Verifikation abstrakter Zustandsmaschinen

Disseration

zur Erlangung des Grades eines Doktors
der Naturwissenschaften (Dr. rer. nat.)
im Fachbereich Informatik
der Universität Ulm

vorgelegt von

Gerhard Schellhorn
geboren in Balingen

Amtierender Dekan: Prof. Dr. Uwe Schöning

Gutachter: Prof. Dr. Wolfgang Reif (Universität Ulm)
Prof. Dr. Hemuth Partsch (Universität Ulm)
Prof. Dr. Egon Börger (Universität Pisa)

Tag der Prüfung: 9. Juni 1999

Danksagung

Mein herzlicher Dank gilt allen, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Da wäre zunächst mein Betreuer Prof. Reif, der mir mit seinen Ratschlägen und seiner Kritik immer wieder die notwendige Motivation vermittelt hat, diese Arbeit zustande zu bringen.

Dann meine Kollegen Kurt Stenzel, Michael Balser und Andreas Thums, die mir sowohl durch Kritik, als auch durch ihre Bereitschaft, mir anderweitige Arbeiten abzunehmen, geholfen haben. Besonders herausheben möchte ich Kurt Stenzel, der mich durch viele Diskussionen zum Thema dieser Arbeit als auch durch seine Arbeiten zur Verbesserung des KIV-Systems unterstützt hat.

Schließlich gebührt ein herzlicher Dank auch den studentischen Hilfskräften, die an der Prolog-WAM-Fallstudie mitgearbeitet haben. Da ist zunächst Wolfgang Ahrendt, dessen Diplomarbeit einen wichtigen Beitrag geleistet hat. Dann die Studenten Harald Vogt, Christoph Duelli und Tobias Vollmer, die ebenfalls viele Stunden mit den Beweisen der Fallstudie verbracht haben. Den letzten beiden danke ich auch für ihre Arbeit beim Korrekturlesen dieser Arbeit.

Zum Schluß möchte ich auch meinen Eltern, meinen Geschwistern und meinen beiden Neffen danken. Sie alle haben auf ihre jeweils eigene Art einen wichtigen Anteil am Gelingen dieser Arbeit.

Ulm, im Juni 1999

Kurzfassung

Diese Arbeit stammt aus dem Themengebiet der Anwendung formaler Methoden im Software Engineering. Grundlage der Arbeit bildet die in [Gur95] definierte Spezifikationssprache der Abstrakten Zustandsmaschinen (engl. abstract state machines, kurz ASMs).

Inhalt der Arbeit ist die Entwicklung praktisch anwendbarer Werkzeugunterstützung für ASMs, sowohl für die Spezifikation, als auch für die Verifikation von Verfeinerungen. Diese soll die Entwicklung korrekter Software von einer abstrakten Anforderungsspezifikation bis hin zu einer durch schrittweise Verfeinerung gewonnenen Implementierung möglich machen. Der Inhalt gliedert sich in vier Teile:

- Einbettung von ASM-Spezifikationen in eine Logik: Die Arbeit definiert eine 1:1-Abbildung von ASMs in die Dynamische Logik (DL). Damit wird der formale Nachweis von ASM-Eigenschaften möglich.

- Modularisierung von Korrektheitsnachweisen für Verfeinerungen: Zwei aus der Literatur bekannte Verfeinerungsbegriffe wurden in DL formalisiert. Ein allgemeines Modularisierungstheorem für den Korrektheitsnachweis von ASM-Verfeinerungen wurde entwickelt, das die bisher aus der Literatur bekannten Theoreme verallgemeinert.

- Implementierung der Ergebnisse im KIV-System: Das KIV-System ist ein Spezifikations- und Verifikationswerkzeug, das algebraische Spezifikationen und DL unterstützt. Eine Reihe von Erweiterungen und Verbesserungen des Systems waren notwendig, um ASMs und ASM-Verfeinerungen zu unterstützen.

- Demonstration der praktischen Einsetzbarkeit der entwickelten Konzepte an einer großen Fallstudie: Die gewählte Fallstudie stammt aus dem Compilerbau und behandelt die Übersetzung von Prolog-Programmen in Assemblercode der Warren Abstract Machine (WAM). Eine informelle Darstellung, die einen als ASM beschriebenen Prolog-Interpreter in 12 systematischen Verfeinerungen in die WAM transformiert, war durch [BR95] vorgegeben. Die formale Spezifikation und Verifikation von 8 der 12 Verfeinerungen in 9 Monaten nahm eine großen Teil des Umfangs der Arbeit ein. Der Vergleich zu zwei anderen Fallstudien mit dem gleichen Thema zeigt,

daß der notwendige Verifikationsaufwand durch die entwickelte Theorie für ASM-Verfeinerungen deutlich geringer war.

Inhaltsverzeichnis

Teil I

Abstrakte Zustandsmaschinen und Korrektheitsbeweise für Verfeinerungen

Kapitel 1

Einleitung

Diese Arbeit stammt aus dem Themengebiet der Anwendung formaler Methoden im Software Engineering. Ziel ist die Entwicklung korrekter Software für sicherheitskritische Anwendungen.

Die Anwendung formaler Methoden setzt eine geeignete *Spezifikationssprache* voraus, in der die Anforderungen an die Software abstrakt und unzweideutig beschrieben werden können. Damit werden sie einer mathematischen Analyse zugänglich. *Validierung durch Beweisen* etwa durch den Nachweis von Sicherheitseigenschaften wird schon in frühen Phasen der Softwareentwicklung möglich, in denen noch keine Implementierung vorliegt. Um anschließend von den Anforderungen systematisch zu einer Implementierung überzugehen, bedarf es weiterhin eines *Verfeinerungsbegriffs*, der es erlaubt, schrittweise und unter Erhaltung der Korrektheit von den Anforderungen zu implementiertem Code überzugehen.

Beweise zur Validierung von Spezifikationen sowie zum Korrektheitsnachweis von Verfeinerungen sind in verschiedenen Detaillierungsgraden möglich, von informellen Beweisskizzen über mathematische Beweise bis hin zu formalen Beweisen in einem maschinenunterstützten Kalkül.

Ziel dieser Arbeit ist es, die Spezifikationssprache der *Abstrakten Zustandsmaschinen* (*Abstract State Machines*, im folgenden kurz ASMs, [Gur95]) im Spezifikations- und Verifikationswerkzeug *KIV* verfügbar zu machen. Insbesondere wird für einen Verfeinerungsbegriff, der bisher existierende verallgemeinert, einen systematische Beweisunterstützung entwickelt, und deren Praxistauglichkeit an einer großen Fallstudie demonstriert.

Die Wahl der Spezifikationssprache begründet sich aus der Tatsache, daß es im wesentlichen zwei Klassen von Spezifikationssprachen gibt: Die erste Klasse sind algebraische Spezifikationssprachen [Wir90], [Gau92], [CoF97] sowie die Verallgemeinerung zu Prozeßalgebren [Mil89], [Bae90]. Diese sehen ein Softwaresystem als allgemeine Datenstruktur mit einer Reihe von darauf definierten Funktionen und Relationen an, die geeignet zusammenwirken. Mathematisch wird ein Softwaresystem als *Algebra* modelliert, eine Spezifikation beschreibt dann eine Klasse von Algebren als mögliche Implementierungen. Ein häufig ver-

wendeter Spezialfall algebraischer Spezifikation ist die modellbasierte Spezifikation, bei der ein Softwaresystem aus Grunddatentypen der Mengenlehre (wie Tupel, Funktionen, Potenzmengen) aufgebaut wird.

Die zweite Klasse sind zustandsbasierte Spezifikationssprachen, die ein Softwaresystem durch *Zustände*, mögliche *Zustandsübergänge* und daraus resultierende Abläufe modellieren. Beispiele sind z. B. Z [Spi88], VDM [Jon90] und RAISE [JC94]. Auch ASMs gehören zu dieser Klasse von Spezifikationssprachen. Zustandsbasierte Spezifikationssprachen bauen zur Beschreibung von Zustandskomponenten auf algebraischen Spezifikationssprachen auf. In gewissem Sinn sind sie sogar ein Spezialfall von algebraischen Spezifikationen, da Zustandsübergänge als Funktionen und Relationen über Zuständen modelliert werden können. Deshalb unterstützen viele Verifikationswerkzeuge nur algebraische Spezifikationen. Der Nachteil dieses Ansatzes ist natürlich, daß die Grundkonzepte von Zustandsübergangssystemen erst nachmodelliert werden müssen.

Das KIV-System unterstützte bisher traditionell den algebraischen Ansatz zur Entwicklung von Software: In KIV gibt es strukturierte, algebraische Spezifikationen mit entsprechender Beweisunterstützung [RSSB98]. Ein ausgefeiltes Verfeinerungskonzept wird unterstützt, das die strukturierte Implementierung von Spezifikationen durch Programmmoduln erlaubt [Rei95].

Diese Arbeit leistet nun einen Beitrag zur Unterstützung des Konzepts der zustandsbasierten Spezifikationen in KIV. ASMs als die zustandsbasierte Spezifikationssprache zu wählen ergab sich im wesentlichen daraus, daß ASMs einen konzeptuell sehr einfachen, aber dennoch sehr flexiblen Ansatz zur Spezifikation von Zustandsübergangssystemen bieten, der eine große Vielfalt an Fallstudien ermöglicht. So wurden ASMs bereits erfolgreich in Fallstudien angewandt, die sich mit so unterschiedlichen Themen wie der Modellierung von Programmiersprachen (z.B. Prolog [BR94], C [GH93] und Java [BS98b]), Kommunikationsprotokollen (z.B. Bakery-Algorithmus [BGR95]), Compilerkorrektheit (z.B. Occam [BD96], Prolog [BR95] und Java [BS98a], [Sch99]), verteilten Systemen (z.B. PVM [BG95]) und Hardwarearchitekturen (z.B. DLX [BM96]) befassen. Einen Überblick über eine große Zahl weiterer Anwendungnen gibt [BH98] sowie die im Internet unter den URLs *http://www.eecs.umich.edu/gasm/* und *http://www.uni-paderborn.de/cs/asm/* verfügbaren Web-Seiten. Die Korrektheitsbeweise sind bisher in den meisten Fallstudien mathematische Beweise, die nicht durch mechanische Verifikationssysteme unterstützt werden.

Um den Formalismus der ASMs, den in Kapitel 2 beschrieben wird, zu unterstützen, mußte zunächst eine Einbettung in die Spezifikationssprache von KIV vorgenommen werden. Dabei hat KIV gegenüber rein algebraischen Spezifikationssystemen den Vorteil, daß Programme über abstrakten Datentypen (deren Semantik Zustandsübergänge sind) bereits unterstützt werden. Insofern konnte als erstes Egebnis dieser Arbeit eine 1:1-Übersetzung der ASM-Regeln in abstrakte Programme definiert werden. Kapitel 3 beschreibt die Spezifikationssprache und die Logik von KIV, sowie die Erweiterungen, die im Rahmen dieser Arbeit daran durchgeführt wurden, und Kapitel 4 definiert die Übersetzung.

Die Einbettung in KIV gibt neben der Möglichkeit, ASMs formal zu spezifizieren auch die Möglichkeit, formale, systemunterstützte Beweise mit der vor-

handenen Programmlogik, der sogenannten *Dynamische Logik* (im folgenden kurz DL) zu führen. Um ASMs systematisch zu unterstützen, fehlt nun noch die Definition eines geeigneten Verfeinerungsbegriffs, der es gestattet, schrittweise und modular von einer abstrakten Spezifikation zu einer konkreten Implementierung überzugehen. Ein derartigen Verfeinerungsbegriff wird in Kapitel 5 definiert und es wird gezeigt, daß sich die Korrektheit einer Verfeinerung in DL ausdrücken läßt.

Den zentralen Kern der Arbeit bildet dann die Entwicklung von Beweisunterstützung für den modularen Nachweis der Korrektheit von Verfeinerungen in Kapitel 6. Ein entsprechendes Modularisierungstheorem wird zunächst in seiner einfachsten Form für die Verfeinerung deterministischer ASMs entwickelt. Anschließend werden verschiedene Verallgemeinerungen für indeterministische ASMs und für iterative Verfeinerung definiert und Bezüge zu anderen Korrektheitsbegriffen hergestellt. Das wesentliches Ergebnis ist eine Verallgemeinerung der bisher bekannten Theorie der Korrektheit von Verfeinerungen: Statt Abstraktionsfunktionen werden nun beliebigen Relationen verwendet, und statt kommutierenden Diagrammen mit je einer Regel, werden m:n-Diagramme mit beliebiger Zahl $m, n \geq 0$ von Regeln betrachtet.

Als eine kleine Anwendung zeigt anschließend Kapitel 7, daß sich die Korrektheit von Peephole-Optimierungen als Korollar aus dem Modularisierungstheorem ergibt.

Die in Kapitel 6 erarbeitete Theorie ist nicht aus der theoretischen Überlegung heraus entstanden, wie sich bestehende Verfeinerungskonzepte verallgemeinern lassen. Konzepte zur Verifikation von Software, die zwar theoretisch schöne Eigenschaften, aber keinen praktischen Nutzen haben, gibt es unseres Erachtens schon zu viele. Die Flexibilität eines Modularisierungskonzepts sowie die Qualität der Beweisunterstützung für die Korrektheitsbeweise muß sich vielmehr an der praktischen Anwendbarkeit beurteilen lassen. Deshalb wurde die Theorie und die Beweisunterstützung anhand einer realistischen, großen Fallstudie entwickelt.

Wir haben dazu als Fallstudie die Übersetzung von Prolog in Assemblercode der *Warren Abstract Machine* (WAM) gewählt. Für die Fallstudie lag bereits eine mathematische Analyse vor [BR95], auf die wir uns stützen konnten. Die Fallstudie zeigt eine vielfältige Bandbreite an Problemen, die bei der Verfeinerung von ASMs, insbesondere im Anwendungsgebiet Compilerkorrektheit auftreten. Die Fallstudie gehört mit einem Arbeitsaufwand von bisher 9 Monaten sicher zu den großen und anspruchsvollen Arbeiten in diesem Gebiet. Im zweiten Teil dieser Arbeit geben wir eine ausführliche Darstellung der Fallstudie, in der die ersten 8 der 12 Verfeinerungen aus [BR95] verifiziert wurden.

Wesentliches Resultat der Fallstudie ist die Demonstration der Leistungsfähigkeit der Theorie. Sie zeigt sich zum Beispiel im Vergleich zu parallel mit anderen Systemen durchgeführten Fallstudien zum gleichen Thema, die einen deutlich höheren Aufwand für kleinere Anteile an der Verifikation hatten. Auch wird die Theorie derzeit [Sch99] bei der Verifikation eines Java-Compilers eingesetzt.

Die Fallstudie zeigt auch den Gewinn, den eine rechnerunterstützte Verifika-

tion gegenüber einer mathematischen Analyse bringt. Obwohl wir der Meinung sind, daß die Analyse in [BR95] schon eine *sehr* gründliche und wohldurchdachte Analyse der Probleme darstellt und keine konzeptuellen Fehler enthält, konnten doch eine Reihe kleinerer Probleme aufgedeckt werden, die zu einem inkorrekten Compiler geführt hätten. Insofern zeigt diese Arbeit, daß sich der hohe Aufwand, den formale, systemunterstützte Verifikation mit sich bringt, lohnt, wenn die Anwendnung eine wirklich fehlerfreie Software (in diesem Fall einen fehlerfreien Compiler) erfordert.

Kapitel 2

Abstrakte Zustandsmaschinen

Abstrakte Zustandsmaschinen (engl. abstract state machines; ASMs) sind eine Spezifikationssprache zur Beschreibung von Soft- und Hardwaresystemen. Die Grundidee einer Abstrakten Zustandsmaschine ist die schrittweise Transformation eines Zustands mit Hilfe von Regeln. Damit gehören sie zur Gruppe der Spezifikationssprachen, deren Semantik ein Zustandsübergangssystem ist. Zustandsübergangssysteme werden im ersten Abschnitt eingeführt. Abschnitt 2.2 gibt dann die grundlegende Definition sequentieller ASMs. Eine Variante dieser Definition, die in der Prolog-WAM-Fallstudie verwendet wird, erklärt Abschnitt 2.3. Verteilte ASMs, mit denen verteilte Systeme modelliert werden können, werden schließlich in Abschnitt 2.4 erklärt. Eine umfassende Darstellung der ASMs, die neben den hier definierten grundlegenden Konzepten noch einige Erweiterungen definiert, gibt [Gur95].

2.1 Zustandsübergangssysteme

Die grundsätzliche Idee eines Zustandsübergangssystems ist die Transformation von Zuständen mit Hilfe von Regeln. Etwas formaler betrachtet, besteht ein Zustandsübergangssystem $ZS = (S, I, \rho)$ aus einer Menge S möglicher Zustände, einer Menge $I \subseteq S$ von initialen (oder Anfangs-)Zuständen sowie einer Zustandsübergangsrelation $\rho : S \times S$. $(st,st') \in \rho$ bedeutet, daß st' ein möglicher Nachfolgezustand von st ist. Eine Menge F von Finalzuständen läßt sich bei dieser Definition als die Menge der Zustände festlegen, für die es keinen Nachfolgezustand gibt. Zustandsübergangssysteme sind eine häufig gewählte, natürliche Formalisierung von Softwaresystemen, da eine typische Berechnung auf einem Rechner mit von-Neumann-Architektur einen Speicherzustand involviert, der durch einen Prozessor fortgeschaltet wird (= Zustandsübergangsrelation). Andere Beispiele sind z.B. alle Arten von endlichen Automaten (die Menge der Zustände ist dann die Menge aller Strings über einem Alphabet),

Rewrite-Systeme (Zustand = Term), Kommunikationsprotokolle und Interpreter von Programmiersprachen (Zustandsübergang = Ausführung einer Instruktion). Auch mathematische Sachverhalte wie der Ableitungsbegriff von logischen Kalkülen lassen sich als Zustandsübergangssystem darstellen.

Ein häufiger Spezialfall von Zustandsübergangssystemen sind sequentielle (auch deterministisch genannte) Systeme, bei denen es zu jedem Zustand st höchstens einen Zustand st' mit $(st,st') \in \rho$ gibt. Für den sequentiellen Fall läßt sich auf den Zuständen, die keine Endzustände sind $(S \setminus F)$, eine Zustandsübergangsfunktion τ durch $\tau(st) = st'$ gdw. $(st,st') \in \rho$ definieren.

Für ein Zustandsübergangssystem ist die Menge der möglichen Abläufe als die Menge aller endlichen $(st_0, \ldots, st_n)$ und unendlichen Folgen $(st_0, st_1, \ldots)$ von Zustandsübergängen mit $(st_i,st_{i+1}) \in \rho$ definiert, die mit einem Anfangszustand $st_0 \in I$ beginnen, und die, falls endlich, in einem Endzustand $st_n \in F$ enden.

2.2 Sequentielle ASMs

ASMs ([Gur95]) sind ein Formalismus zur Definition von Zustandsübergangssystemen. Die Menge der möglichen Zustände wird dabei durch die Menge $Alg(SIG)$ der Algebren einer (einsortigen) Signatur SIG gegeben. Um die Definition von booleschen Ausdrücken und von Partialität zu ermöglichen wird vorausgesetzt, daß die üblichen booleschen Operationen (tt,ff, $\wedge$, $\vee$, etc.) sowie die Konstante *undef* in der Signatur enthalten sind.

Die Menge der Startalgebren I wird in der Regel entweder durch die (mengentheoretische) Angabe von Algebren oder durch eine algebraische Spezifikation angegeben. Die Zustandsübergangsrelation wird durch eine Regel R gegeben. Für die in diesem Abschnitt betrachteten sequentiellen ASMs sind Regeln wie folgt induktiv definiert:

1. $f(\underline{t}) := t'$ ist eine Regel für ein n-stelliges Funktionssymbol f $(n \geq 0)$, und Grundterme $\underline{t}$ und t'. Die Regel ändert f an der Stelle $\underline{t}$ zu t' ab.

2. Wenn $R_1, \ldots, R_n$ Regeln sind, so auch die parallele Ausführung $(R_1, \ldots, R_n)$

3. Wenn $R_1, \ldots, R_n$ Regeln sind, und $\varepsilon_1, \ldots \varepsilon_n$, boolesche Ausdrücke, so auch die bedingte Regel
(**if** ε_1 **then** R_1 **else if** ε_2 **then** R_2 **else** $\ldots$ **if** ε_n **then** R_n)

Die Semantik einer Regel R ist eine Zustandsübergangsfunktion, die zu einer Algebra $\mathcal{A}$ eine neue Algebra $\mathcal{B}$ liefert. Die Definition von $\mathcal{B}$ erfolgt mit Hilfe einer Menge von Modifikationen ('Updates') $Upd(R, \mathcal{A}) = \{(f_1,\underline{a}_1,b_1), \ldots (f_n,\underline{a}_n,b_n)\}$, die aus der Regel R und der gegebenen Algebra $\mathcal{A}$ definiert werden. Jedes Update $(f, \underline{a}, b)$ besteht aus einem n-stelligen Funktionssymbol f, und Werten $\underline{a}, b \in A^{n+1}$ über der Trägermenge A (dem Universum) der Algebra $\mathcal{A}$. Entsprechend der Struktur der Regeln ist die Menge definiert durch

1. $Upd(f(\underline{t}) := t', \mathcal{A}) = \{(f, \underline{t}_{\mathcal{A}}, t'_{\mathcal{A}})\}$
2. $Upd((R_1, \ldots, R_n), \mathcal{A}) = Upd(R_1) \cup \ldots \cup Upd(R_n)$
3. $Upd(\textbf{if}\ \varepsilon_1\ \textbf{then}\ R_1\ \textbf{else}\ \ldots \textbf{else if}\ \varepsilon_n\ \textbf{then}\ R_n) = Upd(R_k)$, wobei k minimal mit $\mathcal{A} \models \varepsilon_k$ ist. Falls für alle $k = 1,\ldots,n$ $\mathcal{A} \not\models \varepsilon_k$ gilt, ist $Upd(\textbf{if} \ldots) = \emptyset$.

Die Menge $Upd(R, \mathcal{A})$ ist *inkonsistent*, falls sie für ein f und einen Vektor $\underline{a}$ mehrere Elemente $(f, \underline{a}, b)$ enthält. In diesem Fall ist die Zustandsübergangsfunktion die Identität, also $\tau(\mathcal{A}) = \mathcal{A}$. Ist $Upd(R, \mathcal{A}) = \emptyset$, so ist ein Endzustand erreicht[1]. Falls $Upd(R, \mathcal{A})$ konsistent und nichtleer ist, wird $\mathcal{B}$ definiert durch:

$$f_{\mathcal{B}}(\underline{a}) = \begin{cases} \mathrm{b} & \text{falls } (\mathrm{f},\underline{\mathrm{a}},\mathrm{b}) \in \mathrm{Upd}(\mathrm{R},\mathcal{A}) \\ \mathrm{f}_{\mathcal{A}}(\underline{\mathrm{a}}) & \text{sonst.} \end{cases}$$

Für jede ASM läßt sich die Menge der Operationen in zwei Teile aufteilen: In eine *dynamische* Hälfte, die auf der linken Seite von Regel-Updates vorkommen, und eine *statische* Hälfte, die während des Ablaufs einer ASM unverändert bleiben.

Die statische Hälfte wird benutzt, um Operationen auf Datenstrukturen zu modellieren (wie + auf natürlichen Zahlen, oder append auf Listen). Natürlich müssen auch die vordefinierten booleschen Operationen dazu zählen.

0-stellige dynamische Operationen (wir bezeichnen sie aus offensichtlichen Gründen nicht als Konstanten) werden dazu benutzt, „Programmvariablen“ zu modellieren. Dynamische Funktionen mit Argumenten werden häufig als Speicher benutzt. Anwendung einer dynamischen Funktion f auf eine Adresse a ergibt dann den Inhalt $f(a)$ des Speichers an der Adresse a. Funktionsmodifikation bedeutet das Überschreiben einer Speicherzelle. Eine dynamische Funktion mit endlichem Grundbereich G läßt sich auch als abstrakte Form eines Arrays mit Indexbereich G deuten.

Sorten werden in ASMs durch einstellige Prädikate modelliert. Um das einfache Hinzufügen eines neuen Elements zu einer Sorte S zu ermöglichen wird häufig folgende Erweiterung verwendet: Es wird eine Sorte *reserve* (d.h. ein einstelliges Prädikat) vordefiniert, das im Initialzustand jeder ASM unendlich viele („Reserve-“) Elemente enthalten muß. Eine neues Regelkonstrukt

import x **in** R **endimport**

erlaubt es, Elemente aus *reserve* an die Variable x zu binden, und R auszuführen. Das Addieren eines Elements zu einer Sorte S kann dann durch

import x **in** S(x) := tt; R **endimport**

erreicht werden, was durch

[1] [Gur95] definiert keine Endzustände für sequentielle ASMs. Wir ergänzen die Definition, da wir für den Verfeinerungsbegriff Endzustände benötigen.

extend S **with** x **in** R **endextend**

abgekürzt wird. Wir verzichten auf eine präzise Definition dieser Erweiterung, da sie einige Tücken birgt (in der Regel R kann nun die lokale Variable x verwendet werden, und geschachtelte import-Konstrukte müssen *sequentiell* neue Elemente liefern) und lediglich technischen Overhead verursacht. Eine ausführliche Definition gibt [Gur95].

2.3 Sequentielle ASMs in der WAM

Die in der Prolog-WAM-Fallstudie in [BR95] verwendeten ASMs verwenden eine Variante der Definition sequentieller ASMs. In dieser Variante dürfen Regeln nur die einfachere Form

$$\textbf{if } \varepsilon \textbf{ then } (\mathrm{f}_1(\underline{\mathrm{t}}_1) := \mathrm{t}_1', \mathrm{f}_2(\underline{\mathrm{t}}_2) := \mathrm{t}_2', \ldots, \mathrm{f}_n(\underline{\mathrm{t}}_n) := \mathrm{t}_n')$$

haben. Dafür enthält jede ASM eine Menge derartiger Regeln. Ein Zustandsübergang besteht nun in der indeterministischen Auswahl einer Regel, deren Test wahr ist, und ihrer Anwendung. Ein Endzustand ist erreicht, wenn keine Regel mehr anwendbar ist. Schließen sich die Regeltests gegenseitig aus, so ist die Regelmenge offenbar äquivalent zu einer geschachtelten if-then-else Regel des vorigen Abschnitts (wobei die Reihenfolge der Regeln beliebig ist). Für die Prolog-WAM-Fallstudie war der gegenseitige Ausschluß der Regeltests intendiert (für eine Stelle, wo die Intention nicht eingehalten wurde, siehe Abschnitt 12.2), so daß die Problematik des Indeterminismus nicht untersucht werden mußte.

2.4 Verteilte ASMs

Auch bei verteilten ASMs besteht der Grundgedanke in der Modifikation eines Zustands durch Regeln[2]. In einer verteilten ASM werden die möglichen Zustandsübergänge aber nicht durch *eine* Regel beschrieben, sondern durch eine endliche Menge A von (aktiven) Agenten, denen jeweils eine Regel aus einer endlichen Menge $\mathcal{R}$ von Regeln zugeordnet ist. Ein Zustandsübergang besteht dann in der indeterministischen Auswahl eines Agenten $a \in A$, und dem Ausführen der ihm zugeordneten Regel. Regeln in verteilten ASMs können sowohl die Menge der Agenten, als auch die Zuordnung der Regeln zu Agenten ändern.

Dazu enthält die Signatur einer verteilten ASM eine Menge N von *Regelnamen*, d.h. statischen Konstanten ν, die Regeln bezeichnen. Für einen Regelnamen ν ist R_ν die zugehörige Regel. Außerdem enthält die Signatur eine (dynamische) Funktion *Rule*, die Agenten auf Regelnamen abbildet. Die Menge

[2] Wir legen hier die Semantik der 'Sequential Runs' zugrunde. [Gur95] gibt noch weitere.

der Agenten wird dann implizit durch die Menge aller Elemente des Trägers gegeben, für die $Rule(a) \in N$ ist.

Als zulässige Zustände einer verteilten ASM sind nicht mehr alle Algebren zulässig, sondern nur noch solche, die die Regelnamen durch verschiedene Konstanten deuten und die eine endliche Menge von Agenten besitzen.

Bei der Definition von Regeln gibt es schließlich gegenüber der sequentiellen Definition noch die Erweiterung, daß diese das Symbol *Self* für den aktuell ausgewählten Agenten verwenden dürfen. Wird in einer Algebra $\mathcal{A}$ eine Regel R von einem Agenten a ausgeführt, so wird bei der Berechnung von $Upd(R, \mathcal{A})$ das Symbol *Self* durch a gedeutet. Dadurch können Regeln mit dem sie ausführenden Agenten parametrisiert werden. Führt ein Agent etwa die Zuweisung

Rule(Self) := undef

aus, so beendet er sich damit. Ein Endzustand einer verteilten ASM ist erreicht, sobald die Menge der Agenten leer ist.

Kapitel 3

Dynamische Logik und Algebraische Spezifikationen

3.1 Dynamische Logik

Dynamische Logik (DL) ist eine Erweiterung der Prädikatenlogik um Programmformeln der Form $\langle\alpha\rangle\ \varphi$ und $[\alpha]\ \varphi$. Dabei ist α ein imperatives Programm, und φ wieder eine Formel der Dynamischen Logik. Die Programme enthalten die in den meisten imperativen Programmiersprachen üblichen Konstrukte wie parallele Zuweisung $\underline{x} := \underline{t}$, sequentielle Komposition $\alpha;\beta$, Verzweigung **if** ε **then** α **else** β, while-Schleife **while** ε **do** α sowie Prozeduraufruf $p(\underline{t};\underline{x})$ mit value-Parametern $\underline{t}$ und var-Parametern $\underline{x}$ Aus theoretischen Gründen sind außerdem das leere Programm **skip**, das nie terminierende Programm **abort**, die i-fache Iteration eines Programms **loop** α **times** i, die Zufallszuweisung $x :=?$ sowie ein mit einer maximalen Rekursionstiefe i versehener Prozeduraufruf **procbound** i **in** $p(\underline{t};\underline{x})$ definiert.

Die Semantik von Programmen $[\![\alpha]\!]$ ist als zweistellige Relation zwischen Zuständen, d.h. Variablenbelegungen im prädikatenlogischen Sinn, definiert. Für ein deterministisches Programm ist die Relation eine partielle Funktion, d.h. es gibt zu jeder Variablenbelegung $\mathbf{z}$ höchstens ein $\mathbf{z}'$, so daß $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ gilt. Das einzige nichtdeterministische Konstrukt ist die Zufallszuweisung: $\mathbf{z}[\![x :=?]\!]\mathbf{z}'$ gilt für alle Zustände $\mathbf{z}' = \mathbf{z}[x \leftarrow a]$, die durch Modifikation der Belegung von x durch einen beliebigen Wert a entstehen.

Die Programmformel $\langle\alpha\rangle\ \varphi$ ist in einem Zustand $\mathbf{z}$ gültig, wenn es ein $\mathbf{z}'$ gibt, so daß $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ gilt, und φ in $\mathbf{z}'$ gilt. Dual dazu gilt $[\alpha]\ \varphi$ in einem Zustand $\mathbf{z}$ genau dann, wenn in jedem Zustand $\mathbf{z}'$ mit $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ die Formel φ gilt.

Die Programmformel $\langle\alpha\rangle\ \varphi$ besagt also, daß es einen terminierenden Ablauf von α gibt, so daß danach φ gilt. Die Gültigkeit von $[\alpha]\ \varphi$ bedeutet, daß nach

jedem terminierenden Ablauf von α hinterher φ gilt. Durch $\varphi \to [\alpha]\ \psi$ bzw. $\varphi \to \langle\alpha\rangle\ \psi$ lassen sich also partielle bzw. totale Korrektheit von Programmen (bezüglich Vorbedingung φ und Nachbedingung ψ) ausdrücken.

Syntax und Semantik von DL sind im Anhang B präzise definiert. Eine Besonderheit ist, daß es sich um eine mehrsortige Logik handelt, die nur Ausdrücke kennt, und nicht zwischen Termen und Formeln unterscheidet. Formeln werden mit Ausdrücken der Sorte bool identifiziert. Dies hat den Vorteil, daß die Logik durch die Einführung von λ-Ausdrücken leicht zu einer higher-order Logik erweiterbar ist. Ein technischer Vorteil ist, daß ein allgemeiner if–then–else Operator ($\varphi \supset t_1; t_2$) zur Verfügung steht (φ eine Formel, $t_1; t_2$ zwei beliebige Ausdrücke derselben Sorte). Dieser Ausdruck ist gleich t_1, falls φ wahr ist, und gleich t_2 sonst.

3.2 Algebraische Spezifikationen

Wir verwenden algebraische Spezifikationen mit den Strukturierungsoperationen Vereinigung (+), Anreicherung (enrich), Umbenennung (rename), Parametrisierung (generic) und Aktualisierung (actualize). Für frei erzeugte Datentypen benutzen wir Datentypdeklarationen (siehe z.B. die in Anhang E definierten Listen), für die automatisch geeignete Axiome generiert werden. Die Syntax sollte selbsterklärend sein, die Semantik ist auf die übliche Weise definiert. Sie stimmt z.B. im wesentlichen mit der Semantikdefinition der standardisierten algebraischen Spezifikationssprache CASL [CoF97] überein.

In Basisspezifikationen erlauben wir als Axiome nicht nur prädikatenlogische, sondern auch beliebige DL-Formeln, sowie Termerzeugtheitsprinzipien und Prozedurdeklarationen. Die Semantik einer Basisspezifikation ist die Menge aller Modelle dieser Axiome (lose Semantik). Eine präzise Definition findet sich am Ende von Anhang B.

3.3 KIV

KIV ist ein Werkzeug zur Entwicklung korrekter Software. Als Spezifikationssprache unterstützt KIV strukturierte, algebraische first-order Spezifikationen. Die Entwicklungsmethodik beruhte bisher auf der schrittweisen, modularen Verfeinerung dieser Spezifikationen durch Programmoduln, deren Korrektheit durch Beweisverpflichtungen in DL ausgedrückt werden kann. Eine umfassende Darstellung der Entwicklungsmethodik gibt [Rei95], die Verifikation von Programmmoduln wird in [RSS95] diskutiert. Die Deduktionsunterstützung in KIV beruht auf einem Sequenzenkalkül für Dynamische Logik. Einen Überblick über die in KIV vorhandene Unterstützung zur Deduktion über algebraischen Spezifikationen gibt [RSSB98].

3.4 Weiterentwicklung der Beweisstrategien

Im Rahmen dieser Arbeit wurden eine Reihe von Verbesserungen am KIV-System, insbesondere an der Beweiserkomponente implementiert. Diese haben wesentlich zur effizienten Bearbeitung von ASM-Verfeinerungen, insbesondere bei der Prolog-WAM-Fallstudie (siehe die Statistik in Abschnitt 19) beigetragen. Sie werden hier kurz stichwortartig zusammengefaßt:

- Erweiterung der Spezifikationssprache von strukturierten prädikatenlogischen, zu strukturierten DL-Spezifikationen mit globalen Prozedurdeklarationen (statt lokalen). Damit wird die Spezifikation von ASMs ermöglicht.
- Abschaffung der Unterscheidung von Termen und Formeln zugunsten von Formel = boolescher Term. Diese Änderung erlaubt es, boolesche dynamische Funktionen (dynamische Prädikate) wie alle anderen dynamischen Funktionen zu behandeln. Sie macht außerdem (unabhängig von dieser Arbeit) eine Erweiterung von DL um higher-order Funktionen (durch die Einführung von λ-Termen) leicht möglich.
- Erweiterung der Beweisstrategie für Programme auf parallele Zuweisungen. Parallele Zuweisungen waren bisher zwar in der Logik vorhanden, wurden aber vom Beweiser nicht unterstützt.
- Induktion über die Rekursionstiefe von Prozeduren. Diese vereinfacht das bisher vorhandene Beweisprinzip (Induktion über Umgebungen, [Ste85]) für die Verifikation rekursiver Prozeduren. Das neue Beweisprinzip spielt eine zentrale Rolle bei der Behandlung der *CHAIN#*-Prozedur in der Prolog-WAM-Fallstudie (siehe Abschnitt 15.2). Es vereinfacht außerdem die Semantikdefinition von DL, sowie den Vollständigkeitsbeweis.
- Erweiterung der Taktiken und Heuristiken für while-Schleifen und loop-Konstrukte, die eine zentrale Rolle in den Beweisverpflichtungen für die Korrektheit von ASM-Verfeinerungen spielen.
- Erweiterungen an verschiedenen anderen Heuristiken, so z.B. an den Heuristiken für das Unfolding von Prozeduren sowie der Quantoreninstanzierung.
- Effizientere Implementierung der Simplifikationsstrategie ([RSSB98]). Die jetzige Implementierung kommt z.B. mit 2000 Simplifikationsregeln, wie sie in der größten Spezifikation der Prolog-WAM-Fallstudie auftreten, zurecht.
- Verschiedene weitere Effizienzverbesserungen, die durch die reine Größe der zu behandelnden Beweisziele notwendig wurden. In einigen Fällen erreichten Sequenzen in der Prolog-WAM-Fallstudie die Größe von 5 Bildschirmseiten, und Beweisbäume hatten bis zu 1000 Knoten.

Kapitel 4

Formalisierung von ASMs in DL

In diesem Kapitel geben wir zunächst eine Übersetzung von ASMs in Algebraische Spezifikationen und Dynamische Logik (DL) an. Die Übersetzung wird im wesentlichen eins zu eins sein, da die Grundkonstrukte sowohl von ASMs als auch von Dynamischer Logik Zuweisungen sind. Da es keine Notwendigkeit gibt, die Semantik von ASMs zu formalisieren, also ASM-Regeln als Relationen über Zuständen zu kodieren, ist DL ein guter Startpunkt zur Verifikation von ASM-Eigenschaften. Die Übersetzung besteht aus drei Schritten. Im ersten Schritt (Abschnitt 4.1) wird gezeigt, daß sich die als Zustände der ASMs verwendeten Algebren in Variablenbelegungen über einer algebraischen Spezifikation transformieren lassen. Der zweite Schritt (Abschnitt 4.2) übersetzt die Regel einer ASM dann in ein imperatives Programm, wobei die im ersten Schritt erhaltenen Variablenbelegungen zu Zwischenzuständen des Programms werden.

Die Abschnitte 4.3 und 4.4 behandeln dann den dritten Schritt, die Übersetzung sequentieller bzw. verteilter ASMs in ein sequentielles Programm.

Das zentrale Beweisprinzip für Aussagen über ASMs ist Induktion über die Zahl der angewandten Regeln. Abschnitt 4.5 zeigt, wie dieses Beweisprinzip in DL formalisiert ist.

In Abschnitt 4.6 diskutieren wir schließlich Alternativen zu unserer Übersetzung.

4.1 Übersetzung der Spezifikationen

Um die abstrakten Datentypen einer ASM in algebraische Spezifikationen zu übersetzen, muß die Signatur zunächst in einen *dynamischen* und einen *statischen* Teil partitioniert werden. Der dynamische Teil der Signatur enthält die Sorten und Operationen, die durch Zuweisungen der ASM modifiziert werden. Der andere, statische Teil der Signatur enthält typischerweise Datentypen wie

Dynfun =
generic specification
parameter sorts dom, codom;
target sorts dynfun;
functions cf : codom → dynfun;
. [.] : dynfun × dom → codom;
. [. ← .] : dynfun × dom × codom → dynfun;
variables f : dynfun; x, y : dom; z : codom;
axioms cf(z) [x] = z,
f [x ← z] [x] = z,
x ≠ y → f [x ← z] [y] = f[y]
end generic specification

Abbildung 4.1 Spezifikation dynamischer Funktionen

Listen, Zahlen und passende Operationen. Für diesen Teil ist keine Übersetzung erforderlich; die Datentypen müssen lediglich geeignet spezifiziert werden.

Die Hauptidee für die Übersetzung des dynamischen Teils ist, die Semantik dynamischer Funktionen als Werte von (gewöhnlichen first-order) Variablen zu kodieren. Updates der ASM werden somit zu Zuweisungen in DL.

0-stellige Funktionen werden einfach in gewöhnliche first-order Variablen übersetzt. Der Fall einer Funktion mit mehreren Argumenten läßt sich durch die Einführung einer Tupelsorte auf den Fall mit einem Argument reduzieren. Für einstellige Funktionen muß der (second-order) Datentyp einer Funktion in einen first-order Datentyp codiert werden, damit eine dynamische Funktion der Wert einer Variable werden kann. Dies kann durch den in Abb. 4.1 gezeigten Datentyp erreicht werden, der algebraische Funktionen von einem Grundbereich *dom* (domain) in einen Zielbereich *codom* (codomain) spezifiziert:

Der Datentyp enthält eine konstante Funktion *cf*(z) für jedes Element z des Zielbereichs. Anwenden dieser Funktion auf ein beliebiges Element x des Grundbereichs ergibt immer z, wie dies im ersten Axiom ausgesagt wird. Die (zweistellige) Operation („*apply*“) zum Anwenden einer dynamischen Funktion f auf ein Element x wird dabei zur besseren Lesbarkeit statt als „$apply(f, x)$“ nur als $f[x]$ geschrieben (beachte: f ist in der algebraischen Codierung eine *Variable*, deren Wert eine Funktion ist). Mit einer geeigneten Konstanten des Zielbereichs werden konstante Funktionen typischerweise zur Initialisierung verwendet.

Eine Zuweisung $f(x) := t$ des ASM-Formalismus wird in der algebraischen Übersetzung zu einer Zuweisung $f := f[x \leftarrow t]$ an die Variable f. Für die an der Stelle x durch t modifizierte Funktion f verwenden wir der besseren Lesbarkeit wegen wieder statt „$modify(f, x, t)$“ die Mixfix-Notation $f[x \leftarrow t]$. Die letzten beiden Axiome beschreiben die Funktion.

Zu der Spezifikation dynamischer Funktionen sollte erwähnt werden, daß es im Gegensatz zur in KIV üblichen Vorgehensweise bei der Spezifikation nichtfreier Datentypen nicht notwendig ist, ein Extensionalitätsaxiom

$$\mathrm{f} = \mathrm{g} \leftrightarrow \forall\ \mathrm{x}.\ \mathrm{f[x]} = \mathrm{g[x]}$$

zu definieren. Ein derartiges Axiom würde es erlauben, (second-order) Gleichungen zwischen Funktionen wie $f = f[x \leftarrow f[x]]$ herzuleiten. Da derartige Gleichungen nicht Bestandteil des ASM-Formalismus sind, werden sie auch in der Übersetzung nicht benötigt. Aus demselben Grund kann auf die Definition eines Induktionsprinzips für dynamische Funktionen (z.B. strukturelle Induktion über *cf* und *modify*) verzichtet werden.

Es ist leicht zu sehen, daß die Menge der Funktionen von *dom* nach *codom* ein Modell der oben gegebenen Spezifikation bildet. Für dieses Modell ergibt sich die gewünschte 1:1 Korrespondenz zwischen der Belegung einer dynamischen Funktion und Belegungen der entsprechenden Variablen in der Übersetzung.

Die Grundform der Übersetzung ergibt eine algebraische Spezifikation, in der weder die Möglichkeiten zur Unterspezifikation noch die Existenz von Sorten (außer zur Einführung von Tupel- und Funktionssorten) genutzt werden. Dies läßt sich natürlich verbessern, indem die Sortenprädikate der ASMs, wo immer möglich, in echte Sorten der algebraischen Spezifikation übersetzt werden. Unterspezifikation kann, wie dies in algebraischen Spezifikationen üblich ist, dazu genutzt werden, um auf das explizite Fehlerelement *undef* zu verzichten.

Eine besondere Rolle spielt bei der Übersetzung noch das Prädikat *reserve* der ASMs, das ein (als unendlich groß angenommenes) Universum von „Vorratselementen“ definiert. Natürlich ist es möglich, das *reserve*-Prädikat wie alle anderen dynamischen Funktionen zu behandeln, und in eine Variable des Datentyps *Dynfun* mit Zielbereich *bool* zu übersetzen. Zur Modellierung des *import*-Konstrukts der ASMs ([Gur95], Abschnitt 3.2) ist dann lediglich eine Funktion *some* zu spezifizieren, die zu einer Belegung der Variablen *reserve*, ein Element x mit $reserve[x] = tt$ liefert. Üblicherweise werden die Elemente des *reserve*-Universums aber nur dazu benutzt, um sie dynamisch in die Trägermengen von Sorten einzufügen (z.B. um die wachsende Knotenmenge eines Suchbaums oder die allokierten Adressen eines Speichers zu modellieren). Für die für derartige Fälle vorgesehene Abkürzung

extend s **with** x **in** R **endextend**,

die ein Element x aus dem *reserve*-Universum entfernt, um es zur Trägermenge der Sorte s hinzufügen, gibt es eine einfachere Möglichkeit zur Übersetzung, die ganz auf das Reserve-Universum verzichtet. Dazu werden die in einem Zustand aktuell vorhandenen Elemente einer Sorte als Belegung einer Variable *se* der Sorte *set* (mit Elementen der Sorte s) gespeichert. Zur Spezifikation kann üblicherweise die in Abb. 4.2 gegebenen Spezifikation endlicher Mengen verwendet werden, da die in ASMs verwendeten Trägermengen in den meisten Fällen *endliche* Mengen sind (ist die initiale Trägermenge unendlich, muß zusätzlich eine entsprechende Konstante spezifiziert werden). Die in der Parameterspezifikation

S definierte Parametersorte s enthält jetzt die (unendlich vielen) potentiellen Elemente, die in die Trägermenge einer dynamischen Sorte aufgenommen werden können. Eine Variable *se* der Sorte *set* speichert die aktuelle Trägermenge der Sorte s. Die Funktion *new*(*se*) liefert ein neues Element der Sorte s. Der obige Sorten-Update kann somit durch

var x = new(se) **in begin** se := se $\cup$ {x}; R **end**

ausgedrückt werden.

Set =
generic specification
  parameter S;
  target
    sorts set;
    constants $\emptyset$: set;
    functions
      { . } : s $\rightarrow$ set;
      . $\cup$. : set $\times$ set $\rightarrow$ set;
      new : set $\rightarrow$ s;
    predicates
      . $\in$. : s $\times$ set;
    variables se, se_1, se_2 : set; x, y :s;
    axioms
      set **generated by** $\emptyset$, { . }, $\cup$;
      $\neg$ x $\in$ $\emptyset$, x $\in$ {y} $\leftrightarrow$ x = y,
      x $\in$ se_1 $\cup$ se_2 $\leftrightarrow$ x $\in$ se_1 $\vee$ x $\in$ se_2,
      se_1 = se_2 $\leftrightarrow$ ($\forall$ x. x $\in$ se_1 $\leftrightarrow$ x $\in$ se_2),
      $\neg$ new(se) $\in$ se
end generic specification

Abbildung 4.2 Algebraische Spezifikation von Mengen

4.2 Übersetzung von Regeln

In diesem Abschnitt wird die Übersetzung von ASM-Regeln in (flache) DL-Programme definiert. Um dieses zu vereinfachen, überlegt man zunächst leicht, daß es genügt, bedingte Regeln zu betrachten, deren Rümpfe Folgen von Update-Instruktionen sind:

if ε_1 **then** U_1 **else**
if ε_2 **then** U_2 **else**
$\vdots$
if ε_n **then** U_n

Wiederholte Anwendung der Transformation

(R, **if** ε **then** R′ **else** R″) ⇒ **if** ε **then** (R,R′) **else** (R,R″)

bringt jede Regel auf diese Form.

Das conditional muß nicht übersetzt werden[1], die Übersetzung einer einzelnen Update-Instruktion $f(t) := t'$ zu $f := f[t \leftarrow t']$ wurde schon im vorigen Abschnitt gegeben. Eine Folge von Updates wird in eine parallele Zuweisung übersetzt. Falls mehrere Updates auf dieselbe Funktion stattfinden, muß dabei die Möglichkeit von inkonsistenten Updates durch entsprechende Checks abgefangen werden. So wird $f(x) := t, f(x') := t'$ zu **if** $x = x' \wedge t \neq t'$ **then skip else** $f := f[x \leftarrow t][x' \leftarrow t']$ übersetzt. (wobei die Tests durch das Ausnutzen von Vorbedingungen häufig noch vereinfacht werden können oder ganz entfallen). Das Vorliegen eines inkonsistenten Updates führt dann entsprechend der ASM-Semantik zu keiner Zustandsänderung. Der einfacheren Lesbarkeit wegen schreiben wir im folgenden $f[x] := t$ statt $f := f[x \leftarrow t]$.

4.3 Übersetzung Sequentieller ASMs

Zur Vereinfachung der weiteren Darstellung nehmen wir ab diesem Abschnitt an, daß der Test, ob noch irgendeine Regel der ASM anwendbar ist, durch ein Prädikat *final* entschieden werden kann (*final* ist einfach die Konjunktion aller negierten Regeltests). Dann ergibt sich als Resultat der Übersetzung folgende Prozedur:

ASM(var $\underline{x}$)
begin
while ¬ final($\underline{x}$) **do** RULE(;$\underline{x}$)
end

Die zulässigen Anfangszustände der ASM werden durch die Vorgabe von geeigneten Anfangsbelegungen der Variablen $\underline{x}$ gegeben. Die Variablen $\underline{x}$ dienen sowohl zur Ein- als auch Ausgabe. Sie speichern die Belegungen der dynamischen Funktionen. Die iterative Anwendung von Regeln erfolgt in einer *while*-Schleife. *RULE* enthält den übersetzten Code der gegebenen ASM-Regel. Die separate Prozedur, die in ihrem Aufruf die Variablen $\underline{x}$ als Var-Parameter erhält (der Strichpunkt vor den Variablen besagt, daß keine Value-Parameter vorhanden sind) wurde zur Definition einer geeigneten Abkürzung verwendet.

[1] beachte, daß **if** ε_n **then** U_n in DL eine Abkürzung für **if** ε_n **then** U_n **else skip** ist

Die Äquivalenz des *while*-Programms zur Semantik-Definition der ASMs ergibt sich durch Betrachtung der Folge der Zustände, die das Programm jeweils zu Beginn der *while*-Schleife einnimmt. Die möglichen Zustandsfolgen sind (modulo der Übersetzung von Algebren zu Variablenbelegungen) genau dieselben wie in der ASM. Eine Restriktion der Ausdrucksmächtigkeit der DL ist lediglich, daß es keine Möglichkeit gibt, direkt über Zustands*folgen* und deren Eigenschaften zu reden. Dazu ist entweder die Einführung von Operatoren analog zu denen in der Temporallogik notwendig, oder die Einführung eines Datentyps für Folgen von Zuständen (sog. „Ströme“).

Für das zentrale Thema dieser Arbeit, ASM-Verfeinerungen, werden wir auf beides weitgehend verzichten können. Insbesondere werden Zustandsfolgen in den Beweisverpflichtungen nicht auftauchen. Für ASMs mit unbeschränktem Indeterminismus werden wir in Abschnitt 6.4 allerdings einen temporallogischen Operator AF benötigen, und die Definition von Trace-Korrektheit in Abschnitt 6.3 erfordert eine Formalisierung von Strömen als (dynamische) Funktionen von natürlichen Zahlen auf Zustände.

4.4 Übersetzung verteilter ASMs

Die wesentliche Problematik bei der Übersetzung verteilter ASMs besteht in der indeterministischen Auswahl eines Agenten a aus der endlichen Menge der Agenten A. Zwar kann die endliche Menge der Agenten mit Hilfe des Datentyps aus Abschnitt 4.1 beschrieben werden, eine zusätzliche Funktion *some* eignet sich aber *nicht* zur Auswahl eines Elements aus dieser Menge, da sie zu einer endlichen Menge s *immer dasselbe* Element $some(s)$ liefert. Eine Lösung ist in DL aber dennoch einfach: Man verwendet eine Prozedur *SOME*, die die aktuelle Menge A der aktiven Agenten als Eingabe erhält, und als Ausgabe den Agenten *Self* liefert, der die Aktion ausführen soll. *Self* ist dabei eine Programmvariable. Für die Prozedur *SOME* werden lediglich die Axiome

$$\mathrm{a} \in \mathrm{A} \rightarrow \langle \mathrm{SOME(A;Self)} \rangle\ \mathrm{Self} = \mathrm{a} \tag{4.1}$$

und

$$[\mathrm{SOME(A;Self)}]\ \mathrm{Self} \in \mathrm{A}$$

gegeben. Sie besagen, daß die Ein-/Ausgaberelation von *SOME* in allen Modellen der Spezifikation identisch zur Elementbeziehung ist (das erste Axiom sagt Obermenge, das zweite Teilmenge). Bei jeder Ausführung der Prozedur *SOME* ist also jede Auswahl eines Agenten möglich, wie dies die ASM-Semantik vorsieht. Eine Implementierung der *SOME*-Prozedur wäre ein konkreter Scheduler für die Agentenauswahl. Allerdings wird eine solche Implementierung meist nicht mehr in jedem Schritt jede Auswahl zulassen, und evtl. auch von anderen Zustandskomponenten als A der ASM abhängen. Deshalb ist es dann sinnvoll,

SOME mit dem kompletten Zustand $\underline{x}$ der ASM zu parametrisieren, und das Axiom (4.1) durch das schwächere Totalitätsaxiom

$$A \neq \emptyset \rightarrow \langle \text{SOME}(\underline{x};\text{Self}) \rangle \text{ true}$$

zu ersetzen. Damit wird dann nur noch verlangt, daß die Ein-Ausgabe-Relation eine (für nichtleere Agentenmengen) totale Teilrelation der Elementbeziehung ist, und es wird möglich, die beiden bei einer ASM-Verfeinerung (s. Kapitel 5) beteiligten Scheduler in Beziehung setzen (etwa dadurch, daß gefordert wird, daß jede Wahl des konkreten Schedulers auch für den abstrakten möglich sein muß). Zu beachten ist auch, daß Restriktionen wie Fairness-Constraints es evtl. notwendig machen, über die Folge der ausgewählten *Self*-Werte insgesamt Aussagen zu machen. Hierzu sind entweder Erweiterungen der Dynamischen Logik oder die explizite Verwendung des Stroms der *Self*-Werte notwendig (vgl. auch die in [Vog97] beschriebene Übersetzung von Linearer Temporallogik (LTL) in Dynamische Logik).

Mit Hilfe der *SOME*-Prozedur wird die verteilte ASM zu

ASM(var $\underline{x}$)
begin while $A \neq \emptyset$ **do**
begin
SOME($\underline{x}$;Self);
if Rule(Self) $= \nu_1$ **then** $\text{RULE}_1(;\underline{x})$ **else**
if Rule(Self) $= \nu_2$ **then** $\text{RULE}_2(;\underline{x})$ **else**
$\vdots$
if Rule(Self) $= \nu_n$ **then** RULE_n $(;\underline{x})$
end
end

übersetzt, wobei die einzelnen Regeln $RULE_1$, $RULE_2$, ..., $RULE_n$ wie bei sequentiellen ASMs übersetzt werden. Es ist zu beachten, daß sowohl der aktive Agent *Self*, die Funktion *Rule*, die Agenten Regeln zuordnet, als auch die Agentenmenge A Bestandteil des Zustandsvektors $\underline{x}$ sind, der wie im sequentiellen Fall alle (zu Variablen übersetzten) dynamischen Funktionen der ASM enthält. Die Regelnamen sind als Aufzählungstyp der Konstanten $\nu_1, \ldots \nu_n$ spezifiziert.

Wie im sequentiellen Fall stimmen die möglichen Zustandsfolgen der *while*-Schleife modulo der Übersetzung der Algebren zu Variablenbelegungen mit den möglichen Abläufen der ASM überein. Um für das folgende eine einheitliche Schreibweise zu den sequentiellen ASMs zu erhalten, schreiben wir auch für verteilte ASMs statt des Schleifenrumpfs $RULE(;\underline{x})$ und verwenden das allgemeine *final*($\underline{x}$)-Prädikat statt des speziellen $A \neq \emptyset$.

4.5 Regelinduktion in DL

Das zentrale Beweisprinzip für Beweise über ASMs, das wir im folgenden benötigen, ist Induktion über die Zahl der angewandten Regeln. In diesem Abschnitt geben wir das formale Pendant für die DL-Formalisierung von ASMs, Induktion über die Anzahl von Schleifendurchläufen, an. Induktion über die Zahl der Durchläufe einer *while*-Schleife ist durch das Omega-Axiom der Dynamischen Logik möglich:

$$\langle \textbf{while } \varepsilon \textbf{ do } \alpha \rangle \ \varphi \leftrightarrow \exists \ \mathrm{i}. \ \langle \textbf{loop if } \varepsilon \textbf{ then } \alpha \textbf{ times } \mathrm{i} \rangle \ (\varphi \wedge \neg \ \varepsilon) \tag{4.2}$$

In diesem Axiom ist i eine natürliche Zahl (über die induziert werden kann), die die Zahl der Schleifendurchläufe zählt. Das loop-Programm **loop** α **times** i bedeutet i-maliges Ausführen des Programms α. Das Axiom (4.2) besagt also, daß eine Formel φ nach der Ausführung einer *while*-Schleife genau dann gilt, wenn es eine genügend groß gewählte Anzahl von Durchläufen durch **if** ε **then** α gibt, so daß anschließend φ gilt, und der Schleifentest ε falsch wird. Man beachte, daß für eine feste Eingabe die Zahl der Iterationen, die für die quantifizierte Variable i gewählt wird, nicht unbedingt die *exakte* Anzahl der Iterationen sein muß, die die *while*-Schleife benötigt. Jede größere Zahl ist ebenfalls zulässig, da das Ausführen von **if** ε **then** α bei falschem ε keinen Effekt hat (durch Ersetzen von **if** ε **then** α durch **if** ε **then** α **else abort** im loop-Konstrukt erhält man eine restriktivere Variante des Axioms, in der die einzig korrekte Instanz für i die exakte Anzahl der Schleifendurchläufe ist).

Das loop-Konstrukt wird in DL rekursiv durch die beiden folgenden Axiome definiert:

$$\begin{array}{l} \langle \textbf{loop } \alpha \textbf{ times } 0 \rangle \ \varphi \leftrightarrow \varphi \\ \langle \textbf{loop } \alpha \textbf{ times } \mathrm{i} +1 \rangle \ \varphi \leftrightarrow \langle \textbf{loop } \alpha \textbf{ times } \mathrm{i} \rangle \ \langle \alpha \rangle \ \varphi \end{array} \tag{4.3}$$

4.6 Alternativen zur Formalisierung

Zur Übersetzung der ASMs in DL sind mehrere Alternativen möglich:

1. Einbettung in eine Higher-Order Variante der Dynamischen Logik

2. Definition einer ASM-Logik: Eine derartige Logik muß die Modifikation von Algebren durch Programme unterstützen. Ein geeigneter Kandidat ist MLCM (Modal Logic of Creation and Modification [GdL94],[GR95]). [Sch95] zeigt einen Versuch, eine Variante von MLCM im KIV-System zu realisieren.

3. Eine weitere Möglichkeit ist es, statt ASMs zu formalisieren, deren Semantik zu formalisieren, also Zustandsübergangssysteme. Dies ist durch Einbettung in die Prädikatenlogik möglich, und wurde für die WAM-Fallstudie in Isabelle [Pus96] durchgeführt (in Isabelle wird Higher-Order Logik verwendet, was aber nicht unbedingt erforderlich ist). An die Stelle von ASM-Regeln tritt dann die explizite Beschreibung einer Zustandsübergangsrelation, sowie die induktive Definition der Relation zwischen Anfangs- und Endzuständen der ASM.

4. Einbettung in eine Temporallogik

Die erste Lösung ist eine Variante unserer Lösung, in der statt eines speziellen Datentyps 'dynamische Funktion' gewöhnliche higher-order Funktionen verwendet werden. Diese Lösung erfordert die Erweiterung von DL um higher-order Ausdrücke (eine derartige Erweiterung ist für KIV angedacht). Sie hätte den Vorteil, daß an die Stelle der speziellen *apply*-Operation für dynamische Funktionen die gewöhnliche Funktionsapplikation treten könnte. Ein Argument für die jetzige Lösung ist, daß sie dynamische Funktionen, die als globale Register dienen und durch Zuweisungen destruktiv überschrieben werden können, und beliebige higher-order Funktionen, die im allgemeinen nicht destruktiv modifiziert werden dürfen, nicht vermischt. Die Trennung könnte also eine effiziente Implementierung der dynamischen Funktionen erleichtern.

Auch die zweite Lösung ist der unseren sehr ähnlich. Sie hat aus unserer Sicht den Nachteil, daß die Definition einer neuen Logik einen deutlich höheren Aufwand hat: Zusätzlich sind neben der Implementierung der Kalkülregeln sowohl die Definition der Semantik der Logik, als auch Korrektheits- und evtl. Vollständigkeitsbeweise für die Regeln der Logik notwendig. Außerdem ist zu beachten, daß es der Korrektheitsnachweis für ASM-Verfeinerungen notwendig machen kann, über dynamische Funktionen zu quantifizieren (ein Beispiel zeigt Abschnitt 11.2), was in MLCM nicht möglich ist.

Die dritte Lösung unterscheidet sich stärker von der unsrigen, da für sie zunächst eine grundlegende Theorie induktiver Relationen (oder eine noch allgemeinere Fixpunkttheorie wie sie in PVS formalisiert wurde, [BDvH+96]) entwickelt werden muß, damit Induktion über die Anzahl angewandter Regeln möglich wird. Eine derartige Theorie wurde z.B. in Isabelle ([Pau94]) definiert. In unserem Ansatz ist die notwendige Fixpunktteorie bereits in der Definition der Logik durch die Axiomatisierung der *while*-Schleifen-Semantik (vergleiche den vorigen Abschnitt 4.5) gegeben. Die Lösung hat für die konkrete Anwendung den Nachteil, daß bei jedem Zustandsübergang der gesamte Zustand betrachtet werden muß („Frame-Problem"). An die Stelle einer Zuweisung an eine Einzelkomponente

$$x_i := f(\underline{y})$$

tritt eine (infix geschriebene) Relation $\Rightarrow$:

$$(x_1, \ldots, x_i, \ldots x_n) \Rightarrow (x_1, \ldots, f(\underline{y}), \ldots x_n)$$

in der der komplette Zustand $(x_1, \ldots, x_n)$ betrachtet werden muß, was notationellen Overhead bedeutet. Außerdem ändern sich bei Addition einer neuen Komponente zum Zustand alle bisherigen Beweise, auch wenn sie die neue Komponente gar nicht involvieren.

Für die generische Definition und den Nachweis des Modularisierungstheorems für ASM-Verfeinerungen, das wir in Kapitel 6 betrachten werden, ist das Frame-Problem natürlich unerheblich, da die Zustände hierbei nur als unspezifizierte, monolithische Parametersorte betrachtet werden. Wir werden deshalb in Abschnitt 6.2.5 nochmals auf die prädikatenlogische Formalisierung eingehen.

Ein Vorteil der Definition induktiver Relationen gegenüber ASMs ist (genau wie bei DL-Programmen) die Möglichkeit beliebiger Rekursion. Für ASMs sind hierfür Erweiterungen des Formalismus notwendig (siehe [GS97]).

Die vierte Lösung schließlich, die Einbettung in eine Temporallogik (z.B. CTL*) ist eine gute Alternative, wenn Eigenschaften einzelner ASMs betrachtet werden sollen. Beziehungen zwischen ASMs (wie etwa Refinement) erfordern aber die simultane Betrachtung mehrerer Zustandsübergangsrelationen, was eine Codierung erschwert (oder eine multimodale Temporallogik erfordert).

Schließlich sollte noch erwähnt werden, daß es zur Transformation der Regeln auf die in Abschnitt 4.1 gezeigte Normalform noch als Alternative die Einführung eines allgemeinen Operators zur Parallelausführung von Regeln gibt. Die Transformation der Regeln in die Normalform kann dann durch Regeln *in* der Logik beschrieben werden. [Sch95] zeigt, wie diese Alternative für MLCM realisiert werden kann. Wir verzichten gegenwärtig darauf, da wir derzeit keinen Weg sehen, die Transformation zu vermeiden (die Inkonsistenz einer Regel läßt sich nur in der Normalform leicht erkennen) und eine Realisierung durch einen Präprozessor gegenüber einer Realisierung durch logische Regeln effizienter scheint.

Kapitel 5

Verfeinerung von ASMs und Formalisierung in DL

Eine Verfeinerung einer ASM $= (S, I, \rho)$ zu einer ASM$' = (S', I', \rho')$ ist gegeben durch eine Relation $IN : I \times I'$ auf den Anfangszuständen und eine Relation $OUT : F \times F'$ auf den Endzuständen F und F'. Häufig werden Spezialfälle betrachtet, in denen Funktionen statt der allgemeinen Relationen IN, OUT gegeben sind.

Definition 1 *Korrektheit und Vollständigkeit von Verfeinerungen*
Eine Verfeinerung von ASM zu ASM$'$ heißt *korrekt*, falls zu jedem endlichen Ablauf $(st'_0, \ldots, st'_n)$ von ASM$'$ (mit $st'_n \in F'$) und jedem Anfangszustand st_0 von ASM, für den $IN(st_0, st'_0)$ gilt, ein endlicher Ablauf $(st_0, \ldots, st_m)$ mit $st_m \in F$ existiert, so daß $OUT(st_m, st'_n)$ gilt. Wir schreiben kurz ASM $\triangleright$ ASM$'$ für eine korrekte Verfeinerung Eine Verfeinerung von ASM zu ASM$'$ heißt *vollständig*, kurz ASM $\triangleleft$ ASM$'$, falls die Verfeinerung von ASM$'$ zu ASM korrekt ist. Wir schreiben ASM $\bowtie$ ASM$'$ für eine korrekte und vollständige Verfeinerung.

Die Korrektheit und Vollständigkeit einer Verfeinerung wird häufig durch die Kommutierung des in Abb. 5.1 Diagramms ausgedrückt:

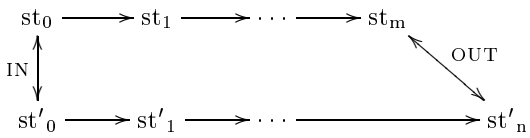


Abbildung 5.1 : diagrammatische Darstellung einer ASM-Verfeinerung

Korrektheit und Vollständigkeit lassen sich relativ zu einer speziellen Algebra definieren, oder relativ zu allen Modellen der gemeinsamen Spezifikation beider ASMs. Die Beweisverpflichtungen, die wir im folgenden Kapitel herleiten

werden, implizieren in jedem einzelnen Modell der gemeinsamen Spezifikation die Korrektheit der Verfeinerung (es gilt nicht nur, daß die Gültigkeit der Beweisverpflichtungen in allen Modellen die Korrektheit impliziert) , insofern spielt die Unterscheidung im folgenden keine Rolle.

Für die Begriffe 'korrekt' und 'vollständig', die der Terminologie der ASMs ([BR95]) entnommen sind, werden in der Literatur verschiedene andere Begriffe verwendet. Im Verifix-Projekt ([GDG+96]) werden z.B. statt „Korrektheit" und „Vollständigkeit" die Begriffe „Vererbung partieller Korrektheit" und „Vererbung totaler Korrektheit" gebraucht. Eine korrekte und vollständige Verfeinerung heißt in der Literatur oft auch Bisimulation, in Fallstudien mit dem NQTHM-System ([BHMY89]) ist der Begriff „Interpreteräquivalenz" (interpreter equivalence) gebräuchlich.

Unser Korrektheitsbegriff vergleicht das Ein-/Ausgabeverhalten und ist daher für ASMs angemessen, die die „Berechnung eines Ergebnisses" beschreiben. Beschreibt eine ASM einen reaktives System, ist ein Korrektheitsbegriff angemessen, der auf einem Vergleich der Abläufe der ASMs beruht. Wir werden einen derartigen Begriff („Trace-Korrektheit") erst in Abschnitt 6.3 definieren, und zeigen, daß sich die Beweisverpflichtungen nur minimal unterscheiden.

5.1 Compilerverifikation

Ein typisches Beispiel für die Anwendung von Verfeinerungen ist die Übersetzung von Programmiersprachen. Dazu werden zwei ASMs betrachtet, die je einen Interpreter für die Quell- und die Zielsprache der Übersetzung darstellen. Startzustände enthalten jeweils das auszuführende Programm. Die *IN*-Relation zwischen den Startzuständen wird durch die Funktion *compile* zum compilieren von Programmen wiedergegeben:

$$\mathrm{IN}(\mathrm{st},\mathrm{st}') \leftrightarrow \mathrm{program}'(\mathrm{st}') = \mathrm{compile}(\mathrm{program}(\mathrm{st})) \wedge \mathrm{I}(\mathrm{st}) \wedge \mathrm{I}'(\mathrm{st}').$$

Üblicherweise sind die Initialzustände durch das Programm allein eindeutig festgelegt. Schwächer wird oft auch gefordert, daß es zu jedem initialen ASM′-Zustand st' einen initialen ASM-Zustand st mit $IN(st,st')$ geben sollte.

Die Ausgaberelation ist meist, daß sich das (abstrakte) Ergebnis des Quellspracheninterpreters mit Hilfe einer Abstraktionsfunktion aus seiner konkreten Repräsentation als Ergebnis des Zielspracheninterpreters zurückgewinnen läßt:

$$\mathrm{OUT}(\mathrm{st},\mathrm{st}') \leftrightarrow \mathrm{result}(\mathrm{st}) = \mathrm{abstract}(\mathrm{result}'(\mathrm{st}')).$$

5.2 Formalisierung der Korrektheit in DL

Als Formalisierung der Korrektheit der Verfeinerung von ASM zur ASM′ ergibt sich

$$\begin{aligned}&\mathrm{ASM} \triangleright \mathrm{ASM}' \equiv \\ &\mathrm{IN}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \wedge \langle \mathrm{ASM}'(;\underline{\mathrm{x}}')\rangle \underline{\mathrm{x}}' = \underline{\mathrm{x}}'_0 \rightarrow \langle \mathrm{ASM}(;\underline{\mathrm{x}})\rangle\ \mathrm{OUT}(\underline{\mathrm{x}},\underline{\mathrm{x}}'_0)\end{aligned} \tag{5.1}$$

In der Formel sind $\underline{x}$ und $\underline{x}'$ zwei disjunkte Vektoren von Variablen, die sich aus der Übersetzung der dynamischen Funktionen der beiden ASMs ergeben. Die Formel besagt, daß $IN(\underline{x}, \underline{x}')$ und die Existenz eines terminierenden Ablaufs von ASM′ mit Ergebnis $\underline{x}'_0$ die Existenz eines terminierende Ablaufs von ASM implizieren muß, so daß für $\underline{x}'_0$ und das Ergebnis dieses Ablaufs die Relation OUT gilt (man beachte, daß das $\underline{x}$ in $IN(\underline{x}, \underline{x}')$ einen beliebigen Initialwert der Variablen beschreibt, während das $\underline{x}$ in $OUT(\underline{x}, \underline{x}'_0)$ die Belegung von $\underline{x}$ *nach* Ablauf von ASM wiedergibt.).

Für die Formalisierung der Vollständigkeit werden einfach die Rollen von ASM und ASM′ vertauscht:

$$\begin{aligned}&\mathrm{ASM} \triangleleft \mathrm{ASM}' \equiv \\ &\mathrm{IN}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \wedge \langle \mathrm{ASM}(;\underline{\mathrm{x}})\rangle \underline{\mathrm{x}} = \underline{\mathrm{x}}_0 \rightarrow \langle \mathrm{ASM}'(;\underline{\mathrm{x}}')\rangle\ \mathrm{OUT}(\underline{\mathrm{x}}_0,\underline{\mathrm{x}}')\end{aligned} \tag{5.2}$$

Die Äquivalenz von ASM und ASM′ ergibt sich dann als Konjunktion von (5.1) und (5.2). Sind die Zustandsvektoren beider ASMs typgleich, und ist $OUT(\underline{x}, \underline{x}')$ als $\underline{x} = \underline{x}'$ definiert, so läßt sich dies zu der Programmäquivalenz

$$\begin{aligned}&\mathrm{ASM} \bowtie \mathrm{ASM}' \equiv \\ &\mathrm{IN}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \rightarrow (\langle \mathrm{ASM}(;\underline{\mathrm{x}})\rangle \underline{\mathrm{x}} = \underline{\mathrm{x}}_0 \leftrightarrow \langle \mathrm{ASM}'(;\underline{\mathrm{x}}')\rangle \underline{\mathrm{x}}' = \underline{\mathrm{x}}_0)\end{aligned}$$

vereinfachen.

Kapitel 6

Eine generische Beweismethode für ASM-Verfeinerungen

Dieses Kapitel bildet den Kern der theoretischen Arbeit. Es wird gezeigt, wie sich Korrektheits- und Vollständigkeitsbeweise für ein Refinement von ASM zu ASM′ modularisieren lassen. Die Beweisverpflichtungen, die die Korrektheit der Modularisierung sichern, wurden in Dynamischer Logik formalisiert, und mit Hilfe von KIV verifiziert.

Die ersten beiden Abschnitte betrachten zunächst sequentielle, deterministische ASMs. Abschnitt 6.1 behandelt als Einführung zunächst den aus der Literatur unter dem Stichwort „Data Refinement“ bekannten Spezialfall: In diesem entspricht eine Regelanwendung von ASM einer Regelanwendung von ASM′, und es ist eine Abstraktionsfunktion gegeben, die Zustände von ASM′ auf Zustände von ASM abbildet.

Abschnitt 6.2 betrachtet den allgemeinen Fall, in dem der Zusammenhang zwischen den Zuständen durch eine beliebige Relation, die wir *Zusammenhangsinvariante* nennen, gegeben ist. Die Einschränkung, daß *eine* Regelanwendung von ASM einer Regelanwendung von ASM′ entsprechen muß, wird aufgegeben. Stattdessen wird nur noch gefordert, daß sich das Ablaufdiagramm aus Abb. 5.1 in Teildiagramme zerlegen läßt, so daß die Zusammenhangsinvariante an den Zerlegungspunkten gilt. Das Hauptergebnis dieses Abschnitts ist dann der Satz, daß sich der Äquivalenzbeweis des Gesamtdiagramms unter dieser Voraussetzung in Äquivalenzbeweise für die Teildiagramme zerlegen läßt. Es wird gezeigt, daß je Teildiagramm nur eine Beweisverpflichtung erforderlich ist, die sowohl für die Korrektheit als auch für die Vollständigkeit hinreicht.

