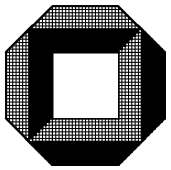


Universität Karlsruhe (TH)
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Prof. Goos

Eine formale Semantik für SSA Zwischensprachen in Isabelle/HOL

Jan Olaf Blech

Diplomarbeit

Betreuerin
Dr. Sabine Glesner

Verantwortlicher
Prof. Dr. Goos

März 2004

Hiermit erkläre ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Kurzfassung

In dieser Arbeit haben wir eine SSA basierte Zwischensprache formal spezifizert und formal in Isabelle/HOL nachgewiesen, dass eine einfache Form von Codeerzeugung die Semantik der transformierten Programme erhält. Dazu haben wir die Semantik von SSA formal angegeben und diese Formalisierung in zwei Teilbereiche unterteilt: einen, der den Datenfluss in Grundblöcken beschreibt und einen zweiten, der den Kontrollfluss und Speicher SSA behandelt. Wir haben weiterhin neben den SSA Datenstrukturen auch deren formale Semantik operational mittels einer Interpretationsfunktion formalisiert. Diese lässt sich ohne große Änderung in ML übersetzen. Weiterhin haben wir eine einfache Maschinensprache formalisiert. Wir haben hier ebenfalls eine formale Semantik operational durch eine Interpreterfunktion spezifiziert.

Es ist uns gelungen, ein zentrales Theorem zu beweisen: Jede topologische Sortierung eines Datenflussgraphen ist eine gültige Codeauswertungsreihenfolge bzw. Erzeugungsreihenfolge. Den Beweis haben wir maschinell mit Isabelle/HOL geführt. Dabei setzen wir im Beweis die Existenz einer Projektionsfunktion voraus. Wir geben diese Projektionsfunktion zwar an, es ist uns jedoch noch nicht gelungen, im Theorembeweiser Isabelle/HOL zu zeigen, dass sie die gewünschten Eigenschaften hat. Die fehlenden Eigenschaften sind in einem Papier und Bleistift Beweis trivial und lassen sich in zukünftigen Arbeiten sicherlich nachweisen.

Insgesamt haben wir damit in dieser Diplomarbeit nachweisen können, dass Transformationen in Übersetzer-Back-Ends formal in einem Theorembeweiser spezifiziert und verifiziert werden können.

Inhaltsverzeichnis

1. Einführung

Korrektheit ist ein wichtiges Kriterium für die Qualität von Software. Um korrekte Programme zu gewährleisten benötigt man korrekte Übersetzer. Neben der Korrektheit ist eine hohe Ausführungsgeschwindigkeit ein Qualitätskriterium für Software. Dafür benötigt man optimierende Übersetzer. In dieser Arbeit beschäftigen wir uns mit der Erstellung verifizierter, also korrekter optimierender Übersetzer.

Wir spezifizieren eine SSA basierte Zwischensprache und führen einen Beweis. Wir weisen die Korrektheit von Codeerzeugung in eine einfache Maschinensprache nach. Damit liefern wir auch ein Beispiel für die Praxistauglichkeit unserer Spezifikation für größere Beweise. Zwischensprachen werden in Übersetzern als Programmrepräsentation zwischen Front-End und Back-End verwendet. Viele Optimierungen können auf der Zwischensprachenebene stattfinden. Auf der Zwischensprache findet die Codeerzeugung statt.

In dieser Arbeit betrachten wir SSA basierte Zwischensprachen. SSA basierte Zwischensprachen sind eine moderne Klasse von Zwischensprachen. SSA steht für static single assignment -deutsch: statische einmal Zuweisung. Sie werden heute in vielen Übersetzern verwendet. Sie zeichnen sich dadurch aus, dass an jede Variable im Programmtext der Zwischensprache nur einmal zugewiesen wird. Wir wollen eine SSA basierte Zwischensprache in dem Theorembeweiser Isabelle/HOL formalisieren und einen Beweis zur Korrektheit der Codeerzeugung führen. Maschinell geführte Beweise zeichnen sich durch einen wesentlich höheren Detailiertheitsgrad gegenüber konventionellen Beweisen aus.

Speicher SSA ist eine SSA Darstellung mit Befehlen für Speicherzugriffe. Auch Speicher SSA betrachten wir in dieser Diplomarbeit. Auf unsere Beweise haben diese zusätzlichen Befehle aber eher einen geringen Einfluss, da sie nur zusätzliche Abhängigkeiten im Datenfluss darstellen. Auf Optimierungen auf SSA Sprachen gehen wir in dieser Diplomarbeit nicht ein. Allerdings sollen die Resultate aus dieser Arbeit benutzt werden können um Korrektheit von Optimierungen zu garantieren. Deshalb wollen wir nicht eine starre Übersetzungsfunktion als korrekt nachweisen, sondern allgemeinere Kriterien betrachten. SSA Sprachen lassen sich durch Graphen als Überlagerung von Daten- und Steuerfluss darstellen.

1.1. Lösungsansatz

Wir formalisieren in dieser Arbeit eine SSA basierten Zwischensprache im Theorembeweiser Isabelle/HOL. Abstrakte Datentypen lassen sich in Isabelle/HOL funktional spezifizieren. Zusätzlich können wir in Isabelle/HOL Korrektheitskriterien und Voraus-

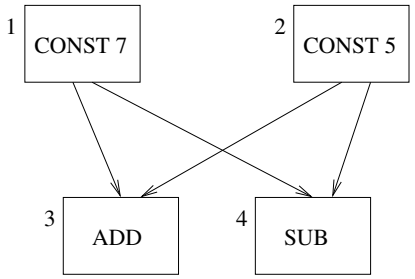


Abbildung 1.1.: Ein SSA-Graph

setzungen formalisieren und unsere Beweise führen.

Wir formalisieren die Semantik für die SSA basierte Zwischensprache mit einer Interpreterfunktion. Auch die Semantik für den Maschinencode wird durch eine Interpreterfunktion spezifiziert. Wir postulieren, dass jede topologische Sortierung eines Grundblocks auf der Zwischensprache eine gültige Codeerzeugungsreihenfolge ist. Die Behauptung formalisieren und beweisen wir in Isabelle/HOL.

Damit haben wir ein allgemeines Kriterium, wie Code aus der Zwischensprache generiert werden kann. Dieses Kriterium liefert viel Spielraum für die Optimierung der Befehlsanordnung.

1.2. Beispiel und Motivation

SSA Sprachen lassen sich als Graphen darstellen. Folgendes Beispiel zeigt einen ganz einfachen SSA Graphen und mögliche Codeerzeugungsreihenfolgen. In Abbildung 1.1 ist der SSAGraph gezeigt. Knoten sind die Befehle, Kanten stellen den Datenfluss dar. Wir nummerieren die Variablen durch (Wertnummern). Hier sind sie mit 1-4 bezeichnet. Jede Operation bekommt eine neue Wertnummer. Gültige Code Reihenfolgen sind in diesem Beispiel dann:

- $[1 : CONST\ 7, 2 : CONST\ 5, 3 : ADD\ 12, 4 : SUB\ 12]$,
- $[2 : CONST\ 5, 1 : CONST\ 7, 3 : ADD\ 12, 4 : SUB\ 12]$,
- $[1 : CONST\ 7, 2 : CONST\ 5, 4 : SUB\ 12, 3 : ADD\ 12]$,
- $[2 : CONST\ 5, 1 : CONST\ 7, 4 : SUB\ 12, 3 : ADD\ 12]$.

Auch können wir in diesem Beispiel die Reihenfolge der Operanden des ADD-Operators vertauschen, da ADD kommutativ ist. Dieses Beispiel erläutert den Zusammenhang zwischen Maschinencode und Zwischensprache. In diesem Beispiel ist zu sehen, dass die vier topologischen Sortierungen des Graphen gültige Codeerzeugungsreihenfolgen sind. Wir weisen in dieser Arbeit nach, dass alle topologischen Sortierungen gültige Codeerzeugungsreihenfolgen sind.

1.3. Übersicht über die Arbeit

In Kapitel 2 stellen wir verwandte Arbeiten vor. In Kapitel 3 geht es um allgemeinere Fragen zum Einsatz von Theorembeweisern. Insbesondere gehen wir auf Isabelle/HOL ein. Kapitel 4 führt die SSA Darstellung ein. In Kapitel 5 betrachten wir unsere Formalisierungen der Datenstrukturen und Funktionen unserer SSA Darstellung. In Kapitel 6 stellen wir dann unseren Beweis dar. In Kapitel 7 bewerten wir unsere Arbeit und geben einen umfangreichen Ausblick.

2. Verwandte Arbeiten

In diesem Kapitel stellen wir Arbeiten zur Übersetzerverifikation und andere mit dem Diplomarbeitsthema in Verbindung stehende Arbeiten vor. Wir betrachten die vorgestellten Arbeiten im Kontext dieser Diplomarbeit.

Die erste Arbeit, in der die Korrektheit eines Übersetzers nachgewiesen wird, ist [15]. Hier wird die Korrektheit eines Übersetzers, der Code für einfache arithmetische Ausdrücke generiert, nachgewiesen. Der Beweis wurde per Hand geführt.

Neuere Arbeiten entstanden unter anderem im Verifix Projekt. Das Verifix Projekt hatte das Ziel korrekte Übersetzer für realistische Programmiersprachen zu bauen. Einen Überblick gibt[13]. Hier wird auch ein Korrektheitsbegriff eingeführt und erläutert. Nach [13] ist eine Übersetzung genau dann als korrekt anzusehen, wenn die Interpretation des Programmtextes und die Interpretation des übersetzten Programmtextes, also das Ausführen des Maschinencodes, das gleiche beobachtbare Verhalten oder einen Fehlerzustand liefern. Ein Fehlerzustand kann zum Beispiel wegen Ressourcenmangel auftreten. Diesen Korrektheitsbegriff verwenden wir auch für diese Diplomarbeit. Weiterhin zeigt der Artikel, wie man den Übersetzungsvorgang in verschiedene Phasen einteilen und getrennt verifizieren kann. So kann man zum Beispiel lexikalische Analyse, syntaktische Analyse, semantische Analyse, Transformationsphase und Codeerzeugung getrennt verifizieren und daraus einen gesamt Korrektheitsbeweis zusammensetzen. Man kann auf diese Weise einen komplett verifizierten Übersetzer erhalten. Dadurch, dass wir uns in dieser Diplomarbeit mit einer Zwischensprache und der Codeerzeugung befassen, brauchen wir eine solche Dekompositionsmöglichkeit.

Die maschinelle Verifikation eines Übersetzers von einem Lisp Dialekt in Transputercode wird in [7] und [6] betrachtet. Hier wurde das Proof Verification System(PVS) eingesetzt. Als Lisp Dialekt wurde ComLisp - eine Teilmenge von Common Lisp - ausdrucksstark genug um als Übersetzerimplementierungssprache zu dienen - verwendet. Der Übersetzer transformiert in einer ersten Phase ComLisp Code in eine Stack Intermediate Language (SIL). Dann wird von SIL in eine weitere Zwischensprache: C^{int} übersetzt. Schließlich wird Transputerassemblercode (TASM) erzeugt und zum Schluss Maschinencode. Die Autoren präsentieren ein Rahmenwerk für die Verifikation ihres Übersetzers im Theorembeweiser PVS. Den Autoren ist es gelungen beachtliche Teile des Übersetzers maschinell zu verifizieren.

Der Artikel [11] beschreibt den gleichen Lisp Übersetzer. Er benutzt das Problem des Erkennens eines Virus, das beim Übersetzen immer an das Compilat angefügt wird, als Einstieg. Darauf aufbauend benötigt er einen anderen Korrektheitsbegriff als [13]: Das Virus muß nicht unbedingt am beobachtbaren Verhalten etwas ändern, soll aber

trotzdem erkannt werden. Daraus folgend wird eine eigene Bootstrappingproblematik diskutiert.

In [22] geht es um die maschinelle Verifikation von Übersetzer Front-Ends. In der Arbeit werden einige Beweise geführt und es wird ein Rahmenwerk für die Verifikation des gesamten Übersetzer Front-Ends geliefert. Die Arbeit führt einige wichtige Begriffe für die Klassifikation von Beweisen ein:

- Ein Programm ist **proven correct** wenn ein Papier und Bleistift Beweis erfolgreich gezeigt hat, dass das Programm seine Spezifikation erfüllt.
- Ein Programm ist **provably correct** wenn ein Papier und Bleistift Korrektheitsbeweis soweit geführt wurde, dass absehbar ist, dass er erfolgreich zu Ende geführt werden kann. Zum Beispiel kann man in einer Fallunterscheidung über im wesentlichen ähnliche Fälle sich nur die interessantesten herausgesucht haben.
- Ein Programm ist **mechanically proven correct** wenn in einem Theorembeweiser erfolgreich gezeigt wurde, dass das Programm seine Spezifikation erfüllt.
- Ein Programm ist **mechanically provably correct** wenn in einem Theorembeweiser einige nicht triviale Lemmata für einen Korrektheitsbeweis geführt wurden und absehbar ist, dass die gewählte Formalisierung sich für einen vollständigen Beweis in einem Theorembeweiser eignet.

Diese Arbeit [22], aber auch die Arbeiten [7, 6] zeigen, dass es unterschiedliche Qualitäten von Beweisen gibt. Insbesondere sind Beweise in Theorembeweisern wesentlich schwieriger zu führen als Papier und Bleistift Beweise. Sie sind allerdings auch wesentlich genauer. Ausserdem zeigen die Arbeiten, dass für Korrektheitsbeweise von Programmen keine Standardtechniken existieren.

Die Arbeit[16] beschreibt die Verifikation der lexikalischen Analyse eines Übersetzers. Die Verifikationsarbeiten wurden in Isabelle/HOL ausgeführt. Die Arbeit beschreibt die Formalisierung und beschreibt detailliert den Beweis. An vielen Stellen werden Alternativen zur gewählten Formalisierung bzw. zum gewählten Beweis diskutiert. Auch diese Arbeit zeigt, dass es noch keine ingenieurmäßige Methode zur Benutzung von Theorembeweisern gibt.

Weitere Beiträge haben wir schon durch unsere Arbeiten[10, 3] auf dem Gebiet der Verifikation der Konstantenfaltung in verschiedenen Ganzzahlarithmetiken geleistet. Hier wurden die Beweise auch maschinell mit dem Isabelle/HOL System durchgeführt.

In [9] wird ein Checker für optimierende Übersetzer vorgestellt. So ein Checker rechnet das von einem Übersetzer gelieferte Ergebnis nach. Dazu erhält er ein vom Übersetzer generiertes Lösungsprotokoll. Für jede Übersetzerphase wird ein eigener Checker geschrieben. Um Korrektheit des Übersetzungsvorgangs zu garantieren, reicht es aus, die Checker zu verifizieren. Ein Checker kann sehr viel einfacher sein als der Übersetzer, da er sich nicht um Optimierungen zu kümmern braucht. Betreffend der Laufzeit ist er es in jedem Fall(Vorausgesetzt $P \neq NP$), da viele Optimierungsprobleme in NP liegen und die Ergebnisse somit in P verifiziert werden können. Der Checkeransatz ist in Abbildung 7.1

illustriert. Wir gehen in Kapitel 7 bei der Diskussion der Ergebnisse dieser Diplomarbeit darauf ein.

Diese Diplomarbeit beschäftigt sich mit Beweisen auf einer SSA Darstellung und mit der Spezifikation einer solchen. In [8] werden die Spezifikationstechniken: Abstract State Machines und Natural Semantics miteinander verglichen. Also Spezifizierungstechniken, mit denen man sich von Zustand zu Zustand "hangelt". In dieser Diplomarbeit werden Spezifikation mit Hilfe eines maschinellen Beweissystems funktional erstellt. Wobei die Formalisierung so gewählt ist, dass auch imperative Anteile (d.h. ein Zustandsbegriff) erkennbar sind. Wir können aus einer nicht funktionalen Sicht die Abarbeitung eines Programms in Zustände einteilen. Die Abarbeitung eines Grundblocks ist dann eine atomare Operation (Funktionsaufruf). Der aktuelle Grundblock zusammen mit dem aktuellen Speicherinhalt ist gerade ein Zustand. Dieses Prinzip spiegelt sich auch in unserer (funktionalen) Formalisierung wieder.

Eine Arbeit, die ich exemplarisch für die prinzipielle Vorgehensweise bei Beweisen mit dem Isabelle System anführen möchte, ist [14]. Hier geht es um "Verified Java Bytecode Verification", also um die Verifikation des Bytecodeverifiers. Bemerkenswert ist die Vorgehensweise und der Aufbau der Arbeit. Nach einer Einleitung wird ein abstraktes Rahmenwerk vorgestellt, das in den darauf folgenden Kapiteln instanziiert und erweitert wird. Die Arbeit liefert somit auch eine Methodik, größere Beweise in Theorembeweisern zu führen.

Insgesamt zeigen die verwandten Arbeiten, dass noch kein Standard-Rezept bezüglich des Theorembeweisens existiert. Es gibt verschiedene Qualitäten von Beweisen, wobei maschinell geführte Beweise genauer sind als Papier und Bleistift Beweise. Nicht immer ist es Ziel einen ganzen Beweis maschinell zu führen. Vielmehr geht es in vielen Arbeiten darum zu zeigen, dass ein Beweis prinzipiell maschinell durchführbar ist.

3. Der Theorem Beweiser Isabelle/HOL

Unsere Formalisierungen und Beweise haben wir im Theorembeweiser Isabelle/HOL durchgeführt. In diesem Kapitel geht es um Besonderheiten des Isabelle/HOL Systems und unsere Gründe Isabelle/HOL zu benutzen.

Das Theorem-Beweissystem Isabelle wurde an der Technischen Universität München und der Cambridge University entwickelt. Die HOL = Higher Order Logic ist die am häufigsten gebrauchte Logik für Isabelle. Eine Übersicht gibt[19]. Eine Einführung [18]. HOL gestattet sowohl das Spezifizieren und Beweisen von prädikatenlogischen Formeln, als auch Beweise über induktiv definierte Datentypen (also auch abstrakte Datentypen).

In diesem Kapitel stellen wir zunächst Isabelle als Spezifikationssprache vor. Weiterhin stellen wir einige prinzipielle Beweistechniken vor. Schließlich stellen wir weitere Eigenschaften von Isabelle vor, die uns die Arbeit in den nachfolgenden Kapiteln erleichtern.

3.1. Isabelle/HOL als Spezifikationssprache

Isabelle basiert auf der funktionalen Programmiersprache ML. Es stehen daher zur Spezifikation insbesondere auch die Elemente funktionaler Programmiersprachen zur Verfügung. Als Datentypen stehen Mengen, Tupel und Bäume sowie Atomare Datentypen zur Verfügung. Listen sind als Spezialfälle von Bäumen definiert (Schlüsselwort "datatype"). Die Konzepte Menge und Baum sind beliebig kombinierbar. Es lassen sich also Mengen von Bäumen, die wiederum Mengen als Attribute haben spezifizieren. Man kann primitiv rekursive Funktionen in Isabelle/HOL relativ leicht spezifizieren: Beispiel:

$consts\ is\_in\_list\ ::\ "Element \Rightarrow Element\ list \Rightarrow bool"$
$primrec$
$"is\_in\_cl\ a\ [] = False"$
$"is\_in\_cl\ a\ (x\#xs) = ((a = x) \vee (is\_in\_cl\ a\ xs))"$

Diese Funktion nimmt ein Element und eine Liste von Elementen und liefert einen Wahrheitswert zurück. Im Basisfall - der leeren Liste - liefert diese Funktion "falsch" als Ergebnis. Wenn das Argument a gleich dem ersten Element der Liste ist oder wenn is_in_cl für die Restliste ohne das erste Element gilt, wird "wahr" zurückgegeben. Diese Funktion liefert "wahr" als Ergebnis, wenn das Element (das als erster Parameter übergeben wurde) in der Liste enthalten ist. Sonst liefert sie "falsch". Man kann in Isabelle auch nicht primitiv rekursive Funktionen spezifizieren. Dazu muss allerdings

ein Terminierungsbeweis geführt werden. Nicht terminierende Funktionen lassen sich in Isabelle/HOL nicht spezifizieren.

Neben dem Ausprogrammieren von Funktionen kann man natürlich auch Spezifikationen mit Hilfe der bekannten Notationen der Prädikatenlogik erstellen. Isabelle/HOL stellt einige vordefinierte Listenoperationen bereit: a#l fügt Element a an Liste l an. k@l fügt Liste k und Liste l zu einer Liste zusammen.

Insgesamt sind die Spezifizierungsmöglichkeiten in Isabelle/HOL so groß, dass sich eine Vielzahl unterschiedlichster Sachen bequem spezifizieren lassen.

3.2. Grundlegende Beweistechniken

In Isabelle formuliert man Beweisziele als Lemmata oder Theoreme. Lemmata, die sich durch einfache Umformungen beweisen lassen, kann Isabelle in der Regel automatisch beweisen. Beispiel für ein Lemma: Ist x das letzte Element einer Liste, so ist x in der Liste enthalten.

$$lemma\ lastelisin: \ ''\forall\ x.\ is\_in\_cl\ x\ \ (l@[x])''$$

Bei komplizierteren Beweisen wird der Suchraum schnell zu groß. Ein großes Lemma kann in kleinere Lemmata zerlegt und diese einzeln bewiesen werden. Die kleineren Lemmata können dann als einzelne Umformungsschritte beim Beweis des großen Lemmas angewandt werden. Ein typischer Fall für so ein kleines Lemma ist zum Beispiel die Substitution einer allquantifizierten Variable durch eine Konstante.

Grundsätzlich können Beweise vorwärts oder rückwärts geführt werden. Bei einem Vorwärtsbeweis folgert man aus den Voraussetzungen das Beweisziel. Bei einem Rückwärtsbeweis wird ein Beweisziel in mehrere Unterziele aufgespalten, die dann bewiesen werden müssen.

Eine Technik des Rückwärtsbeweisens wird auch Taktik genannt. Die Fallunterscheidung ist eine Taktik. In der Regel wird Isabelle selbst keine geeigneten Fallunterscheidungen finden. Mit der Beweistaktik “case_tac” kann man daher eine Fallunterscheidung manuell durchführen. Hier wird das Beweisziel in zwei Unterziele aufgespalten.

