean Local_6 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
14     boolean Local_7 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
15     boolean Local_8 := Guard  (boolean Local_6, boolean Local_4)
16   } next: boolean Local_8 ? BasicBlock_5 : BasicBlock_6
17
18   BasicBlock_12 {
19     boolean Local_10 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
20     boolean Local_5 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_0)
21     boolean Local_6 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
22     boolean Local_7 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Environment_1)
23     boolean Local_8 := Guard  (boolean Local_7, boolean Local_5)
24   } next: boolean Local_8 ? BasicBlock_13 : BasicBlock_14
25
26   BasicBlock_5 {
27     double Local_0 := DoubleConstant 1.0 ()
28   } next: BasicBlock_7
29
30   BasicBlock_6 {
31     boolean Local_9 := Guard  (boolean Local_7, boolean Local_4)
32   } next: boolean Local_9 ? BasicBlock_8 : BasicBlock_9
33
34   BasicBlock_13 {
35     double Local_3 := Call "f4" (double Input_0, double Pre_0)
36   } next: BasicBlock_15
37
38   BasicBlock_14 {
39     boolean Local_9 := Guard  (boolean Local_6, boolean Local_10)
40   } next: boolean Local_9 ? BasicBlock_16 : BasicBlock_17
41
42   BasicBlock_7 {
43     double Post_0 := Choose  (double Local_0, double Local_1, double
   Local_2, double Local_3)
44   } next: --
45
46   BasicBlock_8 {
```

```
47      double Local_1 := DoubleConstant 2.0 ()
48    } next: BasicBlock_7
49
50    BasicBlock_9 {
51      boolean Local_10 := Guard  (boolean Local_7, boolean Local_5)
52    } next: boolean Local_10 ? BasicBlock_10 : BasicBlock_11
53
54    BasicBlock_15 {
55      double Post_0 := Choose  (double Local_0, double Local_1, double
    Local_2, double Local_3)
56      double Output_0 := Move  (double Post_0)
57    } next: --
58
59    BasicBlock_16 {
60      double Local_0 := Call "f1" (double Input_0, double Pre_0)
61    } next: BasicBlock_15
62
63    BasicBlock_17 {
64      boolean Local_4 := Guard  (boolean Local_7, boolean Local_10)
65    } next: boolean Local_4 ? BasicBlock_18 : BasicBlock_19
66
67    BasicBlock_10 {
68      double Local_3 := DoubleConstant 4.0 ()
69    } next: BasicBlock_7
70
71    BasicBlock_11 {
72      double Local_2 := DoubleConstant 3.0 ()
73    } next: BasicBlock_7
74
75    BasicBlock_18 {
76      double Local_1 := Call "f2" (double Input_0, double Pre_0)
77    } next: BasicBlock_15
78
79    BasicBlock_19 {
80      double Local_2 := Call "f3" (double Input_0, double Pre_0)
81    } next: BasicBlock_15
82  }
```

Listing 54: Vollständiges Programm von Listing 29

```
1  AlgebraicDataType_0 {
2    A: A()
3    B: A()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, AlgebraicDataType_0, double
9    out: double
10   state:
11   locals: boolean, boolean, boolean, boolean, double, double,
   double, double, double, double, double, double
12   init: --
13   loop: BasicBlock_2
14
15   BasicBlock_2 {
16     boolean Local_0 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
17     boolean Local_1 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
18     boolean Local_2 := Destruct "A" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
19     boolean Local_3 := Destruct "B" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_1)
20     double Local_4 := Call "f1" (double Input_2)
21     double Local_5 := Call "f2" (double Input_2)
22     double Local_6 := Call "f3" (double Input_2)
23     double Local_7 := Call "f4" (double Input_2)
24     double Local_8 := Guard  (double Local_4, boolean Local_0,
   boolean Local_2)
25     double Local_9 := Guard  (double Local_5, boolean Local_0,
   boolean Local_3)
26     double Local_10 := Guard  (double Local_6, boolean Local_1,
   boolean Local_2)
27     double Local_11 := Guard  (double Local_7, boolean Local_1,
   boolean Local_3)
28     double Output_0 := Choose  (double Local_8, double Local_9,
   double Local_10, double Local_11)
29   } next: --
30 }
```

Listing 55: Ausgangsprogramm für Beispiel in Listing 42

8.2 Zwischenergebnisse der Evaluation

```
1  AlgebraicDataType_0 {
2    hold: hold()
3    sample: sample()
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0, double
9    out: double
10   state: double
11   locals: boolean, double, boolean, double, double
12   init: BasicBlock_2
13   loop: BasicBlock_3
14
15   BasicBlock_2 {
16     double Local_1 := DoubleConstant 0.0 ()
17     double Post_0 := Move  (double Local_1)
18   } next: --
19
20   BasicBlock_3 {
21     boolean Local_2 := Destruct "hold" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
22     double Local_3 := Guard  (double Pre_0, boolean Local_2)
23     boolean Local_0 := Destruct "sample" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
24     double Local_4 := Guard  (double Input_1, boolean Local_0)
25     double Output_0 := Choose  (double Local_3, double Local_4)
26     double Post_0 := Move  (double Output_0)
27   } next: --
28 }
```

Listing 56: Zwischenformat sah

```
1  AlgebraicDataType_0 {
2    hold: hold()
3    sample: sample()
4  }
5
6  Block_4 {
7    env:
8    in: AlgebraicDataType_0, double
9    out: double
10   state: double
11   locals: double, double, boolean
12   init: BasicBlock_5
13   loop: BasicBlock_6
14
15   BasicBlock_5 {
16     double Post_0 := DoubleConstant 0.0 ()
17   } next: --
18
19   BasicBlock_6 {
20     boolean Local_2 := Destruct "sample" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
21   } next: boolean Local_2 ? BasicBlock_7 : BasicBlock_8
22
23   BasicBlock_7 {
24     double Local_1 := Move  (double Input_1)
25   } next: BasicBlock_9
26
27   BasicBlock_8 {
28     double Local_0 := Move  (double Pre_0)
29   } next: BasicBlock_9
30
31   BasicBlock_9 {
32     double Output_0 := Choose  (double Local_0, double Local_1)
33     double Post_0 := Move  (double Output_0)
34   } next: --
35 }
```

Listing 57: optimiertes sah