Abschnitt 6.3 behandelt eine Alternative zum in Kapitel 5 definierten Korrektheitsbegriff. Dieser beruht nicht auf korrektem Ein-/Ausgabeverhalten, sondern macht Aussagen über die Abläufe (Traces) der ASMs. Trace-Korrektheit ist etwas stärker als Korrektheit. Für deterministische ASMs implizieren Kor-

rektheit und Vollständigkeit auch Trace-Korrektheit. Wir geben ein Beispiel, das zeigt, daß dies für indeterministische ASMs nicht mehr so ist. Deshalb definieren wir, bevor wir indeterministische ASMs betrachten, Trace-Korrektheit formal, wobei wir die in der Literatur gebräuchlichen Abstraktionsfunktionen wieder zu beliebigen Zusammenhangsinvarianten verallgemeinern. Wir zeigen, daß sich die Beweisverpflichtungen für Korrektheit und Trace-Korrektheit nur minimal unterscheiden.

Abschnitt 6.4 behandelt indeterministische ASMs. Wir zeigen, welche Modifikationen an den Beweisverpflichtungen notwendig sind, damit das Modularisierungstheorem auch auf indeterministische ASMs anwendbar ist. Im wesentlichen ergeben sich nun für jedes Teildiagramm zwei separate Beweisverpflichtungen für die Korrektheit und Vollständigkeit. Außerdem muß die Komplikation beachtet werden, daß die Größe der Teildiagramme von indeterministisch gewählten Regeln abhängen kann.

Abschnitt 6.5 behandelt Optimierungen des Verfahrens, die bei iterativer Verfeinerung einer ASM zu ASM′ und dann zu ASM″ möglich sind.

Abschnitt 6.6 gibt schließlich einige verwandte Arbeiten. Korrektheit im dem Sinne, daß die gleichen Ausgaben während eines Ablaufs gemacht werden („Behavioral Correctness“), wird als Spezialfall von Trace-Correctness identifiziert.

6.1 Data Refinement

6.1.1 Definition

Der einfachste Fall der Verfeinerung einer sequentiellen ASM ist „Data Refinement“ ([Hoa72]). Die Idee ist, von einer „abstrakten“ Zustandsmenge S in ASM zu einer „konkreten“ Zustandsmenge S' in ASM′ überzugehen (diese Idee bildet auch die Grundlage vieler rein algebraischer Verfeinerungsbegriffe). Wenn also ein Zustand in S etwa aus einer Menge von Elementen besteht, so könnte diese in S' z.B. durch eine Liste von Elementen repräsentiert werden. Der Zusammenhang wird dabei meist durch eine Abstraktionsfunktion

$$\text{abstr} : \text{S}' \to \text{S}$$

gegeben, die konkreten Zuständen abstrakte zuordnet. Die Funktion kann partiell sein, da nicht jeder mögliche konkrete Zustand einen abstrakten repräsentieren muß (z.B. könnten nur duplikatfreie Listen als Repräsentationen von Mengen dienen). Sie braucht auch nicht injektiv zu sein, da mehrere konkrete Zustände denselben abstrakten Zustand repräsentieren können (im Beispiel repräsentieren [1,2] und [2,1] dieselbe Menge). Die Zustandsübergangsfunktion τ' von ASM′ ist bei dieser Art der Verfeinerung so zu wählen, daß sie denselben Effekt auf den konkreten Datenstrukturen erzielt, wie τ von ASM auf den abstrakten Zuständen. Dies läßt sich durch

$$\begin{aligned} &\text{abstr}(\underline{x}') = \underline{x} \wedge \neg\ \text{final}(\underline{x}) \wedge \neg\ \text{final}'(\underline{x}') \\ &\rightarrow \langle \text{RULE}(;\underline{x}) \rangle\ \langle \text{RULE}'(;\underline{x}') \rangle\ \text{abstr}(\underline{x}') = \underline{x} \end{aligned} \tag{6.1}$$

in DL formalisieren ($\underline{x}$ und $\underline{x}'$ sind wieder zwei disjunkte Vektoren von Variablen, die sich aus der Übersetzung der dynamischen Funktionen der beiden ASMs ergeben). Informell wird die Äquivalenz durch die Kommutativität des Diagramms in Abb. 6.1

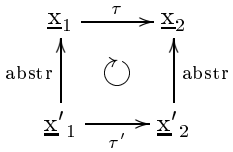


Abbildung 6.1 : kommutierendes 1:1-Diagramm

wiedergegeben. Eine Regelanwendung von ASM ist also äquivalent zu einer Regelanwendung von ASM′, die beiden Systeme arbeiten „gleichlaufend". Daß (6.1) im wesentlichen ausreicht, um die Äquivalenz von ASM und ASM′ zu zeigen, wird durch Induktion über die Anzahl der ausgeführten Regelschritte gezeigt. Informell bedeutet dies, daß kommutierende Diagramme wie in Diagramm 6.2 zusammengesetzt werden:

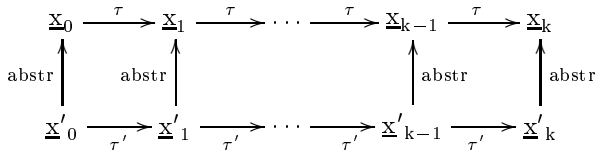


Abbildung 6.2 : kommutierende 1:1-Diagramme

Für den Induktionsanfang wird benötigt, daß Initialzustände durch die Abstraktionsfunktion verbunden sind:

$$\text{IN}(\underline{x},\underline{x}') \rightarrow \text{abstr}(\underline{x}') = \underline{x} \tag{6.2}$$

Dies wird in der Regel schon dadurch gewährleistet, daß $IN(\underline{x}, \underline{x}')$ einfach als $abstr(\underline{x}') = \underline{x}$ definiert wird. Schließlich wird benötigt, daß für zwei Endzustände dieselbe Ausgabe vorliegt

$$\text{abstr}(\underline{x}') = \underline{x} \wedge \text{final}'(\underline{x}') \wedge \text{final}(\underline{x}) \rightarrow \text{OUT}(\underline{x},\underline{x}') \tag{6.3}$$

und beide ASMs nur gleichzeitig einen Endzustand erreichen können:

$$\text{abstr}(\underline{x}') = \underline{x} \rightarrow (\text{final}(\underline{x}) \leftrightarrow \text{final}'(\underline{x}')) \tag{6.4}$$

gilt. Zusammengenommen gilt das Theorem

Theorem 1 *Korrektheit und Vollständigkeit für Data Refinement*
Aus der Gültigkeit von (6.1), (6.2), (6.3) und (6.4) folgt die Korrektheit und Vollständigkeit der Verfeinerung von ASM zu ASM′:

$$(6.1) \wedge (6.2) \wedge (6.3) \wedge (6.4) \Rightarrow \text{ASM} \bowtie \text{ASM}'$$

6.2 Das Modularisierungstheorem

6.2.1 Informelle Beschreibung

In diesem Abschnitt geben wir ein generisches Theorem zur Modularisierung von Äquivalenznachweisen für Verfeinerungen von sequentiellen ASMs. Wir geben zunächst einen informellen Korrektheitsbeweis. Anschließend skizzieren wir dessen Formalisierung in KIV. Schließlich geben wir auch noch einen formalen Beweis in KIV für eine prädikatenlogische Formalisierung der ASMs. Damit ist sichergestellt, daß das Theorem im wesentlichen unabhängig von der Formalisierung der ASMs ist.

Die grundsätzliche Idee des Theorems läßt sich am einfachsten anhand des kommutierenden Diagramms zeigen, das die Äquivalenz zweier ASMs beschreibt. Um diesen Beweis zu modularisieren, zerlegen wir das Diagramm in Teildiagramme, wie dies in Abb. 6.3 gezeigt ist. Zustände an den Kanten der Teildiagramme sind durch eine (beliebige!) Relation *INV*, die wir als *Zusammenhangsinvariante* (coupling invariant) bezeichnen, verbunden. Die Grundannahme, die einer Modularisierung dieser Art zugrunde liegt ist, daß sich die Korrespondenz der Gesamtberechnung, die beide ASMs durchführen, sich auf die Korrespondenz von Teilberechnungen reduzieren läßt, die in beiden ASMs in derselben Reihenfolge vorkommen. Korrespondierende „ähnliche“ Zustände werden durch die Gültigkeit der Zusammenhangsinvariante charakterisiert. Daraus ergibt sich automatisch auch schon die Zerlegung in Teildiagramme (man zeichne einfach korrespondierende Zustände ein). Die Angabe korrespondierende Regelfolgen ist zwar, um die Zerlegung inhaltlich zu verstehen, hilfreich, und wir werden sie in der Prolog-WAM-Fallstudie auch immer angeben. Für die Formalisierung ist sie aber nicht erforderlich.

Da wir bei der Definition der Korrespondenz von Zuständen mittels der Zusammenhangsinvariante völlige Freiheit lassen, kann eine Teilberechnung aus einer beliebigen Zahl von Regelanwendungen bestehen. Die Anzahl kann von

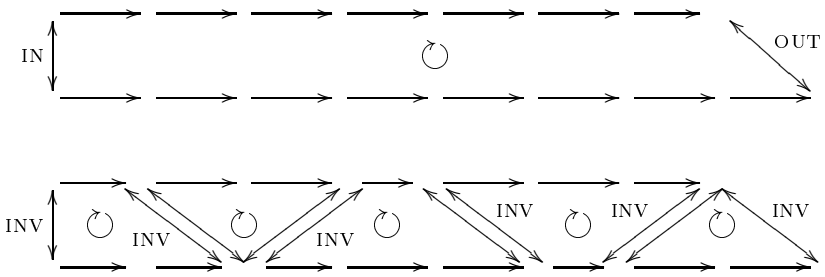


Abbildung 6.3 : Zerlegung des Gesamtdiagramms (oben) in Teildiagramme mit Hilfe einer Zusammenhangsinvariante (unten)

den Werten bestimmter Zustandsvariablen abhängen. Im Extremfall kann es auch vorkommen, daß eine Teilberechnung einer ASM durch Optimierung in der anderen ganz entfällt. In diesem Fall entstehen dreieckige Teildiagramme.

Die Grundannahme, daß beide ASMs korrespondierende Zustände durchlaufen, muß nicht in jedem Fall erfüllt sein (ASM′ könnte durch eine beliebige Programmtransformation aus ASM entstanden sein, z.B. könnte ASM′ die Berechnungsschritte von ASM in umgekehrter Reihenfolge ausführen) Für viele Fälle ist die Grundannahme aber erfüllt, insbesondere sind bei der Compilerverifikation korrespondierende Teilberechnungen durch die Ausführung von Teilprogrammen in der Ausgangs- und Zielsprache in natürlicher Weise immer gegeben.

Die Grundidee des Modularisierungstheorems ist also: Gegeben eine Zerlegung des Gesamtdiagramms in Teildiagramme, so folgt aus dem Nachweis, daß alle Teildiagramme kommutieren, die Äquivalenz der beiden ASMs.

6.2.2 Definition des Theorems

Um diese Idee in ein Theorem umzusetzen, werden wir nun

1. die Zerlegung eines Diagramms in Teildiagramme in DL formal spezifizieren
2. die Beweisverpflichtungen für die Kommutativität von Teildiagrammen in DL angeben
3. das Modularisierungstheorem formulieren und beweisen.

Für die Formalisierung der Zerlegung eines Diagramms nehmen wir an, daß die DL-Formalisierungen von ASM und ASM′ als DL-Programme ASM($\underline{x}$) und ASM′($\underline{x}'$) gegeben sind, wobei $\underline{x}$ und $\underline{x}'$ zwei disjunkte Vektoren von Variablen sind. Eine Korrespondenz zwischen zwei Zuständen wird dann durch eine *Zusammenhangsinvariante*, i.e. eine DL-Formel $INV(\underline{x}, \underline{x}')$, deren freie Variablen in $\underline{x} \cup \underline{x}'$ enthalten sind, gegeben. Für eine Zerlegung des Diagramms in Teildiagramme ist eine Formel *INV*, die genau für die Zustandspaare $(\underline{x}, \underline{x}')$ gilt, die

die Kanten von Teildiagrammen bilden, schon ausreichend, falls keine dreieckigen Teildiagramme vorhanden sind. Dann genügt es zu zeigen, daß für je zwei Zustände, die keine Endzustände sind, ein weiteres kommutierendes Diagramm wie in Abb. 6.4 folgt. Die Größe des Diagramms muß nicht explizit gegeben wer-

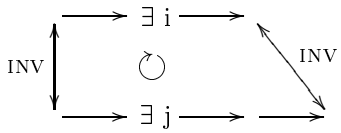


Abbildung 6.4 : generisches kommutierendes Diagramm

den, es genügt, daß es eine positive Zahl von Regelanwendungen in beiden ASMs gibt, so daß anschließend wieder *INV* gilt. In DL formalisiert ergibt dies die Beweisverpflichtung (die Voraussetzung $ndt(\underline{x}, \underline{x}') = mn$ kann ignoriert werden, sie wird weiter unten erklärt):

$$\begin{aligned} &\mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\, \mathrm{final}(\underline{\mathrm{x}}) \wedge \neg\, \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{mn} \\ \rightarrow\ &\exists\ \mathrm{i} > 0.\ \langle \textbf{loop if } \neg\, \mathrm{final}(\underline{\mathrm{x}}) \textbf{ then } \mathrm{RULE}(;\underline{\mathrm{x}}) \textbf{ times } \mathrm{i} \rangle \\ &\quad \exists\ \mathrm{j} > 0.\ \langle \textbf{loop if } \neg\, \mathrm{final}'(\underline{\mathrm{x}}') \textbf{ then } \mathrm{RULE}'(;\underline{\mathrm{x}}') \textbf{ times } \mathrm{j} \rangle \\ &\quad\quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \end{aligned} \tag{6.5}$$

Eine Zusatzproblematik besteht bei dreieckigen Diagrammen. Hier muß verhindert werden, daß ausschließlich dreieckige Diagramme aufeinanderfolgen (wie in den Abb. 6.5 und 6.6), da in diesem Fall ASM′ einen unendlichen Ablauf hätte, ohne daß ASM überhaupt einen Schritt macht (und damit die Vollständigkeit der Verfeinerung verletzt wäre).

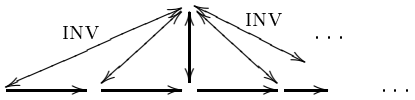


Abbildung 6.5 : Unendliche Folge von 0:n-Diagrammen

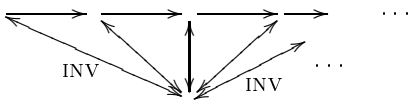


Abbildung 6.6 : Unendliche Folge von m:0-Diagrammen

Da dreieckige Diagramme in der Realität durch Optimierung oft auftreten, müssen wir die Zahl der unmittelbar hintereinander auftretenden Dreiecksdia-

gramme beschränken. Dazu muß für jedes Zustandspaar $(\underline{x}, \underline{x}')$, für das *INV* gilt, zunächst entschieden werden, welcher Typ von Diagramm als nächster folgt:

- Ein m:n-Diagramm, in dem sowohl ASM als auch ASM′ eine positive Zahl von Schritten machen,
- ein m:0-Diagramm, in dem nur ASM eine positive Zahl von Schritten macht, oder
- ein 0:n-Diagramm, in dem nur ASM′ eine positive Zahl von Schritten macht

Wir treffen diese Entscheidung durch die Definition einer Funktion *ndt* („next diagram type"), die zu jedem Zustandspaar $(\underline{x}, \underline{x}')$, für das *INV* gilt, ein Element aus $\{mn, m0, 0n\}$ liefert. Die Beschränkung der Zahl der m:0-Diagramme erfolgt dann mit Hilfe einer Funktion *execm0*, die zu jedem Zustandspaar $(\underline{x}, \underline{x}')$ mit $INV(\underline{x}, \underline{x}')$ und $ndt(\underline{x}, \underline{x}') = m0$ eine natürliche Zahl liefert, die die Anzahl der nun folgenden Dreiecksdiagramme beschränkt. Analog wird *exec0n* für 0:n-Diagramme definiert.

Beweisverpflichtung (6.5) behandelt den Fall m:n und hat daher die Zusatzvoraussetzung $ndt(\underline{x}, \underline{x}') = mn$. Für m:0-Diagramme ergibt sich die folgende Beweisverpflichtung:

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{m0} \wedge \mathrm{execm0}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{k} \\ \rightarrow \exists\ \mathrm{i} > 0.\ \langle \textbf{loop if } \neg\ \mathrm{final}(\underline{\mathrm{x}}) \textbf{ then } \mathrm{RULE}(;\underline{\mathrm{x}}) \textbf{ times } \mathrm{i} \rangle \\ \qquad (\quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \\ \qquad\ \wedge\ (\neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{m0} \rightarrow \mathrm{execm0}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') < \mathrm{k})) \end{array} \tag{6.6}$$

Die Beweisverpflichtung besagt, daß ein m:0-Diagramm die Zusammenhangsinvariante aufrecht erhalten muß und zusätzlich, falls anschließend ein weiteres m:0-Diagramm folgt, der Wert der *execm0*-Funktion abgenommen haben muß (wenn $execm0(\underline{x}, \underline{x}') = k$ ist, können also noch maximal $k + 1$ m:0-Diagramme folgen). Für 0:n-Diagramme ergibt sich dual die Beweisverpflichtung

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \wedge \neg\ \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{0n} \wedge \mathrm{exec0n}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{k} \\ \rightarrow \exists\ \mathrm{j} > 0.\ \langle \textbf{loop if } \neg\ \mathrm{final}'(\underline{\mathrm{x}}') \textbf{ then } \mathrm{RULE}'(;\underline{\mathrm{x}}') \textbf{ times } \mathrm{j} \rangle \\ \qquad (\quad \mathrm{INV}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') \\ \qquad\ \wedge\ (\neg\ \mathrm{final}'(\underline{\mathrm{x}}') \wedge \mathrm{ndt}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') = \mathrm{0n} \rightarrow \mathrm{exec0n}(\underline{\mathrm{x}}, \underline{\mathrm{x}}') < \mathrm{k})) \end{array} \tag{6.7}$$

Man beachte, daß die Beweisverpflichtung für m:0-Diagramme nicht voraussetzt, daß sich ASM′ in keinem Endzustand befindet. Es ist möglich (und kommt in der WAM-Fallstudie auch vor, siehe Abschnitt 13.2), daß ASM′ bereits terminiert hat, während ASM noch „überflüssige" Schritte durchführt (bei data refinement ist eine derartige Situation nicht möglich). Es muß allerdings gefordert werden, daß in diesem Fall *nur* m:0-Diagramme möglich sind:

$$\mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\, \mathrm{final}(\underline{x}) \wedge \mathrm{final}'(\underline{x}') \rightarrow \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{m0} \tag{6.8}$$

Analog muß für n:0-Diagramme

$$\mathrm{INV}(\underline{x}, \underline{x}') \wedge \mathrm{final}(\underline{x}) \wedge \neg\, \mathrm{final}'(\underline{x}') \rightarrow \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{0n} \tag{6.9}$$

gefordert werden. Um den Zusammenhang zur Ein- und Ausgaberelation des Refinements herzustellen, muß schließlich analog zu den Beweisverpflichtungen (6.2) und (6.3) noch

$$\mathrm{IN}(\underline{x}, \underline{x}') \rightarrow \mathrm{INV}(\underline{x}, \underline{x}') \tag{6.10}$$

und

$$\mathrm{INV}(\underline{x}, \underline{x}') \wedge \mathrm{final}(\underline{x}) \wedge \mathrm{final}'(\underline{x}') \rightarrow \mathrm{OUT}(\underline{x}, \underline{x}') \tag{6.11}$$

gefordert werden. Mit diesen Beweisverpflichtungen sind wir nun in der Lage, das Modularisierungstheorem zu formulieren:

Theorem 2 *Modularisierungstheorem für sequentielle ASMs.*
Sind eine Verfeinerung von ASM zu ASM′, ein Prädikat *INV* sowie Funktionen *ndt*, *exec0n*, *execm0* so gegeben, daß die Beweisverpflichtungen (6.5), (6.6), (6.7), (6.8), (6.9), (6.10), (6.11) alle gültig sind, so ist die Verfeinerung von ASM zu ASM′ korrekt und vollständig:

$$\begin{aligned} &\text{ASM deterministisch} \wedge \text{ASM}' \text{ deterministisch} \\ &\wedge (6.5) \wedge (6.6) \wedge (6.7) \wedge (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\ &\Rightarrow \text{ASM} \bowtie \text{ASM}' \end{aligned}$$

Bevor wir den Beweis des Theorems diskutieren, einige Anmerkungen zur Anwendung:

- Das Theorem erfordert keine getrennte Modularisierung für Korrektheit und Vollständigkeit
- Die Hauptschwierigkeit bei der Anwendung des Theorems ist, eine geeignete Zusammenhangsinvariante zu finden. Der Typ des folgenden Diagramms ergibt sich dagegen meist sehr einfach daraus, welcher Fall in den Regeln von ASM oder ASM′ vorliegt. Die Größe von *execm0* (und analog *exec0n*) ist häufig 0, d.h. auf ein m:0-Diagramm folgt sicher kein weiteres. Andernfalls ist *execm0* häufig die Größe einer Datenstruktur aus dem Zustand von ASM, die gerade abgebaut wird.

- Data Refinement ergibt sich als einfacher Spezialfall, in dem $INV(\underline{x}, \underline{x}') \equiv abstr(\underline{x}') = \underline{x}$ ist und ndt konstant mn (also keine dreieckigen Diagramme) ist. Die Beweisverpflichtung (6.1) ergibt sich dann aus (6.5), in dem i und j beide auf 1 gesetzt werden. (6.4) folgt trivial aus (6.8) und (6.9).
- Die gewonnenen Teildiagramme sind von derselben Form wie das Ausgangsdiagramm. Es ist also möglich, das Modularisierungstheorem rekursiv auf Teildiagramme anzuwenden, was in der WAM-Fallstudie für die in Abschnitt 15.2 betrachtete Verfeinerung 5/6 auch gemacht wurde.

6.2.3 Der Beweis des Theorems

Der Beweis des Modularisierungstheorems besteht aus zwei Teilen: Einem Teil, in dem unter Annahme der Beweisverpflichtungen die Korrektheit der Verfeinerung gezeigt wird, und einem zweiten Teil, in dem unter denselben Annahmen die Vollständigkeit bewiesen wird. Da die beiden Beweisteile dual sind (lediglich die Rollen von ASM und ASM′ werden vertauscht), beschränken wir uns auf den Nachweis der Korrektheit.

Der Beweis erfolgt durch Reduktion auf Aussagen, die durch Induktion über die Zahl der angewandten Regeln von ASM′ bewiesen werden können. Um diese Aussagen einfach formulieren zu können, bezeichnen wir für einen ASM-Zustand $\underline{x}$ mit $\underline{x}_i$ den Zustand der ASM, der sich nach i Regelanwendungen ergibt. Formal kann $\underline{x}_i$ in DL durch

$$\underline{y} = \underline{x} \rightarrow \langle \textbf{loop if } \neg \text{ final}(\underline{y}) \textbf{ then } \text{RULE}(;\underline{y}) \textbf{ times } i \rangle \underline{y} = \underline{x}_i$$

definiert werden (beachte, daß wenn $\underline{x}$ ein Endzustand ist, $\underline{x}_i = \underline{x}$ gilt). Wir betrachten dann die durch

$$\text{PROP}(\underline{x}, \underline{x}') \leftrightarrow \exists\, i, j.\ \text{INV}(\underline{x}_i, \underline{x}'_j)$$

definierte Aussage *PROP*. *PROP* besagt informell, daß $(\underline{x}, \underline{x}')$ ein Paar von Zuständen ist, so daß es eine Zahl i von Regelanwendungen von ASM und eine Zahl j von Regelanwendungen von ASM′ gibt, so daß für die dann erreichten Zustände die Zusammenhangsinvariante gilt. Für diese Aussage gilt nun das folgende Lemma:

Lemma 1 *PROP* ist eine Invariante von ASM′: Sind $\underline{x}$, $\underline{x}'$ zwei Zustände von ASM und ASM′ mit $INV(\underline{x}, \underline{x}')$, so gilt $PROP(\underline{x}, \underline{x}'_k)$ für alle Zustände $\underline{x}'_k$, die während des (weiteren) Ablaufs von ASM′ erreicht werden.

Beweis von Lemma 1 Der Beweis erfolgt durch Induktion über die Zahl k der angewandten Regeln. Der Basisfall ($k = 0$) ist trivial. Im Induktionsschritt sind 2 Zustände $\underline{x}$, $\underline{x}'$ mit $INV(\underline{x}, \underline{x}')$ sowie 2 Werte i und j gegeben, so daß $INV(\underline{x}_i, \underline{x}'_{k+j})$ gilt. Gesucht sind i' und j', so daß $INV(\underline{x}_{i'}, \underline{x}'_{(k+1)+j'})$ gilt.

Einfach ist der Fall $j \neq 0$ mit $i' := i$, $j' := j-1$, ebenso der Fall in dem $\underline{x'}_k$ bereits ein Endzustand ist. Andernfalls wird zunächst das unten angegebene Lemma 2 benötigt, um aus der Gültigkeit von $INV(\underline{x}_i, \underline{x'}_k)$ ein $i'' \geq 0$ zu konstruieren mit $INV(\underline{x}_{i+i''}, \underline{x'}_k)$ und entweder $ndt(\underline{x}_{i+i''}, \underline{x'}_k) \neq m0$ oder $final(\underline{x}_{i+i''}, \underline{x'}_k)$. Im ersten Fall folgt aus den Annahmen (6.5) und (6.7) die Existenz von $i''' \geq 0$ und $j''' > 0$, so daß $INV(\underline{x}_{i+i''+i'''}, \underline{x'}_{k+j'''})$ gilt. Somit kann $i' := i'' + i'''$, $j' := (j''' - 1)$ gewählt werden. Im zweiten Fall ist wegen (6.9) nur ein 0:n-Diagramm möglich und der Beweis folgt mit (6.7) genauso. □

Der Beweis benötigte das folgende Lemma, das besagt, daß ausgehend von zwei korrespondierenden Zuständen nur endlich oft ein m:0-Diagramm folgen kann. Der dabei von ASM erreichte Endzustand ist $\underline{x}_i$.

Lemma 2 Für zwei beliebige Zustände $\underline{x}$, $\underline{x'}$ mit $INV(\underline{x}, \underline{x'})$ gibt es ein $i \geq 0$, so daß $INV(\underline{x}_i, \underline{x'})$ und entweder $ndt(\underline{x}_i, \underline{x'}) \neq m0$ oder $final(\underline{x}_i)$ gilt.

Beweis von Lemma 2 Falls nicht $ndt(\underline{x}, \underline{x'})$ schon selbst ungleich $m0$ ist oder schon $final(\underline{x})$ gilt (und somit das Theorem mit $i := 0$ gilt), erfolgt der Beweis durch (noethersche) Induktion über die Größe von $execm0(\underline{x}, \underline{x'})$. Nach (6.6) gibt es ein $i' > 0$, so daß $INV(\underline{x}_{i'}, \underline{x'})$ und entweder $execm0(\underline{x}_{i'}, \underline{x'})$ kleiner geworden ist, so daß die Behauptung durch Anwendung der Induktionshypothese folgt, oder $ndt(\underline{x}_{i'}, \underline{x'}) \neq m0$ ist, so daß die Behauptung mit $i := i'$ folgt. □

Beweis von Theorem 2 Mit Hilfe der Lemmas 1 und 2 läßt sich die Korrektheit der Verfeinerung wie folgt zeigen: Sei $(\underline{x'}, \underline{x'}_1, \ldots \underline{x'}_k)$ ein terminierender Ablauf von ASM′ (es gilt also $final'(\underline{x'}_k)$) und $\underline{x}$ ein Zustand mit $IN(\underline{x}, \underline{x'})$. Dann gilt nach (6.10) auch $INV(\underline{x}, \underline{x'})$. Aus Lemma 1 folgt somit, daß $PROP(\underline{x}, \underline{x'}_k)$ gilt. Dies bedeutet daß es i, j gibt, so daß $INV(\underline{x}_i, \underline{x'}_{k+j})$ gilt. Nun ist nach Definition $\underline{x'}_{k+j} = \underline{x'}_k$, es gilt also $INV(\underline{x}_i, \underline{x'}_k)$. Nach Lemma 2 gibt es nun i', so daß $INV(\underline{x}_{i+i'}, \underline{x'}_k)$ und entweder $ndt(\underline{x}_{i+i'}, \underline{x'}_k) \neq m0$ oder $final(\underline{x}_{i+i'})$ gilt. Da ersteres wegen wegen (6.8) nicht möglich ist, muß $\underline{x}_{i+i'}$ ebenfalls ein Endzustand sein. Daraus folgt schließlich mit (6.11) wie gewünscht die Existenz eines in $\underline{x}_{i+i'}$ terminierenden Ablaufs mit $OUT(\underline{x}_{i+i'}, \underline{x'}_k)$. □

Aus dem Korrektheitsbeweis des Modularisierungstheorems ergibt sich

Korollar 1 *Ist es möglich, eine Verfeinerung durch Zerlegung in m:n-Diagramme zu verifizieren, so gibt es immer auch eine Möglichkeit, die Verfeinerung mit einer Zerlegung in 1:1- und 0:1- und 1:0-Diagramme zu verifizieren*

Als neue Zusammenhangsinvariante muß einfach *PROP* gewählt werden. Allerdings ist es aus praktischer Sicht natürlich nicht sinnvoll, die stärkere Zerlegung und *PROP* tatsächlich zu wählen, da man dann einen Teil des obigen generischen Beweises dann für jede Anwendung neu geführt werden muß. Noch aufwendiger werden die Beweise, wenn man die Regelanwendungen der ASMs aus *PROP* entfernt. Dies ist möglich, wenn alle Diagramme eine feste Größe haben (die unabhängig von der Größe von Datenstrukturen der ASMs ist). Dann

kann man eine Funktion *nextij* definieren, die zu zwei Zuständen die Anzahl der notwendigen Schritte i und j liefert, die notwendig sind, um die nächsten beiden Zustände zu erreichen, für die *INV* gilt. Statt über die möglichen i und j zu quantifizieren, kann *PROP* dann in eine Konjunktion von Formeln

$$\text{nextij}(\underline{x},\underline{x}') = (i,j) \rightarrow \text{INV}(\underline{x}_i, \underline{x}'_j)$$

zerlegt werden, wobei (i, j) alle konkreten Werte durchläuft, die kleiner als die maximale Diagrammgröße sind. Schließlich müssen aus den $\underline{x}_i$ (und analog aus den $\underline{x}'_j$) noch die Regelanwendungen durch symbolisches Ausführen „herausgerechnet" werden (dies geht, da i nun jeweils eine konkrete Zahl ist). Als Ergebnis erhält man ebenfalls eine Zusammenhangsinvariante, mit der sich die Korrektheit zeigen läßt. Da *INV* gerade das Konjunktionsglied für $(i, j) = (0, 0)$ darstellt, ist die so berechnete Zusammenhangsinvariante aber unnötig groß, falls bei der Verfeinerung andere Diagramme als 1:1- und 0:1- und 1:0-Diagramme zu verifizieren sind. Für die Verifikation ist es also empfehlenswert, die Diagramme *so groß wie möglich zu machen*, um die Zusammenhangsinvariante möglichst klein zu halten. Zwei konkrete Beispiele in der Prolog-WAM-Fallstudie, die diesen Sachverhalt beispielhaft zeigen, sind die Verfeinerungen 2/3 und 3/4 (vergleiche die Bemerkung am Ende von Abschnitt 13.2, sowie den Aufwandsvergleich für die beiden Verfeinerungen in KIV vs. in Isabelle in Abschnitt 20).

6.2.4 Formalisierung des Beweises in DL

Der oben informell gegebene Beweis läßt sich in DL formalisieren. Die Eigenschaft *PROP* wird dabei durch

$$\begin{aligned}&\text{PROP}(\underline{x}, \underline{x}') \equiv\\ &\exists\ i, j.\ \langle \textbf{loop if } \neg\ \text{final}(\underline{x}) \textbf{ then } \text{RULE}(;\underline{x}) \textbf{ times } i\rangle \\ &\qquad \langle \textbf{loop if } \neg\ \text{final}'(\underline{x}') \textbf{ then } \text{RULE}'(;\underline{x}') \textbf{ times } j\rangle\ \text{INV}(\underline{x}, \underline{x}')\end{aligned} \tag{6.12}$$

definiert. Der formale Beweis des Modularisierungstheorems erforderte in KIV 452 Beweisschritte und 64 Interaktionen, von denen je die Hälfte für Korrektheit und Vollständigkeit notwendig waren. Darin enthalten sind auch die Beweise für elementare Fakten wie $(\underline{x}_i)_{i'} = \underline{x}_{i+i'}$. Durch Instanzierung (Aktualisierung der Spezifikation) kann das Modularisierungstheorem auf jede konkrete ASM-Verfeinerung angewandt werden. Die vollständige formale Spezifikation und die bewiesenen Theoreme und Lemmata finden sich in C.2. Die Theoreme *corr-step* und *finite-0n* aus dem Anhang entsprechen Lemma 1 bzw. dem Fall $ndt(\underline{x}, \underline{x}') =$ *0n* aus Lemma 2.

6.2.5 Formalisierung des Beweises in Prädikatenlogik

Auch in Prädikatenlogik läßt sich der Beweis des Modularisierungstheorems führen. Bei Verwendung von first-order Logik müssen dazu zunächst Zustandsübergangsrelationen als Datentyp formalisiert werden (in Higher-Order Logik

entfällt dieser Schritt). Am einfachsten ist es an dieser Stelle, den schon in Abschnitt 4 verwendeten Datentyp dynamischer Funktionen zu verwenden. Eine Relation ist dann einfach eine dynamische Funktion r mit einem Paar $st_1 \times st_2$ von Zuständen als Argument und einem booleschen Ergebniswert. $r[st_1 \times st_2]$ gilt dann genau, wenn st_2 ein möglicher Nachfolgezustand von st_1 ist. Die Zustandsübergangsrelation ρ einer ASM läßt sich dann als Konstante des Datentyps Relation darstellen. Da wir sequentielle ASMs betrachten, gilt für ρ das Funktionalitätsaxiom

$$\rho[\mathrm{st}_1 \times \mathrm{st}_2] \wedge \rho[\mathrm{st}_1 \times \mathrm{st}_3] \rightarrow \mathrm{st}_2 = \mathrm{st}_3$$

Das Prädikat *final*, das die Endzustände charakterisiert, wird durch

$$\mathrm{final}(\mathrm{st}) \equiv \neg\ \exists\ \mathrm{st}_0.\ \rho[\mathrm{st} \times \mathrm{st}_0]$$

definiert. Für den Beweis in Prädikatenlogik muß dann die Semantik der ASM formalisiert werden. Um die i-fache Iteration der Regelanwendung auszudrücken, wird eine entsprechende Relation ρ^{i} definiert:

$$\rho^0[\mathrm{st}_1 \times \mathrm{st}_2] \leftrightarrow \mathrm{st}_1 = \mathrm{st}_2$$
$$\rho^{\mathrm{i}+1}[\mathrm{st}_1 \times \mathrm{st}_2] \leftrightarrow \exists\ \mathrm{st}_0.\ \rho^{\mathrm{i}}[\mathrm{st}_1 \times \mathrm{st}_0] \wedge \rho[\mathrm{st}_0 \times \mathrm{st}_2]$$

Sie entspricht in DL der Semantik von

loop if $\neg$ final(st) **then** RULE(st) **else abort times** i

Schließlich ergibt sich daraus die Definition der Ein-Ausgaberelation ρ^* der ASM zu:

$$\rho^*[\mathrm{st}_1 \times \mathrm{st}_2] \leftrightarrow \exists\ \mathrm{i}.\ \rho^{\mathrm{i}}[\mathrm{st}_1 \times \mathrm{st}_2] \wedge \mathrm{final}(\mathrm{st}_2)$$

Sie entspricht genau der Semantik der *while*-Schleife in DL. Die Beweisverpflichtungen und Beweise in der prädikatenlogischen Formalisierung ergeben sich aus den DL-Varianten dann einfach, indem systematisch

$$\exists\ \mathrm{i}.\ \langle \textbf{loop if}\ \neg\ \mathrm{final}(\mathrm{st})\ \textbf{then}\ \mathrm{RULE}(\mathrm{st})\ \textbf{times}\ \mathrm{i} \rangle\ \varphi(\mathrm{st})$$

durch

$$\exists\ \mathrm{i},\mathrm{st}_0\ .\ \rho^{\mathrm{i}}(\mathrm{st},\mathrm{st}_0) \wedge \varphi(\mathrm{st}_0)$$

(mit einer neuen Variable st_0) ersetzt wird. Der Aufwand zum Führen der Beweise ist in Prädikatenlogik (PL) mit 98 Interaktionen etwas höher als in DL. Dies liegt in erster Linie daran, daß in DL die Berechnung der notwendigen Iterationen der *while*-Schleife durch Heuristiken unterstützt wird, während in PL die entsprechende Zahl interaktiv durch Instanzieren von Quantoren angegeben werden muß. Die Anzahl der Beweisschritte ist mit 247 etwas geringer, da die Anwendung von DL-Regeln nun zu einem Teil durch die Anwendung von Rewrite-Schritten ersetzt wird, und die Simplifikationstaktik oft mehrere Rewrite-Regeln in einem Schritt anwendet.

6.3 Trace-Korrektheit

Die in Kapitel 5 gegebene Definition der Korrektheit einer ASM-Verfeinerung beruht auf einem Vergleich der Ein-/Ausgabe-Relationen der beiden ASMs. Eine Alternative zu dieser Definition ist es, die Abläufe von zwei ASMs zu vergleichen. Im einfachsten Fall gibt es wie bei data refinement (Abschnitt 6.1) eine Abstraktionsfunktion *abstr*, so daß für jeden Ablauf $(\underline{x}'_0, \underline{x}'_1, \ldots)$ von ASM′ $(abstr(\underline{x}'_0), abstr(\underline{x}'_1), \ldots)$ ein Ablauf von ASM ist. Der wesentliche Unterschied zu unserer Definition liegt zum einen darin, daß schon in der Definition Bezug auf eine Abstraktionsfunktion für die Programmzustände genommen wird. Zum anderen werden nicht nur endliche Abläufe, sondern auch unendliche betrachtet. Bei einer korrekten Verfeinerung ist es bei dieser Definition nicht mehr erlaubt, einen terminierenden Ablauf von ASM durch einen nicht terminierenden Ablauf von ASM′ zu verfeinern. Für deterministische ASMs ist dieses Verbot nicht sehr gravierend, da für *vollständige* Verfeinerungen die Implementierung eines terminierenden Ablaufs durch einen nicht terminierenden ebenfalls unmöglich ist. Für indeterministische (z.B. verteilte) ASMs, die wir im folgenden Abschnitt allgemein betrachten, besteht aber ein Unterschied. Um den Unterschied zu präzisieren, betrachte man folgende Verfeinerung einer durch

RULE(**var** init,b) ≡
    if init **then** b := false, init := false

gegebenen deterministischen ASM zu einer durch

RULE′(**var** init,b) ≡
    if init **then** b := ?, init := false **else if** b **then** b := b

gegebenen indeterministischen ASM′ (die DL-Anweisung $b := ?$ rät in ASM′ einen booleschen Wert. Sie entspricht der in [Gur95], Abschnitt 4.2 definierten **choose**-Anweisung).

Für einen Ausgangszustand mit $b = init = true$ hat ASM genau einen Ablauf, der *RULE* einmal anwendet, dabei b und *init* auf *false* setzt, und dann terminiert (*RULE* ist nicht mehr anwendbar). ASM′ besitzt diesen Ablauf ebenfalls, wenn bei der ersten Regelanwendung $b = false$ gewählt wird. In ASM′ ist

aber ein zusätzlicher, nicht terminierender Ablauf möglich, in dem $b = \mathit{true}$ gewählt wird. $RULE'$ wird dann unendlich oft angewandt, ohne daß sich der Zustand (mit $b = \mathit{true}$ und $\mathit{init} = \mathit{false}$) weiter ändert.

Diese Verfeinerung ist in unserem Sinn korrekt und vollständig (wenn die IN und OUT-Relation als Identität gewählt werden), da jedem *endlichen* Ablauf der einen ASM ein endlicher Ablauf der anderen ASM zugeordnet ist. Sie ist aber nicht trace-korrekt, da dem unendlichen Ablauf von ASM$'$ kein Ablauf der ASM zugeordnet werden kann.

Ob die Verfeinerung in einem intuitiven Sinn als korrekt angesehen werden kann, hängt davon ab, ob man den Ablauf oder nur das Ergebnis einer ASM beobachten kann. Wenn nur die Ergebnisse relevant sind, so ist die Verfeinerung korrekt, da ASM$'$ keine falschen Ergebnisse (also solche, die ASM nicht liefern kann) liefert. Wenn aber die beiden ASMs als reaktive Systeme angesehen werden, und ein Beobachter (mindestens einige) Zwischenzustände beider ASMs beobachten und vergleichen kann, so würde man die Verfeinerung nicht als korrekt ansehen.

Wir definieren und formalisieren daher nun den Begriff der Trace-Korrektheit allgemein, so daß wir ihn auch für indeterministische ASMs verwenden können. An die Stelle der Abstraktionsfunktion tritt wieder ein allgemeiner „Vergleich beobachtbarer Zustände“, der unserer Zusammenhangsinvariante entspricht. Für Trace-Korrektheit wird dann gefordert, daß es zu jedem Ablauf von ASM$'$ einen Ablauf von ASM und Zwischenpunkte auf den Abläufen geben muß, auf denen die Zusammenhangsinvariante gilt. Für einen endlichen Ablauf von ASM$'$ muß auch der Ablauf von ASM und die Anzahl der Zwischenpunkte endlich sein. Außerdem müssen dann der letzten Zwischenpunkte am Ende beider Abläufe liegen. Für einen unendlichen Ablauf von ASM$'$ muß dagegen sowohl der Ablauf von ASM als auch die Zahl der Zwischenpunkte unendlich sein. Formal bedeutet dies:

Definition 2 Eine Verfeinerung von ASM zu ASM$'$ heißt trace-korrekt, kurz ASM $\blacktriangleright$ ASM$'$, wenn es eine Zusammenhangsinvariante $INV(\underline{x}, \underline{x}')$ gibt, so daß

- zu jedem endlichen Ablauf $(\underline{x'}_0, \underline{x'}_1, \ldots, \underline{x'}_m)$ (mit $\underline{x'}_m \in F'$) von ASM$'$ und jedem Zustand $\underline{x}_0$ mit $IN(\underline{x}_0, \underline{x'}_0)$ ein endlicher Ablauf $(\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_n)$ von ASM (mit $\underline{x}_n \in F$) und zwei streng monoton wachsende endliche Folgen von natürlichen Zahlen $(i_0, i_1, \ldots, i_p)$ und $(j_0, j_1, \ldots, j_p)$ gleicher Länge p existieren, so daß $i_p = m$, $j_p = n$ und für alle $k \leq p$ $INV(\underline{x}_{i_k}, \underline{x'}_{j_k})$ gilt.

- zu jedem unendlichen Ablauf $(\underline{x'}_0, \underline{x'}_1, \ldots)$ von ASM$'$ und jedem Zustand x_0 mit $IN(\underline{x}_0, \underline{x'}_0)$ ein unendlicher Ablauf $(\underline{x}_0, \underline{x}_1, \ldots)$ von ASM und zwei streng monoton wachsende, unendliche Folgen von natürlichen Zahlen $(i_0, i_1, \ldots)$ und $(j_0, j_1, \ldots)$ existieren, so daß für alle n $INV(\underline{x}_{i_n}, \underline{x'}_{j_n})$ gilt.

- (6.11) gilt, d.h. für zwei Endzustände impliziert die Zusammenhangsinvariante OUT

```
stream =
enrich Dynfun[nat,state] with
functions cons : state × stream → stream;
          cdr  : stream        → stream;
variables st : state; s : stream;
axioms cons(st,s)[0] = st,
       cons(st,s)[m +1] = s[m],
       cdr(s)[m] = s[m +1]
end enrich
```

Abbildung 6.7 : Spezifikation von Strömen

Die mit Hilfe der Zusammenhangsinvarianten vergleichbaren Zustände sind also $(\underline{x}_{i_0}, \underline{x}_{i_1}, \ldots)$ und $(\underline{x}_{j_0}, \underline{x}_{j_1}, \ldots)$. Aus der Definition folgt unmittelbar:

Theorem 3 *Beziehungen zwischen Korrektheit und Trace-Korrektheit*
Für zwei abstrakte Zustandsmaschinen ASM und ASM′ gilt:

- ASM ▸ ASM′ ⇒ ASM ▹ ASM′.
- ASM′ deterministisch und ASM ⋈ ASM′ ⇒ ASM ▸ ASM′

Um die Definition der Trace-Korrektheit in DL zu formalisieren, benötigen wir zunächst eine Formalisierung der bisher nur semantisch gegebenen Abläufe einer ASM. Wir benutzen dazu die in Abb. 6.7 gegebene Erweiterung einer Spezifikation dynamische Funktionen von natürlichen Zahlen nach Zuständen der ASM:

Für eine ASM-Regel *RULE* mit Zustandsargument *st* ist ein Strom *s* ein Trace der ASM (mit Anfangszustand *s*[0]), falls das durch

```
Trace(s) ≡
∀ m, st. st = s[m] → ⟨if ¬ final(st) then RULE(;st)⟩ st = s[m +1]
```

spezifizierte Prädikat *Trace*(*s*) zutrifft. Die Definition hängt von der gewählten ASM-Regel *RULE* ab und ist so gewählt, daß einem endlichen Ablauf (st_0, st_1, …, st_m) ein Trace *s* mit $s[k] = st_k$ für $k \leq m$ und $s[k] = st_m$ für $k > m$ entspricht (durch die Abfrage auf ¬ *final*(*st*)). Die Forderung nach Trace-Korrektheit bezüglich *INV*, läßt sich dann durch

```
∀ s′.     Trace′(s′)
    → ∃ s.   Trace(s)
           ∧ ∀ m, k. ∃ i, j.   i ≥ m ∧ j ≥ k ∧ INV(s[i],s′[j])
                             ∧ (final(s[i]) ↔ final′(s′[j]))
```
(6.13)

formalisieren. Darin ist *Trace′* das zur ASM′-Regel *RULE′* und *Trace* das zur ASM-Regel *RULE* definierte Prädikat. Man beachte, daß „*INV* gilt unendlich

oft“ durch „zu je zwei Positionen m, k in den Abläufen gibt es noch zwei größere i, j, bei denen INV gilt“ formalisiert wurde, wie dies auch in temporallogischen Formalisierungen („unendlich oft φ“ $\equiv \Box\Diamond\varphi$) üblich ist. Die Fallunterscheidung für endliche und unendliche Abläufe aus der Definition ist durch unsere formale Definition von Abläufen (die endliche Abläufe zu unendlichen, die den Endzustand wiederholen, verlängert) überflüssig geworden. Der Sonderfall von endlichen Traces ist in der Definition durch die Forderung $\mathit{final}(s[i]) \leftrightarrow \mathit{final}'(s'[j])$ berücksichtigt.

Wir zeigen nun, daß der Unterschied zwischen Korrektheit und Trace-Korrektheit minimal ist. Die Beweisverpflichtungen für Korrektheit implizieren nämlich bereits Trace-Korrektheit für die Zusammenhangsinvariante. Informell gilt dies einfach deshalb, weil wir bei der Zerlegung von 2 Traces in kommutierende Diagramme keine Endlichkeit der Traces voraussetzen, und weder n:∞-Diagramme noch unendlich viele unmittelbar aufeinanderfolgende 0:n-Diagramme zugelassen sind. Eine Analyse des Korrektheitsbeweises zeigt, daß die Bedingung, daß nur endlich viele 0:n-Diagramme erlaubt sind (i.e. daß der Wert von *exec0n* in Beweisverpflichtung (6.7) abnimmt), für die Korrektheit nicht notwendig ist (wohl aber für Vollständigkeit und auch für Trace-Korrektheit). Formal gilt das folgende Theorem:

Theorem 4 Sind die in Theorem 2 genannten Beweisverpflichtungen für Korrektheit erfüllt, so ist die Verfeinerung auch trace-korrekt für die Zusammenhangsinvariante INV. Formal:

$$\begin{aligned} &(6.5) \wedge (6.6) \wedge (6.7) \wedge (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\ \Rightarrow\ &\mathrm{ASM} \blacktriangleright \mathrm{ASM}' \end{aligned}$$

Für den formalen Beweis definieren wir

$$\mathrm{INV}'(\mathrm{st},\mathrm{st}') \equiv \mathrm{INV}(\mathrm{st},\mathrm{st}') \wedge (\mathrm{final}(\mathrm{st}) \leftrightarrow \mathrm{final}(\mathrm{st}'))$$

und zeigen zunächst, daß wenn in zwei Zuständen INV gilt, im weiteren Verlauf auch immer 2 Zustände mit INV' erreicht werden:

Lemma 3 Gelten die Beweisverpflichtungen von Theorem 2 und gilt für einen Trace s' von ASM$'$ $INV(st, s'[0])$, so gibt es einen Trace s von ASM mit $s[0] = st$ und $i, j \geq 0$, so daß $INV'(s[i], s'[j])$ gilt.

Beweis von Lemma 3 Für den Beweis sind 4 Fälle zu betrachten: Die beiden Fälle, daß st und $s'[0]$ beide Endzustände bzw. beide keine Endzustände sind, sind mit $i = j := 0$ trivial. Ist st ein Endzustand, und $s'[0]$ kein Endzustand, so gibt es nach Lemma 2 ein i, so daß $INV(s[i], s'[0])$ gilt, und entweder $\mathit{final}(s[i])$ oder $ndt(s[i], s'[0]) \neq \mathrm{m0}$ gilt. Da der letzte Fall wegen Beweisverpflichtung (6.8) nicht in Frage kommt, folgt die Behauptung mit $j := 0$. Es bleibt als vierter

Fall, daß $s'[0]$ ein Endzustand, aber st kein Endzustand ist. Dieser folgt analog aus dem zu Lemma 2 dualen Lemma. □

Mit Hilfe des Lemmas läßt sich nun zeigen, daß wenn für zwei Zustände INV' gilt, sich immer ein Diagramm mit einer positiven Zahl von Schritten für beide ASMs anfügen läßt, so daß am Ende wieder INV' gilt:

Lemma 4 Gelten die Beweisverpflichtungen von Theorem 2 und gilt für einen Trace s' von ASM' $INV'(st,s'[0])$, so gibt es einen Trace s von ASM mit $s[0] = st$ und $i, j > 0$, so daß erneut $INV'(s[i], s'[j])$ gilt.

Beweis von Lemma 4 Wenn sowohl *final*(st) als auch *final'*$(s'[0])$ gilt, so ist $s'[1] = s'[0]$ und für jeden beliebigen in st beginnenden Trace s $s[1] = s[0] = st$. Somit genügt dann $i = j := 1$. Wenn dagegen sowohl st als auch $s'[0]$ kein Endzustand sind, so sind 3 Fälle zu untersuchen:

- $ndt(st,s'[0]) = mn$. Dann werden (entsprechend Beweisverpflichtung (6.5)) nach $i > 0$ ASM-Schritten und $j > 0$ ASM'-Schritten zwei Zustände erreicht, so daß $INV(s[i], s'[j])$ gilt. Die Behauptung folgt nun mit obigem Lemma 3.
- $ndt(st, s'[0]) = m0$. Lemma 2 liefert ein $i > 0$, so daß $INV(s[i], s'[0])$ und $ndt(s[i], s'[0]) \neq m0$ gilt. Falls nun $ndt(s[i], s'[0]) = mn$, folgt die Behauptung wie im ersten Fall. Andernfalls ist $ndt(s[i], s'[0]) = 0n$, und das nächste 0:n-Diagramm (entsprechend Beweisverpflichtung (6.7)) ergibt ein $j > 0$, so daß $INV(s[i], s'[j])$ gilt. Die Behauptung folgt nun wieder mit Lemma 3
- $ndt(st, s'[0]) = 0n$. Dieser Fall ist dual zum vorigen.

□

Beweis von Theorem 4 Der Beweis folgt nun durch induktives Anhängen von m:n-Diagrammen mit $m, n > 0$, die INV' aufrecht erhalten, entsprechend Lemma 4. Formal konstruieren wir im k-ten Schritt einen k-ten Trace s^k, sowie zwei streng monoton wachsende Folgen $(i_0, \ldots i_k)$ und $(i_0, \ldots i_k)$, so daß für alle $p \leq k$

$$\mathrm{INV'(s^k[i_p],\ s'[j_p])}$$

gilt. Der k-te Trace enthält also k kommutierende Diagramme wie in Abbildung 6.8 gezeigt.

Der Induktionsanfang folgt aus Lemma 3, da in 2 Anfangszuständen der ASMs die Zusammenhangsinvariante gilt. Der Induktionsschritt folgt aus Lemma 4 mit Hilfe des Auswahl-Axioms der Higher-Order Logik

$$(\forall\ \mathrm{x.}\ \exists\ \mathrm{y.}\ \mathrm{p(x,y)}) \rightarrow \exists\ \mathrm{f.}\ \forall\ \mathrm{x.}\ \mathrm{p(x,f(x))}$$

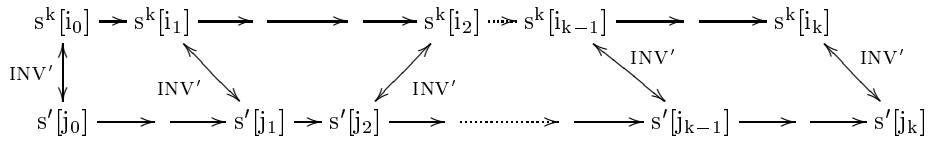


Abbildung 6.8 : kommutierende Diagramme im Beweis von Trace-Korrektheit

Dieses wird benötigt, um aus der *Möglichkeit* des Anbaus eines Diagramms (in Anhang C.3 durch das Prädikat p formalisiert) eine *Funktion* zu konstruieren, die den nächsten Trace s^{k+1}, das nächste i_{k+1} und das nächste j_{k+1} berechnet. Der gesuchte Trace s wird schließlich durch $s[k] := s^k[k]$ definiert. Er stimmt jeweils bis zur Stelle $i_k \geq k$ mit s^k überein. Für die in der Aussage (6.13) des Theorems gesuchten i und j kann nun $i_{max(m,n)}$ und $j_{max(m,n)}$ gewählt werden, da diese beide größer oder gleich m und n sind. □

Die induktive Konstruktion von Tupeln (aus s^k, i_k und j_k) macht den Beweis der Trace-Korrektheit in KIV etwas aufwendiger als den für Korrektheit. Insgesamt waren für den allgemeinsten Fall, indeterministische ASMs mit Diagrammen indeterministischer Größe, den wir im nächsten Abschnitt betrachten, 412 Beweisschritte mit 138 Interaktionen erforderlich (aufbauend auf Lemma 2). Die formal bewiesenen Theoreme finden sich in Anhang C.3.

Für den Spezialfall, in dem alle Diagramme 1:n oder 0:n-Diagramme sind (i.e. der Fall, in dem die Beweisverpflichtungen (6.5) und (6.6) jeweils mit $i := 1$ bewiesen werden können), sind alle Zustände von ASM beobachtbar (i.e. per *INV* mit einem Zustand der ASM′ verbunden). In diesem Fall läßt sich der Beweis des Theorems dahingehend spezialisieren, daß statt einer beliebigen Folge $(i_0, i_1, \ldots)$ die Folge (0,1, ...) gewählt wird. Ein analoges Korollar gilt natürlich auch im dualen Fall von m:1 und m:0-Diagrammen. Somit erhält man die Aussage, daß bei Data Refinement (nur 1:1-Diagramme) für alle n $INV(\underline{x}_n, \underline{x'}_n)$ gilt, als Spezialfall der beiden Korollare.

6.4 Erweiterungen für indeterministische ASMs

In diesem Abschnitt betrachten wir statt sequentieller ASMs beliebige indeterministische ASMs. Verteilte ASMs, wie sie in Abschnitt 4.4 beschrieben wurden, sind ein wichtiges Beispiel für Indeterminismus, aber auch andere Erweiterungen, wie das in [Gur95], Abschnitt 4.1 beschriebene *CHOOSE*-Konstrukt ergeben ASMs, deren Semantik ein indeterministisches Zustandsübergangssystem ist. Im folgenden Abschnitt beschreiben wir zunächst, wie sich das Modularisierungstheorem des vorigen Abschnitts an indeterministische ASMs anpassen läßt. Im zweiten Abschnitt geben wir schließlich ein Beispiel an, das ein Diagramm *indeterministischer Größe* ergibt. Dieses geht über die im ersten Abschnitt be-

trachtete Anpassung hinaus, da dort wie im deterministischen Fall davon ausgegangen wird, daß sich die Größe eines Teildiagramms aus der Kenntnis seiner Anfangszustände bestimmen läßt.

6.4.1 Anpassung des Modularisierungstheorems an indeterministische ASMs

Sieht man sich die Grundidee des Modularisierungstheorems an, so scheint es zunächst, als ob sich die Idee der Zerlegung auch für indeterministische ASMs problemlos anwenden läßt.

Analysiert man allerdings den Beweis des vorigen Abschnitts, so stellt man fest, daß der Determinismus der ASM beim Beweis des Lemmas wesentlich ausgenutzt wurde, um das Kommutieren eines Teildiagramms durch *eine* Beweisverpflichtung auszudrücken.

Dies läßt sich am Beispiel der Bedingung (6.5) zeigen: Für ein indeterministisches System besagt diese Bedingung nur, daß es zu Ausgangszuständen $\underline{x}$ und $\underline{x}'$ für die *INV* gilt, Zahlen i, j geben muß, so daß für *jeweils einen möglichen* Nachfolgezustand $\underline{x}_i$ und ${\underline{x}'}_j$ wieder *INV* gilt. Um Korrektheit zu zeigen, muß aber zu *jedem möglichen* Nachfolgezustand ${\underline{x}'}_j$ ein passender Zustand $\underline{x}_i$ existieren, so daß *INV* gilt. Für Korrektheit muß daher die Bedingung (6.5) zu

$$\begin{aligned} &\mathrm{INV}(\underline{x},\underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x},\ \underline{x}') = \mathrm{mn} \\ \rightarrow\ &\exists\ \mathrm{j} > 0.\ [\textbf{loop if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}] \\ &\quad \exists\ \mathrm{i} > 0.\ \langle\textbf{loop if}\ \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}\rangle \\ &\quad\quad \mathrm{INV}(\underline{x},\ \underline{x}') \end{aligned} \tag{6.14}$$

verallgemeinert werden. Die rechte Seite der Implikation besagt nun, daß es ein j geben muß, so daß für *jede* terminierende Möglichkeit j Regeln von ASM′ anzuwenden, ein i existiert, so daß nach i Regelanwendungen von ASM wieder die Invariante gilt. Daß dies tatsächlich die gewünschte Verallgemeinerung ist, folgt aus der Tatsache, daß ASMs keine nichtterminierenden Regeln kennen, so daß alle Möglichkeiten, j Regeln von ASM′ anzuwenden, terminieren (Aussagen der Form „alle Abläufe eines Programms terminieren“ würden eine Erweiterung der DL erfordern, wie dies in [Gol82], S. 101 diskutiert wird).

Für den Nachweis der Vollständigkeit erhält man analog folgende Beweisverpflichtung für m:n-Diagramme

$$\begin{aligned} &\mathrm{INV}(\underline{x},\ \underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x},\ \underline{x}') = \mathrm{mn} \\ \rightarrow\ &\exists\ \mathrm{i} > 0.\ [\textbf{loop if}\ \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}] \\ &\quad \exists\ \mathrm{j} > 0.\ \langle\textbf{loop if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}\rangle \\ &\quad\quad \mathrm{INV}(\underline{x},\ \underline{x}') \end{aligned} \tag{6.15}$$

Für den Fall, daß sowohl die nächsten i im Ausgangszustand $\underline{x}$ anwendbaren Regeln von ASM als auch die nächsten j anwendbaren Regeln von ASM′ im Zustand $\underline{x}'$ deterministisch sind, sind die Bedingungen (6.5), (6.14) und (6.15)

äquivalent. Falls also bei der Verfeinerung eine deterministische Regel durch eine deterministische Regel verfeinert wird, entsteht für das entsprechende Teildiagramm auch nur die eine Beweisverpflichtung (6.5).

Was für m:n Diagramme gilt, gilt analog auch für m:0 und 0:n-Diagramme. Allerdings entstehen nicht 2 Beweisverpflichtungen, sondern nur jeweils eine. Für die Vollständigkeit von m:0-Diagrammen ist

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\, \mathrm{final}(\underline{x}) \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{m0} \\ \wedge\, \mathrm{execm0}(\underline{x}, \underline{x}') = \mathrm{k} \\ \rightarrow \exists\, \mathrm{i} > 0.\ [\textbf{loop if} \neg\, \mathrm{final}(\underline{x}) \textbf{ then } \mathrm{RULE}(;\underline{x}) \textbf{ times } \mathrm{i}] \\ \quad (\quad \mathrm{INV}(\underline{x}, \underline{x}') \\ \quad \wedge\, (\neg\, \mathrm{final}(\underline{x}) \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{m0} \rightarrow \mathrm{execm0}(\underline{x}, \underline{x}') < \mathrm{k})) \end{array} \tag{6.16}$$

zu zeigen. Für die Korrektheit ist weiterhin das schwächere (6.6) ausreichend. Analog ist für die Korrektheit von 0:n-Diagrammen

$$\begin{array}{l} \quad \mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\, \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{0n} \\ \wedge\, \mathrm{exec0n}(\underline{x}, \underline{x}') = \mathrm{k} \\ \rightarrow \exists\, \mathrm{j} > 0.\ [\textbf{loop if} \neg\, \mathrm{final}'(\underline{x}') \textbf{ then } \mathrm{RULE}'(;\underline{x}') \textbf{ times } \mathrm{j}] \\ \quad (\quad \mathrm{INV}(\underline{x}, \underline{x}') \\ \quad \wedge\, (\neg\, \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{0n} \rightarrow \mathrm{exec0n}(\underline{x}, \underline{x}') < \mathrm{k})) \end{array} \tag{6.17}$$

erforderlich, was die für Vollständigkeit ausreichende Bedingung (6.7) impliziert. Mit den so verstärkten Beweisverpflichtungen läßt sich das Modularisierungstheorem erneut beweisen:

Theorem 5 *Modularisierungstheorem für indeterministische ASMs*
Sind eine Verfeinerung einer indeterministischen ASM zu ASM′, ein Prädikat *INV* sowie Funktionen *ndt*, *exec0n*, *execm0* so gegeben, daß die Beweisverpflichtungen (6.14), (6.15), (6.16), (6.17), (6.8), (6.9), (6.10), (6.11) alle gültig sind, so ist die Verfeinerung von ASM zu ASM′ korrekt und vollständig:

$$\begin{array}{l} \quad (6.14) \wedge (6.15) \wedge (6.16) \wedge (6.17) \\ \wedge\, (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\ \Rightarrow \mathrm{ASM} \bowtie \mathrm{ASM}' \end{array}$$

Für die Korrektheit und Trace-Korrektheit sind (6.14), (6.17), (6.8), (6.9), (6.10), (6.11) sowie statt (6.16) das schwächere (6.6) hinreichend:

$$\begin{array}{l} \quad (6.14) \wedge (6.17) \wedge (6.6) \\ \wedge\, (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\ \Rightarrow \mathrm{ASM} \blacktriangleright \mathrm{ASM}' \end{array}$$

Der Beweis für Korrektheit und Vollständigkeit verläuft genau wie in Abschnitt 6.2.3. Für den Korrektheits- bzw. Vollständigkeitsbeweis werden nun

aber 2 verschiedene Eigenschaften *KPROP* und *VPROP* benötigt, die dual zueinander definiert sind:

$$\begin{aligned}&\text{KPROP}(\underline{x}, \underline{x}') \equiv\\ &\exists\ \text{j}.\ [\textbf{loop if} \neg\ \text{final}'(\underline{x}')\ \textbf{then}\ \text{RULE}'(;\underline{x}')\ \textbf{times}\ \text{j}]\\ &\qquad \exists\ \text{i}.\ \langle\textbf{loop if} \neg\ \text{final}(\underline{x})\ \textbf{then}\ \text{RULE}(;\underline{x})\ \textbf{times}\ \text{i}\rangle\ \text{INV}(\underline{x}, \underline{x}')\end{aligned} \tag{6.18}$$

$$\begin{aligned}&\text{VPROP}(\underline{x}, \underline{x}') \equiv\\ &\exists\ \text{i}.\ [\textbf{loop if} \neg\ \text{final}(\underline{x})\ \textbf{then}\ \text{RULE}(;\underline{x})\ \textbf{times}\ \text{i}]\\ &\qquad \exists\ \text{j}.\ \langle\textbf{loop if} \neg\ \text{final}'(\underline{x}')\ \text{RULE}'(;\underline{x}')\ \textbf{times}\ \text{j}\rangle\ \text{INV}(\underline{x}, \underline{x}')\end{aligned} \tag{6.19}$$

Beim Beweis ist zu beachten, daß „alle Zustände $\underline{x'}_k$, die während des Ablaufs von ASM′ erreicht werden“, nun nicht mehr eindeutig durch $\underline{x'}$ bestimmt sind. Der Beweis für Trace-Korrektheit ist ebenfalls identisch zum Beweis aus Abschnitt 6.3.

6.4.2 Diagramme indeterministischer Größe

Analysiert man die im vorigen Abschnitt gegebene Beweisverpflichtung (6.14), so stellt man fest, daß sie nicht die allgemeinste Form eines kommutierenden Diagramms darstellt, die für die Korrektheit der ASM-Verfeinerung hinreichend wäre. Der Grund ist, daß die Beweisverpflichtung die Zahl j der Durchläufe von ASM′, für die bei geeigneter Wahl eines Nachfolgerzustands $\underline{x'}_j$ alle Nachfolgerzustände $\underline{x}_i$ ein Zustand erreicht werden muß, in dem wieder *INV* gilt, *vor* dem Ablauf von ASM′ festlegt.

Nun kann es aber passieren, daß die Zahl j der Schritte, die erforderlich sind, nicht nur vom Anfangszustand, sondern auch von indeterministischen („Rate-“)Schritten der ASM abhängt. Um dies zu illustrieren, betrachten wir die durch die beiden Regeln *RULE* und *RULE′* definierten ASMs, die beide im Zustand $x = 0$ gestartet werden, und beide in einem Zustand mit $x = 1$ enden:

RULE(**var** x):
if x = 0 **then** x := 1 RULE′(**var** x):

if x = 0 **then choose** y ∈ nat **in** x := y +1 **else**
if x > 1 **then** x := x −1

ASM′ wählt („rät“) im ersten Schritt zufällig eine natürliche Zahl y und belegt die Variable x mit dieser Zahl plus eins, so daß x nach der ersten Regelanwendung mit einer positiven Zahl belegt ist. Diese Zahl wird dann solange dekrementiert, bis 1 erreicht ist. Dies ist offenbar äquivalent zu ASM, die x sofort auf 1 setzt. Dennoch gibt es *keine uniforme* Zahl j von ASM-Regelanwendungen,

so daß für alle möglichen Abläufe von ASM′ nach j Schritten ein zum Endzustand von ASM äquivalenter Zustand erreicht wird. Die Zahl der Durchläufe ist vielmehr abhängig von der Größe von x nach der ersten Regelanwendung.

Betrachtet man kompliziertere Fälle, so muß nicht *eine* indeterministische Regelanwendung zu Beginn die Größe des Diagramms bestimmen, sondern es sind mehrere indeterministische Schritte denkbar, die die Größe beeinflussen. Dennoch ist es trotz der Rateschritte so, daß bei *jedem* Ablauf von ASM′ *irgendwann* ein Zustand erreicht wird, so daß *INV* wieder gilt.

Um dies in DL zu formalisieren, definieren wir einen Operator $AF(\alpha, \varphi)$[1], der besagt, daß iteriertes Ausführen von α immer irgendwann zu einem Zustand führen wird, in dem φ gilt.

Mit Hilfe der in Abschnitt 6.3 definierten Abläufe läßt sich $AF(\alpha, \varphi)$ als Abkürzung für

$$\mathrm{AF}(\alpha, \varphi) \equiv \forall\ \mathrm{s.}\ (\mathrm{Trace(s)} \wedge \underline{\mathrm{x}} = \mathrm{s[0]} \rightarrow \exists\ \mathrm{m.}\ \varphi[\underline{\mathrm{x}} \leftarrow \mathrm{s[m]}]) \tag{6.20}$$

definieren. Dabei sind $\underline{x}$ die in α modifizierten Variablen, s ein Strom von Werten dieses Typs, und *Trace*(s) ist durch

$$\begin{aligned}&\mathrm{Trace(s)} \equiv\\ &\qquad \forall\ \mathrm{m}, \underline{\mathrm{x}}.\ \underline{\mathrm{x}} = \mathrm{s[m]} \rightarrow \langle\alpha\rangle\underline{\mathrm{x}} = \mathrm{s[m\ +1]}\end{aligned}$$

zu definieren. Eine Alternative, die auf die Verwendung von Strömen verzichtet, ist die Erweiterung von DL um einen Operator $AF(\alpha, \varphi)$, dessen Semantik durch die folgende Definition gegeben wird:

Definition 3 $\mathcal{A}, \mathbf{z} \models AF(\alpha,\varphi)$ genau dann, wenn für alle Zustandsfolgen ($\mathbf{z}_0, \mathbf{z}_1, \ldots$) mit $\mathbf{z}_0 = \mathbf{z}$ und $\mathbf{z}_i[\![\alpha]\!]\mathbf{z}_{i+1}$ ein n existiert, so daß $\mathcal{A}, \mathbf{z_n} \models \varphi$ gilt.

Zur Axiomatisierung betrachten wir für eine gegebene Algebra $\mathcal{A}$, festes α und φ die beiden Eigenschaften $AF_1(M)$ und $AF_2(M, \mathbf{z_0})$ für Mengen M von Zuständen und einen festen Ausgangszustand $\mathbf{z_0}$ durch

$$\begin{aligned}&AF_1(M) :\Leftrightarrow \text{Jeder Zustand } \mathbf{z} \text{ ist in } M, \text{ wenn } \mathcal{A}, \mathbf{z} \models \varphi \text{ gilt, oder}\\ &\text{wenn alle Nachfolgezustände } \mathbf{z}' \text{ (für die } \mathbf{z}[\![\alpha]\!]_{\mathcal{A},\mathbf{z}'} \text{ gilt) in } M \text{ sind}\end{aligned} \tag{6.21}$$

und

$$\begin{aligned}&AF_2(M, \mathbf{z_0}) :\Leftrightarrow \text{Jeder Zustand } \mathbf{z} \text{ ist in } M, \text{ wenn er von } \mathbf{z_0} \text{ aus}\\ &\text{erreichbar ist (d.h. auf einem bei } \mathbf{z_0} \text{ beginnenden Trace liegt), und}\\ &\text{wenn } \mathcal{A}, \mathbf{z} \models \varphi \text{ gilt oder alle Nachfolgezustände in } M \text{ sind}\end{aligned} \tag{6.22}$$

definierten Eigenschaften. Für sie gilt folgendes Theorem:

[1] Die Bezeichnung *AF* ist aus der Temporallogik entlehnt, siehe z.B. [Eme90]

Theorem 6 *Charakterisierung von* $AF(\alpha, \varphi)$
Die Menge der Zustände, in denen $AF(\alpha, \varphi)$ gilt, ist gleich dem Durchschnitt aller Mengen M, die die Eigenschaft $AF_1(M)$ haben. In einem Zustand $\mathbf{z_0}$ gilt $AF(\alpha, \varphi)$ genau dann, wenn er im Durchschnitt aller Mengen $AF_2(M, \mathbf{z_0})$ liegt.

Beweis von Theorem 6 Zum Beweis sei M_0 die Menge der Zustände, in denen $AF(\alpha, \varphi)$ gilt, $M_1 := \bigcap \{M \mid AF_1(M)\}$, $M_2(\mathbf{z_0}) := \bigcap \{M \mid AF_2(M,\mathbf{z_0})\}$. Dann gilt $AF_1(M_0)$, wie man sich leicht überzeugt, woraus sofort $M_1 \subseteq M_0$ folgt. Ausserdem gilt bei beliebigem $\mathbf{z_0}$, daß jede Menge M mit der Eigenschaft $AF_1(M)$ auch die Eigenschaft $AF_2(M,\mathbf{z_0})$ hat, da die Eigenschaft (6.21) die Eigenschaft (6.22) für jedes $\mathbf{z_0}$ impliziert. Daraus folgt sofort $M_2(\mathbf{z_0}) \subseteq M_1$. Um die Behauptung zu zeigen, bleibt also nur noch zu zeigen, daß jedes $\mathbf{z_0} \in M_0$ auch in $M_2(\mathbf{z_0})$ liegt. Angenommen, dies wäre nicht so. Dann folgt aus $\mathbf{z_0} \notin M_2(\mathbf{z_0})$, daß es eine Menge M mit $AF_2(M,\mathbf{z_0})$ gibt, die $\mathbf{z_0}$ nicht enthält. Daraus folgt nach (6.22), daß in $\mathbf{z_0}$ zum einen φ nicht gilt, zum anderen, daß es einen Nachfolgerzustand $\mathbf{z_1}$ gibt, der ebenfalls nicht zu M gehört. So fortfahrend, erhält man induktiv eine Folge von Zuständen $\mathbf{z_0}$, $\mathbf{z_1}$, ..., die alle nicht in M (aber von $\mathbf{z_0}$ aus erreichbar!) sind, und für die φ nicht gilt. Dies widerspricht aber der Annahme, daß $\mathbf{z_0} \in M_0$ ist. □

Aus der semantischen Definition von $AF(\alpha,\varphi)$ folgt unmittelbar, daß das Axiom

$$\mathrm{AF}(\alpha,\varphi) \leftrightarrow \varphi \vee [\alpha]\mathrm{AF}(\alpha,\varphi) \tag{6.23}$$

korrekt ist. Aus der Charakterisierung als Durchschnitt aller Mengen M mit $AF_1(M)$ ergibt sich die Gültigkeit des Axioms

$$(\forall \underline{\mathrm{x}}.\ ((\varphi \vee [\alpha]\psi) \rightarrow \psi)) \rightarrow (\mathrm{AF}(\alpha,\varphi) \rightarrow \psi) \tag{6.24}$$

Die Charakterisierung mit $AF_2(M,\mathbf{z})$ ergibt das stärkere Axiom

$$(\forall\ \mathrm{i}.\ [\textbf{loop}\ \alpha\ \textbf{times}\ \mathrm{i}]((\varphi \vee [\alpha]\psi) \rightarrow \psi)) \rightarrow (\mathrm{AF}(\alpha,\varphi) \rightarrow \psi) \tag{6.25}$$

Dieses gestattet es, die Zustände, in denen $(\varphi \vee [\alpha]\psi) \rightarrow \psi$ gezeigt werden muß, auf die vom Ausgangszustand aus erreichbaren einzuschränken. Die beiden Axiome (6.23) und (6.25) können als Axiomatisierung von $AF(\alpha, \varphi)$ verwendet werden, so daß auf die in (6.20) gegebene Definition des AF-Operators mit Hilfe von Strömen verzichtet werden kann.