Bei einem Induktionsbeweis (mit “induct_tac”) wird das Beweisziel auch in zwei Unterziele aufgespalten: Induktionsbasisfall und Induktionsschritt, die getrennt bewiesen werden müssen. Induktionen können natürlich nur auf Variablen von Typen, auf denen ein Induktionsprinzip definiert ist, durchgeführt werden. Das sind in HOL die natürlichen Zahlen und Bäume (inkl. Listen). Das oben angegebene Lemma lässt sich mit einem Induktionsbeweis über l lösen.
Basisfall: is_in_cl x [x]
Induktionsschritt: is_in_cl x l@[x] $\Longrightarrow$ is_in_cl x (a#l)@[x]
Die Einzelnen Beweisziele lassen sich dann mit Vorwärtsbeweisregeln lösen.

3.3. Weiteres zu Isabelle

Im Isabelle System lassen sich Operatoren überladen. Sie lassen sich dann Funktionen zuordnen. Zum Beispiel kann man der oben angegebenen is_in_list Funktion das Symbol $\in$ als infix-Operator zuordnen.

Es gibt verschiedene Notationen, um in Isabelle/HOL einen Beweis zu führen. Neben einer eher an den technischen Bedürfnissen des Beweissystems orientierten Schreibweise gibt es mit der Isar/HOL Notation [17] eine für Menschen besser zu lesende Notation. Diese verwenden wir für unser Haupttheorem in Kapitel 6.

4. SSA Zwischensprachen

In diesem Kapitel beschreiben wir SSA basierte Zwischensprachen.

SSA Darstellungen wurden zuerst 1988 [1, 20, 4, 5] eingeführt. SSA steht für static-single-assignment (deutsch: statische Einmalzuweisung). Im Programmtext der Zwischensprache wird jeder Variablen nur genau einmal ein Wert zugewiesen (statische Einmalzuweisung). Im (dynamischen) Programmablauf kann einer Variablen natürlich beliebig oft ein Wert zugewiesen werden. Wir können die Variablen einfach durchnummerieren und ihnen keine weiteren Namen geben. Eine solche Nummer wird auch Wertnummer genannt. Eine SSA-Darstellung lässt sich leicht mit einem Durchlauf durch den Syntaxbaum generieren. Die SSA Form hat verschiedene Vorteile gegenüber anderen Zwischendarstellungen. Benutzt-Definiert Beziehungen sind durch die statische Einmalvergabe der Wertnummern sofort ersichtlich. Auch der Lebendigkeitsbereich von Variablen und die Verwendung gleicher Teilausdrücke sind direkt aus der SSA Darstellung abzulesen.

Ein Programm in SSA Darstellung besteht aus Grundblöcken. Dieses lässt sich mit Kontrollfluss und Datenflusskanten als ein gerichteter Graph interpretieren. Die einzelnen Grundblöcke in SSA basierten Zwischensprachen lassen sich auch als Graph interpretieren. Hier bildet der Datenfluss in einem Grundblock sogar einen gerichteten azyklische Graphen. Innerhalb eines Grundblocks wird nur der Datenfluss dargestellt. Die Datenflusskanten innerhalb eines Grundblocks stellen gerade die Abhängigkeiten von vorangegangenen Ergebnissen dar.

Die Knoten in einem Grundblock stellen die einzelnen Befehle dar, wie z.B. Addition und Sprung. Die Kanten stellen den Datenfluss dar. Da der Wert einer Variablen im allgemeinen vom Kontrollfluss abhängt, gibt es ϕ-Knoten, die als Argumente Variablennamen aus vorangegangenen Grundblöcken haben($x_{i+1} = \phi(x_0, ...x_i)$). Sie weisen einer Variablen den aktuell gültigen Wert zu. Der Kontrollfluss legt fesst, welches Argument gültig ist. Kann ein Grundblock zum Beispiel von drei anderen Grundblöcken aus erreicht werden, so wird jeder seiner ϕ-Knoten drei Parameter haben. Es wird dann jeweils das erste, zweite oder dritte Argument - je nachdem, ob der Kontrollfluss vom ersten, zweiten oder dritten Grundblock kam - ausgewählt.

Abbildung 4.1 zeigt eine While Schleife, die von 0 bis 10 durchlaufen wird (11 ist die Abbruchbedingung). Grundblöcke sind als Kästchen dargestellt. Innerhalb des Grundblocks wird auf eine Graph-Darstellung verzichtet[1]. Ein ϕ-Knoten für die Variable i ist nur in dem Grundblock mit der Schleifenbedingung nötig. Hier kann der Kontrollfluss vom Start Grundblock - wo i mit 0 belegt wird - kommen. Er kann aber auch vom

[1] Vergleiche Beispiel 1.1 für Graph Darstellung eines Grundblocks

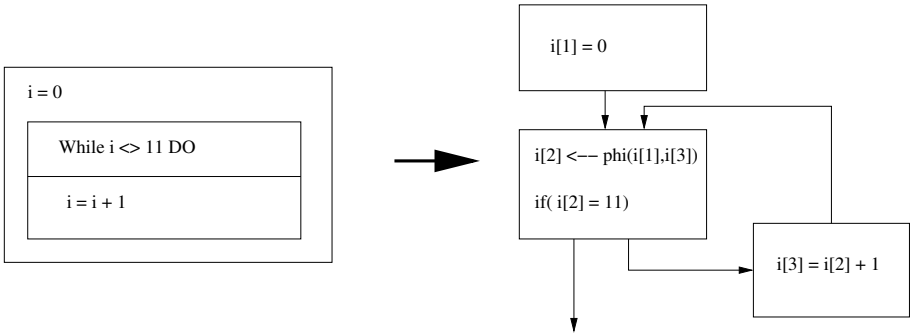


Abbildung 4.1.: SSA-Graph mit ϕ-Knoten

Schleifenrumpf kommen, wo i jedesmal um eins nach oben gezählt wird. Abhängig vom Kontrollfluss bekommt i dann den jeweiligen Wert zugewiesen. Im Schleifenrumpf ist der *phi*-Knoten nicht nötig, weil i notwendigerweise den Wert aus der Schleifenbedingung hat.

Eine SSA Darstellung kann auch Speicherzugriffe darstellen. Die Schreibzugriffe hängen dann Datenflussmäßig von den vorangegangenen Schreibzugriffen ab. Lesezugriffe hängen von der letzten Schreiboperation ab. Zur Übersetzungszeit kann man oft noch nichts über die Adressen sagen. Wir können den Speicher als Variable betrachten, die bei jedem Schreibzugriff dupliziert wird. Wir ordnen jeder Speicherversion eine eigene Wertnummer zu. Prinzipiell können Load Befehle aus allen Versionen Laden. Konzeptuell muss bei der Speicher SSA wenig geändert werden, da die Abhängigkeiten zwischen LOAD und STORE Befehlen auch Datenflussabhängigkeiten sind. [21] beschäftigt sich mit Speicher SSA.

5. Formalisierung einer SSA-Semantik

In diesem Kapitel stellen wir unsere Formalisierung der SSA Semantik vor. Wir definieren eine SSA Darstellung und eine Interpretationsfunktion darauf. Am Anfang stellen wir dar, was überhaupt in Isabelle/HOL umgesetzt werden muss und was für prinzipielle Möglichkeiten uns zur Verfügung stehen. Dann stellen wir eine elegante aber dennoch mächtige Formalisierung vor, mit der wir in Kapitel 6 den Beweis führen, dass jede topologische Sortierung eines Grundblocks eine gültige Codeauswertungsreihenfolge ist. Am Ende zeigen wir noch eine erweiterte Formalisierung, die auch Speicher SSA beinhaltet.

5.1. Vorüberlegungen

Wir betrachten in diesem Abschnitt eine SSA Sprache ohne Speicherbefehle. Wie in Kapitel 4 beschrieben, ist ein Grundblock einer SSA Sprache als gerichteter azyklischer Graph darstellbar. Grundblöcke sind gerade dadurch charakterisiert, dass entweder alle Befehle oder keiner ausgeführt wird. Das Ausrechnen - bzw. ausführen - eines solchen Graphen können wir als atomar betrachten. Wir können dann sagen, dass ein Zustand in der Interpretation des SSA Graphen durch den aktuellen Grundblock und durch einen Speicher von Werten, die den jeweiligen Wertnummern zugeordnet sind, bestimmt ist. Wir bezeichnen im Folgenden diesen Wert Speicher auch als Wertetabelle[1], obwohl unterschiedliche Implementierungen denkbar sind. Es darf zu einem Zeitpunkt nur einen gültigen Wert je Wertnummer geben.

Um einen Interpreter für eine SSA Sprache zu schreiben, bedarf es also nur einer Funktion, die zu einem gegebenen Grundblock und Wertetabelle den nachfolgenden Grundblock und Wertetabelle liefert. Letztendlich hängt die Auswahl des nachfolgenden Grundblocks auch von dem Wert, der einer Wertnummer zugeordnet ist, ab. Also brauchen wir nur eine Funktion, die einen Grundblock ausrechnet und die Wertetabelle aktualisiert.

Wir teilen diese Funktion in drei Phasen auf.

- In der ersten Phase bekommen die Phi-Knoten ihre Werte zugewiesen.
- In der zweiten Phase wird der Grundblock ausgeführt.

[1] Das Wort Speicher halte ich an dieser Stelle für wenig geeignet, da Verwechselungen mit der Speicher SSA entstehen können. Wertetabelle = Speicher für Werte, Adressierung statisch durch Wertnummern, d.h. eine Art beliebig großer Registersatz. Speicher SSA bietet zusätzlich einen Formalismus zum Zugriff auf den Arbeitsspeicher: LOAD, STORE Befehle, Adressierung dynamisch durch Speicheradresse

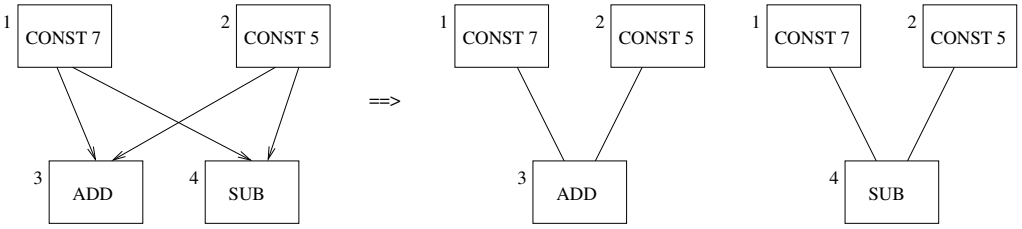


Abbildung 5.1.: Ein SSA-Graph

- In der dritten Phase werden die ausgerechneten Werte eingesammelt und die Wertetabelle aktualisiert.

Wir sehen, dass in der zweiten Phase die Werte der Phi-Knoten wie Konstanten behandelt werden. Sie haben in der ersten Phase Werte zugewiesen bekommen, die sich in den nachfolgenden Phasen nicht mehr verändern. Der nachfolge Grundblock wird durch den Wert einer Wertnummer bestimmt. Also in der dritten Phase.

5.2. Geeignete Datenstrukturen

In Isabelle/HOL stehen uns nur Tupel, Bäume und Mengen als zusammengesetzte Datentypen zur Verfügung. Ein Induktionsprinzip was wir für primitiv-rekursive Funktionen benutzen können ist nur auf Bäumen definiert. Ein Programm können wir als Liste (Spezialfall von einem Baum) oder Menge von Grundblöcken darstellen.

Die Wertetabelle können wir als Funktion betrachten, die zu einer Wertnummer einen Wert liefert. Wir können sie aber auch als Liste von Werten, deren Elemente durch ihre Wertnummern indiziert sind, betrachten. Alternativ können wir die Wertetabelle auch als Menge von (Wertnummer,Wert)-Paaren darstellen. Wir können auch ein einziges (sehr langes) Tupel nehmen, das eine Liste von Werten indiziert durch Wertnummern - nur mit konstanter Länge - ist.

Für das Ausrechnen bzw. das Ausführen eines Grundblocks halten wir ein Induktionsprinzip über den Datenfluss im Grundblock für sehr vorteilhaft. Also kommen in Isabelle/HOL nur Bäume in betracht. Ein Grundblock hat Operationen zwischen denen Datenflussabhängigkeiten bestehen. Die Datenflussabhängigkeiten bilden einen gerichteten azyklischen Graphen. Jeder gerichtete azyklische Graph kann als Menge von Bäumen (Wald) aufgefächert werden. Allerdings müssen dabei manchmal Teilausdrücke dupliziert werden. Abbildung 5.1 zeigt den Grundblock aus der Einleitung als Wald dargestellt. Jeden Wald können wir durch Einfügen eines gemeinsamen Wurzelelements als Baum darstellen.

Da das Auffächern des Datenflusses innerhalb eines Grundblocks das duplizieren von Teilausdrücke beinhaltet benötigen wir zusätzliche Konsistenzbedingungen. Gleiche IDs müssen äquivalente Knoten implizieren.

5.3. Formalisierung des Datenflusses

In diesem Abschnitt beschreiben wir die Formalisierung des Datenflusses innerhalb eines Grundblocks. Wir beschränken uns also hier auf die Formalisierung eines SSA Baumes. Wir haben diesen Teil der Formalisierung zum Führen des Beweises, dass jede topologische Sortierung eines Grundblocks eine gültige Codeauswertungsreihenfolge ist, benutzt. Zuerst beschreiben wir die konkreten Datentypen, dann die Interpretationsfunktionen.

5.3.1. Datentypen

Wie oben diskutiert brauchen wir SSA Bäume und Listen von Code Elementen.
Die SSA Bäume haben wir folgendermaßen formalisiert:

datatype
$SSATree\ =$
$LEAF\ value\ identifier\ |$
$NODE\ operator\ SSATree\ SSATree\ value\ identifier$

Mit folgenden Vereinbarungen:

types
$identifier = nat$
$value = int$
$operator = \text{“}value \Rightarrow value \Rightarrow value''$

Es gibt also Blätter, die Konstanten oder ϕ-Knoten mit zugewiesenem Wert repräsentieren. ϕ-Knoten verhalten sich ja innerhalb eines Grundblocks wie Konstanten. Dann gibt es innere Knoten, die zweistellige Funktionen repräsentieren. Zu beachten ist, dass auch in einem inneren Knoten ein Wert abgespeichert werden kann. Ein Knoten beinhaltet alle seine Vorgänger und keineswegs nur Referenzen auf seine Vorgänger. Somit ist der hier behandelte Baum ein Term[2] .

5.3.2. Die Interpretationsfunktion

Wir brauchen eine Interpretationsfunktionen, die einen SSA Baum ausrechnet. Diese Funktion, die einen SSA Baum ausrechnet haben wir induktiv - als primitiv rekursive Funktion - über den Baumaufbau definiert:

consts
$eval\_tree :: \text{“}SSATree \Rightarrow SSATree''$
primrec
$''eval\_tree(LEAF\ val\ ident) = (LEAF\ val\ ident)''$

[2]Zur Äquivalenz von Termbäumen und Termgraphen [2](S. 66)

$''eval\_tree(NODE\ operat\ tree1\ tree2\ val\ ident) =$
$(NODE$
$operat\ (eval\_tree\ tree1)\ (eval\_tree\ tree2)$
$(operat\ (get\_ssatree\_val\ (eval\_tree\ tree1))\ (get\_ssatree\_val(eval\_tree\ tree2)))$
$ident)''$

Ausgerechnete Werte werden in den Knoten gespeichert.

5.4. Formalisierung des Kontrollflusses

In diesem Abschnitt stellen wir die Formalisierung einer SSA basierten Zwischensprache vor, in der auch Kontrollfluss und Speicher SSA berücksichtigt worden sind. Sie ist eine Erweiterung der Formalisierung aus dem vorangegangenen Abschnitt.

5.4.1. Kontrollfluss Aspekte

Ein Programm besteht in der Regel aus mehreren Grundblöcken. Wir formalisieren das Programm daher als Liste von Grundblöcken. Ein Grundblock ist folgendermaßen definiert:

datatype
$BASICBLOCK\ =\ NEW\ nat\ nat\ \text{“}id \times nat''\ \text{“}id \times nat''\ \text{“}SSATREE\ list''$

Weiterhin benötigen wir eine Wertetabelle. Jedem Knoten in einem SSA Baum ist ja eine Wertnummer zugeordnet. In dieser Wertetabelle werden Werte unter ihren Wertnummern gespeichert. Auf die Tabelle kann auch in anderen Grundblöcken zugegriffen werden, so dass ein globaler Datenfluss möglich ist. Man beachte: Eine Wertetabelle ist in der Formalisierung der SSA Grundblöcke nicht nötig. Wir brauchen sie nur für den grundblockübergreifenden Datenfluss.

In einer Formalisierung, die den Kontrollfluss berücksichtigt müssen wir auch ϕ-Knoten berücksichtigen. Wir machen drei Durchläufe durch einen Baum.

- Wir brauchen, bevor wir mit dem Ausrechnen eines Grundblocks beginnen, einen Durchlauf, der den ϕ-Knoten ihre Werte zuweist. Ein ϕ-Knoten:

  $$PHI\ \text{“}nat\ list''\ value\ id$$

  hat eine Liste von Wertnummern als Argument. Je nachdem, von welchem Vorgänger-Grundblock der Kontrollfluss an den aktuellen Grundblock kommt, wird das entsprechende Argument ausgewählt. Der Wert, der in einem Wertespeicher unter der ausgewählten Wertnummer zu finden ist, wird als Wert dem ϕ-Knoten zugewiesen. Die Reihenfolgenummer, unter der ein Grundblock durch einen Vorgänger erreicht wird, bezeichnen wir hier auch als Rang.

- Sind alle ϕ-Knoten eines Grundblocks ausgewertet, kann der Grundblock ausgerechnet werden. Die ϕ-Knoten verhalten sich dann wie Konstanten.

- Am Ende werden die Ausgerechneten Ergebnisse in die Wertetabelle eingetragen.

Statt einem einzigen SSA Baum als Darstellung für einen Grundblock haben wir hier eine Liste von SSA Bäumen. Ob man eine solche Liste von SSA Bäumen wählt - oder den aufgefächerten Datenfluss DAG durch zusätzliche Top Knoten zu einem einzigen Baum zusammenfasst - ist Geschmackssache. Bei der Listendarstellung müssen wir in einem Durchlauf in allen Bäumen den ϕ-Knoten Werte zuweisen, dann alle Bäume ausrechnen und dann alle Werte einsammeln. Die Listendarstellung vermeidet jedoch "funktionslose" Topknoten.

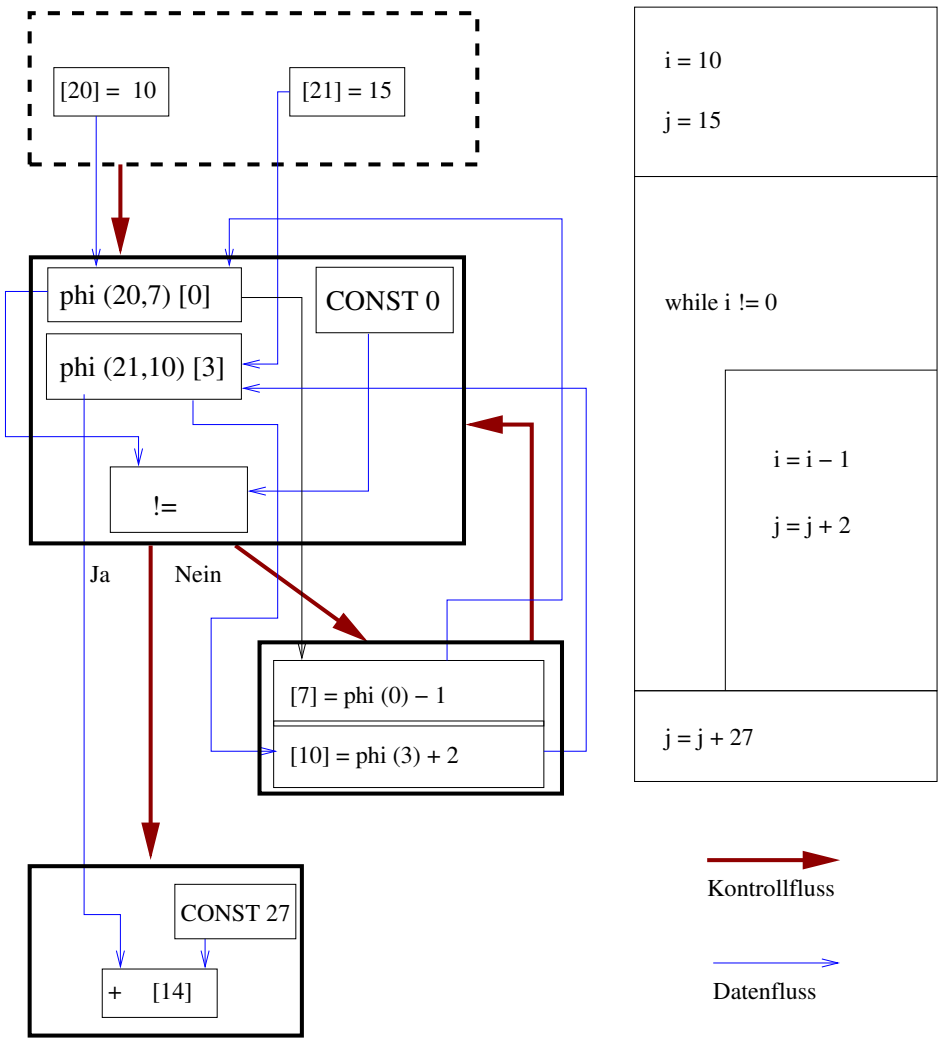
In unserer Formalisierung kann ein Grundblock zwei Nachfolger haben. Diese werden jeweils durch ein $''id \times nat''$ Paar in der Grundblock Definition charakterisiert. id ist die Nummer des Nachfolger-Grundblocks in der Grundblock Liste. nat gibt den Rang an unter dem dieser erreicht wird - also die Auswahl des ϕ-Knoten Arguments. Schließlich gibt es in der Grundblock Definition noch zwei einzelne nat. Das erste gibt eine Wertnummer an. Ist unter dieser Wertnummer am Ende der Grundblockauswertung eine 0 zu finden, so wird das erste $''id \times nat''$ Paar als Nachfolger ausgewählt. Sonst das Zweite. Wir haben einige arithmetische Funktionen und Vergleichsbefehle formalisiert, so dass unsere Formalisierung alle wesentlichen Eigenschaften einer SSA basierten Zwischensprache hat. Das zweite nat gibt den Knoten an, an dem der Gültige Speicherzustand nach Abarbeitung des Grundblocks zu finden ist.