```
1  AlgebraicDataType_0 {
2    hold: hold()
3    sample: sample(double)
4  }
5
6  Block_1 {
7    env:
8    in: AlgebraicDataType_0
9    out: double
10   state: double
11   locals: double, boolean, double, boolean, double, double
12   init: BasicBlock_2
13   loop: BasicBlock_3
14
15   BasicBlock_2 {
16     double Local_0 := DoubleConstant 0.0 ()
17     double Post_0 := Move  (double Local_0)
18   } next: --
19
20   BasicBlock_3 {
21     double Local_2, boolean Local_3 := Destruct "sample"
   AlgebraicDataType_0 (AlgebraicDataType_0 Input_0)
22     double Local_4 := Guard  (double Local_2, boolean Local_3)
23     boolean Local_1 := Destruct "hold" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
24     double Local_5 := Guard  (double Pre_0, boolean Local_1)
25     double Output_0 := Choose  (double Local_4, double Local_5)
26     double Post_0 := Move  (double Output_0)
27   } next: --
28 }
```

Listing 58: Zwischenformat sah2

```
1  AlgebraicDataType_0 {
2    hold: hold()
3    sample: sample(double)
4  }
5
6  Block_4 {
7    env:
8    in: AlgebraicDataType_0
9    out: double
10   state: double
11   locals: boolean, double, double, double
12   init: BasicBlock_5
13   loop: BasicBlock_6
14
15   BasicBlock_5 {
16     double Post_0 := DoubleConstant 0.0 ()
17   } next: --
18
19   BasicBlock_6 {
20     double Local_3, boolean Local_0 := Destruct "sample"
   AlgebraicDataType_0 (AlgebraicDataType_0 Input_0)
21   } next: boolean Local_0 ? BasicBlock_7 : BasicBlock_8
22
23   BasicBlock_7 {
24     double Local_1 := Move  (double Local_3)
25   } next: BasicBlock_9
26
27   BasicBlock_8 {
28     double Local_2 := Move  (double Pre_0)
29   } next: BasicBlock_9
30
31   BasicBlock_9 {
32     double Output_0 := Choose  (double Local_1, double Local_2)
33     double Post_0 := Move  (double Output_0)
34   } next: --
35 }
```

Listing 59: optimiertes sah2-1

```
1  AlgebraicDataType_0 {
2    hold: hold()
3    sample: sample(double)
4  }
5
6  Block_4 {
7    env:
8    in: AlgebraicDataType_0
9    out: double
10   state: double
11   locals: boolean, boolean, double, double, double
12   init: BasicBlock_5
13   loop: BasicBlock_6
14
15   BasicBlock_5 {
16     double Post_0 := DoubleConstant 0.0 ()
17   } next: --
18
19   BasicBlock_6 {
20     double Local_3, boolean Local_1 := Destruct "sample"
   AlgebraicDataType_0 (AlgebraicDataType_0 Input_0)
21     boolean Local_0 := Destruct "hold" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
22   } next: boolean Local_0 ? BasicBlock_7 : BasicBlock_8
23
24   BasicBlock_7 {
25     double Local_2 := Move  (double Pre_0)
26   } next: BasicBlock_9
27
28   BasicBlock_8 {
29     double Local_4 := Move  (double Local_3)
30   } next: BasicBlock_9
31
32   BasicBlock_9 {
33     double Output_0 := Choose  (double Local_4, double Local_2)
34     double Post_0 := Move  (double Output_0)
35   } next: --
36 }
```

Listing 60: optimiertes sah2-2

```
1  AlgebraicDataType_0 {
2    caseA: caseA()
3    caseB: caseB()
4    caseC: caseC()
5  }
6
7  Block_1 {
8    env:
9    in: AlgebraicDataType_0, double
10   out: double
11   state:
12   locals: boolean, double, boolean, double, boolean, double, double,
   double, double
13   init: BasicBlock_2
14   loop: BasicBlock_3
15
16   BasicBlock_2 {
17   } next: --
18
19   BasicBlock_3 {
20     double Local_3 := Call "+" (double Input_1, double Input_1)
21     boolean Local_4 := Destruct "caseC" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
22     double Local_5 := Call "*" (double Input_1, double Input_1)
23     boolean Local_0 := Destruct "caseA" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
24     double Local_1 := Call "-" (double Input_1)
25     boolean Local_2 := Destruct "caseB" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
26     double Local_7 := Guard  (double Local_3, boolean Local_2)
27     double Local_8 := Guard  (double Local_1, boolean Local_0)
28     double Local_6 := Guard  (double Local_5, boolean Local_4)
29     double Output_0 := Choose  (double Local_6, double Local_7,
   double Local_8)
30   } next: --
31 }
```

Listing 61: Zwischenformat choose

```
1  AlgebraicDataType_0 {
2    caseA: caseA()
3    caseB: caseB()
4    caseC: caseC()
5  }
6
7  Block_4 {
8    env:
9    in: AlgebraicDataType_0, double
10   out: double
11   state:
12   locals: double, double, double, boolean, boolean
13   init: BasicBlock_5
14   loop: BasicBlock_6
15
16   BasicBlock_5 {
17   } next: --
18
19   BasicBlock_6 {
20     boolean Local_4 := Destruct "caseC" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
21     boolean Local_3 := Destruct "caseA" AlgebraicDataType_0
   (AlgebraicDataType_0 Input_0)
22   } next: boolean Local_3 ? BasicBlock_7 : BasicBlock_8
23
24   BasicBlock_7 {
25     double Local_1 := Call "-" (double Input_1)
26   } next: BasicBlock_9
27
28   BasicBlock_8 {
29   } next: boolean Local_4 ? BasicBlock_10 : BasicBlock_11
30
31   BasicBlock_9 {
32     double Output_0 := Choose  (double Local_0, double Local_2,
   double Local_1)
33   } next: --
34
35   BasicBlock_10 {
36     double Local_0 := Call "*" (double Input_1, double Input_1)
37   } next: BasicBlock_9
38
39   BasicBlock_11 {
40     double Local_2 := Call "+" (double Input_1, double Input_1)
41   } next: BasicBlock_9
42 }
```

Listing 62: optimiertes choose