Mit Hilfe des AF-Operators können nun auch für Diagramme indeterministischer Größe Beweisverpflichtungen für die Kommutativität aufgestellt werden. Dies geschieht einfach schematisch durch Ersetzen aller Formeln der Form „$\exists\ i.$

[**loop** α **times** i] φ“ durch $AF(\alpha,\varphi)$. Man erhält die folgenden Beweisverpflichtungen:

$$\begin{aligned}
&\mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{mn} \\
\rightarrow\ &\mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}'), \\
&\quad \exists\ \mathrm{i} > 0.\ \langle \textbf{loop if}\ \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x})\ \textbf{times}\ \mathrm{i}\rangle \\
&\qquad \mathrm{INV}(\underline{x}, \underline{x}'))
\end{aligned} \tag{6.26}$$

$$\begin{aligned}
&\mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{mn} \\
\rightarrow\ &\mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x}), \\
&\quad \exists\ \mathrm{j} > 0.\ \langle \textbf{loop if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}')\ \textbf{times}\ \mathrm{j}\rangle \\
&\qquad \mathrm{INV}(\underline{x}, \underline{x}'))
\end{aligned} \tag{6.27}$$

$$\begin{aligned}
&\mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{m0} \wedge \mathrm{execm0}(\underline{x}, \underline{x}') = \mathrm{k} \\
\rightarrow\ &\mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(;\underline{x}), \\
&\quad (\quad \mathrm{INV}(\underline{x}, \underline{x}') \\
&\qquad \wedge\ (\neg\ \mathrm{final}(\underline{x}) \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{m0} \rightarrow \mathrm{execm0}(\underline{x}, \underline{x}') < \mathrm{k})))
\end{aligned} \tag{6.28}$$

$$\begin{aligned}
&\mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{0n} \wedge \mathrm{exec0n}(\underline{x}, \underline{x}') = \mathrm{k} \\
\rightarrow\ &\mathrm{AF}(\textbf{if}\ \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(;\underline{x}'), \\
&\quad (\quad \mathrm{INV}(\underline{x}, \underline{x}') \\
&\qquad \wedge\ (\neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{0n} \rightarrow \mathrm{exec0n}(\underline{x}, \underline{x}') < \mathrm{k})))
\end{aligned} \tag{6.29}$$

Theorem 7 *Modularisierungstheorem für unbeschränkten Indeterminismus*
Sind eine Verfeinerung von ASM zu ASM′, ein Prädikat *INV* sowie Funktionen *ndt*, *exec0n*, *execm0* so gegeben, daß die Beweisverpflichtungen (6.26), (6.27), (6.28), (6.29), (6.8), (6.9), (6.10), (6.11) alle beweisbar sind, so ist die Verfeinerung von ASM zu ASM′ korrekt und vollständig:

$$\begin{aligned}
&(6.26) \wedge (6.27) \wedge (6.28) \wedge (6.29) \\
&\wedge\ (6.8) \wedge (6.9) \wedge (6.10) \wedge (6.11) \\
\Rightarrow\ &\mathrm{ASM} \bowtie \mathrm{ASM}'
\end{aligned}$$

Für Trace-Korrektheit sind (6.26), (6.28), (6.8), (6.9), (6.10), (6.11) sowie statt (6.29) das schwächere (6.7) hinreichend. Für Korrektheit sind dieselben Beweisverpflichtungen hinreichend, in (6.7) kann aber auf die Bedingung, daß *exec0n* abnimmt, verzichtet werden.

$$\begin{aligned}
&(6.26) \wedge (6.28) \wedge (6.7) \\
&\wedge\ (6.8) \wedge (6.9) \wedge (6.10) \wedge,(6.11) \\
\Rightarrow\ &\mathrm{ASM} \blacktriangleright \mathrm{ASM}'
\end{aligned}$$

An den formalen Beweisen in KIV ändert sich lediglich die Definition der Eigenschaften *KPROP* und *VPROP*:

KPROP($\underline{x}$, $\underline{x}'$) ≡
AF(**if** ¬ final′($\underline{x}'$) **then** RULE′(;$\underline{x}'$),
∃ i. ⟨**loop if** ¬ final($\underline{x}$) **then** RULE(;$\underline{x}$) **times** i⟩ INV($\underline{x}$, $\underline{x}'$))

VPROP($\underline{x}$, $\underline{x}'$) ≡
AF(**if** ¬ final($\underline{x}$) **then** RULE(;$\underline{x}$),
∃ j. ⟨**loop if** ¬ final′($\underline{x}'$) **then** RULE′(;$\underline{x}'$) **times** j⟩ INV($\underline{x}$, $\underline{x}'$))

sowie die Verwendung der Axiome (6.23) und (6.25) an Stelle der Axiome (4.3) für loops. Die formalen Beweise für Korrektheit und Vollständigkeit sind, da in KIV der *AF*-Operator derzeit nur als Abkürzung vorhanden ist, mit 466 Beweisschritten und 94 Interaktionen etwas aufwendiger als im deterministischen Fall. Die formale Spezifikation und die bewiesenen Theoreme und Lemmata finden sich in Anhang C.3.

Zum Schluß dieses Abschnitts noch einige weitere Bemerkungen zur Definition des *AF*-Operators: *AF* kann in DL nicht uniform (die Erweiterung von DL um Ströme ist nicht uniform, da der Stromdatentyp vom jeweiligen α abhängt) als Abkürzung eingeführt werden, da $AF(\alpha,\varphi)$ äquivalent ist zur Aussage: Das Programm *AF#*, definiert durch ($\underline{x}$ seien die in α vorkommenden Variablen):

AF#(;**var** $\underline{x}$)
begin
if φ **then**
  begin
  α;
  AF#(;$\underline{x}$)
  end
end

terminiert **immer**, i.e. für jeden Durchlauf. Die Aussage, daß ein indeterministisches Programm immer terminiert, kann aber in DL nicht ausgedrückt werden (siehe [Gol82]). Es gibt aber einen Spezialfall, bei dem dies doch geht:

Theorem 8 *beschränkter Indeterminismus*
Ist α ein immer terminierendes Programm, das nur *beschränkten* Indeterminismus aufweist, das also für jeden Anfangszustand $\mathbf{z}$ nur endlich viele Nachfolgerzustände $\mathbf{z}'$ mit $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ hat, so gilt:

$$AF(\alpha,\varphi) \leftrightarrow \exists\ \mathrm{j}.\ [\textbf{loop if } \neg\ \varphi \textbf{ then } \alpha \textbf{ times } \mathrm{j}]\ \varphi$$

Beweis von Theorem 8 Zum Beweis der Implikation von links nach rechts (die Umkehrung ist trivial) betrachtet man alle Zustandsfolgen von einem festen Startzustand $\mathbf{z}$ auf denen ständig $\neg\ \varphi$ gilt. Diese bilden eine Baumstruktur, die,

wegen der Voraussetzung $AF(\alpha,\varphi)$, keine unendlichen Pfade enthält. Wegen der Voraussetzung, daß α nur beschränkten Indeterminismus verwendet, ist der Baum endlich verzweigend. Nach König's Lemma aus der Mengenlehre (siehe z.B. [Knu73], S. 381–383) muß der Baum daher endlich sein. Die Länge jedes Pfads ist also durch die endliche Tiefe d des Baums beschränkt. Somit reicht $j := d + 1$ aus, um die Formel auf der rechten Seite der Äquivalenz wahrzumachen.

Immer terminierende Programme, die nur beschränkten Indeterminismus aufweisen, ergeben sich aus der Übersetzung von verteilten ASMs nach DL. Im Gegensatz zum Rateschritt einer natürlichen Zahl im Beispielprogramm vom Anfang dieses Abschnitts, der unendlich viele verschiedene Ergebnisse liefern kann, ist der in der Auswahl eines Agenten enthaltene Indeterminismus endlich verzweigend, da er immer nur aus einer endlichen Menge von Agenten auswählt. Somit wird die Erweiterung von DL um den AF-Operator in diesem Fall nicht benötigt.

Für die Beweisverpflichtungen bedeutet dies, daß die alten Beweisverpflichtungen aus dem vorigen Abschnitt beibehalten werden können. Lediglich die Tests $\neg$ *final*($\underline{x}$) bzw. $\neg$ *final'*($\underline{x}'$) der *loop*-Konstrukte müssen durch die komplizierteren Tests

$$\neg\ \mathrm{final}(\underline{\mathrm{x}}) \wedge \neg\ \varphi$$

(und analog für *final'*) ersetzt werden, wobei φ die Nachbedingung des *loop*-Konstrukts ist. Dabei wird ausgenutzt, daß wir beliebige Formeln in den Tests für conditionals zulassen.
Als Beispiel betrachten wir eine ASM′ mit Regel:

RULE′(**var** x):
if x = 0 **then choose** b **in**
if b **then** x := 3
else x:= 2
else if x > 1 **then** x := -1

und ASM′ wie zu Anfang des Abschnitts. ASM′ setzt im ersten Schritt nun x indeterministisch auf 2 oder 3 — es sind also nur endlich viele Wahlmöglichkeiten vorhanden. Daher genügt es für die Korrektheit

$$\begin{aligned}&\exists\ \mathrm{i}.\ [\mathbf{loop\ if}\quad \neg\ \mathrm{x} = 1\\ &\qquad\qquad \wedge \neg\ \exists\ \mathrm{j}.\ \langle \mathbf{loop\ if}\ \neg\ \mathrm{x}' = 1\ \mathbf{then}\ \mathrm{RULE}'(;\mathrm{x}')\rangle\ \mathrm{x} = \mathrm{x}'\\ &\quad \mathbf{then}\ \mathrm{RULE}(;\mathrm{x})]\\ &\qquad \exists\ \mathrm{j}.\ \langle \mathbf{loop\ if}\ \neg\ \mathrm{x}' = 1\ \mathbf{then}\ \mathrm{RULE}'(;\mathrm{x}')\rangle\ \mathrm{x} = \mathrm{x}'\end{aligned}$$

zu zeigen, was für $i = 3$ und $j = 1$ gelingt.

6.5 Erweiterungen für iterative Verfeinerung

In diesem Abschnitt befassen wir uns mit der Problematik, daß die systematische Übersetzung einer Programmiersprache in Assemblercode häufig mehrere

Verfeinerungen erfordert, die jeweils orthogonale Konzepte einführen. Nun zeigt es sich bei der Verifikation von zwei aufeinanderfolgenden Verfeinerungen ASM $\triangleright$ ASM$'$ $\triangleright$ ASM$''$ häufig, daß die beiden notwendigen Zusammenhangsinvarianten *INV* und *INV'* große gemeinsame Teile enthalten (wir werden Beispiele aus der Prolog-WAM-Fallstudie in den Abschnitten 17.2 und 18 behandeln). Diese gemeinsamen Teile bestehen aus Eigenschaften von ASM$'$, die für beide Äquivalenzbeweise relevant sind. Ist $MINV'(\underline{x}')$ ein gemeinsamer Teil von *INV* und *INV'*, so verlangt die bisherige Beweismethode, daß *MINV'* in beiden Äquivalenzbeweisen als invariant nachgewiesen wird. In diesem Abschnitt zeigen wir eine generische Methode, die es erlaubt, diese Beweisverdopplung zu vermeiden. Wir nehmen dazu an, daß die Äquivalenz von ASM und ASM$'$ mit einer Zusammenhangsinvariante *INV* bewiesen wurde. Dann überlegt man leicht, daß die Formel

$$\exists \underline{x}.\ \mathrm{INV}(\underline{x}, \underline{x}') \tag{6.30}$$

in allen Zuständen von ASM$'$ gilt, die „Eckzustände“ der einzelnen kommutierenden Diagramme in der Verfeinerung. Nun lassen sich diese Eckzustände meist einfach durch ein Prädikat $MINVNOW'(\underline{x}')$, das aus einer Disjunktion von ASM$'$-Regeltests besteht, charakterisieren. Somit kann als Invariante MINV$'$ von ASM$'$ die Formel

$$\mathrm{MINVNOW}'(\underline{x}') \rightarrow \exists \underline{x}.\ \mathrm{INV}(\underline{x}, \underline{x}') \tag{6.31}$$

gewählt werden. Da jede schwächere Formel ebenfalls eine Invariante ist, wird man in der Regel eine Formel wählen die von (6.31) impliziert wird, und keine Referenz auf die Variablen $\underline{x}$ von ASM mehr enthält.

Um sicherzustellen, daß *MINVNOW'* die Eckzustände von Diagrammen tatsächlich charakterisiert, müssen die Bedingungen für den Korrektheitsbeweis von ASM zu ASM$'$ verschärft werden (der Vollständigkeitsbeweis ist nicht betroffen). Wir zeigen, wie dies im indeterministischen Fall ohne Diagramme indeterministischer Größe aussieht. Der Spezialfall deterministischer Diagramme (Diamonds statt Boxen) sowie die Verallgemeinerung zu Diagrammen indeterministischer Größe (AF-Operator statt Boxen) sind genau wie in den vorherigen Abschnitten.

Die entscheidende Änderung ist, den Regeltest für ASM$'$ um die Zusatzbedinung $\neg$ $MINVNOW'(\underline{x}')$ zu verschärfen, und in der Nachbedingung $MINVNOW'(\underline{x}')$ zu fordern. Dadurch wird sichergestellt, daß ASM-Regeln genau solange angewandt werden, wie $\neg$ $MINVNOW'(\underline{x}')$ gilt. Für m:n-Diagramme und 0:n-Diagramme ergeben sich so statt der Bedingungen (6.14) und (6.17) nun

$$
\begin{array}{l}
\quad \mathrm{INV}(\underline{x},\underline{x}') \wedge \neg\ \mathrm{final}(\underline{x}) \wedge \mathrm{MINVNOW}'(\underline{x}') \wedge \neg\ \mathrm{final}'(\underline{x}') \\
\wedge\ \mathrm{ndt}(\underline{x}, \underline{x}') = \mathrm{mn} \\
\rightarrow [\textbf{if} \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(\underline{x}')\ ] \\
\quad \exists\ \mathrm{j}.\ [\textbf{loop if} \neg\ \mathrm{final}'(\underline{x}') \wedge \neg\ \mathrm{MINVNOW}'(\underline{x}') \\
\qquad \textbf{then}\ \mathrm{RULE}'(\underline{x}')\ \textbf{times}\ \mathrm{j}] \\
\qquad (\quad \mathrm{MINVNOW}'(\underline{x}') \\
\qquad \wedge\ \exists\ \mathrm{i} > 0.\ \langle \textbf{loop if} \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(\underline{x})\ \textbf{times}\ \mathrm{i}\rangle \\
\qquad\qquad \mathrm{INV}(\underline{x}, \underline{x}'))
\end{array}
\tag{6.32}
$$

$$
\begin{array}{l}
\quad \mathrm{INV}(\underline{x}, \underline{x}') \wedge \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{MINVNOW}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \\
\wedge\ \mathrm{exec0n}(\underline{x}, \underline{x}') = \mathrm{k} \\
\rightarrow [\textbf{if} \neg\ \mathrm{final}'(\underline{x}')\ \textbf{then}\ \mathrm{RULE}'(\underline{x}')] \\
\quad \exists\ \mathrm{j} > 0.\ [\textbf{loop if} \neg\ \mathrm{final}'(\underline{x}') \wedge \neg\ \mathrm{MINVNOW}'(\underline{x}') \\
\qquad \textbf{then}\ \mathrm{RULE}'(\underline{x}')\ \textbf{times}\ \mathrm{j}] \\
\qquad\quad (\quad \mathrm{MINVNOW}'(\underline{x}') \wedge \mathrm{INV}(\underline{x}, \underline{x}') \\
\qquad\quad \wedge\ (\quad \neg\ \mathrm{final}'(\underline{x}') \wedge \mathrm{ndt}(\underline{x}, \underline{x}') = 0\mathrm{n} \\
\qquad\qquad \rightarrow \mathrm{exec0n}(\underline{x}, \underline{x}') < \mathrm{k}))
\end{array}
\tag{6.33}
$$

Die Beweisverpflichtung für m:0-Diagramme (6.6) bleibt unverändert. Mit dieser Änderung der Beweisverpflichtungen läßt sich zeigen:

Theorem 9 *Iterative Verfeinerung*
Aus den Beweisverpflichtungen (6.32), (6.33),(6.6) (6.8), (6.9), (6.10) und (6.11) folgt neben der Korrektheit und Trace-Korrektheit der Verfeinerung von ASM zu ASM′ auch, daß jede Formel $MINV'(\underline{x}')$, für die

$$(\mathrm{MINVNOW}'(\underline{x}') \rightarrow \exists \underline{x}.\ \mathrm{INV}(\underline{x}, \underline{x}')) \rightarrow \mathrm{MINV}'(\underline{x}')$$

gilt, eine Invariante von ASM′ ist, genauer gilt für jede Formel mit dieser Eigenschaft:

$$
\begin{array}{l}
\quad (\exists\ \mathrm{st}.\ \mathrm{IN}(\mathrm{st}, \mathrm{st'})) \\
\rightarrow \forall\ \mathrm{j}.\ [\textbf{loop if} \neg\ \mathrm{final'(st')}\ \textbf{then}\ \mathrm{RULE}'(;\ \mathrm{st'})\ \textbf{times}\ \mathrm{j}]\ \mathrm{MINV}'(\underline{x}')
\end{array}
$$

MINV′($\underline{x}'$) gilt also für alle Zustände, die während der Abarbeitung von ASM′ angenommen werden, sofern nur der Anfangszustand per *IN*-Relation zu einem Zustand von ASM korreliert ist (eine üblicherweise triviale Annahme). Der Beweis des Theorems ergibt sich direkt aus dem bisherigen Korrektheitsbeweis. Lediglich die Definition von *KPROP* ist zu

$$
\begin{array}{l}
\mathrm{KPROP}(\underline{x}, \underline{x}') \equiv \\
\exists\ \mathrm{j}.\ [\textbf{loop if} \neg\ \mathrm{final}'(\underline{x}') \wedge \neg\ \mathrm{MINVNOW}'(\underline{x}') \\
\quad \textbf{then}\ \mathrm{RULE}'(\underline{x}')\ \textbf{times}\ \mathrm{j}] \\
\quad (\quad \mathrm{MINVNOW}'(\underline{x}') \\
\quad \wedge\ \exists\ \mathrm{i}.\ \langle \textbf{loop if} \neg\ \mathrm{final}(\underline{x})\ \textbf{then}\ \mathrm{RULE}(\underline{x})\ \textbf{times}\ \mathrm{i}\rangle \\
\qquad \mathrm{INV}(\underline{x}, \underline{x}'))
\end{array}
\tag{6.34}
$$

abzuändern. Aus der Invarianz von *KPROP* folgt unmittelbar, daß

$$\text{MINVNOW}'(\underline{x}') \rightarrow \exists \underline{x}.\ \text{INV}(\underline{x}, \underline{x}')$$

eine Invariante von ASM′ ist, und somit erst recht die nach Voraussetzung schwächere Formel MINV′.

MINV′ kann also in den Beweisverpflichtungen für die Verfeinerung von ASM′ zu ASM″ als Voraussetzung addiert werden. Das Verfahren, Invarianten für die Maschinen zu gewinnen, läßt sich nun natürlich iterativ fortsetzen: Man definiert für die Verfeinerung von ASM′ zu ASM″ erneut ein Prädikat MINVNOW″, und erhält so wieder eine Invariante MINV″ für ASM″, die für eine weitere Verfeinerung genutzt werden kann.

Anhang C.4 gibt eine Formalisierung des Korrektheitsbeweises für den Fall, daß bereits für ASM eine Maschineninvariante $MINV(\underline{x})$ vorliegt. Der Korrektheitsbeweis erforderte 502 Beweisschritte und 89 Interaktionen. Die oben dargestellten Beweisverpflichtungen ergeben sich als Spezialfall aus den im Anhang genannten für den Fall, in dem keine Invariante für ASM gegeben ist (i.e. einfach $MINV(\underline{x}) = \text{true}$ gesetzt wird).

6.6 Verwandte Arbeiten

Die meisten bekannten Arbeiten zum Äquivalenznachweis von ASMs stammen aus der Compilerverifikation. Die betrachteten Interpreter sind dabei zwar meistens nicht im ASM-Formalismus beschrieben, die verwendeten Formalismen sind aber meist dazu äquivalent.

Der Fall von 1:1-Diagrammen ist in den Arbeiten zur Compilerverifikation der bei weitem häufigste, und oft werden verschiedene Varianten diskutiert, in denen *IN*, *OUT* und *INV* Funktionen in die eine oder andere Richtung sind (z.B. in [BHMY89]). An Verallgemeinerungen wird häufig noch die Verfeinerung sequentieller ASMs durch 1:n-Diagramme mit $n > 0$ betrachtet. Dieser Fall ergibt sich häufig dadurch, daß eine Instruktion der Ausgangssprache in mehrere Instruktionen der Zielsprache übersetzt wird. Der Fall ist nur wenig allgemeiner als Data Refinement, da Induktion über die Anzahl der Regelschritte von ASM direkt zum Ziel führt.

Ein Beispiel zur formalen Verifikation eines Compilers, in dem 1:n-Diagramme vorkommen, ist die Verifikation der Übersetzung einer imperativen Programmiersprache (GYPSY), die in mehreren Verfeinerungen zunächst in eine High-Level Assembler Sprache (Piton) und dann in Maschinencode des FM8502-Prozessors übersetzt wurde. Die mit NQTHM ([BM79], [BM88]) durchgeführte Verifikation ist in ([Moo88],[You88]) beschrieben. Da es in NQTHM keine Existenzquantifikation gibt, wurde die Anzahl der Schritte n von ASM′ die m Schritten von ASM entspricht, durch eine Skolemfunktion $n = clock(m, st_0)$ ausgedrückt (st_0 ist der Initialzustand von ASM).

Eine derartige Skolemfunktion (*num_non_visible*) wird auch in [Cyr93] verwendet. Der dortige Korrektheitsbegriff ist Trace-Korrektheit für sequentielle ASMs bezüglich einer Abstraktionsfunktion *abstr*, wobei alle Zustände der abstrakten ASM sichtbar sind, was einer Beschränkung der möglichen Diagramme auf 1:n mit $n > 0$ entspricht. Das Papier skizziert 2 Beweistechniken. Die erste („Speeding up the Implementation Machine“) entspricht der direkten Verifikation der 1:n-Diagramme mit einer Zusammenhangsinvariante

$$\mathrm{INV}(\underline{x},\underline{x}') \equiv \mathrm{visible}(x') \rightarrow \mathrm{abstr}(\underline{x}') = \underline{x}$$

Die dabei verwendete *visible_I*-Funktion, die *num_non_visible*-viele Schritte von ASM′ in einen Schritt codiert, entspricht genau unserem

loop if $\neg$ final′($\underline{x}'$) **then** RULE′(;$\underline{x}'$) **times** num_non_visible($\underline{x}'$)

Die zweite Beweistechnik („Slowing down the Specification Machine“) zerlegt die 1:n-Diagramme in ein 1:1 und n-1 1:0-Diagramme („Stutter-Schritte“), die getrennt verifiziert werden. Die „Termination“-Bedingung entspricht der Abnahme der *exec0n*-Funktion. Der Ansatz scheint aber die explizite Einführung von Zeit in die Spezifikation zu erfordern. Der Ausblick von [Cyr93] gibt als wünschenswerte Erweiterungen Nichtdeterminismus, Stuttering von beiden Maschinen (also 0:n und m:0-Diagramme), sowie iterative Verfeinerung („hierarchical Decomposition“) an, die wir hier alle behandelt haben.

Beliebige m:n Diagramme werden skizzenhaft in [McG72] betrachtet. Das Papier setzt Determinismus voraus, und betrachtet nur m:n-Diagramme mit $m, n > 0$. Außerdem wird vorausgesetzt, daß die Zusammenhangsinvariante $INV(\underline{x}, \underline{x}')$ die spezielle Form $f_1(\underline{x}) = f_2(\underline{x}')$ hat.

Eine formale Behandlung von m:n-Diagrammen mit $m, n > 0$ ist in dem parallel zu dieser Arbeit entstandenen Papier [Dol98] zu finden. Darin wird der Ansatz von [Cyr93] durch die Verwendung von zwei *num_non_visible*-Funktionen (je eine pro ASM) ausgedehnt. Indeterminismus wird behandelt, allerdings nur beschränkter Indeterminismus (für unbeschränkten Indeterminismus wie in dem in Abschnitt 6.4.2, S. 51 gezeigten Beispiel läßt sich keine *num_non_visible*-Funktion definieren). Auch wird weiterhin eine Abstraktionsfunktion zugrundegelegt.

Eine weitere neue Arbeit zur Verfeinerung von ASMs in der Compilerverifikation ist [ZG97]. Der dort definierte Korrektheitsbegriff ist semantisch definiert (er gibt keine Logik zur formalen Verifikation), verwendet aber als einziger uns bekannter Ansatz eine *Relation* ρ statt einer Abstraktionsfunktion zwischen den Zuständen der beiden ASMs. Die Relation entspricht der Semantik unserer Zusammenhangsinvariante *INV*.

Der Korrektheitsbegriff beruht auf der Gleichheit (modulo Abstraktionsfunktion) der während eines Ablaufs gemachten Ausgaben. Ausgaben werden dabei implizit als Änderungen von Ausgabevariablen definiert. Um diesen Korrektheitsbegriff in unserem Ansatz formal nachzubilden, ist es notwendig, die

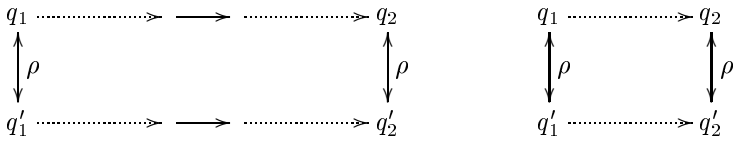


Abbildung 6.9 : Modularisierung nach [ZG97]

ASM so abzuändern, daß sie die Ausgaben explizit in einer Liste *outputlist* aufsammelt (in Sinne von [AL91] führen wir also eine History-Variable ein). Der Korrektheitsbegriff in [ZG97] ist dann äquivalent zu Trace-Korrektheit mit

$\mathrm{IN}(\underline{x},\underline{x}') \equiv \mathrm{outputlist} = \mathrm{outputlist}' = []$
$\mathrm{OUT}(\underline{x},\underline{x}') \equiv \mathrm{map(abstr,outputlist}')= \mathrm{outputlist}$

(dies entspricht den in Theorem 4 gemachten Bedingungen an die Relation ρ). [ZG97] gibt ebenfalls ein Modularisierungstheorem (Theorem 5, „Horizontal Decomposition“) an. Dessen Idee ist es ebenfalls, den Gesamtablauf in Teildiagramme zu zerlegen. Dabei wird gefordert, daß jedes Diagramm höchstens eine Ausgabeaktion hat. Stellt man einen Zustandsübergang, der eine Ausgabe erzeugt, als durchgezogenen Pfeil, eine beliebige Zahl von „stotterden“ Zustandsübergängen, die keine Ausgabe erzeugen, als gepunkteten Pfeil dar, so lassen sich die nach dem Theorem zu verifizierenden Teildiagramme aus Abb. 6.9 veranschaulichen.

Allerdings ist das Theorem nicht korrekt. Zum einen können mit dem Theorem inkorrekte Verfeinerungen mit unendliche Folgen von m:0-Diagrammen wie in Abb. 6.6 (s. Abschnitt 6.2.2) als korrekt nachgewiesen werden, zum anderen fehlen einige implizit gemachte Voraussetzungen. Schließlich sind durch die Formalisierung (versehentlich) 1:n-Diagramme mit $n > 1$ ausgeschlossen.

Die zwar in den betrachteten Beispielen eingehaltenen, aber nicht explizit gemachten Voraussetzungen sind, daß durch externe Funktionen kein unbeschränkter Indeterminismus verursacht wird, sowie daß Ausgaben wie oben in einer *outputlist* gesammelt werden. Ohne diese Voraussetzungen lassen sich die in Abb. 6.10 und in Abb. 6.11 angegebenen Gegenbeispiele konstruieren: die Bilder zeigen die ASMs als Automaten mit zwei Programmvariablen. Die erste speichert den internen Automatenzustand, die zweite die momentane Ausgabe. Die Pfeile entsprechen den möglichen Zustandsübergangen durch Anwendung passender ASM-Regeln. Abb. 6.11 zeigt die unangenehmen Möglichkeiten des unbeschränkten Indeterminismus, der uns in Abschnitt 6.4 zur Einführung des AF-Operators gezwungen hat. Abb. 6.10 nutzt aus, daß aus der Möglichkeit eines Zustandsübergangs von q_1' nach q_2' mit einer Ausgabe nicht folgt, daß auf allen Pfaden von q_1' nach q_2' ebenfalls genau eine Ausgabe stattfindet.

m:n-Diagramme mit $n > 1$ (also insbesondere die oft vorkommenden 1:n-Diagramme) sind durch die Formalisierung ausgeschlossen, da gefordert wird, daß die Kommutierung der Diagramme aus Abb. 6.9 für *jedes* q_2' (und also

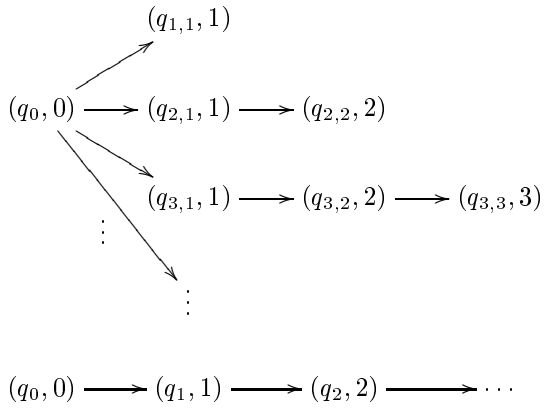


Abbildung 6.10 Inkorrekte Verfeinerung mit unbeschränktem Indeterminismus

Abbildung 6.11 Inkorrekte Verfeinerung ohne *outputlist*

insbesondere für jeden direkten Nachfolger jedes ASM2-Zustands q_1') gezeigt werden muß, statt nur für *irgendeinen* Nachfolger auf jedem bei q_1' beginnenden Pfad, wie dies unser Theorem fordert.

Schließt man neben den oben beschriebenen impliziten Voraussetzungen des Theorems zusätzlich noch unendlichen Folgen von m:0-Diagrammen aus, so läßt sich zeigen, daß Theorem 5 aus [ZG97] ein Spezialfall von Theorem 5, S. 50 ist: Die Problematik unendlicher Folgen von 0:n-Diagrammen tritt nicht auf, da nur solche zugelassen sind, die sich zu einem 1:n-Diagramm erweitern lassen: somit kann immer $ndt(\underline{x}, \underline{x}') \neq 0n$ gewählt werden.

Kapitel 7

Peephole Optimierung

Als eine Anwendung des Haupttheorems zum Äquivalenznachweis von ASM-Verfeinerungen wird in in diesem Abschnitt die sogenannte „Peephole Optimierung“ von Programmcode (meist wird Assemblercode betrachtet) gezeigt. Die Idee einer derartigen Optimierung besteht darin, mit einem Fenster beschränkter Größe (peephole) über ein Stück Programmcode zu wandern, und dabei ineffiziente Instruktionsfolgen durch effizientere zu ersetzen.

Abschnitt 7.1 gibt zunächst einen allgemeinen Ansatz zur Verifikation für den Fall, daß das zu optimierende Codestück keine Sprunginstruktionen enthält (der Gesamtcode darf aber Sprünge enthalten). Es wird gezeigt, daß sich die für die Korrektheit notwendigen Bedingungen einfach durch geeignete Instanzierung des generischen Modularisierungstheorems für ASM-Verfeinerungen ergibt.

Die Idee eines allgemeinen Ansatzes zur Verifikation von Peephole Optimierungen zu definieren, ist aus [DvHPR97] übernommen. Das Papier besteht aus 2 Teilen: Im ersten Teil wird eine Formalisierung von Peephole Optimierung gegeben, und gezeigt, daß gewisse Beweisverpflichtungen für die Korrektheit hinreichen. Im zweiten Teil werden dann eine Reihe Anwendungsbeispiele untersucht, die aus [TvS82] stammen.

Abschnitt 7.2 zeigt, daß der gegebene Ansatz den in [DvHPR97] definierten verallgemeinert. Zwar sind beide Ansätze in dem Sinn generisch, als sie beide unabhängig vom konkreten Code ist. Allerdings wird der Programmcode in [DvHPR97] als Liste von Instruktionen aufgefaßt, was insofern unrealistisch ist, als realer Assemblercode immer auch Sprünge enthält. Die Beschränkung auf linearen Code ohne Sprünge läßt sich in der in [DvHPR97] gegebenen Formalisierung auch nicht auf einfache Weise beseitigen, da der Beweis der Korrektheit essentiell auf einem Induktionsbeweis über die Länge der Codefolge beruht.

Im Gegensatz dazu zeigen wir in Abschnitt 7.3, daß die Beispiele aus [TvS82], die in [DvHPR97] wegen vorhandener Sprunganweisungen nicht betrachtetet werden konnten, sich durch eine minimale Modifikation der Zusammenhangsinvarianten ebenfalls lösen lassen. Der Grund dafür ist, daß die Beispiele alle die Besonderheit aufweisen, daß immer nur die *letzte* Anweisung einer optimierten Anweisungsfolge eine Sprunganweisung ist. Falls Sprünge in der Mitte von

optimierten Anweisungsfolgen liegen, müssen die als kommutierend nachzuweisenden Diagramme an diesen Anweisungen aufgespalten werden, ohne daß sich ansonsten viel ändert. Wir zeigen dies abschließend an einem einfachen Beispiel.

7.1 Formalisierung von Peephole Optimierung

Wir benötigen zunächst die Formalisierung eines allgemeinen Interpreters als ASM. Dazu nehmen wir an, daß der Programmcode in einem Speicher *db* (wir betrachten keinen selbst modifizierenden Code, *db* ist also eine Konstante) gespeichert ist, und mit *code(pc,db)* der Code an der in einem Programmzähler *pc* gespeicherten Adresse aus *db* ausgelesen werden kann. Schließlich benötigen wir eine ASM-Regel *RULE*, die eine Instruktion $i = code(pc,db)$ ausführt, und dabei einen Programmzustand *st* sowie den Programmzähler *pc* verändert. Um die fehlerhafte Ausführung von Instruktionen (etwa Division durch Null, oder den Versuch ein Element von einem leeren Stapel zu nehmen) erkennen zu können, definieren wir ein Prädikat *ok(pc,st,db)*, das besagt, daß die Ausführung der nächsten Instruktion *code(pc,db)* zu keinem Fehler führen wird. Wir nehmen an, daß Fehlerzustände Endzustände sind, in denen die ASM-Regel nicht anwendbar ist. Das reguläre Programmende wird durch eine ausgezeichnete Instruktion *halt* erreicht.

Da wir Sprunganweisungen mitbetrachten wollen, verlangen wir nicht, daß jede Regelanwendung *pc* um eins erhöht. Dennoch spielen solche Anweisungen, die wir im folgenden *linear* nennen, eine wichtige Rolle. Wir definieren deshalb die folgenden Hilfsfunktionen und -prädikate:

$$\text{instrs(pc,db,n)} = [\text{code(pc,db)}, \ldots, \text{code(pc} + \text{n} - 1\text{ ,db)}]$$

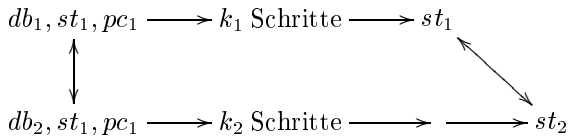
$$\begin{aligned} &\text{lin(i)} \\ \leftrightarrow\ &\forall\ \text{pc,pc}_0\text{,db,st.} \quad \text{code(pc,db)} = \text{i} \wedge \text{pc} = \text{pc}_0 \wedge \text{ok(pc,st,db)} \\ &\qquad \rightarrow \langle \text{RULE\#(db;pc,st)} \rangle\ \text{pc} = \text{pc}_0 + 1 \end{aligned}$$

$$\text{linear(pc,db,n)} \leftrightarrow \forall\ \text{k.}\ 0 \leq \text{k} < \text{n.}\ \text{lin(pc} + \text{k,db)}$$

instrs(pc,db,n) berechnet die Liste der n auf *pc* folgenden Anweisungen. *lin(i)* besagt, daß die Instruktion i linear ist, ihre Ausführung also in jedem Zustand zu einer Erhöhung des *pc* um eins führen wird. *linear(pc,db,n)* besagt, daß alle Instruktionen in *instrs(pc,db,n)* linear sind, also beim Ablauf der ASM der Reihe nach abgearbeitet werden. Derartige Instruktionsfolgen werden durch Peephole Optimierung durch effizientere ersetzt.

Für die Definition der Peephole Optimierung definieren wir nun ein Prädikat *peephole*(st_1, pc_1,db_1,k_1, il_2), das in einem Zustand db_1,st_1,pc_1 der ASM zutreffen sollte, wenn sich die nun abzuarbeitende Instruktionenfolge *instrs*(db_1,pc_1,k_1)

äquivalent durch die Instruktionenfolge il_2 ersetzen läßt. Bezeichnet k_2 die Länge von il_2, so entspricht diese Forderung intuitiv der Kommutativität des k_1:k_2 Diagramms



In Dynamischer Logik formalisiert bedeutet die Forderung, daß

$$
\begin{array}{ll}
 & \mathrm{I(db_1,st_0,pc_0)} \\
\wedge & \exists\ \mathrm{i.}\ \langle \textbf{loop}\ \mathrm{RULE(db_1;pc_0,st_0)}\ \textbf{times}\ \mathrm{i}\rangle\ (\mathrm{pc_0} = \mathrm{pc_1} \wedge\ \mathrm{st_0} = \mathrm{st_1}) \\
\wedge & \mathrm{peephole(st_1,pc_1,db_1,k_1,il_2)} \\
\wedge & \mathrm{db_2} = \mathrm{repl(pc_1,db_1,k_1,\ il_2)} \\
\wedge & \mathrm{pc_1} = \mathrm{pc_2} \wedge \mathrm{st_1} = \mathrm{st_2} \\
\rightarrow & \quad \mathrm{linear(instrs(pc_1,db_1,k_1))} \\
\wedge & \mathrm{linear(il_2)} \\
\wedge & \langle \textbf{loop}\ \mathrm{RULE(db_1;pc_1,st_1)}\ \textbf{times}\ \mathrm{k_1}\rangle \\
 & \quad \langle \textbf{loop}\ \mathrm{RULE(db_2;pc_2,st_2)}\ \textbf{times}\ \mathrm{k_2}\rangle\ \mathrm{st_1} = \mathrm{st_2}
\end{array}
\tag{7.1}
$$

gelten muß. Die Formel

$$
\begin{array}{ll}
 & \mathrm{I(db_1,st_0,pc_0)} \\
\wedge & \exists\ \mathrm{i.}\ \langle \textbf{loop}\ \mathrm{RULE(db_1;pc_0,st_0)}\ \textbf{times}\ \mathrm{i}\rangle\ (\mathrm{pc_0} = \mathrm{pc_1} \wedge \mathrm{st_0} = \mathrm{st_1})
\end{array}
$$

besagt dabei, daß der betrachtete Zustand (pc_1,st_1) aus einem durch ein Prädikat I spezifizierten Initialzustand (st_0,pc_0) erreichbar ist. Diese Annahme ist häufig unnötig, stattdessen können auch einfach alle Zustände (pc_1,st_1) betrachtet werden.

Die Linearitätsbedingungen an $instrs(pc_1,db_1,k_1)$ bzw. il_2 sorgen dafür, daß die Instruktionen auch tatsächlich ausgeführt werden. $repl(pc_1,db_1,k_1,il_2)$ ersetzt die Instruktionen $instrs(pc_1,db_1,k_1)$ durch il_2. Es gilt also:

$$
\begin{array}{ll}
 & \mathrm{db_2} = \mathrm{repl(pc_1,db_1,k_1,\ il_2)} \\
\rightarrow & \forall\ \mathrm{k.\ k} < \mathrm{k_2} \rightarrow \mathrm{code(pc_1{+}k,db_2)} = \mathrm{get(k,il_2)}
\end{array}
\tag{7.2}
$$

Diese Definition von *repl* ist allerdings noch nicht ausreichend. Durch Codeverschiebung um $k_2 - k_1$ muß noch dafür gesorgt werden, daß keine Lücken im Code entstehen und Sprungadressen korrekt angepaßt werden. Um nicht auf konkrete Details von Sprungcode eingehen zu müssen, wird deshalb für *repl* gefordert, daß jede verschobene Instruktion bei $pc' = shift(pc,pc_1, k_2 - k_1)$ denselben Effekt wie die Originalinstruktion bei pc haben soll:

$$\begin{aligned} & \mathrm{db}_2 = \mathrm{repl}(\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\mathrm{il}_2) \\ \wedge\ & (\mathrm{pc} < \mathrm{pc}_1 \vee \mathrm{pc} \geq \mathrm{pc}_1 + \mathrm{k}_1) \\ \wedge\ & \mathrm{pc}' = \mathrm{shift}(\mathrm{pc},\mathrm{pc}_1,\mathrm{k}_2 - \mathrm{k}_1) \wedge \mathrm{st} = \mathrm{st}' \\ \rightarrow\ & \langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st}) \rangle\ \langle \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}',\mathrm{st}') \rangle \\ & \qquad (\mathrm{pc}' = \mathrm{shift}(\mathrm{pc},\mathrm{pc}_1,\ \mathrm{k}_2 - \mathrm{k}_1) \wedge \mathrm{st} = \mathrm{st}') \end{aligned} \tag{7.3}$$

Dabei ist *shift* durch

$$\mathrm{shift}(\mathrm{pc}, \mathrm{pc}_1, \mathrm{n}) = \begin{cases} \mathrm{pc}, \text{ falls } \mathrm{pc} < \mathrm{pc}_1 \\ \mathrm{pc} + \mathrm{n}, \text{ sonst} \end{cases}$$

definiert.

Für die Anwendbarkeit einer Peephole Optimierung fordern wir, daß für konkrete Werte von db_1, pc_1 und il_2 in jedem Zustand *st*, den die ASM erreichen kann, $peephole(st,pc,db_1,k_1,il_2)$ gelten muß. Formal:

$$\begin{aligned} & \mathrm{IN}(\mathrm{db}_1,\mathrm{pc}_0,\mathrm{st}_0) \\ \wedge\ & \exists\ \mathrm{i}.\ \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db};\mathrm{pc}_0,\mathrm{st}_0)\ \textbf{times}\ \mathrm{i} \rangle\ (\mathrm{pc}_0 = \mathrm{pc}_1 \wedge \mathrm{st}_0 = \mathrm{st}_1) \\ \rightarrow\ & \mathrm{peephole}(\mathrm{st}_1,\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\mathrm{il}_2)) \end{aligned} \tag{7.4}$$

Die Bedingung (7.1) gibt eine Bedingung für die Optimierung einer Instruktionenfolge. Die Bedingung ist *lokal*, da in db_1 nur die Anweisungsfolge zwischen pc_1 und $pc_1 + k_1$ eine Rolle spielen. Für Programmcode, der keine Sprunganweisungen enthält, ist die Bedingung bereits ausreichend, um die Ersetzbarkeit der ersten Anweisungsfolge durch die zweite in *jedem* Programm zu garantieren. Für Programme mit Sprüngen benötigen wir offenbar eine Zusatzbedingung: Keine Sprunganweisung darf in den optimierten Code führen. Dies kann durch ein Prädikat *notjumpedto* formalisiert werden:

$$\begin{aligned} & \mathrm{notjumpedto}(\mathrm{pc}_1,\mathrm{k}_1,\mathrm{db}) \\ \leftrightarrow\ & \forall\ \mathrm{st},\ \mathrm{pc}. \quad \neg\ \mathrm{pc}_1 \leq \mathrm{pc} < \mathrm{pc}_1 + \mathrm{k}_1 \\ & \qquad \rightarrow \langle \mathrm{RULE}(\mathrm{db};\mathrm{pc},\mathrm{pc}) \rangle\ \neg\ \mathrm{pc}_1 < \mathrm{pc} < \mathrm{pc}_1 + \mathrm{k}_1 \end{aligned} \tag{7.5}$$

Damit läßt sich nun das folgende Theorem zeigen:

Theorem 10 Gegeben seien eine ASM, ein Prädikat *peephole*, sowie Werte db_1, pc_1, k_1, il_2 so daß (7.1), (7.4) und $notjumpedto(pc_1,k_1,db_1)$ erfüllt sind. Dann ist die Ersetzung von db_1 durch $repl(db_1,pc_1,k_1,il_2)$ (*repl* wie in (7.2) und (7.3) definiert) eine korrekte und vollständige Verfeinerung der ASM.

Für den Beweis zerlegen wir die Abläufe der nichtoptimierten ASM auf db_1 und der optimierten ASM Code $db_2 = repl(db_1,pc_1,k_1,il_2)$ in 1:1-Diagramme, solange $pc \neq pc_1$ ist, sowie in ein $k_1{:}k_2$-Diagramm für den optimierten Code.

Als Zusammenhangsinvariante *INV(pc,st,pc',st')* ergibt sich eine Konjunktion aus den vier Formeln:

$$\exists\ \mathrm{pc}_0,\mathrm{st}_0,\mathrm{i}.\quad \mathrm{I}(\mathrm{db}_0,\mathrm{pc}_0,\mathrm{st}_0) \wedge \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_0;\mathrm{pc}_0,\mathrm{st}_0)\ \textbf{times}\ \mathrm{i}\rangle (\mathrm{pc}_0 = \mathrm{pc}_1 \wedge \mathrm{st}_0 = \mathrm{st}_1)$$

$$\mathrm{db}_2 = \mathrm{repl}(\mathrm{db}_1,\mathrm{pc}_1,\mathrm{k}_1,\mathrm{il}_2)$$

$$\neg\ \mathrm{pc}_1 < \mathrm{pc} < \mathrm{pc}_1 + \mathrm{k}_1$$

$$\mathrm{pc}' = \mathrm{shift}(\mathrm{pc},\mathrm{pc}_1,\mathrm{k}_2 - \mathrm{k}_1) \wedge \mathrm{st}' = \mathrm{st}$$

Diese vier Formeln müssen entsprechend den Beweisverpflichtungen für die Äquivalenz der beiden ASMs aus Kapitel 6 als invariant in dem folgenden k_1:k_2-Diagramm nachgewiesen werden, wenn $pc = pc_1$ gilt, und als invariant in dem folgenden 1:1-Diagramm, wenn $pc \neq pc_1$ gilt.

Für die ersten beiden Formeln ist dies einfach, da die erste Formel eine triviale Invariante der nicht optimierten ASM darstellt. Sie besagt ja lediglich, daß jeder Zwischenzustand aus einem Initialzustand erreicht wird.

Die zweite Formel ist einfach die Compilerannahme zwischen den Programmcodes. Sie ist natürlich ebenfalls invariant, da sie keine durch die ASM veränderten Werte enthält.

Die dritte Formel besagt, daß sich *pc* nicht *innerhalb* des optimierten Abschnittes befindet ($pc = pc_1$ ist möglich), und die vierte gibt den Zusammenhang zwischen den beiden Zuständen (*pc,st*) und (*pc',st'*) der beiden ASMs wieder. Ihre Invarianz folgt für ein folgendes k_1:k_2-Diagramm trivial aus der Annahme (7.1), da die Voraussetzungen der Implikation alle direkt in der Invariante stehen, mit Ausnahme von $peephole(st,pc_1,db_1,k_1,il_2)$, was aber direkt aus (7.4) folgt (die Linearität der Instruktionen impliziert, daß am Ende des Diagramms $pc = pc_1 + k_1$ und $pc' = pc_1 + k_2$ gilt, woraus sich unmittelbar $pc' = shift(pc,pc_1,k_2 - k_1)$ ergibt).

Für die 1:1-Diagramme ergibt sich die dritte Eigenschaft direkt aus der Forderung $notjumpedto(pc_1, k_1, db_1)$ (keine Sprünge in den optimierten Code), die vierte ergibt sich direkt aus der Annahme (7.3) für der *repl*-Funktion.

Um alle in Kapitel 6 definierten Beweisverpflichtungen für die Äquivalenz der ASMs zu erfüllen, ist schließlich noch zu zeigen, daß die Zusammenhangsinvariante in den Initialzuständen gilt. Daraus ergibt die Zusatzforderung, daß die nicht optimierte ASM die Ausführung nicht mitten im optimierten Code beginnt. Zu beachten ist noch, daß sich für $k_1 = 0$ oder $k_2 = 0$ die Besonderheiten von m:0 und 0:n-Diagrammen nicht auswirken, da nicht mehrere aufeinander folgen können, sowie daß die Zusammenhangsinvariante trivial die Nachbedingung $st = st'$ impliziert.

Zusammenfassend zeigt es sich also, daß die Korrektheit von Peephole Optimierung einen Spezialfall des Modularisierungstheorem für ASM-Verfeinerungen

darstellt, sofern der optimierte Code keine Sprunganweisungen enthält. Sprunganweisungen im optimierten Code werden im übernächsten Abschnitt behandelt.

7.2 Vergleich mit der PVS-Formalisierung

In diesem Abschnitt geben wir einen kurzen Vergleich unserer Formalisierung zu der in [DvHPR97] definierten.

Grundsätzlich unterscheiden sich die beiden Formalisierungen zunächst darin, daß in [DvHPR97] die Semantik eines Interpreters (durch die Funktion interprete) sowie deren Äquivalenz (durch das Prädikat $\equiv$) speziell für peephole Optimierung neu definiert werden, während wir bereits allgemein definierte Begriffe (ASMs, deren Semantik und Verfeinerung) instanzieren.

Eine starke Einschränkung der Formalisierung in [DvHPR97] ist, daß nur Programmcode ohne Sprunganweisungen betrachtet wird. Sie ermöglicht es, auf einen Programmzähler *pc* zu verzichten, den Programmcode als eine Liste von Anweisungen aufzufassen, und Beweise durch Induktion über die Länge der Codeliste zu führen. Eine derartige Induktion ist für Programmcode mit Sprunganweisungen nicht möglich.

Bezüglich der notwendigen Bedingungen ergibt unsere Formalisierung gegenüber rein linearem Code wie in [DvHPR97] genau die beiden offensichtlichen Zusatzforderungen

- Der Programmstart darf nicht im optimierten Code liegen.
- Es darf keine Sprunganweisungen geben, die in den optimierten Code springen.

In einigen technischen Punkten sind unsere Definitionen weniger restriktiv (aber auch weniger konkret) als die in [DvHPR97] gegebenen. So verzichten wir auf die Definition eines Regelschemas für jede Regel, sowie auf eine genauere Spezifikation des *peephole*-Prädikat. Wir geben deshalb an, wie sich die Definitionen aus [DvHPR97] durch Spezialisierung unserer Definitionen erhalten lassen:

Ein Regelschema aus [DvHPR97], S. 4, Abb. 1 entspricht in unserer Formalisierung der Definition einer ASM-Regel der Form

if $\mathrm{code(pc,db)} = \mathrm{i_k} \wedge \mathrm{admissible(i_k)(st)}$
then $\mathrm{pc,st} := \mathrm{effect(i_k)(pc+1,st)}$

für jede Instruktion i_k. Wie man sieht, dienen die global definierten Funktionen *admissible* und *effect* lediglich zur funktionalen Codierung der Semantik einer deterministischen Regelanwendung (die Restriktion auf deterministische Regeln ist in unserer Formalisierung ebenfalls nicht vorhanden). Die implizite Restriktion, daß *pc* um eins erhöht wird, ist hier explizit wiedergegeben. Unser Prädikat *ok*(*pc,st,db*) entspricht *admissible*(*code*(*pc,db*),*pc,st*).

Die Funktion *interprete* entspricht der Semantik der ASM. Die leere Menge als Ergebnis entspricht einem Endzustand *st*, in dem nicht *ok(pc,st,db)* gilt. Die Definition des Prädikats „==“ in Abb. 4, Seite 5 fällt mit unserer Definition von ASM-Äquivalenz für den Spezialfall, daß *IN* und *OUT* beide die Identität auf *pc* und *st* sind, zusammen.

Unsere Definition des Prädikats *peephole* ist sehr abstrakt. In [DvHPR97] wird eine konkretere gegeben: Danach wird für Peephole Optimierung eine Liste $[R_1, \ldots R_n]$ von Regeln der Form $R_i = (p_i, r_i, c_i)$ benötigt, die aus 3 Teilen bestehen:

- Einer ersten Liste p_i („Pattern“) von Instruktionen, die ersetzt werden soll.
- Einer zweiten Liste r_i („Replacement“) von Instruktionen, die als Ersetzung für p_i dient.
- Einem Prädikat c_i („Condition“), das die Zustände beschreibt, in denen die Regel anwendbar ist.

Dies entspricht einer Definition von n Prädikaten $peephole_1, \ldots, peephole_n$, die durch

$$\mathrm{peephole_i(st,pc,db,k_1,il_2)} : \leftrightarrow \quad \mathrm{instrs(pc,db,k_1)} = \mathrm{p_i} \\ \wedge\ \mathrm{il_2} = \mathrm{r_i} \wedge \mathrm{c_i(st)}$$

definiert werden. Die Regeln werden der Reihe nach auf das Ausgangsprogramm angewandt (die Gesamtkorrektheit aller Optimierungen ergibt sich aus der Transitivität der Programmäquivalenz). Die Definition in [DvHPR97] erschien uns an dieser Stelle zu speziell, da kein Pattern Matching zwischen dem Pattern und dem aktuellen Code stattfindet (es scheint, daß für jede Instanz eine neue Regel angegeben werden muß), und die Prädikate c_i keinen Bezug zu dem Code haben, der abläuft, bevor *pc* erreicht wird. In der in [DvHPR97] gegebenen Definition kann der Test, ob die Bedingung c_i zutrifft, und die Regel R_i angewendet werden kann, *nur* durch die Inspektion aller erreichbaren Zustände getestet werden, was praktisch gesehen nicht möglich ist. Unsere Definition des *peephole*-Prädikats macht es möglich, beliebige syntaktische Bedingungen für die Anwendbarkeit zu definieren. Ebenso ist eine beliebige Definition von Patterns und Pattern Matching noch möglich. Da die konkrete Definition sowohl von syntaktischen Kriterien für die Anwendbarkeit als auch von Pattern Matching von der Form des konkreten Programmcodes abhängen, haben wir die abstrakte Definition eines *peephole*-Prädikats gewählt.

7.3 Optimierungen von Sprunganweisungen

Dieser Abschnitt beschäftigt sich mit der Optimierung von Codefolgen, die Sprunganweisungen enthalten. Es wird zwar keine generische Methode zur Verifikation gegeben, die gegebenen Beispiele sollten aber zeigen, daß sich auch

Sprunganweisungen mit Hilfe des Modularisierungstheorems leicht behandeln lassen. Lediglich die Zahl der kommutierenden Diagramme nimmt mit der Zahl der betrachteten Sprunganweisungen zu.

Ein Spezialfall sind die in [TvS82] genannten konkreten Optimierungen einer Stackmaschine, die sich mit Sprunganweisungen beschäftigen und deshalb in [DvHPR97] nicht betrachtet wurden. Es handelt sich dabei immer um Codefolgen, bei denen nur die *letzte* Anweisungen der Codefolgen $instrs(pc_1,db_1,k_1)$ und il_2 Sprunganweisungen sind. In diesem Fall genügt es, die Bedingung (7.1) zu

$$
\begin{array}{ll}
 & \mathrm{I}(\mathrm{db}_1,\mathrm{st}_0,\mathrm{pc}_0) \\
 & \wedge\ \exists\ \mathrm{i}.\ \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc}_0,\mathrm{st}_0)\ \textbf{times}\ \mathrm{i}\rangle\ (\mathrm{pc}_0 = \mathrm{pc}_1 \wedge \mathrm{st}_0 = \mathrm{st}_1) \\
 & \wedge\ \mathrm{peephole}(\mathrm{st}_1,\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\mathrm{il}_2) \wedge \mathrm{db}_2 = \mathrm{repl}(\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1,\ \mathrm{il}_2) \\
 & \wedge\ \mathrm{pc}_1 = \mathrm{pc}_2 \wedge \mathrm{pc}_1 = \mathrm{pc} \wedge \mathrm{st}_1 = \mathrm{st}_2 \\
\rightarrow & \quad \mathrm{k}_1 \neq 0 \wedge \mathrm{linear}(\mathrm{instrs}(\mathrm{pc}_1,\mathrm{db}_1,\mathrm{k}_1 - 1)) \\
 & \wedge\ \mathrm{il}_2 \neq [] \wedge \mathrm{linear}(\mathrm{butlast}(\mathrm{il}_2)) \\
 & \wedge\ \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc}_1,\mathrm{st}_1)\ \textbf{times}\ \mathrm{k}_1\rangle \\
 & \qquad \langle \textbf{loop}\ \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}_2,\mathrm{st}_2)\ \textbf{times}\ \mathrm{k}_2\rangle \\
 & \qquad\quad (\mathrm{st}_1 = \mathrm{st}_2 \wedge \mathrm{pc}_2 = \mathrm{shift}(\mathrm{pc}_1,\mathrm{pc},\mathrm{k}_2 - \mathrm{k}_1))
\end{array}
\tag{7.6}
$$

zu verallgemeinern (*butlast* entfernt das letzte Element einer Liste). Die verallgemeinerte Bedingung reicht offenbar immer noch aus, um die Kommutativität des k_1:k_2-Diagramms bei unveränderter Zusammenhangsinvariante zu garantieren. Als zusätzliche Anforderung ist nur sicherzustellen, daß das Sprungziel der beiden letzten Anweisungen (modulo *shift*) dasselbe ist. Daß es außerhalb des optimierten Codes liegt, folgt bereits aus der *notjumpedto*-Forderung (7.5).

Abschließend betrachten wir noch ein einfaches Beispiel zur Optimierung von Sprunganweisungen, bei dem nicht nur die letzte Anweisung eine Sprunganweisung ist. Das Beispiel soll zeigen, daß dann im wesentlichen mehrere kommutierende Diagramme betrachtet werden müssen, die sich aus den Fallunterscheidungen der Sprunganweisungen ergeben.

Für das Beispiel nehmen wir einen Zustand *st* an, bei dem es möglich ist, mit *get*(*l*,*st*) einen Integer-Wert zu selektieren (l sei etwa eine Speicheradresse, *get* ein Speicherzugriff). Drei typische Sprunganweisungen wären dann etwa $BZE(l,n)$, $BNZ(l,n)$, und $BRA(n)$ (branch on zero, branch on not zero, branch unconditionally) mit den ASM-Regeln:

if code(pc,db) = BZE(l,n)
then if get(l,st) = 0
  then pc := pc + n
  else pc := pc + 1

if code(pc,db) = BNZ(l,n)
then if get(l,st) = 0
  then pc := pc + 1

else pc := pc + n

if code(pc,db) = BRA(n)
then pc := pc + n

Eine offensichtliche Optimierung wäre dann für $n > 0$ etwa die von $il_1 = [BZE(l,2)\ BRA(n)]$ zu $il_2 = [BNZ(l,n-1)]$. Falls $instr(pc_1,db_1,2) = il_1$ ist, und weder der Programmstart bei pc_1+1 liegt, noch Sprünge auf diese Adresse existieren, ist diese Optimierung offenbar auch zulässig. Für die Verifikation benötigen wir dieselbe Zusammenhangsinvariante wie im vorigen Abschnitt, und auch die Verifikation des Falls $pc \neq pc_1$ bleibt unverändert. Für die Verifikation des optimierten Codes ($pc = pc_1$) werden nun zwei Diagramme benötigt. Ein 1:1 Diagramm für den Fall, daß $get(l,st) = 0$, und ein 2:1-Diagramm für den Fall $get(l,st) \neq 0$. Daß die beiden Diagramme kommutieren, daß also

$$\begin{aligned} &\mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}') \wedge \mathrm{pc} = \mathrm{pc}_1 \wedge \mathrm{get}(\mathrm{l},\mathrm{st}) = 0 \\ \rightarrow\ &\langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st})\rangle\ \langle \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}',\mathrm{st}')\rangle \\ &\quad \mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}') \end{aligned}$$

und

$$\begin{aligned} &\mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}') \wedge \mathrm{pc} = \mathrm{pc}_1 \wedge \mathrm{get}(\mathrm{l},\mathrm{st}) \neq 0 \\ \rightarrow\ &\langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st})\rangle\ \langle \mathrm{RULE}(\mathrm{db}_1;\mathrm{pc},\mathrm{st})\rangle\ \langle \mathrm{RULE}(\mathrm{db}_2;\mathrm{pc}',\mathrm{st}')\rangle \\ &\quad \mathrm{INV}(\mathrm{db}_1,\mathrm{pc},\mathrm{st},\mathrm{db}_2,\mathrm{pc}',\mathrm{st}') \end{aligned}$$

gilt, läßt sich nun leicht nachrechnen.

Kapitel 8

Zusammenfassung Teil I

Der erste Teil dieser Arbeit befaßte sich mit der Entwicklung von Werkzeugunterstützung für die Spezifikationssprache der ASMs und der Definition von generischen Beweisverpflichtungen für die Verfeinerung von ASMs. Die Arbeit ergab drei wesentliche Resultate:

Erstens die Entwicklung einer natürliche Einbettung der Spezifikationssprache der ASMs in die Dynamische Logik, die es erlaubt, Aussagen über ASMs, insbesondere die Korrektheit von ASM-Verfeinerungen zu formalisieren. Dieses Resultat macht die werkzeugunterstützte Deduktion für ASMs möglich.

Zweitens die Entwicklung einer Theorie zur Modularisierung von Korrektheitsbeweisen für ASM-Verfeinerungen. Die bewiesenen Modularisierungstheoreme verallgemeinern die bisher bekannten Resultate. So sind u.a. Data Refinement und Peephole Optimierung aus der Compilerverifikation ein Spezialfall des Theorems.

Drittens die praktische Umsetzung der ersten beiden Resultate im KIV-System. Zum einen wurden die Modularisierungstheoreme für ASM-Verfeinerungen in KIV selbst formal verifiziert, zum anderen wurden verschiedene Erweiterungen an der Spezifikationssprache und den Deduktionskomponenten vorgenommen, um eine leistungsfähiges Werkzeug für ASMs zu erhalten.

Teil II

Die Prolog-WAM-Fallstudie

Kapitel 9

Einführung und Überblick

Gegenstand der Prolog-WAM-Fallstudie ist der Korrektheitsnachweis für die Übersetzung von Prolog-Programmen in Byte-Code der Warren Abstract Machine (WAM). Die WAM (und Varianten) bildet die Grundlage praktisch aller gängigen Prolog-Implementierungen.

Die Arbeit basiert auf einer systematische Darstellung dieser Übersetzung als 12 ASM-Verfeinerungen in [BR95]. Den Startpunkt bildet ein als ASM spezifizierter Prolog-Interpreter, der die operationale Semantik des Kerns von Prolog (Klauseln mit *!, true* und *fail*) durch den Aufbau eines Suchbaums wiedergibt. Für pures Prolog ist dieser identisch zu dem durch SLD-Resolution aufgebauten Baum. Die Erweiterung der ASM auf volles Prolog ([BR94]) ist inzwischen ein ISO-Standard für die Semantikdefinition von Prolog.

Der erste Prolog-Interpreter, den wir im folgenden als ASM1 bezeichnen, wird anschließend durch ASM-Verfeinerungen schrittweise in einen Interpreter ASM13 von Byte-Code, der WAM transformiert. Parallel zu dieser Transformation wird das Prolog Programm übersetzt. Auf Zwischenebenen besteht der Code teilweise aus noch nicht übersetzten Prolog-Klauseln, teilweise aus WAM-Instruktionen. Die einzelnen Verfeinerungen führen dabei schrittweise maschinennahe Konzepte wie Stacks, Register, Zeigerstrukturen etc. ein. Die Verfeinerungen sind dabei orthogonal: Die Übersetzung der Klauselselektion, die Übersetzung von Einzelklauseln und die Übersetzung von Literalen werden getrennt in unabhängigen Verfeinerungsschritten behandelt. Neben den reinen Übersetzungsschritten sind auch Verfeinerungsschritte vorhanden, die der Optimierung der Datenrepräsentation dienen. Die in der letzten ASM13 verwendeten WAM-Byte-Code Instruktionen sind sehr einfach. Sie bestehen jeweils nur noch aus einer Anzahl von Registertransfers, lassen sich also leicht in den Assemblercode jedes beliebigen Prozessors übersetzen.

Die wesentlichen Ziele bei der Prolog-WAM-Fallstudie waren:

- die formale Spezifikation der in [BR95] gegebenen Übersetzungsschritte und Compilerannahmen
- die Formalisierung der Korrektheit von Verfeinerungen

- die Erarbeitung einer geeigneten Beweismethodik zur Verifikation der Verfeinerungen
- die Entwicklung einer geeigneten Unterstützung in KIV, die einen effizienten Nachweis der Korrektheit der Verfeinerungsschritte ermöglicht
- die informell gegebenen Korrektheitsargumente auch formal nachzuvollziehen, bzw. Fehler zu finden und zu beheben.

Wesentliche Teile der Theorie aus den Kapiteln 4 und 6 sind zur Lösung der ersten beiden Ziele entstanden. Die Entwicklung einer geeigneten Beweisunterstützung erforderte zahlreiche Verbesserungen an KIV, die kurz in 3.3 erwähnt wurden. Wie der Vergleich zur Isabelle-Fallstudie in Abschnitt 20 zeigt, kann die Beweisunterstützung für den Korrektheitsnachweis mit anderen Systemen durchaus konkurrieren. Dennoch erfordert die formale Verifikation einer Verfeinerung im Mittel immer noch einen Personenmonat an Bearbeitungszeit. Im Rahmen dieser Arbeit konnten 8 der 12 Verfeinerungen verifiziert werden.

Die Verifikation ergab im wesentlichen eine Bestätigung der konzeptionellen Arbeit aus [BR95]. Es war bisher nicht notwendig, wesentliche Änderungen an den ASMs vorzunehmen. Auch die angegebenen Ideen für den Korrektheitsnachweis waren für alle Verfeinerungen richtig.

Allerdings zeigte sich schon beim Nachweis der Korrektheit des ersten Verfeinerungsschritts (man vergleiche z.B. den ersten Ansatz am Anfang von Abschnitt 11.2 mit der endgültigen Zusammenhangsinvariante am Ende), daß für eine formale Verifikation eine große Zahl implizit angenommener Eigenschaften explizit gemacht werden müssen. Obwohl viele dieser impliziten Eigenschaften sehr einfach zu finden sind, zeigt die große Zahl, daß noch eine große Lücke zwischen einem mathematischen Nachweis der Korrektheit von Verfeinerung und einem voll ausformalisierten Argument besteht.

Somit ist es auch nicht sehr überraschend, daß eine erhebliche Zahl an kleineren Problemen in den ASMs und in den Compilerannahmen gefunden werden konnten, die bei der informellen Analyse in [BR95] nicht gefunden wurden. Die wichtigsten Probleme waren:

- ASM3 und ASM4 enthalten einen nicht beabsichtigten Indeterminismus, der durch einen verstärkten Regeltest zu beheben ist (siehe 14.2)
- In den Switching-Anweisungen fehlte Backtracking (siehe 15.2)
- Die unify-Instruktion der ASM9 greift auf den Renaming-Index des ersten statt der zweiten Umgebung zu (siehe 17.1)
- Die Compilerannahmen für etliche Refinements waren zwar im Text informell korrekt beschrieben, die Formalisierung mußte aber präzisiert werden (siehe 14.2,15.2)
- ASM1–ASM8 können die Anfrage `?- p(q)` positiv lösen, wenn die beiden Klauseln `p(X) :- X.` und `q.` gegeben sind. Bei der Übersetzung von

Klauseln in Code (beim Übergang zu ASM9) dürfen Klauselrümpfe aber keine Variablen und keine Listen mehr enthalten (siehe 17.2).

Die Probleme sind alle relativ leicht zu beheben. Dennoch zeigt das Ergebnis, daß auch eine sehr sorgfältige informelle Analyse um einen formalen Korrektheitsbeweis ergänzt werden sollte, wenn ein korrekter Compiler das Ziel ist.

Die folgenden Kapitel beschreiben die Probleme im Rahmen der Korrektheitsbeweise detailliert. Sie sind wie folgt organisiert:

Das nächste Kapitel beschreibt den Prolog-Interpreter aus [BR95]. Die dann folgenden Kapitel bestehen dann jeweils aus 2 Abschnitten: Der erste beschreibt die Verfeinerung der ASM aus dem vorherigen Kapitel zu einer neuen ASM. Dieser Abschnitt folgt weitgehend [BR95]. Sofern sich aus der Formalisierung Abweichungen oder Präzisierungen von Annahmen ergaben, werden sie dort erklärt. Der zweite Unterabschnitt beschreibt die formale Verifikation der betrachteten Verfeinerung mit KIV, die Erfahrungen, die dabei gemacht wurden, sowie Korrekturen an den ASMs und den Compilerannahmen, die sich aus der Verifikation ergaben.

Wir haben versucht, die in den jeweiligen Abschnitten für die ASMs und die Zusammenhangsinvarianten notwendigen neuen Funktionen immer sofort zu erklären. Sollten dennoch Unklarheiten bleiben, so kann zusätzlich die volle algebraische KIV-Spezifikation aller verwendeten Funktionen im Anhang E eingesehen werden.

Im folgenden bezeichnet i/j die Verfeinerung von ASMi zu ASMj. Wir verwenden außerdem in jedem Abschnitt zur Verifikation von i/j die Konvention, die Zustandsvariablen von ASMi (genauer: der DL-Übersetzung von ASMi) mit $\underline{x}$ und die Zustandsvariablen von ASMj mit $\underline{x}'$ zu bezeichnen. Wir nehmen jeweils an, daß die beiden Vektoren immer disjunkt sind. Die Regel (im Sinne von Abschnitt 2.2) von ASMi und ASMj bezeichnen wir mit $RULE$ und $RULE'$. $RULE$ (und analog $RULE'$) hat immer die Form

RULE(**var** $\underline{\mathrm{x}}$) **begin**
if ε_1 **then** RULE$_1$($\underline{\mathrm{x}}$) **else**
if ε_2 **then** RULE$_2$($\underline{\mathrm{x}}$) **else**
$\vdots$
if ε_n **then** RULE$_\mathrm{n}$($\underline{\mathrm{x}}$) **end**

wobei $RULE_1$, $RULE_2$, ..., $RULE_n$ Regeln im Sinne von Abschnitt 2.3 sind. Wir werden den Begriff „Regel“ in Zukunft auch ausschließlich im letzteren Sinn verwenden.

Kapitel 10

ASM1 : Ein Prolog-Interpreter

Die beiden wichtigsten Datenstrukturen, die ein Prolog-Interpreter benötigt, um einen Berechnungszustand von Prolog zu beschreiben, sind die *Liste der Prolog Literale*, die noch auszuführen sind, sowie die aktuell berechnete *Substitution*. Ein derartiger Zustand wird modifiziert durch:

1. Unifikation des ersten Literals der Liste, des sogenannten Aktivators *act* (engl. activator) mit dem Kopf einer Klausel
2. Ersetzung von *act* durch den Rumpf dieser Klausel in der Liste
3. Anwendung der unifizierenden Substitution auf die resultierende Liste
4. Komposition der unifizierenden Substitution mit der bisher berechneten Substitution

Wenn die Unifikation während dieses Verfahrens fehlschlägt (zu einem „failure“ führt), müssen mit Hilfe von Backtracking alternative Klauseln versucht werden. Damit dies möglich ist, muß ein Prolog-Interpreter sich die Historie merken, die zum gegenwärtigen Zustand geführt hat. Diese Historie wird üblicherweise durch einen Prolog-Suchbaum repräsentiert, der durch die obigen Operationen aufgebaut wird. Jeder Knoten entspricht einem Berechnungszustand, die Nachfolgerknoten eines Zustands sind die möglichen Nachfolgezustände, die sich durch die Unifikation mit verschiedenen Klauselköpfen ergeben.

In einer ASM wird ein Suchbaum durch eine dynamische Sorte *node* von Knoten modelliert, die durch eine ebenfalls dynamische Funktion *father* verknüpft sind, die zu jedem Knoten seinen Vaterknoten liefert. Der Wurzelknoten wird durch $\perp$ bezeichnet, auf ihm ist *father* undefiniert. Die Information über alternative Klauseln, die an einem Knoten n versucht werden können, wird in einer Liste $cands[n]$ von Kandidatenknoten gespeichert. Jeder Knoten der Kandidatenliste verweist mit Hilfe einer Funktion *cll* (clause line) auf eine Zeile des

Prolog Programms, das in einer Konstanten *db* gespeichert ist. Eine geeignete initiale Liste von Klauselzeilen kann Hilfe der *procdef*-Funktion berechnet werden (zur Spezifikation von *procdef* siehe weiter unten).

Der aktuelle Berechnungszustand, in dem sich der Interpreter befindet, wird durch eine Programmvariable (i.e. eine 0-stellige dynamische Funktion) *currnode* (engl. current node) markiert. Der zu einem Knoten n gehörende Berechnungszustand könnte nun durch 2 Funktionen $glseq[n]$ (engl. goal sequent) und $sub[n]$ gegeben werden.