In der Abarbeitung eines Programms können wir Zustände unterscheiden. Ein Zustand ist durch die Wertetabelle, den aktuellen Grundblock und den Rang, unter dem der aktuelle Grundblock erreicht wurde charakterisiert. Wenn wir Speicher SSA betrachten, müssen wir auch noch den Speicherinhalt mit in den Zustand aufnehmen. Die Funktion, die einen Grundblock abarbeitet, kann als atomar angesehen werden. Sie liefert uns zu einem Zustand den Nachfolgezustand. Da die Funktionen in Isabelle immer terminieren müssen haben wir Interpreterfunktionen, die eine beliebige, aber dennoch endliche Anzahl von Grundblöcken nacheinander ausführen können. Letztendlich ist die Semantik aber auch für nicht terminierende Programme durch die einfache Grundblockabarbeitungsfunktion bestimmt.

5.4.2. Eine While-Schleife

Dieses Beispiel (Abbildungen 5.2 und 5.3) zeigt eine einfache While-Schleife, die eine Variable mit 10, die andere mit 15 initialisiert. Von der ersten Variablen wird in jedem Durchlauf eins abgezogen. Zur zweiten wird zwei hinzuaddiert. Abbruchbedingung für die Schleife ist, dass die erste Variable den Wert 0 hat. Im letzten Grundblock wird dann noch 27 zur zweiten Variablen addiert.

Als Knotentypen werden in diesem Beispiel nur ϕ- Knoten, Konstanten und NODE-Knoten für arithmetische Operationen wie ADD($\lambda\ x\ y.x + y$), SUB($\lambda\ x\ y.x - y$) und EQL($\lambda\ x\ y.(if\ (x = y) then\ 1\ else\ 0)$) verwendet. Startzustand ist:



[14] = [3] + Const 2727

Abbildung 5.2.: SSA Darstellung und Programmquelltext

```
  (NEW 2 999
  (2,0) (1,0)
  [
  (NODE EQL
    (PHI [20,7] 0 0)  (*i*)
    (CONST 0 1)
  0 2),
  (PHI [21,10] 0 3) (*j*)
  ] ) ,
(*While Body*)
  (NEW 4 999
  (0,1) (0,0) (* es kann nur erster Fall auftreten *)
  [
  (NODE SUB
    (PHI [0] 0 5) (*i*)
    (CONST 1 6)
  0 7),
  (NODE ADD
     (PHI [3] 0 8)
     (CONST 2 9)
  0 10)
  ]),
(*Last BB*)
  (NEW 11 999
  (2,0) (2,0)
  [
  (NODE ADD
    (PHI [3] 0 12) (*j*)
    (CONST 0 13)
  0 14)
  ])]
```

Abbildung 5.3.: Programm - Darstellung in Isabelle/HOL

NEWSTATE (0,0,0) (λ x .([0,0,0, 0,1,0,1,0,0, 2,0,1,0,27,0, 0,0,0,0,0,10,15] ! x)) (λ x. 0)
Wertetabelle und Speicher sind Funktionen (Wertnummer ⇒ Wert) bzw. (Speicherstelle ⇒ Speicherinhalt)

Der erste Grundblock (Grundblock 0) ist zugleich die Schleifenbedingung. Ist unter Wertnummer 2 am Ende eine 0 zu finden, so wird der Block 2 mit Rang 0 als Nachfolger ausgewählt. Das ist zugleich der Schleifenabbruch. Sonst ist Grundblock 1 an der Reihe. Was unter Wertnummer 2 zu finden ist, bestimmt ein ϕ-Knoten. Wird die Schleifenbedingung zum ersten Mal (Rang 0) aufgerufen, so wird Wertnummer 20 ausgewählt. Hier ist eine 10 zu finden. Sonst wird dieser Grundblock vom Schleifenrumpf aus erreicht. Dann wird Wertnummer 7 ausgewählt. Das Ergebnis wir unter Wertnummer 0 gespeichert. Im Schleifenrumpf (Grundblock 1) wird Wert 0 genommen und um eins subtrahiert. Das Ergebnis ist unter Wertnummer 7 zu finden. Auf diese Weise wird in jedem Schleifendurchlauf das Ergebnis einer Variablen (i) um eins vermindert. Ähnliches geschieht mit Variable j, die immer um zwei erhöht wird. Im letzten Grundblock wird dann noch die Konstante 27 zu j addiert. Abbildung 5.2 zeigt das Beispiel in zwei Darstellungen. Wertnummern sind in eckigen Klammern geschrieben. Der Datenfluss innerhalb von Grundblöcken ist übersichtshalber nur teilweise als Graph dargestellt. Abbildung 5.3 zeigt den Isabelle Text.

5.4.3. Speicher SSA

Speicher SSA stellt zusätzlich zu den sonstigen SSA Konstrukten Möglichkeiten bereit, auf den Speicher zuzugreifen. Der Speicher ist in unserer Formalisierung im Prinzip genauso organisiert wie die Wertetabellen. Im Gegensatz zu dem rein Wertnummern basierten Ansatz, stehen die Speicheradressen hier jedoch nicht statisch fest. Wir können daher nicht immer mit Sicherheit sagen, ob und wo ein geschriebener Wert gelesen wird. Daher gibt es eine Ordnung auf den Schreib- und Leseoperationen. Während die Leseoperationen untereinander beliebig vertauscht werden können, hängen sie von der letzten Speicheroperation ab. Speicheroperationen hängen von allen vorangegangenen Speicheroperationen ab. Prinzipiell behandeln wir Speicheroperationen wie andere Operationen. Wir duplizieren sie, wenn wir den DAG als Baum auffächern und duplizieren damit auch den Speicher.
Wir haben die drei Knoten:

- den $LOAD\ SSATree\ SSATree\ value\ identifier$ Knoten, der den Wert an der Speicherstelle, die durch den einen SSATREE vorgegeben ist, aus dem Speicher, der durch den anderen SSATree (Knoten muß STORE oder MEMORY sein) vorgegeben ist, als “value” speichert;
- den $STORE\ SSATree\ SSATree\ SSATree\ (nat \Rightarrow int)\ identifier$ Knoten, der den er den Wert des einen SSA Baums an der Stelle, die durch den anderen SSATree vorgegeben ist in einem Baum, der durch den letzten SSATree vorgegeben ist (Knoten muß STORE oder MEMORY sein), speichert.
- Der $MEMORY\ (nat \Rightarrow int)\ identifier$ steht für den Speicherzustand zu Beginn des Grundblockausrechnens.

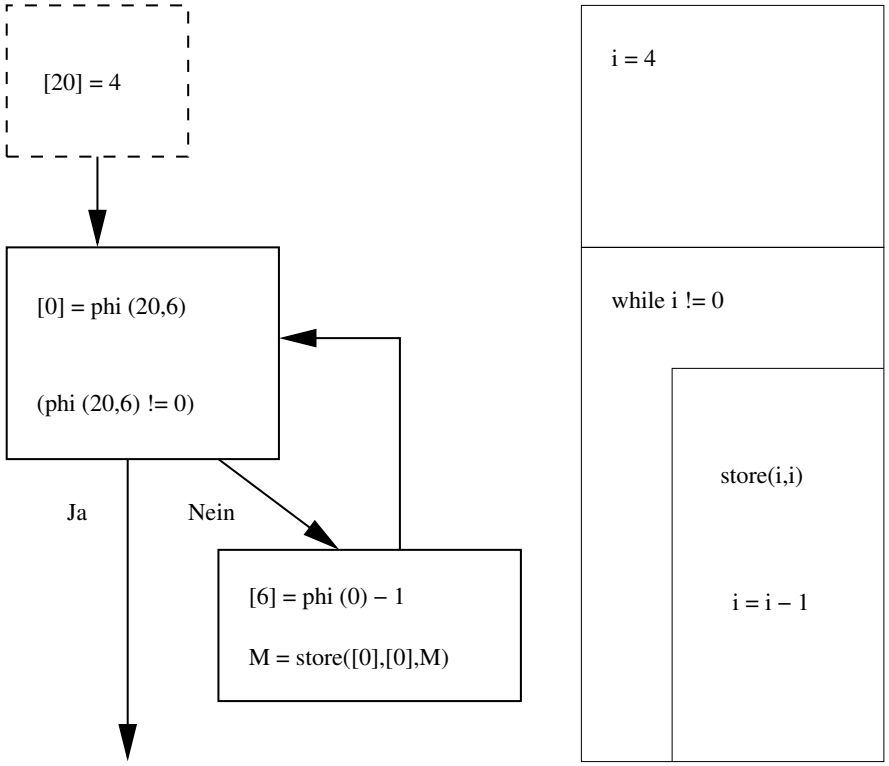


Abbildung 5.4.: Speicher - SSA Beispiel

Wir geben in der Grundblockdefinition einen Identifier für einen Speicherknoten mit an. Dieser enthält den gültigen Speicher nach dem Ausführen des Grundblocks. Dieser Speicher ist der Speicher, den der nächste Grundblock als initialen Speicher mitbekommt.

5.4.4. Eine While-Schleife mit Speicherzugriff

Folgendes Beispiel (Abbildungen 5.4 und 5.5) stellt eine While Schleife dar. Es gibt zwei Grundblöcke: While-Bedingung und While-Schleifen-Rumpf. In jedem Durchlauf findet ein Speicherzugriff statt. Es wird jeweils an die i-te Speicherzelle die Zahl i geschrieben (STORE None (PHI [0] None 4) (PHI [0] None 4) 7)).

Initialisiert wird das ganze durch folgenden Startzustand:

NEWSTATE (0,0,0) (λ x.[0,0,0,1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,4,0]! x) (λ x. 0)

Das (λ x. 0) ist der initiale Speicher. Nach Ablauf des Programms hat er die Form: (λ x.[0,1,2,3,4]!x).

```
 [
(*First BB: COND Block*)
  (NEW  2 0
  (2,0) (1,0)
  [
  (NODE EQL
    (PHI [20,6] 0 0)  (*i*)
    (CONST 0 1)
  0 2)
  ]) ,
(*While Body*)
  (NEW  3 7
  (0,1) (0,0) (* es kann nur erster Fall auftreten *)
  [
  (NODE SUB
    (PHI [0] 0 4) (*i*)
    (CONST 1 5)
  0 6),
  (STORE
     (PHI [0] 0 4)
     (PHI [0] 0 4)
     (MEMORY (\<lambda> x.0) 0)
  (\<lambda> x.13) 7)
  ]) ,
(*Last BB*)
  (NEW 11 0
  (2,0) (2,0)
  []
```

Abbildung 5.5.: Speicher - Beispiel: Isabelle Text

Unsere Formalisierung hat alle wesentlichen Eigenschaften einer SSA basierten Zwischensprache. Sie ist an manchen Stellen sehr allgemein gehalten, so kann ein NODE Knoten beliebige Funktionen darstellen. Anhand zweier Beispiele haben wir gezeigt, wie sich Programme in unserer Formalisierung darstellen lassen.

6. Korrektheit der Codeerzeugung

In diesem Kapitel präsentieren wir den Beweis für die Aussage: “Jede topologische Sortierung des Datenflusses eines Grundblocks ist eine gültige Codeerzeugungsreihenfolge”. Das zentrale Problem ist die Äquivalenz eines SSA Programmstücks und eines Codestücks nachzuweisen. Wir betrachten dafür die Äquivalenz von Grundblöcken. Wir verlangen als Definition einer korrekten Codeerzeugungsreihenfolge, dass jedes Ergebnis (also (Wertnummer, Wert)-Paar), dass in der SSA Darstellung ausgerechnet wird auch bei der Ausführung des Codes in der Wertetabelle erscheint. Für den Beweis müssen wir eine Maschinencodesprache mit Interpretationsfunktion definieren. Wir müssen ebenfalls definieren, was eine topologische Sortierung.

6.1. Äquivalenz von SSA Sprache und Code

Wir wollen nicht nur die Semantik einer SSA Sprache definieren, sondern insbesondere auch Korrektheit von Codeerzeugung nachweisen. Dazu brauchen wir selbstverständlich eine Semantik eines Maschinencodes. Wir stellen Maschinencode als sequentielle Liste von Befehlen dar. Er wird beim Ausführen d.h. interpretieren Elementweise abgearbeitet. Zwischenergebnisse halten wir auch hier in einer Wertetabelle. Unsere Maschinensprache enthält keine Sprungbefehle, weil wir hier nur die Übersetzung einzelner Grundblöcke betrachten.

Wir beweisen, dass jede topologische Sortierung eines Datenflussgraphen im Grundblock eine gültige Codeerzeugungs- bzw. Codeabarbeitungsreihenfolge ist. Dazu müssen wir zunächst einmal sagen was eine topologische Sortierung ist. Baumelemente (Jeder Wald lässt sich durch einfügen eines Topelements als Baum darstellen) und Listenelemente entsprechen sich nicht direkt gegenseitig. Bäume und Listen haben auch ein unterschiedliches Aufbau-(Induktions-)prinzip. Im einzelnen ist zu formalisieren:

- Zu jedem Knoten im Baum gibt es ein entsprechendes Codeelement in der Liste.
- Jedes Listenelement hat einen korrespondierenden Knoten im Baum.
- Die Liste enthält keine Duplikate.
- Die Liste enthält keine zwei Elemente mit gleicher ID und der Baum enthält keine zwei unterschiedlichen Teilbäume mit gleicher ID.
- Die durch den Baum vorgegebenen Abhängikeiten müssen auch in der Liste eingehalten werden. In der Codeliste darf kein Code Element vor einem anderen Code

Element kommen, dessen korrespondierender Baumknoten ein Elternknoten von dem, dem anderen Code Element entsprechenden, Baumknoten ist.

Es gibt auch andere Mengen von Bedingungen, die den gleichen Begriff einer topologischen Sortierung ausdrücken. Beim Umsetzen der Bedingungen in einen Theorembeweiser müssen weitere Bedingungen eingefügt werden, die in einem Papier und Bleistift Beweis als "selbstverständlich" angenommen werden. Codeelemente nehmen diese Gestalt an:

datatype
CodeElement =
L value identifier |
N operator identifier identifier identifier

Codeelemente bestehen also entweder aus einer Konstante: Semantik Lade Wert "val" als Wertnummer "identifier" oder aus Operationen. Die Semantik einer solchen Operation (N op id1 id2 id3) lautet: Lade id1 id2 aus Wertetabelle und führe Operation aus; speichere das Ergebnis unter Wertnummer id3.
Die Interpretationsfunktion, die eine Codeliste ausrechnet ist etwas komplizierter. Im Baum stehen alle für einen Rechenschritt benötigten Zwischenergebnisse - beim Ausrechnen von den Blättern zur Wurzel - in den Kinderknoten. In der Codeliste gibt es aber immer nur einen Vorgänger und keinen Platz für das Speichern von Zwischenergebnissen. Wir führen deshalb einen Speicher oder Wertetabelle in Form einer Speicherzugriffsfunktion ein. Die Zugriffsfunktion liefert zu jeder Wertnummer einen Wert bzw. undefiniert. Die Zugriffsfunktion muss beim Schreiben eines Wertes in den Speicher bzw. in die Wertetabelle angepasst werden. Die Interpretationsfunktion wird mit einer initialen Speicherzugriffsfunktion aufgerufen und liefert eine Funktion, die die komplette Wertetabelle nach Abarbeitung der Code Liste darstellt. Die Funktion hat dann folgende Gestallt:

consts
eval_codelist :: "*CodeList* ⇒ (*nat* ⇒ *value*) ⇒ (*nat* ⇒ *value*)"
primrec
"*eval_codelist* [] *vf* = *vf*"
"*eval_codelist* (*x*#*xs*) *vf* = (*eval_codelist xs* ("
case x of
(*L val ident*) ⇒ (*if* (*ident* = *a*) *then val else* (*vf a*)) |
(*N operat id*1 *id*2 *ident*) ⇒
(*if* (*ident* = *a*) *then* (*operat* (*vf id*1) (*vf id*2)) *else* (*vf a*))))"

In jedem Abarbeitungsschritt wird die Zugriffsfunktion um eine if-Abfrage ergänzt. Das stellt das Schreiben eines Wertes in den Speicher dar. Beim Aufrufen der Zugriffsfunktion wird verglichen, ob die gewünschte Wertnummer mit dem zuletzt geschriebenen

übereinstimmt. Ist das der Fall wird der geschriebene Wert zurückgegeben. Ist das nicht der Fall wird im restlichen Speicher geguckt. Das bedeutet, dass die alte Zugriffsfunktion aufgerufen wird. Somit haben wir mit dieser Definition einen funktionalen Speicher.

Als initiale Funktion sollten wir eine Funktion wählen, die überall undefiniert ist. Das erreichen wir mit dem Hilbert'schen Epsilon Operator. Er liefert uns ein Element zurück, über das nichts bekannt ist, ausser, dass es ein Prädikat erfüllt. Hier ist das Prädikat gerade die Konstante "False":

$$undef\_f \equiv \lambda x.Eps(\lambda x.False)$$

Die Zugriffsfunktion auf Listen in Isabelle ist übrigens analog definiert. Sie liefert auch $Eps(\lambda x.False)$ Werte für Zugriffe oberhalb der Länge der Liste.

Die Wertetabelle haben wir hier als Funktion definiert, die zu einer Wertnummer einen Wert liefert. Wir haben insgesamt drei Implementierungsmöglichkeiten ausprobiert. Nur der von uns gewählte Ansatz hat alle Vorteile.

- Listen von (Wertnummer,Wert) Paaren brauchen, weil mehrere gleiche Wertnummern in sie geschrieben werden können, zusätzliche Axiome. Das verkompliziert Beweise und macht das Axiomensystem unübersichtlich.
- Listen, bei denen die Position eines Elements die Wertnummer ist, haben dieses Problem nicht. Man muss allerdings eine maximale Länge vorgeben, um Einfügeoperationen in die Liste einfach zu gestalten[1].
- Bei dem Ansatz, den Speicher als Funktion zu repräsentieren haben wir keine dieser Einschränkungen.

6.2. Die topologische Sortierung

In diesem Abschnitt stellen wir unsere Formalisierung des Begriffs der topologischen Sortierung vor. Grundsätzliche Anforderungen haben wir schon in Abschnitt 6.1 beschrieben. Hier betrachten wir die konkrete Formalisierung in Isabelle/HOL.

In einem Lehrbuch [12] findet man in etwas folgende Definition: Eine topologische Sortierung zu einer endlichen Halbordnung $(U, \preceq)$ ist eine lineare Ordnung auf U, so dass aus $a \preceq b$ folgt $a \leq b$. Bei uns sind die Elemente im Graphen und in der Liste jedoch von verschiedenen Datentypen. Daher sind sie in erster Näherung nicht vergleichbar. Insbesondere ist die Baumdarstellung, bei der ein Baum alle Teilbäume enthält, mit den Elementen der Code Liste nicht ohne weiteres verträglich. Wie oben angesprochen benötigen wir auch noch weitere Konsistenzbedingungen, die hier auch noch formalisiert werden sollen.

Wir benötigen dafür ein paar Hilfsfunktionen. ce_ify liefert zu einem Baum das korrespondierende Code Element. Es ist sehr einfach definiert, wobei get_ssatree_id den

[1] In der Praxis ist das natürlich keine echte Beschränkung, weil Speicher immer endlich ist. Allerdings ist ein solche Maximallänge der Speicherliste eine Bedingung, die man auch in Beweisen immer "mitschleppen" muß.

Identifier zu einem Baum liefert:

consts
$ce\_ify :: \text{“}SSATree \Rightarrow CodeElement''$
primrec
$\text{“}ce\_ify\ (LEAF\ val\ ident) = (L\ val\ ident)''$
$''ce\_ify\ (NODE\ operat\ kid1\ kid2\ val\ ident) =$
$(N\ operat\ (get\_ssatree\_id\ kid1)\ (get\_ssatree\_id\ kid2)\ ident)''$

Weiterhin benötigen wir die Funktionen:

- is_in_tree oder $\in$: Sie schaut, ob ein Baum in einem anderen als Teilbaum enthalten ist.
- is_in_cl oder $\in$: Ist ein Code Element in einer Code Liste enthalten?
- get_ce_id: Liefert die ID von einem Code Element

Exemplarisch geben wir hier die Funktion is_in_tree an:
consts
$is\_in\_tree :: \text{“}SSATree \Rightarrow SSATree \Rightarrow bool''\ (infixr\ \text{“}\in''\ 65)$
primrec
$''is\_in\_tree\ T\ (LEAF\ val\ ident) = (T = (LEAF\ val\ ident))''$
$''is\_in\_tree\ T\ (NODE\ operat\ kid1\ kid2\ val\ ident) =$
$((T = (NODE\ operat\ kid1\ kid2\ val\ ident)) \vee (is\_in\_tree\ T\ kid1) \vee (is\_in\_tree\ T\ kid2))''$

Folgende Bedingungen haben wir formalisiert:

tops1: Zu jedem Teilbaum von tree gibt es ein korrespondierendes Listenelement in clist: $tops1\ clist\ tree \equiv (\forall a.((a \in tree) \longrightarrow (\exists b.((b \in clist) \longrightarrow (ce\_ify\ a = b)))))$
tops2: Zu jedem Element von clist gibt es einen korrespondierenden Teilbaum von tree: $tops2\ clist\ tree \equiv (\forall b.((b \in clist) \longrightarrow (\exists a.((a \in tree) \longrightarrow (ce\_ify\ a = b)))))$
tops3: Kommt ein Element in der Reihenfolge der Code Liste vor einem anderen Element (direkt oder transitiv), so haben sie unterschiedliche IDs: $tops3\ clist\ tree \equiv \forall a\ b.\ succ\_clist\ clist\ a\ b \longrightarrow get\_ce\_id\ a \neq get\_ce\_id\ b$

tops3 bedeutet, dass keine zwei Elemente in der Codeliste die gleiche ID haben dürfen. Eine wichtige, unmittelbare Folgerung daraus ist, dass keine zwei gleichen Elemente in der Code Liste vorkommen dürfen.

tops4:
Die Liste besteht aus mindestens einem Element:
$tops4\ clist\ T \equiv \exists x\ l.clist = l@[x] \wedge$
Sind y, t im Baum enthalten und y vor t , dann kommt das zu y korrespondierende Code Element vor dem, das zu t korrespondiert:
$\forall\ y\ t.(is\_in\_tree\ y\ T \wedge is\_in\_tree\ t\ T) \longrightarrow ((is\_in\_tree\ y\ t \wedge y \neq t \longrightarrow succ\_clist(l@[x])\ (ce\_ify\ y)\ (ce\_ify\ t))) \wedge$
Ist e in der Code Liste enthalten und ungleich dem letzten Element, dann kommt e vor dem letzten Element
$\forall e.(is\_in\_cl\ e\ (l@[x]) \wedge e \neq x \longrightarrow (succ\_clist(l@[x])\ e\ x)) \wedge$
a kommt nicht vor b oder b kommt nicht vor a
$\forall a\ b.\neg(succ\_clist\ (l@[x])\ a\ b) \vee \neg(succ\_clist\ (l@[x])\ b\ a)))$

Die letzte Zeile ist die Verallgemeinerung von: entweder kommt a vor b oder b vor a. Mit der oben angegebenen Definition können aber auch Einelementige Listen topologisch sortiert sein.