```
1  Block_0 {
2    env:
3    in: int
4    out: int, int
5    state: int, int, int, int
6    locals: int, int, int, bool, boolean, int, boolean, int, int, int,
   int
7    init: BasicBlock_1
8    loop: BasicBlock_2
9
10   BasicBlock_1 {
11     int Local_7 := IntConstant 1 ()
12     int Local_10 := IntConstant 0 ()
13     int Post_3 := Move  (int Local_10)
14     int Post_2 := Move  (int Local_7)
15     int Post_0 := Move  (int Local_10)
16     int Post_1 := Move  (int Local_10)
17   } next: --
18
19   BasicBlock_2 {
20     int Post_3 := Move  (int Pre_3)
21     int Local_5 := Call "+" (int Pre_1, int Pre_2)
22     int Output_0 := Move  (int Pre_0)
23     bool Local_3 := Call "=" (int Input_0, int Output_0)
24     boolean Local_6 := Destruct "F" bool (bool Local_3)
25     int Local_8 := Guard  (int Pre_2, boolean Local_6)
26     boolean Local_4 := Destruct "T" bool (bool Local_3)
27     int Local_9 := Guard  (int Local_5, boolean Local_4)
28     int Local_2 := Choose  (int Local_8, int Local_9)
29     int Post_1 := Move  (int Local_2)
30     int Post_0 := Move  (int Input_0)
31     int Post_2 := Move  (int Pre_2)
32     int Local_1 := Guard  (int Pre_3, boolean Local_4)
33     int Local_0 := Guard  (int Pre_1, boolean Local_6)
34     int Output_1 := Choose  (int Local_0, int Local_1)
35   } next: --
36 }
```

Listing 63: Zwischenformat rle

```
1  Block_3 {
2    env:
3    in: int
4    out: int, int
5    state: int, int, int, int
6    locals: int, int, int, int, boolean, bool
7    init: BasicBlock_4
8    loop: BasicBlock_5
9
10   BasicBlock_4 {
11     int Post_2 := IntConstant 1 ()
12     int Post_3 := IntConstant 0 ()
13     int Post_0 := Move  (int Post_3)
14     int Post_1 := Move  (int Post_3)
15   } next: --
16
17   BasicBlock_5 {
18     int Post_3 := Move  (int Pre_3)
19     int Output_0 := Move  (int Pre_0)
20     bool Local_5 := Call "=" (int Input_0, int Output_0)
21     boolean Local_4 := Destruct "T" bool (bool Local_5)
22   } next: boolean Local_4 ? BasicBlock_6 : BasicBlock_7
23
24   BasicBlock_6 {
25     int Local_1 := Call "+" (int Pre_1, int Pre_2)
26     int Local_3 := Move  (int Pre_3)
27   } next: BasicBlock_8
28
29   BasicBlock_7 {
30     int Local_0 := Move  (int Pre_2)
31     int Local_2 := Move  (int Pre_1)
32   } next: BasicBlock_8
33
34   BasicBlock_8 {
35     int Post_1 := Choose  (int Local_0, int Local_1)
36     int Post_0 := Move  (int Input_0)
37     int Post_2 := Move  (int Pre_2)
38     int Output_1 := Choose  (int Local_2, int Local_3)
39   } next: --
40 }
```

Listing 64: optimiertes rle

```
AlgebraicDataType_4 {
  Attack: Attack()
  Decay: Decay()
  Release: Release()
  Sustain: Sustain()
}

FunctionTypes {
  FunctionType_0 = ComponentType(bool => double)
  FunctionType_1 = ComponentType( => double)
  FunctionType_2 = ComponentType(double => double)
  FunctionType_3 = SessionType(double => double)
  FunctionType_5 = SessionType( => double)
}

Block_6 {
  env:
  in: FunctionType_2, FunctionType_2, FunctionType_1,
FunctionType_2
  out: FunctionType_0
  state:
  locals:
  init: BasicBlock_7
  loop: BasicBlock_8

  BasicBlock_7 {
  } next: --

  BasicBlock_8 {
    FunctionType_0 Output_0 := Closure Block_9 (FunctionType_1
Input_2, FunctionType_2 Input_0, FunctionType_2 Input_3,
FunctionType_2 Input_1)
  } next: --
}

Block_9 {
  env: FunctionType_1, FunctionType_2, FunctionType_2,
FunctionType_2
  in: bool
  out: double
  state: double, AlgebraicDataType_4, void, void, void, void, void,
FunctionType_3, void, AlgebraicDataType_4, AlgebraicDataType_4,
AlgebraicDataType_4, FunctionType_3, void, FunctionType_3, void,
FunctionType_5, void, double, AlgebraicDataType_4, double
```

```
38    locals: boolean, double, double, FunctionType_5, void, void,
    double, FunctionType_3, void, void, boolean, double, double,
    double, FunctionType_3, void, void, boolean, double, double,
    double, double, FunctionType_3, void, void, boolean, double,
    double, double, double, double, AlgebraicDataType_4,
    AlgebraicDataType_4, bool, boolean, AlgebraicDataType_4, boolean,
    AlgebraicDataType_4, AlgebraicDataType_4, void, double,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    bool, boolean, AlgebraicDataType_4, boolean, AlgebraicDataType_4,
    AlgebraicDataType_4, boolean, AlgebraicDataType_4, boolean,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    double
39    init: BasicBlock_10
40    loop: BasicBlock_11
41
42    BasicBlock_10 {
43      AlgebraicDataType_4 Local_64 := Construct "Release"
    AlgebraicDataType_4 ()
44      AlgebraicDataType_4 Post_1 := Move  (AlgebraicDataType_4
    Local_64)
45      AlgebraicDataType_4 Local_49 := Construct "Decay"
    AlgebraicDataType_4 ()
46      AlgebraicDataType_4 Local_44 := Construct "Attack"
    AlgebraicDataType_4 ()
47      FunctionType_3 Local_22, void Local_23 := Start
    (FunctionType_2 Environment_1, double Pre_0)
48      FunctionType_3 Post_14 := Move  (FunctionType_3 Local_22)
49      double Local_65 := DoubleConstant 0.0 ()
50      double Post_20 := Move  (double Local_65)
51      void Post_15 := Move  (void Local_23)
52      AlgebraicDataType_4 Post_19 := Move  (AlgebraicDataType_4
    Local_64)
53      FunctionType_5 Local_3, void Local_4 := Start  (FunctionType_1
    Environment_0)
54      void Post_17 := Move  (void Local_4)
55      FunctionType_5 Post_16 := Move  (FunctionType_5 Local_3)
56      FunctionType_3 Local_7, void Local_8 := Start  (FunctionType_2
    Environment_3, double Pre_0)
57      void Post_2 := Move  (void Local_4)
58      FunctionType_3 Post_12 := Move  (FunctionType_3 Local_7)
59      FunctionType_3 Local_14, void Local_15 := Start
    (FunctionType_2 Environment_2, double Pre_0)
60      void Post_4 := Move  (void Local_15)
61      void Post_3 := Move  (void Local_23)
62      AlgebraicDataType_4 Post_10 := Move  (AlgebraicDataType_4
    Local_44)
63      AlgebraicDataType_4 Post_9 := Move  (AlgebraicDataType_4
    Local_49)
64      double Local_19 := DoubleConstant 1.0 ()
65      double Post_18 := Move  (double Local_19)
66      FunctionType_3 Post_7 := Move  (FunctionType_3 Local_14)
67      AlgebraicDataType_4 Local_35 := Construct "Sustain"
    AlgebraicDataType_4 ()
```