Um die *cut*-Anweisung behandeln zu können, ist jedoch eine Erweiterung der Repräsentation des Berechnungszustands notwendig. Ein *cut* sollte den *father* des gegenwärtigen Knotens auf denjenigen Knoten abändern, dessen Aktivator *act* die *cut*-Anweisung in die Liste der zu bearbeitenden Literale einführte. Um diese Abänderung durchführen zu können, muß dieser Knoten gemerkt werden. Eine uniforme Lösung für dieses Problem ist, den *father* des früheren *currnode* an jeden Klauselrumpf anzuhängen, der bei der Unifikation an die Liste der zu bearbeitenden Literale angehängt wird. Die resultierende Datenstruktur *decglseq* (engl. decorated goal sequence) ist eine Liste, die aus Paaren besteht, deren erste Komponente ein Teil eines Klauselrumpfes (ein *goal*), und deren zweite Komponente ein Knoten (der sog. Cutpoint *ctpt*) ist. Sie hat also folgende Gestalt:

$$\text{decglseq} = [\langle \underbrace{[\overbrace{g_{1,1}}^{\text{act}}, g_{1,2}, \ldots, g_{1,k_1}]}_{\text{goal}}, \overbrace{n_1}^{\text{ctpt}} \rangle, \ldots, \langle [g_{m,1}, \ldots, g_{m,k_m}], n_m \rangle]$$

$$\text{cont} = [\langle \quad [g_{1,2}, \ldots, g_{1,k_1}], n_1 \rangle, \ldots, \langle [g_{m,1}, \ldots, g_{m,k_m}], n_m \rangle]$$

Die Abkürzung *cont* (engl. continuation), die die *decglseq* ohne *act* bezeichnet, erleichtert im folgendenden die Konstruktion von modifizierten *decglseq*.

Um die Regeln von ASM1 einzuführen, betrachten wir nun das folgende Prolog-Programm

```
1 p :- fail.          3 q.
2 p :- q,!,true.      4 p.
```

das in einer Konstante *db* (database) in der initialen ASM gespeichert ist. Die angegebenen (normalerweise nicht explizit vorhandenen) Zeilennummern erleichtern die folgenden Erklärungen.

Wir nehmen des weiteren eine Anfrage `?- p.` an. Diese wird als *decglseq* des Knotens A im initialen Suchbaum gespeichert. Dieser ist in Abb. 10.1 dargestellt.

Die beiden mit $\perp$ und A bezeichneten Knoten bilden die initiale Trägermenge der dynamischen Sorte *node*. Die Trägermenge wird durch ASM-Regeln erweitert. Die in der Funktion *father* gespeicherte Baumstruktur ist durch den Pfeil wiedergegeben, d.h. es gilt $father[A] = \perp$. Der Wurzelknoten $\perp$ dient lediglich als Merker, wann die Suche beendet ist. Er speichert (in ASM1) keinerlei Information. Der initiale Startknoten *currnode* ist A, was durch den doppelten

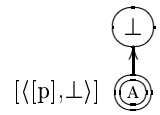


Abbildung 10.1

Kreis um *A* angezeigt wird. Die berechneten Substitutionen für jeden Berechnungszustand, die durch die Funktion *sub* den Knoten zugeordnet werden, sind in der Abbildung nicht gezeigt, da sie in unserem Beispiel stets leer sind.

Der Ablauf des Interpreters wird durch 2 Programmvariablen (i.e. 0-stellige dynamische Funktionen) *mode* und *stop* gesteuert. Der Wert von *mode* wechselt zwischen *call* (Call-Modus) und *select* (Select-Modus), während der Wert von *stop* stets *run* ist. Die Suche wird durch das Setzen von *run* auf *halt* beendet, d.h. alle ASM-Regeln haben in ihrer Anwendbarkeitsbedingung den Test *stop* = *run*.

Im *call*-Modus, der der initiale Modus ist, werden Kandidatenknoten berechnet (wir nehmen an, daß alle Guards, die auf *act* Bezug nehmen, implizit die Tests *decglseq* $\neq$ [], *goal* $\neq$ [] enthalten, und lassen den Test auf *stop* = *run* weg):

call rule
  if is_user_defined(act) $\wedge$ mode = call
  then let[cll_1,...,cll_n] = procdef(act,db)
    extend node
    by tmp_1,...,tmp_n
    with father[tmp_i] := currnode
      cll[tmp_i] := cll_i
      cands := [tmp_1,...,tmp_n]
    endextend
    mode := select

Die für einen *currnode* berechneten Kandidatenknoten, i.e. *cands*[*currnode*], werden in der Regel mit *cands* abgekürzt, und wir werden im folgenden auch die analogen Abkürzungen *father*, *decglseq* und *sub* verwenden.

Das **extend**-Konstrukt stammt aus [BR95]. Es allokiert einen Knoten für jede Klausel, deren Kopf mit dem Aktivator „unifizieren könnte“. Die entsprechende Liste von Klauselzeilen wird von *procdef(act,db)* berechnet. Die Funktion *procdef* ist zunächst unterspezifiziert: Es wird lediglich angenommen, daß sie mindestens die Klauseln liefert, deren Kopf mit *act* unifizieren, und höchstens all diejenigen, deren Kopf mit demselben Prädikatsymbol wie *act* beginnen. Strenggenommen stellt die Verwendung des **extend**-Konstrukts mit einer unbeschränkten Zahl zu allokierender Knoten eine Erweiterung zu [Gur95] dar. In DL kann die Erweiterung mit Hilfe einer Prozedur, die die Liste *procdef(act,db)* abarbeitet, realisiert werden. Das Ergebnis der Anwendung der *call*-Regel ist in Abb. 10.2 dargestellt:

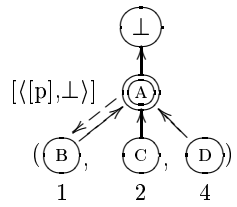


Abbildung 10.2

Die *cands*-Liste des Knotens A wird durch einen gestrichelten Pfeil zum ersten Knoten der Liste und Klammern um die Knoten wiedergegeben. Die Klauselzeilen, die mit Hilfe der Funktion *cll* an den Kandidatenknoten hängen, sind durch die entsprechenden Zahlen unterhalb der Knoten aufgeführt. Die *call*-Regel ändert den Wert der *mode*-Variable zu *select*, wodurch die *select*-Regel aktiviert wird:

select rule
  if is_user_defined(act) ∧ mode = select
  then if cands = []
    then backtrack
    else let clau = rename(clause(cll[car(cands)],db),vireg)
      let mgu = unify(act,head(clau))
      if mgu = failure
      then cands := cdr(cands)
      else currnode := car(cands)
        decglseq(car(cands)) := mgu $\hat{}_d$ [⟨body(clau), father⟩ | cont]
        sub[car(cands)] := sub ∘ mgu
        cands := cdr(cands)
        vireg := vireg +1
        mode := call

wobei

backtrack ≡
  if father = ⊥
  then stop := halt
    subst := failure
  else currnode := father
    mode := select

Anwendung der Regel verursacht Backtracking, falls keine Kandidatenknoten mehr vorhanden sind. Andernfalls entfernt die Regel iterativ all die Kandidatenknoten, deren Klauselkopf nicht mit *act* unifiziert. Wenn der erste Kandidat erreicht wird, für den ein allgemeinster Unifikator *mgu* (engl. most general uni-

fier) existiert (die Funktion *clause* selektiert die Klausel an einer Klauselzeile[1], der Variablenindex *vireg* wird zur Erzeugung neuer Instanzen der Klauselvariablen verwendet), wird dieser Knoten zum *currnode*. Die infix geschriebene Operation $\hat{}_d$ wendet den *mgu* auf die neue *decglseq* an.

Das Ergebnis der Anwendung der *select*-Regel ist in Abb. 10.3 zu sehen. Der

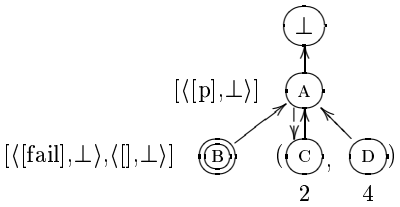


Abbildung 10.3

Wert der *mode*-Variable ist nun wieder *call*. Da der Aktivator *fail* nun nicht benutzerdefiniert ist, wird die *fail*-Regel angewandt.

fail rule
if act = fail **then backtrack**

Diese setzt den *currnode* wieder auf A. Zu beachten ist, daß der Knoten B nicht deallokiert wird (d.h. er bleibt in der Trägermenge der *node*-Sorte). Im *select*-Modus wird nun C, der nächste Kandidatenknoten von A selektiert. Dessen *decglseq* ergibt sich zu $[\langle [q, !, true], \bot\rangle\ , \langle [], \bot\rangle\ ]$. Anschließend allokiert die *call*-Regel einen neuen Knoten E für die einzige anwendbare Klausel `q`. Nach der Auswahl von E erreicht die ASM den in Abb. 10.4 gezeigten Zustand.

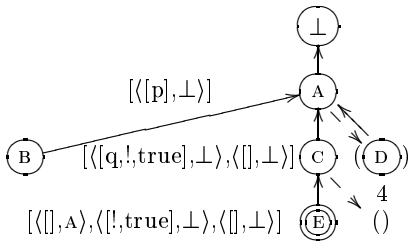


Abbildung 10.4

Das nun leere *goal* wird mit der *goal success*-Regel gelöscht.

goal success rule
if decglseq $\neq$ [] $\wedge$ goal = []
then decglseq := cdr(decglseq)

[1] Da *clause* offenbar vom aktuellen Prologprogramm abhängt, haben wir gegenüber [BR95] ein Argument *db* addiert

Anschließend ist der Aktivator ein Cut. Der Cut wird durch die *cut*-Regel aus der *decglseq* entfernt.

cut rule
if act = !
then father := ctpt
decglseq := cont

Die Regel setzt dabei den *father* des aktuellen Knotens E auf den Knoten *ctpt* aus der *decgoalseq*, der hier der $\perp$-Knoten ist (Abb. 10.5). Dadurch wird die bei A bestehende Alternative D „abgeschnitten“. Die *cut*-Regel ist die einzige Stelle, an der *ctpt* verwendet wird.

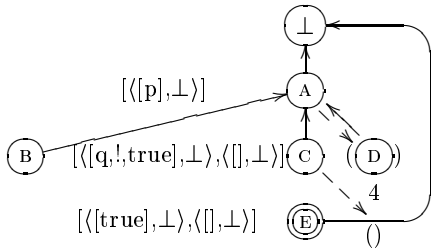


Abbildung 10.5

Für den Aktivator *true* führt die ASM dann die folgende Regel aus:

true rule
if act = true **then** decglseq := cont

Das Ende der Interpretation des Prolog-Programms wird dann nach zweimaliger Anwendung der *goal sucess*-Regel erreicht. Da dadurch *decglseq*[E] leer wird, ist die ursprüngliche Anfrage komplett gelöst. Deshalb setzt die *query success*-Regel nun die Antwortsubstitution (die hier leer ist), und beendet den Interpreter durch das Setzen von *stop* auf *halt*:

query success rule
if decglseq(currnode) = []
then stop := halt
subst := sub

Betrachtet man eine Variante des Beispielprogramms, in der die Klausel `p :- q,!,true` durch `p :- q,!,r` ersetzt wurde, so erreicht ASM1 ebenfalls den in Abb. 10.5 gezeigten Zustand. Doch nun wird durch die *call*-Regel ein Knoten F mit einer leeren Kandidatenliste allokiert (da keine Klauseln vorhanden sind, deren Kopf mit `r` unifiziert). Anschließend wird durch die *select*-Regel Backtracking sowohl für F als auch für E ausgelöst. Da der Vaterknoten von E jetzt $\perp$ ist, beendet sich der Interpreter in der *backtrack*-Routine schließlich durch das Setzen von *stop* := *halt* und *subst* := *failure*.

Kapitel 11

1/2: Von Suchbäumen zu Stacks

11.1 Definition von ASM2

In diesem Abschnitt beschreiben wir die erste Verfeinerung von ASM1 in Richtung der Warren Abstract Machine (WAM). Wir folgen dabei [BR95], [Section 1.2]. Die neue ASM2 unterscheidet sich von der vorherigen im wesentlichen in drei Punkten:

Zum ersten wird die Funktion *father* zu b umbenannt. Dies zeigt an, daß b nun abwärts ($b$ackwards) die Knoten eines Stacks verzeigert.

Zum zweiten stellt ASM2 Register *cllreg, decglseqreg, breg* und *subreg* zur Verfügung, die zu den Inhalten von *cll, decglseq, father* und *sub* für den *currnode* korrespondieren. Dadurch wird die Allokation des *currnode* komplett vermieden.

Zum dritten berechnet ASM2 keine Liste von Kandidatenknoten mehr, sondern nur noch den *ersten* Kandidatenknoten. Die entsprechende Klauselzeile wird direkt mit *cllreg* angesprochen. Damit dies möglich ist, wird in ASM2 vorausgesetzt, daß Klauseln entsprechend des führenden Prädikatsymbols ihrer Köpfe geblockt werden. Ein spezielle Markierung *null* dient dazu, das Ende eines solchen Blocks anzuzeigen. Die „compilierte“ Repräsentation des Beispielprogramms aus dem vorigen Abschnitt hat dann die folgende Form:

```
1 p :- fail.       3 p.      5 q.
2 p :- q,!,true.   4 null    6 null
```

Eine neue $procdef_2$-Funktion wird nun benötigt. $procdef_2(act,db_2)$ ergibt nun die erste Klauselzeile, deren Kopf möglicherweise mit act unifiziert. Für $act = p$ erhält man $procdef_2(p,db_2) = 1$, die erste Klauselzeile mit einer Klausel mit Kopf p. Der Zusammenhang zu der *procdef* Funktion von ASM1 wird durch eine *Compilerannahme* über die $compile_{12}$ Funktion hergestellt, die als Axiom im Äquivalenzbeweis für 1/2 verwendet wird. Sie lautet:

$$\begin{array}{l} \quad \mathrm{db}_2 = \mathrm{compile}_{12}(\mathrm{db}) \\ \rightarrow \langle \mathrm{CLLS\#}(\mathrm{procdef}_2(\mathrm{act},\mathrm{db}_2),\mathrm{db}_2),\mathrm{db}_2;\mathrm{col})\rangle \\ \qquad \mathrm{mapclause}(\mathrm{procdef}(\mathrm{act},\mathrm{db}),\mathrm{db}) = \mathrm{mapclause}'(\mathrm{col},\mathrm{db}_2) \end{array}$$

Die Prozedur *CLLS#*[1] sammelt dabei aufeinanderfolgende Zeilennummern auf, bis ein *null* erreicht wird und die Funktionen *mapclause* und *mapclause'* selektieren jeweils die Klauseln an jeder Zeilennummer. Man beachte, daß wir *nicht* wie [BR95] (S. 17) angenommen haben, daß die Literale in der ersten Datenbank nach führendem Prädikatsymbol sortiert sind, und daß die Gleichheit *procdef(act,db) = col* der Adressen gilt. Stattdessen fordern wir nur die Gleichheit der an den Adressen gespeicherten Klauseln. Diese Abschwächung ist notwendig, andernfalls kann die Compilerannahme für keine Implementierung der *procdef*-Funktion erfüllt werden, die das Literal genauer als nur das führende Prädikatsymbol betrachtet: man beachte dazu, daß unter der stärkeren Annahme $procdef_2(p(f(X)),db_2)$, $procdef_2(p(g(X)),db_2)$ und $procdef_2(p(X),db_2)$ nicht drei verschiedene Ergebnisse liefern kann, da die aufgesammelten Listen immer mit dem einen für die p-Klauseln vorhandenen *null* enden müssen . Unsere schwärchere Annahme dagegen läßt sich erfüllen, sie erfordert im allgemeinen Codeverdopplung, die später durch die Einführung von Switching-Instruktionen (siehe Abschnitt 15.1) wieder aufgehoben wird.

Statt eine Kandidatenliste zu allokieren, schreibt ASM2 einfach den Wert von $procdef_2(act,db_2)$ ins *cllreg*. Dem Entfernen des ersten Elements aus der Liste *cands*[*currnode*] der Kandidaten in ASM1 entspricht in ASM2 nun das Inkrementieren von *cllreg*. Sobald die in *cllreg* gespeicherte Adresse *null* wird, sind keine Kandidaten mehr verfügbar.

Da ASM2 keinen *currnode* mehr zu allokieren braucht, muß ein neuer Knoten nur noch im *select*-Modus allokiert werden, um die gegenwärtigen Register-Inhalte zu retten. Die neue *call*- und *select*-Regel ergeben sich so zu

call rule
if is_user_defined(act) ∧ mode = call
then cllreg := $\mathrm{procdef}_2$(act,db_2)
mode := select

select rule
if is_user_defined(act) ∧ mode = select
then if clause(cllreg,db_2) = null
then backtrack
else let cla = rename(clause(cllreg,db_2),vireg)
let mgu = unify(act, head(cla))
if mgu = failure

[1] Eine Prozedur und keine Funktion wird hier verwendet, um auszuschließen, daß die Spezifikation durch ein db_2, das kein *null* enthält, inkonsistent wird. Siehe dasselbe Argument für *STACK#* im folgenden Abschnitt, S. 95

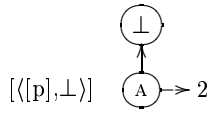
```
        then cllreg := cllreg +1
        else let tmp = new(s)
             s := s ∪ {tmp}
             breg := tmp
             b[tmp] := breg
             decglseq[tmp] := decglseqreg
             sub[tmp] := subreg
             cll[tmp] := cllreg +1
             decglseqreg := mgu ^_d [⟨body(cla),breg⟩ | cont]
             subreg := subreg ∘ mgu
             vireg := vireg +1
             mode := call
```

wobei

```
backtrack ≡
    if breg = ⊥
    then stop := halt
         subst := failure
    else decglseqreg := decglseq[breg]
         subreg := sub[breg]
         breg := b[breg]
         cllreg := cll[breg]
         mode := select
```

Die anderen Regeln von ASM1 bleiben im wesentlichen unverändert, lediglich die Umbenennung der Funktion *father* zu *b* ist zu beachten. Außerdem werden die Abkürzungen *decglseq*, *father* und *sub* (für *decglseq*[*currnode*] etc.) nun durch die Register *decglseqreg*, *breg* und *subreg* ersetzt.

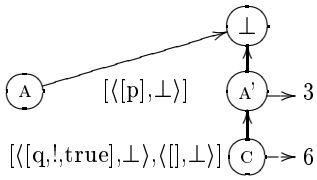
In unserem Beispiel durchläuft ASM2 nun die in Abb.11.1 und 11.2 gezeigten Zustände. Diese entsprechen den Zuständen von ASM1 aus den Abb. 10.3 und 10.4.



decglseqreg = [⟨[fail],⊥⟩,⟨[],⊥⟩]
breg = A

Abbildung 11.1

Gestrichelte Pfeile zeigen nun direkt auf den Wert von *cll* für einen Knoten. Da der früher in *currnode* gespeicherete Berechnungszustand nun in Registern gespeichert wird, entfällt die Allokation der Knoten B und D aus ASM1.



decglseqreg = [⟨[],A′⟩,⟨[!,true],⊥⟩,⟨[],⊥⟩]
breg = C

Abbildung 11.2

Auf der anderen Seite wird nun, wenn Knoten A durch Backtracking besucht wird (durch das Ausführen der *fail*-Anweisung im in Abb. 10.1 gezeigten Zustand), dessen Berechnungszustand in Register geschoben. Die anschließende *select*-Regel allokiert nun einen neuen, Knoten A' mit ganz ähnlichen Daten wie A. Die Beseitigung dieser Redundanz ist Gegenstand der nächsten Verfeinerung.

In ASM2 sind die allokierten Knoten, die noch in Zukunft besucht werden können, immer genau diejenigen, die von *breg* über die b Funktion erreicht werden können. Sie bilden einen Stack. Es ist aber zu beachten, daß neben den Stack-Knoten noch weitere, unerreichbare Knoten in der Trägermenge der Sorte *node* existieren können. Ein Beispiel ist hier der Knoten A. Dies ist eine der Problematiken, die sich bei der Verifikation von 1/2 auswirken wird. Das Tupel der Werte *decglseq*[n], *sub*[n], *cll*[n] und b[n], die an einem Stackknoten n hängen, wird üblicherweise als *Choicepoint* bezeichnet.

11.2 Äquivalenzbeweis 1/2

Dieser Abschnitt beschreibt die formale Verifikation der ersten Verfeinerung mit KIV. Das Hauptaugenmerk liegt in diesem Abschnitt nicht in der Anwendung der Theorie zur Verifikation von ASM-Refinements, sondern auf den praktischen Problemen, die sich bei der Durchführung der Verifikation ergeben. Diese bestehen hauptsächlich in der inkrementellen Entwicklung einer geeigneten Zusammenhangsinvariante. Wir zeigen an der Verfeinerung exemplarisch, daß

- der in [BR95] angegebene Zusammenhang zwischen den Zuständen für einen streng formal geführten Beweis bei weitem nicht ausreichend ist.
- eine große Zahl von zusätzlichen Eigenschaften formuliert werden muß, die zu Beginn der Verifikation nicht vorhersehbar sind, die aber notwendig sind, um die Korrektheit zu garantieren.
- für die effiziente Verifikation von ASM-Verfeinerungen ein System notwendig ist, das eine sehr gute Unterstützung für die inkrementelle Verifikation

von Behauptungen bietet.

Um den letzten Punkt sicherzustellen, waren viele Details am KIV-System weiterzuentwickeln. Einige wurden in Abschnitt 3.4 beschrieben.

Die folgende Darstellung der Entwicklung der Zusammenhangsinvariante macht es leider notwendig, den Leser mit einer großen Zahl von Details zu konfrontieren. Der Leser, der nicht an den Details der Verifikation sondern nur an ihren Ergebnissen interessiert ist, möge die 9 Eigenschaften aus [BR95], die am Anfang des folgenden Abschnitts gegeben werden, mit der letztendlichen Zusammenhangsinvariante am Ende des Abschnitts vergleichen. Ansonsten kann er die folgenden Ausführungen überspringen, und sich auf die Zusammenfassung am Ende des Abschnitts beschränken.

Die initiale Zusammenhangsinvariante Die Verfeinerung von ASM1 zu ASM2 ändert die Ablaufstruktur des Interpreters nicht. Eine Regelanwendung von ASM1 entspricht einer Anwendung der Regel gleichen Namens in ASM2. Wir haben also den Fall des *data refinement* vorliegen. Für die Beweisverpflichtungen aus Kapitel 6 bedeutet dies, daß $ndtype(\underline{x}, \underline{x}')$ konstant mn ist, und in der Beweisverpflichtung (6.5) $i = j = 1$ gewählt werden kann. Diese Beweisverpflichtung vereinfacht sich demnach zu

$$\begin{array}{l} \quad \mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}'),\ \mathrm{stop} = \mathrm{run},\ \mathrm{stop}' = \mathrm{run} \\ \vdash\ \langle \textbf{if}\ \mathrm{stop} = \mathrm{run}\ \textbf{then}\ \mathrm{RULE}\rangle \\ \quad \langle \textbf{if}\ \mathrm{stop}' = \mathrm{run}\ \textbf{then}\ \mathrm{RULE}'\rangle\ \mathrm{INV}(\underline{\mathrm{x}},\underline{\mathrm{x}}') \end{array} \tag{11.1}$$

und zerfällt in 5 Fälle, je einen für jede der 5 Regeln der ASMs. Die anderen Beweisverpflichtungen (6.10), (6.8), (6.9) und (6.11) sind alle trivial, da *INV* die Formel $stop = stop'$ enthalten wird. Somit bleibt als einzig kritischer Punkt „nur noch“ das Finden einer geeigneten Zusammenhangsinvariante $INV(\underline{x}, \underline{x}')$, so daß die Formel (11.1) für jedes korrespondierende Paar von Regeln $RULE_i$, $RULE_{i'}$ gilt.

Eine erste Annäherung an die Gestalt von *INV* wird schon in [BR95], S. 17f gegeben. Dort wird eine Hilfsfunktion F vorgeschlagen, die die Knoten des Stacks von ASM2 auf korrespondierende Knoten des Suchbaums von ASM1 abbildet (siehe Abb. 11.3).

Schon in [Sch94] wird festgestellt, daß F nicht statisch definiert werden kann, sondern induktiv über die Zahl der Regelanwendungen konstruiert werden muß. Dies erfordert einen Formalismus, der die Modifikation einer Funktion während des Beweises erlaubt.

In DL ist die Lösung einfach, da wir dynamische Funktionen als Datentyp zur Verfügung stellen (siehe die Spezifikation *Dynfun* in Abschnitt 4.1). Somit ist F eine Datenstruktur, über die quantifiziert werden kann. Unsere Zusammenhangsinvariante sichert dann die Existenz einer geeigneten Funktion F für je zwei korrespondierende Zustände der beiden Interpreter ASM1 und ASM2. F wird dann durch Instanzierung modifiziert. Somit lassen sich die auf S. 17f

in [BR95] angegebenen Eigenschaften zu den folgenden Konjunktionsgliedern in unserer Zusammenhangsinvariante (für Variablen, die sowohl in ASM1 als auch in ASM2 vorkommen, verwenden wir wieder die Konvention, die Variable aus ASM2 mit einem Strich zu versehen):

$\exists$ F:

1 decglseq[currnode] = decglseqreg

2 sub[currnode] = subreg

3a mapclause(map(cll, cands[currnode]),db)
= mapclause′(clls(cllreg,db_2),db_2)

3b every(father,cands[currnode], currnode)

4 father[currnode] = F[breg]

5 decglseq[F[n]] = decglseq′[n]

6 sub[F[n]] = sub′[n]

7a mapclause(map(cll, cands[F[n]]),db)
= mapclause′(clls(cll′[n],db_2),db_2)

7b every(father, cands[F[n]], F[n])

8 father[F[n]] = b[n]

9 F[$\perp$] = $\perp$

In den Formeln bedeutet *every*(*father*,*cands*[n], n), daß n der Vaterknoten aller *cands*[n] ist.

Betrachtet man die Gleichungen 1 und 5 genauer, stellt man fest, daß sie so gar nicht gelten. Zwar sind die Goals identisch, jeder Cutpoint n muß aber durch $F[n]$ ersetzt werden. Bereits in [Sch94] wird deshalb eine entsprechende Funktion F_d mit den Axiomen

$$\mathrm{F_d}(\mathrm{F},[]) = []$$

$$\mathrm{F_d}(\mathrm{F},[\langle \mathrm{goal},\mathrm{ctpt}\rangle \mid \mathrm{dgl}]) = [\langle \mathrm{goal}, \mathrm{F}(\mathrm{ctpt})\rangle \mid \mathrm{F_d}(\mathrm{F}, \mathrm{dgl})]$$

definiert, sowie 1 und 5 durch

1 decglseq[currnode] = $\mathrm{F_d}$(F, decglseqreg)

5 decglseq[F[n]] = $\mathrm{F_d}$(F, decglseq′[n])

ersetzt. Außerdem werden die offensichtlichen Gleichungen:

10 stop = stop′ $\wedge$ mode = mode′ $\wedge$ vireg = vireg′

addiert. Die Formeln $1-10$ bildeten die erste Version der Zusammenhangsinvariante $INV(\underline{x}, \underline{x}')$ mit der die formale Verifikation mit dem KIV-System begonnen wurde.

Entwicklung der korrekten Zusammenhangsinvariante Es zeigt sich, daß die erste Variante nicht ausreicht, um die Korrektheit zu zeigen. Vielmehr waren ein Dutzend Iterationen notwendig, um die korrekte Invariante zu finden. Die Fehlversuche nahmen wesentlich mehr Zeit in Anspruch, als die erfolgreiche Verifikation mit der korrekten Invariante. Wir geben nun einen groben Überblick über die Suche, und zeigen wie versteckte Annahmen in den Beweisen sichtbar wurden. Beim Nachweis derartiger Annahmen zeigten sich weitere Lücken in den Annahmen usw., so daß insgesamt ein evolutionärer Beweisprozeß entstand.

Injektivität von *F* Nach nur 5 Minuten (und 6 Interaktionen) des ersten formalen Beweises erreichten wir das folgende Beweisziel:

$$F[breg] = F[\bot] \rightarrow breg = \bot \tag{11.2}$$

Die Formel gilt (vgl. Abb. 11.3), aber woraus folgt sie? Ein kurze Analyse des von KIV angezeigten Beweisbaums zeigt, daß die Beweissituation dadurch entsteht, daß wir versuchen, zu zeigen, daß genau dann, wenn ASM2 durch Backtracking anhält (mit *failure*), dies auch für ASM1 gilt.

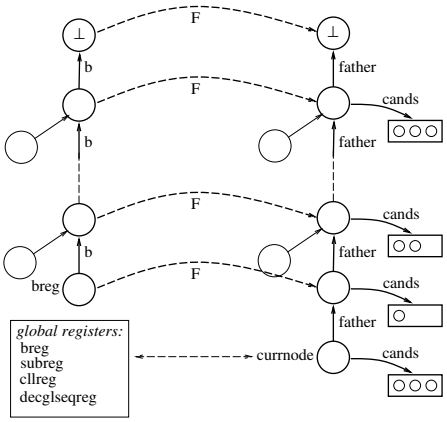


Abbildung 11.3

Offenbar wird die *Injektivität* von F benötigt, wie auch aus Abb. 11.3. ersichtlich wird. Wir addieren also

```
11  F injon s
```

zu *INV*, wobei *injon* durch

$$F\ injon\ s \equiv \forall\ n,n_1.\ n \in s \wedge n_1 \in s \wedge F[n] = F[n_1] \rightarrow n = n_1$$

definiert ist. Dies macht die Injektivität als Vorbedingung für jeden Regeläquivalenzbeweis verfügbar. Auf der anderen Seite ist es nun notwendig, die Injektivität auch zu zeigen.

Stack-Charakterisierung Unglücklicherweise, ist die Injektivität von F anzunehmen, zu grob. Der Beweisversuch scheitert mit einem Beweisziel, das verlangt, die Injektivität von $F[new(s') \leftarrow currnode]$ zu zeigen. Wir sind nicht in der Lage, zu zeigen, daß die *select*-Regel die Injektivität von F erhält (nach der *select*-Regel muß F den neuen Knoten $new(s')$ auf *currnode* abbilden). Eine genauere Analyse ergibt, daß es in der Tat Situationen gibt, in denen dies nicht möglich ist. Abb. 11.4 zeigt eine solche Situation, in der zwei verschiedene Knoten von ASM2 auf denselben Knoten von ASM1 abgebildet werden.

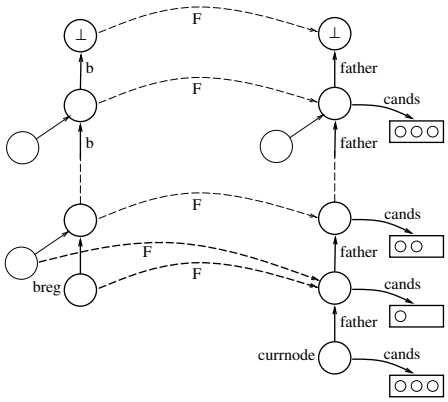


Abbildung 11.4

Das Problem entsteht durch bereits verworfene Knoten in ASM2, die nicht mehr im Stack liegen, d.h. nicht mehr von *breg* aus durch Verfolgen der durch die Funktion b gegebenen Verzeigerung zu erreichen sind, die aber dennoch in der Menge allokierter Knoten vorhanden sind. Die Funktion F ist aber momentan auch auf diesen Knoten definiert, was die Injektivität verletzt. Auf der kleineren Menge der Stackknoten ist F aber injektiv. Deshalb benötigen wir eine Charakterisierung des Stacks, so daß sich alle Eigenschaften, die wir über Knoten gemacht haben, auf die Stackknoten beschränken lassen.

Die Charakterisierung des Stacks ist auch notwendig, um ein weiteres Beweisziel desselben Beweises zu schließen:

$$\text{cands}[\text{currnode} \leftarrow \text{x}][\text{F[n]}] = \text{cands[F[n]]}$$

Hier muß bewiesen werden, daß die Modifikation von *cands* an der Stelle *currnode* keine Knoten in der Zielmenge von F berührt. Dazu benötigen wir zusätzlich:

12 $\text{F[n]} \neq \text{currnode}$

Aber auch diese Formel ist nicht immer wahr, wie man an Abb. 11.5 sieht, die einen Zustand nach Backtracking zeigt. Auch hier gilt lediglich, daß *currnode* nicht im Bild der *Stack*-Knoten unter F liegt.

Bei der Charakterisierung der Stackknoten ist zu beachten, daß die rekursive Definition einer Funktion *stackof* mit den Axiomen

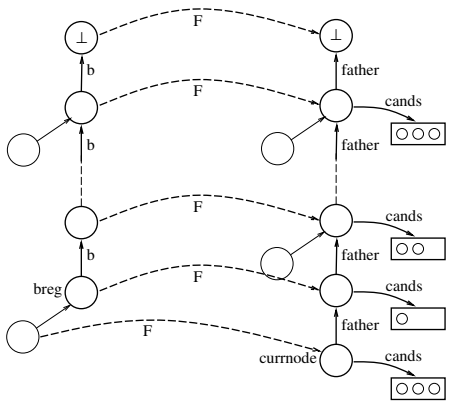


Abbildung 11.5

stackof(b,⊥) = [],

breg ≠ ⊥ → stackof(b,breg) = [breg | stackof(b,b[breg])]

zu einer inkonsistenten Spezifikation führt, da es möglich ist, dynamische Funktionen zu konstruieren, die eine zyklische Verzeigerung herstellen (für einen Knoten $n \neq \perp$ und eine beliebige Funktion b läßt sich für $b' := b[n \leftarrow n]$ die Formel $stackof(b', n) = [n|stackof(b', n)]$ herleiten, die dem Listenaxiom $x \neq [a|x]$ widerspricht).

Ein korrekter Ansatz zur Charakterisierung der Stackknoten ist es, ein Programm *STACK#* zu definieren (die Terminierung des Programms charakterisiert zyklenfreie Strukturen):

```
STACK#(n, b; var stack)
begin
if n = ⊥ then stack := [] else
  begin STACK#(b[n], b; stack); stack := [n | stack] end
end
```

Abbildung 11.6 : Charakterisierung von zyklenfreien Stacks

Definiert man nun ψ(n) als die Konjunktion aller Eigenschaften, die von einem selektierten Knoten n abhängen (5 bis 8 und 11) und φ als die Konjunktion der restlichen Formeln (1 bis 4, 9, 10 und 12), so hat die neue Zusammenhangsinvariante *INV* die Form:

$$\exists \text{ F: } \varphi \wedge \langle \text{STACK\#(breg, b; stack)} \rangle \ (\forall \text{ n. n} \in \text{stack} \rightarrow \psi(\text{n})) \qquad (11.3)$$

Sie besagt also, daß (für ein geeignetes F) φ gilt, und daß *STACK#* terminiert und eine Ergebnisliste *stack* liefert, so daß ψ für alle Elemente dieser Liste gilt.

Cutpoints Der Äquivalenzbeweis für die beiden *cut*-Regeln mit dieser Version von *INV* zeigt eine neue Schwierigkeit: ψ muß für den neuen Stack garantiert werden, der durch die Anwendung der *cut*-Regel abgeändert wurde. Dieser Stack beginnt beim gerade berechneten neuen *breg*, welches auf den ersten Cutpoint der bisherigen *decglseq* gesetzt worden. Nun ist klar, daß der neue Stack die Eigenschaften ψ vom alten erbt, da er ein Teil des alten Stacks vor der Regelanwendung ist. Dies kann aber nicht bewiesen werden, da nicht bekannt ist, daß die Cutpoints immer Knoten aus dem aktuellen Stack sind. Diese Eigenschaft muß also zur Invariante addiert werden:

Deshalb definieren wir ein entsprechendes Prädikat *cutptsin* (infix geschrieben), durch die Axiome

$$\begin{array}{l}\text{[] cutptsin stack,}\\ \text{[⟨goal,ctpt⟩ |dgl] cutptsin stack} \leftrightarrow \text{ctpt} \in \text{stack} \wedge \text{dgl cutptsin stack}\end{array} \tag{11.4}$$

und fordern:

decglseqreg cutptsin stack

In der ersten Version überprüfte die Definition von *cutptsin* lediglich, ob alle Cutpoints des ersten Arguments im zweiten enthalten sind. Da jedes *decglseq*[n] durch Backtracking in das *decglseqreg* geladen werden kann, muß die Eigenschaft auch für alle Stackknoten

decglseq'[n] cutptsin (stack from b[n])

gefordert werden. Dabei ist *stack from* $b[n]$ der bei $b[n]$ beginnende Teilstack von *stack*. Die infix geschriebene Funktion *from* kann durch

[] from n = [],

n ≠ n' → [n|l] from n' = l from n',

[n|l] from n = [n|l]

definiert werden. Durch die neuen Formeln ändert sich *INV* zu:

```
∃ F.   φ
     ∧ ⟨STACK#(breg, b; stack)⟩
           (   decglseqreg cutptsin stack
            ∧ (∀ n.     n ∈ stack
                     →    ψ(n)
                        ∧ decglseq'[n] cutptsin (stack from b[n])
```

Immer noch ist diese Invariante aber nicht stark genug. Der Beweis scheitert, da bei der Anwendung der *cut*-Regel nicht sichergestellt ist, daß die weiter hinten im *decglseqreg* stehenden Cutpoints im verkürzten Stack bleiben. Diese Eigenschaft gilt nur, weil die Cutpoints im *decglseqreg* in der *richtigen Reihenfolge* (vgl. Abb. 11.7) im Stack stehen. Deshalb muß die Axiomatisierung (11.4)

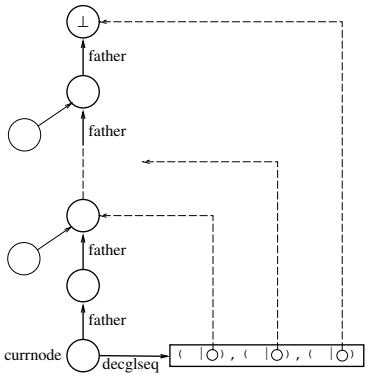


Abbildung 11.7

cutptsin zu

[] cutptsin stack,

[⟨goal,ctpt⟩ |dgl] cutptsin stack
↔ ctpt ∈ stack
∧ dgl cutptsin (stack **from ctpt**)

verstärkt werden, wobei *INV* syntaktisch unverändert bleiben kann. Die Änderung hat in diesem Fall keine Auswirkungen auf die bereits geführten Beweise, da die bis dahin in ihnen verwendeten Lemmata über *cutptsin* gültig bleiben (was vom „Korrektheitsmanagement“ von KIV überprüft wird).

Weitere Eigenschaften Immer noch ist die Zusammenhangsinvariante unvollständig. Einige weitere Beweisversuche zeigen nun, daß es auch notwendig ist, Eigenschaften des Suchbaums von ASM1 explizit zu machen. Einige dieser Eigenschaften sind (informell): Kein Kandidatenknoten ist im Bild von F, keine Kandidatenliste hat Duplikate, je zwei Kandidatenlisten sind disjunkt, usw. Als Ergebnis erhält man schließlich nach insgesamt einem Dutzend Versuchen (Tippfehlerkorrekturen nicht mitgezählt) die folgende Zusammenhangsinvariante. Alle in der Invariante aufgeführten Eigenschaften wurden tatsächlich im Beweis verwendet.
$\mathrm{INV}_{12} \equiv$

$\exists$ F. stop = stop′ ∧ mode = mode′ ∧ vireg = vireg′ ∧ subreg = sub[currnode]

$\wedge$ F[$\bot$] = $\bot$ $\wedge$ F[breg] = father[currnode] $\wedge$ $\bot \neq$ currnode
$\wedge$ F_d(F, decglseqreg) = decglseq[currnode]
$\wedge$ $\bot \in s' \wedge \bot \in s \wedge$ currnode $\in$ s
$\wedge$ (mode = select
$\rightarrow$ $\langle$CLLS#(cllreg, db_2;col)$\rangle$
mapclause$'$(col, db_2) = mapclause(map(cll, cands[currnode]), db)
$\wedge$ every(father, cands[currnode], currnode)
$\wedge \neg$ currnode $\in$ cands[currnode] $\wedge \neg \bot \in$ cands[currnode]
$\wedge$ cands[currnode] $\subseteq$ s $\wedge$ nodups(cands[currnode]))
$\wedge$ $\langle$STACK#(breg, b; stack)$\rangle$
(decglseqreg cutptsin stack $\wedge$ candsdisjoint(F, cands, stack)
$\wedge$ F injon stack
$\wedge$ nocands(F, cands, stack) $\wedge$ stack $\subseteq$ s$'$
$\wedge \forall$ n. n $\in$ stack
$\rightarrow$ sub$'$[n] = sub[F[n]] $\wedge$ F[b[n]] = father[F[n]]
$\wedge$ F_d(F, decglseq$'$[n]) = decglseq[F[n]]
$\wedge$ $\langle$CLLS#(cll$'$[n],db_2;col)$\rangle$
mapclause$'$(col,db_2) = mapcl(map(cll, cands[F[n]]), db)
$\wedge$ every(father, cands[F[n]], F[n])
$\wedge$ F[n] $\neq$ currnode $\wedge$ F[n] $\in$ s $\wedge$ nodups(cands[F[n]])
$\wedge$ cands[F[n]] $\subseteq$ s $\wedge \neg$ currnode $\in$ cands[F[n]]
$\wedge$ (mode = select
$\rightarrow$ $\neg$ F[n] $\in$ cands[currnode]
$\wedge$ disjoint(cands[F[n]], cands[currnode]))
$\wedge$ decglseq$'$[n] cutptsin (stack from b[n]))

Kapitel 12

2/3: Wiederverwendung von Choicepoints

12.1 Definition von ASM3

Obwohl ASM2 schon weniger Knoten allokiert als ASM1, gibt es noch zwei weitere Möglichkeiten zur Optimierung, die in ASM3 und ASM4 ausgenutzt werden.

Dieser Abschnitt beschreibt zunächst die Wiederverwendung von Choicepoints. Wir folgen dabei [BR95], Kapitel 1.3. Die Optimierung kann am einfachsten am Beispiel des vorigen Abschnitts gezeigt werden: Wenn dort die erste Alternative für den Aktivator `p` versucht wird, wird ein neuer Knoten A allokiert, und die Werte $decglseq[A]$, $sub[A]$ und $cll[A]$ für den neuen Choicepoint gesetzt.

Da die erste Alternative zu keiner Lösung führt, wird der Knoten A schließlich durch Backtracking verlassen. Der Knoten A wird aus dem Stack entfernt, und ist nicht mehr zugreifbar. Die anschließend ausgeführte *select*-Regel allokiert für die zweite Alternative für einen neuen Knoten A'. Dieser erhält dieselben Werte wie A, außer daß $cll[A']$ gegenüber $cll[A]$ inkrementiert wurde (siehe Abb. 11.2, S. 90 in Abschnitt 11.2).

Die Optimierung vermeidet nun die Deallokation und erneute Allokation von Choicepoints. Stattdessen wird der alte Choicepoint wiederverwendet. Um dies zu realisieren, wird das Entfernen des Choicepoints im else-Zweig von Backtracking gestrichen, und eine neue *retry*-Regel definiert, die durch das Setzen von $mode := retry$ aktiviert wird. Die Retry-Regel kombiniert nun den Effekt des else-Zweiges von Backtracking und der folgenden *select*-Regel: Sie entfernt dem aktuellen Choicepoint n nur, falls keine Alternative mehr vorhanden ist, und inkrementiert andernfalls den Wert von $cll[n]$. Die bisherige Select-Regel wird nun nur noch für die erste Alternative für einen Aktivator verwendet, und wird in *try*-Regel umbenannt. Der Test der bisherigen *select*-Regel, ob noch eine Alternative existiert, kann deshalb in die *call*-Regel vorverlegt werden. Da die

retry- und *try*-Regel nun große Teile gemeinsam hätten (Unifikation mit dem Aktivator, Erhöhen von *vireg* etc.), wird, um Redundanz zu vermeiden, der gemeinsame Teil der beiden Regeln in eine *enter*-Regel geschoben, die durch *mode* := *enter* aktiviert wird. Insgesamt ergibt sich so die folgende Regelmenge

call rule
if mode = call ∧ is_user_defined(act)
then if clause(procdef_2(act,db_2)) = null
then backtrack
else cllreg := procdef_2(act,db_2)
ctreg := breg
mode := try

enter rule
if mode = enter
then let cla = rename(clause(cllreg,db_2),vireg)
let mgu = unify(act, hd(cla))
if mgu = nil
then backtrack
else decglseqreg := mgu $\hat{}_d$ [<bdy(cla),ctreg> | cont]
subreg := subreg ∘ mgu
vireg := vireg +1
mode := call

goal success rule
if goal = [] ∧ decglseqreg ≠ []
then decglseqreg := cdr(decglseqreg)

query success rule
if decglseqreg = [] **then** stop := halt
subst := subreg

try rule
if mode = try
then mode := enter
let tmp = new(s)
s := s ∪ {tmp}
breg := tmp
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
cll[tmp] := cllreg +1

retry rule
if mode = retry
then if clause(cll[breg],db_2) = null
then deep-backtrack

else decglseqreg := decglseq[breg]
subreg := sub[breg]
cllreg := cll[breg]
ctreg := b[breg]
mode := enter

cut rule
if act = ! **then** father := cutpt
decglseqreg := cont

fail rule
if act = fail **then backtrack**

wobei

backtrack ≡
if breg = ⊥
then stop := halt
subst := failure
else mode := retry

Zu beachten ist noch, daß die *enter*-Regeln ein neues Register *ctreg* verwendet, um den Cutpoint *ctpt* des neu berechneten *decglseqreg* zu setzen. Dies ist notwendig, da nach der *retry*-Regel nun *b*[*breg*] statt wie bisher *breg* als *ctpt* verwendet werden muß. Die *call*- und die *retry*-Regel übernehmen das Setzen von *ctreg*.

12.2 Äquivalenzbeweis 2/3

Die Beschreibung der Optimierung von ASM2 nach ASM3 legt nahe, bei der Verifikation der Verfeinerung nicht einzelne Regeln zu betrachten, sondern zunächst zueinander korrespondierende Zustände zu suchen, und dann Gruppen aufeinanderfolgender Regelanwendungen zu suchen, die die Korrespondenz aufrecht erhalten. Zwei offensichtlich korrespondierende Zustände sind diejenigen, in denen beide ASMs im *call*-Modus sind. In diesen sind die Belegungen der Register und die beiden Choicepoint-Stacks offenbar identisch (modulo Umbenennung der Stackknoten). Nur wenig komplizierter ist der Zusammenhang zwischen den beiden Zuständen, wenn ASM3 ein *retry* und ASM2 das zugehörige *select* ausführt. In diesem Fall stimmen die Registerinhalte von ASM2 mit dem obersten Choicepoint des Stacks von ASM3 überein, und der Rest des Stacks von ASM3 ist identisch zum Stack von ASM2. Schreibt man *regs*, *stack* bzw. *regs*′, *stack*′ für die Register und den Stack von ASM2 bzw. ASM3, so ergibt sich daraus ein erster Ansatz für eine Zusammenhangsinvariante zu

$$\mathrm{INV23}(\mathrm{regs},\mathrm{stack},\mathrm{regs}',\mathrm{stack}') \equiv \mathrm{CINV} \vee \mathrm{RINV}$$

mit

$\text{CINV} \equiv \text{mode} = \text{call} \wedge \text{mode}' = \text{call} \wedge \text{regs} = \text{regs}' \wedge \text{stack} = \text{stack}'$,

$\text{RINV} \equiv \text{mode} = \text{select} \wedge \text{mode}' = \text{retry} \wedge \text{stack}' = \text{push(regs,stack)}$

Betrachtet man die Regelfolgen, die von korrespondierenden Zuständen wieder zu korrespondierenden führen, so erhält man die kommutierenden Diagramme aus Abb. 12.1.

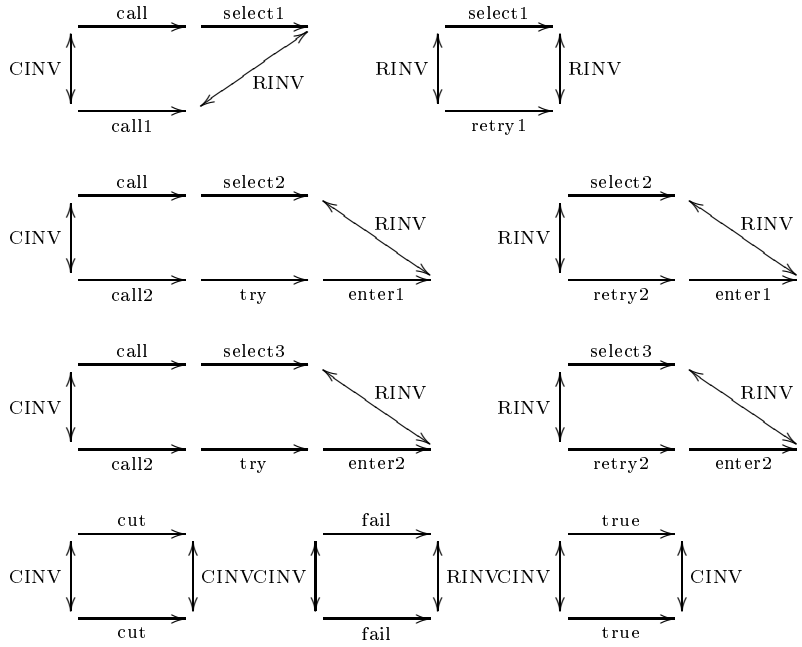


Abbildung 12.1 : Kommutierende Diagramme für die Verfeinerung 2/3

select1, *select2* und *select3* sind dabei die drei Unterfälle der *select*-Regel, analog sind *retry1* etc. definiert. Die in Kapitel 6 gegebene Theorie zeigt, daß der Kommutativitätsnachweis für die angegebenen Diagramme ausreichend ist, um die Äquivalenz von ASM2 und ASM3 zu zeigen (die quantifizierten Variablen i und j in der Beweisverpflichtung (6.5) sind nach Fallunterscheidung über die möglichen Regelpaare einfach entsprechend der Größe der Diagramme zu instantiieren). Die kommutierenden Diagramme wie auch der erste Ansatz für eine Zusammenhangsinvariante stimmen mit dem in [BR95] gegebenen überein.

Allerdings sieht man für die formale Verifikation sofort, daß erneut eine Abbildung F zwischen den Knoten notwendig ist (ASM3 allokiert ja weniger Knoten als ASM2). Somit treten erneut ähnliche Probleme auf, die schon bei der ersten Verfeinerung eine Rolle spielten, wie die Injektivität von F auf dem aktuellen Stack, sowie die *cutptsin*-Eigenschaft. Neu benötigt wird gegenüber der

Verifikation von 1/2 die Eigenschaft, daß alle *decglseq*[*n*] nichtleer sind, und das erste Goal darin mit einem benutzerdefinierten Literal beginnt (wir schreiben wieder *goal*[*n*] und *act*[*n*] für diese Komponenten). Nur so kann gezeigt werden, daß die auf Backtracking folgende Regel nur ein *retry* sein kann, und nicht etwa ein *goal success*.

Die Verwendung der Theorie aus Kapitel 6 erleichtert die Verifikation enorm, da es keine Notwendigkeit gibt, eine Zusammenhangsinvariante für die Zwischenzustände innerhalb der Diagramme auszuarbeiten (siehe dazu auch den Vergleich zu Isabelle in Abschnitt 20).

Ein ersten Versuch, alle Diagramme als kommutierend nachzuweisen, gelang nach 2 Wochen. Dieser erste Versuch verwendete eine Vorversion der jetzt vorliegenden Theorie. Diese erlaubte zwar die Verwendung beliebiger kommutierender Diagramme, erforderte aber noch einen getrennten Korrektheits- und Vollständigkeitsbeweis mit zwei verschiedenen Zusammenhangsinvarianten, sowie (wie sich inzwischen gezeigt hat) den Nachweis des Modularisierungstheorems für die Instanz 2/3. 8 Versuche waren notwendig, um die beiden korrekten Zusammenhangsinvarianten zu finden.

In einem zweiten Versuch, der die jetzt vorliegende Theorie verwendet, konnte die Äquivalenz von ASM2 und ASM3 in wenigen Stunden bewiesen werden. Diese Zeitdauer ist natürlich durch den schon erfolgreichen ersten Versuch günstig beeinflußt, ein etwas realeres Bild ergibt sich aus einem Vergleich der Zahl der notwendigen Interaktionen in den erfolgreichen Beweisen: Statt 234 Interaktionen waren 75 notwendig, um die folgende Invariante als korrekt nachzuweisen:

$$
\begin{array}{l}
\mathrm{INV}_{23} \equiv \\
\quad \mathrm{stop} = \mathrm{stop}' \\
\wedge\ (\mathrm{stop} = \mathrm{success} \rightarrow \mathrm{subreg} = \mathrm{subreg}') \\
\wedge\ \bot \in \mathrm{s} \wedge \bot \in \mathrm{s}' \\
\wedge\ (\qquad \mathrm{stop} \neq \mathrm{run} \\
\qquad\quad (*\ \mathrm{CINV}\ *) \\
\quad \vee \quad \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{mode} = \mathrm{call} \wedge \mathrm{mode}' = \mathrm{call} \\
\qquad \wedge\ \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{subreg} = \mathrm{subreg}' \\
\qquad \wedge\ (\exists\ \mathrm{F}. \quad \mathrm{F}[\bot] = \bot \wedge \mathrm{breg} = \mathrm{F}[\mathrm{breg}'] \\
\qquad\qquad\quad \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseqreg}') = \mathrm{decglseqreg} \\
\qquad\qquad\quad \wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}) \rangle \\
\qquad\qquad\qquad (\quad \langle \mathrm{STACK\#}(\mathrm{breg}, \mathrm{b}; \mathrm{stack}_0) \rangle\ \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) = \mathrm{stack}_0 \\
\qquad\qquad\qquad\quad \wedge\ \mathrm{F\ injon\ stack} \wedge \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) \subseteq \mathrm{s} \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\qquad\qquad\qquad\quad \wedge\ \mathrm{decglseqreg}'\ \mathrm{cutptsin\ stack} \\
\qquad\qquad\qquad\quad \wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad\qquad \rightarrow \quad \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}[\mathrm{F}[\mathrm{n}]] \wedge \mathrm{cll}'[\mathrm{n}] = \mathrm{cll}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseq}'[\mathrm{n}]) = \mathrm{decglseq}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{decglseq}'[\mathrm{n}]\ \mathrm{cutptsin\ cdr(stack\ from\ n)} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}'[\mathrm{n}] \neq [] \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{n}])))) \\
\qquad\quad (*\ \mathrm{RINV}\ *) \\
\quad \vee \quad \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{mode} = \mathrm{select} \wedge \mathrm{mode}' = \mathrm{retry} \\
\qquad \wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \wedge \mathrm{decglseqreg} \neq [] \wedge \mathrm{goal} \neq [] \\
\qquad \wedge\ \mathrm{is\_user\_defined(act)} \wedge \mathrm{breg}' \neq \bot \\
\qquad \wedge\ \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{sub}'[\mathrm{breg}'] = \mathrm{subreg} \wedge \mathrm{cll}'[\mathrm{breg}'] = \mathrm{cllreg} \\
\qquad \wedge\ (\exists\ \mathrm{F}.\ \langle \mathrm{STACK\#}(\mathrm{b}'[\mathrm{breg}'], \mathrm{b}'; \mathrm{stack}) \rangle \\
\qquad\qquad\quad (\quad \langle \mathrm{STACK\#}(\mathrm{breg}, \mathrm{b}; \mathrm{stack}_0) \rangle\ \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) = \mathrm{stack}_0 \\
\qquad\qquad\quad\ \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseq}'[\mathrm{breg}']) = \mathrm{decglseqreg} \wedge \mathrm{F}[\bot] = \bot \\
\qquad\qquad\quad\ \wedge\ \mathrm{F\ injon\ stack} \wedge \mathrm{F_l}(\mathrm{F}, \mathrm{stack}) \subseteq \mathrm{s} \wedge \mathrm{stack} \subseteq \mathrm{s}' \wedge \mathrm{breg}' \in \mathrm{s}' \\
\qquad\qquad\quad\ \wedge\ \mathrm{decglseq}'[\mathrm{breg}']\ \mathrm{cutptsin\ stack} \\
\qquad\qquad\quad\ \wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad \rightarrow \quad \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}[\mathrm{F}[\mathrm{n}]] \wedge \mathrm{cll}'[\mathrm{n}] = \mathrm{cll}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{F_d}(\mathrm{F}, \mathrm{decglseq}'[\mathrm{n}]) = \mathrm{decglseq}[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{decglseq}'[\mathrm{n}]\ \mathrm{cutptsin\ cdr(stack\ from\ n)} \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}[\mathrm{n}] \neq [] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{is\_user\_defined}(\mathrm{act}[\mathrm{n}])))))
\end{array}
$$

Kapitel 13

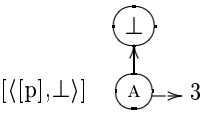
3/4: Entfernung leerer Choicepoints

13.1 Definition von ASM4

In der Verfeinerung von ASM2 zu ASM3 wurde die unnötige Deallokation und Reallokation von Choicepoints beseitigt. Es gibt aber noch eine weitere Optimierungsmöglichkeit, und zwar Choicepoints mit einer leeren Liste von Alternativen („leere Choicepoints“).

Zum Beispiel zeigen beide Choicepoints A' (in ASM3 wird hier A wiederverwendet) und C der Abb. 11.2, S. 90 aus Abschnitt 11.2 auf eine leere Klauselliste, d.h. es gilt $clause(cll[A'],db_2) = clause(cll[C],db_2) = null$. Wird ein leerer Choicepoint durch eine *retry*-Regel besucht, so wird direkt *deep-backtrack* aufgerufen, was einfach dazu führt, daß der Choicepoint gelöscht wird. Deshalb ist es sinnvoll, die Entstehung derartiger Choicepoints durch vorausschauende Tests zu verhindern. Für die *try*-Regel bedeutet dies, keinen Choicepoint anzulegen, wenn $procdef_2(act,db_2)$ nur eine einzelne Klausel liefert. In der *retry*-Regel kann der Choicepoint, statt ihn zu modifizieren, ganz beseitigt werden, wenn seine Alternativen leer werden. Im Gegenzug kann dann der Test auf einen leeren Choicepoint in der *retry*-Regel entfallen. Dem Zustand von ASM2 aus Abb. 11.2 entspricht in ASM4 dann der in Abb. 13.1 gezeigte.

Die modifizierte *try*- und *retry*-Regel bekommen also die Form:



decglseqreg = [⟨[],A⟩,⟨[!,true],⊥⟩,⟨[],⊥⟩]
breg = A

Abbildung 13.1

try rule
if mode = try
then mode := enter
if clause′(cllreg +1, db_2) ≠ null
then let tmp = new(s)
s := s ∪ {tmp}
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
cll[tmp] := cllreg +1
breg := tmp

retry rule
if mode = retry
then decglseqreg := decglseq[breg]
subreg := sub[breg]
cllreg := cll[breg]
ctreg := b[breg]
mode := enter
/* look ahead guard */
if clause(cll[breg] +1,db_2) ≠ null
then cll[breg] := cll[breg] +1
else breg := b[breg]

13.2 Äquivalenzbeweis 3/4

Um die Äquivalenz von ASM3 und ASM4 zu zeigen, ist offenbar eine Bijektion F notwendig, die nichtleere Choicepoints von ASM3 und entsprechenden Choicepoints in ASM4 in Beziehung setzt. Die Richtung der Definition ist unerheblich, sie bestimmt lediglich, welcher der beiden Stacks durch einen Aufruf von *STACK#* berechnet werden muß. In Anlehnung an [BR95] haben wir den Stack von ASM3 auf den von ASM4 abgebildet.

Als wesentliches Problem bei der Erstellung einer Zusammenhangsinvariante bleibt nun die Zuordnung der Cutpoints. Dazu definieren wir ein Programm *F#*, das jeden Cutpoint von ASM3 auf den nächsttieferliegenden nichtleeren abbildet. Das Programm *G#* wendet *F#* auf alle Cutpoints einer *decglseq* an. Anwendung von *G#* und dann von *F* (mit F_d) auf eine *decglseq* von ASM3 ergibt dann eine *decglseq* von ASM4. Eine prädikatenlogische Definition kommt wegen möglicher zyklischer Verzeigerungen wieder nicht in Frage. Abb. 13.2 zeigt die Zuordnung zwischen zwei Choicepoint-Stacks, wobei leere Choicepoints aus ASM3 durch einen nicht ausgefüllten Kreis symbolisiert sind.

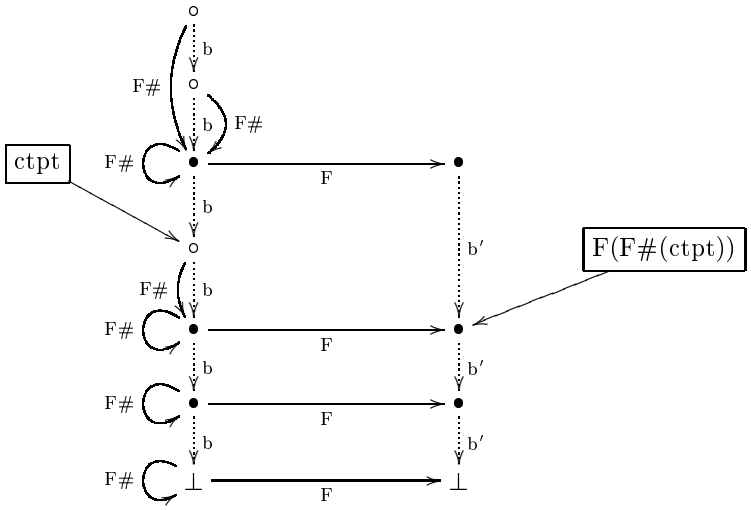


Abbildung 13.2 : Zuordnung der Choicepoints von ASM3 zu ASM4

Die formale Definition einer Prozedur *F#*, sowie einer Prozedur *G#*, die *F#* auf alle Cutpoints *decglseqreg* anwendet, lautet:

F#(n,b,cll,db_2;**var** n_0)
begin
if n = ⊥
then n_0 := n
else if clause(cll[n],db_2) = null
  then F#(b[n],b,cll,db_2;n_0)
  else n_0 := n
end

G#(decglseqreg,b,cll,db_2;**var** $decglseqreg_0$)
begin
if decglseqreg = [] **then** $decglseqreg_0$:= [] **else**
var $ctpt_0$, $cont_0$ **in**
  begin

```
    F#(ctpt,b,cll,db2;ctpt0);
    G#(cont,b,cll,db2;cont0);
    decglseqreg0 := [⟨act, ctpt0⟩ | cont0]
    end
end
```

Die hier gegebene Definition korrigiert und vereinfacht die in [BR95] enthaltene Definition. Als ersten Ansatz für eine Zusammenhangsinvariante erhält man mit diesen Überlegungen:

$$
\begin{array}{l}
\mathrm{INV}_{34} \equiv \\
\exists\ \mathrm{F}. \\
\quad \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{subreg} = \mathrm{subreg}' \\
\wedge\ \mathrm{cllreg} = \mathrm{cllreg}' \wedge \mathrm{F}[\bot] = \bot \wedge \mathrm{mode} = \mathrm{mode}' \\
\wedge\ \langle \mathrm{F\#(breg,\ b,\ cll,\ db_2;\ breg_0)} \rangle\ \mathrm{F}[\mathrm{breg}_0] = \mathrm{breg}' \\
\wedge\ \langle \mathrm{STACK\#(breg,b;stack)} \rangle \\
\qquad \langle \mathrm{G\#(decglseqreg,b,cll,db_2;bf\ var\ decglseqreg_0)} \rangle \\
\qquad\quad \mathrm{F_d(F,decglseqreg_0)} = \mathrm{decglseqreg}' \\
\qquad \wedge\ \forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \\
\qquad\quad \rightarrow \quad \mathrm{sub[n]} = \mathrm{sub}'\mathrm{[n]} \wedge \mathrm{cll[n]} = \mathrm{cll}'\mathrm{[n]} \\
\qquad\quad\ \wedge\ \langle \mathrm{F\#(b[n],\ b,\ cll,\ db_2;\ n_0)} \rangle \\
\qquad\quad\quad \mathrm{F[n_0]} = \mathrm{b}'\mathrm{[F[n]]}
\end{array}
$$

Die beiden Konjunktionsglieder mit Aufrufen von $F\#$ sowie $F[\bot] = \bot$ geben dabei die Konstruktion des ASM4-Stacks aus dem ASM3-Stack wieder. Für die Zuordnung der Regeln ist klar, daß die meisten Regelanwendungen aus ASM3 auch eine korrespondierende Regelanwendung in ASM4 besitzen. Lediglich Aufrufe der *retry*-Regel, die einen leeren Choicepoint durch Aufruf von *deep-backtrack* entfernen, haben keinen korrespondierenden Schritt in ASM4. Es ergibt sich also ein 1:0-Diagramm für diesen Fall und 1:1-Diagramme sonst. Die Funktion *ndt* aus Kapitel 6 hat also im Gegensatz zu den bisherigen Verfeinerungen nicht mehr konstant den Wert *mn*. Sie ist vielmehr durch

$$
\begin{array}{l}
\quad \mathrm{stop} = \mathrm{run} \wedge \mathrm{decglseqreg} \neq [] \wedge \mathrm{goal} \neq [] \\
\wedge\ \mathrm{mode} = \mathrm{retry} \wedge \mathrm{clause(cll[breg],db_2)} = \mathrm{null} \\
\supset\ \mathrm{ndt}(\underline{\mathrm{x}},\underline{\mathrm{x}}') = \mathrm{m0}\ ;\ \mathrm{ndt}(\underline{\mathrm{x}},\underline{\mathrm{x}}') = \mathrm{mn}
\end{array}
$$

zu definieren[1]. $\underline{x}$ =*decglseqreg, decglseq, stop, ...* bzw. $\underline{x}'$ = *decglseqreg'*, *decglseq'*, *stop'*, ... bezeichnen dabei wie immer die Vektoren aller (zu Programmvariablen übersetzten) dynamischen Funktionen von ASM3 bzw. ASM4. Für die Verifikation der Äquivalenz von ASM3 und ASM4 benötigen wir wegen der Existenz dreieckiger Diagramme nun entsprechend der Theorie aus Kapitel 6 eine Funktion *exec0n*. Diese sollte die Anzahl der aufeinanderfolgenden 1:0-Diagramme, d.h. der aufeinanderfolgenden *deep-backtrack*-Aufrufe beschränken.

[1] $A \supset B; C$ kürzt $(A \rightarrow B) \wedge (\neg\ A \rightarrow C)$ ab, siehe Anhang B.

Eine derartige Beschränkung ist hier offensichtlich durch die Höhe des ASM3-Stacks gegeben. Die Beweisverpflichtung (6.6) aus Kapitel 6 läßt sich dann zu

$$\begin{aligned}
&\quad \text{stop} = \text{run} \wedge \text{INV}_{34} \wedge \text{decglseqreg} \neq [] \wedge \text{goal} \neq [] \\
&\wedge\ \text{mode} = \text{retry} \wedge \text{clause(cll[breg],db}_2) = \text{null} \\
&\wedge\ \langle \text{STACK\#(breg,b;stack)} \rangle\ \#(\text{stack}) = \text{m} \\
&\rightarrow \langle \text{RULE}_3 \rangle\ (\quad \text{INV}_{34} \\
&\qquad\qquad \wedge\ (\langle \text{STACK\#(breg,b;stack)} \rangle\ \#(\text{stack}) < \text{m} \vee \text{stop} = \text{failure}))
\end{aligned}$$

instanzieren. Das Disjunktionsglied $ndt(\underline{x}, \underline{x}') \neq m0$ aus der Nachbedingung der generischen Beweisverpflichtung wurde dabei zu $stop = failure$ verschärft, da dies offensichtlich der einzige Fall ist, in dem ASM3 den Stack nicht verkleinert.

Zu beachten ist für die Verifikation nun, daß die Vorbedingung *keine* Anforderung $stop' = run$ enthält. Im Gegenteil ist laut Beweisverpflichtung (6.9) aus Kapitel 6 zu zeigen, daß

$$\text{stop} = \text{run} \wedge \text{stop}' \neq \text{run} \wedge \text{INV}_{34} \rightarrow \text{ndt}(\underline{\text{x}},\underline{\text{x}}') = \text{m0}$$

gilt. Daraus ergibt sich ein wesentliches Problem für die Verifikation: Es ist möglich, daß ASM4 bereits terminiert hat, während ASM3 noch leere Choicepoints abarbeiten muß. Das Problem der *nichtsimultanen Terminierung* verursacht eine Reihe von Zusatzkomplikationen für die Zusammenhangsinvariante, da sichergestellt werden muß, daß sie auch für die beschriebene Situation gilt. Es gilt dann nämlich nicht mehr $stop = stop'$, und auch $mode = mode'$ wird verletzt, wie man sich leicht überzeugt. Die beiden Eigenschaften müssen in der Invariante zu

$$\begin{aligned}
&(\text{stop}' \neq \text{failure} \rightarrow \text{stop} = \text{stop}' \wedge \text{mode} = \text{mode}') \\
&\wedge\ (\quad \text{stop}' = \text{failure} \wedge \text{stop} \neq \text{failure} \\
&\quad \rightarrow \text{mode} = \text{retry} \wedge \text{breg}' = \bot)
\end{aligned}$$

abgeschwächt werden. Zusammen mit der Eigenschaft

$$\langle \text{F\#(breg, b, cll, db}_2\text{; breg}_0) \rangle\ \text{F[breg}_0\text{]} = \text{breg}'$$

aus der bisherigen Invariante wird nun garantiert, daß im kritischen Fall alle Choicepoints im verbleibenden Stack von ASM3 leer sind.

Der so erhaltene erste Ansatz für die Invariante reicht wie immer noch nicht aus, um die Äquivalenz der beiden ASMs zu zeigen. Benötigt werden wie in 1/2 und 2/3 die Injektivität der Funktion F (diesmal nur auf den *nichtleeren* Choicepoints), die *cutptsin*-Eigenschaft, sowie die Existenz von $act[n]$ für jeden Choicepoint n. Dazu kommen eine Reihe von Vorbedingungen für die einzelnen Regelanwendungen wie $mode' = retry \rightarrow breg' \neq \bot$, sowie eine Charakterisierung des *ctreg* bzw. *ctreg'*-Registers. Diese Eigenschaften sind aber sehr einfach zu sehen, so daß nach zwei Wochen und 5 Iterationen die folgende, korrekte

Zusammenhangsinvariante bestimmt werden konnte:

$$
\begin{array}{l}
\mathrm{INV}_{34} \equiv \\
\exists\ \mathrm{F}. \\
\quad (\mathrm{mode} = \mathrm{try} \rightarrow \mathrm{ctreg} = \mathrm{breg} \wedge \mathrm{clause}'(\mathrm{cllreg}, \mathrm{db}_2) \neq \mathrm{null}) \\
\wedge\ (\quad \mathrm{mode} = \mathrm{enter} \\
\quad \rightarrow \quad \mathrm{breg} \neq \bot \wedge \mathrm{ctreg} = \mathrm{b}[\mathrm{breg}] \wedge \mathrm{subreg} = \mathrm{sub}[\mathrm{breg}] \\
\quad \wedge\ \mathrm{clause}'(\mathrm{cllreg}, \mathrm{db}_2) \neq \mathrm{null} \wedge \mathrm{cllreg}{+}1 = \mathrm{cll}[\mathrm{breg}] \\
\quad \wedge\ \mathrm{decglseqreg} = \mathrm{decglseq}[\mathrm{breg}]) \\
\wedge\ (\mathrm{mode}' = \mathrm{retry} \rightarrow \mathrm{breg}' \neq \bot) \\
\wedge\ (\mathrm{mode}' = \mathrm{try} \rightarrow \mathrm{ctreg}' = \mathrm{breg}' \wedge \mathrm{clause}'(\mathrm{cllreg}', \mathrm{db}_2) \neq \mathrm{null}) \\
\wedge\ (\mathrm{mode}' = \mathrm{enter} \rightarrow \mathrm{clause}'(\mathrm{cllreg}', \mathrm{db}_2) \neq \mathrm{null}) \\
\wedge\ (\quad \mathrm{mode}' = \mathrm{enter} \wedge \mathrm{clause}'(\mathrm{cllreg}'{+}1, \mathrm{db}_2) \neq \mathrm{null} \\
\quad \rightarrow \mathrm{breg}' \neq \bot \wedge \mathrm{ctreg}' = \mathrm{b}'[\mathrm{breg}']) \\
\wedge\ (\mathrm{mode}' = \mathrm{enter} \wedge \mathrm{clause}'(\mathrm{cllreg}'{+}1, \mathrm{db}_2) = \mathrm{null} \rightarrow \mathrm{ctreg}' = \mathrm{breg}') \\
\\
\wedge\ \mathrm{F}[\bot] = \bot \wedge \bot \in \mathrm{s} \wedge \bot \in \mathrm{s}' \wedge \mathrm{breg} \in \mathrm{s} \wedge \mathrm{ctreg} \in \mathrm{s} \\
\wedge\ \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{subreg} = \mathrm{subreg}' \wedge \mathrm{cllreg} = \mathrm{cllreg}' \\
\wedge\ (\mathrm{mode} = \mathrm{retry} \rightarrow \mathrm{breg} \neq \bot \wedge \mathrm{decglseqreg} \neq [] \wedge \mathrm{goal} \neq []) \\
\wedge\ (\mathrm{decglseqreg}' = [] \vee \mathrm{goal} = [] \rightarrow \mathrm{mode} = \mathrm{call}) \\
\wedge\ (\mathrm{stop}' \neq \mathrm{failure} \rightarrow \mathrm{mode} = \mathrm{mode}' \wedge \mathrm{stop} = \mathrm{stop}') \\
\wedge\ (\quad \mathrm{stop}' = \mathrm{failure} \wedge \mathrm{stop} \neq \mathrm{failure} \\
\quad \rightarrow \mathrm{stop} = \mathrm{run} \wedge \mathrm{mode} = \mathrm{retry} \wedge \mathrm{breg}' = \bot) \\
\wedge\ \langle \mathrm{F}\#(\mathrm{breg}, \mathrm{b}, \mathrm{cll}, \mathrm{db}_2; \mathrm{n}_0)\rangle\ \mathrm{F}[\mathrm{n}_0] = \mathrm{breg}' \\
\wedge\ \langle \mathrm{G}\#(\mathrm{decglseqreg}, \mathrm{b}, \mathrm{cll}, \mathrm{db}_2; \mathrm{decglseqreg}_0)\rangle \\
\qquad \mathrm{F}_\mathrm{d}(\mathrm{F}, \mathrm{decglseqreg}_0) = \mathrm{decglseqreg}' \\
\wedge\ \langle \mathrm{STACK}\#(\mathrm{breg}, \mathrm{b}; \mathrm{stack})\rangle \\
\qquad (\quad \mathrm{stack} \subseteq \mathrm{s} \wedge (\mathrm{mode} \neq \mathrm{retry} \rightarrow \mathrm{decglseqreg}\ \mathrm{cutptsin}\ \mathrm{stack}) \\
\qquad \wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad \rightarrow \quad \mathrm{decglseq}[\mathrm{n}]\ \mathrm{cutptsin}\ \mathrm{cdr}(\mathrm{stack\ from\ n}) \\
\qquad\qquad \wedge\ \mathrm{decglseq}[\mathrm{n}] \neq [] \wedge \mathrm{goal}[\mathrm{n}] \neq []) \\
\qquad \wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{stack} \wedge \mathrm{clause}'(\mathrm{cll}[\mathrm{n}], \mathrm{db}_2) \neq \mathrm{null} \\
\qquad\qquad \rightarrow \quad \mathrm{F}[\mathrm{n}] \in \mathrm{s}' \wedge \mathrm{F}[\mathrm{n}] \neq \bot \wedge \mathrm{decglseq}[\mathrm{n}] \neq [] \wedge \mathrm{goal} \neq [] \\
\qquad\qquad \wedge\ \langle \mathrm{F}\#(\mathrm{b}[\mathrm{n}], \mathrm{b}, \mathrm{cll}, \mathrm{db}_2; \mathrm{n}_0)\rangle\ \mathrm{F}[\mathrm{n}_0] = \mathrm{b}'[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad \wedge\ \langle \mathrm{G}\#(\mathrm{decglseq}[\mathrm{n}], \mathrm{b}, \mathrm{cll}, \mathrm{db}_2; \mathrm{decglseqreg}_0)\rangle \\
\qquad\qquad\qquad \mathrm{F}_\mathrm{d}(\mathrm{F}, \mathrm{decglseqreg}_0) = \mathrm{decglseq}'[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad \wedge\ \mathrm{cll}[\mathrm{n}] = \mathrm{cll}'[\mathrm{F}[\mathrm{n}]] \wedge \mathrm{sub}[\mathrm{n}] = \mathrm{sub}'[\mathrm{F}[\mathrm{n}]] \\
\qquad\qquad \wedge\ (\forall\ \mathrm{n}_1. \quad \mathrm{n}_1 \in \mathrm{stack} \wedge \mathrm{clause}'(\mathrm{cll}[\mathrm{n}_1], \mathrm{db}_2) \neq \mathrm{null} \\
\qquad\qquad\qquad\qquad \wedge\ \mathrm{n} \neq \mathrm{n}_1 \\
\qquad\qquad\qquad\qquad \rightarrow \mathrm{F}[\mathrm{n}] \neq \mathrm{F}[\mathrm{n}_1])))
\end{array}
$$

Nachträglich ließe sich die Invariante noch durch Zusammenfassung von deterministisch aufeinander folgenden 1:1-Diagrammen zu größeren Diagrammen vereinfachen. Dies ist für die Regelfolgen *call* (zweiter Fall, in dem kein Backtracking ausgelöst wird) *try*, *enter* (ergibt ein 3:3-Diagramm) und *retry*, *enter* (2:2-Diagramm) möglich. Dadurch verschwinden die Regeln *try* und *enter* von

den Eckpunkten der Diagramme, so daß die Invariante für $mode = try$, $mode' = try$, $mode = enter$ oder $mode' = enter$ nicht mehr gelten muß. Alle Konjunktionsglieder, die eine dieser Bedingungen als Voraussetzung haben, also die ersten 11 Zeilen der Invariante, können so aus der Invariante entfernt werden.