6.3. Das Haupttheorem: Beweisskizze und Hilfssätze

Wir bauen in diesem Abschnitt auf den vorangegangenen Definitionen auf. Unser Beweis gliedert sich in diverse Hilfslemmata. Wir stellen die interessantesten in diesem Unterabschnitten vor.

6.3.1. Beweisskizze

Unser zentrales Problem ist, dass die SSA Bäume über den Baum Aufbau ausgerechnet werden, die Code Liste aber über den Listenaufbau. Somit haben wir zwei verschiedene Induktionsprinzipien. In unserem Beweis machen wir als erstes eine Induktion über den Aufbau der SSA Bäume. Im Basisfall zeigen wir:

- is_topsort (list,LEAF val ident) $\Longrightarrow$ val wird auch in der Code Liste ausgerechnet und unter Wertnummer ident abgespeichert.

  Das geschieht, indem wir zeigen, aus
  is_topsort (list,LEAF val ident) folgt list = [L val ident]
  Damit läßt sich der Basisfall leicht lösen.

Der Induktionsschritt ist etwas komplizierter:

- Aus
  $\forall list'$. is_topsort ($list'$, kid1) $\Longrightarrow$ alles, was in kid1 ausgerechnet wird, wird auch in der Code Liste "$list'$" ausgerechnet und unter entsprechenden Wertnummern abgespeichert.
  und
  $\forall list''$. is_topsort ($list''$, kid2) $\Longrightarrow$ alles, was in kid2 ausgerechnet wird, wird auch in der Code Liste "$list''$" ausgerechnet und unter entsprechenden Wertnummern

abgespeichert.
folgt
$\forall list$. is_topsort($list$, NODE fun kid1 kid2 val ident) $\Longrightarrow$ alles, was in "NODE fun kid1 kid2 val ident" ausgerechnet wird, wird auch in der Code Liste "$list$" ausgerechnet und unter entsprechenden Wertnummer abgespeichert.

kid1 und kid2 sind die beiden Vorgänger von "NODE fun kid1 kid2 val ident". Wir können das $\forall list$. is_topsort($list$, NODE fun kid1 kid2 val ident) mit in die Voraussetzungen aufnehmen. Man kann jetzt die Allquantoren über die Listen nach vorne ziehen. Aus den beiden Allquantoren aus der Voraussetzung über $list'$ und $list''$ werden Existenzquantoren. Wir können sie mit $list' = proj(list, kid1)$ bzw. $list'' = proj(list, kid2)$ mit Funktionssymbolen belegen. proj ist eine Funktion, die gerade die Elemente aus list auf list' bzw. list" projeziert, so dass list' eine topologische Sortierung von kid1 und list" eine topologische Sortierung von kid2 ist.

Wir können jetzt aus der Induktionsvoraussetzung und den Eigenschaften der Projektionsfunktion folgern, dass alle Werte, die in den Teilbäumen kid1 und kid2 ausgerechnet werden auch in der Code Liste ausgerechnet werden.

Wir machen eine Fallunterscheidung: Für jeden Teilbaum t des ausgerechneten Baums "NODE fun kid1 kid2 val ident" soll ja gelten, dass ein entsprechender Wert unter passender Wertnummer in der Code Liste ausgerechnet wird. Die drei Fälle lauten:

1. t ist Teilbaum von kid1
2. t ist Teilbaum von kid2
3. t ist gerade der Wurzelknoten " NODE fun kid1 kid2 val ident"

Die Fälle 1 und 2 folgen aus Voraussetzungen und den Eigenschaften der Projektionsfunktion. Für Fall 3 müssen wir noch beweisen, dass das letzte Element in der Code Liste gerade das korrespondierende Element zum Wurzelknoten ist. Aus der Tatsache, dass die Werte der Kinderknoten richtig in Code Liste und Baum ausgerechnet werden, folgt, dass auch der Wurzelknoten richtig in Code Liste und Baum ausgerechnet wird.

6.3.2. Kompostitionalität der eval_codelist Funktion

Die eval_codelist Funktion arbeitet eine Code Liste ab und schreibt Werte in einen Speicher. Beim Abarbeiten greift sie auf alte Werte in diesem Speicher zurück. Als Ergebnis liefert diese Funktion den Endzustand des Speichers. Aufgerufen wird sie mit der Code Liste und einem initialen Speicher als Argumente. Die Kompositionseigenschaft der eval_codelist Funktion sagt aus, dass man zuerst die Funktion auf dem ersten Teil der Code Liste ausführen darf, um dann das Ergebnis als initialen Speicher für den zweiten Teil der Code Liste zu benutzen. Der Beweis - für diese Kompositionseigenschaft - ist auf dem Papier trivial, in Isabelle muß er jedoch erst trickreich geführt werden. Der Beweis ist unabhängig von der Darstellung des Speichers. Die Kompositionseigenschaft sieht formalisiert in Isabelle folgendermaßen aus:

lemma ec4 : "*eval_codelist* (*a@b*) *f* = *eval_codelist b* (*eval_codelist a f*)″

a und b sind die beiden Code Teillisten. f ist der initiale Speicherzustand (in unserer aktuellen Formalisierung nehmen wir einen funktionalen Speicher)
Folgende zwei Hilfslemmata brauchen wir:

lemma ec1 : "*eval_codelist b* (*eval_codelist* (*a # list*) *r*) =
eval_codelist b (*eval_codelist list* (*eval_codelist* [*a*] *r*))″

lemma ec2 : "*eval_codelist* ((*a # list*) @ *b*) *r* =
eval_codelist (*list* @ *b*) (*eval_codelist* [*a*] *r*)″

Sie lassen sich trivial beweisen. Aus dem dritten Hilfslemma folgt unsere Behauptung unmittelbar. Hier wird eine Induktion über a geführt. Die Verankerung folgt unmittelbar.

lemma ec3 : "∀ *r*. *eval_codelist* (*a@b*) *r* = *eval_codelist b* (*eval_codelist a r*)"

Im Induktionsschritt:

∀ *r*. *eval_codelist* (*list* @ *b*) *r* = *eval_codelist b* (*eval_codelist list r*) ⟹
∀*r*.*eval_codelist*((*a # list*) @ *b*) *r* = *eval_codelist b* (*eval_codelist* (*a # list*) *r*)

werden die beiden Lemmata ec1 und ec2 jeweils auf der passenden Seite der Gleichung rechts des Folgepfeiles substituiert. Das interessante an diesem Beweis ist, dass er mit völlig unterschiedlichen Speicherdarstellungen funktioniert. Dieser Beweis erlaubt es uns insbesondere dass letzte Element einer Code Liste abzuspalten und die Auswertung der anderen isoliert zu betrachten.

6.3.3. Das letzte Element in einer Code Liste

In diesem Beweis geht es um die Form, die das letzte Element in einer topologisch sortierten Code Liste haben muß. Ist eine Code Liste nämlich bezüglich eines Baumes T topologisch sortiert, so muß das letzte Element gerade ce_ify T sein. Also gerade das zum Wurzelelement des Baumes korrespondierende Codelistenelement.

theorem endlist :
"(*tops4* (*l*@[*x*]) *T* ∧ *tops1* (*l*@[*x*]) ∧ *tops2* (*l*@[*x*]) *T*) ⟹ *x* = *ce_ify T*″

Um diesen Beweis zu führen brauchen wir die Eigenschaften tops1, tops2 und tops4 aus der Definition der topologischen Sortierung. Aus tops2 folgern wir in Lemma th_h6:
(∃ *y*. *x* = *ce_ify y* ∧ *is_in_tree y T*)
Zu x existiert also ein korrespondierender Teilbaum y von T. Wir können den Existenzquantor rausschmeissen und y als Konstante betrachten (Skolemisierung). Aus tops1 folgt mit Lemma tops1for4:

$\forall\, t.\ is\_in\_tree\ t\ T\ \longrightarrow\ is\_in\_cl\ (ce\_ify\ t)\ (l@[x])$

Jeder Teilbaum von T hat ein korrespondierendes Element in der Code Liste. Sogar folgende Regel wird gebraucht und muß mit Induktion über T separat bewiesen werden:

$T\ \in\ (T :: SSATree)$

T ist ein Teilbaum von sich selbst. Durch Einsetzen der Definition von tops4 und Instantiierung des ∀ Quantors über e mit "ce_ify T" ergibt sich, eine Aussage, die offensichtlich geeignet ist einen Widerspruchsbeweis zu führen (wir wollen ja gerade ce_ify T = ce_ify y = x zeigen; für x ist hier schon x = ce_ify y benutzt worden):

$ce\_ify\ T\ \in\ l\ @\ [ce\_ify\ y]\ \wedge\ ce\_ify\ T\ \neq\ ce\_ify\ y\ \longrightarrow$
$succ\_clist\ (l\ @\ [ce\_ify\ y])\ (ce\_ify\ T)\ (ce\_ify\ y)$

Zweimal werden noch Quantoren aus den tops Voraussetzungen instantiiert, so dass wir folgende Eigenschaft haben:

$\neg\ succ\_clist\ (l\ @\ [ce\_ify\ y])\ (ce\_ify\ y)\ (ce\_ify\ T)\ \vee$
$\neg\ succ\_clist\ (l\ @\ [ce\_ify\ y])\ (ce\_ify\ T)\ (ce\_ify\ y)$

Daraus kann Isabelle das endlist Theorem automatisch Beweisen. Wenn man den Beweis per Hand führt, würde man einen Widerspruchsbeweis wählen. Also aus der Negation der Behauptung die Negation der Voraussetzung folgern. Umformungen dieser Art ($''a\ \longrightarrow\ b\ \Longrightarrow\ \neg\ b\ \longrightarrow\ \neg\ a''$) kann Isabelle natürlich automatisch vornehmen bzw. werden im Suchraum durchgeführt.

6.4. Das Haupttheorem: der Beweis

In diesem Abschnitt führen wir den Beweis, dass jede topologische Sortierung des SSA Baums eine gültige Codeerzeugungsreihenfolge ist. Der Beweis ist hier in der für Menschen leichter lesbaren Isar Notation dargestellt.
Unser Haupttheorem sieht formalisiert folgendermaßen aus:

theorem $t1$:
assumes $a1$: "$projchar1''$ *and* $a2$: "$projchar2''$ *and* $a3$: "$projchar3''$*and*
$a4$: "$projchar4''$ *and* $a3a$: "$projchar3a''$ *and* $a4a$: "$projchar4a''$
shows
"$\forall clist.((is\_topsort\ clist\ tree) \longrightarrow$
$(\forall t.(t \in (eval\_tree\ tree)) \longrightarrow$
$(\exists ident\ val.(val = (eval\_codelist\ clist(\lambda a.(Eps(\lambda a.False))))ident)$
$\wedge(val = get\_ssatree\_val\ t) \wedge (ident = get\_ssatree\_id\ t)))))$"

Wir zeigen also, dass für jede Liste die eine topologische Sortierung eines Baums ist folgendes gilt: Jeder Teilbaum des ausgerechneten Baums enthält einen ausgerechneten Wert und eine ID. Nach dem Abarbeiten der Liste findet sich in der Wertetabelle unter der gleichen ID eben dieser Wert.

Wir ziehen für den Beweis weitere Voraussetzungen hinzu. Die projchar1 bis projchar4a sind Eigenschaften einer Projektionsfunktion (proj), die wir für den Beweis brauchen. Wir können diese Funktion angeben:
consts

$proj :: \text{“}CodeList \Rightarrow SSATree \Rightarrow CodeList''$
primrec
$\text{“}proj\ []\ tree = []''$
$\text{“}proj\ (x\#xs)\ tree =$
$(if(\exists y.y \in tree \wedge x = ce\_ify\ y)\ then\ (x\#(proj\ xs\ tree))\ else\ (proj\ xs\ tree))''$

Die Funktion bildet die Elemente einer Liste auf eine Teilliste ab. Übrig bleiben die Elemente, die korrespondierende Elemente im Baum haben. Wir brauchen diese Funktion, um für einen Teilbaum eine topologisch sortierte Liste zu erhalten, wenn wir bereits für den übergeordneten Baum eine topologisch sortierte Liste haben. Leider ist es uns bis jetzt noch nicht gelungen maschinell zu Beweisen, dass diese Funktion die gewünschten Eigenschaften hat. Deshalb definieren wir sie über die Eigenschaften.

Die Eigenschaften lauten:

Ist eine Liste topologische Sortierung eines Baums, so ist die proj der Liste auf Elemente des ersten Teilbaums eine topologische Sortierung des ersten Teilbaums:
$projchar1 \equiv \forall l\ x\ operat\ kid1\ kid2\ val\ ident.$
$is\_topsort\ (l@[x])\ (NODE\ operat\ kid1\ kid2\ val\ ident) \longrightarrow ((is\_topsort\ (proj\ l\ kid1)\ kid1)$

Ist eine Liste topologische Sortierung eines Baums, so ist die proj der Liste auf Elemente des zweiten Teilbaums eine topologische Sortierung des zweiten Teilbaums:
$projchar2 \equiv \forall l\ x\ operat\ kid1\ kid2\ val\ ident.$
$is\_topsort\ (l@[x])\ (NODE\ operat\ kid1\ kid2\ val\ ident) \longrightarrow ((is\_topsort\ (proj\ l\ kid2)\ kid2)$

Wird in einer Projektion einer topologisch sortierten Liste auf den ersten Teilbaum ein Wert ausgerechnet, so wird der gleiche Wert auch in der gesamt Liste ausrechnet:
$projchar3 \equiv \forall SSATree1\ SSATree2\ a\ b\ x\ ident\ operat\ val\ l\ t.$
$(is\_topsort\ (l@[x])\ (NODE\ operat\ SSATree1\ SSATree2\ val\ ident) \wedge (b = get\_ssatree\_id\ t) \wedge (t \in eval\_tree\ SSATree1)) \longrightarrow ((a = (eval\_codelist\ (proj\ l\ SSATree1)(\lambda x.\ SOME\ x.False))\ b) \longrightarrow (a = (eval\_codelist\ (l@[x])\ (\lambda x.SOME\ x.False))\ b))$

Wie oben, aber gesamt Liste ohne letztes Element. (Das kommt nie in der Projektion auf einen Teilbaum vor)
$projchar3a \equiv \forall SSATree1\ SSATree2\ a\ b\ x\ ident\ operat\ val\ l\ t.$
$(is\_topsort\ (l@[x])\ (NODE\ operat\ SSATree1\ SSATree2\ val\ ident) \wedge (b = get\_ssatree\_id\ t) \wedge (t \in eval\_tree\ SSATree1)) \longrightarrow ((a = (eval\_codelist\ (proj\ l\ SSATree1)(\lambda x.\ SOME\ x.False))\ b) \longrightarrow (a = (eval\_codelist\ l\ (\lambda x.SOME\ x.False))\ b))$

Das gleiche für den zweiten Teilbaum
$projchar4 \equiv \forall SSATree1\ SSATree2\ a\ b\ x\ ident\ operat\ val\ l\ t.$
$(is\_topsort\ (l@[x])\ (NODE\ operat\ SSATree1\ SSATree2\ val\ ident) \wedge (b = get\_ssatree\_id\ t) \wedge (t \in eval\_tree\ SSATree2)) \longrightarrow ((a = (eval\_codelist\ (proj\ l\ SSATree2)(\lambda x.\ SOME\ x.False))\ b) \longrightarrow (a = (eval\_codelist\ (l@[x])\ (\lambda x.SOME\ x.False))\ b))$

```
projchar4a ≡ ∀SSATree1 SSATree2 a b x ident operat val l t.
(is_topsort (l@[x]) (NODE operat SSATree1 SSATree2 val ident) ∧
(b = get_ssatree_id t) ∧ (t ∈ eval_tree SSATree2)) ⟶ ((a =
(eval_codelist (proj l SSATree2)(λx. SOME x.False)) b) ⟶ (a =
(eval_codelist l (λx.SOME x.False)) b))
```

Auf dem Papier sind diese Eigenschaften trivial zu beweisen. Das Schwierige an einem maschinellen Beweis ist die "Rückkoppelung": Aus den is_topsort Eigenschaften des Elternbaums, Elternliste werden die is_topsort Eigenschaften von Kindbaum und Kindliste gefolgert. Wir können nicht die eine Seite des Folgepfeils anpassen ohne die andere zu verändern, was das "automatische" Beweisfinden erschwert.

Weiterhin benötigen wir für den Beweis einige Lemmata, deren Beweise teilweise in den vorangehenden Abschnitten erläutert sind. Die weniger interessanten Lemmata finden sich im Anhang B.
Der Erste Schritt im Beweis des Theorems ist eine Induktion über den Baum:

```
proof (induct tree)
    fix x y
    show"∀clist.is_topsort clist (LEAF x y) ⟶
    ∀t.t ∈ eval_tree (LEAF x y) ⟶
    ∃ident val.val = eval_codelist clist(λa.SOME a.False) ident∧
    val = get_ssatree_val t ∧ ident = get_ssatree_id t)"by (simp add : lx_h3)"
next
```

Mit fix werden freie Variablen in beliebige Konstanten umgewandelt. Der Basisfall ist durch Anwendung des Lemmas lx_h3 zu lösen. Mit next geht es dann weiter zum Induktionsschritt:

```
next
    from a1 a2 a3 a4 a3a a4a
    show"⋀ fun SSATree1 SSATree2 int nat.
    [|∀clist.is_topsort clist SSATree1 ⟶
    (∀t.t ∈ eval_tree SSATree1 ⟶
    ∃ident val.val = eval_codelist clist(λa.SOME a.False)ident∧
    eval = get_ssatree_val t ∧ ident = get_ssatree_id t);
    ∀clist.is_topsort clist SSATree2 ⟶
    (∀t.t ∈ eval_tree SSATree2 ⟶
    ∃ident val.val = eval_codelist clist(λa.SOME a.False)ident|]
    ⟹
    ∀clist.is_topsort clist (NODE fun SSATree1 SSATree2 int nat) ⟶
    (∀t.t ∈ eval_tree (NODE fun SSATree1 SSATree2 int nat) ⟶
    ∃ident val.val = eval_codelist clist(λa.SOME a.False)ident
proof(clarify)
```

Hier wird also noch einmal explizit der Induktionsschritt angegeben, die Voraussetzungen a1 bis a4a aufgeführt und dann geht es weiter mit einer Umformung. Alles aus dem

gefolgert wird (as1,as2,as3 und as4), wird nach dieser Umformung als Voraussetzung betrachtet (alles was oben auf der linken Seite eines Folgepfeils steht):

```
fix fun nat int SSATree1 SSATree2 t clist
assume
    as1 : "∀clist.is_topsort clist SSATree1 ⟶
    (∀t.t ∈ eval_tree SSATree1 ⟶
    ∃ident val.val = eval_codelist clist(λa.SOME a.False)ident∧
    val = get_ssatree_valt ∧ ident = get_ssatree_id t)"and
    as2 : "∀clist.is_topsort clist SSATree2 ⟶
    (∀t.t ∈ eval_tree SSATree2 ⟶
    ∃ident val.val = eval_codelist clist(λa.SOME a.False)ident∧
    eval = get_ssatree_valt ∧ ident = get_ssatree_id t)"and
    as3 : "is_topsort clist (NODE fun SSATree1 SSATree2 int nat)"and
    as4 : "t ∈ eval_tree (NODE fun SSATree1 SSATree2 int nat)"
```

clist taucht hier sowohl ∀-gebunden, als auch als beliebige Konstante auf. Für den Beweis brauchen wir als erstes ein paar Eigenschaften über den Zusammenhang von der Konstante clist, x und l. Diese lassen sich aus as3 unter Zuhilfenahme des Lemmas lx_h4_fff folgern:

```
from as3 have h1 : "∃x.clist = l@[x]"by (frule_tac lx_h4_fff)
from h1 obtain l x where h1a : "clist = l@[x]"by auto
```

obtain macht wie fix aus freien Variablen beliebige Konstanten. Als nächstes wird aus as3 und lx_h4 eine Eigenschaft einmal mit freien Variablen und dann mit Konstanten vorbelegt (hence) gefolgert:

```
from as3 have h2a : "is_topsort (lv@[xv]) (NODE fun SSATree1 SSATree2 int nat) ⟹
xv = ce_ify (NODE fun SSATree1 SSATree2 int nat)" by (frule_tac lx_h4)
hence h2 : "is_topsort (l@[x]) (NODE fun SSATree1 SSATree2 int nat) ⟹ x =
ce_ify (NODE fun SSATree1 SSATree2 int nat)"by(frule_tac lx_h4)
```