```
68      void Post_8 := Move  (void Local_15)
69      double Post_0 := Move  (double Local_65)
70      AlgebraicDataType_4 Post_11 := Move  (AlgebraicDataType_4
    Local_35)
71      void Post_13 := Move  (void Local_8)
72      void Post_6 := Move  (void Local_8)
73      void Post_5 := Move  (void Local_4)
74    } next: --
75
76    BasicBlock_11 {
77      double Local_21, void Local_24 := Sample  (FunctionType_3
    Pre_14, void Pre_3, double Pre_0)
78      double Local_20 := Call "+" (double Pre_0, double Local_21)
79      double Local_13, void Local_16 := Sample  (FunctionType_3
    Pre_7, void Pre_4, double Pre_0)
80      double Local_12 := Call "-" (double Pre_0, double Local_13)
81      double Local_18 := Call "max" (double Pre_20, double Local_12)
82      boolean Local_17 := Destruct "Release" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
83      double Local_28 := Guard  (double Local_18, boolean Local_17)
84      double Local_1, void Local_5 := Sample  (FunctionType_5 Pre_16,
    void Pre_5)
85      double Local_6, void Local_9 := Sample  (FunctionType_3 Pre_12,
    void Pre_6, double Pre_0)
86      double Local_2 := Call "-" (double Pre_0, double Local_6)
87      double Local_11 := Call "max" (double Local_1, double Local_2)
88      boolean Local_10 := Destruct "Decay" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
89      double Local_29 := Guard  (double Local_11, boolean Local_10)
90      boolean Local_0 := Destruct "Sustain" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
91      double Local_30 := Guard  (double Pre_0, boolean Local_0)
92      double Local_26 := Call "min" (double Pre_18, double Local_20)
93      boolean Local_25 := Destruct "Attack" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
94      double Local_27 := Guard  (double Local_26, boolean Local_25)
95      double Output_0 := Choose  (double Local_27, double Local_28,
    double Local_29, double Local_30)
96      double Local_40, void Local_39 := Sample  (FunctionType_5
    Pre_16, void Pre_2)
97      bool Local_33 := Call "<=" (double Output_0, double Local_40)
98      boolean Local_34 := Destruct "T" bool (bool Local_33)
99      bool Local_47 := Call ">=" (double Output_0, double Pre_18)
100     boolean Local_50 := Destruct "F" bool (bool Local_47)
101     AlgebraicDataType_4 Local_51 := Guard  (AlgebraicDataType_4
    Pre_1, boolean Local_50)
102     boolean Local_48 := Destruct "T" bool (bool Local_47)
103     AlgebraicDataType_4 Local_52 := Guard  (AlgebraicDataType_4
    Pre_9, boolean Local_48)
104     AlgebraicDataType_4 Local_54 := Choose  (AlgebraicDataType_4
    Local_51, AlgebraicDataType_4 Local_52)
105     boolean Local_53 := Destruct "T" bool (bool Input_0)
106     AlgebraicDataType_4 Local_31 := Guard  (AlgebraicDataType_4
    Pre_1, boolean Local_53)
107     void Post_15 := Move  (void Pre_15)
108     void Post_17 := Move  (void Pre_17)
```

```
109     boolean Local_36 := Destruct "F" bool (bool Local_33)
110     AlgebraicDataType_4 Local_45 := Guard  (AlgebraicDataType_4
    Pre_10, boolean Local_53)
111     void Post_2 := Move  (void Local_39)
112     AlgebraicDataType_4 Post_10 := Move  (AlgebraicDataType_4
    Pre_10)
113     AlgebraicDataType_4 Post_9 := Move  (AlgebraicDataType_4 Pre_9)
114     double Post_18 := Move  (double Pre_18)
115     AlgebraicDataType_4 Local_37 := Guard  (AlgebraicDataType_4
    Pre_1, boolean Local_36)
116     AlgebraicDataType_4 Local_38 := Guard  (AlgebraicDataType_4
    Pre_11, boolean Local_34)
117     AlgebraicDataType_4 Local_41 := Choose  (AlgebraicDataType_4
    Local_37, AlgebraicDataType_4 Local_38)
118     AlgebraicDataType_4 Local_42 := Guard  (AlgebraicDataType_4
    Local_41, boolean Local_53)
119     boolean Local_55 := Destruct "F" bool (bool Input_0)
120     AlgebraicDataType_4 Local_56 := Guard  (AlgebraicDataType_4
    Pre_19, boolean Local_55)
121     AlgebraicDataType_4 Local_32 := Choose  (AlgebraicDataType_4
    Local_56, AlgebraicDataType_4 Local_31)
122     AlgebraicDataType_4 Local_62 := Guard  (AlgebraicDataType_4
    Local_32, boolean Local_0)
123     AlgebraicDataType_4 Local_46 := Choose  (AlgebraicDataType_4
    Local_56, AlgebraicDataType_4 Local_45)
124     AlgebraicDataType_4 Local_60 := Guard  (AlgebraicDataType_4
    Local_46, boolean Local_17)
125     AlgebraicDataType_4 Local_57 := Guard  (AlgebraicDataType_4
    Local_54, boolean Local_53)
126     AlgebraicDataType_4 Local_58 := Choose  (AlgebraicDataType_4
    Local_56, AlgebraicDataType_4 Local_57)
127     AlgebraicDataType_4 Local_59 := Guard  (AlgebraicDataType_4
    Local_58, boolean Local_25)
128     AlgebraicDataType_4 Local_43 := Choose  (AlgebraicDataType_4
    Local_56, AlgebraicDataType_4 Local_42)
129     AlgebraicDataType_4 Local_61 := Guard  (AlgebraicDataType_4
    Local_43, boolean Local_10)
130     AlgebraicDataType_4 Local_63 := Choose  (AlgebraicDataType_4
    Local_59, AlgebraicDataType_4 Local_60, AlgebraicDataType_4
    Local_61, AlgebraicDataType_4 Local_62)
131     void Post_6 := Move  (void Local_9)
132     void Post_5 := Move  (void Local_5)
133     AlgebraicDataType_4 Post_1 := Move  (AlgebraicDataType_4
    Local_63)
134     FunctionType_3 Post_14 := Move  (FunctionType_3 Pre_14)
135     double Post_20 := Move  (double Pre_20)
136     AlgebraicDataType_4 Post_19 := Move  (AlgebraicDataType_4
    Pre_19)
137     FunctionType_5 Post_16 := Move  (FunctionType_5 Pre_16)
138     FunctionType_3 Post_12 := Move  (FunctionType_3 Pre_12)
139     void Post_4 := Move  (void Local_16)
140     void Post_3 := Move  (void Local_24)
141     FunctionType_3 Post_7 := Move  (FunctionType_3 Pre_7)
142     void Post_8 := Move  (void Pre_8)
143     double Post_0 := Move  (double Output_0)
```