Kapitel 14

4/5: lineare Übersetzung der Backtrackingstruktur

14.1 Definition von ASM5

Die ersten drei Verfeinerungen können als Optimierung der ersten ASM angesehen werden, die die Repräsentation des Prolog-Programms nicht verändern. In der Verfeinerung von ASM4 nach ASM5 wird dagegen die Prädikatstruktur von Prolog compiliert. Dazu werden die ersten Instruktionen eingeführt, die sich schließlich auch in der WAM finden werden. Wir weichen gegenüber [BR95] in diesem Abschnitt zunächst insofern ab, als der Code von ASM5 zunächst *lineare* Ketten, nicht die komplizierteren *geschachtelten* Ketten enthält, die erst in ASM6 eingeführt werden (eine genauere Definition der beiden Begriffe folgt später). Der Grund liegt einfach darin, daß sich am Übergang von ASM4 zu ASM5 die typischen Probleme studieren lassen, die sich aus einer Compilerannahme ergeben, ohne daß gleichzeitig die Problematik von m:n-Diagrammen gelöst werden muß.

Die Grundidee der Verfeinerung besteht darin, die Kontrolle über die Ausführung von Prolog-Code aus der *mode*-Variable in den Code zu verlagern. Dazu wird die Rolle des *cllreg* geändert. Es zeigt nun nicht mehr in eine Liste von Klauseln, sondern wird ein Programmzähler, der in eine Liste von *Anweisungen* zeigt. Es wird deshalb auch zu *preg* umbenannt (von engl. *p*rogram counter). Analog wird *cll*[*n*] zu *p*[*n*].

Die bei *preg* vorliegende Anweisung wird nun von einer Funktion *code* geliefert, die die Funktion *clause* ersetzt. Abfragen der Variablen *mode* werden nun durch Abfragen über die vorliegende Anweisung bei $code(preg,db_5)$ ersetzt, wobei db_5 die Datenbasis von ASM5 ist. Die möglichen Anweisungen von ASM5 sind zum Teil weiterhin Klauseln (die Einzelklauseln werden erst in den Verfeinerungen 8/9 und 9/10 in feinere Anweisungen zerlegt). Zusätzlich werden nun die Kontrollinstruktionen *try_me_else*, *retry_me_else* und *trust_me* eingeführt, die die Ausführung der Regeln *try* und *retry* (then und else-Fall) anstoßen.

Um die Funktion der Kontrollinstruktionen zu verstehen, betrachten wir als Beispiel die folgenden Klauseln für ein Prolog-Prädikat `p`:

```
p(X)    :- body1.
p(f(X)) :- body2.
p(g(X)) :- body3.
p(g(X)) :- body4.
```
(14.1)

In der Verfeinerung von ASM4 zu ASM5 werden sie übersetzt zu

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: retry_me_else(L4)
    p(g(X)) :- body3.
L4: trust_me
    p(g(X)) :- body4.
```
(14.2)

Die Labels `L1` – `L4` dienen als symbolische Adressen. Bei einer Anfrage `?- p(X)` an das Prolog-Programm setzt die *call*-Regel (die jetzt aufgerufen wird, wenn sich *preg* an einer speziellen Adresse *start* befindet) *preg* jetzt auf die Adresse `L1` (die Adresse *failcode* als Ergebnis der *procdef*-Funktion zeigt jetzt an, daß keine unifizierbaren Klauseln vorhanden sind).

call rule
  if is_user_defined(act) $\wedge$ preg = start
  then ctreg := breg
    if code(procdef_5(act,db_5)) = failcode
    then backtrack
    else preg := procdef_5(act,db_5)

mit

backtrack $\equiv$
  if breg = $\bot$
  then stop := failure
  else preg := p[breg]

Das Ausführen der `try_me_else(L2)` Anweisung bei Adresse `L1` durch die *try_me* rule hat nun denselben Effekt, den die *try* rule in ASM4 hatte.

try_me rule
  if code(preg,db_5) = try_me_else(N)
  then let tmp = new(s)
    s := s $\cup$ {tmp}

breg := tmp
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
p[tmp] := N
preg := preg +1

Die im Choicepoint abgelegte Adresse für alternative Klauseln ist `L2`, und die Ausführung des Programms geht mit der folgenden Adresse weiter. Die dort stehende Klausel wird durch die *enter* rule ausgeführt, die dieselbe Wirkung wie in ASM4 hat. Da sie bei erfolgreicher Unifikation die *call* rule aktivieren muß, setzt sie *preg* := *start*.

enter rule
if is_user_defined(act) $\wedge$ code(preg,db_5) = clause
then let cla = rename(clause,vi)
let mgu = unify(act, hd(cla))
if mgu = nil
then backtrack
else decglseqreg := mgu $\hat{}_d$ [<bdy(cla),ctreg> | cont]
subreg := subreg $\circ$ unify
vi := vi +1
preg := start

Wenn durch Backtracking *preg* zu `L3` oder zu `L5` wird, werden die *retry_me* rule bzw. die *trust_me* rule ausgeführt, die dem then- bzw. else-Fall der *retry* rule von ASM4 entsprechen. Die Fallunterscheidung wird also nicht mehr zur Laufzeit getroffen, sondern in die Kompilation verlagert.

retry_me rule
if code(preg,db_5) = retry_me_else(N)
then decglseqreg := decglseq[breg]
subreg := sub[breg]
ctreg := b[breg]
p[breg] := N
preg := preg +1

trust_me rule
if code(preg,db_5) = trust_me
then decglseqreg := decglseq[breg]
ctreg := b[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

Ganz allgemein werden die zu einem Prädikat gehörenden Klauseln der Originalprogramms in ein Codefragment der Datenbasis von ASM5 compiliert,

das mit einem `try_me_else`-Anweisung beginnt, dann abwechseln Klauseln und `retry_me_else`-Anweisungen enthält. Die letzte Klausel wird durch ein `trust_me` eingeleitet. Ein derartiges Codefragment wird als *lineare Kette* bezeichnet (engl. *linear chain*). Die Anforderung, daß alle Codefragmente zu lineare Ketten übersetzt werden müssen, wird formal durch die Compilerannahme für die Verfeinerung 4/5 beschrieben:

$$\begin{array}{l} \quad \mathrm{db}_5 = \mathrm{compile}_{45}(\mathrm{db}_2) \\ \rightarrow [\mathrm{CLLS\#}(\mathrm{procdef}_2(\mathrm{act},\mathrm{db}_2),\mathrm{db}_2;\mathrm{col}_1)] \\ \quad\quad \langle \text{L-CHAIN\#}(\mathrm{procdef}_5(\mathrm{act},\mathrm{db}_5),\mathrm{db}_5;\mathrm{col}_2)\rangle \\ \quad\quad\quad\quad \mathrm{mapclause}'(\mathrm{col}_1,\mathrm{db}_2) = \mathrm{mapclause}'(\mathrm{col}_2,\mathrm{db}_5) \end{array} \tag{14.3}$$

Darin sind $procdef_2$ und db_2 die in den ASMs 2,3 und 4 verwendete *procdef*-Funktion und Datenbasis. $procdef_5$ ist die neue *procdef*-Funktion für ASM5 und db_5 ist das compilierte Prolog-Programm. Die Prozedur *L-CHAIN#* terminiert genau dann, wenn das Codefragment bei $procdef_5(act,db_5)$ eine lineare Kette ist. Sie liefert die in ihr enthaltenen Adressen von Klauseln. Genau wie bei *stackof* (siehe S. 95 in Abschnitt 11.2) genügt eine Definition einer prädikatenlogischen Funktion *l-chain* durch rekursive Gleichungen **nicht**. Durch die Terminierung müssen zyklische Ketten als Ergebnis der Kompilation ausgeschlossen werden.

Eine präzise Definition des *L-CHAIN#*-Programms ist in Anhang D.1 enthalten.

14.2 Äquivalenzbeweis 4/5

Analysiert man die Verfeinerung von ASM4 nach ASM5, so stellt man fest, daß neben der Verlagerung von *mode* in den Code noch eine Vorverlegung des Tests $clause(procdef_2(act,db_2)) = null$ von der *try rule* in die *call rule* stattfindet. Diese Modifikation läßt sich auch rein in ASM4 durchführen. Dazu werden die *try rule* und die *call rule* durch

call rule
    if stop = run ∧ mode = call
      ∧ is_user_defined(act)
    then if clause($\mathrm{procdef}_2$(act,db_2)) = null
        then backtrack
        else cllreg := $\mathrm{procdef}_2$(act,db_2)
            ctreg := breg
            if clause($\mathrm{procdef}_2$(act,db_2)+1, db_2) ≠ null
            then mode := try
            else mode := enter

try rule
    if stop = run ∧ mode = try

then mode := enter
let tmp = new(s)
s := s ∪ {tmp}
breg := tmp
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
cll[tmp] := cllreg +1

ersetzt. Bezeichnet man das Ergebnis als ASM4a, so enthält die Verfeinerung von ASM4a zu ASM5 nur noch 1:1-Diagramme.

Für die Verifikation der Verfeinerung von ASM4 nach ASM4a muß für den Fall, daß $mode = call$ ist und kein Backtracking stattfindet, ein 2:1- bzw. ein 2:2-Diagramm für die *call rule* und die *try rule* betrachtet werden, je nachdem ob $clause(procdef_2(act, db_2)) = null$ gilt oder nicht. Die in KIV durchgeführte Verifikation ist trivial, da als Zusammenhangsinvariante offensichtlich die Identität genügt. Wir gehen daher nicht weiter darauf ein.

Die Verifikation von 4a/5 war Gegenstand der Diplomarbeit von Wolfgang Ahrendt an der Universität Karlsruhe ([Ahr95]) und ist auch auch in [SA98] ausführlich dargestellt.

Etwa ein Monat Arbeit und 9 Iterationen waren notwendig, um die korrekte Zusammenhangsinvariante zu finden. Die Komplexität des Beweises ist in etwa so hoch wie die der Verifikation der Verfeinerung 1/2. Die wesentliche Problematik bei der Erstellung der Zusammenhangsinvariante ist es, die Compileranannahmen in gültige Beziehungen zwischen den den Choicepoints umzuformen. So muß z.B. für den Fall $mode = retry$ für jeden Choicepoint n gelten, daß die in $pc[n]$ beginnende Code-Kette der ASM5 mit einem *retry_me_else* oder *trust_me* beginnt, und dieselben Klauseln enthält, wie die bei *cll[n]* beginnende Klauselliste. Formal bedeutet dies, daß

$\langle$CLLS#(cll[n], db_2; col_1)$\rangle$
  $\langle$L-CHAIN-RETRY-ME#(p[n], db_5; col_2)$\rangle$
    mapcode(col_2, db_5) = mapclause$'$(col_1, db_2))

gelten muß. Typisch für echte Kompilationsschritte ist die Verwendung einer Unterprozedur der in der Compilerannahme definierten Prozedur *L-CHAIN#* (für die Definition siehe Anhang D.1). Man stellt fest, daß es, um die in der Zusammenhangsinvariante auftretenden Formeln möglichst einfach zu halten, am besten ist, die Prozeduren der Compilerannahmen entsprechend des Ablaufs der ASM zu strukturieren.

Das wohl wichtigste Ergebnis der Verifikation von 4a/5 war die Aufdeckung eines unbeabsichtigten Indeterminismus in ASM3 und ASM4. Das Problem wurde in dieser Verfeinerung gefunden, da sie vor den Verfeinerungen 2/3 und 3/4 verifiziert wurde. Um das Problem zu erkennen, betrachte man die *fail rule* aus ASM3 (S. 101), die unverändert in ASM4 übernommen wird. Die offensichtliche Intention bei der Definition der Regel war, daß anschließend die *retry rule* ausgeführt werden sollte. Aber die Verifikation der Verfeinerung 4a/5 zeigte, daß

die Ausführung der Regel ihre eigene Vorbedingung nicht falsifiziert. Das Regelsystem ist somit indeterministisch, und das Ausführen einer *fail*-Anweisung kann zu einer Endlosschleife führen.

Obwohl der Fehler leicht zu beheben ist — ein zusätzliches Konjunktionsglied $mode = call$ im Regeltest der *fail rule* reicht aus — ist dies unserer Meinung nach ein typischer Fehler, der auch durch sehr genaue Inspektion der ASM nicht zu finden ist (und natürlich *mußte* der Code für die Erstellung einer Zusammenhangsinvarianten sehr genau analysiert werden). Beim Lesen des Codes wird man den Indeterminismus immer intuitiv richtig auflösen. Die formale ASM-Definition und ebenso die letztendliche Implementierung des Prolog-Compilers kennt aber natürlich keine Intentionen, und wird den Konflikt bei der Sequentialisierung (so wie dies unsere Umsetzung in ein geschachteltes if-then-else auch tat) evtl. falsch auflösen.

Die formale Verifikation führt im Gegensatz zur informellen Analyse beim Versuch, die *fail rule* zu verifizieren, unvermeidlich zur Entdeckung des Fehlers. Die ermittelte Zusammenhangsinvariante für die Verifikation ist:

$$
\begin{array}{l}
\mathrm{INV}_{45} \equiv \\
\quad \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{vireg} = \mathrm{vireg}' \wedge \mathrm{subreg} = \mathrm{subreg}' \wedge \mathrm{breg} = \mathrm{breg}' \\
\wedge\ \mathrm{ctreg} = \mathrm{ctreg}' \wedge \mathrm{decglseqreg} = \mathrm{decglseqreg}' \wedge \mathrm{s} = \mathrm{s}' \wedge \mathrm{breg} \in \mathrm{s} \wedge \mathrm{ctreg} \in \mathrm{s} \\
\wedge\ \mathrm{decglseqreg\ ctpelem\ s} \\
\wedge\ (\mathrm{mode} = \mathrm{call} \rightarrow \mathrm{preg} = \mathrm{start}) \\
\wedge\ (\mathrm{mode} = \mathrm{retry} \rightarrow \mathrm{breg} \neq \bot \wedge \mathrm{preg} = \mathrm{p}[\mathrm{breg}']) \\
\wedge\ (\mathrm{mode} = \mathrm{enter} \rightarrow \mathrm{code}(\mathrm{preg}, \mathrm{db}_5) = \mathrm{mkcl}(\mathrm{the\_clau}(\mathrm{clause}'(\mathrm{cllreg}, \mathrm{db}_2)))) \\
\wedge\ (\quad \mathrm{mode} = \mathrm{try} \\
\quad \rightarrow \quad \mathrm{is\_user\_defined}(\mathrm{act}) \\
\qquad \wedge\ \langle \mathrm{CLLS\#}(\mathrm{cllreg}, \mathrm{db}_2; \mathrm{col}_1) \rangle \\
\qquad\quad \langle \text{L-CHAIN-TRY-ME\#}(\mathrm{preg}, \mathrm{db}_5; \mathrm{col}_2) \rangle \\
\qquad\qquad \mathrm{mapcode}(\mathrm{col}_2, \mathrm{db}_5) = \mathrm{mapclause}'(\mathrm{col}_1, \mathrm{db}_2)) \\
\wedge\ (\mathrm{decglseqreg} = [] \vee \mathrm{goal} = [] \vee \mathrm{act} = ! \vee \mathrm{act} = \mathrm{true} \rightarrow \mathrm{mode} = \mathrm{call}') \\
\wedge\ (\forall\ \mathrm{n}. \quad \mathrm{n} \in \mathrm{s} \wedge \mathrm{n} \neq \bot \\
\qquad \rightarrow \quad \mathrm{b}[\mathrm{n}] \in \mathrm{s} \wedge \mathrm{decglseq}[\mathrm{n}]\ \mathrm{ctpelem\ s} \wedge \mathrm{sub}[\mathrm{n}] = \mathrm{sub}'[\mathrm{n}] \\
\qquad\quad \wedge\ \mathrm{b}[\mathrm{n}] = \mathrm{b}'[\mathrm{n}] \wedge \mathrm{decglseq}[\mathrm{n}] = \mathrm{decglseq}'[\mathrm{n}] \wedge \mathrm{decglseq}[\mathrm{n}] \neq [] \\
\qquad\quad \wedge\ \mathrm{goal} \neq [] \wedge \mathrm{is\_user\_defined}(\mathrm{act}[\mathrm{n}]) \\
\qquad\quad \wedge\ \langle \mathrm{CLLS\#}(\mathrm{cll}[\mathrm{n}], \mathrm{db}_2; \mathrm{col}_1) \rangle \\
\qquad\qquad \langle \text{L-CHAIN-RETRY-ME\#}(\mathrm{p}[\mathrm{n}], \mathrm{db}_5; \mathrm{col}_2) \rangle \\
\qquad\qquad\quad \mathrm{mapcode}(\mathrm{col}_2, \mathrm{db}_5) = \mathrm{mapclause}'(\mathrm{col}_1, \mathrm{db}_2))
\end{array}
$$

Kapitel 15

5/7: strukturierte Übersetzung der Backtrackingstruktur

15.1 Definition von ASM6 und ASM7

In den ersten 5 ASM's wird das Problem, wie „relevante" Klauseln, deren Kopf mit den Aktivator unifizieren könnten, in die Verwendung einer unterspezifizierten *procdef*-Funktion codiert. In ASM7 wird diese Unterspezifikation durch die Angabe von Instruktionsfolgen zur Selektion der relevanten Klauseln aufgehoben.

Für die Konkretisierung der *procdef*-Funktion sind zunächst die beiden folgenden Extreme möglich:

- Eine einfache Implementierung, in der *procdef*(*act,db*) alle Klauseln liefert, deren Kopf dasselbe führenden Prädikatsymbol wie *act* hat. Diese Lösung ist ineffizient, da sie zu einer linearen Suche in den Klauseln führt, und eine große Zahl von (aufwendigen) fehlschlagenden Unifikationsversuchen verursacht: Man betrachte z.Bsp. eine Sammlung von Fakten $\mathtt{p(c_1)}, \ldots, \mathtt{p(c_n)}$ in einer Datenbasis.

- Eine aufwendige Lösung, die nur die mit dem Aktivator unifizierbaren Klauseln auswählt. Eine derartige Lösung ist mit Hilfe von sog. „discrimination nets" (siehe etwa [Gra96]) möglich. Sie codiert die gesamte Unifikation bereits in die Klauselselektion.

Die in der WAM verwendete Lösung ist ein Kompromiß zwischen den beiden Extremen. Sie verwendet die einfache *procdef*-Funktion in der *call*-Regel und zusätzlich *Switching*-Anweisungen, die, abhängig von führenden Funktionssymbolen in Argumenten von *act*, zu relevanten „Gruppen" von Klauseln springen.

Wenn etwa der Aktivator die Form `p(t1,f(t2))` hat, könnte eine Switching-Anweisung etwa zu einer Gruppe von Klauseln springen, deren zweites Argument entweder eine Variable oder `f` ist. Klauseln,deren zweites Argument ein anderes Funktionssymbol als `f` ist, würden nicht betrachtet.

Bevor Switching-Anweisungen eingeführt werden können, muß zunächst die Gruppierung von Klauseln möglich gemacht werden. Dies geschieht in ASM6 durch die Einführung von Anweisungsfolgen, die *geschachtelte* Ketten bilden. Geschachtelte Ketten sind genau wie lineare Ketten definiert, eine geschachtelte Kette darf aber an jeder Stelle, an der in einer linearen Kette eine Klausel steht, wieder eine geschachtelte Kette enthalten. Eine derartige innere Kette kann sinnvoll für eine Gruppe ähnlicher Klauseln gebildet werden, und von den Switching-Anweisungen in ASM7 als Ganzes übersprungen werden.

Betrachtet man das Beispielprogramm (14.1) aus Abschnitt 14.1, so können etwa die letzten beiden Klauseln gruppiert werden. Dies ergibt die bei `L4` beginnende Unterkette im folgenden Code:

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: trust_me                                        (15.1)
L4: try_me_else(L5)
    p(g(X)) :- body3.
L5: trust_me
    p(g(X)) :- body4.
```

Die Umstellung von linearen Ketten auf geschachtelte Ketten erfordert nur eine minimale Änderung der ASM-Codes. In der *retry_me_else* und *trust_me*-Anweisung kann *ctreg* nicht mehr mit $b[breg]$ geladen werden, da der Cutpoint des gerade aktiven Goals nicht mehr der Vater von *breg* sein muß. Vielmehr müssen *alle* Choicepoints, die für den aktuellen Aktivator aufgebaut wurden, ignoriert werden. Diese Zahl ist gerade die Schachtelungstiefe innerhalb der Codekette, die die ASM gerade abarbeitet. Für das *trust_me* bei `L5` also 2, die korrekte Zuweisung an *ctreg* in der entsprechenden Regel wäre also $ctreg := b[b[breg]]$. Das *trust_me* bei `L3` müßte hingegen weiterhin *ctreg* auf $b[breg]$ setzen. Um das Problem zu lösen, gibt es 2 Alternativen, die in [BR95] durch die nicht konkretisierte Anweisung *restore_cutpoint* wiedergegeben wurden: Entweder muß jede *retry_me_else* und *trust_me*-Anweisung ein zusätzliches Argument erhalten, das die Tiefe innerhalb der geschachtelten Kette wiedergibt, oder das korrekte *ctreg* muß innerhalb eines Choicepoints gespeichert werden. Wir haben die zweite Lösung gewählt, die nach [AK91] die Standardlösung darstellt. Sie erfordert eine zusätzliche Komponente *ct* für jeden Choicepoint. Die neuen *try_me_else*, *retry_me_else* und *trust_me*-Regeln lauten also:

try_me rule

if code(preg,db_7) = try_me_else(N)

then let tmp = new(s)
s := s ∪ {tmp}
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
p[tmp] := N
breg := tmp
ct[tmp] := ctreg
preg := preg +1

retry_me_else rule
if code(preg,db_7) = retry_me_else(N)
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := N
preg := preg +1

trust_me rule
if code(preg,db_7) = trust_me
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

Nachdem ASM6 die Gruppierung von Anweisungen möglich gemacht hat, können in ASM7 nun Switching-Anweisungen Ketten und Unterketten vorangestellt werden. Es gibt 3 Typen:

- `switch_on_term(i,Lv,Lc,Ll,Ls)` springt zu Adresse `Lv`, `Lc`, `Ll` bzw. `Ls`, abhängig davon, ob das *i*-te Argument *arg(act,i)* des Aktivators eine Variable, eine Konstante, eine Liste oder ein Funktionsterm ist.

- `switch_on_struct(i,N,T)`, nimmt an, daß bereits sichergestellt ist, daß *arg(act,i)* ein Funktionsterm ist. Verzweigt wird je nach führendem Funktionssymbol von *arg(act,i)*. Die Sprungadresse wird dabei durch Nachschlagen in einer Tabelle von Tripeln (*f,j,*`L`) bestimmt. Von dieser Tabelle wird angenommen, daß sie an der Adresse `T` im Speicher liegt, und `N` Tripel enthält. Falls der Unterterm *arg(act,i)* mit ein Term mit führendem Funktionssymbol *f* und *j* Untertermen ist, springt die Instruktion nach `L`. Die Auswahl des passenden Labels ist in die abstrakte Funktion *hashs* codiert. Im geschilderten Fall gilt also:

  hashs(arg(act,i),N,T,db_7) = `L`

- `switch_on_const(i,N,T)` nimmt analog zu `switch_on_struct` an, daß bereits sichergestellt wurde, daß *arg(act,i)* eine Konstante ist, und daß bei

Adresse T eine Tabelle mit N Paaren (c,L) gespeichert ist. Für die abstrakte Funktion *hashc* gilt dann analog

$$\text{hashs}(\text{arg}(\text{act,i}),\text{N,T},\text{db}_7) = \text{L}$$

falls $arg(act,i) = c$ ist

In unserem Beispiel könnten bei L4 etwa die folgenden Switching-Anweisungen addiert werden:

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: retry_me_else(L3)
    p(f(X)) :- body2.
L3: trust_me
L4: switch_on_term(L7,failcode,failcode,L6)                    (15.2)
L6: switch_on_struct(1,1,T)
L7: try_me_else(L5)
    p(g(X)) :- body3.
L5: trust_me
    p(g(X)) :- body4.
```

Adresse T enthielte dann eine einelementige Liste mit dem Element (g,1,L7). *failcode* ist eine spezielle Adresse, die zu Backtracking führt. Diese Adresse muß von *hashs* und *hashc* auch zurückgegeben gegeben werden, wenn das Funktions- bzw. Konstantensymbol nicht in der Tabelle gefunden wird. Dies führt zu folgenden ASM-Regeln für die Switching-Anweisungen.

switch_on_term rule
if code(preg, db_7) = switch_on_term(i, N_s, N_c, N_v, N_l)
then let x_i = arg(act,i)
if is_struct(x_i) **then** preg := N_s **else**
if is_const(x_i) **then** preg := N_c **else**
if is_var(x_i) **then** preg := N_v **else**
if is_list(x_i) **then** preg := N_l;
if preg = failcode **then backtrack**

switch_on_constant rule
if code(preg, db_7) = switch_on_constant(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashc(table, tabsize, constsym(x_i), db_7);
if preg = failcode **then backtrack**

switch_on_structure rule
if code(preg, db_7) = switch_on_structure(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashs(table, tabsize, funct(x_i), arity(x_i),db_7);
if preg = failcode **then backtrack**

Man beachte, daß zwar die *failcode*-Adresse in den in [BR95] gegebenen Beispielen vorkommt, daß aber der Aufruf von Backtracking in dem in Anhang 2 gegebenen Regeln fehlt. In den in [AK91] gegebenen Regeln für *switch_on_struct* und *switch_on_const* ist der Aufruf vorhanden, für die *switch_on_term*-Anweisung wird er aber auch nur durch die nirgends explizit gemachte Annahme, daß die *failcode*-Adresse die Adresse der Backtracking-Routine ist, realisiert.

Um die Verwendung von Klauseln in mehreren Ketten zu ermöglichen, werden in ASM6 außerdem die Instruktionen *try*(L), *retry*(L) und *trust*(L) eingeführt. Deren Effekt ist identisch zu dem der *try_me_else*(L), *retry_me_else*(L) und *trust_me*-Anweisungen, lediglich die Rolle von L und *preg* +1 als Adresse des anzulegenden Choicepoints bzw. Fortsetzungsadresse wird vertauscht.

try rule
if code(preg,db_7) = try(N)
then let tmp = new(s)
s := s ∪ {tmp}
b[tmp] := breg
decglseq[tmp] := decglseqreg
sub[tmp] := subreg
p[tmp] := preg +1
breg := tmp
ct[tmp] := ctreg
preg := N

retry rule
if code(preg,db_7) = retry(N)
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := preg +1
preg := N

trust rule
if code(preg,db_7) = trust(N)
then decglseqreg := decglseq[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := N

Im Beispiel von oben wäre eine Möglichkeit, die neuen Anweisungen sinnvoll einzusetzen, z.Bsp:

```
    switch_on_term(L2,failcode,failcode,L1)
L1: switch_on_struct(1,1,T)
L2: try_me_else(L4)
    p(X)    :- body1.
L3: retry_me_else(L6)
L4: p(f(X)) :- body2.
L5: retry_me_else(L8)
L6: p(g(X)) :- body3.
L7: trust_me
L8: p(g(X)) :- body4.
L9: try(L6)
    trust(L8)
```
(15.3)

wobei die Tabelle T die beiden Einträge (g,1,L6) und (f,1,L9) enthalten würde. Mit einem Aktivator `p(f(X))` würde ASM7 zunächst über die beiden Switching-Anweisungen ausführen, und dann mit den bei L9 stehenden *try* und *trust*-Anweisungen die beiden Klauseln bei L6 und L8 versuchen.

Zum Schluß sollte bemerkt werden, daß die oben gegebenen Codeschema nur zwei von vielen möglichen Schemata sind. Die Compilerannahme für die Verfeinerung von 5/7 läßt noch eine große Zahl von Alternativen zu, unter anderem auch die in [AK91] diskutierten Varianten "one-level switching" und "two-level switching".

Die Compilerannahme

$$\begin{aligned} &\mathrm{db}_6 = \mathrm{compile}_{56}(\mathrm{db}_5) \rightarrow [\text{L-CHAIN\#}(\mathrm{procdef}_5(\mathrm{act},\mathrm{db}_5),\mathrm{db}_5;\ \mathrm{col}_1)] \\ &\quad \langle \text{CHAIN\#}(\mathrm{procdef}_6(\mathrm{act},\mathrm{db}_6),\mathrm{db}_6;\ \mathrm{col}_2)\rangle \\ &\qquad \mathrm{mapcode}(\mathrm{col}_1,\ \mathrm{db}_5) = \mathrm{mapcode}(\mathrm{col}_2,\ \mathrm{db}_6) \end{aligned} \tag{15.4}$$

für 5/6 ist analog zu der für 4/5. Durch die Einführung von Switching-Anweisungen wird in ASM7 die Selektion innerhalb der Klauseln für ein Prädikat von der *procdef*-Funktion in die Switching-Funktion verlagert. Lediglich die Wahl des Startadresse für ein Prädikatsymbol muß noch von der *procdef*-Funktion übernommen werden. Diese kann nun als Tabelle (d.h. abstrakt durch eine dynamische Funktion) $procdef_7$, die vom Verfeinerungsschritt 6/7 erzeugt wird, übernommen werden. Somit gilt für $compile_{67}(db_6) := \langle procdef_7, db_7\rangle$:

$$\begin{aligned} &[\text{CHAIN\#}(\mathrm{procdef}_6(\mathrm{act},\mathrm{db}_6),\mathrm{db}_6;\ \mathrm{col}_1)] \\ &\quad \langle \text{S-CHAIN\#}(\mathrm{act},\ \mathrm{procdef}_7[\mathrm{id}(\mathrm{act})],\mathrm{db}_7;\ \mathrm{col}_2)\rangle \\ &\qquad \mathrm{mapcode}(\mathrm{col}_1,\ \mathrm{db}_6) = \mathrm{mapcode}(\mathrm{col}_2,\ \mathrm{db}_7) \end{aligned} \tag{15.5}$$

Der Selektor *id* selektiert das führende Prädikatsymbol incl. Stelligkeit eines Literals. Wir haben die Selektion des führenden Prädikatsymbols schon hier eingeführt, da uns dies die logische Realisierung des in [BR95], S. 27 genannten Verfeinerungsgedankens für die Klauselselektion erschien. In [BR95] findet

die Selektion des führenden Prädikatsymbols erst (ohne daß darauf im Text eingegangen wird) in der letzten ASM (der WAM) realisiert.

Die Programme *CHAIN#* und *S-CHAIN#* in der Compilerannahme charakterisieren geschachtelte Ketten (engl. nested chains) bzw. geschachtelte Ketten mit Switching. Die konkrete Definition von *S-CHAIN#* ist in Anhang D.2 gegeben. Sie ist deutlich komplexer als die in [BR95] gegebene, da sie u.a. präzisiert, daß Switching-Anweisungen nur *am Beginn* von Unterketten stehen dürfen.

15.2 Äquivalenzbeweis 5/7

Ein informelles Argument für die Äquivalenz von ASM5, ASM6 und ASM7 ist, daß sie alle dieselben Kandidatenklauseln versuchen. Etwas präziser formuliert, rufen alle 3 ASMs dieselbe Folge von *call-* und *enter*-Regeln mit denselben Aktivatoren *act* und jeweils denselben Kandidatenknoten (in der bei *preg* beginnenden (Rest-)Kette) auf. Leider ist dieses informelle Argument, wie es auch in [BR95] gegeben wird, für einen formalen Beweis bei weitem nicht ausreichend. Zwar läßt sich daraus eine Zerteilung des Gesamtablaufs in Teildiagramme gewinnen, deren Eckpunkte Zustände mit $preg = start$ und $is\_clause(code(preg, db))$ sind. Das Argument gibt aber weder eine Zuordnung von Zuständen, noch einen Hinweis, wie das Kommutieren der Teildiagramme nachgewiesen werden kann.

Um die Verifikation in den Griff zu bekommen, sind deshalb die folgenden drei Probleme zu lösen, die in den folgenden Abschnitten diskutiert werden:

- Es ist ein präziser Zusammenhang zwischen den Choicepoint-Stacks zu bestimmen.
- Hat man die richtige Zuordnung gefunden, so muß daraus eine Zuordnung der Cutpoints in den *decglseq*'s bestimmt werden. Daraus ergibt sich dann ein erster Ansatz für eine Zusammenhangsinvariante
- Schließlich bleibt als drittes Problem die Verifikation der Teildiagramme. Diese haben nun keine fest vorgegebene Größe mehr, wie dies bisher der Fall war. Vielmehr ist die Größe der Diagramme jetzt von der Anzahl der Instruktionen in einer Codekette abhängig. Wir diskutieren 2 Methoden, Diagramme, deren Größe von der Größe einer Datenstruktur abhängt, zu verifizieren.

Versucht man, das erste Problem zu lösen, so stellt man sofort fest, daß es einfacher ist, das Refinement 5/7 zu verifizieren als die Verfeinerung 6/7. Im ersten Fall muß der für einen Aktivator aufgebaute *eine* Choicepoint von ASM5 mit den entsprechenden Choicepoints in ASM7 verglichen werden, im zweiten Fall müssen 2 Mengen von Choicepoints verglichen werden. Deshalb haben wir zwar, quasi als „Vorstudie“ für die auftretenden Probleme, die Verfeinerung 5/6 verifiziert, dann aber direkt die Verfeinerung 5/7 in Angriff genommen. Wir diskutieren daher im folgenden die Lösung der drei obengenannten Probleme für die Verfeinerung 5/7, und gehen jeweils darauf ein, inwieweit die Verifikation von 5/6 einfacher ist.

Zuordnung der Choicepoint-Stacks Der Zusammenhang zwischen den beiden Stacks wurde zunächst sowohl bei 5/6 als auch bei 5/7 durch eine dynamische Funktion H : *node* $\rightarrow$ *nodelist* modelliert, die zu jedem ASM5-Choicepoint die passenden ASM6 bzw. ASM7-Choicepoints liefert. Die Funktion wird wie die Funktion F aus der Verifikation von 1/2 (siehe Abschnitt 11.2) in der Zusammenhangsinvariante existentiell quantifiziert. Aneinanderhängen der (jeweils nichtleeren) Listen $H[n]$ für die Stackknoten von ASM5 sollte den Stack von ASM6 bzw. ASM7 ergeben. Die bei $p[n]$ hängende (Rest-)Kette (mit *CHAIN-RET#* berechnet) sollte dieselben Klauseln enthalten, wie sie sich durch Zusammenhängen der bei $p'[n']$ mit $n' \in H[n]$ beginnenden Ketten ergibt (Programm *APP-CHAINS-RET#*). Außerdem sollten die Goals in *decglseq*[n], und *sub*[n] identisch zu *decglseq*[n'] und *sub*[n'] sein. Formalisiert bedeutet dies:

$$
\begin{array}{l}
\langle \text{STACK\#(breg,b;stack)} \rangle \\
\quad (\quad \langle \text{STACK\#(breg}', \text{b}'; \text{stack}') \rangle \ \text{stack}' = \text{H}_1(\text{H,stack}) \\
\quad\ \wedge \forall \text{ n. n} \in \text{stack} \\
\qquad\qquad \rightarrow \langle \text{L-CHAIN-RET\#(p[n], db}_5; \text{col}_1) \rangle \\
\\
\qquad \langle \text{S-APP-CHAINS-RET\#(decglseq}', \text{p,H[n]}, \text{db}_7; \text{col}_2) \rangle \\
\qquad\qquad\qquad \text{mapclause}(\text{col}_1, \text{db}_5) = \text{mapclause}(\text{col}_2, \text{db}_7)
\end{array}
$$

Es zeigt sich, daß dieser Zusammenhang zwar für die Verifikation von 5/6 genügt, für 5/7 aber nicht ausreicht. Der Grund ist, daß in ASM7 ein Choicepoint n möglich ist, dessen bei $p[n]$ gespeicherte Kette *keine* Klauseln enthält (der passende Aufruf von *S-CHAIN-RET#* liefert die leere Liste). Wir nennen einen solchen Choicepoint im folgenden *leeren Choicepoint*. Für einen leeren Choicepoint aus ASM7 kann es vorkommen, daß *in ASM5 kein passender Choicepoint vorhanden ist.*

Ein Beispiel für einen solchen leeren Choicepoint läßt sich etwa für das folgende Beispielprogramm geben, wobei die Tabelle T die beiden Einträge $(f, 1, \texttt{L5})$ und $(g, 1, \texttt{L7})$ enthält:

```
L1: try_me_else(L2)
    p(X)    :- body1.
L2: trust_me
    switch_on_term(L4,failcode,failcode,L3)
L3: switch_on_struct(1,2,T)                          (15.6)
L4: try_me_else(L6)
L5: p(f(X)) :- body2.
L6: trust_me
L7: p(g(X)) :- body3.
```

Für einen Aktivator `p(h(c))` ist ein solcher „leerer Choicepoint“ während der Abarbeitung der ersten Klausel vorhanden. Zu diesem Zeitpunkt zeigt $p[n]$ für den durch die *try_me_else*-Anweisung aufgebauten Choicepoint n nach `L2`. Die Ausführung der bei `L2` stehenden Anweisungen wird aber (im *switch_on_struct*)

zum Backtracking führen, ohne daß eine weitere Klausel versucht wird. Dennoch ist der Choicepoint vorhanden, während *body1* abgearbeitet wird. In ASM5 wird dagegen für den Aktivator `p(h(c))` gar kein Choicepoint aufgebaut, da nach Compilerannahme

$$\langle \text{L-CHAIN\#}(\text{procdef}_5(\text{act},\text{db}_5),\ \text{db}_5;\ \text{col}_1)\rangle\ \text{col}_1 = [\texttt{p(X) :- body1}]$$

der Code in ASM5 nur aus genau der ersten Klausel besteht. Somit ist das Bild des ASM5-Stacks unter H nicht mehr der ganze ASM7-Stack, sondern die Bilder $H[n]$ und $H[b[n]]$ zweier aufeinanderfolgender Knoten werden getrennt durch eine beliebig große (!) Zahl leerer Choicepoints.

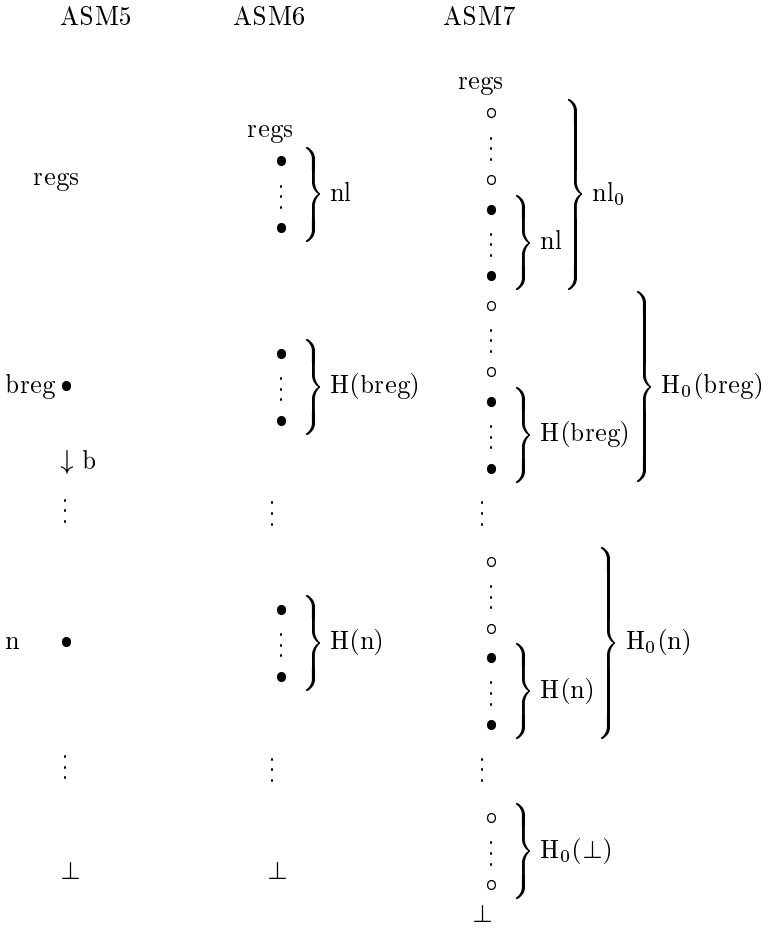


Abbildung 15.1

In Abbildung 15.1 ist die Situation graphisch wiedergegeben. Leere Choicepoints sind durch '∘' angedeutet. *regs* entspricht der aktuellen Belegung der

Register *decglseqreg,subreg* und *cllreg*. Die Abbildung zeigt, daß den Registerinhalten von ASM5 nicht nur die Registerinhalte von ASM6 bzw. ASM7 entsprechen, sondern zusätzlich die Inhalte von Choicepoints, die in einer Liste nl enthalten sind. Die Abbildung zeigt außerdem, daß die Problematik der leeren Choicepoints in ASM7 durch eine Funktion H_0 und eine zusätzliche Liste nl_0 formalisiert wurde. Zu beachten ist auch, daß am unteren Ende des Stacks eine zusätzliche Liste leerer Choicepoints von $H_0(\bot)$ vorhanden sein kann. Diese verursacht genau wie in der Verfeinerung 3/4 Probleme bei der Terminierung der ASMs.

Zuordnung der Cutpoints Für die Verifikation von 5/6 wird ein Cutpoint *ctpt* der ASM5 einfach auf $car(H[ctpt])$, den obersten korrespondierenden Cutpoint von ASM6, abgebildet. Eine Funktion $H_d(H,decglseq[n])$ bildet alle Cutpoints aus $decglseq[n]$ entsprechend ab.

Die dazu analoge Annahme, daß *ctpt* auf $car(H_0[ctpt])$ abgebildet werden muß, bestand auch bei den ersten Verifikationsversuchen von 5/7. Eine genaue Analyse der Fehlschläge dieser ersten Versuche zeigte aber, daß der *ctpt* entsprechende Cutpoint *irgendwo* innerhalb der leeren Choicepoints zwischen $H[ctpt]$ und $H[b[ctpt]]$ liegen kann, oder das erste Element von $H[b[ctpt]]$ sein kann. Dabei gibt es auch noch die Ausnahme, in der $b[ctpt] = \bot$ ist. Dann liegt der entsprechende Cutpoint in $H_0[\bot]$ oder ist selbst $\bot$. Die formale Definition der Ähnlichkeit zwischen *decglseq*'s aus ASM5 und ASM7 lautet somit ($cdr([])$ ist hier als [] zu definieren):

$$
\begin{array}{l}
\mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [],\ []),\\
\neg\ \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [\langle \mathrm{goal,ctpt}\rangle,\mathrm{dgl}],[]),\\
\neg\ \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [],[\langle \mathrm{goal}',\mathrm{ctpt}'\rangle,\ \mathrm{dgl}']),\\
\quad \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ [\langle \mathrm{goal,ctpt}\rangle,\ \mathrm{dgl}],[\langle \mathrm{goal}',\mathrm{ctpt}'\rangle,\ \mathrm{dgl}'])\\
\leftrightarrow \quad \mathrm{eqh}(\mathrm{H}_0,\ \mathrm{H},\ \mathrm{dgl},\ \mathrm{dgl}') \wedge \mathrm{goal} = \mathrm{goal}'\\
\quad \wedge\ (\mathrm{ctpt} = \bot \supset \mathrm{ctpt}' \in \mathrm{H}_0[\bot] \vee \mathrm{ctpt}' = \bot;\\
\qquad\qquad\qquad \mathrm{ctpt}' \in \mathrm{H}_0[\bot] \wedge \mathrm{ctpt}' \notin \mathrm{cdr}(\mathrm{H}[\mathrm{ctpt}])
\end{array}
$$

Diagramme datenstrukturabhängiger Größe Die in den Verfeinerungen 5/6 und 5/7 auftretenden kommutierenden Diagramme sind nicht, wie in den bisher betrachteten Fällen, Diagramme vom Typ $m{:}n$ mit konstantem m, n (etwa $m = 1$, $n = 2$, ...). Vielmehr wird n durch die Anzahl der Instruktionen bestimmt, die bis zur nächsten Klausel abzuarbeiten sind. Die Endlichkeit von n wird zwar implizit durch die Terminierung des *CHAIN#* bzw. des *S-CHAIN#*-Programms aus der Compilerannahme garantiert, für ein formales (induktives) Argument für die ist aber eine explizit abnehmende Größe n erforderlich. Für 5/6 ist dies einfach, da die Zahl n der Instruktionen innerhalb einer Kette direkt zur Zahl der Klauseln in der Kette korespondiert. Für ASM7 ist dies, wiederum verursacht durch die Existenz (beliebig langer) leerer Ketten, nicht so. Deshalb

muß für ASM7 die im Anhang D.3 definierte Prozedur *S-COUNT#* eingeführt werden, die explizit die Zahl der verbleibenden Instruktionen in einer Restkette zählt. Die Terminierung von *S-COUNT#* ist zwar intuitiv klar, da sie derselben Rekusionsstruktur wie *S-CHAIN#* folgt, für einen formalen Nachweis wird aber das neue, in Abschnitt 3 dargestellte Beweisprinzip der *Induktion über die Schachtelungstiefe von Prozeduren* benötigt. Dieses gestattet den einfachen Nachweis der Terminierung von *S-COUNT#* (sowie aller im Anhang D.3 genannten Hilfsprozeduren).

Für den eigentlichen Nachweis der Kommutierung der Diagramme bieten sich dann 2 Alternativen an, die wir im folgenden diskutieren: Die rekursive Zerlegung der Diagramme, und der Nachweis von Hilfsbehauptungen für die Einzel-ASMs:

rekursive Zerlegung der Diagramme Bei dieser Technik, die bei 5/6 angewandt wurde, werden die *m*:*n*-Diagramme mit datenstrukturabhängigem *n* selbst als Refinement aufgefaßt, und mit denselben Prinzipien wie das Originaldiagramm bearbeitet, also mit Hilfe des Modularisierungstheorems in kleinere Teildiagramme zerlegt. Diese Vorgehensweise scheint hier sehr natürlich, da sich eine Zusammenhangsinvariante für zwei Zwischenzustände während der Abarbeitung eines Diagramms aus dem Fall, daß beide ASMs direkt vor einer Klausel stehen, durch Verallgemeinerung ergibt: Für 5/6 tritt an die Stelle des Falls $is\_clause(code(preg,db_5)) \wedge is\_clause(code(preg',db_6))$ aus der Zusammenhangsinvariante INV_{56} des Gesamtsystems die schwächere Forderung, daß die gerade abzuarbeitetende Instruktionsfolgen von ASM5 und ASM6 beide zur selben Klausel führen müssen. Die so entstehenden schwächere Invariante $WINV_{56}$ für Zwischenzustände gilt unabhängig vom konkreten Zwischenzustand von ASM5 oder ASM6. Sie zerlegt also die in Abb. 15.2 gezeigten Diagramme in 1:0 und 0:1-Teildiagramme.

Paare von Zuständen, die bezüglich $WINV_{56}$ korrespondieren, sind durch gestrichelte Linien verbunden. *call1* und *call2* bezeichnen wie schon früher den ersten bzw. zweiten Fall der *call rule*. Der Zusatz „(a)“ bezeichnet den Fall beim backtracking, in dem $breg = \perp$ ist, in dem die ASM also die Berechnung mit *failure* beendet. Der Zusatz „(A)“ bzw. „(B)“ teilt den erfolgreichen Aufruf der *call rule* auf in den Fall, in dem nur eine Klausel zu bearbeiten ist, und den Fall, in dem mehrere Klauseln versucht werden müssen (in dem die folgende Anweisung also ein *try_me_else* bzw. ein *try* sein muß). *ret** bezeichnet eine beliebige Anzahl aufeinanderfolgender *retry-*, *retry_me-*, *trust-* oder *trust_me-*Anweisungen, und *tr** eine beliebige Zahl von *try-* oder *try_me-*Anweisungen. Das Ergebnis der rekursiven Anwendung der Theoreme sind die in Abb. 15.3 gezeigten Diagramme.

Gegenüber einer Zerlegung des Gesamtbeweises gleich in kleinere Unterdiagramme hat die Vorgehensweise den Vorteil einer etwas größeren Modularität und etwas kleinerer Invarianten. Diese wird erkauft durch die Notwendigkeit, zwei Zusammenhangsinvarianten INV_{56} *und* $WINV_{56}$ bestimmen zu müssen. Dieses Problem ist hier aber nicht sehr gravierend, da offenbar die Beziehung

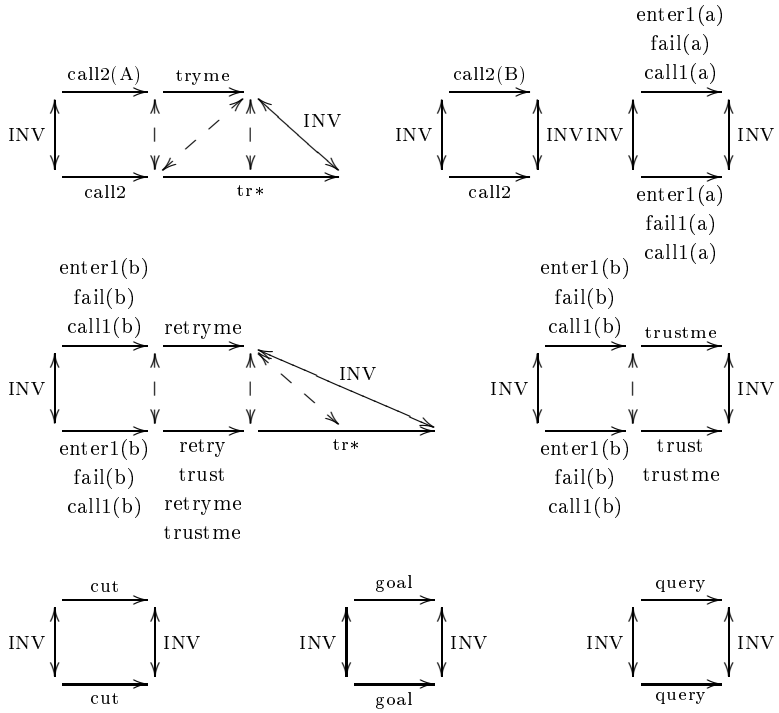


Abbildung 15.2 : Kommutierende Diagramme für die Verfeinerung 5/6

$$
\begin{array}{l}
\quad \text{preg} \neq \text{start} \\
\rightarrow (\quad \text{INV}_{56} \\
\qquad \leftrightarrow \quad \text{WINV}_{56} \wedge \text{is\_clause}(\text{code}(\text{preg},\text{db}_5)) \\
\qquad\quad \wedge \text{is\_clause}(\text{code}(\text{preg}',\text{db}_6)))
\end{array}
\tag{15.7}
$$

gelten muß. Daher genügt es, sich $WINV_{56}$ zu überlegen, und die Äquivalenz (15.7) zur Konstruktion von INV_{56} für den Fall $preg \neq start$ zu benutzen (der Fall $preg = start$ ist relativ einfach). Die Verfeinerung konnte in 2 Wochen und 8 Iterationen erfolgreich geführt werden. Die Verallgemeinerung von *INV* zu *WINV* stellte hier kein wesentliches Problem dar. Es ergaben sich die folgenden Zusammenhangsinvarianten:

$$
\begin{array}{l}
\text{HINV}_{56} \equiv \\
\exists\ \text{h}.\quad \bot \in \text{s} \wedge \bot \in \text{s}' \wedge \text{h}[\bot] = [\bot] \wedge \text{ctreg} \in \text{s} \wedge \text{ctreg}' \in \text{s}' \\
\qquad \wedge\ \text{stop} = \text{stop}' \wedge \text{vireg} = \text{vireg}' \wedge (\text{h}[\text{breg}] \neq [] \rightarrow \text{car}(\text{h}[\text{breg}]) = \text{breg}') \\
\qquad \wedge (\quad \neg (\quad \text{is\_retry\_me}(\text{code}(\text{preg}', \text{db}_6)) \vee \text{is\_retry}(\text{code}(\text{preg}', \text{db}_6)) \\
\qquad\qquad\quad \vee \text{is\_trust\_me}(\text{code}(\text{preg}', \text{db}_6)) \vee \text{is\_trust}(\text{code}(\text{preg}', \text{db}_6))) \\
\qquad\quad \rightarrow \text{ctreg}' = \text{car}(\text{h}[\text{ctreg}]))
\end{array}
$$

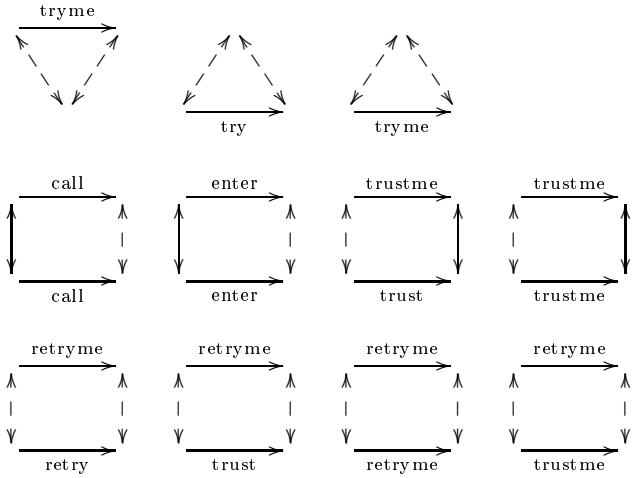


Abbildung 15.3 : Unterdiagramme für die Verfeinerung 5/6

$\wedge$ subreg = subreg$'$ $\wedge$ hdg(h, decglseqreg) = decglseqreg$'$
$\wedge$ (preg = start $\leftrightarrow$ preg$'$ = start)
$\wedge$ (decglseqreg = [] $\vee$ goal = [] $\vee$ act = ! $\vee$ act = true $\rightarrow$ preg = start)
$\wedge$ (is_clause(code(preg, db$_5$)) $\wedge$ ctreg $\neq$ breg
$\rightarrow$ ctreg = b[breg] $\wedge$ breg $\neq \perp$
$\wedge$ decglseq[breg] = decglseqreg $\wedge$ sub[breg] = subreg)
$\wedge$ (preg $\neq$ start
$\rightarrow$ is_clause(code(preg, db$_5$)) $\wedge$ is_clause(code(preg$'$, db$_6$))
$\wedge$ code(preg, db$_5$) = code(preg$'$, db$_6$)) $\wedge$ ctreg$'$ = car(h[ctreg])
$\wedge$ $\langle$STACK#(breg, b; stack)$\rangle$
(stack $\subseteq$ s $\wedge$ decglseqreg cutptsin stack
$\wedge$ $\langle$STACK#(breg$'$, b$'$; stack$'$)$\rangle$
(stack$'$ = hl(h, stack) $\wedge$ stack$'$ $\subseteq$ s$'$)
$\wedge$ ($\forall$ n. n $\in$ stack
$\rightarrow$ decglseq[n] $\neq$ [] $\wedge$ goal[n] $\neq$ []
$\wedge$ is_user_defined(act[n]) $\wedge$ h[n] $\neq$ []
$\wedge$ decglseq[n] cutptsin cdr(stack from n)
$\wedge$ ($\forall$ n$_0$. n$_0$ $\in$ h[n]
$\rightarrow$ sub[n] = sub$'$[n$_0$]
$\wedge$ hdg(h, decglseq[n]) = decglseq$'$[n$_0$]
$\wedge$ ct[n$_0$] = car(h[b[n]]))
$\wedge$ $\langle$L-CHAIN-RETRY-ME#(p[n], db$_5$; col)$\rangle$
$\langle$APP-CHAINS-RET#(p$'$, h[n], db$_6$; col$_2$)$\rangle$
mapcode(col, db$_5$) = mapcode(col$_2$, db$_6$)))
$\wedge$ STACKINV$_{56}$(true)

$$
\begin{array}{l}
\mathrm{WINV}_{56} \equiv \\
\exists\ \mathrm{h}.\quad \bot \in \mathrm{s} \wedge \bot \in \mathrm{s}' \wedge \mathrm{h}[\bot] = \bot +_{sl} [] \wedge \mathrm{ctreg} \in \mathrm{s} \wedge \mathrm{ctreg}' \in \mathrm{s}' \\
\quad \wedge\ \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop} = \mathrm{stop}' \wedge \mathrm{vireg} = \mathrm{vireg}' \\
\quad \wedge\ (\mathrm{h}[\mathrm{breg}] \neq [] \rightarrow \mathrm{car}(\mathrm{h}[\mathrm{breg}]) = \mathrm{breg}') \\
\quad \wedge\ (\quad \neg\ (\quad \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\qquad\qquad \vee\ \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6))) \\
\qquad \rightarrow\ \mathrm{ctreg}' = \mathrm{car}(\mathrm{h}[\mathrm{ctreg}])) \\
\quad \wedge\ \mathrm{subreg} = \mathrm{subreg}' \wedge \mathrm{hdg}(\mathrm{h}, \mathrm{decglseqreg}) = \mathrm{decglseqreg}' \\
\quad \wedge\ \mathrm{preg} \neq \mathrm{start} \wedge \mathrm{preg}' \neq \mathrm{start} \\
\quad \wedge\ \mathrm{decglseqreg} \neq [] \wedge \mathrm{goal} \neq [] \wedge \mathrm{act} \neq ! \wedge \mathrm{act} \neq \mathrm{true} \\
\quad \wedge\ (\mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \rightarrow \mathrm{is\_user\_defined}(\mathrm{act}) \wedge \mathrm{ctreg} = \mathrm{breg}) \\
\quad \wedge\ (\quad \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \wedge \mathrm{ctreg} \neq \mathrm{breg} \\
\qquad \rightarrow\quad \mathrm{ctreg} = \mathrm{b}[\mathrm{breg}] \wedge \mathrm{breg} \neq \bot \\
\qquad\quad \wedge\ \mathrm{decglseq}[\mathrm{breg}] = \mathrm{decglseqreg} \wedge \mathrm{sub}[\mathrm{breg}] = \mathrm{subreg}) \\
\quad \wedge\ (\mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \rightarrow \mathrm{ctreg}' = \mathrm{car}(\mathrm{h}[\mathrm{ctreg}])) \\
\quad \wedge\ (\quad \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\qquad \rightarrow\ \mathrm{is\_user\_defined}(\mathrm{act}')) \\
\quad \wedge\ (\quad \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\qquad\quad \vee\ \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\qquad \rightarrow\quad \mathrm{breg}' \neq \bot \wedge \mathrm{preg}' = \mathrm{p}'[\mathrm{breg}'] \wedge \mathrm{ctreg}' = \mathrm{car}(\mathrm{h}[\mathrm{b}[\mathrm{breg}]]) \\
\qquad\quad \wedge\ (\mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)))) \\
\quad \wedge\ (\quad \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \\
\qquad \rightarrow\quad (\quad \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\qquad\quad \vee\ \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6))) \\
\qquad\quad \wedge\ \mathrm{breg} \neq \bot \wedge \mathrm{preg} = \mathrm{p}[\mathrm{breg}] \\
\qquad\quad \wedge\ \langle \text{L-CHAIN-RETRY-ME\#}(\mathrm{preg}, \mathrm{db}_5; \mathrm{col})\rangle \\
\qquad\qquad \langle \text{APP-CHAINS-RET\#}(\mathrm{p}', \mathrm{h}[\mathrm{breg}], \mathrm{db}_6; \mathrm{col}_2)\rangle \\
\qquad\qquad\quad \mathrm{mapcode}(\mathrm{col}, \mathrm{db}_5) = \mathrm{mapcode}(\mathrm{col}_2, \mathrm{db}_6)) \\
\quad \wedge\ (\quad \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \\
\qquad \rightarrow\quad (\mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6))) \\
\qquad\quad \wedge\ \langle \text{L-CHAIN-TRY-ME\#}(\mathrm{preg}, \mathrm{db}_5; \mathrm{col})\rangle \\
\qquad\qquad \langle \text{CHAIN-REC\#}(\mathrm{preg}', \mathrm{db}_6; \mathrm{col}_1)\rangle \\
\qquad\qquad\quad \mathrm{mapcode}(\mathrm{col}, \mathrm{db}_5) = \mathrm{mapcode}(\mathrm{col}_1, \mathrm{db}_6)) \\
\quad \wedge\ (\mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \wedge \neg\ \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\qquad \rightarrow\quad (\mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6))) \\
\qquad\quad \wedge\ \mathrm{breg} \neq \bot \wedge \mathrm{decglseqreg} = \mathrm{decglseq}[\mathrm{breg}] \\
\qquad\quad \wedge\ \mathrm{subreg} = \mathrm{sub}[\mathrm{breg}] \wedge \mathrm{ctreg} = \mathrm{b}[\mathrm{breg}] \\
\qquad\quad \wedge\ \langle \text{L-CHAIN-RETRY-ME\#}(\mathrm{p}[\mathrm{breg}], \mathrm{db}_5; \mathrm{col})\rangle \\
\qquad\qquad \langle \text{CHAIN-REC\#}(\mathrm{preg}', \mathrm{db}_6; \mathrm{col}_1)\rangle \\
\qquad\qquad\quad \langle \text{APP-CHAINS-RET\#}(\mathrm{p}', \mathrm{h}[\mathrm{breg}], \mathrm{db}_6; \mathrm{col}_2)\rangle \\
\qquad\qquad\qquad\quad \mathrm{the\_cl}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) +_{cli} \mathrm{mapcode}(\mathrm{col}, \mathrm{db}_5) \\
\qquad\qquad\qquad\quad = \mathrm{mapcode}(\mathrm{col}_1 \odot_{col} \mathrm{col}_2, \mathrm{db}_6)) \\
\quad \wedge\ (\quad \mathrm{is\_try}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\qquad \rightarrow\ \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5))) \\
\quad \wedge\ (\mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \rightarrow \mathrm{code}(\mathrm{preg}, \mathrm{db}_5) = \mathrm{code}(\mathrm{preg}', \mathrm{db}_6)) \\
\quad \wedge\ (\quad \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5)) \vee \mathrm{is\_try\_me}(\mathrm{code}(\mathrm{preg}, \mathrm{db}_5))
\end{array}
$$

$\vee$ is_retry_me(code(preg, db_5)) $\vee$ is_trust_me(code(preg, db_5)))
$\wedge$ STACKINV$_{56}$($\neg$ is_retry_me(code(preg, db_5)))

STACKINV$_{56}$ $\equiv$
$\langle$STACK#(breg, b; stack)$\rangle$
(stack $\subseteq$ s $\wedge$ (cond $\rightarrow$ decglseqreg cutptsin stack)
$\wedge$ $\langle$STACK#(breg$'$, b$'$; stack$'$)$\rangle$
(stack$'$ = hl(h, stack) $\wedge$ stack$'$ $\subseteq$ s$'$)
$\wedge$ ($\forall$ n. n $\in$ stack
$\rightarrow$ decglseq[n] $\neq$ [] $\wedge$ goal[n] $\neq$ [] $\wedge$ is_user_defined(act[n])
$\wedge$ decglseq[n] cutptsin cdr(stack from n)
$\wedge$ ($\forall$ n_0. n_0 $\in$ h[n]
$\rightarrow$ sub[n] = sub$'$[n_0] $\wedge$ ct[n_0] = car(h[b[n]])
$\wedge$ hdg(h, decglseq[n]) = decglseq$'$[n_0]
$\wedge$ (n $\neq$ breg
$\vee$ $\neg$ is_try_me(code(preg$'$, db_6))
$\wedge$ $\neg$ is_try(code(preg$'$, db_6))
$\vee$ is_try_me(code(preg, db_5))
$\rightarrow$ h[n] $\neq$ []
$\wedge$ $\langle$CHAIN-RETRY-ME-FL#(p[n], db_5; col)$\rangle$
$\langle$APP-CHAINS-RET#(p$'$, h[n], db_6; col_2)$\rangle$
mapcode(col, db_5) = mapcode(col_2, db_6))))

Hilfsbehauptungen über die ASMs Wenn man den Äquivalenzbeweis für 5/6 analysiert, so stellt man fest, daß in den Beweisen der 0:1-Diagramme eine große Zahl von Eigenschaften der ASM5 als invariant gezeigt werden müssen, die implizit über den Zusammenhang zu ASM5 in die Invariante codiert sind. Eine Alternative dazu ist es, direkt Hilfsbehauptungen über ASM6 allein zu formulieren, die die Abarbeitung von Ketten der ASM6 betreffen. Wir haben für ASM7 zunächst mit der Technik der rekursiven Zerlegung der Diagramme eine Lösung erarbeitet. Es zeigte sich, daß die Verallgemeinerung der Invarianten INV_{57} zu $WINV_{57}$ im Gegensatz zu 5/6 gewaltige Schwierigkeiten verursacht: $WINV_{57}$ hat letztlich den 4-fachen Umfang von $WINV_{56}$. Zu der Bestimmung der korrekten Version und damit zur Verifikation von 5/7 wurden 2 Monate und 20 Iterationen benötigt. Deshalb haben wir anschließend eine Alternative versucht. Wie die Statistik am Ende des Abschnitts zeigt, führte diese Technik zu sehr viel kleineren Beweisen. Für komplizierte Verfeinerungen ist diese Technik deshalb vorzuziehen, auch wenn sie die Problematik, eine korrekte Zusammenhangsinvariante zu finden, um das zusätzliche Problem verschärft, Hilfsbehauptungen zu finden, die nicht nur korrekt sind, sondern auch in den Gesamtbeweis „passen“.

Als Hilfsbehauptungen über ASM7 wurde zunächst formuliert, daß die Abarbeitung einer beliebigen (Rest-)Kette zu einem der folgenden Ergebnisse führt:

- Falls die Kette leer und $breg = \perp$ ist, wird der Lauf von ASM beendet mit $stop = failure$

- Falls die Kette leer und *breg* $\neq \perp$ ist, erreicht die ASM einen Zustand, in dem die Kette komplett abgearbeitet und soeben mit Backtracking verlassen wurde, d.h. *decglseqreg, subreg, ctreg, vireg* sowie der Stack sind dann unverändert, *preg* zeigt auf das oberste Stack-Element *p*[*breg*].

- Falls die Kette nichtleer ist, wird ein Zustand erreicht, in dem die erste Klausel der Kette erreicht ist, d.h. *decglseqreg, subreg, ctreg, vireg* sind ebenfalls unverändert, *preg* zeigt auf die erste Klausel der Kette. Auf den Stack wurden eine Anzahl von Choicepoints gelegt, die alle *decglseqreg, subreg* und *ctreg* enthalten, und deren Ketten aneinandergehängt genau die Klauseln der Ausgangskette bis auf die erste enthalten.

Formalisiert ergibt dies das Lemma *chain7*:

$$
\begin{array}{l}
\quad \mathrm{decglseq}' = \mathrm{decglseq}'_0 \wedge \mathrm{sub}' = \mathrm{sub}'_0 \wedge \mathrm{ct} = \mathrm{ct}_0 \wedge \mathrm{p}' = \mathrm{p}'_0 \\
\wedge\ \mathrm{b}' = \mathrm{b}'_0 \wedge \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \perp \in \mathrm{s}'_0 \\
\wedge\ \mathrm{decglseqreg}'_0 \neq [] \wedge \mathrm{goal}'_0 \neq [] \wedge \mathrm{is\_user\_defined}(\mathrm{act}'_0) \\
\wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}')\rangle\ \mathrm{stack}' = \mathrm{stack} \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\wedge\ (\quad \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \\
\quad \vee\ \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \\
\quad \supset\ \mathrm{stack} \neq [] \wedge \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack})] \wedge \mathrm{decglseqreg}' \neq [] \\
\quad \wedge\ \mathrm{goal}' \neq [] \wedge \mathrm{decglseqreg}'_0 = \mathrm{decglseq}'[\mathrm{car}(\mathrm{stack})] \\
\quad \wedge\ \mathrm{subreg}'_0 = \mathrm{sub}'[\mathrm{car}(\mathrm{stack})] \wedge \mathrm{ctreg}'_0 = \mathrm{ct}[\mathrm{car}(\mathrm{stack})] \\
\quad \wedge\ \mathrm{stack}_0 = \mathrm{cdr}(\mathrm{stack})\ ; \\
\quad\ \ \mathrm{subreg}'_0 = \mathrm{subreg}' \wedge \mathrm{decglseqreg}'_0 = \mathrm{decglseqreg}' \\
\quad \wedge\ \mathrm{ctreg}'_0 = \mathrm{ctreg}' \wedge \mathrm{stack}_0 = \mathrm{stack}) \\
\wedge\ \langle \mathrm{S\text{-}ANY\text{-}CHAIN\#}(\mathrm{act}'_0, \mathrm{preg}', \mathrm{db}_7; \mathrm{col})\rangle \\
\quad \mathrm{col} = \mathrm{col}_0 \\
\rightarrow \exists\ \mathrm{kappa}. \\
\quad \langle \textbf{loop} \\
\quad\ \ \textbf{if}\ \mathrm{stop}' = \mathrm{run}\ \textbf{then} \\
\quad\ \ \mathrm{RULE}'(\mathrm{mkco3res}(\mathrm{db}_7, \mathrm{procdeftab}); \mathrm{s}', \mathrm{vireg}', \mathrm{stop}', \mathrm{breg}', \\
\quad\ \ \qquad \mathrm{ctreg}', \mathrm{sub}', \mathrm{subreg}', \mathrm{decglseq}', \mathrm{decglseqreg}', \mathrm{p}', \\
\quad\ \ \qquad \mathrm{preg}', \mathrm{b}', \mathrm{ct}) \\
\quad\ \ \textbf{times}\ \mathrm{kappa}\rangle \\
\quad\ \ (\quad \mathrm{col}_0 = [] \\
\quad\ \ \supset\ \mathrm{stack}_0 = [] \supset \mathrm{stop}' = \mathrm{failure} \wedge \mathrm{breg}' = \perp\ ; \\
\quad\ \ \qquad \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack}_0)] \wedge \mathrm{decglseqreg}' = \mathrm{decglseqreg}'_0 \\
\quad\ \ \wedge\ \mathrm{subreg}' = \mathrm{subreg}'_0 \wedge \mathrm{ctreg}' = \mathrm{ctreg}'_0 \\
\quad\ \ \wedge\ \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \mathrm{stop}' = \mathrm{run} \\
\quad\ \ \wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack})\rangle \\
\quad\ \ \qquad (\quad \mathrm{stack} = \mathrm{stack}_0 \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\quad\ \ \qquad\ \ \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\quad\ \ \qquad\ \ \qquad\ \ \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\quad\ \ \qquad\ \ \qquad\ \ \qquad\ \ \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}] \\
\quad\ \ \qquad\ \ \qquad\ \ \qquad\ \ \wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}]))\ ; \\
\quad\ \ \qquad \mathrm{decglseqreg}' = \mathrm{decglseqreg}'_0 \wedge \mathrm{subreg}' = \mathrm{subreg}'_0
\end{array}
$$

$$
\begin{array}{l}
\wedge\ \mathrm{ctreg}' = \mathrm{ctreg}'_0 \wedge \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \\
\wedge\ \mathrm{stop}' = \mathrm{run} \wedge \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \wedge \mathrm{preg}' = \mathrm{car}(\mathrm{col}_0) \\
\wedge\ (\exists\ \mathrm{nl}.\quad \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack})\rangle \\
\qquad\qquad (\mathrm{stack} = \mathrm{append}(\mathrm{nl}, \mathrm{stack}_0) \wedge \mathrm{stack} \subseteq \mathrm{s}') \\
\qquad \wedge\ \langle \text{S-APP-CHAINS-RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}, \mathrm{db}_7; \\
\qquad\qquad \mathrm{col})\rangle\ \mathrm{col} = \mathrm{cdr}(\mathrm{col}_0) \\
\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl} \\
\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseqreg}'_0 \\
\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{subreg}'_0 \wedge \mathrm{ct}[\mathrm{n}] = \mathrm{ctreg}'_0) \\
\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack}_0 \\
\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}] \\
\qquad\qquad\quad \wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}])))
\end{array}
$$

Der Beweis erfolgt durch Induktion über die Anzahl der Instruktionen der Kette. Auf diesem Lemma aufbauend kann gezeigt werden, daß wenn soeben Backtracking stattfindet und ein Stack von Choicepoints vorliegt, der mit einer Reihe leerer Choicepoints beginnt, ein Zustand erreicht wird, in dem die leeren Choicepoints alle abgebaut wurden. Formalisiert ergibt dies das Lemma *emptychains7*:

$$
\begin{array}{l}
\quad \mathrm{decglseq}' = \mathrm{decglseq}'_0 \wedge \mathrm{sub}' = \mathrm{sub}'_0 \wedge \mathrm{ct} = \mathrm{ct}_0 \wedge \mathrm{p}' = \mathrm{p}'_0 \\
\wedge\ \mathrm{b}' = \mathrm{b}'_0 \wedge \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \bot \in \mathrm{s}'_0 \\
\wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \\
\wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}')\rangle\ \mathrm{stack}' = \mathrm{stack} \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\wedge\ (\quad \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \\
\quad \vee\ \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7))) \\
\wedge\ \mathrm{stack} = \mathrm{append}(\mathrm{nl},\mathrm{stack}_0) \wedge \mathrm{stack} \neq [] \wedge \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack})] \\
\wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl} \\
\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}'[\mathrm{n}] \neq [] \\
\qquad\quad \wedge\ \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{n}])) \\
\wedge\ \langle \text{S-APP-CHAINS-RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}, \mathrm{db}_7; \mathrm{col})\rangle\ \mathrm{col} = [] \\
\rightarrow\ \exists\ \mathrm{kappa}.\ \langle \textbf{loop} \\
\qquad\qquad \textbf{if}\ \mathrm{stop}' = \mathrm{run}\ \textbf{then} \\
\qquad\qquad\ \mathrm{RULE}'(\mathrm{mkco3res}(\mathrm{db}_7, \mathrm{procdeftab}); \mathrm{s}', \mathrm{vireg}', \mathrm{stop}', \mathrm{breg}', \\
\qquad\qquad\qquad \mathrm{ctreg}', \mathrm{sub}', \mathrm{subreg}', \mathrm{decglseq}', \mathrm{decglseqreg}', \mathrm{p}', \\
\qquad\qquad\qquad \mathrm{preg}', \mathrm{b}', \mathrm{ct}) \\
\qquad\qquad \textbf{times}\ \mathrm{kappa}\rangle \\
\qquad\qquad (\mathrm{stack}_0 = [] \supset \mathrm{stop}' = \mathrm{failure} \wedge \mathrm{breg}' = \bot\ ; \\
\qquad\qquad\quad \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack}_0)] \\
\qquad\qquad\quad \wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \\
\qquad\qquad\quad \wedge\ \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \mathrm{stop}' = \mathrm{run} \\
\qquad\qquad\quad \wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack})\rangle \\
\qquad\qquad\qquad (\quad \mathrm{stack} = \mathrm{stack}_0 \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\qquad\qquad\qquad\ \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\ \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}]
\end{array}
$$

$\wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}])))$

Schließlich benötigt man die Kombination von *chain7* und *emptychains7*, also ein Lemma *nextclause7*, das besagt, daß Backtracking in einen Stack von Choicepoints zum ersten nichtleeren Choicepoint führt, und die dort gefundene Kette zu einer Klausel und neuen Choicepoints reduziert wird. Formal:

$$
\begin{array}{l}
\quad \mathrm{decglseq}' = \mathrm{decglseq}'_0 \wedge \mathrm{sub}' = \mathrm{sub}'_0 \wedge \mathrm{ct} = \mathrm{ct}_0 \wedge \mathrm{p}' = \mathrm{p}'_0 \\
\wedge\ \mathrm{b}' = \mathrm{b}'_0 \wedge \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{stop}' = \mathrm{run} \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \bot \in \mathrm{s}'_0 \\
\wedge\ \mathrm{decglseqreg}' \neq [] \wedge \mathrm{goal}' \neq [] \\
\wedge\ \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack}')\rangle\ \mathrm{stack}' = \mathrm{stack} \wedge \mathrm{stack} \subseteq \mathrm{s}' \\
\wedge\ (\quad \mathrm{is\_retry}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_retry\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \\
\quad \vee\ \mathrm{is\_trust}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \vee \mathrm{is\_trust\_me}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7))) \\
\wedge\ \mathrm{stack} = \mathrm{append}(\mathrm{nl},\mathrm{stack}_0) \wedge \mathrm{stack}_0 \neq [] \wedge \mathrm{preg}' = \mathrm{p}'[\mathrm{car}(\mathrm{stack})] \\
\wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl} \\
\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] \neq [] \wedge \mathrm{goal}'[\mathrm{n}] \neq [] \wedge \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{n}])) \\
\wedge\ \langle \text{S-APP-CHAINS-RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}, \mathrm{db}_7; \mathrm{col})\rangle\ \mathrm{col} = [] \\
\wedge\ \langle \text{S-CHAIN-RET\#}(\mathrm{act}'[\mathrm{car}(\mathrm{stack}_0)], \mathrm{p}'[\mathrm{car}(\mathrm{stack}_0)], \mathrm{db}_7; \mathrm{col})\rangle \\
\qquad \mathrm{col} = \mathrm{col}_0 \\
\wedge\ \mathrm{col}_0 \neq [] \wedge \mathrm{decglseq}'[\mathrm{car}(\mathrm{stack}_0)] \neq [] \\
\wedge\ \mathrm{goal}'[\mathrm{car}(\mathrm{stack}_0)] \neq [] \\
\wedge\ \mathrm{is\_user\_defined}(\mathrm{act}'[\mathrm{scar}(\mathrm{stack}_0)]) \\
\rightarrow \exists\ \mathrm{kappa}.\ \langle \textbf{loop} \\
\qquad \textbf{if}\ \mathrm{stop}' = \mathrm{run}\ \textbf{then} \\
\qquad \mathrm{RULE}'(\mathrm{mkco3res}(\mathrm{db}_7, \mathrm{procdeftab}); \mathrm{s}', \mathrm{vireg}', \mathrm{stop}', \mathrm{breg}', \\
\qquad\quad \mathrm{ctreg}', \mathrm{sub}', \mathrm{subreg}', \mathrm{decglseq}', \mathrm{decglseqreg}', \mathrm{p}', \\
\qquad\quad \mathrm{preg}', \mathrm{b}', \mathrm{ct}) \\
\qquad \textbf{times}\ \mathrm{kappa}\rangle \\
\qquad (\quad \mathrm{decglseqreg}' = \mathrm{decglseq}'_0[\mathrm{car}(\mathrm{stack}_0)] \\
\qquad \wedge\ \mathrm{subreg}' = \mathrm{sub}'_0[\mathrm{car}(\mathrm{stack}_0)] \wedge \mathrm{ctreg}' = \mathrm{ct}_0[\mathrm{car}(\mathrm{stack}_0)] \\
\qquad \wedge\ \mathrm{vireg}' = \mathrm{vireg}'_0 \wedge \mathrm{s}'_0 \subseteq \mathrm{s}' \wedge \mathrm{stop}' = \mathrm{run} \\
\qquad \wedge\ \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg}', \mathrm{db}_7)) \wedge \mathrm{preg}' = \mathrm{car}(\mathrm{col}_0) \\
\qquad \wedge\ (\exists\ \mathrm{nl}_1.\quad \langle \mathrm{STACK\#}(\mathrm{breg}', \mathrm{b}'; \mathrm{stack})\rangle \\
\qquad\qquad\qquad (\quad \mathrm{stack} = \mathrm{append}(\mathrm{nl}_1, \mathrm{cdr}(\mathrm{stack}_0)) \\
\qquad\qquad\qquad\ \wedge\ \mathrm{stack} \subseteq \mathrm{s}') \\
\qquad\qquad \wedge\ \langle \text{S-APP-CHAINS-RET\#}(\mathrm{decglseq}', \mathrm{p}', \mathrm{nl}_1, \mathrm{db}_7; \\
\qquad\qquad\qquad \mathrm{col})\rangle\ \mathrm{col} = \mathrm{cdr}(\mathrm{col}_0) \\
\qquad\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{nl}_1 \\
\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseqreg}' \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{subreg}' \wedge \mathrm{ct}[\mathrm{n}] = \mathrm{ctreg}') \\
\qquad\qquad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{cdr}(\mathrm{stack}_0) \\
\qquad\qquad\qquad\qquad \rightarrow\quad \mathrm{decglseq}'[\mathrm{n}] = \mathrm{decglseq}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{sub}'[\mathrm{n}] = \mathrm{sub}'_0[\mathrm{n}] \wedge \mathrm{b}'[\mathrm{n}] = \mathrm{b}'_0[\mathrm{n}] \\
\qquad\qquad\qquad\qquad\quad \wedge\ \mathrm{ct}[\mathrm{n}] = \mathrm{ct}_0[\mathrm{n}] \wedge \mathrm{p}'[\mathrm{n}] = \mathrm{p}'_0[\mathrm{n}])))
\end{array}
$$

Mit diesen Hilfsbehauptungen lassen sich die Diagramme von 5/7 dann wie in Abb. 15.4 gezeigt zerlegen.

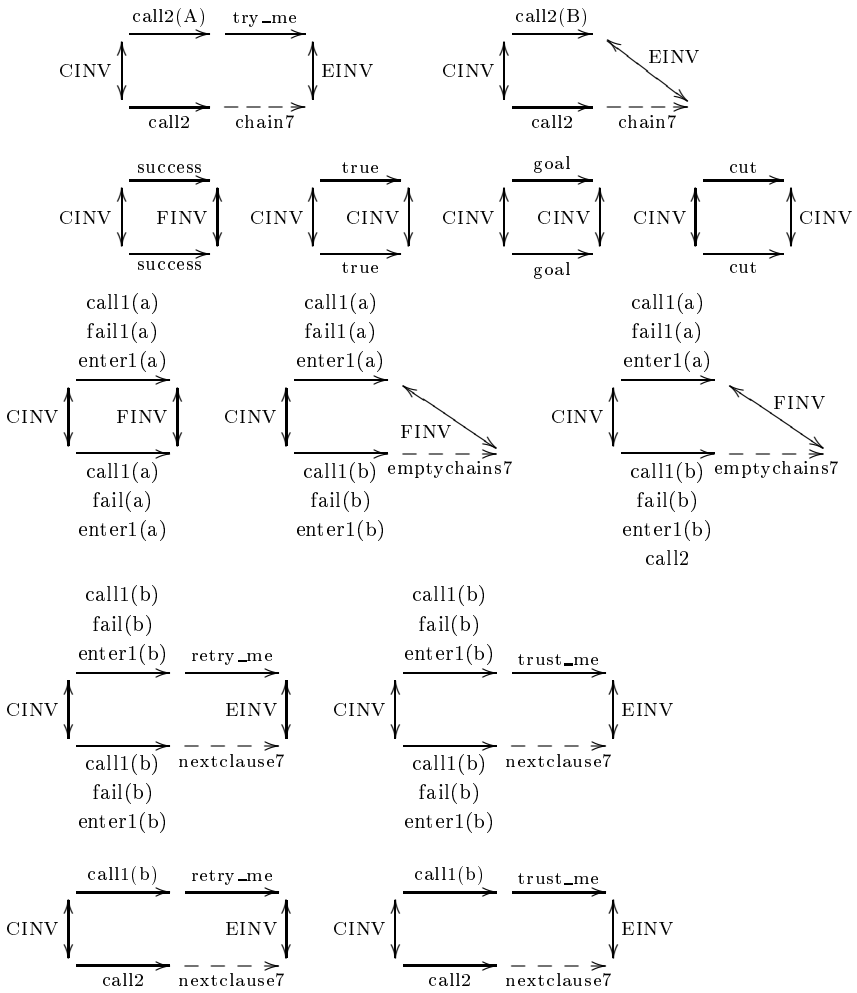


Abbildung 15.4 : Kommutierende Diagramme für die Verfeinerung 5/7

CINV ist der Fall der in der Zusammenhangsinvariante, in dem $preg = start$ gilt, *EINV* der Fall, in dem die nächste Instruktion eine Klausel ist. Im Fall *FINV* ist der Ablauf beider ASMs beendet. Es zeigt sich, daß die kompliziertesten Beweise diejenigen sind, in denen Backtracking aufgerufen wird (die 7 Diagramme der unteren Hälfte von Abbildung 15.4). Die Beweise für die verschiedenen Varianten der ersten 5 Diagramme, können wie angedeutet überlagert werden. Als Vorbedingung genügt dabei die Invariante, wenn die erste auszuführende Regel von ASM5 und ASM7 durch einen Backtrack-Aufruf ersetzt wird. Die letzten beiden Diagramme lassen sich auf die jeweils darüberliegenden reduzieren, in dem sofort das *chain7*-Lemma angewandt wird (um die leere Kette der ASM7 zu beseitigen).

Als Gesamtergebnis ergeben sich für die rekursive Zerlegung in Teildiagramme ein Aufwand von 17009 Beweisschritten und 1521 Interaktionen. Die Verifikation unter Verwendung von Hilfsbehauptungen konnte in 1 Woche durchgeführt werden. Sie erforderte nur noch 7473 Beweisschritte und 1351 Interaktionen.

Kapitel 16

7/8: Umgebungen und Stack-Sharing

16.1 Definition von ASM8

Nachdem mit ASM7 die Kompilation der Prädikatstruktur abgeschlossen ist, wird in der Verfeinerung von ASM7 nach ASM8 nun die Kompilation der einzelnen Klauseln vorbereitet. Dazu ist es zunächst erforderlich, die Datenstruktur der *decglseq*'s so umzucodieren, daß die darin enthaltenen Goals durch Zeiger in den Code des Rumpfs von Klauseln ersetzt werden können. Um dies möglich zu machen, wird in ASM8 die bei Unifikation berechnete Substitution nicht mehr direkt auf den Klauselrumpf der betrachteten Klausel angewandt, sondern erst angewandt, wenn ein neuer Aktivator *act* berechnet wird. Dadurch enthalten die einzelnen *goal*'s aus den *decglseq*'s nun alle Endstücke von Klauselrümpfen des Prolog-Programms. Zwar enthalten die *goal*'s noch umbenannte Variablen, weshalb sie noch nicht direkt durch Zeiger in den Quellcode ersetzt werden können (dies wird im nächsten Verfeinerung 8/9 geschehen), dennoch haben durch den Verzicht auf sofortige Anwendung der berechneten Substitution in der *enter rule* das alte und das neu berechnete *decglseqreg* nun einen Großteil an Informationen gemeinsam, der durch Umstrukturierung nun nur noch einmal gespeichert werden muß (Sharing).

Dies geschieht wie folgt: Statt ein *decglseqreg* mit $[\langle goal_1, ctpt_1\rangle, \langle goal_2, ctpt_2\rangle, \langle goal_3, ctpt_3\rangle, \ldots]$ zu belegen, wird $goal_1$ in einem neuen Register *goalreg* direkt zugreifbar gemacht. Für die restliche Information wird eine *Umgebung* (engl. environment) angelegt. Formal ist eine Umgebung analog zu einem Choicepoint. Sie besteht aus ein Element einer dynamischen Sorte *envnode*, die in einem Register *ereg* gespeichert wird (analog zu *breg*). An dem Knoten hängen dann mit Hilfe einer dynamischen Funktion *cutpt* der aktuelle Cutpoint: $cutpt[ereg] = ctpt_1$. Das zweite Goal von *decglseqreg* wird mit einer dyn. Funktion *cg* (von engl. continuation goal) adressiert: $cg[ereg] = goal_2$. Die Verkettung zur restlichen Information (also zu $ctpt_2$, $goal_3$, etc.) wird über eine Verkettung der Umgebungen

mit einer Funktion *ce : envnode* $\rightarrow$ *envnode* (engl. continuation environment) realisiert. Es ist also $cutpt[ce[ereg]] = ctpt_2$, $cg[ce[ereg]] = goal_3$ usw..

Die Umcodierung der Informationen des *decglseqreg* macht natürlich eine analoge Umcodierung der Informationen für einen Choicepoint n notwendig. An die Stelle von $decglseqreg[n]$ treten in ASM9 zwei Funktionen $goal[n]$ und $e[n]$, die analog zu *goalreg* und *ereg* sind.

Die Repräsentationsänderung für die *decglseq*'s wirft die Frage auf, ob Umgebungen auf einem separaten (Umgebungs-)Stack realisiert werden müssen. Dies ist nicht so, es ist möglich Umgebungen und Choicepoints auf demselben Stack abzulegen, und dabei eine echte Stack-Disziplin einzuführen, die nicht mehr benötigte Stack-Knoten (destruktiv) überschreibt. Die Sorte *envnode* ist also identisch zur Sorte *node*.

In [BR95] wird dazu zunächst in ASM8 ein gemeinsamer, aber nicht destruktiv modifizierter Stack eingeführt, und ein *Hiding Lemma* (S. 33) formuliert, das es dann erlaubt, den Stack in ASM9 in einen destruktiv modifizierten umzuwandeln. Dies erschien uns für die Verifikation ungünstig, weil in der Zwischenstufe eine dynamische Funktionen *tos* eingeführt werden muß, die ein Maximum zweier Knoten abhängig von der dynamischen Stack-Verkettungsfunktion − liefern muß (S. 32). Eine derartige Definition ist zwar möglich, aber aufwendig. Sie würde nur in ASM8 benötigt, und ist unnötig, wenn man in ASM8 sofort zur Stack-Repräsentation von ASM9 übergeht, was wir getan haben. Das *Hiding Lemma* wird somit zum Bestandteil der Verifikation der Verfeinerung 7/8. Die Einführung eines destruktiv modifizierten Stacks, der sowohl Choicepoints als auch Environments enthält, wird in unserer Lösung komplett in der Verfeinerung 7/8 abgehandelt, und nicht auf 2 Verfeinerungen aufgeteilt.

Um einen destruktiv modifizierten Stack zu erlauben, führen wir auf den Knoten des Stacks, die nun mehr und mehr die Rolle von Adressen erhalten, eine totale Ordnung $\ll$, sowie Funktionen $+1$ und -1 zum Inkrementieren und Dekrementieren der Adresse des obersten Stackelements ein. Allokation und Deallokation von Knoten erfolgt nun nicht mehr mit der Funktion *new* relativ zu der bisher verwendeten Menge s der allokierten Knoten (i.e. zur Trägermenge der dynamischen Sorte *node* der ASM), sondern durch Inkrementieren des Zeigers auf das oberste Stackelement. Der Stackzeiger wird nun auch dekrementiert, wenn Umgebungen oder Choicepoints nicht mehr zugreifbar werden, so daß die Inhalte alter Stackknoten (nach erneutem Inkrementieren) überschrieben werden. Allokierte Knoten, die außerhalb des gegenwärtigen Stacks liegen, sind deshalb ab ASM8 nicht mehr vorhanden. Die Register *breg* und *ereg* enthalten nun zwei Zeiger in den Stack. Den Kern der Aussage des *Hiding-Lemmas* bildet nun die Aussage, daß die zu einem Choicepoint n gehörenden Umgebungen $e[n]$, $ce[e[n]]$, ... immer unterhalb von n liegen, sofern neue Knoten (sowohl Choicepoint- als auch Umgebungsknoten) immer bei $max(breg,ereg) + 1$, also oberhalb des Maximums von *breg* und *ereg* allokiert werden. Dasselbe gilt dann auch umgekehrt für die Umgebungsknoten n': Die Choicepoints $cutpt[n']$, $cutpt[b[n']]$ liegen ebenfalls immer unterhalb von n'. Auch diese werden durch das Überschreiben des Knotens $max(breg,ereg)+1$ dann nicht betroffen. Für die ASM8 ergeben sich so folgende Regeln:

backtrack ≡
    if breg = ⊥ **then** stop := failure
                else preg := p[breg]

call rule
    let act = subreg ^$_t$ car(goalreg)
    if preg = start ∧ is_user_defined(act)
    then if procdef$_7$(act,db$_7$) = failcode
        then backtrack
        else preg := procdef$_7$(act,db$_7$)
            ctreg := breg

cut rule
    let act = subreg ^$_t$ car(goalreg)
    if act = !
    then breg := cutpt[ereg]
        goalreg := rest(goalreg)

enter rule
    if is_clause(code(preg, db$_7$))
    then let cla = rename(clause(code(preg, db$_7$)), vireg)
        let act = subreg ^$_t$ car(goalreg)
        let mgu = unify(act, hd(cla))
        if mgu = nil
        then backtrack
        else let tmp = max(ereg,breg)+1
            ce[tmp] := ereg
            ereg := tmp
            cg[tmp] := rest(goalreg)
            cutpt[tmp] := ctreg
            goalreg := bdy(cla)
            subreg := subreg ∘ mgu
            vireg := vireg +1
            preg := start

fail rule
    let act = subreg ^$_t$ car(goalreg)
    if act = fail
    then backtrack

goal success rule
    if goalreg = [] ∧ ¬ ereg = ⊥
    then goalreg := cg[ereg]
        ereg := ce[ereg]

query success rule
    if goalreg = [] ∧ ereg = ⊥
    then stop := success

retry rule
if code(preg,db_7) = retry(N)
then ereg := e[breg]
goalreg[breg] := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := preg +1
preg := N

retry_me_else rule
if code(preg,db_7) = retry_me_else(N)
then ereg := e[breg]
goalreg := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := N
preg := preg +1

switch_on_constant rule
let act = subreg $\hat{}_t$ car(goalreg)
if code(preg, db_7) = switch_on_constant(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashc(table, tabsize, constsym(x_i), db_7);
if preg = failcode **then backtrack**

switch_on_structure rule
let act = subreg $\hat{}_t$ car(goalreg)
if code(preg, db_7) = switch_on_structure(i, tabsize, table)
then let x_i = arg(act,i)
preg := hashs(table, tabsize, funct(x_i), arity(x_i), db_7);
if preg = failcode **then backtrack**

switch_on_term rule
let act = subreg $\hat{}_t$ car(goalreg)
if code(preg, db_7) = switch_on_term(i, N_s, N_c, N_v, N_l)
then let x_i = arg(act,i)
if is_struct(x_i) **then** preg := N_s **else**
if is_const(x_i) **then** preg := N_c **else**
if is_var(x_i) **then** preg := N_v **else**
if is_list(x_i) **then** preg := N_l;
if preg = failcode **then backtrack**

true rule
let act = subreg $\hat{}_t$ car(goalreg)
if act = true
then goalreg := rest(goalreg)

trust rule
if code(preg,db_7) = trust(N)
then ereg := e[breg]
goalreg := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := N

trust_me rule
if code(preg,db_7) = trust_me
then ereg := e[breg]
goalreg := goal[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

try rule
if code(preg,db_7) = try(N)
then let tmp = max(ereg,breg) +1
b[tmp] := breg
e[tmp] := ereg
goal[tmp] := goalreg
sub[tmp] := subreg
p[tmp] := preg +1
breg := tmp
ct[tmp] := ctreg
preg := N

try_me rule
if code(preg,db_7) = try_me_else(N)
then let tmp = max(ereg,breg)+1
b[tmp] := breg
e[tmp] := ereg
goal[tmp] := goalreg
sub[tmp] := subreg
p[tmp] := N
breg := tmp
ct[tmp] := ctreg
preg := preg +1

16.2 Äquivalenzbeweis 7/8

Die Verifikation von 7/8 stellt 3 wesentliche Probleme: Erstens muß der Zusammenhang der zwischen den *decglseq*'s und den Komponenten von ASM8

präzisiert werden. Hier ergab sich eine Modifikation der *query success*-Regel, und der 1:1-Zuordnung von Regeln, die im folgenden Abschnitt diskutiert wird. Zum zweiten muß in der Zusammenhangsinvariante die Korrektheit des Stack-Sharings explizit gemacht werden. Schließlich führt die Verschiebung der Substitution zu einer zusätzlichen Compilerannahme für 7/8. Die 3 Probleme werden in den folgenden Abschnitten diskutiert.

Zusammenhang der Umgebungen zu den *decglseq*'s Für die Verifikation von 7/8 war es zunächst notwendig, die Initialisierung der Umgebungen, den Zusammenhang der *decglseq*'s aus ASM7 mit den Komponenten von ASM8 sowie das Abbruchkriterium der ASM8 zu präzisieren. Diese hängen eng zusammen, da die Wahl der initialen Umgebung sowohl die Zusammenhangsinvariante als auch das Terminierungskriterium in der *query success*-Regel beeinflußt. Die im vorigen Abschnitt gezeigten ASM-Regeln enthalten bereits die hierfür gegenüber [BR95] notwendigen Präzisierungen.

Für die Initialisierung haben wir *ereg* auf $\perp$ gesetzt. Sowohl die Funktion *ce* als auch die Funktion *cutpt* müssen für $\perp$ auf $\perp$ zeigen. Die Initialisierung von *cg* ist beliebig, *goalreg* wird mit der initialen Anfrage belegt. Aus der Initialisierung ergibt sich die folgende Berechnungsvorschrift für *decglseqreg* und *decglseq*[*n*] aus ASM7 aus den Komponenten von ASM8:

$\langle$STACK#(ereg,ce;estack)$\rangle$
  decglseqreg = subreg $\hat{}_d$ [$\langle$goalreg, cutpt[ereg]$\rangle$ |
                                  decglseqof(cutpt, cg, ce, estack)]

$\langle$STACK#(e[n],ce;estack′)$\rangle$ decglseq[F[n]]
  = sub[F[st]] $\hat{}_d$ F_d(F,[$\langle$goal[n],cutpt[e[n]]$\rangle$ |
                              decglseqof(cutpt, cg, ce, estack′)])

Dabei wird die Zuordnung der Choicepoints von ASM8 zu Choicepoints von ASM7 wie in den früheren Verfeinerungen 1/2, 2/3, etc. durch eine dynamische Funktion F hergestellt. *estack* bzw. *estack′* sind der bei *ereg* bzw. *e*[*n*] beginnende, mit *ce* verkettete Stack von Umgebungen. Die Knoten dieser Stacks werden mit demselben Programm *STACK#* (zur Definition siehe Abschnitt 11.2) wie bei den Choicepoints berechnet. Die Funktion *decglseqof* sammelt die Information an den entsprechenden Knoten auf:

decglseqof(cutpt,cg,ce,[]) = []

decglseqof(cutpt,cg,ce,[n | estack])
  = [$\langle$cg[n],cutpt[ce[n]]$\rangle$ | decglseqof(cutpt,cg,ce,estack)]

Bis hierhin stimmen die Definitionen wohl mit den in [BR95] gegebenen überein. Lediglich die Initialisierung von *ereg* mit $\perp$ fehlt, der Zusammenhang der Register wurde präzisiert, und die Definition der Funkion G (S. 32f), die wegen möglicher Zyklen als Programm zu realisieren wäre, wurde in den Aufruf

von *STACK#* und *decglseqof* aufgeteilt. Abgewichen sind wir hingegen beim Terminierungskriterium: In [BR95] wird das Kriterium für *query success* (in unsere Notation übersetzt) als

$$\text{goalreg} = \bot \wedge \langle \text{STACK\#(ereg,ce;estack)} \rangle \ \forall\ \text{n} \in \text{estack.}\ \text{goal[n]} = [\,]$$

gegeben. Nun ist es zwar korrekt, die Berechnung zu beenden, wenn keine Goals mehr zu bearbeiten sind. Der Abbruchtest ist aber sehr aufwendig, da alle *goal*[*n*] untersucht werden müssen. Die Optimierung beseitigt außerdem alle Anwendungen der *goal success*-Regel in ASM7, die zum Schluß in ASM7 vor dem *query success* stattfinden. Die behauptete 1:1-Zuordnung der ASM-Regeln ist dann nicht gegeben. Analysiert man die folgende ASM9, so stellt man fest, daß die Optimierung dort außerdem (im wesentlichen) wieder rückgängig gemacht wird. Wir haben deshalb den Regeltest für *query success* zu

$$\text{goalreg} = \bot \wedge \text{ereg} = \bot$$

abgeändert. Dies entspricht genau dem Regeltest der folgenden ASM9, und entspräche in ASM7 einem Test *decglseqreg* = [⟨[ ],*ctpt*⟩]. Damit werden also die letzte *goal success* und *query success*-Anwendung aus ASM7 zur Anwendung von *query success* in ASM8 zusammengezogen Es ist an dieser Stelle also ein 2:1-Diagramm zu verifizieren. Das 2:1-Diagramm seinerseits ist unvermeidlich, da man aus der Zusammenhangsinvariante leicht abliest, daß es gar keine Möglichkeit gibt, in ASM8 einen Zustand zu repräsentieren, der *decglseqreg* = [ ] entspricht.

Stack-Sharing Die heikelste Aufgabe bei der Erstellung der Zusammenhangsinvariante stellt die Explizitmachung des Stack-Sharing in ASM8 dar. Die Zusammenhangsinvariante muß sicherstellen, daß das Allokieren von Choicepoints und Umgebungen keine noch relevanten Choicepoints oder Umgebungen überschreibt. Um in den folgenden Formeln passende Aufrufe des *STACK#*-Programms zu sparen bezeichnen wir dazu im folgenden mit *estack* den aktuellen (bei *ereg* beginnenden) Stack von Umgebungen, mit *bstack* den (bei *breg* beginnenden) aktuellen Stack von Choicepoints, und mit *estack*[*n*] den mit *e*[*n*] beginnenden Umgebungsstack eines Choicepoints *n*. Dann werden in der Zusammenhangsinvariante zunächst die folgenden offensichtlichen Eigenschaften gebraucht:

- Der Choicepointstack *bstack* und der Umgebungsstack *estack* sind disjunkt (durch das Prädikat *disjoint*(*estack,bstack*) formalisiert)
- Der Choicepointstack *bstack* ist ebenfalls disjunkt zu jedem Umgebungsstack *estack*[*n*] für jeden Choicepoint $n \in$ *bstack*
- Die Choicepoints in *bstack* sind bzgl. $\ll$ absteigend geordnet (durch *ordered*(*bstack*) formalisiert)

- Auch die Elemente der Umgebungsstacks *estack* und *estack*[n] sind absteigend geordnet
- Die Umgebung $e[n]$ eines Choicepoints n liegt unterhalb des Choicepoints (dies ist die Aussage des Hiding Lemmas).

Leider reichen diese Eigenschaften nicht aus. Die Verifikation zeigte, daß noch eine Anzahl weiterer Eigenschaften notwendig ist, die zunächst nicht offensichtlich sind. Die beiden wichtigsten sind:

- *breg* liegt nie unterhalb von *cutpt*[*ereg*]
- $ct[n]$ liegt nie oberhalb des Choicepoints n, und auch nie unterhalb von *cutpt*$[e[n]]$

Hinzu kommen Eigenschaften, die garantieren, daß keine Zustände unterhalb von $\bot$ liegen, sowie die schon öfter benötigten *cutptsin*-Eigenschaften für Cutpoints.

Verschiebung der Substitution Die Verschiebung der Substitution auf den spätmöglichsten Zeitpunkt scheint zunächst eine harmlose Modifikation zu sein. Versucht man aber die Verträglichkeit der beiden *enter*-Regeln von ASM7 und ASM8 zu zeigen, so stößt man auf das Problem, daß die *auf einen Aktivator in ASM7 und ASM8 angewandte Substitutionen verschieden sind.* Betrachten wir dazu die Situation, in der ein Aktivator mit dem Kopf einer bereits mit *vireg* umbenannten Klausel $H : -B$ unifiziert wird. Nehmen wir an, daß die bisher berechnete Substitution in *subreg* und *subreg*$'$, sowie die beiden Aktivatoren *act* und *act*$'$ gleich sind. Dann berechnen sowohl ASM7 als auch ASM8 denselben Unifikator *mgu*. Beide berechnen dann ein neues goal, das aus dem Literal B besteht. ASM7 instanziert B sofort mit *mgu*, ASM8 berechnet nur die neue Substitution *subreg* $\circ$ *mgu*. Wenn nun B selbst zum Aktivator wird, instanziert ASM8 B mit dieser zusammengesetzten Substitution, und nicht nur mit dem *mgu*. Damit die beiden Aktivatoren gleich sind, muß also

$$(\text{subreg} \circ \text{mgu}) \hat{}_d \text{ B} = \text{mgu} \hat{}_d \text{ B}$$

gelten. Dies ist der Fall, weil die Anwendung von *subreg* keinen Effekt auf B hat. Die Klausel $H : -B$, also insbesondere B wurde ja mit einem *vireg* umbenannt, das vorher nicht verwendet wurde. Somit sollten die Variablenbindungen von *subreg* keine mit *vireg* umbenannten Variablen enthalten. Um dieses Argument zu formalisieren, haben wir Prädikate $cl <_{cvi}$ *vireg*, $L <_{tvi}$ *vireg*, $dgl <_{dvi}$ *vireg* und *subreg* $<_{svi}$ *vireg* definiert, die besagen, daß die Klausel *cl*, die decorated goal-Liste *dgl*, das Literal L bzw. die Substitution *subreg* keine mit *vireg* umbenannten Variablen enthalten. Der Nachweis, daß *subreg* keine Wirkung auf das Literal B hat, reduziert sich dann darauf, daß für die Umbenennungsfunktion *rent* auf Literalen

$$\text{rent(L,vireg)} <_{tvi} \text{vireg} + 1$$

gilt. Diese Forderung kann aber bei einer natürlichen Definition der Umbennung, die die Homomorphie-Eigenschaft besitzt (für die also $rent(f(t),vireg) = f(rent(t,vireg))$ gilt), nur gewährleistet werden, **wenn das umzubennende Literal keine bereits umbenannten Variablen enthält**. Daraus ergibt sich die

> **Compilerannahme für die Verfeinerung 7/8:** Das zu compilierende Prolog-Programm enthält keine umbenannten Variablen.

Die Annahme ist natürlich in der Realität einfach dadurch gegeben, daß umbenannte Variablen keine einlesbare Repräsentation haben. Dennoch macht die formale Verifikation hier eine implizite Anforderung explizit, die bei der informellen Analyse nicht entdeckt wurde.

Die neue Compilerannahme formulieren wir über die Ausgangsdatenbasis *db* von ASM1:

$$\text{mapclause(procdef(lit,db),db)} <_{clvi} 0$$

Sie läßt sich über die bisherigen Compilerannahmen leicht auf die Datenbasis db_7 von ASM7 propagieren.

Als Zusammenhangsinvariante erhalten wir schließlich nach 12 Versuchen mit einem Aufwand von 1 Monat die folgende Formel:

$$
\begin{array}{l}
\text{INV}_{78} \equiv \\
\exists\ \text{F}.\quad \text{F}[\bot] = \bot \wedge \bot \in \text{s} \\
\quad \wedge\ (\text{stop} = \text{run} \rightarrow \text{decglseqreg} \neq []) \\
\quad \wedge\ \text{stop} = \text{stop}' \wedge \text{preg} = \text{preg}' \wedge \text{vireg} = \text{vireg}' \wedge \text{subreg} = \text{subreg}' \\
\quad \wedge\ \text{ctreg} = \text{F}[\text{ctreg}'] \wedge \text{breg} = \text{F}[\text{breg}'] \wedge \text{ce}[\bot] = \bot \wedge \text{cutpt}[\bot] = \bot \\
\quad \wedge\ \neg\ \text{breg}' \ll \bot \wedge (\text{breg}' \neq \bot \rightarrow \text{b}'[\text{breg}'] \ll \text{breg}') \wedge \neg\ \text{ereg} \ll \bot \\
\quad \wedge\ \text{subreg}' <_{svi} \text{vireg}' \wedge \neg\ \text{breg}' \ll \text{cutpt}[\text{ereg}] \\
\quad \wedge\ (\quad \text{preg}' \neq \text{start} \wedge \text{stop}' = \text{run} \\
\qquad \rightarrow \text{goalreg} \neq [] \wedge \text{is\_ret}(\text{code}(\text{preg}', \text{db}_7)) \\
\qquad\quad \supset \text{breg} \neq \bot \wedge \text{preg}' = \text{p}'[\text{breg}']\ ; \\
\qquad\qquad \neg\ \text{breg}' \ll \text{ctreg}' \wedge \neg\ \text{ctreg}' \ll \text{cutpt}[\text{ereg}] \\
\qquad\quad \wedge\ \langle \text{S-CHAIN-REC\#(act, preg, db}_7\text{; col)}\rangle\ \text{tt})) \\
\quad \wedge\ \langle \text{STACK\#(breg}', \text{b}'\text{; stack}')\rangle \\
\qquad (\quad \langle \text{STACK\#(breg, b; stack)}\rangle\ (\text{F}_1(\text{F}, \text{stack}') = \text{stack} \wedge \text{stack} \subseteq \text{s}) \\
\qquad\ \wedge\ \text{F injon stack}' \wedge \text{ordered(stack}') \\
\qquad\ \wedge\ (\quad \text{stop} = \text{run} \\
\qquad\quad \rightarrow \langle \text{STACK\#(ereg, ce; estack)}\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle \text{decglseqreg}' := [\langle \text{goalreg, cutpt[ereg]} \rangle \mid \\
\qquad\qquad\qquad\qquad \text{decglseqof(cutpt, cg, ce, estack)}] \\
\quad (\ \ \text{decglseqreg} = \text{subreg} \mathbin{\hat{}}_d \mathrm{F_d}(\mathrm{F}, \text{decglseqreg}') \\
\quad \wedge\ \text{decglseqreg}' <_{dvi} \text{vireg}' \\
\quad \wedge\ \text{disjoint(estack, stack}') \wedge \text{ordered(estack)} \\
\quad \wedge\ (\text{preg}' = \text{start} \supset \text{decglseqreg}' \text{ cutptsin stack}' \ ; \\
\qquad\qquad \neg\ \text{is\_ret(code(preg}', \mathrm{db}_7)) \\
\qquad\qquad \rightarrow\ \ \text{decglseqreg}' \text{ cutptsin stack}' \text{ from ctreg}' \\
\qquad\qquad\quad \wedge\ (\text{ctreg}' = \bot \vee \text{ctreg}' \in \text{stack}')))) \\
\wedge\ \forall\ \text{n.}\ \mathrm{STACKINV}_{78}),
\end{array}
$$

wobei

$$
\begin{array}{l}
\text{is\_ret(instr)} \leftrightarrow\ \ \text{is\_retry(instr)} \vee \text{is\_retry\_me(instr)} \\
\qquad\qquad\qquad \vee\ \text{is\_trust(instr)} \vee \text{is\_trust\_me(instr)}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{STACKINV}_{78} \equiv \\
\quad \text{n} \in \text{stack}' \\
\rightarrow\ \ \text{sub[F[n]]} = \text{sub}'\text{[n]} \wedge \text{p[F[n]]} = \text{p}'\text{[n]} \wedge \text{ct[F[n]]} = \text{F[ct}'\text{[n]]} \\
\quad \wedge\ \text{b[F[n]]} = \text{F[b}'\text{[n]]} \wedge (\text{ct}'\text{[n]} \neq \bot \rightarrow \text{ct}'\text{[n]} \in \text{cdr(stack}' \text{ from n})) \\
\quad \wedge\ \neg\ \text{n} \ll \text{ct}'\text{[n]} \wedge \text{e[n]} \ll \text{n} \wedge \neg\ \text{e[n]} \ll \bot \wedge \neg\ \text{ct}'\text{[n]} \ll \text{cutpt[e[n]]} \\
\quad \wedge\ \neg\ \text{breg}' \ll \text{n} \wedge \text{goal[n]} \neq [] \wedge \text{sub}'\text{[n]} <_{svi} \text{vireg}' \\
\quad \wedge\ \langle \text{S-CHAIN-RET\#(act(F[n]), p[F[n]], db}_7\text{; col)} \rangle\ \text{tt} \\
\quad \wedge\ \langle \text{STACK\#(e[n], ce; estack)} \rangle \\
\qquad \langle \text{decglseqreg}' := [\langle \text{goal[n], cutpt[e[n]]} \rangle \mid \\
\qquad\qquad\qquad\qquad \text{decglseqof(cutpt, cg, ce, estack)}] \rangle \\
\qquad\quad (\ \ \text{decglseqreg}' \text{ cutptsin stack}' \text{ from ct}'\text{[n]} \\
\qquad\quad \wedge\ \text{decglseqreg}' <_{dvi} \text{vireg}' \\
\qquad\quad \wedge\ \text{disjoint(estack, stack}' \text{ from n)} \wedge \text{ordered(estack)} \\
\qquad\quad \wedge\ \text{decglseq[F[n]]} = \text{sub[F[n]]} \mathbin{\hat{}}_d \mathrm{F_d}(\mathrm{F}, \text{decglseqreg}')
\end{array}
$$

Kapitel 17

8/9: Compilation von Klauseln

17.1 Definition von ASM9

Im Refinement von ASM8 zu ASM9 wird die Bearbeitung von Klauseln in Einzelinstruktionen für jedes Literal zerlegt. Dazu wird im Speicher db_9 von ASM9 statt einer Klausel `p :- q1, ...qn` nun die Instruktionsfolge

$$\begin{array}{l} \text{allocate} \\ \text{unify(p)} \\ \text{call}(\mathrm{q}_1) \\ \ldots \\ \text{call}(\mathrm{q}_n) \\ \text{deallocate} \\ \text{proceed} \end{array} \tag{17.1}$$

abgelegt. Die Rolle des *goalreg* aus ASM8 im Fall $preg = start$ wird nun vom Programmzähler $preg'$ der ASM9 selbst übernommen (solange $preg \neq start$ gilt, sind *preg* und $preg'$ gleich). $goalreg = [q_i, \ldots q_n]$ entspricht nun der Situation, in der $preg'$ auf eine Anweisung $call(q_i)$ zeigt. Die Situation in ASM8, in der *preg* auf eine Klausel zeigt und ein Aufruf der *enter*-Regel erfolgt, entspricht nun der Situation, in der $preg'$ auf das *allocate* zeigt. Die Abarbeitung der *enter*-Regel wird durch die Abarbeitung der 2 Anweisungen *allocate* und *unify*(*p*) ersetzt. Analog wird die Bearbeitung von *goal success* (ein leeres *goalreg* aus ASM8 entspricht der Situation, in der $preg'$ auf *deallocate* zeigt) durch die Abarbeitung von der *deallocate* und *proceed* ersetzt. Die Aufspaltung von *enter* und *goal success* in je zwei Anweisungen wäre für die hier gemachte Übersetzung nicht notwendig, sie führt aber sofort die in der WAM verwendeten Instruktionen ein, und erlaubt außerdem in späteren Übersetzungsschritten Optimierungen.

Um *goalreg* abschaffen zu können, muß noch beachtet werden, daß die bisher schon bei der Neubelegung von *goalreg* in der *enter*-Regel durchgeführte Umbenennung (mit *vireg*) der Klauselvariablen nun auf den Zeitpunkt der Verwendung des Aktivators aufgeschoben werden muß. Es ist deshalb notwendig, den entsprechenden Umbenennungsindex nun mit Hilfe einer dynamischen Funktion *vi* in der aktuellen Umgebung sowie in den Umgebungen aller Choicepoints zu speichern.

Mit der Ersetzung von *goalreg* durch *preg* wird in der Verfeinerung 8/9 analog auch das durch *cg* bezeichnete Goal durch einen Zeiger *cp* in den Code ersetzt.

Um die Übersetzung zu vervollständigen, muß schließlich noch die Anfrage $q_i, \ldots q_n$ übersetzt werden. Das Ergebnis ist:

$$\begin{array}{l} \text{call}(\text{q}_1) \\ \ldots \\ \text{call}(\text{q}_n) \\ \text{null} \end{array} \tag{17.2}$$

In [BR95] wird anstelle der von uns verwendeten Instruktion *null*[1] die Instruktion *proceed* verwendet. Der Anwendbarkeitstest für die *query success*-Regel lautet dort

$$\text{code}(\text{preg},\text{db}_9) = \text{proceed} \wedge \text{code}(\text{cpreg},\text{db}_9) = \text{proceed}$$

Dies ist inkorrekt, wenn das letzte Literal einer Anfrage ein *!* oder *true* ist, da beide Anweisungen das *cpreg* nicht erhöhen, sondern auf der Adresse der Instruktion belassen. Es ergäbe sich eine Endlosschleife durch wiederholte Ausführung der letzten Anweisung. Zu unserer Lösung mit einer expliziten *null*-Anweisung gibt es zwei Alternativen:

- Die *cut* und *true*-Regel setzen am Ende *cpreg* auf *preg*. Diese Lösung ist ineffizient, da das Setzen von *cpreg* im normalen Ablauf völlig überflüssig ist.
- Der Compiler entfernt die Literale *true* und *!* am Ende einer Anfrage. Sie haben ohnehin keine Wirkung. Die Lösung ist für die beiden gegebenen Konstrukte möglich, allerdings ist sie insofern problematisch, als bei einer Erweiterung von Prolog um weitere nicht-benutzerdefinierte Konstrukte (wie etwa *assert*) das Problem erneut betrachtet werden muß.

Zu beachten ist bei beiden Alternativlösungen außerdem, daß sie gegenüber gegenüber unserer zwei Irregularitäten aufweisen:

[1] um keine weitere Instruktion einzuführen, wird die Instruktion *null*, die in ASM2 das Ende einer Klauselliste anzeigte, wiederverwendet.

- Eine leere Query muß speziell durch Initialisierung von *cpreg* mit *preg* behandelt werden, oder ganz verboten werden (in unserer Lösung ist keine Spezialbehandlung notwendig, *cpreg* muß nicht initialisiert werden). Die leere Query ergibt im ersten Fall ein zusätzlich zu verifizierendes 1:1-Diagramm.

- Die von [BR95] angegebene und in unserer Lösung gültige Regelabbildung von *goal success* auf *deallocate* und *proceed* (1:2-Diagramm) wird *nicht* eingehalten. Vielmehr korrespondieren bei der ersten Lösung (bei einer nichtleeren Anfrage) die letzten beiden Anwendungen der Regeln *goal success* und *query success* aus ASM8 dann zu den beiden Regelanwendungen *deallocate* und *query success* von ASM9. Bei der zweiten Lösung entstehen außerdem durch das Weglassen der *true*- und !-Literale zusätzliche 1:0-Diagramme.

In [AK91] wird die Frage der erfolgreichen Abarbeitung einer Anfrage gar nicht betrachtet. Eine Anfrage scheint nur in die entsprechenden *call*-Instruktionen compiliert zu werden, und das Ende eines Programms scheint implizit durch das Erreichen der dem letzten *call* folgenden Adresse gegeben zu sein.

Um die eben dargestellte Annahmen in eine Compilerannahme zu übersetzen, wird zunächst folgende Prozedur benötigt, die aus compiliertem Code die Klausel bzw. die Anfrage zurückgewinnt:

```
UNLOAD#(coa, db9; var cl)
begin
    if code(coa,db9) = allocate ∨ is_unify(code(coa+1,db9))
    then var goalreg = []
         in begin
            UNLOADREC#(coa+2),db9,true; goalreg);
            cl := <unifylit(code(coa+1,db9)),goalreg>
            end
         else abort
end;
```

```
UNLOADREC#(coa, db9, flag; var goalreg)
begin
    var instr = code(coa,db9)
    in if flag ∧ (instr = deallocate)
       then begin
            if code(coa+1,db9) = proceed then goalreg := []
                                         else abort end
            else if ¬ flag ∧ (instr = null′) then goalreg := []
            else if is_call(instr) then begin
                 UNLOADREC#(coa+1,db9,flag; goalreg);
                 goalreg := [calllit(instr) | goalreg]
                 end
            else abort
    end;

QUERY#(coa, db9; var goalreg)
begin
    UNLOADREC#(coa, db9, false; goalreg)
end
```

Die Prozeduren *UNLOAD#* und *QUERY#* berechnen aus dem Klauselcode der Form (17.1) bzw. aus dem Anfragecode der Form (17.2) die Klausel bzw. Anfrage, aus der er entstanden ist. Die Hilfsprozedur *UNLOADREC#* sammelt aufeinanderfolgende *call*-Instruktionen. Ist das übergebene *flag* = *tt*, so muß am Ende ein *allocate* und *proceed* stehen (Klauselcode), sonst ein *null* (Anfragecode). Die bisherige Definition von Ketten mit Switching (*S-CHAIN#*'s, siehe Anhang D.2) wird durch Ersetzung der Anweisung