Im folgenden wird das ∀-Quantifizierte clist aus as2 mit (proj l SSATree1) für den ersten Teilbaum bzw. (proj l SSATree2) für den zweiten Teilbaum instanziiert. Dann wird der Ausdruck vereinfacht:

```
from as1 have "is_topsort (proj l SSATree1) SSATree1 ⟶
(∀t.t ∈ eval_treeSSATree1 ⟶
∃ident val.val = eval_codelist (proj l SSATree1)(λa.SOME a.False) ident ∧ val =
get_ssatree_val t ∧ ident = get_ssatree_id t)"
by simp
hence h3 : "is_topsort (proj l SSATree1) SSATree1 ⟶
(∀t.t ∈ eval_treeSSATree1 ⟶
eval_codelist (proj l SSATree1)(λa.SOME a.False)(get_ssatree_id t) =
get_ssatree_val t)"
by simp
```

```
from as2 have "is_topsort (proj l SSATree2) SSATree2 ⟶
(∀t.t ∈ eval_treeSSATree2 ⟶
∃ident val.val = eval_codelist (proj l SSATree2)(λa.SOME a.False) ident ∧ val =
get_ssatree_val t ∧ ident = get_ssatree_id t)"
by simp
hence h4 : "is_topsort (proj l SSATree2) SSATree2 ⟶
(∀t.t ∈ eval_treeSSATree2 ⟶
eval_codelist (proj l SSATree1)(λa.SOME a.False)(get_ssatree_id t) =
get_ssatree_val t)"
by simp
```

Mit h5a h5 und xnature werden verschiedene Zusammenhänge zwischen x, l und anderen Konstanten gefolgert:

```
from as3 h2 h1a have h5a :
"is_topsort (l@[x]) (NODE fun SSATree1 SSATree2 int nat)" by simp
from as3 h2 h1a have h5 :
"is_topsort (l@[N fun (get_ssatree_id SSATree1) (get_ssatree_id SSATree2) nat])
(NODE fun SSATree1 SSATree2 int nat)" by simp  from h5a h2 have xnature :
"x = N fun (get_ssatree_id SSATree1) (get_ssatree_id SSATree2) nat" by simp
```

Im folgenden werden ein paar Eigenschaften jeweils für den ersten und den zweiten Teilbaum bewiesen. Sie ergeben sich ziemlich direkt aus den Voraussetzungen und Lemmata die wir Bewiesen haben. h6a, h7a, h6b und h7b sagen, dass Werte, die in den Teilbäumen berechnet werden auch beim Abarbeiten der Code Liste - bzw. in der Code Liste ohne letztes Element - ausgerechnet werden. Die Eigenschaften müssen aber hier für den Beweis aufbereitet werden:

```
from h5 a1 have h6 : "is_topsort (proj l SSATree1) SSATree1'' by (frule_tac ax1_h1)
from h6 h3 have h6a : "∀t.t ∈ eval_tree SSATree1 ⟶
(get_ssatree_val t =
(eval_codelist (proj l SSATree1) (λa.SOME a.False)) (get_ssatree_id t))'' by simp
from a3a h5 h6a have h6b : "∀t.t ∈ eval_tree SSATree1 ⟶
get_ssatree_val t =
eval_codelist l (λa.SOME a.False) (get_ssatree_id t)'' by (frule_tac ax3_h1a)
from a3 h5 h6a have h6c : "∀t.t ∈ eval_tree SSATree1 ⟶
get_ssatree_val t =
eval_codelist (l@[N fun (get_ssatree_id SSATree1) (get_ssatree_id SSATree2) nat])
(λa.SOME a.False) (get_ssatree_id t)'' by (frule_tac ax3_h1)
```

```
from h5 a2 have h7 : "is_topsort (proj l SSATree2) SSATree2'' by (frule_tac ax2_h1)
from h7 h4 have h7a : "∀t.t ∈ eval_tree SSATree2 ⟶
(get_ssatree_val t =
(eval_codelist (proj l SSATree2) (λa.SOME a.False)) (get_ssatree_id t))'' by simp
from a4a h5 h7a have h7b : "∀t.t ∈ eval_tree SSATree2 ⟶
get_ssatree_val t =
eval_codelist l (λa.SOME a.False) (get_ssatree_id t)'' by (frule_tac ax3_h2a)
from a4 h5 h7a have h7c : "∀t.t ∈ eval_tree SSATree2 ⟶
get_ssatree_val t =
eval_codelist (l@[N fun (get_ssatree_id SSATree1) (get_ssatree_id SSATree2) nat])
(λa.SOME a.False) (get_ssatree_id t)'' by (frule_tac ax3_h2)
```

Was Bewiesen werden soll muss noch einmal hingeschrieben werden (jetzt ohne Voraussetzungen, die wir oben in den Assumptions stehen haben):

```
show "∀ident val.val = eval_codelist clist(λa.SOME a.False) ident ∧ val =
get_ssatree_val t ∧ ident = get_ssatree_id t''
```

Der Beweis geht mit einer Fallunterscheidung über den Teilbaum t weiter. Der Fall wird mit assume angegeben und wir nennen ihn hier ta1. Für alle Teilbäume in SSATree1 ist der Fall mit den Eigenschaften h6c und h1a schnell geführt. Der Wert[2] vom ausgerechneten t wird auch durch die Code Liste berechnet und erhält die gleiche Wertnummer:

```
proof (simp, cases)''
    assume ta1 : "t ∈ eval_treeSSATree1''
    with h6c h1a xnature
    show "eval_codelist clist(λa.SOME a.False) (get_ssatree_id t) = get_ssatree_val t''
by simp next
```

Das gleiche gilt für den zweiten Teilbaum. Etwas umständlich ist, dass wir erst noch die Negation des ersten Falls (assume ta1n) als Voraussetzung aufnehmen müssen:

[2] Der Wert, der einem Baum zugeordnet ist, ist der Wert der Wurzel.

```
assume ta1n : "¬t ∈ eval_tree SSATree1"
    show "eval_codelist clist(∀a.SOME a.False) (get_ssatree_id t) = get_ssatree_val t"
proof (cases)
    assume ta2 : "t ∈ eval_tree SSATree2"
    with h7c h1a xnature
    show "eval_codelist clist(λa.SOME a.False) (get_ssatree_id t) = get_ssatree_val t"
    by simp
next
```

Letzter Fall: t ist gerade der Hauptbaum. Zuerst schreiben wir die Annahme hin, dann beweisen wir ein paar Hilfseigenschaften:

```
assume ta2n : "¬t ∈ eval_tree SSATree2"
show "eval_codelist clist(λa.SOME a.False) (get_ssatree_id t) = get_ssatree_val t"
proof (cases)
    assume ta3 : "t = NODE fun (eval_tree SSATree1) (eval_tree SSATree2)
    (fun (get_ssatree_val (eval_tree SSATree1))
    (get_ssatree_val (eval_tree SSATree2))) nat"
```

Hier wird gezeigt, dass der Wert, des ausgerechneten ersten Teilbaums, auch in der Code Liste ohne letztes Element ausgerechnet wird. Er hat die selbe Wertnummer (idforever ist ein Lemma, das sagt, dass sich die id eines Knoten im Baum beim Ausrechnen nicht ändert):

```
from h6b have h8 : "eval_tree SSATree1 ∈ eval_tree SSATree1 ⟶
get_ssatree_val (eval_tree SSATree1) =
eval_codelist l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree1)) "
by blast
with h8 th_h2 have h8a : "get_ssatree_val (eval_tree SSATree1) =
eval_codelist l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree1)) "
by simp
with h8a idforever have h8b : "get_ssatree_val (eval_tree SSATree1) =
eval_codelist l (λa. SOME a. False) (get_ssatree_id  SSATree1) "
by simp
```

Das gilt natürlich auch für den zweiten Teilbaum:

```
from h7b have h9 : "eval_tree SSATree2 ∈ eval_tree SSATree2 ⟶
get_ssatree_val (eval_tree SSATree2) =
eval_codelist l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree2)) "
by blast
with h9 th_h2 have h9a : "get_ssatree_val (eval_tree SSATree2) =
eval_codelist l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree2)) "
by simp
with h9a idforever have h9b : "get_ssatree_val (eval_tree SSATree2) =
eval_codelist l (λa. SOME a. False) (get_ssatree_id SSATree2) "
by simp
```

Jetzt wird der letzte Fall bewiesen. Hier geht auch die oben separat bewiesene Kompositionseigenschaft der eval Funktion mit ein(ec4):

```
    from h8b h9b ec4 ta3 h1a xnature
    show "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) =
    get_ssatree_val t" by simp
next
```

Wir müssen allerdings noch zeigen, dass keine weiteren Fälle existieren:

```
assume ta3n : "¬ t = NODE fun (eval_tree SSATree1) (eval_tree SSATree2)
(fun (get_ssatree_val (eval_tree SSATree1)) (get_ssatree_val (eval_tree SSATree2))) nat "
with as4 ta1n ta2n ta3n
show "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) =
get_ssatree_val t" by simp
qed
qed
qed
qed
qed
```

Aus den Negationen der Fälle (ta1n, ta2n,ta3n) ergibt sich ein Widerspruch (t muß in einem Teilbaum oder in der Wurzel enthalten sein!, es gibt keinen weiteren Fall mehr). Daraus können wir alles und auch unsere Behauptung für diesen nicht existierenden Fall folgern. Die ganzen Fallunterscheidungen und Beweisebenen werden mit qed beendet. Damit ist die Behauptung bewiesen.

6.5. Anmerkungen und Alternativen zum gewählten Vorgehen

Eine Standard-Beweis-Engineering Methode gibt es nicht. Die Alternativen, den oben beschriebenen Beweis zu führen sind sehr unterschiedlich. Um einen Beweis in einem Theorembeweiser zu führen ist es notwendig alles genau Verstanden zu haben. Man sollte einen Papier und Bleistift Beweis gemacht haben, ehe man anfängt den Beweis in einem Theorembeweiser zu führen. Viele Beweise lassen sich in Theorembeweisern nur

führen, wenn man eine geschickte Formalisierung des Sachverhaltes wählt. Oft kommt man erst beim Versuch etwas zu Beweisen auf eine geschicktere Formalisierung.

In diesem Beweis benutzen wir die Bedingungen für die topologische Sortierung. Wir benutzen sie in einer nicht-induktiven Form. Alternativ wären induktiv definierte Prädikate Denkbar. Mit induktiv definierten Prädikaten ist die topsort-Bedingung zum Beispiel als primitiv-rekursive Funkion über den SSA Baum definiert. Ein Problem mit induktiv definierten Prädikaten ist, dass es manchmal schwer zu sehen ist, was das Prädikat überhaupt macht. Man muß noch einmal nachweisen, dass das induktiv definierte Prädikat auch die gewünschten Eigenschaften hat.

Unser Ansatz ist an einen “Papier und Bleistift” Beweis angelehnt. Die einzelnen Beweisschritte sind sehr Detailliert. Der prinzipielle Aufbau des Beweises ist für den menschlichen Leser leicht nachzuvollziehen (vergleiche 6.3.1). Die benötigten Voraussetzungen werden als prädikatenlogische Formeln eingebracht. Auch diese Form ist für die meisten menschlichen Leser leichter zu verstehen als vielfach verschachtelte induktiv definierte Korrektheitskriterien.

7. Zusammenfassung und Ausblick

Dieses Kapitel gliedert sich in vier Abschnitte. Im ersten werden die Ergebnisse dieser Arbeit nocheinmal zusammengefasst. Im zweiten bewerten wir die Ergebnisse dieser Arbeit. Dann zeigen wir, wie man die Ergebnisse dieser Arbeit benutzen kann, um einen einfachen Resultat Checker für die Übersetzung von Grundblöcken zu bauen. Schließlich geben wir im vierten Abschnitt einen Ausblick auf zukünftige Arbeiten.

7.1. Ergebnisse dieser Arbeit

In dieser Arbeit haben wir eine SSA basierte Zwischensprache vollständig formalisiert. Wir haben die Formalisierung in zwei Teilbereiche unterteilt. Einen, der den Datenfluss in Grundblöcken beschreibt. Einen zweiten, der den Kontrollfluss und Speicher SSA behandelt.

Wir haben dazu neben den SSA Datenstrukturen auch deren formale Semantik operational mittels einer Interpretationsfunktion formalisiert. Diese ist vollständig funktional spezifiziert und lässt sich ohne große Änderung in ML übersetzen. Weiterhin haben wir eine einfache Maschinensprache formalisiert. Wir haben hier ebenfalls eine Interpreterfunktion funktional ausprogrammiert.

Es ist uns gelungen, ein zentrales Theorem zu beweisen: Jede topologische Sortierung eines Datenflussgraphen ist eine gültige Codeauswertungsreihenfolge bzw Erzeugungsreihenfolge. Den Beweis haben wir maschinell mit Isabelle/HOL geführt. Dabei setzen wir im Beweis die Existenz der Projektionsfunktion voraus. Wir geben die Projektionsfunktion zwar an, es ist uns jedoch noch nicht gelungen im Theorembeweiser Isabelle/HOL zu zeigen, dass sie die gewünschten Eigenschaften hat. Die fehlenden Eigenschaften sind in einem Papier und Bleistift Beweis trivial.

Wir haben wesentliche Erkenntnisse über die Einsatzmöglichkeiten des Isabelle Theorembeweisers in der Verifikation von Übersetzer-Back-Ends gewonnen.

7.2. Bewertung und Nutzen der Ergebnisse

Grundsätzlich sind wir davon überzeugt, dass ein in einem Theorembeweiser geführte Beweis genauer ist als ein “pen and pencil” bzw. “Papier und Bleistift” Beweis. Der in Kapitel 6 in Auszügen vorgestellte Beweis ist auf dem Papier auf vier DIN-A4 Seiten zu führen gewesen. Viele Voraussetzungen und Eigenschaften, die auf dem Papier intuitiv vorausgesetzt werden, müssen in einem Beweissystem extra spezifiziert werden. Randbedingungen, über die man sich auf dem Papier leichter hinwegsetzen kann, müssen

eingehalten werden. An der Beweisarchitektur ändert sich prinzipiell nichts. Die Beweisschritte in einem Theorem Beweiser sind oft sehr klein.

Dass die Eigenschaften der proj-Funktion nicht komplett maschinell bewiesen werden konnten, ist störend. Es ist durchaus im Bereich des Möglichen, dass wir Randbedingungen Übersehen haben, unter denen die postulierten Eigenschaften nicht gelten. Wir sind allerdings davon überzeugt, dass es sich hierbei nur um Kleinigkeiten handeln könnte. Sie wären mit kleinen Änderungen an den Voraussetzungen und wahrscheinlich ohne Änderungen am Beweis zu beheben Auf dem Papier sind die Eigenschaften der proj-Funktion trivial zu beweisen. Für das Führen von Beweisen in Theorem Beweisern gibt es keine standard Techniken. Das sich manche Teile eines größeren Beweises in akzeptabeler Zeit nicht maschinell Beweisen lassen ist die Regel (siehe Kapitel 2). Uns ist es gelungen den Beweis fast vollständig maschinell zu führen. Das zeigt, dass unsere Formalisierung und die Beweisidee geeignet gewählt sind.

Das Ergebnis unseres Beweises, nämlich die Tatsache, dass jede topologische Sortierung eines Grundblocks eine gültige Code Auswertungsreihenfolge ist, lässt Spielraum für Optimierungen. Befehle, zwischen denen keine Datenabhängigkeiten bestehen, können beliebig vertauscht werden. Dadurch können Wartezeiten verringert und Code für Piplineabarbeitung optimiert werden.

7.3. Integration der Ergebnisse in den Checker - Ansatz

In Kapitel 2 und [9] ist der Checkeransatz für Übersetzer erwähnt. Für verschiedene Phasen im Übersetzer rechnet jeweils ein Checker zum Beispiel mit Hilfe eines Lösungsprotokolls das Ergebnis nach (siehe Abbildung 7.1). Ein Vorteil der Checker ist, dass man, um korrekte Übersetzer zu erhalten, nur die Checker verifizieren muss. Kommen die verifizierten Checker jeweils zu dem gleichen Ergebnis wie der Übersetzer und sind alle Übersetzerphasen mit Checkern versehen, können wir sicher sein, dass der Übersetzer das Programm richtig übersetzt hat. Natürlich haben wir mit dem Checker nur ein Hinreichendes Kriterium für korrekten Code. Der Checker kann korrekten Code als Falsch zurückweisen. Er darf allerdings nie falschen Code als Richtig annehmen.

Unser Prädikat is_topsort kann als ein einfacher Checker zur Übersetzung von Grundblöcken benutzt werden. Er erhält die SSA Zwischensprache und das Ergebnis der Codeerzeugung als Eingaben. Erfüllen diese beiden Eingaben das is_topsort Kriterium nicht, so sagen wir, dass kein korrekter Code erzeugt wurde. Wird is_topsort zu wahr ausgewertet, so können wir daher sicher sein, dass im Back-End des Übersetzers kein Fehler aufgetreten ist. In diesem Fall braucht der Checker kein Lösungsprotokoll und ist in dieser Diplomarbeit sogar schon verifiziert worden. Abbildung 7.2 zeigt den Ansatz.

7.4. Ausblick auf zukünftige Arbeiten

Wir haben gezeigt, dass Code für einen Grundblock in der Reihenfolge einer topologischen Sortierung der Datenabhängigkeiten des Grundblocks generiert werden kann. Das

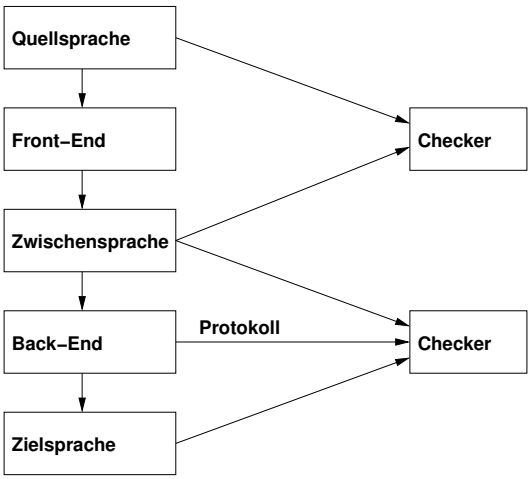


Abbildung 7.1.: Die Checkerarchitektur für Übersetzer

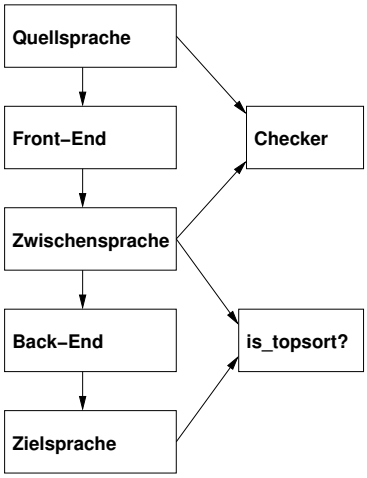


Abbildung 7.2.: Integration von is_topsort in die Checkerarchitektur

ist ein hinreichendes Kriterium für korrekten Code. Es kann allerdings ein wenig restriktiv sein. Optimierungen, wie die Eliminierung von totem Code erlaubt das is_topsort Kriterium zum Beispiel nicht, obwohl es eine semantikerhaltende Optimierung ist. Wir wollen in Zukunft weitere Kriterien finden, unter denen auch eine korrekte Codeerzeugung garantiert wird.

Eine grundsätzliche Frage ist, ob die Darstellung des Grundblocks als Term bzw. als Baum die beste Wahl gewesen ist. Wir haben verschiedene Darstellungen ausprobiert. Grundsätzlich glauben wir, dass ein Induktionsprinzip auf dem Graph notwendig ist. Dieses erlaubt uns Graphen mit einer primitiv-rekursiven Funktion abzuarbeiten. Primitiv-rekursive Funktionen garantieren Terminierung und in Isabelle/HOL ist es relativ leicht Beweise damit zu führen. Die Alternativen erscheinen wesentlich komplizierter. In Isabelle/HOL sind die kürzeren Alternativen in der Regel zu bevorzugen. Dadurch entsteht oft ein höherer Abstraktionsgrad. HOL erlaubt Induktion nur über Bäumen. Es besteht allerdings die Möglichkeit HOL zu erweitern und vielleicht könnte es gelingen einen Datentyp für gerichtete azyklische Graphen (DAG) mit Induktionsprinzip in Isabelle einzubauen. Das haben wir auch schon Testweise versucht, aber es hat sich als schwieriger als ursprünglich gedacht herausgestellt. Um einen praxistauglichen Datentyp DAG zu definieren, müssten wir wahrscheinlich die Higher Order Logic (HOL) selbst erweitern. Weiterhin besteht die Möglichkeit induktiv definierte Mengen zu verwenden. Hier definiert man sich das Induktionsprinzip selbst.

Eine Aufgabe in zukünftigen Arbeiten wird sein, auch Grundblockübergreifenden Datenfluss zu betrachten. Gerade hier verbirgt sich eine Menge Optimierungspotential. Hierfür ist die Baumdarstellung ungeeignet. Der Grundblockübergreifende Datenfluss ist kein DAG mehr. Eine Idee hierfür ist den Datenflussgraphen als Menge von Elementen mit Referenzen auf andere Elemente darzustellen. Man kann Ordnungen auf den Elementen definieren, die einen Grundblock DAG repräsentieren. Dabei wird das Problem zu lösen sein, auf diesen Strukturen algorithmisch zu Arbeiten (d.h. wir müssen Funktionen definieren, die so eine Struktur ausrechnen oder ausführen können).

Eine weitere Frage ist, ob die Definition der topologischen Sortierung gut gewählt ist. Wir sind axiomatisch vorgegangen und haben prädikatenlogische Formeln aufgestellt. Man kann aber auch eine induktive Definition wählen. Hier wird entscheidet ein induktiv definiertes Prädikat ob eine Liste eine topologische Sortierung eines Baums ist. Das ist gerade eine primitiv rekursive Funktion, die über den Aufbau des Baums definiert ist. Ob so eine induktive Definition tatsächlich die gewünschten Eigenschaften hat, ist nicht immer leicht zu sehen. Man kann aber versuchen die gewünschten axiomatischen Eigenschaften aus der induktiven Definition herzuleiten.

Eine Aufgabe für die nahe Zukunft ist es sicherlich die Eigenschaften der proj Funktion zu beweisen.

Auch das betrachten der Äquivalenz von Codestücken ist ein interessantes Thema. Codestück Äquivalenzen sind ein Spezialfall von Graph Äquivalenzen. Ein Kalkül hierfür, von dem ich denke, dass er leicht in Isabelle/HOL umzusetzen ist, könnte zum Beispiel für die Verifikation von Peephole Optimierungen eingesetzt werden.