```
144     AlgebraicDataType_4 Post_11 := Move  (AlgebraicDataType_4
    Pre_11)
145     void Post_13 := Move  (void Pre_13)
146   } next: --
147 }
```

Listing 65: Zwischenformat adsr

```
1   AlgebraicDataType_4 {
2     Attack: Attack()
3     Decay: Decay()
4     Release: Release()
5     Sustain: Sustain()
6   }
7
8   FunctionTypes {
9     FunctionType_0 = ComponentType(bool => double)
10    FunctionType_1 = ComponentType( => double)
11    FunctionType_2 = ComponentType(double => double)
12    FunctionType_3 = SessionType(double => double)
13    FunctionType_5 = SessionType( => double)
14  }
15
16  Block_12 {
17    env:
18    in: FunctionType_2, FunctionType_2, FunctionType_1,
    FunctionType_2
19    out: FunctionType_0
20    state:
21    locals:
22    init: BasicBlock_13
23    loop: BasicBlock_14
24
25    BasicBlock_13 {
26    } next: --
27
28    BasicBlock_14 {
29      FunctionType_0 Output_0 := Closure Block_15 (FunctionType_1
    Input_2, FunctionType_2 Input_0, FunctionType_2 Input_3,
    FunctionType_2 Input_1)
30    } next: --
31  }
32
33  Block_15 {
34    env: FunctionType_1, FunctionType_2, FunctionType_2,
    FunctionType_2
35    in: bool
36    out: double
37    state: double, AlgebraicDataType_4, void, void, void, void, void,
    FunctionType_3, void, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, FunctionType_3, void, FunctionType_3, void,
    FunctionType_5, void, double, AlgebraicDataType_4, double
38    locals: bool, bool, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, AlgebraicDataType_4,
    AlgebraicDataType_4, AlgebraicDataType_4, boolean, boolean,
    boolean, boolean, boolean, boolean, boolean, boolean, double,
    double, double, double, double, double, double, double, double,
    double, double, double
39    init: BasicBlock_16
40    loop: BasicBlock_17
41
42    BasicBlock_16 {
```

```
43      AlgebraicDataType_4 Post_1 := Construct "Release"
    AlgebraicDataType_4 ()
44      AlgebraicDataType_4 Post_9 := Construct "Decay"
    AlgebraicDataType_4 ()
45      AlgebraicDataType_4 Post_10 := Construct "Attack"
    AlgebraicDataType_4 ()
46      FunctionType_3 Post_14, void Post_15 := Start  (FunctionType_2
    Environment_1, double Pre_0)
47      double Post_20 := DoubleConstant 0.0 ()
48      AlgebraicDataType_4 Post_19 := Move  (AlgebraicDataType_4
    Post_1)
49      FunctionType_5 Post_16, void Post_17 := Start  (FunctionType_1
    Environment_0)
50      FunctionType_3 Post_12, void Post_13 := Start  (FunctionType_2
    Environment_3, double Pre_0)
51      void Post_2 := Move  (void Post_17)
52      FunctionType_3 Post_7, void Post_4 := Start  (FunctionType_2
    Environment_2, double Pre_0)
53      void Post_3 := Move  (void Post_15)
54      double Post_18 := DoubleConstant 1.0 ()
55      AlgebraicDataType_4 Post_11 := Construct "Sustain"
    AlgebraicDataType_4 ()
56      void Post_8 := Move  (void Post_4)
57      double Post_0 := Move  (double Post_20)
58      void Post_6 := Move  (void Post_13)
59      void Post_5 := Move  (void Post_17)
60    } next: --
61
62    BasicBlock_17 {
63      double Local_27, void Post_3 := Sample  (FunctionType_3 Pre_14,
    void Pre_3, double Pre_0)
64      double Local_29, void Post_4 := Sample  (FunctionType_3 Pre_7,
    void Pre_4, double Pre_0)
65      boolean Local_17 := Destruct "Release" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
66      boolean Local_15 := Destruct "Decay" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
67      boolean Local_18 := Destruct "Attack" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
68      boolean Local_21 := Destruct "T" bool (bool Input_0)
69      boolean Local_22 := Destruct "F" bool (bool Input_0)
70    } next: boolean Local_22 ? BasicBlock_18 : BasicBlock_19
71
72    BasicBlock_18 {
73      AlgebraicDataType_4 Local_9 := Move  (AlgebraicDataType_4
    Pre_19)
74    } next: BasicBlock_19
75
76    BasicBlock_19 {
77      double Local_33, void Post_5 := Sample  (FunctionType_5 Pre_16,
    void Pre_5)
78      double Local_23, void Post_6 := Sample  (FunctionType_3 Pre_12,
    void Pre_6, double Pre_0)
79      boolean Local_19 := Destruct "Sustain" AlgebraicDataType_4
    (AlgebraicDataType_4 Pre_1)
80    } next: boolean Local_19 ? BasicBlock_20 : BasicBlock_21
```