```
if is_clause(instr) then col := [co]
```

durch

```
if instr = allocate then UNLOAD#(preg; co)
```

zu *C-CHAIN#*'s modifiziert. Damit ließe sich als schwächste Compilerannahme für $compile_{79}(procdef_7, db_7, query) = (procdef_9, db_9, preg_9)$

$$\begin{aligned}&[\text{S-CHAIN\#}(\text{act},\text{procdef}_7[\text{id}(\text{act}),\text{db}_7],\text{db}_7;\text{col})]\\&\quad\langle\text{C-CHAIN\#}(\text{act},\text{procdef}_9[\text{id}(\text{act}),\text{db}_9],\text{db}_9;\text{col})\\&\qquad\text{mapcode}(\text{col}_1,\text{db}_7) = \text{mapcode}(\text{col}_2,\text{db}_9)\\&\wedge\ \langle\text{QUERY\#}(\text{preg}_9,\text{db}_9;\text{co})\rangle\ \text{mapcode}(\text{co},\text{db}_9) = \text{query}\end{aligned} \tag{17.3}$$

formulieren. Diese Annahme würde es aber erlauben, den Code für Switching und Backtracking erneut beliebig umzustrukturieren. Dies ist natürlich nicht intendiert. Deshalb ist eine stärkere Annahme zu treffen, die nur die Klauseln

durch den Klauselcode ersetzt. Dafür ist zu beachten, daß es durch das Einfügen von neuem Code zu einer Codeverschiebung kommt. Um diese zu beschreiben, definieren wir eine Funktion C : *codesort* $\rightarrow$ *codesort*. Da die Funktion vom Eingabeprogramm abhängt, muß sie als dynamische Funktion spezifiziert werden. Es wäre möglich die Funktion C als zusätzliches Ergebnis von $compile_9$ zu liefern, da aber nur ihre Existenz relevant ist, lautet die Compilerannahme:

$$\begin{array}{ll} & \mathrm{db}_2 = \mathrm{compile2}(\mathrm{compile1}(\mathrm{db})) \\ \rightarrow & \quad \langle \mathrm{QUERY\#}(\mathrm{preg}_9,\mathrm{db}_9;\mathrm{co})\rangle\ \mathrm{mapcode}(\mathrm{co},\mathrm{db}_9) = \mathrm{query} \\ & \wedge\ \exists\ \mathrm{C}.\ (\quad \mathrm{eqpdt}(\mathrm{procdef}_7,\mathrm{procdef}_9,\mathrm{C}) \\ & \qquad \wedge\ \mathrm{eqcode}(\mathrm{db}_7,\mathrm{db}_9,\mathrm{C})) \end{array} \tag{17.4}$$

Dabei besagt *eqpdt*($procdef_7$,$procdef_9$,C), daß die beiden Zugriffstabellen modulo der durch C gegebenen Codeverschiebung gleich sind:

$$\begin{array}{ll} & \mathrm{eqpdt}(\mathrm{procdef}_7,\mathrm{procdef}_9,\mathrm{C}) \\ \leftrightarrow & \forall\ \mathrm{p/n.}\ \mathrm{C}[\mathrm{procdef}_7[\mathrm{p/n}]] = \mathrm{procdef}_9[\mathrm{p/n}] \end{array}$$

eqcode(db_7,db_9,C) bedeutet, daß alle Instruktionen, bis auf Klauseln, modulo der Codeverschiebung auf sich selbst abgebildet werden. Es gilt also z.Bsp.

$$\begin{array}{ll} & \mathrm{eqcode}(\mathrm{db}_7,\mathrm{db}_9,\mathrm{C}) \wedge \mathrm{code}(\mathrm{preg},\mathrm{db}_7) = \mathrm{retry}(\mathrm{N}) \\ \rightarrow & \mathrm{code}(\mathrm{C}[\mathrm{preg}],\mathrm{db}_9) = \mathrm{retry}(\mathrm{C}[\mathrm{N}]) \end{array}$$

und analog für alle anderen Instruktionen. Für Klauseln gilt:

$$\begin{array}{ll} & \mathrm{eqcode}(\mathrm{db}_7,\mathrm{db}_9,\mathrm{C}) \wedge \mathrm{code}(\mathrm{preg},\mathrm{db}_7) = \mathrm{clause} \\ \rightarrow & \langle \mathrm{UNLOAD\#}(\mathrm{C}[\mathrm{preg}],\mathrm{db}_9;\mathrm{c})\rangle \mathrm{c} = \mathrm{clause} \end{array}$$

Die Regeln für ASM9 lauten:

backtrack $\equiv$
  if breg $= \bot$ **then** stop := failure
    else preg := p[breg]

call rule
  if code(preg,db_9) = call(lit) $\wedge$ is_user_defined(lit)
  then if $\mathrm{procdef}_9$(lit,db_9) = failcode
    then backtrack
    else cpreg := preg +1
      preg := $\mathrm{procdef}_9$(lit,db_9)
      ctreg := breg

true rule
  if code(preg,db_9) = call(!)
  then breg := cutpt[ereg]
    preg := preg +1

allocate rule
if code(preg, db_9) = allocate
then let tmp = max(ereg,breg)++
ce[tmp] := ereg
ereg := tmp
cp[tmp] := cpreg
vi[tmp] := vireg
cutpt[tmp] := ctreg
preg := preg +1

unify rule
if code(preg, db_9) = unify(trm)
then let act = subreg $\hat{}_t$ rent′(calllit(code(cpreg −1, db_9)), ce[ereg], vi)
let mgu = unify(act, rent(trm, vireg))
if mgu = nil
then backtrack
else subreg := subreg ∘ mgu
vireg := vireg +1
preg := preg +1

deallocate rule
if code(preg,db_9) = deallocate
then cpreg := cp[ereg]
ereg := ce[ereg]
preg := preg +1

true rule
if code(preg,db_9) = call(fail)
then backtrack

proceed rule
if code(preg,db_9) = proceed
then preg := cpreg

query success rule
if code(preg,db_9) = null′
then stop := success

retry rule
if code(preg,db_9) = retry(N)
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := preg +1
preg := N

retry_me_else rule
if code(preg,db_9) = retry_me_else(N)
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
p[breg] := N
preg := preg +1

switch_on_constant rule
let act = subreg $\hat{}_t$ rent′(calllit(code(cpreg −1, db_9)), ereg, vi)
if code(preg, db_9) = switch_on_constant(i, tabsize, table)
then let x_i = arg(act, i)
preg := hashc(table, tabsize, constsym(x_i), db_9);
if preg = failcode **then backtrack**

switch_on_structure rule
let act = subreg $\hat{}_t$ rent′(calllit(code(cpreg −1, db_9)), ereg, vi)
if code(preg, db_9) = switch_on_structure(i, tabsize, table)
then let x_i = arg(act, i)
preg := hashs(table, tabsize, funct(x_i), arity(x_i), db_9);
if preg = failcode **then backtrack**

switch_on_term rule
let act = subreg $\hat{}_t$ rent′(calllit(code(cpreg −1, db_9)), ereg, vi)
if code(preg, db_9) = switch_on_term(i, N_s, N_c, N_v, N_l)
then let x_i = arg(act, i)
if is_struct(x_i) **then** preg := N_s **else**
if is_const(x_i) **then** preg := N_c **else**
if is_var(x_i) **then** preg := N_v **else**
if is_list(x_i) **then** preg := N_l;
if preg = failcode **then backtrack**

true rule
if code(preg,db_9) = call(true)
then preg := preg +1

trust rule
if code(preg,db_9) = trust(N)
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := N

trust_me rule
if code(preg,db_9) = trust_me
then ereg := e[breg]
cpreg := cp[breg]
ctreg := ct[breg]
subreg := sub[breg]
breg := b[breg]
preg := preg +1

try rule
if code(preg,db_9) = try(N)
then let tmp = max(ereg,breg)++
b[tmp] := breg
e[tmp] := ereg
cp[tmp] := cpreg
sub[tmp] := subreg
p[tmp] := preg +1
breg := tmp
ct[tmp] := ctreg
preg := N

try_me rule
if code(preg,db_9) = try_me_else(N)
then let tmp = max(ereg,breg)++
b[tmp] := breg
e[tmp] := ereg
cp[tmp := cpreg
sub[tmp] := subreg
p[tmp] := N
breg := tmp
ct[tmp] := ctreg
preg := preg +1

17.2 Äquivalenzbeweis 8/9

Für den Äquivalenzbeweis von ASM8 und ASM9 haben wir zum ersten Mal die in Abschnitt (6.5) beschriebene Technik für iterative Verfeinerung genutzt. Statt alle Information in die Zusammenhangsinvariante INV_{89} zu codieren, haben wir zunächst aus INV_{78} eine Maschineninvariante $MINV_8$ hergeleitet. Da alle Diagramme aus 7/8 n:1-Diagramme sind, genügt es, direkt

$$\mathrm{INV}_{78} \rightarrow \mathrm{MINV}_8$$

zu zeigen, um $MINV_8$ als Vorbedingung für alle kommutierenden Diagramme verwenden zu dürfen. Um auch für das folgende Refinement eine Maschineninvariante $MINV_9$ zur Verfügung zu haben, benötigen wir, wie dies im Abschnitt

(6.5) dargestellt ist, außerdem ein Prädikat $INVNOW_8$, das die Zustände von ASM9 charakterisiert, in denen die Zusammenhangsinvariante INV_{89} gilt. Nun werden in der Verfeinerung 8/9 alle Regeln mit 1:1-Diagrammen verfeinert, bis auf die *enter*-Regel und die *goal success*-Regel, die durch *allocate unify* bzw. *deallocate proceed* verfeinert werden. Die Zusammenhangsinvariante, gilt also lediglich nicht in den Zwischenzuständen dieser 1:2-Diagramme. Somit ist

$$\begin{aligned} &\mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9) \\ \equiv\ &\mathrm{code}(\mathrm{preg}',\mathrm{db}_9) \neq \mathrm{proceed} \wedge \neg\ \mathrm{is\_unify}(\mathrm{code}(\mathrm{preg}',\mathrm{db}_9)) \end{aligned}$$

Die Beweisverpflichtungen für die beiden 1:2-Diagramme ergeben sich dann als Spezialfall mit $j := 2$ und $i := 1$ von Beweisverpflichtung (6.32) aus Abschnitt (6.5):

$$\begin{aligned} &\mathrm{INV}_{89} \wedge \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \\ &\wedge\ \mathrm{MINV}_8 \wedge \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg},\mathrm{db}_7)) \\ \rightarrow\ &\langle \mathrm{RULE}_9 \rangle\ (\quad \neg\ \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9) \\ &\qquad\qquad \wedge\ \langle \mathrm{RULE}_9 \rangle\ \langle \mathrm{RULE}_8 \rangle \\ &\qquad\qquad\quad (\mathrm{INV}_{89} \wedge \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9)) \end{aligned}$$

$$\begin{aligned} &\mathrm{INV}_{89} \wedge \mathrm{stop} = \mathrm{run} \wedge \mathrm{stop}' = \mathrm{run} \\ &\wedge\ \mathrm{MINV}_8 \wedge \mathrm{is\_clause}(\mathrm{code}(\mathrm{preg},\mathrm{db}_7)) \\ \rightarrow\ &\langle \mathrm{RULE}_9 \rangle\ (\quad \neg\ \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9) \\ &\qquad\qquad \wedge\ \langle \mathrm{RULE}_9 \rangle\ \langle \mathrm{RULE}_8 \rangle \\ &\qquad\qquad\quad (\mathrm{INV}_{89} \wedge \mathrm{INVNOW}_9(\mathrm{preg}',\mathrm{db}_9)) \end{aligned}$$

Für die Definition der Zusammenhangsinvariante ergaben sich 4 wesentliche Probleme:

Korrekte Behandlung der Terminierung In den ersten Beweisversuchen hatten wir versucht, uns genau an [BR95] zu halten. Dabei zeigten sich zunächst die schon im vorigen Abschnitt geschilderten Probleme: Zunächst mußte die Wahl der Diagramme korrigiert werden (ein spezielles Diagramm für die leere Anfrage und ein 2:2-Diagramm für *goal success*, *query success* aus ASM8 vs. *deallocate query success* in ASM9). Schließlich scheiterte der Versuch, die Äquivalenz der cut-Regeln zu zeigen, da die cut-Regel von ASM9 *cpreg* nicht verändert. Daraus ergab sich dann die im vorigen Abschnitt angegebenen korrigierte Regel für *query success* in ASM9.

Keine Instanzierung des Literals in der call-Regel Sowohl beim Test auf *is_user_defined* als auch bei der Bestimmung des führenden Prädikatsymbols wurde in den *call*-Regeln bis ASM8 immer der *instanzierte* Aktivator verwendet. In ASM9 wird nun stattdessen für beide Zwecke das *nicht instanzierte* Literal L aus der Instruktion $call(L)$ benutzt. Für die Berechnung des führenden Prädikatsymbols haben wir dabei den von [BR95] erst in der folgenden Verfeinerung 9/10 durchgeführten Schritt vorgezogen. Dies geschah, um die ohnehin

schon sehr komplexen Verifikation der Termrepräsentation nicht mit unnötigen Zusatzproblemen zu belasten.

Die Verifikation zeigte nun, daß bei Verwendung des nicht instanzierten Literals der Umfang der akzeptierten Prolog-Teilsprache eingeschränkt werden muß: Die bisherigen ASMs 1–8 beantworten eine Anfrage `?- p(q).` bei gegebenen Klauseln `p(X) :- X.` und `q.` positiv. ASM9 kann eine solche Anfrage nicht lösen, da für die nicht instantiierte Variable `X` kein führendes Prädikatsymbol definiert ist. ASM9 in [BR95] versucht sogar inkorrekt, bei einer Anfrage `?- p(!)` auf das obige Programm, das führende Prädikatsymbol des Cut zu bestimmen, statt einen Cut auszuführen. Die Schwierigkeit, ein führendes Prädikatsymbol definieren zu müssen, ergibt sich auch für Listen als Rumpfliterale. Da übliche Prologimplementierungen kein „Listenprädikat“ zulassen, sondern die Anfrage ein derartiges Literal als Kommando zum Laden von Dateien interpretieren, benötigen wir also:

> **Compilerannahme für die Verfeinerung 8/9:** Kein Literal der Query und kein Literal einer Klausel aus dem Prologprogramm ist eine Variable oder eine Liste.

Natürlich konnten auch die bisherigen ASMs eine Anfrage `?- X.` nicht „sinnvoll“ lösen, da es für eine Variable `X` keine sinnvolle Selektion eines führenden Prädikatsymbols gibt. Dies spielte aber für die Korrektheit der Verfeinerungen keine Rolle. Wie auch immer die Selektionsfunktion in Fall einer Variable spezifiziert wird, alle bisherigen ASMs verhielten sich gleich. Das Problem zeigt somit, daß ohne die obige Compilerannahme die Ausgangsdefinition der Prologsemantik unvollständig ist.

Würden wir die Compilerannahme für die Verfeinerung 8/9 neu treffen, so würde sie in die Zusammenhangsinvariante einfließen. Für alle Literale, für die wir aus der Maschineninvariante $MINV_8$ für ASM8 schon wissen, daß sie nicht umbenannt sind, würden wir in INV_{89} nun noch zusätzlich benötigen, daß es sich um keine Variablen und keine Listen handelt. Dies würde bedeuten, daß die Ketten, aus denen die Literale extrahiert werden, zweimal, sowohl in $MINV_8$ als auch in INV_{89} berechnet werden müßten. Um dies zu vermeiden, haben wir das in der Compilerannahme 7/8 verwendete Prädikat $cl <_{cvi} vireg$ so verstärkt, daß es nun beide Compilerannahmen umfaßt. Die Beweise für die Verfeinerung 7/8 ändern sich dadurch nicht (die Voraussetzungen werden ja nur stärker), und die Annahme, daß keine Variablen in Klauselrümpfen vorhanden sind, wird nun bereits durch $MINV_8$ abgedeckt.

Verschiebung der Umbenennung auf die Verwendung des Aktivators Da goals in ASM9 nicht mehr explizit in einem Register gespeichert werden, sondern nur noch über Zeiger auf Klauselcode referenziert werden, muß die vor der Unifikation notwendige Umbenennung der Klauselvariablen auf die Verwendung der Einzelliterale des Rumpfs als Aktivator aufgeschoben, und der korrekte Umbenennungsindex in der Umgebung zwischengespeichert. Zur Rekonstruktion

von Goals aus Codezeigern verwenden wir wegen der Terminierungsproblematik die Prozedur *UNLOADREC#* aus der Compilerannahme, für die Umbenennung haben wir zunächst eine Funktion *reng* verwendet, die ein Goal mit einem Umbenennungsindex *vireg* umbenennt (*reng* ist einfach homomorph zur schon früher verwendeten Funktion *rename* für die Anwendung einer Umbenennung auf Klauseln definiert). In [BR95] ist das Aufsammeln von Literalen und das Anwenden der Umbenennung beides in die auf Seite S. 34f definierte Funktion g codiert. Die Annahme $goalreg = g(Ptr,vireg)$ lautet also in unserer Notation:

$$\langle \text{UNLOADREC\#(Ptr,db}_9\text{;goal)}\rangle\ \text{goalreg} = \text{reng(goalreg,vireg)}$$

Es zeigt sich aber, daß diese Annahme nicht korrekt ist, wenn *goalreg* ein Teil der ursprünglichen Anfrage ist. Diese darf nämlich *nicht* umbenannt werden. Der Fall liegt immer dann vor, wenn versucht wird, auf das unspezifizierte $vi[ereg]$ bei $ereg = \perp$ zuzugreifen. Wir haben den Ausnahmefall explizit durch die Definition einer Funktion $reng'(goalreg,ereg,vi)$ behandelt, die durch

$$\text{reng}'(\text{goalreg},\perp,\text{vi}) = \text{goalreg}$$
$$\text{ereg} \neq \perp \rightarrow \text{reng}'(\text{goalreg},\perp,\text{vi}) = \text{reng(goalreg,vi[ereg])}$$

spezifiziert ist. Eine Alternative wäre gewesen, $vi[\perp]$ so zu initialisieren, daß die Anwendung des Renamings keine Wirkung hat (also etwa Initialisierung von $vi[\perp]$ mit 0, Initialisierung von *vireg* mit 1, und Definition von $reng(goalreg,0)$ durch *goalreg*).

Rekonstruktion des *goalreg* aus ASM8 aus Daten der ASM9 Zentral für die Definition der Zusammenhangsinvariante ist es, in jedem Zustand den korrekten Zusammenhang zwischen den in ASM8 explizit gespeicherten Goals, und den in ASM9 nur implizit über Zeiger in den Code vorhandenen Goals herzustellen. Im wesentlichen mußte dazu der in [BR95] S. 34 gegebene Continuation Pointer Constraint sowie die Rekonstruktion der Register von ASM8 aus den Daten von ASM9 präzise formalisiert werden. Es zeigt sich, daß die in [BR95], S. 35 gegebene uniforme Rekonstruktion von *goalreg* so nicht möglich ist. Vielmehr ergeben sich drei Fälle:

Im ersten Fall befindet sich ASM8 im Fall $preg = start$, und *goalreg* wird rekonstruiert durch

$$\langle \text{UNLOADREC\#(preg}', \text{db}_9, \text{ereg}' \neq \perp ; \text{goalreg}_0) \text{ end}\rangle$$
$$(\ \text{reng}'(\text{goalreg}_0, \text{ereg}', \text{vi}) = \text{goalreg}$$
$$\wedge\ \text{nonvargoal}(\text{goalreg}_0))$$

Die Nachbedingung $nonvargoal(goalreg_0)$ codiert die Compilerannahme, daß die Literale von *goalreg* weder umbenannt, noch Variablen oder Listen sind.

Im zweiten Fall befinden sich beide ASMs vor einer *retry*-, *retry_me*-, *trust*- oder *trust_me*-Anweisung. In diesem Fall ist kein *goalreg* zu rekonstruieren (es

wird ja gerade aus dem letzten Choicepoint rekonstruiert). Für diesen Fall ist auch zu beachten, daß die beiden Umgebungsregister *ereg* und *ereg′ verschieden* sein können: Bei einem *enter* mit backtracking wird zwar *ereg* aus ASM8 nicht verändert, wohl aber *ereg′* aus ASM9 durch das *allocate*.

Der Continuation Pointer Constraint wird in den beiden ersten Fällen nicht benötigt, er spielt nur eine Rolle für den noch verbleibenden Fall: In diesem ist $preg' = C[preg]$ und *goalreg* wird über *cpreg* -1 berechnet:

$$\begin{aligned}&\langle \text{UNLOADREC\#}(\text{cpreg} -1,\ \text{db}_9,\ \text{ereg}' \neq \bot;\ \text{goalreg}_0)\ \text{end}\rangle\\ &\quad (\quad \text{reng}'(\text{goalreg}_0,\ \text{ereg}',\ \text{vi}) = \text{goalreg}\\ &\quad\ \wedge\ \text{nonvargoal}(\text{goalreg}_0))\end{aligned}$$

Bei der Bestimmung dieser Formel hatten wir mehrfach statt $ce'[ereg']$ *ereg′* verwendet, da sich sonst die Verfeinerung der *enter*-Regel zu *allocate unify* nicht verifizieren ließ. Nach genauerer Analyse zeigte sich schließlich, daß das Problem im verwendeten Umbenennungsindex in der *unify*-Regel lag. [BR95] verwendet über die Abkürzungen *goal* indirekt den Umbenennungsindex $vi[ereg]$ bei der Bestimmung des Aktivators *act*. Dies ist korrekt für die Switching-Regeln und die *call*-Regel, aber nicht für die *unify*-Regel, da unmittelbar zuvor in der *allocate*-Regel schon der *neue* Umbenennungsindex für das bei erfolgreicher Unifikation erst zu erzeugende neue Goal auf den Umgebungsstack gelegt wurde. Der korrekte Umbenennungsindex für den Aktivator steht bei $vi[ce[ereg]]$, sofern $ce[ereg] \neq \bot$. Deshalb ruft die korrigierte *unify*-Regel die Funktion *rent′* mit $ce[ereg]$ auf.

Insgesamt ergaben sich nach 3 Wochen und 8 Iterationen folgende Zusammenhangsinvariante:

$$\begin{aligned}&\text{INV}_{89} \equiv\\ &\quad \text{vireg} = \text{vireg}' \wedge \text{stop} = \text{stop}' \wedge \text{breg} = \text{breg}' \wedge \text{ctreg} = \text{ctreg}' \wedge \text{sub} = \text{sub}'\\ &\wedge \text{subreg} = \text{subreg}' \wedge \text{ct} = \text{ct}' \wedge \text{b} = \text{b}' \wedge \text{e} = \text{e}' \wedge \text{cutpt}'[\bot] = \bot\\ &\wedge (\quad \text{stop} = \text{run}\\ &\quad \rightarrow \quad \neg\ \text{is\_unify}(\text{code}(\text{preg}',\ \text{db}_9)) \wedge \text{code}(\text{preg}',\ \text{db}_9) \neq \text{proceed}\\ &\qquad \wedge (\quad \text{preg} = \text{start}\\ &\qquad\ \supset \quad \text{ereg} = \text{ereg}'\\ &\qquad\quad \wedge\ \langle \text{UNLOADREC\#}(\text{preg}',\ \text{db}_9,\ \text{ereg}' \neq \bot;\ \text{goalreg}_0)\rangle\\ &\qquad\qquad\quad \text{reng}'(\text{goalreg}_0,\ \text{ereg}',\ \text{vi}) = \text{goalreg} \wedge \text{nonvargoal}(\text{goalreg}_0);\\ &\qquad\qquad \neg\ \text{is\_call}(\text{code}(\text{preg}',\ \text{db}_9))\\ &\qquad\quad \wedge\ \text{code}(\text{preg}',\ \text{db}_9) \neq \text{deallocate}\\ &\qquad\quad \wedge\ \text{preg}' = \text{C[preg]}\\ &\qquad\quad \wedge \quad \neg\ \text{is\_ret}(\text{code}(\text{preg},\ \text{db}_7))\\ &\qquad\qquad \rightarrow \quad \text{ereg} = \text{ereg}'\\ &\qquad\qquad\quad \wedge\ \langle \text{UNLOADREC\#}(\text{cpreg}-1,\ \text{db}_9,\ \text{ereg}' \neq \bot;\ \text{goalreg}_0)\rangle\\ &\qquad\qquad\qquad \text{reng}'(\text{goalreg}_0,\text{ereg}',\text{vi}) = \text{goalreg} \wedge \text{nonvargoal}(\text{goalreg}_0))\\ &\qquad \wedge\ \langle \text{STACK\#}(\text{ereg},\ \text{ce};\ \text{estack})\rangle\\ &\qquad\quad \forall\ \text{n}. \quad \text{n} \in \text{estack}\\ &\qquad\qquad \rightarrow \quad \text{ce[n]} = \text{ce}'\text{[n]} \wedge \text{cutpt[n]} = \text{cutpt}'\text{[n]}\\ &\qquad\qquad\quad \wedge\ \langle \text{UNLOADREC\#}(\text{cp[n]},\ \text{db}_9,\ \text{ce[n]} \neq \bot;\ \text{goalreg}_0)\rangle\\ &\qquad\qquad\qquad \text{reng}'(\text{goalreg}_0,\ \text{ce[n]},\ \text{vi}) = \text{cg[n]} \wedge \text{nonvargoal}(\text{goalreg}_0)\end{aligned}$$

$$
\begin{array}{l}
\wedge\ \langle \mathrm{STACK\#(breg,\ b;\ stack)}\rangle \\
\quad \forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\quad\quad \rightarrow\quad \mathrm{p'[n]} = \mathrm{C[p[n]]} \\
\quad\quad\quad \wedge\ \langle \mathrm{STACK\#(e[n],\ ce;\ estack)}\rangle \\
\quad\quad\quad\quad \forall\ \mathrm{n_0}.\quad \mathrm{n_0} \in \mathrm{estack} \\
\quad\quad\quad\quad\quad \rightarrow\quad \mathrm{ce[n_0]} = \mathrm{ce'[n_0]} \wedge \mathrm{cutpt[n_0]} = \mathrm{cutpt'[n_0]} \\
\quad\quad\quad\quad\quad\quad \wedge\ \langle \mathrm{UNLOADREC\#(cp[n_0],\ db_9,} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathrm{ce[n_0]} \neq \bot;\ \mathrm{goalreg_0})\rangle \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathrm{reng'(goalreg_0,\ ce[n_0],\ vi)} = \mathrm{cg[n_0]} \\
\quad\quad\quad\quad\quad\quad\quad \wedge\ \mathrm{nonvargoal(goalreg_0)} \\
\quad\quad\quad \wedge\ \langle \mathrm{UNLOADREC\#(cp[n] - 1,\ db_9,\ e[n]} \neq \bot;\ \mathrm{goalreg_0})\rangle \\
\quad\quad\quad\quad \mathrm{reng'(goalreg_0,e[n],vi)} = \mathrm{goal[n]} \wedge \mathrm{nonvargoal(goalreg_0)} \\
\wedge\ \mathrm{eqcode(db_7,\ db_9,\ C)} \\
\wedge\ \mathrm{eqpdt(procdeftab_7,\ procdeftab_9,\ C))}
\end{array}
$$

Die Invariante, und damit die Anzahl der nachzuweisenden Teilformeln, wäre ohne die verwendete Technik der iterativen Verfeinerung etwa doppelt so groß, wie die folgenden Invariante $MINV_8$ für ASM8 zeigt:

$$
\begin{array}{l}
\mathrm{MINV_8} \equiv \\
\quad \mathrm{stop} = \mathrm{run} \\
\rightarrow\quad (\mathrm{preg} \neq \mathrm{start} \rightarrow \mathrm{goalreg} \neq []) \wedge \mathrm{ce}[\bot] = \bot \wedge \mathrm{cutpt}[\bot] = \bot \\
\quad \wedge\ (\quad \mathrm{is\_retry\_me(code(preg,\ db_8))} \vee \mathrm{is\_retry(code(preg,\ db_8))} \\
\quad\quad \vee\ \mathrm{is\_trust\_me(code(preg,\ db_8))} \vee \mathrm{is\_trust(code(preg,\ db_8))} \\
\quad\quad \rightarrow \mathrm{breg} \neq \bot \wedge \mathrm{preg} = \mathrm{p[breg]}) \\
\quad \wedge\ (\quad \mathrm{preg} \neq \mathrm{start} \\
\quad\quad \wedge \neg\ \mathrm{is\_retry\_me(code(preg,\ db_8))} \wedge \neg\ \mathrm{is\_retry(code(preg,\ db_8))} \\
\quad\quad \wedge \neg\ \mathrm{is\_trust\_me(code(preg,\ db_8))} \wedge \neg\ \mathrm{is\_trust(code(preg,\ db_8))} \\
\quad\quad \rightarrow \langle \mathrm{S\text{-}CHAIN\text{-}REC\#(subreg}\ \hat{}_t\ \mathrm{car(goalreg),\ preg,\ db_8;\ col)}\rangle \\
\quad\quad\quad \mathrm{mapcode(col,\ db_8)} <_{clvi} 0) \\
\quad \wedge\ \langle \mathrm{STACK\#(breg,\ b;\ stack)}\rangle \\
\quad\quad \langle \mathrm{b\text{-}list\#(ereg,\ ce;\ estack)}\rangle \\
\quad\quad\quad (\quad (\quad \mathrm{preg} = \mathrm{start} \\
\quad\quad\quad\quad \vee \neg\ \mathrm{is\_retry\_me(code(preg,\ db_8))} \\
\quad\quad\quad\quad \rightarrow \mathrm{cutpt[ereg]} \in \mathrm{stack} \vee \mathrm{cutpt[ereg]} = \bot) \\
\quad\quad\quad \wedge\ \mathrm{ordered(stack)} \wedge \mathrm{ordered(estack)} \wedge \mathrm{disjoint(stack,\ estack)} \\
\quad\quad\quad \wedge\ (\forall\ \mathrm{n}.\quad \mathrm{n} \in \mathrm{stack} \\
\quad\quad\quad\quad\quad \rightarrow\quad \mathrm{e[n]} \ll \mathrm{n} \wedge \mathrm{goal[n]} \neq [] \\
\quad\quad\quad\quad\quad\quad \wedge\ \langle \mathrm{STACK\#(e[n],\ ce;\ estack)}\rangle \\
\quad\quad\quad\quad\quad\quad\quad (\quad \mathrm{disjoint(estack,\ stack\ from\ n)} \\
\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ \mathrm{ordered(estack))} \\
\quad\quad\quad\quad\quad\quad \wedge\ \langle \mathrm{S\text{-}CHAIN\text{-}RET\#(sub[n]}\ \hat{}_t\ \mathrm{car(goal[n]),} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathrm{p[n],\ db_8;\ col)}\rangle \\
\quad\quad\quad\quad\quad\quad\quad \mathrm{mapcode(col,\ db_8)} <_{clvi} 0))
\end{array}
$$

$MINV_8$ codiert die schon in der Verfeinerung 7/8 bewiesenen Eigenschaften, wie Disjunktheit des Umgebungs- und des Choicepoint-Stacks. Diese Eigenschaf-

ten konnten für die Verfeinerung 8/9 vorausgesetzt werden, und mußten nicht erneut bewiesen werden.

Kapitel 18

9/10: Kompilation von Termen

Dieser Abschnitt beschreibt unsere bisherige Arbeit an der ersten Verfeinerung aus Kapitel 4 in [BR95]. Diese Verfeinerung ist neben 5/7 wohl die komplizierteste Verfeinerung. Sie konnte zwar noch nicht vollständig bewiesen werden, dennoch haben die Formalisierung und erste Beweisversuche eine ganze Reihe von Problemen aufgedeckt. Diese bestanden zum einen in eigenen Unklarheiten und Mißverständnissen zu verschiedenen Aspekten der Verfeinerung, zum anderen in der Formalisierung der in [BR95] nur noch sehr informell gehaltenen Korrektheitsaussagen. Wir verzichten deshalb auch auf eine umfassende Darstellung der Verfeinerung, und skizzieren nur einige in der Verfeinerung steckenden Probleme sowie die bisher gefundenen Lösungsansätze.

Der wesentliche Aspekt der Verfeinerung 9/10 ist die Repräsentation von Termen durch verzeigerte Geflechte auf einem neu eingeführten Heap, und die Übersetzung von Literalen in Anweisungen, die derartige Geflechte aufbauen und unifizieren. Dies ist aber leider nicht die einzige Modifikation an der bisherigen ASM9. Eine ganze Reihe weiterer Aspekte spielt ebenfalls eine Rolle:

- Die Implementierung von ASM10 enthält keinen Occur-Check. Wie aber kann die Bedingung „unter der Voraussetzung, daß ASM9 keinen Occur Check aufruft“ formalisiert werden?

- Statt Substitutionen zu speichern, speichert ASM10 nun auf einem weiteren Stack, dem sog. *trail*, Variablenbindungen. Wenn beim Backtracking eine alte Substitution benötigt wird, werden die zwischenzeitlich aufgebauten Variablenbindungen destruktiv wieder rückgängig gemacht.

- Der Umgebungs- und Choicepointstack von ASM10 ist „flach“. Er hat keine eigene Struktur mehr wie der bisherige. Die verschiedenen Komponenten sind jetzt in aufeinanderfolgenden Adressen gespeichert, und werden uniform über eine Funktion *val* zugegriffen.

- ASM10 in [BR95] behandelt keinen Cut. Dieser wird erst am Ende des Kapitels 4 wieder eingeführt.

- Variablenumbenennung wird statt mit Hilfe eines Umbenennungsindex durch das Anlegen einer Variable an einer neuen Speicheradresse bewerkstelligt. Durch die *allocate*-Anweisung wird suggeriert, daß diese Adresse statt einer *global neuen* Heapadresse eine *lokal neue* Adresse des Umgebungsstacks sein darf. Es zeigt sich aber, daß diese Annahme falsch ist (was nicht bedeutet, daß die in [BR95] gegebene ASM inkorrekt ist, s.u.). Damit stellt sich das Problem, wie die korrekte Abbildung von global umbenannten Variablen aus ASM9 auf Speicherzellen von ASM10 ausieht.

- Es zeigt sich, daß die gespeicherten Substitutionen von ASM9 *nicht* zu denen aus ASM10 korrespondieren. Vielmehr werden bestimmte Variablenbindungen, die keine Rolle mehr spielen, vorzeitig weggeworfen.

Nur die ersten vier der oben genannten Aspekte werden in [BR95] explizit behandelt. Um die Probleme zu reduzieren, haben wir versucht, all die Aspekte, die nicht an die Ersetzung von Termen durch Geflechte gebunden sind, aus der Verfeinerung zu entfernen. Dazu haben wir zum ersten die Strukturierung des Umgebungsstacks beibehalten. Die in einer Umgebung gespeicherten Variablen werden in unserer Modellierung mit einer Funktion $x : env \times nat \to node$ referenziert: $x(ereg,m)$ liefert die m-te Variable der aktuellen Umgebung (die Sorte *node* ist nun einfach die Sorte der Speicheradressen; sie umfaßt *env*). Die in der WAM vorgesehene Aufteilung des Hauptspeichers, in der heap-Adressen an niedereren Adressen als die Stack-Adressen liegen, ergibt eine recht komplexe Ordnungsstruktur $\ll$ auf den Speicherzellen, für die die Axiome

$$\perp + \, m_1 \ll x(\perp + \, m_2, m_3)$$

$$x(\perp + \, m_0, m_1) \ll x(\perp + \, m_2, m_3) \leftrightarrow m_0 < m_2 \vee m_0 \ll m_2 \wedge m_1 < m_3$$

gelten. Die Funktion *val: heap* $\to$ *termrep* dient nun nur noch zur Bestimmung von Speicherinhalten des heaps.

Zum zweiten ermöglicht uns die Beibehaltung der Strukturierung der Stacks auch die Beibehaltung des Cuts (eine Anweisung zum Entfernen von auf dem trail abgelegten Variablenbindungen ist natürlich erforderlich; ansonsten behalten wir einfach die alten Register der vorigen ASM) .

Zum dritten haben wir den Occur-Check der Unifikation beibehalten. Das „Meta-Theorem“, welches besagt, daß wenn die Occur-Check-Routine nicht aufgerufen wird, sie auch weggelassen werden kann, gilt ja trivialerweise auch für ASM10. Außerdem hat es uns die Beibehaltung des Occur-Check erlaubt, die sowohl in [AK91], S. 14 als auch in [BR95], S. 39 gemachte Aussage, es genüge, den Occur-Check in die *bind*-Routine zu integrieren, zu falsifizieren: ein Occur-Check ist auch in unify_value erforderlich.

Zum vierten haben wir versucht, die Änderung der Umbenennungsstrategie schon auf Termebene einzuführen. Die Idee war, daß das Umbenennen einer Variable X mit dem aktuellen Umbenennungsindex durch die Belegung einer

neuen Stackadresse $x(ereg,m)$ ersetzt werden kann. Dieser Übergang von global neuen Variablen zu neuen Variablen relativ zum gerade aktiven Stack in der *allocate*-Anweisung von ASM10 in [BR95] suggeriert, in der die neuen Variablen genau so allokiert werden. Wir haben also eine Variante ASM9a von ASM9 definiert, die statt des globalen Umbenennungsindex neue Stack-Locations $x(ereg,n)$ zum Umbenennen der Variablen verwendet. Nach einigen Verifikationsversuchen stellten wir jedoch beim Versuch, die Äquivalenz der *deallocate*-Regeln nachzuweisen fest, daß dies nicht möglich war, weil sich die in dieser Regel deallokierten Variablen noch innerhalb von bis dahin berechneten Substitutionen befinden können. Bei erneuter Allokation einer Umgebung würden die Variablen überschrieben, und somit das Ergebnis inkorrekt. Eine genaue Analyse zeigte nun, daß in ASM10 eine neue Variable X zwar zunächst im Stack allokiert wird, sie aber immer dann, wenn sie innerhalb eines Terms T vorkommt ($X \in vars(T)$), der an eine Variable gebunden wird, aus dem Stack in den heap verschoben wird (durch die Instruktionen *unify_variable* und *unify_local_value*). Somit wird eine Klauselvariable von der WAM evtl. *mehrmals* umbenannt.

Eine gewisse Hilfe zum Verständnis der Vorgänge ergibt sich aus [AK91]. In der dort dargestellten ersten Variante der WAM werden Variablen nicht in der *allocate*-Anweisung auf dem Stack initialisiert, sondern bei Ihrem ersten Vorkommen auf den Heap gelegt. Allerdings gibt es auch dort die Ausnahme, bei der eine Variable bei Ihrem ersten Vorkommen sofort an einen Term gebunden wird (in der Instruktion *get_variable*, die etwa für eine Variable X im Klauselkopf `p(X)` generiert wird). In diesem Fall ist leicht zu sehen, daß die Variable auf dem Stack allokiert werden darf, da sie keine Rolle für die weitere Berechnung spielen wird. Sowohl die in [AK91], als auch die in [BR95] im weiteren Verlauf gezeigten Optimierungen (insbes. die „Last Call Optimization“ LCO) sind stark mit der Frage verknüpft, unter welchen Umständen Variablen im Stack statt im Heap allokiert werden können. Diese Fragestellung muß unserer derzeitigen Ansicht nach nicht mit der der Verfeinerung 9/10 verknüpft werden. Es scheint uns einfacher zu sein, in einem separaten Verfeinerungsschritt nur genau einmal Variablen vom Stack in den Heap zu schieben, und damit die zulässigen Belegungen für Stack und Heap (Stack und Heap Variables Constraint) zu ändern.

Dies scheint auch besonders deshalb wünschenswert, weil die Haupttheoreme der Verfeinerung, das Getting und das Putting Lemma von der genauen Definition der Constraints abhängen. Der Heap und Stack Variables Constraint lassen sich nämlich nicht, wie in [BR95] dargestellt, nachträglich als invariant in den Getting- und Putting-Anweisungen nachweisen, vielmehr hat sich gezeigt, daß sehr präzise Definitionen der beiden Constraints *notwendige Voraussetzungen* für die Gültigkeit des Getting und Putting Lemmas darstellen. Letztendlich bilden die beiden Constraints einen wesentlichen Bestandteil der Zusammenhangsinvariante für die Verfeinerung 9/10.

Jede Modifikation an der Definition der beiden Constraints (also insbesondere jede Modifikation an der Allokation von Variablen auf dem Stack oder Heap) bedeutet, daß erneut die Invarianz durch alle Putting und Getting-Anweisungen nachgewiesen werden muß. Deshalb haben wir derzeit für die Verfeinerung 9/10

die erste Definition von [AK91] übernommen. Diese Lösung enthält gegenüber [BR95] eine ineffizientere *put_variable*-Anweisung (die die Variable im Heap allokiert), keine *unify_local_value*-Anweisung und statt der Initialisierung von Variablen in *allocate* deren Initialisierung beim ersten Vorkommen, wie dies in [BR95] erst später (S. 58f) eingeführt wird.

Die Version erlaubt einen sehr einfachen Stack- und Heap-Variablen-Constraint, der besagt, daß die Repräsentation von Termen komplett auf dem Heap liegen muß, mit Ausnahme der führenden in einem (Stack- oder globalen) Register gespeicherten Zelle. Diese darf keine Variable (= Ref-Zelle auf sich selbst) sein. Ordnungsbeziehungen zwischen Adressen spielen noch keine Rolle. Wir nehmen im Moment an, daß diese Version der WAM es auch erlaubt, eine bijektive Abbildung (dynamische Funktion) zu definieren, die die mit *vireg* global umbenannten Variablen auf neue Heap-Variablen abbildet. Die Funktion sollte nur genau jeweils beim Abarbeiten der Instruktionen erweitert werden müssen, die dem ersten Vorkommen einer Variable entsprechen (spätere Modifikation sollten nicht notwendig sein).

Gedacht ist dann daran, das Shifting von Variablen aus dem Heap in den Stack (und die damit verbundene Definition von stärkeren Constraints, die Einführung von temporären/permanenten Variablen, sowie die Definition neuer Instruktionen wie put_unsafe_value etc.) in *einem* separaten Verfeinerungsschritt einzuführen.

Nicht vermeiden läßt sich auch bei der in [AK91] definierten ASM10 der Umstand, daß sie weniger Variablenbindungen speichert als ASM9. Unsere momentane Annahme ist, daß das (implizite) Wegwerfen von Variablenbindungen in ASM10, das bei der Deallokation einer Umgebung *ereg* stattfindet, genau einem expliziten Entfernen aller Variablenbindungen an mit *vi*[*ereg*] umbenannte Variablen aus *subreg* in ASM9 entspricht. Gemäß unserer Philosophie, die Verfeinerung 9/10 von soviel Ballast wie möglich zu befreien, haben wir deshalb eine Funktion *remove(subreg,vi*[*ereg*]*)* definiert und separat verifiziert, daß eine Modifikation der *deallocate*-Regel von ASM9 zu

deallocate rule
if code(preg,db_9) = deallocate
then cpreg := cp[ereg]
ereg := ce[ereg]
preg := preg +1
subreg := remove(subreg,vi[ereg])

keine signifikante Auswirkung auf das Ergebnis der Berechnung hat: wenn die Berechnung terminiert, haben beide berechneten Substitutionen dieselbe Wirkung auf die Anfrage. Dies konnte in 2 Wochen mit 3 Iterationen verifiziert werden. Die Zusammenhanginvariante INV_{99a} und die Maschineninvariante $MINV_9$ sind

$INV_{99_a} \equiv$
$\text{stop} = \text{stop}' \wedge \text{breg} = \text{breg}' \wedge \text{ctreg} = \text{ctreg}' \wedge \text{cpreg} = \text{cpreg}'$
$\wedge\ \text{ereg} = \text{ereg}' \wedge \text{preg} = \text{preg}' \wedge \text{vireg} = \text{vireg}' \wedge \text{cp} = \text{cp}'$

$\wedge$ p = p$'$ $\wedge$ b = b$'$ $\wedge$ e = e$'$ $\wedge$ ce = ce$'$ $\wedge$ ct = ct$'$ $\wedge$ vi = vi$'$
$\wedge$ cutpt = cutpt$'$ $\wedge$ cutpt[$\bot$] = $\bot$ $\wedge$ ce[$\bot$] = $\bot$
$\wedge$ subreg $<_{svi}$ vireg $\wedge$ subreg$'$ $<_{svi}$ vireg
$\wedge$ ($\forall$ lit. lit $<_{tvi}$ 0 $\rightarrow$ subreg $\hat{}_t$ lit = subreg$'$ $\hat{}_t$ lit)
$\wedge$ $\langle$STACK#(ereg, ce; estack)$\rangle$
(slnodups(estack) $\wedge$ nlnodups(vilist(vi, estack))
$\wedge$ ($\neg$ is_ret(code(preg, db$_9$)) $\wedge$ stop = run
$\rightarrow$ vilist(vi, estack) $<_{nl}$ vireg)
$\wedge$ ($\forall$ n, lit. lit $<_{tvi}$ 0 $\wedge$ n $\in$ estack
$\rightarrow$ subreg $\hat{}_t$ rent(lit, vi[n]) = subreg$'$ $\hat{}_t$ rent(lit, vi[n])))
$\wedge$ $\langle$STACK#(breg, b; stack)$\rangle$
$\forall$ n, lit. lit $<_{tvi}$ 0 $\wedge$ n $\in$ stack
$\rightarrow$ sub[n] $\hat{}_t$ rent$'$(lit, e[n], vi) = sub$'$[n] $\hat{}_t$ rent$'$(lit, e[n], vi)
$\wedge$ sub[n] $\hat{}_t$ lit = sub$'$[n] $\hat{}_t$ lit
$\wedge$ sub[n] $<_{svi}$ vireg $\wedge$ sub$'$[n] $<_{svi}$ vireg
$\wedge$ $\langle$STACK#(e[n], ce; estack$_0$)$\rangle$
(vilist(vi, estack$_0$) $<_{nl}$ vireg
$\wedge$ slnodups(estack$_0$) $\wedge$ nlnodups(vilist(vi, estack$_0$))
$\wedge$ ($\forall$ n$_0$. n$_0$ $\in$ estack$_0$
$\rightarrow$ sub[n] $\hat{}_t$ rent(lit, vi[n$_0$])
= sub$'$[n] $\hat{}_t$ rent(lit, vi[n$_0$])))

MINV$_9$ $\equiv$
stop = run
$\rightarrow$ $\neg$ is_unify(code(preg, db$_9$)) $\wedge$ code(preg, db$_9$) $\neq$ proceed
$\wedge$ (is_try(code(preg, db$_9$)) $\vee$ is_try_me(code(preg, db$_9$))
$\vee$ code(preg, db$_9$) = allocate $\vee$ is_sw_const(code(preg, db$_9$))
$\vee$ is_sw_term(code(preg, db$_9$)) $\vee$ is_sw_struct(code(preg, db$_9$))
$\vee$ code(preg, db$_9$) = allocate
$\supset$ $\langle$C-CHAIN-REC#(subreg $\hat{}_t$ rent$'$(calllit(code(cpreg$-$1), db$_9$)),
ereg, vi), preg, db$_9$; cli)$\rangle$ cli $<_{clvi}$ 0
$\wedge$ $\langle$UNLOADREC#(cpreg $-$ 1, db$_9$, ereg $\neq$ $\bot$; goalreg)$\rangle$
(goalreg $\neq$ [] $\wedge$ nonvargoal(goalreg)) ;
is_ret(code(preg, db$_9$)) $\supset$ breg $\neq$ $\bot$ $\wedge$ preg = p[breg] ;
$\langle$UNLOADREC#(preg, db$_9$, ereg $\neq$ $\bot$; goalreg)$\rangle$
nonvargoal(goalreg))
$\wedge$ $\langle$STACK#(breg, b; stack)$\rangle$
$\langle$STACK#(is_ret(code(preg, db$_9$)) $\supset$ e[breg] ; ereg, ce; estack)$\rangle$
(ordered(estack) $\wedge$ ordered(stack) $\wedge$ disjoint(estack, stack)
$\wedge$ ($\neg$ is_ret(code(preg, db$_9$))
$\rightarrow$ cutpt[ereg] $\in$ stack $\vee$ cutpt[ereg] = $\bot$)
$\wedge$ ($\forall$ n. n $\in$ estack
$\rightarrow$ $\langle$UNLOADREC#(cp[n], db$_9$, ce[n] $\neq$ $\bot$; goalreg)$\rangle$
nonvargoal(goalreg))
$\wedge$ ($\forall$ n. n $\in$ stack
$\rightarrow$ e[n] $\ll$ n

$$
\begin{array}{l}
\wedge\ \langle \mathrm{STACK\#(e[n],\ ce;\ estack_0)}\rangle \\
\qquad (\quad \mathrm{disjoint(estack_0,\ stack\ from\ n)} \\
\qquad\ \wedge\ \mathrm{ordered(estack_0)} \\
\qquad\ \wedge\ (\forall\ \mathrm{n_0}.\quad \mathrm{n_0} \in \mathrm{estack_0} \\
\qquad\qquad\qquad \rightarrow \langle \mathrm{UNLOADREC\#(cp[n_0],\ db_9,} \\
\qquad\qquad\qquad\qquad\qquad \mathrm{ce[n_0]} \neq \bot;\ \mathrm{goalreg})\rangle \\
\qquad\qquad\qquad\qquad\quad \mathrm{nonvargoal(goalreg)}))) \\
\wedge\ \langle \mathrm{UNLOADREC\#(cp[n]-1,db_9,\ e[n]} \neq \bot;\ \mathrm{goalreg})\rangle \\
\qquad (\mathrm{goalreg} \neq [] \wedge \mathrm{nonvargoal(goalreg)}) \\
\wedge\ \langle \mathrm{C\text{-}CHAIN\text{-}RET\#(} \\
\qquad\quad \mathrm{sub[n]}\ \hat{}_t\ \mathrm{rent'(calllit(code(cp[n]-1,\ db_9)),} \\
\qquad\quad \mathrm{e[n],\ vi),\ p[n],\ db_9;\ cli)}\rangle \\
\qquad \mathrm{cli} <_{clvi} 0))
\end{array}
$$

Kapitel 19

Statistiken

Die folgende Tabelle gibt einen Überblick über den Verifikationsaufwand für die Prolog-WAM-Fallstudie. Für jede Verfeinerung sind die Zahl der notwendigen Beweisschritte, die Zahl der dabei notwendigen Interaktionen, und die Zahl der Theoreme aufgeführt. Diese Zahlen sind jeweils aus der aktuell vorliegenden KIV-Version 4 entnommen. Die Anzahl der notwendigen Iterationen, um eine korrekte Invariante zu erhalten, und die erforderliche Zeit, um die Verifikation schließlich erfolgreich zu führen, beziehen sich dagegen zum Teil auf ältere Systemversionen (für 1/2 und 4/5 die KIV-Version 1, für 2/3,3/4,5/6 und 5/7 KIV-Version 3).

	1/2	2/3	3/4	4/5	5/6
Beweisschitte	1074	1760	2546	1722	5341
Interaktionen	161	124	300	87	672
Theoreme	15	13	22	17	42
Iterationen	12	8	5	9	8
Verif. zeit	2 Mo.	2 Wo.	1 Wo.	1 Mo.	2 Wo.
Größe von INV	20	25	25	14	53

	5/7	7/8	8/9	9/9a
Beweisschitte	7558	3445	4295	3045
Interaktionen	1383	336	377	426
Theoreme	39	21	19	19
Iterationen	17	12	8	4
Verif. zeit	2 Mo.	1 Mo.	3 Wo.	2 Wo.
Größe von INV	36	36	23+17	18+23

Der Verifikationsaufwand insgesamt beträgt derzeit insgesamt ca. 9 Personenmonate, inclusive der Verifikation von 1771 prädikatenlogischer Hilfsbehauptungen, die 17458 Beweisschritte und 3393 Interaktionen erforderten. Einige weitere statistische Daten:

- Der Umfang der prädikatenlogischen Lemmata hat sich seit der Verifikation der Verfeinerung 5/7 fast vervierfacht. Die wesentlichen Ursachen sind zum einen die große Zahl der ab der Verfeinerung 8/9 benötigten Lemmata über Unifikation, Umbenennung und Substitution. Einige dieser Lemmata erforderten im Gegensatz zu den anderen Beweisen (meist 0–2 Interaktionen) wegen der komplizierten Terminierungsordnung von Unifikation auch aufwendigere Beweise (bis zu 20 Interaktionen). Ein zweiter Grund ist, daß für ASM10 eine große Zahl einfacher Lemmata zur Repräsentation von Termen benötigt wird, die bereits beweisen wurden.

- Gegenüber den in [SA97] genannten Zahlen, die sich auf KIV-Version 3 bezogen, haben sich Verbesserungen ergeben. Am markantesten ist die Reduktion der Invariantengröße für 5/7 von 97 auf 36 Zeilen, durch eine Modifikation der Beweistechnik (s. Abschnitt 15.2). Bei den Verfeinerungen 2/3 und 3/4 wurde nun die Beweisverpflichtungsgenerierung eingesetzt. In [SA97] wurden die jetzt generischen Beweise für das allgemeine Modularisierungstheorem noch für die einzelnen Instanzen geführt (de facto führten die Beweise für die Einzelinstanzen zur jetzt vorliegenden, allgemeinen Theorie).

- Die Verbesserungen bei der Beweisunterstützung und -automatisierung im KIV-System (ohne die methodischen Fortschritte durch das Modularisierungstheorem) im Verlauf dieser Fallstudie können am deutlichsten an der Verifikation der ersten Verfeinerung gezeigt werden, da hier nur 1:1-Diagramme vorkommen: Erforderte die Verifikation von 1/2 in KIV-Version 1 noch 378 Interaktionen, so reduzierte sich die Zahl der Interaktionen in KIV-Version 3 auf 246. In KIV-Version 4 beträgt sie nur noch 161.

- Auch für die notwendige Einarbeitungszeit in KIV gibt die Verfeinerung 1/2 ein gutes Maß. Die Verifikation in KIV-Version 3, wurde nämlich von Harald Vogt, einem Studenten, der vorher ein einsemestriges Praktikum zu KIV besucht hatte und vorher keine Vorkenntnisse zur Fallstudie hatte, in 80 Stunden Arbeit von KIV-Version 1 nach 3 portiert (die Portierung von Version 3 nach 4 erforderte dann nur noch ca. 1 Tag Arbeit)

- Die Größe der Interpreter beginnt mit 120 Zeilen imperativen (Pseudo-PASCAL-)codes und erreicht 300 Zeilen für ASM9. ASM10 (im wesentlichen identisch zur WAM) ist wegen der Vielzahl neuer Instruktionen mit 950 Zeilen Code deutlich größer.

Kapitel 20

Verwandte Fallstudien

Die Literatur kennt eine riesige Zahl von Arbeiten, die sich auf Papier mit der Korrektheit von Compilern beschäftigen. Einen Überblick gibt z.B. [Joy90]. Umfangreichere Arbeiten wurden z.B. im VLISP-Projekt ([GRW95]) und im PROCOS-Projekt ([BLH93]) durchgeführt.

Die weitaus meisten Arbeiten beschäftigt sich, genau wie diese, mit der Korrektheit der Übersetzung („compiling correctness"). Die effizienten Implementierung von Compilern („compiler correctness") wurde bisher noch weniger untersucht, ist aber ein Forschungsgegenstand des gerade laufenden Verifix-Projekts ([GDG+96]).

Wesentlich weniger zahlreich sind Arbeiten, die sich mit der systemunterstützten, formalen Verifikation beschäftigen. Die wohl umfangreichste Arbeit in diesem Gebiet ist die mit NQTHM durchgeführte formale Verifikation eines Compilers, der die imperativen Sprache Gypsy zunächst in Assembler Code und dann in Maschinencode des Prozessors FM8502 übersetzt ([Moo88], [You88]).

Die Verifikation der Übersetzung von Prolog in die WAM war neben dem unserer Arbeit zugrundeliegenden Papier [BR95] auch das Thema der Arbeit [Rus92]. Diese Arbeit macht allerdings einige Vereinfachungen (insbes. werden weder Cuts noch Switching behandelt), und enthält keine Strukturierung des Beweises in verschiedene Verfeinerungen. Ein Versuch, die Arbeit zu formalisieren, scheiterte an der zu hohen Komplexität des Beweises. Deshalb versuchte V. Austel in [Aus98] ein strukturieren Korrektheitsbeweis im HOL-System ([Gor88]). Dieser verfeinert im Gegensatz zu [BR95] zuerst die Termrepräsentation und erst dann die Kontrollstruktur Die Arbeit ist u.E. kaum verständlich. Sie erforderte über einen Jahr Arbeitsaufwand, und zu ihrer Fertigstellung wäre laut Aussage des Autors mindestens ein weiteres Jahr notwendig.

Interessant an der Arbeit ist die These, daß ein wesentliches Problem, das in [BR95] nur unzureichend behandelt ist, in der Einführung der Termrepräsentation in einer einzigen Verfeinerung 9/10 liegt. Nun zeigen unsere Überlegungen in Abschnitt 18, daß die Einführung der Datenrepräsentation in einem Schritt in der Tat in mehrere Schritte aufgeteilt werden muß, um die verschiedenen eingeführten Konzepte sauber zu verifizieren. Wir denken aber, daß die derzeit

vorgeschlagene Aufteilung in mehrere Schritte dieses leisten wird, und sehen keine prinzipiellen Probleme.

Eine weitere parallel zu dieser Arbeit entstandene formale Behandlung der Übersetzung von Prolog in die WAM ist die Arbeit von C. Pusch ([Pus96]) mit dem Isabelle-System ([Pau94]). Die Spezifikation basiert auf induktiv definierte Relationen über den Vektor der Zustandsvariablen. Durch die Verwendung von Polymorphie und Pattern Matching ist die Notation in Isabelle deutlich kompakter (aber für einen ungeübten Leser auch kryptischer) als die ASM-Notation (und erst recht als unsere PASCAL-artige Notation in der DL-Übersetzung).

Der Startpunkt der Arbeit basiert auf einer Interpreterdefinition, die bereits Stacks von Choicepoints, nicht Suchbäume verwendet. Die Stacks werden im Gegensatz zu unserer verzeigerten Strukur als Listen modelliert. Das Aufsammeln von Stack-Knoten mit der Prozedur STACK# entfällt dadurch. Dies bewirkt eine geringfügige Vereinfachung der Beweise, die allerdings dadurch erkauft wird, daß die Einführung einer verzeigerten Stack-Struktur, die spätestens in ASM8 notwendig wird, wenn der Choicepoint- und der Environment-Stack überlagert werden, zusätzlich verifiziert werden müßte.

Es werden vier Verfeinerungen verifiziert: In der ersten Verfeinerung werden Cutpoints, die bis dahin als Teillisten des aktuellen Stacks repräsentiert waren, durch Zeiger (i.e. Positionen im Stack) ersetzt. Die zweite zeigt, daß statt alle Klauseln als Kandidatenklauseln auszuwählen, auch eine *procdef*-Funktion verwendet werden kann, die nur die Klauseln mit gleichem führenden Prädikatsymbol auswählt. Die so entstehende ASM ist modulo Notation zu unserer ASM2 äquivalent, und die letzten beiden verifizierten Verfeinerungsschritte 2/3 und 3/4 zu unseren identisch (lediglich die Konstrukte *true* und *fail* werden nicht behandelt, weswegen das in Abschnitt 14.2 geschilderte Problem im Regeltest der *fail*-Regel nicht gefunden werden konnte).

Der Verifikationsaufwand für die vier Verfeinerungen wird in [Pus96] als 6 Personenmonate und 3500 Interaktionen angegeben. Der größere Teil entfiel dabei, wie aus den Beweisskripten ersichtlich ist, auf die Verifikation der Verfeinerungen 2/3 und 3/4. Diese Zahlen liegen mehr als doppelt so hoch als die von uns erzielten. Die Ursachen liegen wohl im wesentlichen an 2 Problemen: Zum einen wurde keine Beweistechnik für die in 2/3 und 3/4 auftretenden *m:n*-Diagramme entwickelt. Die Diagramme wurden vielmehr in *1:n*-Diagramme zerlegt, wie dies in Abschnitt 6.2.3, S. 41 skizziert wurde. Dadurch ergab sich eine starke Aufblähung der Invarianten.

Zum anderen wurden zwei getrennte, asymmetrische Beweise für Korrektheit und Vollständigkeit geführt. Die Asymmetrie liegt wohl zum einen an der Verwendung von Abstraktionsfunktionen, die eine zusätzliche Definition ihres Domains (config_ok) erfordern (aber asymmetrisch dazu, keine Definition des Codomains). Zum anderen ist der Determinismus des Zustandsübergangssystems entscheidend dafür, daß nur ein Beweis geführt werden muß. In der Codierung von ASMs im Kalkül der Dynamischen Logik wird Determinismus schon syntaktisch durch das Axiom

$$\langle\alpha\rangle\ \varphi \equiv [\alpha]\ \varphi$$

für deterministische Programme α unterstützt (dieses Axiom wird im Korrektheitsbeweis des Theorems verwendet). In der Formalisierung durch eine induktive Relation muß der Determinismus (also die Rechtseindeutigkeit der induktiv definierten Relation) aber erst induktiv hergeleitet werden.

Kapitel 21

Zusammenfassung Teil II

Im zweiten Teil dieser Arbeit haben wir die im ersten Teil entwickelten Theorie an einer Fallstudie auf ihre Praxistauglichkeit hin untersucht. Die Fallstudie stammt aus dem Gebiet der Compilerverifikation, und gehört mit 9 Monaten Aufwand für die reine Verifikation unter den formalen, systemunterstützten Fallstudien zu den sehr großen Fallstudien.

Inhalt der Fallstudie war die formale Verifikation von 8 der 12 in [BR95] gegebenen Verfeinerungen, durch die ein Prolog-Programm in Assemblercode der Warren Abstract Machine übersetzt wird. Die Fallstudie war relativ typisch für die Compilerverifikation, eine ganze Reihe von Problemen, die in dieser Fallstudie auftraten, wie z.B. die Einführung von Registern, Stacks, Umgebungen (Stack Frames) die Optimierung von Kontrollstrukturen (Switching), oder die Umsetzung von abstrakten Datenstrukturen in verkettete Zeigerstrukturen, sollten auch bei anderen Programmiersprachen eine wichtige Rolle spielen.

Die Fallstudie zeigte, daß der voll formale Nachweis für die Korrektheit einer Verfeinerung wegen der zahlreichen, impliziten Annahmen sehr viel aufwendiger war, als dies angesichts der schon sehr gründlichen mathematischen Analyse in [BR95] zu vermuten war. Der zusätzliche Aufwand zahlte sich aber in dem Sinn aus, als durch die Analyse eine Reihe von kleineren Fehlern, die in der mathematischen Analyse noch offenblieben, gefunden und beseitigt werden konnten.

Um die Verifikation in den Griff zu bekommen, war sowohl der volle Gebrauch der im ersten Teil entwickelten Theorie notwendig, als auch ein sehr leistungsfähiges Werkzeug zur Verifikation. Das eingesetzte KIV-System hat sich während der Arbeit and der Fallstudie in seiner Leistungsfähigkeit deutlich verbessert.

Schließlich zeigte auch der Vergleich mit zwei parallel zu dieser Arbeit durchgeführten Fallstudien mit anderen Systemen (HOL, Isabelle) zum selben Thema, daß die entwickelten Theorie und die Leistungsfähigkeit des KIV-Systems einen deutlich niedrigeren Aufwand ermöglichte.

Kapitel 22

Ausblick

Die im Rahmen dieser Arbeit durchgeführte Fallstudie zeigt noch nicht vollständig die Korrektheit der Übersetzung von Prolog-Code in Code der WAM. Es fehlen noch 4 Verfeinerungen bis echter WAM-Code vorliegt. Die ersten beiden sind nach unserer Analyse in Abschnitt 18 relativ aufwendig zu verifizieren, die anderen beiden (Environment Trimming und Aufgabe der Strukturierung von Environments und Choicepoints) sollten eher einfach sein. Der Verifikationsaufwand, um die Übersetzungskorrektheit der Codegenerierung vollständig zu verifizieren, sollte nach derzeitiger Schätzung noch ca. 2–3 Monate betragen.

Um aus der Fallstudie einen echten verifizierten Prolog-Compiler zu erhalten, müßte dann noch ein Compiler implementiert werden, der die Compilerannahmen erfüllt. Dies sollte für eine einfache Variante mit rekursiv definierten DL-Programmen unproblematisch sein, da die Compilerannahmen schon weitgehend algorithmisch sind (mit Ausnahme von Switching).

Noch interessanter als imperative Programme zur Implementierung zu verwenden, wäre es, die Ideen des Verifix-Projekts [GDG+96] aufzugreifen und als Implementierungssprache für den Compiler Prolog selbst zu nehmen. Damit erhielte man die Möglichkeit einen effizienten Compiler durch Kompilation mit sich selbst zu erhalten („Bootstrapping“).

Die Definition eines Prolog-Compilers in Prolog wäre also eine Liste von Klauseln für ein 2-stelliges Prädikat *compile*. Anfragen an das Prädikat hätten die Form $compile(t, X)$, wobei t ein als Term codiertes Prolog-Programm wäre, X die Ausgabevariable, deren Belegung am Ende die generierten WAM-Instruktionen als Prolog-Term codiert.

Um den Konnex zu Programmen und WAM-Instruktionslisten herzustellen („Reflektion“), wären zwei einfache Konversionsfunktionen *clauselist-to-term* und *term-to-instructionlist* notwendig. Diese sind hier recht einfach zu definieren, da Prolog eine untypisierte Sprache ist (die Sprache mit dem einfachsten Reflektionsprinzip, der Quote-Operation ist LISP, da hier Programme und Datenstrukturen identisch sind; für typisierte Sprachen stellt Reflektion ein härteres Problem dar). Anschließend könnte man den Prolog-Code db_{compile} des Prolog-Compilers dadurch verifizieren, daß gezeigt wird, daß seine Ausführung mit

ASM1 und einer Anfrage $compile(t, X)$ für jedes (als Term codiertes) Prologprogramm t als Ergebnis eine (als Term codierte) Liste von Instruktionen liefert, für die die Compilerannahmen ($CompAssum$) gelten. Formal:

$$\begin{aligned} & \text{t} = \text{clauselist-to-term(db)} \\ \wedge\ & \langle \text{ASM1}(\text{db}_{\text{compile}}, \text{compile(t,X); subst}) \rangle\ \text{subst} = [\text{X} \leftarrow \text{t}'] \\ \rightarrow\ & \text{CompAssum(db, term-to-instructionlist(t}')) \end{aligned}$$

Damit erhielte man einen Compiler dessen Korrektheit nur noch davon abhängt, daß ASM1 eine korrekte Semantikdefinition von Prolog darstellt, der (trivialen) Korrektheit der Reflektionsfunktionen sowie natürlich der Korrektheit des Verifikationswerkzeugs.

Für das bootstrapping des Compilers mit sich selbst, um einen mit WAM-Code implementierten Compiler zu erhalten, gäbe es dann drei Möglichkeiten: Entweder der WAM-Code könnte durch Einsetzen von db_{compile} für db in obigem Theorem und symbolisches Ausführen berechnet werden. Dies ist der Idealfall, da weiterhin nur die Korrektheit des Verifikationssystems relevant ist. Nach den Erfahrungen meines Kollegen Kurt Stenzel mit der Java-ASM ist dies aber sehr aufwendig, und könnte sich als undurchführbar herausstellen. Eine zweite Möglichkeit ist natürlich, das Bootstrapping mit einem (oder mehreren) gängigen Prolog-Compilern durchzuführen. Eine letzte Möglichkeit ist schließlich, die KIV-Codegenerierung zu nutzen, die es gestattet, aus den abstrakten Programmen von ASM1 ausführbaren LISP-Code herzustellen, und das Bootstrapping mit diesem Lisp-Code durchzuführen. Die letzten beiden Methoden sind aus theoretischer Sicht nicht genau so sicher, da sie die Korrektheit anderer Compiler verlangen (mindestens für das eine betrachtete Programm des Prolog-Compilers bzw. der ASM), dennoch sollte die Wahrscheinlichkeit für Fehler, wenn beide Methoden denselben Code ergeben, de facto gleich Null sein.

Anhang A

Verwendete Notationen

Dieser Abschnitt gibt die in der Arbeit verwendeten grundlegenden Notationen.

Für eine Menge S bezeichnet $\mathcal{P}(S)$ die Potenzmenge von S, $\mathcal{P}^{\omega}(S)$ die Menge aller endlichen Teilmengen von S. S^n ist die Menge aller n-Tupel über S ($n \geq 0$). Wir schreiben $x_1 \ldots x_n$ und $(x_1, \ldots, x_n)$ für n-Tupel. Wenn n aus dem Kontext ersichtlich oder beliebig ist, schreiben wir auch $\underline{x}$. S^* ist die Vereinigung aller Mengen S^n für $n \geq 0$. Sie enthält auch das leere Tupel (). S^+ ist S^* ohne das leere Tupel. $\hat{S}^n$ ist die Menge aller duplikatfreien n-Tupel $x_1 \ldots x_n \in S^n$ mit $x_i \neq x_j$ für alle $1 \leq i < j \leq n$. $\hat{S}^*$ ist die Vereinigung aller $\hat{S}^n$. Wir verwenden die Notation $M = \bigcup_{s \in S} M_s$, für eine (mit den Elementen aus S indizierte) Familie von Mengen M_s. Dabei wird stets angenommen, daß die Mengen M_s paarweise disjunkt sind. $M_{s_1 \ldots s_n}$ kürzt $M_{s_1} \times \ldots \times M_{s_n}$ ab und $\hat{M}_{s_1 \times \ldots \times s_n}$ steht für $M_{s_1} \times \ldots \times M_{s_n} \cap \hat{M}^n$. Für zwei Tupel $(x_1, \ldots, x_n) \in S^n$ und $(x'_1, \ldots, x'_m) \in S^m$ definieren wir die Konkatenation $(x_1, \ldots, x_n) : (x'_1, \ldots, x'_m)$ als $(x_1, \ldots, x_n, x'_1, \ldots, x'_m) \in S^{n+m}$. Wir identifizieren S mit S^1, so daß $x : (x_1, \ldots, x_n)$ identisch zu $(x, x_1, \ldots, x_n) \in S^{n+1}$ ist.

Ist eine Funktion $f : M \to N$ gegeben, so nehmen wir immer an, daß die homomorphe Erweiterung zu einer Funktion auf Tupeln aus M^n durch f$((x_1, \ldots, x_n)) := (f(x_1), \ldots, f(x_n))$ gegeben ist. Die homomorphe Erweiterung von f auf Teilmengen von M ist analog definiert.

Anhang B

Syntax und Semantik der Dynamischen Logik

B.1 Syntax der Dynamischen Logik

Definition 4 *Signatur*
Eine Signatur $SIG = (S, OP, X, P)$ besteht aus einer endlichen Menge von Sorten S, einer Familie $OP = \bigcup_{\underline{s} \in S^*, s' \in S} OP_{\underline{s},s'}$ von Operationen (mit Argumentsorten $\underline{s}$ und Zielsorte s'), einer Familie $X = \bigcup_{s \in S} X_s$ von jeweils abzählbar vielen Variablen, und einer Familie $P = \bigcup_{\underline{s} \in S^*, \underline{s}' \in S^*} P_{\underline{s},\underline{s}'}$ von Prozedurnamen mit value-Parametern der Sorten $\underline{s}$ und reference-Parametern der Sorten $\underline{s}'$ (Prozedurnamen werden in Programmen verwendet).

Es wird vorausgesetzt, daß S mindestens die Sorten *bool* und *nat*, sowie die üblichen Operationen auf diesen Sorten (*true*,*false*,$\wedge$,$\vee$,$\rightarrow$, $\leftrightarrow$,$\neg$, 0,+1,$-$1,+) enthält.

Definition 5 *DL expressions*
Für eine mehrsortige Signatur *SIG*, sind die Menge der Ausdrücke *DLEXPR* $= \bigcup_{s \in S} DLEXPR_s$, und die Menge PROG der Programme definiert als die kleinsten Mengen mit

- $X_s \subseteq DLEXPR_s$ für jedes $s \in S$
- Wenn $f \in OP_{\underline{s},s}$ und $\underline{t} \in DLEXPR_{\underline{s}}$ dann $f(\underline{t}) \in DLEXPR_s$
- Wenn $\varphi \in FMA$ und $\underline{x} \in \hat{X}_{\underline{s}}$ dann $\forall\, \underline{x}.\varphi \in FMA$
- Wenn $\varphi \in FMA$ und $\underline{x} \in \hat{X}_{\underline{s}}$ dann $\exists\, \underline{x}.\varphi \in FMA$
- Wenn $t, t' \in DLEXPR_s$, dann $t = t' \in FMA$
- Wenn $\varphi \in FMA$ und $t, t' \in DLEXPR_s$, dann $(\varphi \supset t; t') \in DLEXPR_s$
- Wenn $\underline{x} \in \hat{X}_s$ und $\underline{t} \in U_{\underline{s}}$, wobei $U_s = T_s \cup \{?\}$, dann $\underline{x} := \underline{t} \in PROG$

- Wenn $\alpha \in PROG$, $\underline{x} \in \hat{X}_s$ und $\underline{t} \in U_{\underline{s}}$, wobei $U_s = T_s \cup \{?\}$, dann **var** $\underline{x} = \underline{t}$ **in** $\alpha \in PROG$
- **skip**, **abort** $\in PROG$
- Wenn $\alpha, \beta \in PROG$, dann $\alpha; \beta \in PROG$
- Wenn $\alpha, \beta \in PROG$ und $\varepsilon \in BXP$ dann **if** ε **then** α **else** $\beta \in PROG$
- Wenn $\alpha \in PROG$ und $\varepsilon \in BXP$ dann **while** ε **do** $\alpha \in PROG$
- Wenn $\alpha \in PROG$ und $\kappa \in T_{nat}$ dann **loop** α **times** $\kappa \in PROG$
- Wenn $p \in P_{\underline{s},\underline{s}'}$, $\underline{t} \in T_{\underline{s}}$, $\underline{x} \in \hat{X}_{\underline{s}'}$ und $\kappa \in T_{\mathrm{nat}}$ dann $\mathrm{p}(\underline{t}; \underline{x}) \in PROG$ sowie **procbound** κ **in** $\mathrm{p}(\underline{t}; \underline{x}) \in PROG$. Das letzte Programm ist ein Aufruf von p mit durch κ beschränkter maximaler Rekursionstiefe

In der Definition kürzt *FMA* (Formeln) $DLEXPR_{\mathrm{bool}}$ ab, die Menge T_s (Terme der Sorte s) ist die Teilmenge von $DLEXPR_s$, die weder Quantoren noch Programme enthält. *BXP* (Boolesche Ausdrücke) ist T_{bool}.

Bemerkung 1 Wir verwenden wie in Pascal **begin** ... **end** zur Klammerung von Ausdrücken. **if** ε **then** α dient als Abkürzung für **if** ε **then** α **else skip**

Bemerkung 2 Die Tests in while-Schleifen und conditionals sind in der obigen Definition auf boole'sche Ausdrücke beschränkt ($\varepsilon \in BXP$). Dies ist für Anwendungsprogramme ausreichend. Für theoretische Untersuchungen verwenden wir gelegentlich auch beliebige Formeln. Diese Erweiterung ist unproblematisch, alles weitere gilt analog auch für beliebige $\varepsilon \in FMA$.

Definition 6 *Zugewiesene Variablen*
Die Menge der zugewiesenen Variablen $asgv(\alpha)$ eines Programms α ist definiert durch:

- $\mathrm{asgv}(\textbf{skip}) = \mathrm{asgv}(\textbf{abort}) = \emptyset$
- $\mathrm{asgv}(\alpha; \beta) = \mathrm{asgv}(\alpha) \cup \mathrm{asgv}(\beta)$
- $\mathrm{asgv}(\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta) = \mathrm{asgv}(\alpha) \cup \mathrm{asgv}(\beta)$
- $\mathrm{asgv}(\textbf{while } \varepsilon \textbf{ do } \alpha) = \mathrm{asgv}(\alpha)$
- $\mathrm{asgv}(\textbf{loop } \alpha \textbf{ times } \kappa) = \mathrm{asgv}(\alpha)$
- $\mathrm{asgv}(\textbf{var } \underline{x} = \underline{t} \textbf{ in } \alpha) = \mathrm{asgv}(\alpha) \setminus \underline{x}$
- $\mathrm{asgv}(p(\underline{t}; \underline{x})) = \underline{x}$
- $\mathrm{asgv}(\textbf{procbound } p(\underline{t}; \underline{x}) \textbf{ times } \kappa) = \underline{\mathrm{x}}$

Definition 7 *Aufgerufene Prozeduren*
calledprocs(α) ist die Menge aller Prozeduren, die in einem Programm α aufgerufenen werden.

Definition 8 *Prozedurdeklarationen und Prozedurdeklarationslisten*
Die Menge *PD* der Prozedurdeklarationen ist die Menge aller $p(\underline{x};\ \mathbf{var}\ \underline{y}).\alpha$ mit $p \in P_{\underline{s},\underline{s}'}$ $\underline{x},\underline{y} \in \hat{X}_{\underline{s},\underline{s}'}$, $\alpha \in PROG$ und $asgv(\alpha) \subseteq \underline{\mathrm{x}} \cup \underline{\mathrm{y}}$. α darf keine in der Rekursionstiefe beschränkten Prozeduraufrufe enthalten. p ist die definierte Prozedur der Prozedurdeklaration, α der Rumpf der Prozedur.

PDL ist die Menge aller Listen von Prozedurdeklarationen, so daß die aufgerufenen Prozeduren in den Rümpfen eine Teilmenge der definierten Prozeduren sind.

B.2 Semantik der Dynamischen Logik

Definition 9 *Algebra*
Eine Algebra $\mathcal{A}$ über einer Signatur *SIG* besteht aus einer nichtleeren Trägermenge A_s für jede Sorte s, einer Funktion $f_{\mathcal{A}} : A_{\underline{s}} \to A_{s'}$ für jedes $f \in OP_{\underline{s},s'}$. Für jede Prozedur $p \in P_{\underline{s},\underline{s}'}$ und jedes $n \in \mathbb{N}$ enthält $\mathcal{A}$ eine Relation $[\![p]\!]_{\mathcal{A},n}$ auf $A_{\underline{s},\underline{s}',\underline{s}'}$. (die Semantik von p bei Beschränkung der maximalen Rekursionstiefe auf n). $[\![p]\!]_{\mathcal{A},0}$ muß die leere Relation sein. $[\![p]\!]_{\mathcal{A}}$ bezeichnet die Semantik der Prozedur, die die Vereinigung aller $[\![p]\!]_{\mathcal{A},n}$ ist. Die Relation definiert den Zusammenhang zwischen initialen Werten der value- und reference-Parameter und den Ergebniswerten in den reference-Parametern.

Für die vordefinierten Sorten nehmen wir $A_{\mathrm{bool}} = \{tt,ff\}$, $A_{\mathrm{nat}} = \mathbb{N}$ an. Die Operationen auf Booleans und nat. Zahlen haben ihre übliche Semantik.

Definition 10 *Zustände*
Zur einer Signatur *SIG* und einer Algebra $\mathcal{A}$ über dieser Signatur, ist ein Zustand $\mathbf{z} \in ST_{\mathcal{A}}$ als eine Abbildung definiert, die Variablen der Sorte s auf Werte aus A_s abbildet. Der Zustand $\mathbf{z}[\underline{x} \leftarrow \underline{a}]$ ist die Abänderung des Zustands $\mathbf{z}$ an den Variablen $\underline{x}$ durch die Werte $\underline{a}$.

Definition 11 *Semantik von Ausdrücken*
Zu einer Algebra $\mathcal{A}$ und einem Zustand $\mathbf{z}$ sind die Semantik $[\![e]\!]_{\mathbf{z}} \in A_s$ eines DL-Ausdrucks $e \in DLEXPR_s$, und die Semantik $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ eines Programms (eine infix geschriebene Relation auf Zuständen) wie folgt definiert:

- $[\![x]\!]_{\mathbf{z}} = \mathbf{z}(x)$
- $[\![f(\underline{t})]\!]_{\mathbf{z}} = f_{\mathcal{A}}([\![\underline{t}]\!]_{\mathbf{z}})$ für $f \in OP_{\underline{s},s'}$ und $\underline{t} \in T_{\underline{s}}$
- $[\![\forall\ \underline{x}.e]\!]_{\mathbf{z}} = tt$ mit $\underline{x} \in \hat{X}_{\underline{s}}$ gdw. $[\![e]\!]_{\mathbf{z}[\underline{x} \leftarrow \underline{a}]} = tt$ für alle Werte $\underline{a} \in \mathcal{A}_{\underline{s}}$
- $[\![\exists\ \underline{x}.e]\!]_{\mathbf{z}} = tt$ mit $\underline{x} \in \hat{X}_{\underline{s}}$ gdw. $[\![e]\!]_{\mathbf{z}[\underline{x} \leftarrow \underline{a}]} = tt$ für einen Wert $\underline{a} \in \mathcal{A}_{\underline{s}}$
- $[\![(\varepsilon \supset e; e')]\!]_{\mathbf{z}}$ ist $[\![e]\!]_{\mathbf{z}}$, falls $[\![\varepsilon]\!]_{\mathbf{z}} = tt$, und $[\![e']\!]_{\mathbf{z}}$ sonst.
- $\mathbf{z}[\![\mathbf{skip}]\!]\mathbf{z}'$ gdw. $\mathbf{z} = \mathbf{z}'$
- $[\![\mathbf{abort}]\!]$ ist die leere Relation

- $\mathbf{z}[\![\underline{x}{:=}\underline{t}]\!]\mathbf{z}'$ gdw. $\mathbf{z}' = \mathbf{z}[\underline{x} \leftarrow [\![\underline{t}]\!]_{\mathbf{z}}]$, wobei jedes $[\![?]\!]_{\mathbf{z}}$ einen beliebigen Wert darstellt.
- $\mathbf{z}[\![\alpha; \beta]\!]\mathbf{z}'$ gdw. es ein $\mathbf{z}''$ gibt mit $\mathbf{z}[\![\alpha]\!]\mathbf{z}''$ und $\mathbf{z}''[\![\beta]\!]\mathbf{z}'$
- $\mathbf{z}[\![\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\ ]\!]\mathbf{z}'$ gdw.
  entweder $[\![\varepsilon]\!]_{\mathbf{z}} = tt$ und $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ oder $[\![\varepsilon]\!]_{\mathbf{z}} = ff$ und $\mathbf{z}[\![\beta]\!]\mathbf{z}'$
- $\mathbf{z}[\![\textbf{loop } \alpha \textbf{ times } \kappa]\!]\mathbf{z}'$ gdw.
  es Zustände $\mathbf{z}_0 := \mathbf{z}, \mathbf{z}_1, \ldots, \mathbf{z}_n := \mathbf{z}'$ gibt mit $n := [\![\kappa]\!]_{\mathbf{z}}$ so daß
  $\mathbf{z}_{i-1}[\![\alpha]\!]\mathbf{z}_i$ für jedes $1 \leq i \leq n$
- $\mathbf{z}[\![\textbf{while } \varepsilon \textbf{ do } \alpha]\!]\mathbf{z}'$ gdw.
  es Zustände $\mathbf{z}_0 := \mathbf{z}, \mathbf{z}_1, \ldots, \mathbf{z}_n := \mathbf{z}'$ gibt so daß
  $\mathbf{z}_{i-1}[\![\alpha]\!]\mathbf{z}_i$ für $1 \leq i \leq n$,
  $[\![\varepsilon]\!]_{\mathbf{z}_i} = tt$ für $1 \leq i < n$ und $[\![\varepsilon]\!]_{\mathbf{z}'} = ff$
- $\mathbf{z}[\![\textbf{var } \underline{x} = \underline{t} \textbf{ in } \alpha]\!]\mathbf{z}'$ gdw. $\mathbf{z}[\underline{x} \leftarrow \underline{a}][\![\alpha]\!]\mathbf{z}''$ und $\mathbf{z}' = \mathbf{z}''[\underline{x} \leftarrow [\![\underline{x}]\!]_{\mathbf{z}}]$ wobei $a_i = [\![t_i]\!]$ für $t_i \neq ?$ und andernfalls a_i beliebig ist.
- $\mathbf{z}[\![p(\underline{t}, \underline{x})]\!]\mathbf{z}'$ gdw. $\mathbf{z}(\underline{t}), \mathbf{z}(\underline{x}), \mathbf{z}'(\underline{x}) \in [\![p]\!]$ und $\mathbf{z}(y) = \mathbf{z}'(y)$ für alle $y \notin \underline{x}$
- $\mathbf{z}[\![\textbf{procbound } \kappa \textbf{ in } p(\underline{t}, \underline{x})]\!]\mathbf{z}'$ gdw. $\mathbf{z}(\underline{t}), \mathbf{z}(\underline{x}), \mathbf{z}'(\underline{x}) \in [\![p]\!]_n$, wobei $n = [\![\kappa]\!]_{\mathbf{z}}$, und $\mathbf{z}(y) = \mathbf{z}'(y)$ für alle $y \notin \underline{x}$
- $[\![\langle\alpha\rangle\ \varphi]\!]_{\mathbf{z}} = tt$ gdw. es ein $\mathbf{z}'$ gibt mit $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ und $[\![\varphi]\!]_{\mathbf{z}'} = tt$
- $[\![[\alpha]\ \varphi]\!]_{\mathbf{z}} = tt$ gdw. für alle $\mathbf{z}'$ mit $\mathbf{z}[\![\alpha]\!]\mathbf{z}'$ gilt: $[\![\varphi]\!]_{\mathbf{z}'} = tt$

Bemerkung 3 Die Semantik von Ausdrücken und Programmen ist eindeutig definiert, da jeder Fall der Definition die Zahl der elementaren Anweisungen in dem betrachteten Ausdruck verringert.

Definition 12 *Semantik von Prozedurdeklarationslisten*
Für eine Prozedurdeklarationsliste δ gilt $\mathcal{A} \models \delta$ gdw. für jede in δ enthaltene Prozedurdeklaration $p(\underline{x}; \underline{y}).\alpha$ und jedes $\kappa = 0 + 1 \ldots + 1$ (Repräsentation einer Zahl $n \geq 0$) gilt:

$$[\![\textbf{procbound } \kappa + 1 \textbf{ in } p(\underline{x}; \underline{y})]\!] = [\![\textbf{procbound } \kappa \textbf{ in } \alpha]\!]$$

Dabei steht **procbound** κ **in** α für das Programm, das aus α durch Ersetzung aller Prozeduraufrufe $q(\underline{\sigma}; \underline{z})$ mit **procbound** κ **in** $q(\underline{\sigma}; \underline{z})$ (für jeden Prozedurbezeichner q) hervorgeht.

Bemerkung 4 Eine Prozedurdeklaration legt die Semantik einer Prozedur eindeutig fest. Zum Beweis zeigt man induktiv, daß jedes $[\![p]\!]_{\mathcal{A},n}$ durch die obige Gleichung festgelegt wird. Es läßt sich so auch leicht zeigen, daß die $[\![p]\!]_{\mathcal{A},n}$ monoton wachsende Relationen sind.

Definition 13 *models-operator*

- $\mathcal{A}, \mathbf{z} \models \varphi$ gilt für eine Formel φ gdw. $[\![\varphi]\!]_{\mathbf{z}} = tt$
- $\mathcal{A} \models \varphi$ gilt gdw. in allen Zuständen $\mathbf{z}$: $\mathcal{A}, \mathbf{z} \models \varphi$
- $\models \varphi$ gilt gdw. wenn in jeder Algebra $\mathcal{A}$: $\mathcal{A} \models \varphi$
- $\Phi \models \psi$ gilt gdw. wenn in jeder Algebra $\mathcal{A}$ aus $\mathcal{A} \models \varphi$ für jedes $\varphi \in \Phi$ auch $\mathcal{A} \models \psi$ folgt.

Bemerkung 5 Die folgenden Aussagen gelten, falls i weder in α noch in ε vorkommt. Die ersten beiden charakterisieren while-Schleifen (sie erlauben Induktion über die Zahl der Iterationen). Die dritte Aussage erlaubt es, loop's zu vermeiden, deren Zähler in α vorkommt.

- $\models \langle \textbf{while } \varepsilon \textbf{ do } \alpha \rangle \ \varphi \leftrightarrow \exists\, i. \langle \textbf{loop if } \varepsilon \textbf{ then } \alpha \textbf{ times } i \rangle \ (\varphi \wedge \neg\, \varepsilon)$
- $\models \langle \textbf{loop } \alpha \textbf{ times } \kappa + 1 \rangle \ \varphi \leftrightarrow \langle \alpha; \textbf{loop } \alpha \textbf{ times } \kappa \rangle \ \varphi$
- $\models \langle \textbf{loop } \alpha \textbf{ times } \kappa \rangle \ \varphi \leftrightarrow (\forall\, i. i = \kappa \rightarrow \langle \textbf{loop } \alpha \textbf{ times } i \rangle \ \varphi)$

Bemerkung 6 Sei $\mathcal{A}$ eine Algebra mit $\mathcal{A} \models \delta$ für eine Prozedurdeklarationsliste δ, die eine Prozedurdeklaration $p(\underline{x}; \textbf{var } \underline{y}).\alpha$ enthält. Dann charakterisieren die folgenden drei Formeln die rekursive Prozedur (d.h. ihre Gültigkeit ist äquivalent zu der Prozedurdeklaration). Prozedurdeklarationslisten können also als Abkürzungen für Axiome betrachtet werden. Die Formeln erlauben auch die Induktion über die Schachtelungstiefe sowie ein Unfolding von Prozeduren. Die erste Formel gilt unabhängig von der Algebra. In der dritten Formel müssen $\underline{x}_0$ und $\underline{y}_0$ neue Variablen derselben Sorten wie $\underline{x}$ und $\underline{y}$ sein. **procbound** κ **in** α steht wieder für das Programm, das aus α durch Ersetzung aller Prozeduraufrufe $q(\underline{\sigma}; \underline{z})$ mit **procbound** κ **in** $q(\underline{\sigma}; \underline{z})$ hervorgeht.

- $\models \langle p(\underline{t}; \underline{z}) \rangle \ \varphi \leftrightarrow \exists\, \kappa. \langle \textbf{procbound } \kappa \textbf{ in } p(\underline{t}; \underline{z}) \rangle \ \varphi$
- $\mathcal{A} \models \langle \textbf{procbound } \kappa + 1 \textbf{ in } p(\underline{t}; \underline{z}) \rangle \ \varphi \leftrightarrow$
  $\langle \underline{x}_0, \underline{y}_0, \underline{x}, \underline{y} := \underline{x}, \underline{y}, \underline{t}, \underline{z}; \textbf{procbound } \kappa \textbf{ in } \alpha; \underline{x}, \underline{y}, \underline{y}_0 := \underline{x}_0, \underline{y}_0, \underline{y}; \underline{z} := \underline{y}_0 \rangle \ \varphi$
- $\mathcal{A} \models \langle p(\underline{t}; \underline{z}) \rangle \ \varphi$
  $\leftrightarrow \langle \underline{x}_0, \underline{y}_0, \underline{x}, \underline{y} := \underline{x}, \underline{y}, \underline{t}, \underline{z}; \alpha; \underline{x}, \underline{y}, \underline{y}_0 := \underline{x}_0, \underline{y}_0, \underline{y}; \underline{z} := \underline{y}_0 \rangle \ \varphi$

Definition 14 *(Basis-)Spezifikationen*
Eine Basis-Spezifikation *SPEC* = (*SIG,Ax,GAx,PAx*) besteht aus

- einer Signatur *SIG* = (*S,OP,P,X*).
- einer Menge von Axiomen *Ax* (Formeln über *SIG*).
- einer Menge *GAx* von Erzeugtheitsklauseln der Form: $s_1, \ldots s_n$ **generated by** $f_1, \ldots f_m$ (n,m > 0). Dabei sind $s_1, \ldots s_n \in \hat{S}^*$ und alle f_j haben eine Zielsorte in $s_1, \ldots s_n$.

- eine Menge PAx von Prozedurdeklarationslisten über SIG. Ist eine Prozedur in mehreren Prozedurdeklarationslisten erklärt, müssen die Deklarationen übereinstimmen.

Definition 15 *Semantik von Spezifikationen*
Eine Algebra $\mathcal{A}$ ist ein Modell von $SPEC$ (geschrieben als $\mathcal{A} \models SPEC$, wenn sie eine Algebra über der Signatur der Spezifikation ist, so daß

- $\mathcal{A} \models \varphi$ für jedes $\varphi \in Ax$
- Für jede Erzeugtheitsklausel $s_1, \ldots, s_n$ **generated by** $f_1, \ldots, f_m \in GAx$ und jedes $i = 1 \ldots n$, läßt sich jedes Element $a \in \mathcal{A}_{s_i}$ als die Semantik $a = [\![t]\!]_{\mathbf{z}}$ eines Terms t erhalten, der keine Variablen der Sorten $s_1, \ldots s_n$ enthält, und dessen Operationssymbole nur aus $\{f_1, \ldots, f_m\}$ stammen.
- $\mathcal{A} \models \delta$ für jedes $\delta \in PAx$

Bemerkung 7 Zu jedem Modell einer Spezifikation (SIG,Ax,GAx,$\emptyset$), deren Signatur keine Prozedurbezeichner enthält, gibt es genau eine Erweiterung zu einem Modell einer um Prozedurdeklarationen PAx und die Menge P der darin definierten Prozedurbezeichner erweiterten Spezifikation ($SIG \cup P$,Ax,GAx, PAx).

Bemerkung 8 Wir schreiben $SPEC \models \varphi$, wenn in jedem Modell $\mathcal{A}$ von $SPEC$ $\mathcal{A} \models \varphi$ gilt.

Theorem 11 *Korrektheit und Vollständigkeit*
Die Theorie einer Basisspezifikation läßt sich korrekt und vollständig axiomatisieren, wenn man für jedes Generiertheitsprinzip $s_1, \ldots s_n$ **generated by** f_1, $\ldots$, f_m eine (algorithmisch nicht nutzbare) Omega-Regel addiert. Diese lautet: Wenn für eine Formel $\varphi(x)$ mit freier Variable x aus einer der Sorten $s_1, \ldots s_n$ alle (evtl. unendlich vielen) Formeln $\varphi(t)$ ableitbar sind für jeden Term t, der nur mit den Konstruktoren $f_1, \ldots f_m$ gebildet ist, und nur Variablen enthält, die nicht aus den Sorten $s_1, \ldots s_n$ enthält, dann ist $\forall\, x.\varphi(x)$ ableitbar.

Bei der Realisierung eines Kalküls werden die Omega-Regeln durch entsprechende strukturelle Induktionsprinzipien ersetzt. Diese sind zwar schwächer als Omega-Regeln aber für die praktische Anwendung ausreichend.

Wir wollen den Beweis hier nicht führen. Die Idee des Beweises beruht auf der Übersetzung aller DL-Formeln in äquivalente prädikatenlogische Formeln. Dazu wird jedes Programm α in eine Relation R_α (Eingabe: Die Variablen des Programms, Ausgabe: die zugewiesenen Variablen des Programms) übersetzt. Damit reduziert sich der Korrektheits- und Vollständigkeitsbeweis auf prädikatenlogische Spezifikationen mit Generiertheitsklauseln, für die die korrekte und vollständige Axiomatisierbarkeit mit Hilfe der Omega-Regel bekannt ist (siehe [Rei98]).

Anhang C

Spezifikationen und Lemmata für das Modularisierungstheorem

C.1 Allgemeine Spezifikationen

Die Spezifikationen für natürliche Zahlen, Listen und dynamische Funktionen finden sich in Anhang E

diagtype =
data specification
    diagtype = mn | 0n | m0;
    variables c: diagtype;
end data specification

---

state =
specification
    sorts state;
    variables st: state;
end specification

---

f-state-state =
actualize Dynfun **with parameter** state **by morphism**
  dom → state, codom → state, dynfun → f-state-state,
  .[.] → . ⌈ . ⌉$_s$
end actualize

---

iterate =
enrich nat, f-state-state **with**
functions . ^ . : f-state-state × nat → f-state-state **prio** 9;

axioms

it-base-ax : $(f \hat{} 0)\lceil st \rceil_s = st$,
it-rec-ax : $(f \hat{} m +1)\lceil st \rceil_s = f\lceil (f \hat{} m)\lceil st \rceil_s \rceil_s$

end enrich

---

stream =
actualize Dynfun **with** nat, **parameter** state **by morphism**
dom → nat, codom → state, dynfun → stream,
. [.] → . ⌈ . ⌉, f → s,
end actualize

---

enrstream =
enrich stream, iterate **with**
functions

cons	:	state × stream	→	stream;
cdr	:	stream	→	stream;
app	:	stream × nat × stream	→	stream;
nthcdr	:	stream × nat	→	stream;

axioms

cons-base-ax : $cons(st, s)\lceil 0 \rceil = st$,
cons-rec-ax : $cons(st, s)\lceil m +1 \rceil = s\lceil m \rceil$,
cdr-ax : $cdr(s)\lceil m \rceil = s\lceil m +1 \rceil$,
app-base-ax : $app(s, 0, s_0) = s_0$,
app-rec-ax : $app(s, m +1, s_0) = cons(s\lceil 0 \rceil, app(cdr(s), m, s_0))$,
nthcdr-base-ax : $nthcdr(s, 0) = s$,
nthcdr-rec-ax : $nthcdr(s, m +1) = nthcdr(cdr(s), m)$,
streamchoice :
$(\forall\ m.\ \exists\ st_1.\ st_1 = (f \hat{} m)\lceil st_0 \rceil_s)$
$\rightarrow (\exists\ s.\ \forall\ m.\ s\lceil m \rceil = (f \hat{} m)\lceil st_0 \rceil_s)$

end enrich

---

tuple =
data specification
using enrstream
tuple = mkt (. .s : stream, . .i : nat, . .j : nat);
variables t_1, t_0, t: tuple;
end data specification

---

f-tup-tup =
actualize iterate **with** tuple **by morphism**
state → tuple, f-state-state → f-tup-tup, . ⌈ . ⌉$_s$ → . ⟦ . ⟧,
st → t, f → ft
end actualize

---

rule =
enrich enrstream **with**
predicates
Trace : stream;
final : state;
procedures
RULE : → state; (: beliebige Prozedur als ASM-Regel :)

axioms

Trace-def :
Trace(s)
↔ (∀ m, st. st = s⌈m⌉
→ ⟨**if** ¬ final(st) **then** RULE(; st)⟩ st = s⌈m +1⌉),

final-def : (: Regel terminiert nicht ⇒ Endzustand :)
(¬ ⟨RULE(; st)⟩ true) → final(st) ,

choice : (: Auswahlaxiom für RULE :)
(∀ st. ⟨**if** ¬ final(st) **then** RULE(; st)⟩ true)
→ ∃ f. ∀ st$_0$. ⟨st := st$_0$; **if** ¬ final(st) **then** RULE(; st)⟩ st = f⌈st$_0$⌉$_s$

end enrich

---

rule' =
rename rule **by morphism**
stream → stream', state → state', . ⌈ . ⌉ → . ⌈ . ⌉', cons → cons',
cdr → cdr', app → app', nthcdr → nthcdr', Trace → Trace',
final → final', RULE → RULE', s → s', st → st'
end rename

C.2 Verfeinerung deterministischer ASMs

C.2.1 Spezifikation

detequiv =
enrich rule, rule', diagtype **with**
functions

ndt : state × state' → diagtype;
exec0n : state × state' → nat;
execm0 : state × state' → nat;

predicates

INV : state × state'; (: Zusammenhangsinvariante :)
IN : state × state'; (: Eingaberelation :)
OUT : state × state'; (: Ausgaberelation :)
PROP : state × state';

variables i, j, k: nat;

axioms

init-ax : IN(st, st') → INV(st, st'),
finboth-ax : final(st) ∧ final'(st') ∧ INV(st, st') → OUT(st, st'),
fin1-ax : final(st) ∧ INV(st, st') ∧ ¬ final'(st') → ndt(st, st') = 0n,
fin2-ax : final'(st') ∧ INV(st, st') ∧ ¬ final(st) → ndt(st, st') = m0,

mton-ax :
INV(st, st') ∧ ¬ final(st) ∧ ¬ final'(st') ∧ ndt(st, st') = m0
→ ⟨**if** ¬ final(st) **then** RULE(; st) ⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
⟨**if** ¬ final'(st') **then** RULE'(; st') ⟩
∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j ⟩ INV(st, st'),

0ton-ax :
INV(st, st') ∧ ¬ final'(st') ∧ ndt(st, st') = 0n ∧ exec0n(st, st') = k
→ ⟨**if** ¬ final'(st') **then** RULE'(; st') ⟩
∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j⟩
(INV(st, st') ∧ (¬ final'(st') ∧ ndt(st, st') = 0n → exec0n(st, st') < k)),

mto0-ax :
INV(st, st') ∧ ¬ final(st) ∧ ndt(st, st') = m0 ∧ execm0(st, st') = k
→ ⟨**if** ¬ final(st) **then** RULE(; st) ⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(INV(st, st') ∧ (¬ final(st) ∧ ndt(st, st') = m0 → execm0(st, st') < k)),

prop-def :
PROP(st, st')
↔ ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j⟩ INV(st, st')
end enrich

C.2.2 Bewiesene Theoreme

finite-0ton (der interessante Fall von Lemma 2 aus Abschnitt 6.2.3)

INV(st, st'), ndt(st, st') = 0n, ¬ final'(st')

$\vdash \langle$**if** $\neg$ final'(st') **then** RULE'(; st') $\rangle$
$\quad \exists$ j. $\langle$**loop if** $\neg$ final'(st') **then** RULE'(; st') **times** j$\rangle$
$\quad\quad$ (INV(st, st') $\wedge$ (final'(st') $\vee$ ndt(st, st') $\neq$ 0n))

- used lemmas : 0ton-ax
- used by : compl-step, completeness

finite-mto0

INV(st, st'), ndt(st, st') = m0, $\neg$ final(st)
$\vdash \langle$**if** $\neg$ final(st) **then** RULE(; st) $\rangle$
$\quad \exists$ i. $\langle$**loop if** $\neg$ final(st) **then** RULE(; st) **times** i$\rangle$
$\quad\quad$ (INV(st, st') $\wedge$ (final(st) $\vee$ ndt(st, st') $\neq$ m0))

- used lemmas : mto0-ax
- used by : corr-step, correctness

corr-step (Lemma 1 aus Abschnitt 6.2.3)

PROP(st, st') $\vdash \langle$**if** $\neg$ final'(st') **then** RULE'(; st')$\rangle$ PROP(st, st')

- used lemmas : finite-mto0, fin1-ax, 0ton-ax, mton-ax, prop-def
- used by : correctness

compl-step

PROP(st, st') $\vdash \langle$**if** $\neg$ final(st) **then** RULE(; st)$\rangle$ PROP(st, st')

- used lemmas : finite-0ton, fin2-ax, mto0-ax, mton-ax, prop-def
- used by : completeness

correctness (Korrektheit der Verfeinerung)

IN(st, st')
$\vdash$ [**while** $\neg$ final'(st') **do** RULE'(; st')]
$\quad \langle$**while** $\neg$ final(st) **do** RULE(; st)$\rangle$ OUT(st, st')

- used lemmas : finboth-ax, fin2-ax, finite-mto0, corr-step, init-ax, prop-def

completeness (Vollständigkeit der Verfeinerung)

IN(st, st')
$\vdash$ [**while** $\neg$ final(st) **do** RULE(; st)]
$\quad \langle$**while** $\neg$ final'(st') **do** RULE'(; st')$\rangle$ OUT(st, st')

- used lemmas : finboth-ax, fin1-ax, finite-0ton, compl-step, init-ax, prop-def

C.3 Verfeinerung indeterministischer ASMs – Diagramme indeterministischer Größe

C.3.1 Spezifikation

genindeteqtrace =
enrich rule, rule', f-tup-tup, diagtype **with**
functions
ndt : state × state' → diagtype;
exec0n : state × state' → nat;
execm0 : state × state' → nat;
predicates
INV : state × state';
INV' : state × state';
KPROP : state × state';
VPROP : state × state';
IN, : state × state';
OUT : state × state';
p : stream' × tuple × tuple;
variables i, i_0, i_1, j, j_0, k: nat;

axioms

init-ax : IN(st, st') → INV(st, st'),
finboth-ax : final(st) ∧ final'(st') ∧ INV(st, st') → OUT(st, st'),
fin1-ax : final(st) ∧ INV(st, st') ∧ ¬ final'(st') → ndt(st, st') = 0n,
fin2-ax : final'(st') ∧ INV(st, st') ∧ ¬ final(st) → ndt(st, st') = m0,

mton-corr-ax :
INV(st, st') ∧ ndt(st, st') = m0 ∧ Trace'(s')
∧ st' = s'⌈0⌉′ ∧ ¬ final(st) ∧ ¬ final'(st')
→ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ j. ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i ⟩ INV(st, s'⌈j +1⌉′),

0ton-corr-ax :
INV(st, st') ∧ ndt(st, st') = 0n ∧ exec0n(st, st') = k
∧ Trace'(s') ∧ st' = s'⌈0⌉′ ∧ ¬ final'(st')
→ ∃ j. INV(st, s'⌈j +1⌉′)
∧ (¬ final'(s'⌈j +1⌉′) ∧ ndt(st, s'⌈j +1⌉′) = 0n → exec0n(st, s'⌈j +1⌉′) < k),

mto0-corr-ax : (: folgt aus mto0-comp-ax, genügt für Trace-Korrektheit :)
INV(st, st') ∧ ndt(st, st') = m0 ∧ execm0(st, st') = k ∧ ¬ final(st)
→ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(INV(st, st') ∧ (¬ final(st) ∧ ndt(st, st') = m0 → execm0(st, st') < k)),

mton-comp-ax :
INV(st, st') ∧ ndt(st, st') = m0 ∧ Trace(s)
∧ st = s⌈0⌉ ∧ ¬ final(st) ∧ ¬ final'(st')
→ ⟨**if** ¬ final'(st') **then** RULE'(; st')⟩
∃ i. ∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j ⟩ INV(s⌈i +1⌉, st'),

mto0-comp-ax :
INV(st, st') ∧ ndt(st, st') = m0 ∧ execm0(st, st') = k ∧ Trace(s)
∧ st = s⌈0⌉ ∧ ¬ final(st)
→ ∃ i. INV(s⌈i +1⌉, st')
∧ (¬ final(s⌈i +1⌉) ∧ ndt(s⌈i +1⌉, st') = m0 → execm0(s⌈i +1⌉, st') < k),

0ton-comp-ax : (: folgt aus 0ton-corr-ax :)
INV(st, st') ∧ ndt(st, st') = 0n ∧ exec0n(st, st') = k ∧ ¬ final'(st')
→ ⟨**if** ¬ final'(st') **then** RULE'(; st')⟩
∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j⟩
(INV(st, st') ∧ (¬ final'(st') ∧ ndt(st, st') = 0n → exec0n(st, st') < k)),

choice-ax : (: Auswahlaxiom :)
(∀ t. ∃ t_0. p(s', t, t_0)) → (∃ ft. ∀ t. p(s', t, ft⟦t⟧)),

diagonal-ax : (: Auswahlaxiom :)
∀ m. ∃ st. st = (ft ↑ m)⟦mkt(s_0, 0, 0)
→ ∃ s. ∀ m. s⌈m⌉ = ft ↑ m⟦mkt(s_0, 0, 0)⟧.s⌈m⌉,

kprop-def :
KPROP(st, st')
↔ ∀ s'. st' = s'⌈0⌉′ ∧ Trace'(s')
→ ∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(∃ m. INV(st, s'⌈m⌉′)),

vprop-def :
VPROP(st, st')
↔ ∀ s. st = s⌈0⌉ ∧ Trace(s)
→ ∃ j. ⟨**loop if** ¬ final'(st') **then** RULE'(; st') **times** j⟩
(∃ m. INV(s⌈m⌉, st')),

inv'-def : INV'(st, st') ↔ INV(st, st') ∧ (final(st) ↔ final'(st'))

p-def : (: Prädikat, das das Anhängen von Diagrammen beschreibt :)
p(s', t, t_0)
↔ INV'(t.s⌈t.i⌉, s'⌈t.j⌉′) ∧ Trace(t.s) ∧ Trace'(s')
→ Trace(t_0.s) ∧ (∀ i_1. ¬ t.i < i_1 → t.s⌈i_1⌉ = t_0.s⌈i_1⌉)
∧ t.i < t_0.i ∧ t.j < t_0.j ∧ INV'(t_0.s⌈t_0.i⌉, s'⌈t_0.j⌉′),

end enrich

C.3.2 Bewiesene Theoreme

fin-0ton

INV(st, st'), ndt(st, st') = 0n, ¬ final'(st')
⊢ ⟨**if** ¬ final'(st') **then** RULE′(; st')⟩
∃ j. ⟨**loop if** ¬ final'(st') **then** RULE′(; st') **times** j⟩
(INV(st, st') ∧ (final'(st') ∨ ndt(st, st') ≠ 0n))

- used lemmas : 0ton-comp-ax
- used by : compl-step, completeness

fin-mto0

INV(st, st'), ndt(st, st') = m0, ¬ final(st)
⊢ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(INV(st, st') ∧ (final(st) ∨ ndt(st, st') ≠ m0))

- used lemmas : mto0-corr-ax
- used by : add-diagram, corr-step, correctness, equiv-final

finite-0ton

ndt(st, st') = 0n, INV(st, st'), Trace'(s'), s'⌈0⌉′ = st', ¬ final'(st')
⊢ ∃ j. INV(st, s'⌈j +1⌉′) ∧ (final'(s'⌈j +1⌉′) ∨ ndt(st, s'⌈j +1⌉′) ≠ 0n)

- used lemmas : 0ton-corr-ax
- used by : add-diagram, equiv-final

corr-step

KPROP(st, st') ⊢ [**if** ¬ final'(st') **then** RULE′(; st')] KPROP(st, st')

- used lemmas : fin-mto0, fin1-ax, 0ton-corr-ax, mton-corr-ax, kprop-def
- used by : correctness

compl-step

VPROP(st, st') ⊢ [**if** ¬ final(st) **then** RULE(; st)] VPROP(st, st')

- used lemmas : fin-0ton, fin2-ax, mto0-comp-ax, mton-comp-ax, vprop-def
- used by : completeness

correctness (Korrektheit der Verfeinerung)

IN(st, st')
$\vdash$ [**while** $\neg$ final'(st') **do** RULE'(; st')]
⟨**while** $\neg$ final(st) **do** RULE(; st)⟩ OUT(st, st')

- used lemmas : fin-mto0, finboth-ax, fin2-ax, corr-step, init-ax, kprop-def

completeness (Vollständigkeit der Verfeinerung)

IN(st, st')
$\vdash$ [**while** $\neg$ final(st) **do** RULE(; st)]
⟨**while** $\neg$ final'(st') **do** RULE'(; st')⟩ OUT(st, st')

- used lemmas : finboth-ax, fin1-ax, fin-0ton, compl-step, init-ax, vprop-def

equiv-final (Lemma 3 aus Abschnitt 6.3)

INV(st, st'), Trace'(s'), s'⌈0⌉' = st'
$\vdash$ $\exists$ i. ⟨**loop if** $\neg$ final(st) **then** RULE(; st) **times** i⟩ ($\exists$ j. INV'(st, s'⌈j⌉'))

- used lemmas : fin2-ax, fin-mto0, fin1-ax, finite-0ton, inv'-def
- used by : add-diagram

add-diagram (Lemma 4 aus Abschnitt 6.3)

INV'(st, st'), Trace'(s'), s'⌈0⌉' = st'
$\vdash$ ⟨**if** $\neg$ final(st) **then** RULE(; st)⟩
$\exists$ i. ⟨**loop if** $\neg$ final(st) **then** RULE(; st) **times** i⟩
($\exists$ j. INV'(st, s'⌈j +1⌉'))

- used lemmas : fin1-ax, 0ton-corr-ax, fin-mto0, fin2-ax, mto0-corr-ax, finite-0ton, equiv-final, mton-corr-ax, inv'-def
- used by : totality

totality (Totalität der Relation, die Diagrammanfügung beschreibt)

$\vdash$ $\forall$ s', t. $\exists$ t_0. p(s', t, t_0)

- used lemmas : inv'-def, p-def, add-diagram
- used by : choice-concl, ind-choice-concl

choice-concl (Existenz einer Funktion, die ein Diagramm addiert)

$\vdash \exists$ ft. p(s', t, ft⟦t⟧)

- used lemmas : totality, choice-ax

ind-choice-concl (Spezialfall von *choice-concl* für $ft \uparrow m$)

$\vdash \exists$ ft. $\forall$ m. p(s', (ft $\uparrow$ m)⟦t⟧, (ft $\uparrow$ m +1)⟦t⟧)

- used lemmas : totality, choice-ax
- used by : trace-correctness

diagonal (Diagonalisierungsargument für m konstruierte Diagramme)

$t_0 = mkt(s_0, 0, 0)$, $t = (ft \uparrow m)⟦t_0⟧$,
$Trace(s_0)$, Trace'(s'), $INV'(s_0\lceil 0\rceil, s'\lceil 0\rceil')$,
$\forall$ k. p(s', (ft $\uparrow$ k)⟦t_0⟧, (ft $\uparrow$ k +1)⟦t_0⟧)
$\vdash$ $INV'(t.s\lceil t.i\rceil, s'\lceil t.j\rceil')$
$\wedge$ m $\leq$ t.i $\wedge$ m $\leq$ t.j $\wedge$ Trace(t.s)
$\wedge$ ($\forall$ i, j. i < j $\wedge$ j $\leq$ m
$\rightarrow$ (ft $\uparrow$ i)⟦t_0⟧.i < (ft $\uparrow$ j)⟦t_0⟧.i $\wedge$ (ft $\uparrow$ i)⟦t_0⟧.j < (ft $\uparrow$ j)⟦t_0⟧.j)
$\wedge$ ($\forall$ j, k. j $\leq$ m $\wedge$ k $\leq$ (ft $\uparrow$ j)⟦t_0⟧.i $\rightarrow$ (ft $\uparrow$ j)⟦t_0⟧.s$\lceil$k$\rceil$ = t.s$\lceil$k$\rceil$)

- used lemmas : p-def, inv'-def
- used by : trace-correctness

trace-correctness (Trace-Korrektheit der Verfeinerung)

Trace'(s'), $INV'(st, s'\lceil 0\rceil')$
$\vdash \exists$ s. Trace(s) $\wedge$ s$\lceil 0\rceil$ = st $\wedge$ ($\forall$ m, k. $\exists$ i, j. m $\leq$ i $\wedge$ k $\leq$ j $\wedge$ $INV'(s\lceil i\rceil, s'\lceil j\rceil'))$

- used lemmas : diagonal, diagonal-ax, ind-choice-concl, inv'-def

C.4 Iterative Verfeinerung für indeterministische ASMs

C.4.1 Spezifikation

it-indetcorr =
enrich rule, rule', diagtype **with**
functions

ndt : state × state' → diagtype ;
exec0n : state × state' → nat ;
execm0 : state × state' → nat ;

predicates

INV : state × state';
IN : state × state';
OUT : state × state';
KPROP : state × state';
MINV : state; (: existierende Invariante für ASM :)
MINVNOW : state';
MINV' : state'; (: konstruierte Invariante für ASM′ :)

variables i, j, j_0, k: nat;

axioms

minv-ax :
IN(st, st')
→ ∀ i. [**loop if** ¬ final(st) **then** RULE(; st) **times** i] MINV(st),

init-ax : IN(st, st') → INV(st, st') ∧ MINVNOW(st'),
finboth-ax : final(st) ∧ final'(st') ∧ INV(st, st') ∧ MINV(st) → OUT(st, st'),
fin1-ax : final(st) ∧ INV(st, st') ∧ ¬ final'(st') ∧ MINV(st) → ndt(st, st') = 0n,
fin2-ax : final'(st') ∧ INV(st, st') ∧ ¬ final(st) ∧ MINV(st) → ndt(st, st') = m0,

mton-corr-ax :
INV(st, st') ∧ MINV(st) ∧ ¬ final(st) ∧ ¬ final'(st') ∧ ndt(st, st') = m0
→ [**if** ¬ final'(st') **then** RULE'(; st')]
∃ j. [**loop if** ¬ final'(st') ∧ ¬ MINVNOW(st') **then** RULE'(; st') **times** j]
(MINVNOW(st')
∧ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩ INV(st, st')),

0ton-corr-ax :
INV(st, st') ∧ MINV(st) ∧ MINVNOW(st') ∧ ¬ final'(st')
∧ ndt(st, st') = 0n ∧ exec0n(st, st') = k
→ [**if** ¬ final'(st') **then** RULE'(; st')]
∃ j. [**loop if** ¬ final'(st') ∧ ¬ MINVNOW(st') **then** RULE'(;st') **times** j]
(INV(st, st') ∧ MINVNOW(st')
∧ (¬ final'(st') ∧ ndt(st, st') = 0n → exec0n(st, st') < k)),

mto0-corr-ax :
INV(st, st') ∧ MINV(st) ∧ MINVNOW(st') ∧ ¬ final(st)
∧ ndt(st, st') = m0 ∧ execm0(st, st') = k
→ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(INV(st, st') ∧ (¬ final(st) ∧ ndt(st, st') = m0 → execm0(st, st') < k)),

kprop-def :
KPROP(st, st')
↔ ∀ i. [**loop if** ¬ final(st) **then** RULE(; st) **times** i] MINV(st)
∧ (∃ j. [**loop if** ¬ final'(st') ∧ ¬ MINVNOW(st')
then RULE'(; st') **times** j]
(MINVNOW(st')
∧ (∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
INV(st, st')))),

minv'-def : (MINVNOW(st') → (∃ st. INV(st, st') ∧ MINV(st))) → MINV'(st')

end enrich

C.4.2 Bewiesene Theoreme

finite-0ton

INV(st, st'), MINV(st), ndt(st, st') = 0n, ¬ final'(st'), MINVNOW(st')
⊢ [**if** ¬ final'(st') **then** RULE′(; st')]
∃ j. [**loop if** ¬ final'(st') **then** RULE′(; st') **times** j]
(INV(st, st') ∧ MINVNOW(st') ∧ (final'(st') ∨ ndt(st, st') ≠ 0n))

- used lemmas : 0ton-corr-ax

finite-mto0

∀ i. [**loop if** ¬ final(st) **then** RULE(; st) **times** i] MINV(st),
INV(st, st'), ndt(st, st') = m0, ¬ final(st), MINVNOW(st')
⊢ ⟨**if** ¬ final(st) **then** RULE(; st)⟩
∃ i. ⟨**loop if** ¬ final(st) **then** RULE(; st) **times** i⟩
(INV(st, st')
∧ (final(st) ∨ ndt(st, st') ≠ m0)
∧ (∀ i. [**loop if** ¬ final(st) **then** RULE(; st) **times** i] MINV(st)))

- used lemmas : mto0-corr-ax
- used by : corr-step, correctness

corr-step

KPROP(st, st') ⊢ [**if** ¬ final'(st') **then** RULE′(; st')] KPROP(st, st')

- used lemmas : finite-mto0, fin1-ax, 0ton-corr-ax, mton-corr-ax, kprop-def

- used by : corr-j-steps, correctness

kprop-minv'

KPROP(st, st') ⊢ MINV'(st')

- used lemmas : minv'-def, kprop-def
- used by : newinvariance

in-kprop

IN(st, st') ⊢ KPROP(st, st')

- used lemmas : init-ax, minv-ax, kprop-def
- used by : corr-j-steps, correctness, newinvariance

corr-j-steps

KPROP(st, st')
⊢ [**loop if** ¬ final'(st') **then** RULE′(; st') **times** j] KPROP(st, st')

- used lemmas : in-kprop, kprop-def, corr-step
- used by : newinvariance

correctness

IN(st, st')
⊢ [**while** ¬ final'(st') **do** RULE′(; st')]
⟨**while** ¬ final(st) **do** RULE(; st)⟩ OUT(st, st')

- used lemmas : finboth-ax, fin2-ax, finite-mto0, kprop-def, corr-step, in-kprop

newinvariance (Theorem 9 aus Abschnitt 6.5)

∃ st. IN(st, st')
⊢ ∀ j. [**loop if** ¬ final'(st') **then** RULE′(; st') **times** j] MINV'(st')

- used lemmas : kprop-minv', in-kprop, corr-j-steps

Anhang D

Definition zulässiger Codesequenzen (Ketten)

D.1 Definition linearer Ketten