Danksagung

Allen Mitarbeitern des IPD Goos danke ich für die angenehme Arbeitsatmosphäre, Herrn Prof. Dr. Gerhard Goos für die Unterstützung und sein Interesse an dem Thema, vor allem aber Frau Dr. Sabine Glesner für die intensive Betreuung und die zahlreichen Gespräche und Diskussionen.

Meinen Eltern, meiner Schwester und meinen Freunden danke ich für die moralische Unterstützung, die mir immer zuteil wurde.

Literaturverzeichnis

[1] Alpern, Wegman, and Zadeck. Detecting equality of variables in programs. In *Proc. 15th ACM Symposium on Principles of Programing Languages*. ACM, ACM Press, 1988.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, United Kingdom, 1998.

[3] Jan Olaf Blech. Spezifikation und maschinelle Verifikation von Konstantenfaltung in Übersetzern, Universität Karlsruhe, IPD Goos, Mai 2003.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *POPL 1989*, 1989.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13(4)*, pages 451–490, 1991.

[6] A. Dold, F. W. v. Henke, V. Vialard, and W. Goerigk. A Mechanically Verified Compiling Specification for a Realistic Compiler. Ulmer Informatik-Berichte 02-03, Universität Ulm, Fakultät für Informatik, 2002.

[7] Axel Dold and Vincent Vialard. A mechanically verified compiling specification for a lisp compiler. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Sience*, volume 2245 of *LNCS*, pages 144–155. springer, 2001.

[8] Sabine Glesner. Asms versus natural semantics: A comparison with new insights. In *Abstract State Machines - Advances in Theory and Applications*, volume LNCS Vol. 2589. Springer Verlag, March 2003.

[9] Sabine Glesner. Using program checking to ensure the correctness of compiler implementations. *Journal of Universal Computer Science (J.UCS)*, 9(3):191–222, March 2003.

[10] Sabine Glesner and Jan Olaf Blech. Classifying and Formally Verifying Integer Constant Folding. In Jens Knoop and Wolf Zimmermann, editors, *COCV 2003: Compiler Optimization meets Compiler Verification*, volume 82, Warsaw, Poland,

April 2003. Workshop at the ETAPS Conferences, Electronic Notes in Theoretical Computer Science (ENTCS).

[11] Wolfgang Goerigk and Ulrich Hoffmann. Rigorous compiler implementation correctness: How to prove the real thing correct. In *FM-Trends 1998*, pages 122–136, 1998.

[12] Gerhard Goos. *Vorlesungen über Informatik Band 1: Grundlagen und funktionales Programmieren*. Springer, 3. Auflage, Jan 2000.

[13] Gerhard Goos and Wolf Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rüdiger Olderog, editors, *Correct System Design*, volume 1710, pages 201–230. Springer, Nov 1999.

[14] Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[15] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 1967.

[16] Tobias Nipkow. Verified lexical analysis. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 1–15. Springer, 1998. Invited talk.

[17] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer, 2003.

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

[19] Lawrence C Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, pages 363–397, 1989.

[20] Rosen, Wegman, and Zadeck. Global value numbers and redundant computations. In *Proc. 15th ACM Symposium on Principles of Programing Languages*. ACM, ACM Press, 1988.

[21] Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Notices*, volume 30. ACM, March 1995.

[22] Debora Weber-Wulff. *Contributions to Mechanical Proofs of Correctness for Compiler Front-Ends*. PhD thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts Universität Kiel, 1997.

A. Die Isabelle/HOL Formalisierung und Beweise

In dieser Datei ist die Formalisierung eines SSA Grundblocks, sowie der in Kapitel 6 vorgestellte Beweis zu finden.

```
theory maintheorem = Main:

types
  identifier = nat
  value = int
  operator = "value ⇒ value ⇒ value"

datatype
  SSATree =
     LEAF value identifier |
     NODE operator SSATree SSATree value identifier
consts
  get_ssatree_val :: "SSATree ⇒ value"
  get_ssatree_id :: "SSATree ⇒ identifier"
primrec
  "get_ssatree_val (LEAF  val ident) = val"
  "get_ssatree_val (NODE operat tree1 tree2 val ident) = val"
primrec
  "get_ssatree_id (LEAF  val ident) = ident"
  "get_ssatree_id (NODE operat tree1 tree2 val ident) = ident"

consts
  eval_tree :: "SSATree ⇒ SSATree"
primrec
  "eval_tree (LEAF val ident) = (LEAF val ident)"
  "eval_tree (NODE operat tree1 tree2 val ident) =
  (NODE operat (eval_tree tree1) (eval_tree tree2) (operat (get_ssatree_val (eval_tree tree1))
(get_ssatree_val(eval_tree tree2))) ident)"



datatype
  CodeElement =
```

```
              L  value identifier |
              N operator identifier identifier identifier

consts
  get_ce_id :: "CodeElement ⇒ identifier"
primrec
  "get_ce_id (L val ident) = ident"
  "get_ce_id (N operat kid1 kid2 ident) = ident"

consts
  ce_ify :: "SSATree ⇒ CodeElement"
primrec
  "ce_ify (LEAF val ident) = (L val ident)"
  "ce_ify (NODE operat kid1 kid2 val ident) = (N operat (get_ssatree_id kid1) (get_ssatree_id
kid2) ident)"

types
  CodeList = "CodeElement list"
  ValueList = "(nat × value) list"

consts
   eval_codelist :: "CodeList ⇒ (nat ⇒ value) ⇒ (nat ⇒ value)"
primrec
   "eval_codelist [] vf = vf"
   "eval_codelist (x # xs) vf = (eval_codelist xs (λ a. (
  case x of
  (L val ident) ⇒ (if (ident = a) then val else (vf a) )  |
  (N operat id1 id2 ident) ⇒ (if (ident = a) then (operat (vf id1) (vf id2 )) else (vf a)))))
  "

constdefs undef_f :: "identifier ⇒ int"
"undef_f  ≡ (λ x. Eps (λ x. False))"

lemma "(eval_codelist [L 13 1,L 24 2,N (λ x y. x + y) 1 2 3] undef_f) 3 = 37"
apply auto
done




lemma ec1: "eval_codelist b (eval_codelist (a # list) r) = eval_codelist b (eval_codelist
list (eval_codelist [a] r))"
apply simp
done

lemma ec2: "eval_codelist ((a # list) @ b) r = eval_codelist (list @ b) (eval_codelist [a]
r)"
apply simp
done

lemma ec3: "∀ r. eval_codelist (a@b) r = eval_codelist b (eval_codelist a r) "
apply (induct_tac a)
```

apply *simp*
apply *clarify*
apply *(subst ec1)*
apply *(subst ec2)*
apply *blast*
done

lemma *ec4: "eval_codelist (a@b) f = eval_codelist b (eval_codelist a f)"*
apply *(simp add: ec3)*
done

consts *is_in_cl :: " CodeElement ⇒ CodeList ⇒ bool"* (**infixr** *"∈" 65)*
primrec
"is_in_cl a []= False"
"is_in_cl a (x#xs) = ((a = x) ∨ (is_in_cl a xs))"

consts
succ_clist :: "CodeList ⇒ CodeElement ⇒ CodeElement ⇒ bool"
primrec
"succ_clist [] a b = False"
"succ_clist (x#xs) a b = (if (a = x) then (b ∈ xs) else (succ_clist xs a b))"

consts
is_in_tree :: "SSATree ⇒ SSATree ⇒ bool" (**infixr** *"∈" 65)*
primrec
"is_in_tree T (LEAF val ident) = (T = (LEAF val ident))"
"is_in_tree T (NODE operat kid1 kid2 val ident) = ((T = (NODE operat kid1 kid2 val ident))
∨ (is_in_tree T kid1) ∨ (is_in_tree T kid2))"

constdefs
succ_tree :: "SSATree ⇒ SSATree ⇒ bool" (**infixr** *"<" 65)*
"succ_tree T1 T2 ≡ ((is_in_tree T1 T2) ∧ (T1 ≠ T2))"

consts
proj :: "CodeList ⇒ SSATree ⇒ CodeList"
primrec
"proj [] tree = []"
"proj (x#xs) tree = (if (∃ y. y ∈ tree ∧ x = ce_ify y) then (x#(proj xs tree)) else (proj
xs tree))"

constdefs
tops1 :: "CodeList ⇒ SSATree ⇒ bool"
"tops1 clist tree ≡ (∀ a. ((is_in_tree a tree) ⟶ (∃ b. ((b ∈ clist) ∧ (ce_ify
a = b))))) "
constdefs
tops2 :: "CodeList ⇒ SSATree ⇒ bool"

```
  "tops2 clist tree ≡  (∀ b. ((b ∈  clist) ⟶ (∃  a.((a ∈ tree) ∧ (ce_ify a = b)) )))"

constdefs
  tops3 :: "CodeList ⇒ SSATree ⇒ bool"
  "tops3 clist tree ≡   ∀ a b. succ_clist clist a b ⟶ get_ce_id a ≠ get_ce_id b"
constdefs
  tops4 ::"CodeList ⇒ SSATree ⇒ bool"
  "tops4 clist T ≡ (∃ x l. clist = l@[x] ∧
  (∀ y  t. (is_in_tree y  T ∧ is_in_tree  t  T ) ⟶
(is_in_tree y  t ∧ y ≠ t ⟶  succ_clist (l@[x]) (ce_ify y) (ce_ify t))) ∧
  (∀ e. (is_in_cl e (l@[x]) ∧  e ≠ x) ⟶ (succ_clist (l@[x]) e x)) ∧
  (∀  a b. ¬ (succ_clist (l@[x]) a b) ∨ ¬ (succ_clist (l@[x])  b a))) "
```

```
lemma idforever: "get_ssatree_id (eval_tree tree) = get_ssatree_id tree"
apply (case_tac tree)
apply simp
apply simp
done
```

```
constdefs
  is_topsort :: "CodeList ⇒ SSATree ⇒ bool"
  "is_topsort clist tree ≡ (
  (tops1 clist tree)
  ∧
  (tops2 clist tree)
  ∧
  (tops3 clist tree)
  ∧
  (tops4 clist tree)
  )"
```

```
constdefs projchar1 ::"bool"
"projchar1 ≡ ∀ l x operat kid1 kid2 val ident. is_topsort (l@[x]) (NODE operat kid1 kid2
val ident) ⟶ (is_topsort (proj l kid1) kid1)"
constdefs projchar2 ::"bool"
"projchar2 ≡ ∀ l x operat kid1 kid2 val ident. is_topsort (l@[x]) (NODE operat kid1 kid2
val ident) ⟶ (is_topsort (proj l kid2) kid2)"
constdefs projchar3  ::"bool"
"projchar3 ≡ ∀ SSATree1 SSATree2 a b x ident operat val l t.
(is_topsort (l@[x]) (NODE operat SSATree1 SSATree2 val ident) ∧ (b = get_ssatree_id t) ∧
(t ∈ eval_tree SSATree1 ) ) ⟶ ((a = (eval_codelist (proj l SSATree1) (λx. SOME x. False))
```

```
b) ⟶ (a = (eval_codelist (l@[x]) (λx. SOME x. False)) b))"
constdefs projchar3a  ::"bool"
"projchar3a ≡ ∀ SSATree1 SSATree2 a b t x ident operat val l.
(is_topsort (l@[x]) (NODE operat SSATree1 SSATree2 val ident)  ∧ (b = get_ssatree_id t) ∧
(t ∈ eval_tree SSATree1 ))⟶ ((a = (eval_codelist (proj l SSATree1) (λx. SOME x. False))
b) ⟶ (a = (eval_codelist l (λx. SOME x. False)) b))"
constdefs projchar4  ::"bool"
"projchar4 ≡ ∀ SSATree1 SSATree2 a b x ident operat val l t.
(is_topsort (l@[x]) (NODE operat SSATree1 SSATree2 val ident) ∧ (b = get_ssatree_id t) ∧
(t ∈ eval_tree SSATree2 ) )⟶ ((a = (eval_codelist (proj l SSATree2) (λx. SOME x. False))
b) ⟶ (a = (eval_codelist (l@[x]) (λx. SOME x. False)) b))"
constdefs projchar4a  ::"bool"
"projchar4a ≡ ∀ SSATree1 SSATree2 a b t x ident operat val l.
(is_topsort (l@[x]) (NODE operat SSATree1 SSATree2 val ident)  ∧ (b = get_ssatree_id t) ∧
(t ∈ eval_tree SSATree2 ))⟶ ((a = (eval_codelist (proj l SSATree2) (λx. SOME x. False))
b) ⟶ (a = (eval_codelist l (λx. SOME x. False)) b))"



lemma lastelisin: "∀ x. is_in_cl x  (l@[x])"
apply auto
apply (induct_tac l)
apply auto
done

lemma th_h1: "(∀ t. is_in_tree t T ⟶ (ce_ify t) ∈ (l @ [ce_ify y])) ⟹ (is_in_tree T
T ⟶ (ce_ify T) ∈ (l @ [ce_ify y]))"
apply auto
done


lemma th_h2: "T ∈ (T::SSATree)"
apply (induct_tac T)
apply auto
done

lemma th_h3: "∀e. e ∈ l @ [ce_ify y] ∧ e ≠ ce_ify y ⟶ succ_clist (l @ [ce_ify y]) e
(ce_ify y) ⟹ e ∈ l @ [ce_ify y] ∧ e ≠ ce_ify y ⟶ succ_clist (l @ [ce_ify y]) e (ce_ify
y)"
apply auto
done

lemma th_h4: "∀ a b. ¬ succ_clist (l @ [ce_ify y]) a b ∨ ¬ succ_clist (l @ [ce_ify y])
b a ⟹ ∀ b. ¬ succ_clist (l @ [ce_ify y]) a b ∨ ¬ succ_clist (l @ [ce_ify y]) b a"
apply blast
done

lemma th_h5: "∀ b. ¬ succ_clist (l @ [ce_ify y]) (ce_ify y) b ∨ ¬ succ_clist (l @ [ce_ify
y]) b (ce_ify y) ⟹  ¬ succ_clist (l @ [ce_ify y]) (ce_ify y) b ∨ ¬ succ_clist (l @ [ce_ify
y]) b (ce_ify y)"
apply auto
done

lemma th_h6: "tops2 (l@[x]) T ⟹ (∃ y. x = ce_ify y ∧ is_in_tree y  T )"
```

```
apply (insert lastelisin)
apply (simp add:tops2_def)
apply auto
done

lemma tops1for4: "tops1 (l@[x]) T ⟹ (∀ t. is_in_tree t T ⟶ is_in_cl (ce_ify t) (l@[x]))
"
apply (unfold tops1_def)
apply auto
done

theorem endlist: "(tops4 (l@[x]) T ∧ tops1 (l@[x]) T ∧ tops2 (l@[x]) T) ⟶ x = ce_ify
T"
apply (unfold tops4_def)
apply clarify
apply (drule th_h6)
apply clarify
apply (drule tops1for4)
apply (drule_tac a="ce_ify y" in th_h4)
apply (drule_tac b="ce_ify T" in th_h5)
apply (drule_tac T="T" in th_h1)
apply (insert th_h2)
apply (drule_tac e="ce_ify T" in th_h3)
apply force
done

theorem endlist_a0: "(tops4 (l@[x]) T ∧  tops1 (l@[x]) T ∧ tops2 (l@[x]) T ) ⟹ x = ce_ify
T"
apply (simp add: endlist)
done


theorem endlist_a: "⟦tops4 (l@[x]) T ; tops1 (l@[x]) T; tops2 (l@[x]) T⟧⟹ x = ce_ify T"
apply (rule endlist_a0)
apply auto
done









consts is_in :: " int ⇒ int list ⇒ bool"
primrec
"is_in a []= False"
"is_in a (x#xs) = ((a = x) ∨ (is_in a xs ))"


lemma l2_h1b: "(∀a b. succ_clist l a b ⟶ get_ce_id a ≠ get_ce_id b) ⟹ succ_clist l
a b ⟶ a ≠ b"
apply auto
```

done

lemma *l2_h1: "is_topsort l (LEAF val ident) ⟶ (l = [(L val ident)])"*
apply *(simp add: is_topsort_def tops1_def tops2_def)*
apply *(simp add: tops3_def)*
apply *(case_tac l)*
apply *simp*
apply *(case_tac list)*
apply *simp*
apply *(case_tac "a= (L val ident)")*
apply *(case_tac "aa = (L val ident)")*
apply *clarify*
apply *(drule_tac a="L val ident"* **and** *b="L val ident"* **in** *l2_h1b)*
apply *force*
apply *clarify*
apply *simp*
apply *clarify*
apply *simp*
done

lemma *lx_h1:" ((is_topsort clist (LEAF val ident)) ⟶ (clist = [L val ident])) "*
apply *(simp add: l2_h1)*
done

lemma *lx_h2: "((clist = [L val ident] ⟶ is_topsort clist (LEAF val ident)))"*
apply *(simp add: is_topsort_def tops1_def tops2_def tops3_def tops4_def)*
done

lemma *lx_h3:" ((is_topsort clist (LEAF val ident)) = (clist = [L val ident]))"*
apply *auto*
apply *(simp add: lx_h1)*
apply *(simp add: lx_h2)*
done

lemma *lx_h4_fff: "is_topsort clist (NODE operat t1 t2 val ident) ⟹ ∃ l x. clist = (l@[x])"*
apply *(simp add: is_topsort_def)*
apply *(simp add:tops4_def)*
apply *auto*
done

lemma *lx_h4: "is_topsort (l@[x]) (NODE operat t1 t2 val ident) ⟹ x = ce_ify (NODE operat t1 t2 val ident)"*
apply *(simp add: is_topsort_def)*
apply *clarify*

```
apply (drule_tac  endlist_a)
apply auto
done

lemma lx_h5: "∀ clist.
          is_topsort clist tree ⟶
          (∀t.
              is_in_tree t (eval_tree tree) ⟶
              (∃ ident val.
                  val = (eval_codelist clist (λa. SOME a. False) ident ) ∧ val =
get_ssatree_val t ∧ ident = get_ssatree_id t))
  ⟹
          is_topsort clist tree ⟶
          (∀t.
              is_in_tree t (eval_tree tree) ⟶
              (∃ ident val.
                  val = (eval_codelist clist (λa. SOME a. False) ident ) ∧ val =
get_ssatree_val t ∧ ident = get_ssatree_id t))"
apply auto
done

lemma ax1_h1: "⟦projchar1;is_topsort (l @ [N operat (get_ssatree_id SSATree1) (get_ssatree_id
SSATree2) ident]) (NODE operat SSATree1 SSATree2 val ident)⟧ ⟹ (is_topsort (proj l  SSATree1)
SSATree1)"
apply (simp add:projchar1_def)
apply force
done

lemma ax2_h1: "⟦projchar2;is_topsort (l @ [N operat (get_ssatree_id SSATree1) (get_ssatree_id
SSATree2) ident]) (NODE operat SSATree1 SSATree2 val ident)⟧ ⟹ (is_topsort (proj  l  SSATree2)
SSATree2)"
apply (simp add:projchar2_def)
apply force
done


lemma ax3_h1: "⟦projchar3;is_topsort (l @ [N operat (get_ssatree_id SSATree1) (get_ssatree_id
SSATree2) ident]) (NODE operat SSATree1 SSATree2 val ident);∀ t. t ∈ eval_tree SSATree1 ⟶
(get_ssatree_val t = (eval_codelist (proj  l SSATree1) (λa. SOME a. False)) (get_ssatree_id
t))⟧ ⟹ ∀ t. t ∈ eval_tree SSATree1 ⟶ (get_ssatree_val t = (eval_codelist (l @ [N operat
(get_ssatree_id SSATree1) (get_ssatree_id SSATree2) ident])  (λa.
SOME a. False)) (get_ssatree_id t))"
apply (simp add: projchar3_def)
done

lemma ax3_h1a: "⟦projchar3a;is_topsort (l @ [N operat (get_ssatree_id SSATree1) (get_ssatree_id
SSATree2) ident]) (NODE operat SSATree1 SSATree2 val ident);∀ t. t ∈ eval_tree SSATree1 ⟶
(get_ssatree_val t = (eval_codelist (proj  l SSATree1) (λa. SOME a. False)) (get_ssatree_id
t))⟧ ⟹ ∀ t. t ∈ eval_tree SSATree1 ⟶ (get_ssatree_val t = (eval_codelist l  (λa.
SOME a. False)) (get_ssatree_id t))"
apply (simp add: projchar3a_def)
done

lemma ax3_h2: "⟦projchar4;is_topsort (l @ [N operat (get_ssatree_id SSATree1) (get_ssatree_id
```

```
SSATree2) ident]) (NODE operat SSATree1 SSATree2 val ident);∀ t. t ∈ eval_tree SSATree2 ⟶
(get_ssatree_val t = (eval_codelist (proj  l SSATree2) (λa. SOME a. False)) (get_ssatree_id
t))⟧ ⟹ ∀ t. t ∈ eval_tree SSATree2 ⟶ (get_ssatree_val t = (eval_codelist (l @ [N operat
(get_ssatree_id SSATree1) (get_ssatree_id SSATree2) ident])  (λa. SOME a. False)) (get_ssatree_id
t))"
apply (simp add: projchar4_def)
done

lemma ax3_h2a: "⟦projchar4a;is_topsort (l @ [N operat (get_ssatree_id SSATree1) (get_ssatree_id
SSATree2) ident]) (NODE operat SSATree1 SSATree2 val ident);∀ t. t ∈ eval_tree SSATree2 ⟶
(get_ssatree_val t = (eval_codelist (proj  l SSATree2) (λa. SOME a. False)) (get_ssatree_id
t))⟧ ⟹ ∀ t. t ∈ eval_tree SSATree2 ⟶ (get_ssatree_val t = (eval_codelist l  (λa. SOME
a. False)) (get_ssatree_id t))"
apply (simp add: projchar4a_def)
done

lemma lls_1: "∀t. t ∈ eval_tree tree ⟶ get_ssatree_val t = (eval_codelist l (λa. SOME
a. False)) (get_ssatree_id t) ⟹ t ∈ eval_tree tree ⟶ get_ssatree_val t = (eval_codelist
l (λa. SOME a. False)) (get_ssatree_id t)"
apply auto
done

lemma delete_next: " b ⟹ True" by auto

theorem t1: "(projchar1 ∧ projchar2 ∧ projchar3 ∧ projchar4∧ projchar3a ∧ projchar4a)
⟹ (∀ clist. ((is_topsort clist tree) ⟶ (∀  t.(t ∈ (eval_tree tree)) ⟶  (∃ ident val.
( val = (eval_codelist clist (λ a. (Eps (λ a. False)))) ident) ∧ (val = get_ssatree_val t)∧
(ident = get_ssatree_id t))))) "

apply (induct_tac tree)
apply (simp add: lx_h3)
apply clarify
apply (frule_tac lx_h4_fff)
apply clarify
apply (frule_tac lx_h4)
apply (drule_tac clist = "proj l SSATree1" in lx_h5)
apply (drule_tac clist = "proj l SSATree2" in lx_h5)
apply clarify
apply simp
apply (frule_tac ax1_h1)
apply simp
apply (frule_tac ax2_h1)
apply simp
apply (frule_tac ax3_h1a)
apply simp
apply simp
apply (frule_tac  ax3_h2a)
apply simp
apply simp
```

```
apply (frule_tac ax3_h1)
apply simp
apply simp
apply (frule_tac  ax3_h2)
apply simp
apply simp
apply simp

apply (case_tac "t ∈ eval_tree SSATree1")
apply simp

apply (case_tac "t ∈ eval_tree SSATree2")
apply simp

apply (case_tac "t = NODE fun (eval_tree SSATree1) (eval_tree SSATree2) (fun (get_ssatree_val
(eval_tree SSATree1)) (get_ssatree_val (eval_tree SSATree2))) nat ")
apply (drule_tac t ="eval_tree SSATree1" in lls_1)
apply (drule_tac t ="eval_tree SSATree2" in lls_1)
apply (simp add:th_h2)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply (drule delete_next)
apply simp
apply (simp only: ec4)
apply (simp add:idforever)
apply simp
done



lemma "is_topsort [L 1 2] (LEAF 1 2)"
apply (simp add: is_topsort_def tops1_def  tops2_def tops3_def tops4_def )
done

lemma "¬ is_topsort [L 2 2] (LEAF 1 2)"
apply (simp add: is_topsort_def tops1_def  tops2_def tops3_def tops4_def )
done


lemma "¬ is_topsort [L 2 2] (NODE a b c d e)"
apply (simp add: is_topsort_def tops1_def  tops2_def tops3_def tops4_def )
apply auto
```