```
81
82    BasicBlock_20 {
83      double Local_31 := Move  (double Pre_0)
84    } next: boolean Local_21 ? BasicBlock_22 : BasicBlock_23
85
86    BasicBlock_21 {
87    } next: boolean Local_17 ? BasicBlock_24 : BasicBlock_25
88
89    BasicBlock_22 {
90      AlgebraicDataType_4 Local_11 := Move  (AlgebraicDataType_4
    Pre_1)
91    } next: BasicBlock_23
92
93    BasicBlock_23 {
94      AlgebraicDataType_4 Local_4 := Choose  (AlgebraicDataType_4
    Local_9, AlgebraicDataType_4 Local_11)
95    } next: BasicBlock_26
96
97    BasicBlock_24 {
98    } next: boolean Local_21 ? BasicBlock_27 : BasicBlock_28
99
100   BasicBlock_25 {
101   } next: boolean Local_15 ? BasicBlock_29 : BasicBlock_30
102
103   BasicBlock_26 {
104     double Output_0 := Choose  (double Local_24, double Local_34,
    double Local_25, double Local_31)
105     double Local_30, void Post_2 := Sample  (FunctionType_5 Pre_16,
    void Pre_2)
106     void Post_15 := Move  (void Pre_15)
107     void Post_17 := Move  (void Pre_17)
108     AlgebraicDataType_4 Post_10 := Move  (AlgebraicDataType_4
    Pre_10)
109     AlgebraicDataType_4 Post_9 := Move  (AlgebraicDataType_4 Pre_9)
110     double Post_18 := Move  (double Pre_18)
111   } next: boolean Local_15 ? BasicBlock_31 : BasicBlock_32
112
113   BasicBlock_27 {
114     AlgebraicDataType_4 Local_2 := Move  (AlgebraicDataType_4
    Pre_10)
115   } next: BasicBlock_28
116
117   BasicBlock_28 {
118     AlgebraicDataType_4 Local_5 := Choose  (AlgebraicDataType_4
    Local_9, AlgebraicDataType_4 Local_2)
119     double Local_28 := Call "-" (double Pre_0, double Local_29)
120     double Local_34 := Call "max" (double Pre_20, double Local_28)
121   } next: BasicBlock_26
122
123   BasicBlock_29 {
124     double Local_32 := Call "-" (double Pre_0, double Local_23)
125     double Local_25 := Call "max" (double Local_33, double
    Local_32)
126   } next: BasicBlock_26
127
128   BasicBlock_30 {
```

```
129     double Local_26 := Call "+" (double Pre_0, double Local_27)
130     double Local_24 := Call "min" (double Pre_18, double Local_26)
131   } next: BasicBlock_26
132
133   BasicBlock_31 {
134   } next: boolean Local_21 ? BasicBlock_33 : BasicBlock_34
135
136   BasicBlock_32 {
137   } next: boolean Local_18 ? BasicBlock_35 : BasicBlock_36
138
139   BasicBlock_33 {
140     bool Local_0 := Call "<=" (double Output_0, double Local_30)
141     boolean Local_20 := Destruct "F" bool (bool Local_0)
142   } next: boolean Local_20 ? BasicBlock_37 : BasicBlock_38
143
144   BasicBlock_34 {
145     AlgebraicDataType_4 Local_3 := Choose  (AlgebraicDataType_4
    Local_9, AlgebraicDataType_4 Local_14)
146   } next: BasicBlock_36
147
148   BasicBlock_35 {
149   } next: boolean Local_21 ? BasicBlock_39 : BasicBlock_40
150
151   BasicBlock_36 {
152     AlgebraicDataType_4 Post_1 := Choose  (AlgebraicDataType_4
    Local_8, AlgebraicDataType_4 Local_5, AlgebraicDataType_4 Local_3,
    AlgebraicDataType_4 Local_4)
153     FunctionType_3 Post_14 := Move  (FunctionType_3 Pre_14)
154     double Post_20 := Move  (double Pre_20)
155     AlgebraicDataType_4 Post_19 := Move  (AlgebraicDataType_4
    Pre_19)
156     FunctionType_5 Post_16 := Move  (FunctionType_5 Pre_16)
157     FunctionType_3 Post_12 := Move  (FunctionType_3 Pre_12)
158     FunctionType_3 Post_7 := Move  (FunctionType_3 Pre_7)
159     void Post_8 := Move  (void Pre_8)
160     double Post_0 := Move  (double Output_0)
161     AlgebraicDataType_4 Post_11 := Move  (AlgebraicDataType_4
    Pre_11)
162     void Post_13 := Move  (void Pre_13)
163   } next: --
164
165   BasicBlock_37 {
166     AlgebraicDataType_4 Local_7 := Move  (AlgebraicDataType_4
    Pre_1)
167   } next: BasicBlock_41
168
169   BasicBlock_38 {
170     AlgebraicDataType_4 Local_6 := Move  (AlgebraicDataType_4
    Pre_11)
171   } next: BasicBlock_41
172
173   BasicBlock_39 {
174     bool Local_1 := Call ">=" (double Output_0, double Pre_18)
175     boolean Local_16 := Destruct "T" bool (bool Local_1)
176   } next: boolean Local_16 ? BasicBlock_42 : BasicBlock_43
177
```

```
178   BasicBlock_40 {
179     AlgebraicDataType_4 Local_8 := Choose  (AlgebraicDataType_4
    Local_9, AlgebraicDataType_4 Local_12)
180   } next: BasicBlock_36
181
182   BasicBlock_41 {
183     AlgebraicDataType_4 Local_14 := Choose  (AlgebraicDataType_4
    Local_7, AlgebraicDataType_4 Local_6)
184   } next: BasicBlock_34
185
186   BasicBlock_42 {
187     AlgebraicDataType_4 Local_13 := Move  (AlgebraicDataType_4
    Pre_9)
188   } next: BasicBlock_44
189
190   BasicBlock_43 {
191     AlgebraicDataType_4 Local_10 := Move  (AlgebraicDataType_4
    Pre_1)
192   } next: BasicBlock_44
193
194   BasicBlock_44 {
195     AlgebraicDataType_4 Local_12 := Choose  (AlgebraicDataType_4
    Local_10, AlgebraicDataType_4 Local_13)
196   } next: BasicBlock_40
197 }
```

Listing 66: optimiertes adsr

```
1  FunctionTypes {
2    FunctionType_0 = ComponentType( => double)
3  }
4
5  Block_1 {
6    env:
7    in: double
8    out: FunctionType_0
9    state:
10   locals:
11   init: BasicBlock_2
12   loop: BasicBlock_3
13
14   BasicBlock_2 {
15   } next: --
16
17   BasicBlock_3 {
18     FunctionType_0 Output_0 := Closure Block_4 (double Input_0)
19   } next: --
20 }
21
22 Block_4 {
23   env: double
24   in:
25   out: double
26   state: double, double, double, double, double
27   locals: double, double, double, double, double, double, boolean,
   double, boolean, double, double, double, double, double, bool,
   double, double, double, double, double
28   init: BasicBlock_5
29   loop: BasicBlock_6
30
31   BasicBlock_5 {
32     double Local_13 := DoubleConstant 1.0 ()
33     double Local_19 := DoubleConstant 0.0 ()
34     double Post_0 := Move  (double Local_19)
35     double Local_5 := DoubleConstant 2.0 ()
36     double Post_3 := Move  (double Local_5)
37     double Post_2 := Move  (double Local_13)
38     double Post_1 := Move  (double Local_13)
39     double Local_17 := DoubleConstant 1.570796 ()
40     double Post_4 := Move  (double Local_17)
41   } next: --
42
43   BasicBlock_6 {
44     double Local_16 := Call "*" (double Pre_1, double Environment_0)
45     double Output_0 := Move  (double Pre_0)
46     double Local_15 := Call "/" (double Local_16, double Pre_4)
47     double Local_18 := Call "+" (double Output_0, double Local_15)
48     double Local_12 := Call "*" (double Pre_1, double Local_18)
49     bool Local_14 := Call ">=" (double Local_12, double Pre_2)
50     boolean Local_8 := Destruct "F" bool (bool Local_14)
51     double Local_9 := Guard  (double Local_18, boolean Local_8)
52     double Post_3 := Move  (double Pre_3)
53     boolean Local_6 := Destruct "T" bool (bool Local_14)
54     double Local_0 := Call "-" (double Pre_1)
```