done

lemma *"is_topsort [ce_ify (LEAF 1 2), ce_ify (LEAF 2 3) ,ce_ify (NODE a (LEAF 1 2) (LEAF 2 3) b 1)] (NODE a (LEAF 1 2) (LEAF 2 3) b 1)"*
apply *(simp add: is_topsort_def tops1_def)*
apply *auto*
apply *(simp add: tops2_def)*
apply *force*
apply *(simp add: tops3_def)*
apply *(simp add: tops4_def)*
apply *force*
done

theorem *t1v2:*
assumes *a1: "projchar1"* **and** *a2: "projchar2"* **and** *a3: "projchar3"* **and** *a4: "projchar4"* **and** *a3a: "projchar3a"* **and** *a4a: "projchar4a"*
shows *"(∀ clist. ((is_topsort clist tree) ⟶ (∀ t.(t ∈ (eval_tree tree)) ⟶ (∃ ident val. (val = (eval_codelist clist (λ a. (Eps (λ a. False)))) ident) ∧ (val = get_ssatree_val t)∧ (ident = get_ssatree_id t))))) "*
proof *(induct tree)*
fix *x y*
show *"∀clist. is_topsort clist (LEAF x y) ⟶*
(∀t. t ∈ eval_tree (LEAF x y) ⟶
(∃ident val. val = eval_codelist clist (λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t))" **by** *(simp add: lx_h3)*
next
from *a1 a2 a3 a4 a3a a4a*
show *"⋀fun SSATree1 SSATree2 int nat.*
⟦∀clist. is_topsort clist SSATree1 ⟶
(∀t. t ∈ eval_tree SSATree1 ⟶ (∃ident val. val = eval_codelist clist (λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t));
∀clist. is_topsort clist SSATree2 ⟶
(∀t. t ∈ eval_tree SSATree2 ⟶ (∃ident val. val = eval_codelist clist (λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t))⟧
⟹ ∀clist. is_topsort clist (NODE fun SSATree1 SSATree2 int nat) ⟶
(∀t. t ∈ eval_tree (NODE fun SSATree1 SSATree2 int nat) ⟶
(∃ident val. val = eval_codelist clist (λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t))"

proof *(clarify)*
fix *fun nat int SSATree1 SSATree2 t clist*
assume
as1: "∀clist. is_topsort clist SSATree1 ⟶
(∀t. t ∈ eval_tree SSATree1 ⟶ (∃ident val. val = eval_codelist clist (λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t))" **and**
as2: "∀clist. is_topsort clist SSATree2 ⟶
(∀t. t ∈ eval_tree SSATree2 ⟶ (∃ident val. val = eval_codelist clist (λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t))" **and**
as3: "is_topsort clist (NODE fun SSATree1 SSATree2 int nat)"
and
as4: "t ∈ eval_tree
(NODE fun SSATree1 SSATree2 int nat)"

```
    from as3 have h1: "∃l x. clist = l @ [x]" by (frule_tac lx_h4_fff)
    from h1 obtain  l x  where h1a: "clist = l @ [x]"  by auto
    from as3 have h2a: "is_topsort (lv@[xv]) (NODE fun SSATree1 SSATree2 int nat) ⟹ xv
=
ce_ify (NODE fun SSATree1 SSATree2 int nat)" by (frule_tac lx_h4)
    hence h2:  "is_topsort (l@[x]) (NODE fun SSATree1 SSATree2 int nat) ⟹ x = ce_ify
(NODE fun SSATree1 SSATree2 int nat)" by (frule_tac lx_h4)
    from as1 have  "is_topsort (proj l SSATree1) SSATree1 ⟶
      (∀t. t ∈ eval_tree SSATree1 ⟶ (∃ident val. val = eval_codelist (proj l SSATree1)
(λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t))"
      by simp
    hence h3: "is_topsort (proj l SSATree1) SSATree1 ⟶
      (∀t. t ∈ eval_tree SSATree1 ⟶ eval_codelist (proj l SSATree1) (λa. SOME a. False)
(get_ssatree_id t) = get_ssatree_val t)" by simp
    from as2 have  "is_topsort (proj l SSATree2) SSATree2 ⟶
      (∀t. t ∈ eval_tree SSATree2 ⟶ (∃ident val. val = eval_codelist (proj l SSATree2)
(λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident = get_ssatree_id t))"
      by simp
    hence h4: "is_topsort (proj l SSATree2) SSATree2 ⟶
      (∀t. t ∈ eval_tree SSATree2 ⟶ eval_codelist (proj l SSATree2) (λa. SOME a. False)
(get_ssatree_id t) = get_ssatree_val t)" by simp
    from as3 h2 h1a have h5a:  "is_topsort (l @ [x]) (NODE fun SSATree1 SSATree2 int nat)"
by simp
    from as3 h2 h1a have  h5:  "is_topsort (l @ [N fun
(get_ssatree_id SSATree1) (get_ssatree_id SSATree2) nat]) (NODE fun SSATree1 SSATree2 int
nat)" by simp
    from h5a h2 have xnature: "x =  N fun (get_ssatree_id SSATree1) (get_ssatree_id SSATree2)
nat" by simp

    from h5 a1 have h6: "is_topsort (proj l SSATree1) SSATree1" by (frule_tac ax1_h1)
    from h6 h3 have h6a:  "∀ t. t ∈
eval_tree SSATree1 ⟶   (get_ssatree_val t =
(eval_codelist (proj  l SSATree1) (λa. SOME a. False)) (get_ssatree_id t))" by simp
    from a3a h5 h6a have h6b: " ∀t. t ∈ eval_tree SSATree1 ⟶ get_ssatree_val t =
eval_codelist l (λa. SOME a. False) (get_ssatree_id t)"  by  (frule_tac  ax3_h1a)
    from a3 h5 h6a have h6c:    "∀t. t ∈ eval_tree SSATree1 ⟶
      get_ssatree_val t =
eval_codelist (l @ [N fun (get_ssatree_id SSATree1) (get_ssatree_id SSATree2) nat]) (λa. SOME
a. False) (get_ssatree_id t)" by (frule_tac ax3_h1)

    from h5 a2 have h7: "is_topsort (proj l SSATree2) SSATree2" by (frule_tac ax2_h1)
    from h7 h4 have h7a:  "∀ t. t ∈ eval_tree SSATree2 ⟶   (get_ssatree_val t =
(eval_codelist (proj  l SSATree2) (λa. SOME a. False)) (get_ssatree_id t))" by simp
    from a4a h5 h7a have h7b: " ∀t. t ∈ eval_tree SSATree2 ⟶ get_ssatree_val t =
eval_codelist l (λa. SOME a. False) (get_ssatree_id t)"  by  (frule_tac  ax3_h2a)
    from a4 h5 h7a have h7c:    "∀t. t ∈ eval_tree SSATree2 ⟶
      get_ssatree_val t =
eval_codelist (l @ [N fun (get_ssatree_id SSATree1) (get_ssatree_id SSATree2) nat]) (λa. SOME
a. False) (get_ssatree_id t)" by (frule_tac ax3_h2)
    show "∃ident val. val =
eval_codelist clist (λa. SOME a. False) ident ∧ val = get_ssatree_val t ∧ ident =
get_ssatree_id t"
    proof (simp,cases)
      assume ta1: "t ∈ eval_tree SSATree1"
```

```
      with h6c h1a xnature
      show  "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) = get_ssatree_val
t" by simp
    next
      assume ta1n: "¬ t ∈ eval_tree SSATree1"
      show  "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) = get_ssatree_val
t"
      proof (cases)
        assume ta2: "t ∈ eval_tree SSATree2 "
        with h7c h1a xnature
        show  "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) = get_ssatree_val
t" by simp
      next
        assume ta2n: "¬ t ∈ eval_tree SSATree2 "
        show  "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) = get_ssatree_val
t"
        proof (cases)
          assume ta3: "t = NODE fun (eval_tree SSATree1) (eval_tree SSATree2) (fun (get_ssatree_val
(eval_tree SSATree1)) (get_ssatree_val (eval_tree SSATree2))) nat "
          from h6b have h8: "eval_tree SSATree1 ∈ eval_tree SSATree1 ⟶ get_ssatree_val
(eval_tree SSATree1) = eval_codelist l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree1))
" by  blast
          with h8 th_h2 have h8a:  "get_ssatree_val (eval_tree SSATree1) = eval_codelist
l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree1)) "  by simp
          with h8a idforever have h8b:  "get_ssatree_val (eval_tree SSATree1) = eval_codelist
l (λa. SOME a. False) (get_ssatree_id  SSATree1) "  by simp
          from h7b have h9: "eval_tree SSATree2 ∈ eval_tree SSATree2 ⟶ get_ssatree_val
(eval_tree SSATree2) = eval_codelist l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree2))
" by  blast
          with h9 th_h2 have h9a:   "get_ssatree_val (eval_tree SSATree2) = eval_codelist
l (λa. SOME a. False) (get_ssatree_id (eval_tree SSATree2)) "  by simp
          with h9a idforever have h9b:  "get_ssatree_val (eval_tree SSATree2) = eval_codelist
l (λa. SOME a. False) (get_ssatree_id  SSATree2) "  by simp
          from h8b h9b ec4 ta3  h1a xnature
          show  "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) = get_ssatree_val
t"  by simp
        next
          assume ta3n: "¬ t = NODE fun (eval_tree SSATree1) (eval_tree SSATree2) (fun (get_ssatree_val
(eval_tree SSATree1)) (get_ssatree_val (eval_tree SSATree2))) nat "
          with as4 ta1n ta2n ta3n
          show   "eval_codelist clist (λa. SOME a. False) (get_ssatree_id t) = get_ssatree_val
t"  by simp
        qed
      qed
    qed
  qed
qed




end
```

B. Speicher SSA Formalisierung

In diesem Abschnitt werden die zwei Dateien basic und eval gezeigt. In ihnen wird Speicher SSA formalisiert.

theory *basic = Main:*

types *phiargs = "nat list"*

types *value = "int"*

types
  identifier = nat
  operat = "value ⇒ value ⇒ value"

types
  memory = "nat ⇒ value"

datatype
  SSATree = CONST value identifier |
  PHI phiargs value identifier |
  NODE operat SSATree SSATree value identifier |
  LOAD SSATree SSATree value identifier |
  STORE SSATree SSATree SSATree memory identifier |
  MEMORY memory identifier

constdefs
  ADD :: "value ⇒ value ⇒ value"
  "ADD a b ≡ a + b"
constdefs
  SUB :: "value ⇒ value ⇒ value"
  "SUB a b ≡ a - b"
constdefs
  GRT :: "value ⇒ value ⇒ value"
  "GRT a b ≡ (if (b < a) then 1 else 0)"
constdefs

```
  EQL :: "value ⇒ value ⇒ value"
  "EQL a b ≡ (if (a  = b) then 1 else 0)"
```

datatype

```
  BASICBLOCK = NEW identifier identifier "identifier×nat"  "identifier×nat"  "SSATree list"
```

types

```
  valuevector = "identifier ⇒ value"
```

datatype

```
  state = NEWSTATE "identifier × identifier × nat" valuevector memory
```

types

```
  cfg = "BASICBLOCK list"
```

consts

```
  get_id :: "SSATree ⇒ identifier"
```

primrec

```
  "get_id (CONST val ident) = ident"
  "get_id (PHI pa val ident) = ident"
  "get_id (NODE operat tree1 tree2 val ident) = ident"
  "get_id (LOAD  tree memtree val ident) = ident"
  "get_id (STORE  tree1 tree2 memtree val ident) = ident"
  "get_id (MEMORY  val ident) = ident"
```

consts

```
  get_val :: "SSATree ⇒ value"
```

primrec

```
  "get_val (CONST val ident) = val"
  "get_val (PHI pa val ident) = val"
  "get_val (NODE operat tree1 tree2 val ident) = val"
  "get_val (LOAD  tree memtree val ident) = val"
```

consts

```
  get_mem :: "SSATree ⇒ memory"
```

primrec

```
  "get_mem (STORE  tree1 tree2 memtree val ident) = val"
  "get_mem (MEMORY val ident) = val"
```

consts

```
  get_valn :: "BASICBLOCK ⇒ nat"
  get_valm :: "BASICBLOCK ⇒ nat"
  get_true_target :: "BASICBLOCK ⇒ (identifier × identifier)"
```

```
  get_false_target :: "BASICBLOCK ⇒ (identifier × identifier)"
  get_tree_list :: "BASICBLOCK ⇒ (SSATree list)"
  get_tree_list2 :: "BASICBLOCK ⇒ (SSATree list)"
primrec
  "get_valn (NEW valn valm a b c ) = valn"
primrec
  "get_valm (NEW valn valm a b c ) = valm"
primrec
  "get_true_target (NEW valn valm a b c ) = a"
primrec
  "get_false_target (NEW valn valm a b c ) = b"
primrec
  "get_tree_list (NEW valn valm a b c ) = c"
```

end

theory *eval = basic + Main:*

```
consts
  eval_phi :: "SSATree ⇒ valuevector ⇒ nat ⇒ SSATree"
primrec
  "eval_phi (CONST  val ident) svv rang  = (CONST val ident)"
  "eval_phi (PHI valnlist val ident) svv rang  = (PHI valnlist (svv (valnlist ! rang)) ident)"

  "eval_phi (NODE operat tree1 tree2 val ident) svv rang = (NODE operat (eval_phi tree1 svv
rang) (eval_phi tree2 svv rang) val ident) "
  "eval_phi (LOAD tree memtree  val  ident) svv rang = (LOAD  (eval_phi tree svv rang) (eval_phi
memtree svv rang) val ident) "
  "eval_phi (STORE tree1 tree2 memtree val ident) svv rang =  (STORE  (eval_phi tree1 svv
rang) (eval_phi tree2 svv rang) (eval_phi memtree svv rang) val ident) "
  "eval_phi (MEMORY m ident) svv rang= (MEMORY m ident)"
```

```
consts
  eval :: "SSATree ⇒ memory ⇒ SSATree"
primrec
  "eval (CONST val ident) m = (CONST val ident)"
  "eval (PHI pa val ident) m = (PHI pa val ident)"
  "eval (NODE operat tree1 tree2 val ident) m = (NODE operat  (eval tree1 m) (eval tree2 m)
(operat (get_val(eval tree1 m)) (get_val (eval tree2 m))) ident)"
  "eval (LOAD tree memtree val ident) m = (LOAD (eval tree m)  (eval memtree m) ((get_mem
(eval memtree m)) (nat (get_val (eval tree m))))   ident)"
  "eval (STORE  tree1 tree2 memtree val ident) m = (STORE (eval tree1 m) (eval tree2 m) (eval
memtree m) (λ (x::nat). (if (x = nat (get_val (eval tree1 m))) then (get_val (eval tree2 m))
```

```
else ((get_mem (eval memtree m)) x))) ident)"
  "eval (MEMORY mold ident) m = (MEMORY m ident)"
```

consts

```
  pic_tree_vals :: "SSATree ⇒ valuevector ⇒ valuevector"
```

primrec

```
  "pic_tree_vals (CONST val ident) vv = (λ x. (if (x = ident) then val else (vv x)))"
  "pic_tree_vals (PHI pa val ident) vv = (λ x. (if (x = ident) then val else (vv x)))"
  "pic_tree_vals (NODE operat tree1 tree2 val ident) vv  = (pic_tree_vals tree1 (pic_tree_vals
tree2 (λ x. (if (x = ident) then val else (vv x)))))"
  "pic_tree_vals (LOAD  tree memtree val ident) vv = (pic_tree_vals memtree (pic_tree_vals
tree ((λ x. (if (x = ident) then val else (vv x)) ))))"
  "pic_tree_vals (STORE  tree1 tree2 memtree val ident) vv  = (pic_tree_vals memtree
(pic_tree_vals tree2 (pic_tree_vals tree1 vv)))"
  "pic_tree_vals (MEMORY m ident) vv = vv"
```

consts

```
  eval_phi_list:: "SSATree list ⇒ valuevector ⇒ nat ⇒ SSATree list"
```

primrec

```
  "eval_phi_list [] svv rang = []"
  "eval_phi_list (x#xs) svv rang = ((eval_phi x svv rang )#(eval_phi_list xs svv rang ))"
```

consts

```
  eval_tree_list::"SSATree list ⇒ memory ⇒ SSATree list"
```

primrec

```
  "eval_tree_list []   memo = []"
  "eval_tree_list (tree#xs)  memo = (eval tree memo) # (eval_tree_list xs   memo) "
```

consts

```
  pic_tree_vals_list ::"SSATree list ⇒ valuevector  ⇒ valuevector"
```

primrec

```
  "pic_tree_vals_list [] vv = vv"
  "pic_tree_vals_list (x#xs) vv = (pic_tree_vals_list xs (pic_tree_vals x vv))"
```

consts

```
  pic_tree_mem :: "SSATree ⇒ nat ⇒ memory ⇒ memory"
```

primrec

```
  "pic_tree_mem (CONST val ident) valm m = m"
  "pic_tree_mem (PHI pa val ident) valm m = m"
  "pic_tree_mem (NODE operat tree1 tree2 val ident) valm m = (pic_tree_mem tree1 valm
(pic_tree_mem tree2 valm m))"
  "pic_tree_mem (LOAD  tree memtree val ident) valm m = (pic_tree_mem memtree valm
(pic_tree_mem tree valm m))"
  "pic_tree_mem (STORE  tree1 tree2 memtree val ident) valm m  = (if (valm = ident) then val
else (pic_tree_mem memtree valm (pic_tree_mem tree2 valm (pic_tree_mem tree1 valm m))))"
```

```
  "pic_tree_mem (MEMORY mold ident) valm m = (if (valm = ident) then mold else m)"


consts
  pic_mem_list :: "SSATree list ⇒ nat ⇒ memory ⇒ memory"
primrec
  "pic_mem_list [] valm m = m"
  "pic_mem_list (x#xs) valm m = pic_mem_list xs valm (pic_tree_mem x valm m)"




constdefs
  process_block :: "BASICBLOCK ⇒ state ⇒ state"
  "process_block b s ≡
  (
  case b of (NEW valn valm (tt,rtt) (ft,rft) treelist) ⇒ case s of (NEWSTATE (curr,pred,rang)
vv memo) ⇒
  (NEWSTATE

  (
  (if  (((pic_tree_vals_list (eval_tree_list (eval_phi_list treelist vv rang)  memo) vv) valn)
= 0) then
  (ft,curr,rft) else (tt,curr,rtt))
  )


 (pic_tree_vals_list (eval_tree_list (eval_phi_list treelist vv rang)  memo) vv)

 (pic_mem_list (eval_tree_list (eval_phi_list treelist vv rang)  memo) valm memo)

  ))"


constdefs
get_bblock_from_id :: "cfg ⇒ nat ⇒ BASICBLOCK"
"get_bblock_from_id g ident ≡ (g ! ident)"

constdefs
step :: "cfg ⇒ state ⇒ state"
"step g s ≡ (process_block (get_bblock_from_id g (case s of (NEWSTATE (curr,pred,rang) vv
memo) ⇒ curr)) s)"

consts
stepn :: "cfg ⇒ state ⇒ nat ⇒ state"
primrec
"stepn g s 0 = s"
"stepn g s (Suc n) = (stepn g (step g s) n)"

end
```

C. Beispiele

In diesem Abschnitt sind die zwei konkreten Beispiele zu finden, die auch in Kapitel 5 vorgestellt wurden.

theory *wlexample = eval + basic + Main:*

constdefs *twe_graph :: "cfg"*
"twe_graph ≡ [
*(*First BB: COND Block*)*
(NEW
2 999
(2,0) (1,0)
[(Doing nothing except changing value number ⇒ dominance frontier *)*
(NODE EQL

*(PHI [20,7] 0 0) (*i*)*
(CONST 0 1)
0 2)
,
*(PHI [21,10] 0 3) (*j*)*
]