```
55     double Local_2 := Guard  (double Local_0, boolean Local_6)
56     double Post_2 := Move  (double Pre_2)
57     double Local_4 := Call "*" (double Pre_3, double Pre_1)
58     double Local_7 := Call "-" (double Local_4, double Local_18)
59     double Local_10 := Guard  (double Local_7, boolean Local_6)
60     double Local_11 := Choose  (double Local_9, double Local_10)
61     double Post_0 := Move  (double Local_11)
62     double Local_1 := Guard  (double Pre_1, boolean Local_8)
63     double Local_3 := Choose  (double Local_1, double Local_2)
64     double Post_1 := Move  (double Local_3)
65     double Post_4 := Move  (double Pre_4)
66   } next: --
67 }
```

Listing 67: Zwischenformat triangle

```
1  FunctionTypes {
2    FunctionType_0 = ComponentType( => double)
3  }
4
5  Block_7 {
6    env:
7    in: double
8    out: FunctionType_0
9    state:
10   locals:
11   init: BasicBlock_8
12   loop: BasicBlock_9
13
14   BasicBlock_8 {
15   } next: --
16
17   BasicBlock_9 {
18     FunctionType_0 Output_0 := Closure Block_10 (double Input_0)
19   } next: --
20 }
21
22 Block_10 {
23   env: double
24   in:
25   out: double
26   state: double, double, double, double, double
27   locals: bool, double, double, double, double, double, double,
   double, double, double, boolean
28   init: BasicBlock_11
29   loop: BasicBlock_12
30
31   BasicBlock_11 {
32     double Post_2 := DoubleConstant 1.0 ()
33     double Post_0 := DoubleConstant 0.0 ()
34     double Post_3 := DoubleConstant 2.0 ()
35     double Post_1 := Move  (double Post_2)
36     double Post_4 := DoubleConstant 1.570796 ()
37   } next: --
38
39   BasicBlock_12 {
40     double Local_1 := Call "*" (double Pre_1, double Environment_0)
41     double Output_0 := Move  (double Pre_0)
42     double Local_9 := Call "/" (double Local_1, double Pre_4)
43     double Local_3 := Call "+" (double Output_0, double Local_9)
44     double Local_2 := Call "*" (double Pre_1, double Local_3)
45     bool Local_0 := Call ">=" (double Local_2, double Pre_2)
46     double Post_3 := Move  (double Pre_3)
47     boolean Local_10 := Destruct "T" bool (bool Local_0)
48     double Post_2 := Move  (double Pre_2)
49   } next: boolean Local_10 ? BasicBlock_13 : BasicBlock_14
50
51   BasicBlock_13 {
52     double Local_6 := Call "-" (double Pre_1)
53     double Local_5 := Call "*" (double Pre_3, double Pre_1)
54     double Local_4 := Call "-" (double Local_5, double Local_3)
55   } next: BasicBlock_15
```

```
56
57   BasicBlock_14 {
58     double Local_8 := Move  (double Local_3)
59     double Local_7 := Move  (double Pre_1)
60   } next: BasicBlock_15
61
62   BasicBlock_15 {
63     double Post_0 := Choose  (double Local_8, double Local_4)
64     double Post_1 := Choose  (double Local_7, double Local_6)
65     double Post_4 := Move  (double Pre_4)
66   } next: --
67 }
```

Listing 68: optimiertes triangle

```
1  FunctionTypes {
2    FunctionType_0 = ComponentType( => double)
3  }
4
5  Block_1 {
6    env:
7    in: double
8    out: FunctionType_0
9    state:
10   locals:
11   init: BasicBlock_2
12   loop: BasicBlock_3
13
14   BasicBlock_2 {
15   } next: --
16
17   BasicBlock_3 {
18     FunctionType_0 Output_0 := Closure Block_4 (double Input_0)
19   } next: --
20 }
21
22 Block_4 {
23   env: double
24   in:
25   out: double
26   state: double, double, double, double
27   locals: double, double, double, double, boolean, double, boolean,
   double, double, double, double, bool, double, double, double, double, double
28   init: BasicBlock_5
29   loop: BasicBlock_6
30
31   BasicBlock_5 {
32     double Local_12 := DoubleConstant 1.570796 ()
33     double Local_11 := Call "/" (double Environment_0, double
   Local_12)
34     double Local_14 := DoubleConstant 0.0 ()
35     double Post_1 := Move  (double Local_14)
36     double Local_15 := DoubleConstant 1.0 ()
37     double Post_0 := Move  (double Local_15)
38     double Post_2 := Move  (double Local_11)
39     double Post_3 := Move  (double Local_15)
40   } next: --
41
42   BasicBlock_6 {
43     double Local_13 := Call "+" (double Pre_1, double Pre_2)
44     bool Local_10 := Call ">=" (double Local_13, double Pre_3)
45     boolean Local_4 := Destruct "T" bool (bool Local_10)
46     double Local_0 := Call "-" (double Local_13, double Pre_3)
47     double Local_2 := Guard  (double Local_0, boolean Local_4)
48     boolean Local_6 := Destruct "F" bool (bool Local_10)
49     double Local_1 := Guard  (double Local_13, boolean Local_6)
50     double Output_0 := Move  (double Pre_0)
51     double Local_5 := Call "-" (double Output_0)
52     double Local_8 := Guard  (double Local_5, boolean Local_4)
53     double Post_2 := Move  (double Pre_2)
54     double Local_7 := Guard  (double Output_0, boolean Local_6)
```