)

,
*(*While Body*)*

(NEW
4 999
(0,1) (0,0) (es kann nur erster Fall auftreten *)*
[
(NODE SUB
*(PHI [0] 0 5) (*i*)*
(CONST 1 6)
0 7)
,
(NODE ADD
(PHI [3] 0 8)

```
  (CONST 2 9)
  0 10)
  ]

  )

  ,

(*Last BB*)
  (NEW
  11 999
  (2,0) (2,0)
  [
  (NODE ADD
  (PHI [3] 0 12) (*j*)
  (CONST 0 13)
  0 14)
  ]
  )
  ]"

constdefs twe_startstate :: "state"
  "twe_startstate ≡ NEWSTATE (0,0,0) (λ x .([0,0,0, 0,1,0,1,0,0, 2,0,1,0,27,0, 0,0,0,0,0,10,15]
! x)) (λ x. 0)"

lemma [simp]:"step twe_graph twe_startstate =
  NEWSTATE (1, 0, 0)  (λx. if x = 3 then 15
        else if x = 0 then 10 else if x = Suc 0 then 0 else if x = 2 then 0 else [0, 0, 0,
0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 27, 0, 0, 0, 0, 0, 0, 10, 15] ! x)  (λ x. 0)"
apply (simp add :  twe_graph_def twe_startstate_def)
apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done

lemma [simp]: "step twe_graph (step twe_graph twe_startstate) =
  NEWSTATE (0, 1, 1)   (λx. if x = 8 then 15
        else if x = 9 then 2
             else if x = 10 then 17
                  else if x = 5 then 10
                       else if x = 6 then 1
                            else if x = 7 then 9
                                 else if x = 3 then 15
                                      else if x = 0 then 10
                                           else if x = Suc 0 then 0 else if x = 2 then 0
else [0, 0, 0, 0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 27, 0, 0, 0, 0, 0, 0, 10, 15] ! x) (λ x. 0)"

 apply (simp add :  twe_graph_def twe_startstate_def)
apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done

lemma [simp]: "step twe_graph (step twe_graph (step twe_graph twe_startstate)) =
  NEWSTATE (1, 0, 0)   (λx. if x = 3 then 17
        else if x = 0 then 9
```

```
             else if x = Suc 0 then 0
                  else if x = 2 then 0
                       else if x = 8 then 15
                            else if x = 9 then 2
                                 else if x = 10 then 17
                                      else if x = 5 then 10
                                           else if x = 6 then 1
                                                else if x = 7 then 9
                                                     else if x = 3 then 15
                                                          else if x = 0 then 10
                                                               else if x = Suc 0 then 0
                                                                    else if x = 2 then 0
else [0, 0, 0, 0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 27, 0, 0, 0, 0, 0, 0, 10, 15] ! x)  (λ x. 0)"
```

apply *(simp add : twe_graph_def twe_startstate_def)*

apply *(simp add: step_def get_bblock_from_id_def process_block_def ADD_def EQL_def SUB_def GRT_def)*
done

lemma *[simp]: "step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph twe_startstate)))))))))))))))))))) = NEWSTATE (0, 1, 1) (λx. if x = 8 then 33*

```
         else if x = 9 then 2
              else if x = 10 then 35
                   else if x = 5 then 1
                        else if x = 6 then 1
                             else if x = 7 then 0
                                  else if x = 3 then 33
                                       else if x = 0 then 1
                                            else if x = Suc 0 then 0
                                                 else if x = 2 then 0
                                                      else if x = 8 then 31
                                                           else if x = 9 then 2
                                                                else if x = 10 then 33
                                                                     else if x = 5 then 2
                                                                          else if x = 6 then
1
                                                                               else if x =
7 then 1
                                                                                    else if
x = 3 then 31
   else if x = 0 then 2
        else if x = Suc 0 then 0
             else if x = 2 then 0
                  else if x = 8 then 29
                       else if x = 9 then 2
                            else if x = 10 then 31
                                 else if x = 5 then 3
                                      else if x = 6 then 1
                                           else if x = 7 then 2
                                                else if x = 3 then 29
                                                     else if x = 0 then 3
```

```
                                                                  else if x = Suc 0 then 0
                                                                       else if x = 2 then 0
                                                                            else if x = 8 then 27
                                                                                 else if x = 9 then
2
                                                                                      else if x =
10 then 29
                                                                                           else if
x = 5 then 4
  else if x = 6 then 1
       else if x = 7 then 3
            else if x = 3 then 27
                 else if x = 0 then 4
                      else if x = Suc 0 then 0
                           else if x = 2 then 0
                                else if x = 8 then 25
                                     else if x = 9 then 2
                                          else if x = 10 then 27
                                               else if x = 5 then 5
                                                    else if x = 6 then 1
                                                         else if x = 7 then 4
                                                              else if x = 3 then 25
                                                                   else if x = 0 then 5
                                                                        else if x = Suc 0
then 0
                                                                             else if x = 2
then 0
                                                                                  else if
x = 8 then 23
 else if x = 9 then 2
      else if x = 10 then 25
           else if x = 5 then 6
                else if x = 6 then 1
                     else if x = 7 then 5
                          else if x = 3 then 23
                               else if x = 0 then 6
                                    else if x = Suc 0 then 0
                                         else if x = 2 then 0
                                              else if x = 8 then 21
                                                   else if x = 9 then 2
                                                        else if x = 10 then 23
                                                             else if x = 5 then 7
                                                                  else if x = 6 then 1
                                                                       else if x = 7 then
6
                                                                            else if x = 3
then 21
                                                                                 else if x
= 0 then 7
else if x = Suc 0 then 0
     else if x = 2 then 0
          else if x = 8 then 19
               else if x = 9 then 2
                    else if x = 10 then 21
```

```
                        else if x = 5 then 8
                             else if x = 6 then 1
                                  else if x = 7 then 7
                                       else if x = 3 then 19
                                            else if x = 0 then 8
                                                 else if x = Suc 0 then 0
                                                      else if x = 2 then 0
                                                           else if x = 8 then 17
                                                                else if x = 9 then 2
                                                                     else if x = 10 then
19
                                                                          else if x = 5 then
9
                                                                               else if x
= 6 then 1
                                                                                    else
if x = 7 then 8
    else if x = 3 then 17
         else if x = 0 then 9
              else if x = Suc 0 then 0
                   else if x = 2 then 0
                        else if x = 8 then 15
                             else if x = 9 then 2
                                  else if x = 10 then 17
                                       else if x = 5 then 10
                                            else if x = 6 then 1
                                                 else if x = 7 then 9
                                                      else if x = 3 then 15
                                                           else if x = 0 then 10
                                                                else if x = Suc 0 then 0
                                                                     else if x = 2 then 0
else [0, 0, 0, 0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 27, 0, 0, 0, 0, 0, 0, 10, 15] ! x)  (λ x. 0)"
```

apply *(simp add : twe_graph_def twe_startstate_def)*

apply *(simp add: step_def get_bblock_from_id_def process_block_def ADD_def EQL_def SUB_def GRT_def)*
done

lemma *[simp]: "step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph twe_startstate))))))))))))))))))) = NEWSTATE (2, 0, 0)*

```
     (λx. if x = 3 then 35
          else if x = 0 then 0
               else if x = Suc 0 then 0
                    else if x = 2 then 1
                         else if x = 8 then 33
                              else if x = 9 then 2
                                   else if x = 10 then 35
                                        else if x = 5 then 1
                                             else if x = 6 then 1
                                                  else if x = 7 then 0
                                                       else if x = 3 then 33
```

```
                                                                  else if x = 0 then 1
                                                                       else if x = Suc 0 then 0
                                                                            else if x = 2 then 0
                                                                                 else if x = 8 then
31
                                                                                      else if x
= 9 then 2
                                                                                           else
if x = 10 then 33
    else if x = 5 then 2
         else if x = 6 then 1
              else if x = 7 then 1
                   else if x = 3 then 31
                        else if x = 0 then 2
                             else if x = Suc 0 then 0
                                  else if x = 2 then 0
                                       else if x = 8 then 29
                                            else if x = 9 then 2
                                                 else if x = 10 then 31
                                                      else if x = 5 then 3
                                                           else if x = 6 then 1
                                                                else if x = 7 then 2
                                                                     else if x = 3 then 29
                                                                          else if x = 0 then
3
                                                                               else if x =
Suc 0 then 0
                                                                                    else if
x = 2 then 0
   else if x = 8 then 27
        else if x = 9 then 2
             else if x = 10 then 29
                  else if x = 5 then 4
                       else if x = 6 then 1
                            else if x = 7 then 3
                                 else if x = 3 then 27
                                      else if x = 0 then 4
                                           else if x = Suc 0 then 0
                                                else if x = 2 then 0
                                                     else if x = 8 then 25
                                                          else if x = 9 then 2
                                                               else if x = 10 then 27
                                                                    else if x = 5 then 5
                                                                         else if x = 6 then
1
                                                                              else if x =
7 then 4
                                                                                   else if
x = 3 then 25
  else if x = 0 then 5
       else if x = Suc 0 then 0
            else if x = 2 then 0
                 else if x = 8 then 23
                      else if x = 9 then 2
```

```
                        else if x = 10 then 25
                             else if x = 5 then 6
                                  else if x = 6 then 1
                                       else if x = 7 then 5
                                            else if x = 3 then 23
                                                 else if x = 0 then 6
                                                      else if x = Suc 0 then 0
                                                           else if x = 2 then 0
                                                                else if x = 8 then 21
                                                                     else if x = 9 then
2
                                                                          else if x = 10
then 23
                                                                               else if
x = 5 then 7
 else if x = 6 then 1
      else if x = 7 then 6
           else if x = 3 then 21
                else if x = 0 then 7
                     else if x = Suc 0 then 0
                          else if x = 2 then 0
                               else if x = 8 then 19
                                    else if x = 9 then 2
                                         else if x = 10 then 21
                                              else if x = 5 then 8
                                                   else if x = 6 then 1
                                                        else if x = 7 then 7
                                                             else if x = 3 then 19
                                                                  else if x = 0 then 8
                                                                       else if x = Suc 0 then
0
                                                                            else if x = 2
then 0
                                                                                 else if x
= 8 then 17
else if x = 9 then 2
     else if x = 10 then 19
          else if x = 5 then 9
               else if x = 6 then 1
                    else if x = 7 then 8
                         else if x = 3 then 17
                              else if x = 0 then 9
                                   else if x = Suc 0 then 0
                                        else if x = 2 then 0
                                             else if x = 8 then 15
                                                  else if x = 9 then 2
                                                       else if x = 10 then 17
                                                            else if x = 5 then 10
                                                                 else if x = 6 then 1
                                                                      else if x = 7 then 9
                                                                           else if x = 3 then
15
                                                                                else if x
= 0 then 10
```

```
                                                                                       else
if x = Suc 0 then 0
    else if x = 2 then 0 else [0, 0, 0, 0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 27, 0, 0, 0, 0, 0, 0,
10, 15] ! x)(λ x. 0) "
 apply (simp add :  twe_graph_def twe_startstate_def)
apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done

lemma [simp]: "step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph
(step twe_graph (step twe_graph (step twe_graph (step twe_graph
(step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph
(step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph
(step twe_graph twe_startstate))))))))))))))))))))) =
NEWSTATE (2, 2, 0) (λx. if x = 12 then 35
         else if x = 13 then 0
              else if x = 14 then 35
                   else if x = 3 then 35
                        else if x = 0 then 0
                             else if x = Suc 0 then 0
                                  else if x = 2 then 1
                                       else if x = 8 then 33
                                            else if x = 9 then 2
                                                 else if x = 10 then 35
                                                      else if x = 5 then 1
                                                           else if x = 6 then 1
                                                                else if x = 7 then 0
                                                                     else if x = 3 then 33
                                                                          else if x = 0 then
1
                                                                               else if x =
Suc 0 then 0
                                                                                    else if
x = 2 then 0
   else if x = 8 then 31
        else if x = 9 then 2
             else if x = 10 then 33
                  else if x = 5 then 2
                       else if x = 6 then 1
                            else if x = 7 then 1
                                 else if x = 3 then 31
                                      else if x = 0 then 2
                                           else if x = Suc 0 then 0
                                                else if x = 2 then 0
                                                     else if x = 8 then 29
                                                          else if x = 9 then 2
                                                               else if x = 10 then 31
                                                                    else if x = 5 then 3
                                                                         else if x = 6 then
1
                                                                              else if x =
7 then 2
                                                                                   else if
x = 3 then 29
```

```
  else if x = 0 then 3
       else if x = Suc 0 then 0
            else if x = 2 then 0
                 else if x = 8 then 27
                      else if x = 9 then 2
                           else if x = 10 then 29
                                else if x = 5 then 4
                                     else if x = 6 then 1
                                          else if x = 7 then 3
                                               else if x = 3 then 27
                                                    else if x = 0 then 4
                                                         else if x = Suc 0 then 0
                                                              else if x = 2 then 0
                                                                   else if x = 8 then 25
                                                                        else if x = 9 then
2
                                                                             else if x = 10
then 27
                                                                                  else if
x = 5 then 5
 else if x = 6 then 1
      else if x = 7 then 4
           else if x = 3 then 25
                else if x = 0 then 5
                     else if x = Suc 0 then 0
                          else if x = 2 then 0
                               else if x = 8 then 23
                                    else if x = 9 then 2
                                         else if x = 10 then 25
                                              else if x = 5 then 6
                                                   else if x = 6 then 1
                                                        else if x = 7 then 5
                                                             else if x = 3 then 23
                                                                  else if x = 0 then 6
                                                                       else if x = Suc 0 then
0
                                                                            else if x = 2
then 0
                                                                                 else if x
= 8 then 21
else if x = 9 then 2
     else if x = 10 then 23
          else if x = 5 then 7
               else if x = 6 then 1
                    else if x = 7 then 6
                         else if x = 3 then 21
                              else if x = 0 then 7
                                   else if x = Suc 0 then 0
                                        else if x = 2 then 0
                                             else if x = 8 then 19
                                                  else if x = 9 then 2
                                                       else if x = 10 then 21
                                                            else if x = 5 then 8
                                                                 else if x = 6 then 1
```

```
                                                                    else if x = 7 then 7
                                                                         else if x = 3 then
19
                                                                              else if x
= 0 then 8
                                                                                   else
if x = Suc 0 then 0
    else if x = 2 then 0
         else if x = 8 then 17
              else if x = 9 then 2
                   else if x = 10 then 19
                        else if x = 5 then 9
                             else if x = 6 then 1
                                  else if x = 7 then 8
                                       else if x = 3 then 17
                                            else if x = 0 then 9
                                                 else if x = Suc 0 then 0
                                                      else if x = 2 then 0
                                                           else if x = 8 then 15
                                                                else if x = 9 then 2
                                                                     else if x = 10 then 17
                                                                          else if x = 5 then
10
                                                                               else if x =
6 then 1
                                                                                    else if
x = 7 then 9
   else if x = 3 then 15
        else if x = 0 then 10 else if x = Suc 0 then 0 else if x = 2 then 0 else [0, 0, 0,
0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 27, 0, 0, 0, 0, 0, 0, 10, 15] ! x) (λ x. 0)"
 apply (simp add :  twe_graph_def twe_startstate_def)

apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done

end
```

```
theory wlmemex = Main + basic + eval:
```

```
constdefs twe_graph :: "cfg"
  "twe_graph ≡ [
(*First BB: COND Block*)
  (NEW
```

```
  2 0
  (2,0) (1,0)

  [
  (NODE EQL

  (PHI [20,6] 0 0)  (*i*)
  (CONST 0 1)
  0 2)
  ]

  )

  ,
(*While Body*)

  (NEW
  3 7

  (0,1) (0,0) (* es kann nur erster Fall auftreten *)

  [
  (NODE SUB
  (PHI [0] 0 4) (*i*)
  (CONST 1 5)
  0 6)
  ,
  (STORE
  (PHI [0] 0 4)
  (PHI [0] 0 4)
  (MEMORY (λ x.0) 0)
  (λ x.13) 7)
  ]
  )

  ,

(*Last BB*)
  (NEW

  11 0

  (2,0) (2,0)
  []

  )
  ]"
```

constdefs *twe_startstate :: "state"*
"twe_startstate ≡ NEWSTATE (0,0,0) (λ x.[0,0,0,1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,4,0]!x) (λ x. 0)"

lemma *[simp]: "step twe_graph twe_startstate =*

```
NEWSTATE (1, 0, 0) (λx. if x = 0 then 4 else if x = Suc 0 then 0 else if x = 2 then 0 else
[0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0] ! x) (λ x. 0)"
apply (simp add :  twe_graph_def twe_startstate_def)
apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done

lemma [simp]: "step twe_graph (step twe_graph twe_startstate) =
NEWSTATE (0, 1, 1) (λx. if x = 4 then 4
         else if x = 4 then 4
              else if x = 4 then 4
                   else if x = 5 then 1
                        else if x = 6 then 3
                             else if x = 0 then 4
                                  else if x = Suc 0 then 0 else if x = 2 then 0 else [0, 0,
0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0] ! x) (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x)
  "
 apply (simp add :  twe_graph_def twe_startstate_def)
apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done

lemma [simp]: "step twe_graph (step twe_graph (step twe_graph twe_startstate)) =
NEWSTATE (1, 0, 0)(λx. if x = 0 then 3
         else if x = Suc 0 then 0
              else if x = 2 then 0
                   else if x = 4 then 4
                        else if x = 4 then 4
                             else if x = 4 then 4
                                  else if x = 5 then 1
                                       else if x = 6 then 3
                                            else if x = 0 then 4
                                                 else if x = Suc 0 then 0 else if x = 2 then
0 else [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0] ! x)  (λx. if x
= 4 then get_val (eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx.
0)) x)
 "
 apply (simp add :  twe_graph_def twe_startstate_def)
apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done


lemma [simp]: "step twe_graph (step twe_graph (step twe_graph (step twe_graph twe_startstate)))
=
NEWSTATE (0, 1, 1) (λx. if x = 4 then 3
         else if x = 4 then 3
              else if x = 4 then 3
                   else if x = 5 then 1
                        else if x = 6 then 2
                             else if x = 0 then 3
                                  else if x = Suc 0 then 0
                                       else if x = 2 then 0
```

```
                                        else if x = 4 then 4
                                             else if x = 4 then 4
                                                  else if x = 4 then 4
                                                       else if x = 5 then 1
                                                            else if x = 6 then 3
                                                                 else if x = 0 then 4
                                                                      else if x = Suc
0 then 0
                                                                                    else if x =
2 then 0
                                                                                          else [0,
0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 4, 0] ! x) (λx. if x = 3 then get_val
(eval (PHI [0] 3 4) (λx. if x = 4 then get_val (eval (PHI [0] 4 4) (λx. 0)) else get_mem
(eval (MEMORY (λx. 0) 0) (λx. 0)) x))
         else get_mem (eval (MEMORY (λx. 0) 0) (λx. if x = 4 then get_val (eval (PHI [0]
4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x)) x)"
 apply (simp add :  twe_graph_def twe_startstate_def)
apply (simp add: step_def get_bblock_from_id_def   process_block_def ADD_def EQL_def SUB_def
GRT_def)
done

lemma [simp]: "step twe_graph (step twe_graph (step twe_graph (step twe_graph (step twe_graph
(step twe_graph (step twe_graph (step twe_graph (step twe_graph twe_startstate)))))))) =
NEWSTATE (2,0,0) (λx. if x = 0 then 0
         else if x = Suc 0 then 0
              else if x = 2 then 1
                   else if x = 4 then 1
                        else if x = 4 then 1
                             else if x = 4 then 1
                                  else if x = 5 then 1
                                       else if x = 6 then 0
                                            else if x = 0 then 1
                                                 else if x = Suc 0 then 0
                                                      else if x = 2 then 0
                                                           else if x = 4 then 2
                                                                else if x = 4 then 2
                                                                     else if x = 4 then 2
                                                                          else if x = 5 then
1
                                                                               else if x =
6 then 1
                                                                                    else if
x = 0 then 2
   else if x = Suc 0 then 0
        else if x = 2 then 0
             else if x = 4 then 3
                  else if x = 4 then 3
                       else if x = 4 then 3
                            else if x = 5 then 1
                                 else if x = 6 then 2
                                      else if x = 0 then 3
                                           else if x = Suc 0 then 0
                                                else if x = 2 then 0
                                                     else if x = 4 then 4
```

```
                                                                  else if x = 4 then 4
                                                                       else if x = 4 then 4
                                                                            else if x = 5 then 1
                                                                                 else if x = 6 then
3
                                                                                      else if x =
0 then 4
                                                                                           else if
x = Suc 0 then 0
  else if x = 2 then 0 else [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 4,
0] ! x) (λx. if x = Suc 0
         then get_val (eval (PHI [0] 1 4)
                        (λx. if x = 2
                             then get_val (eval (PHI [0] 2 4)
                                            (λx. if x = 3
                                                 then get_val (eval (PHI [0] 3 4)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                 else get_mem (eval (MEMORY (λx. 0) 0)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                               x))
                             else get_mem (eval (MEMORY (λx. 0) 0)
                                            (λx. if x = 3
                                                 then get_val (eval (PHI [0] 3 4)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                 else get_mem (eval (MEMORY (λx. 0) 0)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                               x))
                                          x))
         else get_mem (eval (MEMORY (λx. 0) 0)
                        (λx. if x = 2
                             then get_val (eval (PHI [0] 2 4)
                                            (λx. if x = 3
                                                 then get_val (eval (PHI [0] 3 4)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                 else get_mem (eval (MEMORY (λx. 0) 0)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                               x))
                             else get_mem (eval (MEMORY (λx. 0) 0)
                                            (λx. if x = 3
                                                 then get_val (eval (PHI [0] 3 4)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                 else get_mem (eval (MEMORY (λx. 0) 0)
                                                                (λx. if x = 4 then get_val
(eval (PHI [0] 4 4) (λx. 0)) else get_mem (eval (MEMORY (λx. 0) 0) (λx. 0)) x))
                                                               x))
                                          x))
              x)"
```

apply *(simp add : twe_graph_def twe_startstate_def)*
apply *(simp add: step_def get_bblock_from_id_def process_block_def ADD_def EQL_def SUB_def GRT_def)*
done

end