```
55     double Local_3 := Choose  (double Local_1, double Local_2)
56     double Post_1 := Move  (double Local_3)
57     double Local_9 := Choose  (double Local_7, double Local_8)
58     double Post_0 := Move  (double Local_9)
59     double Post_3 := Move  (double Pre_3)
60   } next: --
61 }
```

Listing 69: Zwischenformat rectangle

```
1  FunctionTypes {
2    FunctionType_0 = ComponentType( => double)
3  }
4
5  Block_7 {
6    env:
7    in: double
8    out: FunctionType_0
9    state:
10   locals:
11   init: BasicBlock_8
12   loop: BasicBlock_9
13
14   BasicBlock_8 {
15   } next: --
16
17   BasicBlock_9 {
18     FunctionType_0 Output_0 := Closure Block_10 (double Input_0)
19   } next: --
20 }
21
22 Block_10 {
23   env: double
24   in:
25   out: double
26   state: double, double, double, double
27   locals: double, double, double, double, double, bool, boolean
28   init: BasicBlock_11
29   loop: BasicBlock_12
30
31   BasicBlock_11 {
32     double Local_4 := DoubleConstant 1.570796 ()
33     double Post_2 := Call "/" (double Environment_0, double Local_4)
34     double Post_1 := DoubleConstant 0.0 ()
35     double Post_0 := DoubleConstant 1.0 ()
36     double Post_3 := Move  (double Post_0)
37   } next: --
38
39   BasicBlock_12 {
40     double Local_4 := Call "+" (double Pre_1, double Pre_2)
41     bool Local_5 := Call ">=" (double Local_4, double Pre_3)
42     boolean Local_6 := Destruct "T" bool (bool Local_5)
43     double Output_0 := Move  (double Pre_0)
44     double Post_2 := Move  (double Pre_2)
45   } next: boolean Local_6 ? BasicBlock_13 : BasicBlock_14
46
47   BasicBlock_13 {
48     double Local_0 := Call "-" (double Local_4, double Pre_3)
49     double Local_2 := Call "-" (double Output_0)
50   } next: BasicBlock_15
51
52   BasicBlock_14 {
53     double Local_1 := Move  (double Local_4)
54     double Local_3 := Move  (double Output_0)
55   } next: BasicBlock_15
```

```
56
57   BasicBlock_15 {
58     double Post_1 := Choose  (double Local_1, double Local_0)
59     double Post_0 := Choose  (double Local_3, double Local_2)
60     double Post_3 := Move  (double Pre_3)
61   } next: --
62 }
```

Listing 70: optimiertes rectangle

Literaturverzeichnis

BC94 Preston Briggs, Keith D. Cooper: Effective Partial Redundancy Elimination. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94), 1994, 159-170

BG97 Rastislav Bodík, Rajiv Gupta: Partial dead code elimination using slicing transformations. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97), 1997, 159-170

BGG05 J.O. Blech, L. Gesellensetter, S. Glesner: Formal verification of dead code elimination in Isabelle/HOL. In: Third IEEE International Conference on Software Engineering and Formal Methods (SEFM '05), 2005, 200-209

CFRWZ89 R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck: An Efficient Method of Computing Static Single Assignment Form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89), 1989, 25-35

Cho13 Fred Chow: Intermediate Representation - ACM Queue. http://queue.acm.org/detail.cfm?id=2544374 , Stand: 05.11.2014

Cli95 Cliff Click: Global code motion/global value numbering. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95), 1995, 246-257

FKCX94 Lawrence Feigen, David Klappholz, Robert Casazza, Xing Xue: The Revival Transformation. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94), 1994, 421-434

GCC:GCSE Global CSE/Partial Redundancy Elimination. http://gcc.gnu.org/news/gcse.html , Stand: 19.09.2014

JHM04 Wesley M. Johnston, J. R. Paul Hanna, Richard J. Millar: Advances in Dataflow Programming Languages. In: ACM Comput. Surv. 36 (2004), Nr. 1, 1-34. http://doi.acm.org/10.1145/1013208.1013209

K3BA Kapitel 3: Boolesche Algebra. http://www.staff.uni-giessen.de/~gc1099/dm/DM%20II%20-%20Kapitel%203%20-%20Boolesche%20Algebra.pdf , Stand: 05.11.2014

KCLLTC99 Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, Fred Chow: Partial Redundancy Elimination in SSA Form. In: ACM Trans. Program. Lang. Syst. 21 (1999), Nr. 3, 627-676. http://doi.acm.org/10.1145/319301.319348

KRS94 Jens Knoop, Oliver Rüthing, Bernhard Steffen: Partial dead code elimination. In: SIGPLAN Not. 29 (1994), Nr. 6, 147-158. http://doi.acm.org/10.1145/773473.178256

KSK09 Uday Khedker, Amitabha Sanyal, Bageshri Karkare: Data Flow Analysis: Theory and Practice. CRC Press, Inc., 2009

McC56 E.J. McCluskey: Minimization of Boolean functions. In: Bell System Technical Journal, The 35 (1956), Nr. 6, 1417-1444. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6769983&isnumber=6769977

PSS03 V.K. Paleri, Y.N. Srikant, P. Shankar: Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. In: Science of Computer Programming 48 (2003), Nr. 1, 1-20. http://dx.doi.org/10.1016/S0167-6423(02)00083-7

WG84 William M. Waite, Gerhard Goos: Compiler construction. Third Edition. Springer-Verlag New York, 1996, http://www.researchgate.net/publication/2545268_Compiler_Construction/file/79e41509284a50e6b3.pdf

WL14a Baltasar Trancón y Widemann, Markus Lepper: Foundations of Total Functional Data-Flow Programming. In: Proceedings 5th Workshop on Mathematically Structured Functional Programming, Grenoble, France, 12 April 2014 (Electronic Proceedings in Theoretical Computer Science), 2014, 143-167

WL14b Baltasar Trancón y Widemann, Markus Lepper: [Demo Abstract] Sound and Soundness: Practical Total Functional Data-flow Programming. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design (FARM '14), 2014, 35-36

WZ91 Mark N. Wegman, F. Kenneth Zadeck: Constant Propagation with Conditional Branches. In: ACM Trans. Program. Lang. Syst. 13 (1991), Nr. 2, 181-210. http://doi.acm.org/10.1145/103135.103136

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Alle von mir aus anderen Veröffentlichungen übernommenen Passagen sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ilmenau, 13.03.2015

Andreas Loth

Digital unterschrieben von Andreas Loth
DN: c=DE, o=Technische Universitaet Ilmenau, cn=Andreas Loth, email=andreas.loth@tu-ilmenau.de
Datum: 2015.03.12 17:23:06 +01'00'

Andreas Loth