```
L-CHAIN#(co, db5; var col)
begin
var instr = code(co, db5)
in if is_try_me(instr)
   then L-CHAIN-TRY-ME#(co, db5; col)
   else if is_clause(instr)
        then col := [co]
        else if instr = nil'
             then col := []
             else abort
end;

L-CHAIN-TRY-ME#(co, db5; var col)
begin
var instr = code(co, db5),
    follow = code(co +1, db5)
in if instr = try_me_else(N)
   then if is_clause(follow)
        then begin
             L-CHAIN-RETRY-ME#(N, db5; col);
             col := [co +1 | col]
             end
        else abort
   else abort
end;

L-CHAIN-RETRY-ME#(co, db5; var col)
```

```
begin
var instr = code(co, db5),
    follow = code(co +1, db5)
in if instr = retry_me_else(N)
   then if is_clause(follow)
        then begin
             L-CHAIN-RETRY-ME#(where(instr), db5; col);
             col := [co +1 | col]
             end
        else abort
   else if is_trust_me(instr)
        then if is_clause(follow)
             then col := [co +1]
             else abort
        else abort
end
```

D.2 Definition geschachtelter Ketten mit Switching

```
S-ANY-CHAIN#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if is_retry_me(instr) ∨ is_trust_me(instr)
   then S-CHAIN-RETRY-ME#(trm, co, db7; col)
   else if is_retry(instr) ∨ is_trust(instr)
        then S-CHAIN-RETRY#(trm, co, db7; col)
        else S-CHAIN-REC#(trm, co, db7; col)
end;

S-CHAIN#(trm, co, db7; var col)
begin
if co = failcode then col := []
else S-CHAIN-REC#(trm, co, db7; col)
end;

S-CHAIN-REC#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if is_clause(instr) then col := [co] else
   if instr = try(N) then var col2 = []
                          in begin
                             S-CHAIN-REC#(trm, N, db7; col);
                             S-CHAIN-RETRY#(trm, co +1, db7; col2);
                             col := append(col, col2)
```

```
                                end
   else
   if instr = try_me_else(N) then var col_2 = []
                                    in begin
                                       S-CHAIN-REC#(trm, co +1, db_7; col);
                                       S-CHAIN-RETRY-ME#(trm, N, db_7; col_2);
                                       col := append(col, col_2)
                                       end
   else
   if ¬ is_struct(trm) ∨ arity(trm) < argindex(instr) then abort else
   var xi = arg(trm, argindex(instr))
   in if instr = switch_on_term(argindex, N_s, N_c, N_v, N_l)
      then
      if is_struct(xi) then S-CHAIN#(trm, N_s, db_7; col) else
      if is_const(xi) then S-CHAIN#(trm, N_c, db_7; col) else
      if is_var(xi) then S-CHAIN#(trm, N_v, db_7; col) else
      if is_list(xi) then S-CHAIN#(trm, N_l, db_7; col) else abort
      else if instr = switch_on_constant(argindex, tabsize, table)
           then if is_const(xi)
                then S-CHAIN#(trm, hashc(table, tabsize, constsym(xi),
                                   db_7), db_7; col)
                else abort
      else if instr = switch_on_structure(argindex, tabsize, table)
           then if is_struct(xi)
                then S-CHAIN#(trm, hashs(table, tabsize, funct(xi),
                                   arity(xi), db_7), db_7; col)
                else abort
           else abort
      end;

S-CHAIN-RETRY-ME#(trm, co, db_7; var col)
begin
var instr = code(co, db_7)
in if instr = retry_me_else(N)
   then var col_2 = []
        in begin
           S-CHAIN-REC#(trm, co +1, db_7; col);
           S-CHAIN-RETRY-ME#(trm, N, db_7; col_2);
           col := append(col,col_2)
           end
   else if is_trust_me(instr)
        then S-CHAIN-REC#(trm, co +1, db_7; col)
        else abort
end;

S-CHAIN-RETRY#(trm, co, db_7; var col)
```

```
begin
var instr = code(co, db7)
in if instr = retry(N)
   then var col2 = []
        in begin
           S-CHAIN-REC#(trm, N, db7; col);
           S-CHAIN-RETRY#(trm, co +1, db7; col2);
           col := append(col, col2)
           end
   else if instr = trust(N)
        then S-CHAIN-REC#(trm, N, db7; col)
        else abort
end;

S-CHAIN-RET#(trm, co, db7; var col)
begin
var instr = code(co, db7)
in if is_retry_me(instr) ∨ is_trust_me(instr)
   then S-CHAIN-RETRY-ME#(trm, co, db7; col)
   else if is_retry(instr) ∨ is_trust(instr)
        then S-CHAIN-RETRY#(trm, co, db7; col)
        else abort
end;

S-APP-CHAINS-RET#(decglseq', p, stl, db7; var col)
begin
if stl = [] then col := []
else var col2 = []
     in begin
        S-CHAIN-RET#(acg[car(stl), p[car(stl)], db7; col);
        S-APP-CHAINS-RET#(decglseq', p, cdr(stl), db7; col2);
        col := append(col, col2)
        end
end
```

D.3 Definition der Kettenlänge geschachtelter Ketten mit Switching

```
C-S-ANY-CHAIN#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if is_retry_me(instr) ∨ is_trust_me(instr) then
       C-S-CHAIN-RETRY-ME#(trm, co, db7; m)
    else if is_retry(instr) ∨ is_trust(instr) then
```

```
        C-S-CHAIN-RETRY#(trm, co, db7; m)
    else C-S-CHAIN-REC#(trm, co, db7; m)
end;

C-S-CHAIN#(trm, co, db7; var m)
begin
if co = failcode then m := 0
else C-S-CHAIN-REC#(trm, co, db7; m)
end;

C-S-CHAIN-REC#(trm, co, db7; var m)
begin
var instr = code(co, db7)
in if is_clause(instr) then m := 0 else
   if instr = try(N) then C-S-CHAIN-TRY#(trm, N, db7; m); else
   if instr = try_me(N) then C-S-CHAIN-TRY-ME#(trm, N, db7; m); else
   if ¬ is_struct(trm) ∨ arity(trm) < argindex(instr) then abort
   else var xi = arg(trm, argindex(instr)) in
      if instr = switch_on_term(argindex, Ns, Nc, Nv, Nl) then
         if is_struct(xi) then
            if Ns = failcode then m := 0
            else begin C-S-CHAIN-REC#(trm, Ns, db7; m); m := m +1 end
         else if is_const(xi) then
            if Nc = failcode then m := 0
            else begin C-S-CHAIN-REC#(trm, Nc, db7; m); m := m +1 end
         else if is_var(xi) then
            if Nv = failcode then m := 0
            else begin C-S-CHAIN-REC#(trm, Nv, db7; m); m := m +1 end
         else if is_list(xi) then
            if Nl = failcode then m := 0
            else begin C-S-CHAIN-REC#(trm, Nl, db7; m); m := m +1 end
         else abort
      else if instr = switch_on_constant(argindex, tabsize, table) then
         if is_const(xi) then
            var preg = hashc(table, tabsize, constsym(xi)) in
                if preg = failcode then m := 0
                else begin
                     C-S-CHAIN-REC#(trm, preg, db7; m);
                     m := m +1
                     end
         else abort
      else if instr = switch_on_structure(argindex, tabsize, table) then
         if is_struct(xi) then
            var preg = hashs(table, tabsize, funct(xi)) in
                if preg = failcode then m := 0
                else begin
```

```
                    C-S-CHAIN-REC#(trm, preg, arity(xi), db7; m);
                    m := m +1
                    end
        else abort
     else abort
end;

C-S-CHAIN-TRY-ME#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if instr = try_me(N) then
       var m0 = 0 in begin
            C-S-CHAIN-REC#(trm, co +1, db7; m);
            C-S-CHAIN-RETRY-ME#(trm, N, db7; m0);
            m := (m + m0) +1
            end
    else abort
end;

C-S-CHAIN-TRY#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if instr = try(N) then
       var m0 = 0 in begin
            C-S-CHAIN-REC#(trm, N, db7; m);
            C-S-CHAIN-RETRY#(trm, co +1, db7; m0);
            m := (m + m0) +1
            end
    else abort
end;

C-S-CHAIN-RETRY-ME#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if instr = retry_me(N) then
       var m0 = 0 in begin
            C-S-CHAIN-REC#(trm, co +1, db7; m);
            C-S-CHAIN-RETRY-ME#(trm, N, db7; m0);
            m := (m + m0) +1
            end
    else if trust_me(instr) then
       begin C-S-CHAIN-REC#(trm, co +1, db7; m); m := m +1 end
    else abort
end;

C-S-CHAIN-RETRY#(trm, co, db7; var m)
begin
```

```
var instr = code(co, db7) in
    if instr = retry(N) then
      var m0 = 0 in begin
          C-S-CHAIN-REC#(trm, N, db7; m);
          C-S-CHAIN-RETRY#(trm, co +1, db7; m0);
          m := (m + m0) +1
          end
    else if instr = trust(N) then
      begin C-S-CHAIN-REC#(trm, N, db7; m); m := m +1 end
    else abort
end;

C-S-CHAIN-RET#(trm, co, db7; var m)
begin
var instr = code(co, db7) in
    if is_retry_me(instr) ∨ is_trust_me(instr) then
      C-S-CHAIN-RETRY-ME#(trm, co, db7; m)
    else if is_retry(instr) ∨ is_trust(instr) then
      C-S-CHAIN-RETRY#(trm, co, db7; m)
    else abort
end;

C-S-APP-CHAINS-RET#(decglseq', p, stl, db7; var m)
begin
if stl = [] then m := 0
else var m0 = 0 in begin
          C-S-CHAIN-RET#(acg[car(stl), p[car(stl)], db7; m);
          C-S-APP-CHAINS-RET#(decglseq', p, cdr(stl), db7; m0);
          m := (m + m0) +1
          end
end
```

Anhang E

Spezifikationen der Prolog-WAM-Fallstudie

E.1 Library-Spezifikationen

elem =
specification
  sorts elem;
  variables a, b, c : elem;
end specification

---

elemI =
rename elem **by morphism**
  elem → elem', a → a', b → b', c → c'
end rename

---

elemII =
rename elem **by morphism**
  elem → elem", a → a", b → b", c → c"
end rename

---

pair =
generic data specification
  parameter elemI + elemII
  pair = ⟨ . , . ⟩ (fst : elem', snd : elem");
  variables p, p_0, p_1 : pair;
end generic data specification

Generated axioms:

pair **freely generated by** $\langle\ .\ ,\ .\ \rangle$;
$\mathrm{fst}(\langle a', a''\rangle) = a'$,
$\mathrm{snd}(\langle a', a''\rangle) = a''$,
$\langle a', a''\rangle = \langle a'_0, a''_0\rangle \leftrightarrow a' = a'_0 \wedge a'' = a''_0$,
$\langle \mathrm{fst}(p), \mathrm{snd}(p)\rangle = p$

vartermpair =
actualize pair **with parameter** node, term **by morphism**
elem' → nodesort, elem" → term, pair → pairvarterm
end actualize

varvarpair =
actualize pair **with parameter** node **by morphism**
elem' → nodesort, elem" → nodesort, pair → varvarpair
end actualize

termtermpair =
actualize pair **with** term **by morphism**
elem' → term, elem" → term, pair → termtermpair
end actualize

decgoal =
actualize pair **with** goalsort, **parameter** node **by morphism**
elem' → goalsort, elem" → nodesort, pair → decgoal
end actualize

clause =
actualize pair **with** term, goal **by morphism**
elem' → term, elem" → goalsort,
pair → clausesort, p → cl
end actualize

ident =
actualize pair **with parameter** atom, nat **by morphism**
elem' → atomsort, elem" → nat, pair → ident
end actualize

procdeftable =
actualize pair **with** ident, **parameter** code **by morphism**
elem' → ident, elem" → codesort,
pair → procdeftable, p → pdt
end actualize

comp3result =

actualize pair
with parameter program2, procdeftable
by morphism
elem' → program", elem" → procdeftable, pair → comp3result
.1 →.db, .2 →.pdtab, p → co3res
end actualize

Dynfun =
generic specification
parameter sorts dom, codom;
target sorts dynfun;
functions cf : codom → dynfun;
. [.] : dynfun × dom → codom;
. [. ← .] : dynfun × dom × codom → dynfun;
variables f : dynfun; x, y : dom; z : codom;
axioms cf(z) [x] = z,
f [x ← z] [x] = z,
x ≠ y → f [x ← z] [y] = f[y]
end generic specification

F-no-no =
actualize Dynfun **with parameter** node **by morphism**
dom → nodesort, codom → nodesort,
Dynfun → funnodenode, f → F
end actualize

vi =
actualize Dynfun **with** nat **by morphism**
dom → nat, codom → nodesort, Dynfun → vifun, f → vi
end actualize

F-co-co =
actualize Dynfun **with parameter** code **by morphism**
dom → codesort, codom → codesort,
Dynfun → funcodecode, f → C
end actualize

cll =
actualize Dynfun **with parameter** node, **parameter** code **by morphism**
dom → nodesort, codom → codesort,
Dynfun → cllfun, f → cll
end actualize

decglseq =
actualize Dynfun **with** decgoallist **by morphism**

dom → nodesort, codom → decgoallist,
Dynfun → decgoalseqfun, f → decglseq
end actualize

cands =
actualize Dynfun **with** nodelist **by morphism**
dom → nodesort, codom → nodelist,
Dynfun → candsfun, f → cands
end actualize

p =
actualize Dynfun **with parameter** code **by morphism**
dom → nodesort, codom → codesort,
Dynfun → pfun, f → p
end actualize

cg =
actualize Dynfun **with** goal **by morphism**
dom → nodesort, codom → goal, Dynfun → cgfun, f → cg
end actualize

cp =
actualize Dynfun **with parameter** node, **parameter** code **by morphism**
dom → nodesort, codom → codesort,
Dynfun → cpfun, f → cp
end actualize

sub =
actualize Dynfun **with** substitution **by morphism**
dom → nodesort, codom → substitution,
Dynfun → subfun, f → sub
end actualize

goalfun =
actualize Dynfun **with parameter** node, goal **by morphism**
dom → nodesort, codom → goalsort,
Dynfun → goalfun, f → goal
end actualize

H-no-nol =
actualize Dynfun **with** nodelist **by morphism**
dom → nodesort, codom → nodelist,
Dynfun → funnodenodelist, f → H
end actualize

---

nat-basic1 =
data specification
nat = 0 | . +1 (. −1 : nat);
variables i, j, k, m : nat;
order predicates . < . : nat × nat;
end data specification

Generated axioms:

nat **freely generated by** 0, +1;
i +1 −1 = n,
i +1 = j +1 ↔ i = j,
0 ≠ i +1,
i = 0 ∨ i = i −1 +1,
¬ i < i,
i < j ∧ j < k → i < k,
¬ i < 0,
i < j +1 ↔ i = j ∨ i < j

---

nat =
enrich nat-basic1 **with**
functions . + . : nat × nat → nat;
. − . : nat × nat → nat **prio** 8 **left**;
predicates
. ≤ . : nat × nat;
. > . : nat × nat;
. ≥ . : nat × nat;

axioms

i + 0 = i,
i + j +1 = (i + j)+1,
i − 0 = i,
i − j +1 = (i − j)−1,
i ≤ j ↔ ¬ j < i,
i > j ↔ j < i,
i ≥ j ↔ ¬ i < j

end enrich

---

set =
generic specification
parameter elem **using** nat **target**

sorts set;
constants $\emptyset$: set;
functions
$\{ . \}$: elem $\rightarrow$ set;
$. \cup .$: set $\times$ set $\rightarrow$ set **prio** 9 **left**;
predicates
$. \in .$: elem $\times$ set;
$. \subseteq .$: set $\times$ set;
variables s, s' : set;

axioms

set **generated by** $\emptyset$, $\{ . \}$, $. \cup .$
$\neg$ a $\in \emptyset$,
a $\in \{b\} \leftrightarrow$ a = b,
a $\in$ s $\cup$ s' $\leftrightarrow$ a $\in$ s $\vee$ a $\in$ s',
s = s' $\leftrightarrow$ ($\forall$ a. a $\in$ s $\leftrightarrow$ a $\in$ s'),
s $\subseteq$ s' $\leftrightarrow$ ($\forall$ a. a $\in$ s $\rightarrow$ a $\in$ s')

end generic specification

nodeset =
actualize set **with parameter** node **by morphism**
elem $\rightarrow$ nodesort, set $\rightarrow$ nodeset
end actualize

---

list-data =
generic data specification
parameter elem **using** nat
list = []
| [. | .] (car : elem, cdr : list)
;
variables x, y, z : list;
size functions # : list $\rightarrow$ nat ;
order predicates . $\ll$. : list $\times$ list;
end generic data specification

Generated axioms:

list **freely generated by** [], [. | .]
car([a | x]) = a,
cdr([a | x]) = x,
[a | x] = [b | y] $\leftrightarrow$ a = b $\wedge$ x = y,
[] $\neq$ [a | x],
x = [] $\vee$ x = [car(x) | cdr(x)],
#([]) = 0,

#([a | x]) = #(x)+1,
¬ x ≪ x,
x ≪ y ∧ y ≪ z → x ≪ z,
¬ x ≪ [],
y ≪ [a | x] ↔ y = x ∨ y ≪ x

---

list =
enrich list-data **with**
functions

append	:	list × list	→	list;
rmdup	:	list	→	list;
pos	:	list × elem	→	nat;
rev	:	list	→	list;

predicates

. ∈ .	:	elem × list;
. subli .	:	list × list;
. subse .	:	list × list;
. ⊂ .	:	list × list;
dups	:	list;
nodups	:	list;

axioms

append([], x) = x,
append([a | x], y) = [a | append(x, y)],
a ∈ x ↔ (∃ y, z. x = append(y, [a | z]),
[] subli x,
¬ [a | x] subli [],
[a | x] subli [b | y] ↔ a = b ∧ x subli y ∨ a ≠ b ∧ [a | x] subli y,
[] subse x,
[a | x] subse y ↔ a ∈ y ∧ x subse y,
nodups([]),
nodups([a | x]) ↔ ¬ a ∈ x ∧ nodups(x),
dups(x) ↔ ¬ nodups(x),
rmdup([]) = [],
a ∈ x → rmdup([a | x]) = rmdup(x),
¬ a ∈ x → rmdup([a | x]) = [a | rmdup(x)],
x ⊂ y ↔ #(rmdup(x)) < #(rmdup(y)) ∧ x subse y
pos([a | x], a) = 0,
a ≠ b → pos([a | x], b) = pos(x, b) +1,
rev([]) = [],
rev([a | x]) = append(rev(x), [a])

end enrich

substitution =
actualize list **with** pairvarterm **by morphism**
elem → pairvarterm, list → substitution, x → su
end actualize

goal =
actualize list **with** term **by morphism**
elem → term, list → goal, x → go
end actualize

natlist =
actualize list **with** nat **by morphism**
elem → nat, list → natlist, x → nl
end actualize

varlist =
actualize list **with parameter** node **by morphism**
elem → nodesort, list → varlist, x → vl
end actualize

nodelist =
actualize list **with parameter** node **by morphism**
elem → nodesort, list → nodelist, x → stack
end actualize

codelist =
actualize list **with parameter** code **by morphism**
elem → codelist, list → codesort, x → col
end actualize

decgoallist =
actualize list **with** decgoal **by morphism**
elem → decgoal, list → decgoallist, x → dgl
end actualize

clauselist =
actualize list **with** clause **by morphism**
elem → clause, list → clauselist, x → cli

renaming =
actualize list **with** varvarpair **by morphism**
elem → varvarpair, list → renaming
end actualize

E.2 Spezifikationen für ASM1 (PrologTree)

enrnodeset =
enrich nodeset **with**
functions new : nodeset → elem;

axioms

¬ new(s) ∈ s, new([]) = ⊥

end enrich

---

mode =
data specification
modesort = select | call;
variables mode : modesort;
end data specification

Generated axioms:

modesort **freely generated by** select, call;
select ≠ call,
mode = select ∨ mode = call

---

stopmode =
data specification
stopmodesort = success | failure | run;
variables stop : stopmodesort;
end data specification

Generated axioms:

stopmodesort **freely generated by** success, failure, run;
failure ≠ run, success ≠ run, success ≠ failure,
stop = success ∨ stop = failure ∨ stop = run

---

node =
specification
sorts nodesort;
constants ⊥ : nodesort;
variables n : nodesort;

end specification

---

atom =
specification
sorts atomsort;
constants cutsym , failsym, truesym : atomsort;
variables at : atomsort;

axioms

cutsym $\neq$ failsym,
failsym $\neq$ truesym,
truesym $\neq$ cutsym

end specification

---

term =
data specification
using nat, **parameter** atom, **parameter** ordnode
term = struct (funct : atomsort, args : termlist) **with** is_struct
| mkconst (constsym : atomsort) **with** is_const
| mkvar (varsym : nodesort) **with** is_var
| mklist (lcar : term, lcdr : term) **with** is_list
;
termlist = the_one (and_only : term)
| tcons (tcar : term, tcdr : termlist)
;
variables trm, trm$_0$: term; trmli, trmli$_0$: termlist;
size functions tlen : termlist $\rightarrow$ nat ;
order predicates . $<_{tl}$. : termlist $\times$ termlist;
end data specification

Generated axioms:

term, termlist **freely generated by** struct, mkconst, mkvar, mklist, the_one, tcons;

$\vdots$

---

subst =
enrich decgoallist, ident, enrterm **with**
functions

. $\hat{}_d$.	:	decgoallist $\times$ substitution	$\rightarrow$	decgoallist;
. $\hat{}_{sg}$.	:	substitution $\times$ goalsort	$\rightarrow$	goalsort;
. $\hat{}_t$.	:	substitution $\times$ term	$\rightarrow$	term;
. $\hat{}_{tl}$.	:	substitution $\times$ termlist	$\rightarrow$	termlist;

axioms

su $\hat{}_{sg}$ [] = [],
su $\hat{}_{sg}$ [trm | go] = [su $\hat{}_t$ trm | su $\hat{}_{sg}$ go],
su $\hat{}_d$ [] = [],
su $\hat{}_d$ [⟨go, st⟩ | dgl] = [⟨su $\hat{}_{sg}$ go, st⟩ | su $\hat{}_d$ dgl],
su $\hat{}_t$ struct(at, trmli) = struct(at, su $\hat{}_{tl}$ trmli),
su $\hat{}_t$ mklist(trm, trm_0) = mklist(su $\hat{}_t$ trm, su $\hat{}_t$ trm_0),
[] $\hat{}_t$ mkvar(va) = mkvar(va),
[⟨va, trm⟩ | su] $\hat{}_t$ mkvar(va) = trm,
va ≠ va_0 → [⟨va_0, trm⟩ | su] $\hat{}_t$ mkvar(va) = su $\hat{}_t$ mkvar(va),
su $\hat{}_t$ mkconst(at) = mkconst(at),
su $\hat{}_{tl}$ the_one(trm) = the_one(su $\hat{}_t$ trm),
su $\hat{}_{tl}$ tcons(trm, trmli) = tcons(su $\hat{}_t$ trm, su $\hat{}_{tl}$ trmli),
[] o su = su,
[⟨va_0, trm⟩ | su] o su_0 = [⟨va_0, su_0 $\hat{}_t$ trm⟩ | su o su_0]

end enrich

---

substornil =
data specification
using subst
substornil = oksubst(the_subst : substitution) | nil;
variables subst : substornil;
end data specification

Generated axioms:

substornil **freely generated by** nil, oksubst;
the_subst(oksubst(su)) = su,
oksubst(su) = oksubst(su_0) ↔ su = su_0,
oksubst(su) ≠ nil,
subst = oksubst(the_subst(subst)) ∨ subst = nil

---

enrterm =
enrich term, substornil **with**
constants ! : term; true : term; fail : term;
functions

. ∘ .	:	substitution × substornil	→	substitution;
. o .	:	substitution × substitution	→	substitution;
arity	:	term	→	nat;
arg	:	term × nat	→	term;
. $\odot_{tl}$.	:	termlist × termlist	→	termlist;

predicates is_user_defined : term;
variables su, su_1, su_2 : substitution;

axioms

$\text{the\_one(trm)} \odot_{tl} \text{trmli} = \text{tcons(trm, trmli)}$,
$\text{tcons(trm, trmli)} \odot_{tl} \text{trmli}_1 = \text{tcons(trm, trmli} \odot_{tl} \text{trmli}_1)$,
$! = \text{mkconst(cutsym)}$,
$\text{true} = \text{mkconst(truesym)}$,
$\text{fail} = \text{mkconst(failsym)}$,
$\text{is\_user\_defined(trm)} \leftrightarrow \text{trm} \neq \text{true} \wedge \text{trm} \neq \text{fail} \wedge \text{trm} \neq !$,
$\text{arity(trm)} = \text{tlen(args(trm))}+1$,
$\text{args(trm)} = \text{the\_one}(\text{trm}_1) \rightarrow \text{arg(trm, 0 +1)} = \text{trm}_1$,
$\quad \text{args(trm)} = \text{tcons}(\text{trm}_1, \text{trmli})$
$\rightarrow \quad \text{arg(trm, 0 +1)} = \text{trm}_1$
$\quad \wedge\ (0 < \text{n} \rightarrow \text{arg(trm, n +1)} = \text{arg(struct(funct(trm), trmli), n))}$,
$\text{su} \circ \text{oksubst}(\text{su}_0) = \text{su o su}_0$

end enrich

---

unify =
enrich substornil **with**
functions unify : term × term → substornil;

end enrich

---

code =
specification
sorts codesort;
constants failcode : codesort;
functions
. +1 : codesort → codesort;
. −1 : codesort → codesort;
variables co : codesort;

axioms

$\text{co} +1 -1 = \text{co}$,
$\text{co} -1 +1 = \text{co}$

end specification

---

program =
specification
sorts program;
variables db : program;
end specification

---

union0 = mode + stopmode + unify + clauselist + rename + enrnodeset +
sub + cll + subst + F-no-no + decglseq + enrterm

---

clausefun =
enrich clause, **parameter** code, **parameter** program **with**
functions clause : codesort × program → clausesort;

end enrich

---

procdef =
enrich term, codelist, **parameter** program **with**
functions procdef : term × program → codelist ;

end enrich

PrologTree =
enrich union0 + cands + procdef + clausefun **with**
functions

mapclause	:	codelist × program	→	clauselist;
map	:	cllfun × nodelist	→	codelist;

predicates

every	:	funnodenode × nodelist × nodesort;
disjoint	:	nodelist × nodelist;
disjointls	:	nodelist × nodeset;

variables father: funnodenode;

axioms

mapclause([], db) = [],
mapclause([co | col], db) = [clause(co, db) | mapclause(col, db)],
every(father, [], n),
every(father, [n_1 | stack], n)
↔ father[n_1] = n ∧ every(father, stack, n),
map(cll, []) = [],
map(cll, [n | stack]) = [cll[n] | map(cll, stack)],
disjoint(stack, $stack_0$) ↔ (∀ n. n ∈ stack → ¬ n ∈ $stack_0$),
disjointls(stack, s) ↔ (∀ n. n ∈ stack → ¬ n ∈ s)

end enrich

E.3 Spezifikationen für ASM2 (TreeToStack)

procdef2 =
enrich term, **parameter** program, **parameter** code **with**
functions procdef_2 : term × program → codesort;

end enrich

---

clauseornull =
data specification
using clause
clauseornull = mkclau(the_clau : clausesort) | null;
variables cln : clauseornull;
end data specification

Generated axioms:

clauseornull **freely generated by** null, mkclau;
the_clau(mkclau(cl)) = cl,
mkclau(cl) = $\text{mkclau}(\text{cl}_0) \leftrightarrow \text{cl} = \text{cl}_0$,
mkclau(cl) ≠ null,
cln = mkclau(the_clau(cln)) ∨ cln = null

---

clauseIfun =
enrich code, clauseornull, program **with**
functions clause' : codesort × program → clauseornull;

axioms

clause'(failcode, db) = null

end enrich

PrologStack =
enrich union0 + procdef2 + codelist + nodelist + clauseIfun **with**
functions
. from . : nodelist × nodesort → nodelist **prio** 7;
cdr : nodelist → nodelist;
predicates
. cutptsin . : decgoallist × nodelist;
. ctpelem . : decgoallist × nodeset;
. ⊆ . : nodelist × nodeset;

axioms

mapclause'([], db) = [],
mapclause'([co | col], db) = [the_clau(clause'(co, db)) |
mapclause'(col, db)],
[] cutptsin stack,
  [⟨go, n⟩ | dgl] cutptsin stack
↔ (n = ⊥ ∨ n ∈ stack) ∧ dgl cutptsin (stack from n),
[] from n = [],
[n | stack] from n = [n | stack],
$n_1 \neq n \rightarrow [n_1 \mid stack]$ from n = stack from n,
[] ctpelem s,
[⟨go, n⟩ | dgl] ctpelem s ↔ n ∈ s ∧ dgl ctpelem s,
stack ⊆ s ↔ (∀ n. n ∈ stack → n ∈ s),
cdr([]) = [],
cdr([n | stack]) = stack

end enrich

CompAssum1 =
enrich PrologTree, PrologStack **with**
  functions $compile_{12}$: program → program;
  variables lit : term; db : program;

axioms

⟨CLLS#($procdef_2$(lit, $compile_{12}$(db)), $compile_{12}$(db); col)⟩
  mapclause(procdef(lit, db), db) = mapclause'(col, $compile_{12}$(db))

end enrich

Tree+Stack+F =
enrich F-no-no, PrologTree, PrologStack **with**
  functions
    F_d : funnodenode × decgoallist → decgoallist;
    F_s : funnodenode × nodeset → nodeset;
  predicates
    candsdisjoint : funnodenode × candsfun × nodelist;
    . injon . : funnodenode × nodelist;
    nocands : funnodenode × candsfun × nodelist;

axioms

F_d(F, []) = [],
F_d(F, [⟨go, n⟩ | dgl]) = [⟨go, F[n]⟩ | F_d(F, dgl)],
F_s(F, ∅) = ∅,
F_s(F, s ∪ {n}) = F_s(F, s) ∪ {F[n]},
  candsdisjoint(F, cands, stack)

$\leftrightarrow \forall$ n, n_1. n $\in$ stack $\wedge$ $n_1 \in$ stack $\wedge$ n $\neq$ n_1
$\rightarrow$ disjoint(cands[F[n_1]], cands[F[n]]),
F injon stack
$\leftrightarrow$ ($\forall$ n, n_1. n $\in$ stack $\wedge$ $n_1 \in$ [$\perp$ | stack] $\wedge$ n $\neq$ n_1 $\rightarrow$ F[n] $\neq$ F[n_1]),
nocands(F, cands, stack)
$\leftrightarrow \forall$ n, n_1. n $\in$ stack $\wedge$ $n_1 \in$ [$\perp$ | stack] $\rightarrow \neg$ F[n_1] $\in$ cands[F[n]]

end enrich

TreetoStack = CompAssum1 + Tree+Stack+F

E.4 Spezifikationen für ASM3 (ReuseChoicep)

rmode =
data specification
rmodesort = try | retry | enter | call;
variables rmode : rmodesort;
end data specification

Generated axioms:

rmodesort **freely generated by** try, retry, enter, call;
enter $\neq$ call, retry $\neq$ call, retry $\neq$ enter,
try $\neq$ call, try $\neq$ enter, try $\neq$ retry,
rmode = try $\vee$ rmode = retry $\vee$ rmode = enter $\vee$ rmode = call

---

PrologStack+F =
enrich F-no-no, Tree+Stack+F **with**
functions F_l : funnodenode $\times$ nodelist $\rightarrow$ nodelist;

axioms

F_l(F, []) = [],
F_l(F, [n | stack]) = [F[n] | F_l(F, stack)]

end enrich

ReuseChoicep = PrologStack+F + rmode

E.5 Spezifikationen für ASM4 (DetermDetect)

DetermDetect = PrologStack+F + rmode

E.6 Spezifikationen für ASM5 (CompPredStruct)

instr+clau =
data specification
using nat, clause, varlist, **parameter** code
instr-or-cl = try_me_else (where : codesort) **with** is_try_me
| retry_me_else (where : codesort) **with** is_retry_me
| trust_me **with** is_trust_me
| try (what : codesort) **with** is_try
| retry (what : codesort) **with** is_retry
| trust (what : codesort) **with** is_trust
| switch_on_term (argindex : nat,
vlabel : codesort, clabel : codesort,
llabel : codesort, slabel : codesort)
with is_sw_term
| switch_on_constant (argindex : nat,
tabsize : nat, table : codesort)
with is_sw_const
| switch_on_structure (argindex : nat,
tabsize : nat, table : codesort)
with is_sw_struct
| mkcl (the_cl : clausesort) **with** is_clause
| mkcall (calllit : term) **with** is_call
| mkunify (unifylit : term) **with** is_unify
| allocate
| deallocate
| proceed
| null
| code_of_start
;
variables ioc : instr-or-cl;
end data specification

Generated axioms:

instr-or-cl **freely generated by** trust_me, allocate, deallocate, proceed, nil', code_of_start, try_me_else, retry_me_else, try', retry', trust, switch_on_term, switch_on_constant, switch_on_structure, mkcl, mkcall, mkunify;

⋮

---

procdef3 =
enrich term, **parameter** program2, **parameter** code **with**
functions procdef$_3$: term × program" → codesort;

end enrich

---

codefun =
enrich parameter code, **parameter** program2, instr+clau **with**
constants start : codesort;
functions code : codesort × program" → instr-or-cl;

axioms

co = start ↔ code(co, db$_2$) = code_of_start,
code(failcode, db$_2$) = nil'

end enrich

CompAssum2 =
enrich CompAssum1, instr+clau, codefun, procdef3 **with**
functions

compile$_{45}$	:	program	→	program";
mapcode	:	codelist × program"	→	clauselist;

variables lit : term; db$_2$: program; db$_5$: program";

axioms

mapcode([], db$_5$) = [],
mapcode([co | col], db$_5$) = [the_cl(code(co, db$_5$)) | mapcode(col, db$_5$)],
[CLLS#(procdef$_2$(lit, db$_2$), db$_2$; col$_1$)]
⟨CHAIN-FL#(procdef$_3$(lit, compile$_{45}$(db$_2$)), compile$_{45}$(db$_2$); col$_2$)⟩
mapcode(col$_2$, compile$_{45}$(db$_2$)) = mapclause'(col$_1$, db$_2$)

end enrich

CompPredStruct = CompAssum2 + PrologStack+F + rmode + p

E.7 Spezifikationen für ASM6 (CompPredStruct2)

hash =
enrich nat, **parameter** atom,
parameter code, **parameter** program2 **with**
functions
hashc : codesort × nat × atomsort × program" → codesort;
hashs : codesort × nat × atomsort × nat × program" → codesort;

end enrich

CompAssum3a =
enrich CompAssum2, p, hash **with**
functions compile_{56} : program" → program";

axioms

[CHAIN-FL#(procdef_2(lit, db_5), db_5; col_1)]
⟨CHAIN#(procdef_3(lit, compile_{56}(db_5)), compile_{56}(db_5); col_2)⟩
mapcode(col_1, db_5) = mapcode(col_2, compile_{56}(db_5))

end enrich

CompPredStruct2 = CompAssum3a + PrologStack+H + p

E.8 Spezifikationen für ASM7 (Switching)

idfun =
enrich enrterm, ident **with**
functions id : term → ident;

axioms

is_struct(trm) → id(trm) = mkident(funct(trm), arity(trm)),
is_const(trm) → id(trm) = mkident(constsym(trm), 0)

end enrich

CompAssum3 =
enrich comp3result, CompAssum2, p, hash, idfun **with**
functions compile_{57} : program" → comp3result;

axioms

[CHAIN-FL#(procdef$_2$(lit, db$_5$), db$_5$; col$_1$)]
⟨S-CHAIN#(lit, compile$_{57}$(db$_5$).pdt[id(lit)], compile$_{57}$(db$_5$).db; col$_2$)⟩
mapcode(col$_1$, db$_5$) = mapcode(col$_2$, compile$_{57}$(db$_5$).db)

end enrich

PrologStack+H =
enrich PrologStack, H-no-nol **with**
functions

H_d	:	funnodenodelist × decgoallist	→	decgoallist;
H_l	:	funnodenodelist × nodelist	→	nodelist;
car	:	nodelist	→	nodesort;

axioms

H_d(h, []) = [],
H_d(h, [⟨go, n⟩ | dgl] = [⟨go, car(h[n])⟩ | H_d(h, dgl)],
H_l(h, []) = [],
H_l(h, [n | stack]) = append(h[n], H_l(h, stack)),
car([]) = ⊥,
car([n | stack]) = n

end enrich

Switching =
enrich CompAssum3, PrologStack+H, p **with**
functions . $-_{sl}$. : nodelist × nodelist → nodelist;
predicates
eqh : funnodenodelist × funnodenodelist × decgoallist × decgoallist;
. $<=_s$. : nodelist × nodelist;

axioms

eqh(h, h$_0$, [], []),
¬ eqh(h, h$_0$, [⟨go, n⟩ | dgl], []),
¬ eqh(h, h$_0$, [], [⟨go$_0$, n$_0$⟩ | dgl$_0$]),
eqh(h, h$_0$, [⟨go, n⟩ | dgl], [⟨go$_0$, n$_0$⟩ | dgl$_0$])
↔ go = go$_0$
∧ (n = ⊥ ⊃ n$_0$ ∈ h$_0$[⊥] ∨ n$_0$ = ⊥
; n$_0$ ∈ h$_0$[n] ∧ ¬ n$_0$ ∈ cdr(h[n]))
∧ eqh(h, h$_0$, dgl, dgl$_0$),
stack $<=_s$ stack$_0$ ↔ stack $\ll_s$ stack$_0$ ∨ stack = stack$_0$,
stack $<=_s$ stack$_0$ → (stack$_0$ $-_{sl}$ stack) $\odot_{sl}$ stack = stack$_0$

end enrich

E.9 Spezifikationen für ASM8 (ShareCont)

ordnode =
enrich parameter node **with**
functions

. +1	:	nodesort	→	nodesort;
. −1	:	nodesort	→	nodesort;
max	:	nodesort × nodesort	→	nodesort;

predicates . ≪ . : nodesort × nodesort;

axioms

$n +1 -1 = n$,
$n -1 +1 = n$,
$n \ll n +1$,
$\neg\ n \ll n$,
$n_1 \ll n_2 \vee n_1 = n_2 \vee n_2 \ll n_1$,
$n \ll n_0 \wedge n_0 \ll n_1 \rightarrow n \ll n_1$,
$n_1 \ll n_2 \rightarrow \max(n_1, n_2) = n_2$,
$\neg\ n_1 \ll n_2 \rightarrow \max(n_1, n_2) = n_1$

end enrich

---

rensubst =
enrich substitution, renaming **with**
functions

. ^$_r$.	:	renaming × term	→	term;
. ^$_{rl}$.	:	renaming × termlist	→	termlist;

axioms

rn ^$_r$ struct(at, trmli) = struct(at, rn ^$_{rl}$ trmli),
rn ^$_r$ mklist(trm, trm$_0$) = mklist(rn ^$_r$ trm, rn ^$_r$ trm$_0$),
[] ^$_r$ mkvar(va) = mkvar(va),
[⟨va$_1$,va$_2$⟩ | rn] ^$_r$ mkvar(va$_1$) = mkvar(va$_2$),
va ≠ va$_1$ → [⟨va$_1$,va$_2$⟩ | rn] ^$_r$ mkvar(va) = rn ^$_r$ mkvar(va),
rn ^$_r$ mkconst(at) = mkconst(at),
rn ^$_{rl}$ the_one(trm) = the_one(rn ^$_r$ trm),
rn ^$_{rl}$ tcons(trm, trmli) = tcons(rn ^$_r$ trm, rn ^$_{rl}$ trmli)

end enrich

---

less-vi =
enrich subst, vi, varlist, actrenterm, unify, rename **with**
functions

rentl : termlist × nat → termlist;
rentl' : termlist × nodesort × vifun → termlist;
rent' : term × nodesort × vifun → term;
reng' : goalsort × nodesort × vifun → goalsort;
renv : nodesort × nat → nodesort;

predicates

$. <_{svi} .$: substitution × nat;
$. <_{tvi} .$: term × nat;
$. <_{tlvi} .$: termlist × nat;
$. <_{gvi} .$: goalsort × nat;
$. <_{dvi} .$: decgoallist × nat;
$. <_{cvi} .$: clausesort × nat;
$. <_{vvi} .$: nodesort × nat;
$. <_{vlvi} .$: varlist × nat;

variables lit : term;

axioms

$su <_{svi} i \wedge su_0 <_{svi} i \rightarrow su \circ su_0 <_{svi} i,$
$su <_{svi} i \rightarrow su \hat{}_t rent(trm, i) = rent(trm, i),$
$trm <_{tvi} i \wedge trm_1 <_{tvi} i \wedge unify(trm, trm_1) \neq nil$
$\rightarrow the\_subst(unify(trm, trm_1)) <_{svi} i,$
$trm <_{tvi} i \wedge i < j \rightarrow trm <_{tvi} j,$
$trm <_{tvi} 0 \rightarrow rent(trm, i) <_{tvi} i + 1,$
$[] <_{gvi} i,$
$[trm \mid go] <_{gvi} i \leftrightarrow trm <_{tvi} i \wedge go <_{gvi} i,$
$[] <_{dvi} i,$
$[\langle go, ctpt\rangle \mid dgl] <_{dvi} i \leftrightarrow go <_{gvi} i \wedge dgl <_{dvi} i,$
$\langle lit, go\rangle <_{cvi} i \leftrightarrow lit <_{tvi} i \wedge go <_{gvi} i,$
$[] <_{svi} i,$
$[\langle va_0, trm\rangle \mid su] <_{svi} i$
$\leftrightarrow mkvar(va_0) <_{tvi} i \wedge trm <_{tvi} i \wedge su <_{svi} i,$
$rent(mkvar(va), i) = mkvar(renv(va, i)),$
$va <_{vvi} 0 \rightarrow \neg\, renv(va, i) <_{vvi} i,$
$rent(mkconst(at), i) = mkconst(at),$
$rent(struct(at, trmli), i) = struct(at, rentl(trmli, i)),$
$rent(mklist(trm, trm_0), i) = mklist(rent(trm, i), rent(trm_0, i)),$
$rentl(the\_one(trm), i) = the\_one(rent(trm, i)),$
$rentl(tcons(trm, trmli), i) = tcons(rent(trm, i), rentl(trmli, i)),$
$the\_one(trm) <_{tlvi} i \leftrightarrow trm <_{tvi} i,$
$tcons(trm, trmli) <_{tlvi} i \leftrightarrow trm <_{tvi} i \wedge trmli <_{tlvi} i,$
$struct(at, trmli) <_{tvi} i \leftrightarrow trmli <_{tlvi} i,$
$mkconst(at) <_{tvi} i,$
$mklist(trm, trm_0) <_{tvi} i \leftrightarrow trm <_{tvi} i \wedge trm_0 <_{tvi} i,$
$mkvar(va) <_{tvi} i \leftrightarrow va <_{vvi} i,$
$[] <_{vlvi} i,$
$[va \mid vl] <_{vlvi} i \leftrightarrow va <_{vvi} i \wedge vl <_{vlvi} i,$

va $<_{vvi}$ 0 ∧ va_0 $<_{vvi}$ 0
→ (renv(va, i) = renv(va_0, j) ↔ va = va_0 ∧ i = j),
rentl'(trmli, ctpt, vi)
= (ctpt ≠ ⊥ ⊃ rentl(trmli, vi[ctpt]) ; trmli),
rent'(trm, ctpt, vi)
= (ctpt ≠ ⊥ ⊃ rent(trm, vi[ctpt]) ; trm),
reng'(go, ctpt, vi)
= (ctpt ≠ ⊥ ⊃ reng(go, vi[ctpt]) ; go)

end enrich

RenAssum =
enrich CompAssum3, less-vi **with**
predicates

. $<_{clvi}$.	:	clauselist, nat;
nonvargoal	:	goalsort;

axioms

mapclause(procdef(lit,db),db) $<_{clvi}$ 0,
[] $<_{clvi}$ i,
[cl | cli] $<_{clvi}$ i ↔ cl $<_{cvi}$ i ∧ nonvargoal(bdy(cl)) ∧ cli $<_{clvi}$ i,
nonvargoal([]),
nonvargoal([trm | go])
↔ ¬ is_var(trm) ∧ ¬ is_list(trm) ∧ trm $<_{tvi}$ 0 ∧ nonvargoal(go)

end enrich

---

rename =
enrich nat, clause **with**
functions

ren	:	clausesort × nat	→	clausesort;
rent	:	term × nat	→	term;
reng	:	goalsort × nat	→	goalsort;

axioms

ren(mkclause(trm, go), i) = mkclause(rent(trm, i), reng(go, i)),
reng([], i) = [],
reng([trm | go], i) = [rent(trm, i) | reng(go, i)]

end enrich

ShareCont =
enrich parameter ordnode, cg, PrologStack+F,
goalfun, RenAssum **with**

functions
decglseqof : funnodenode × cgfun × funnodenode × nodelist
→ decgoallist;
predicates
ordered : nodelist;

axioms

decglseqof(cutpt, cg, ce, []) = [],
decglseqof(cutpt, cg, ce, [n | stack])
= [⟨cg[n], cutpt[ce[n]]⟩ | decglseqof(cutpt, cg, ce, stack)],
ordered([]),
ordered([n]) ↔ ⊥ ≪ n,
ordered([n | n_0 | stack]) ↔ n_0 ≪ n ∧ ordered([n_0 | stack])

end enrich

E.10 Spezifikationen für ASM9 (CompClause)

comp4result =
data specification
using procdeftable, **parameter** program2
comp4result = mkco4res (. .pc : codesort, . .pdtab : procdeftable,
. .dbc : program");
variables co4res : comp4result;
end data specification

Generated axioms:

comp4result **freely generated by** mkco4res;
mkco4res(co, procdeftab, db_7).pc = co,
mkco4res(co, procdeftab, db_7).pdtab = procdeftab,
mkco4res(co, procdeftab, db_7).dbc = db_7,
mkco4res(co, procdeftab, db_7) = mkco4res(co_0, $procdeftab_0$, db'_7)
↔ co = co_0 ∧ procdeftab = $procdeftab_0$ ∧ db_7 = db'_7,
mkco4res(co4res.pc, co4res.pdtab, co4res.dbc) = co4res

CompAssum4 =
enrich CompAssum3, clauselist, comp4result, F-co-co, RenAssum **with**
functions $compile_{89}$: comp3result × goalsort → comp4result ;
predicates
eqpdt : procdeftable × procdeftable × funcodecode;
eqcode : program" × program" × funcodecode;
variables pdtab : procdeftable; query, goalreg : goalsort;

axioms

$\langle db_7, procdef_7\rangle = compile_{57}(compile_{45}(compile_{12}(db)))$
$\rightarrow \exists\ C.\quad eqpdt(procdef_7, compile_{89}(\langle db_7, procdef_7\rangle, goalreg).pdtab, C)$
$\wedge\ eqcode(db_7, compile_{89}(\langle db_7, procdef_7\rangle, goalreg).dbc, C),$

$eqpdt(pdtab_0, pdtab, C) \leftrightarrow \forall\ lit.\ pdtab[id(lit)] = C[pdtab_0[id(lit)]],$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = mkcl(cl_0)$
$\rightarrow \langle UNLOAD\#(C[co], db_9; cl)\rangle\ cl = cl_0$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = try\_me\_else(N)$
$\rightarrow code(C[co], db_9) = try\_me\_else(C[N])$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = retry\_me\_else(N)$
$\rightarrow code(C[co], db_9) = retry\_me\_else(C[N])$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = trust\_me$
$\rightarrow code(C[co], db_9) = trust\_me$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = try(N)$
$\rightarrow code(C[co], db_9) = try(C[N])$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = retry(N)$
$\rightarrow code(C[co], db_9) = retry(C[N])$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = trust(N)$
$\rightarrow code(C[co], db_9) = trust(C[N])$

$eqcode(db_7, db_9, C) \wedge code(co,db_7) = failcode$
$\rightarrow code(C[co], db_9) = failcode$

$eqcode(db_7, db_9, C)$
$\wedge\ code(co,db_7) = switch\_on\_term(argindex, N_s, N_c, N_v, N_l))$
$\rightarrow code(C[co], db_9) = switch\_on\_term(argindex, C[N_s], C[N_c], C[N_v], C[N_l])$

$eqcode(db_7, db_9, C)$
$\wedge\ code(co,db_7) = switch\_on\_constant(argindex, tabsize, co)$
$\rightarrow \exists\ co_0.\quad code(C[co], db_9) = switch\_on\_constant(argindex, tabsize, co_0)$
$\wedge\ \forall\ at.\quad C[hashc(co, tabsize, at, db_7)]$
$= hashc(co_0, tabsize, at, db_9))$

$compile_{57}(compile_{45}(compile_{12}(db))) = \langle db_7, procdef_7\rangle$
$\wedge\ nonvargoal(goalreg)$
$\rightarrow \langle QUERY\#(compile_4(db_9, goalreg).pc, compile_4(db_9, goalreg).dbc; go)\rangle$
$go = goalreg$

end enrich

CompClause = CompAssum4 + ShareCont + cp

E.11 Spezifikationen für ASM9a (Renaming)

termvarli =
enrich varlist, enrterm **with**
functions

tvarli	:	term	→	varlist;
tlvarli	:	termlist	→	varlist;

axioms

tvarli(mkconst(at)) = [],
tvarli(mkvar(va)) = [va | []],
tvarli(mklist(trm, trm_1)) = rmdup(append(tvarli(trm), tvarli(trm_1))),
tvarli(struct(at, trmli)) = tlvarli(trmli),
tlvarli(the_one(trm)) = tvarli(trm),
tlvarli(tcons(trm, trmli)) = rmdup(append(tvarli(trm), tlvarli(trmli))

end enrich

---

ren =
enrich natlist, termvarli, less-vi, nodelist **with**
functions

dom	:	renaming	→	varlist;
codom	:	renaming	→	varlist;
. $\hat{}_{rv}$.	:	renaming × nodesort	→	nodesort **prio** 9;
vilist	:	vifun × nodelist	→	natlist;

predicates . $<_{nl}$. : natlist × nat;

axioms

dom([]) = [],
dom([⟨va,va_1⟩ | rn]) = [va | dom(rn)],
codom([]) = [],
codom([⟨va,va_1⟩ | rn]) = [va_1 | codom(rn)],
[] $\hat{}_{rv}$ va = va,
[⟨va_1 | va_2⟩, rn] $\hat{}_{rv}$ va_1 = va_2,
va ≠ va_1 → [⟨va_1,va_2⟩ | rn] $\hat{}_{rv}$ va = rn $\hat{}_{rv}$ va,
vilist(vi, []) = [],
vilist(vi, [st | stl]) = [vi[st] | vilist(vi, stl)],
[] $<_{nl}$ n,
[m | nl] $<_{nl}$ n ↔ m < n ∧ nl $<_{nl}$ n

end enrich

---

goalvarli =
enrich Renstack, clause **with**
functions

gvarli	:	goalsort	→	varlist;
clvarli	:	clausesort	→	varlist;

axioms

gvarli([]) = [],
gvarli([trm | go]) = rmdup(append(tvarli(trm), gvarli(go))),
clvarli(⟨trm,go⟩) = rmdup(append(tvarli(trm), gvarli(go)))

end enrich

---

enrunify =
enrich subst, unify, termtermpair, termvarli, Renstack **with**
functions

unifylist	:	termlist × termlist	→	substornil;
$\#_t$.	:	term	→	nat;
$\#_{tl}$.	:	termlist	→	nat;
suv	:	substitution	→	varlist;
sudom	:	substitution	→	varlist;
sucod	:	substitution	→	varlist;
. $\hat{}_{rs}$.	:	renaming × substitution	→	substitution **prio** 9;
. $\hat{}_{rsf}$.	:	renaming × substornil	→	substornil **prio** 9;
remove	:	substitution × nat	→	substitution;

predicates
(: Terminierungsordnung für unify :)

. ≪ .	:	termtermpair × termtermpair;
occurs	:	nodesort × term;
occurslist	:	nodesort × termlist;
disj	:	varlist × varlist;

variables trmli, trmli_1 : termlist; ttp, ttp_1 : termtermpair;

axioms

remove([], i) = [],
remove([⟨va,trm⟩ | su], i)
= (va $<_{vvi}$ (i +1) ∧ va $<_{vvi}$ i ⊃ remove(su, n); [⟨va,trm ⟩ | remove(su,n)]),
tlen(trmli) = tlen(trmli_1)
→ unify(struct(at, trmli), struct(at, trmli_1)) = unifylist(trmli, trmli_1),
at ≠ at_1 → unify(struct(at, trmli), struct(at_1, trmli_1)) = nil,
tlen(trmli) ≠ tlen(trmli_1)
→ unify(struct(at, trmli), struct(at_1, trmli_1)) = nil,
unify(mklist(trm, trm_0), mklist(trm_1, trm_2))
= unifylist(tcons(trm, the_one(trm_0)), tcons(trm_1, the_one(trm_2))),

$$
\begin{array}{l}
\mathrm{at} \neq \mathrm{at}_1 \rightarrow \mathrm{unify}(\mathrm{mkconst}(\mathrm{at}),\ \mathrm{mkconst}(\mathrm{at}_1)) = \mathrm{nil},\\
\mathrm{unify}(\mathrm{mkconst}(\mathrm{at}),\ \mathrm{mkconst}(\mathrm{at})) = \mathrm{oksubst}([]),\\
\quad \mathrm{unify}(\mathrm{mkvar}(\mathrm{va}),\ \mathrm{trm})\\
= (\mathrm{occurs}(\mathrm{va},\ \mathrm{trm}) \supset \mathrm{nil};\ \mathrm{oksubst}([\langle \mathrm{va},\mathrm{trm}\rangle \mid []])),\\
\quad \neg\ \mathrm{is\_var}(\mathrm{trm})\\
\rightarrow \quad \mathrm{unify}(\mathrm{trm},\ \mathrm{mkvar}(\mathrm{va}))\\
\quad = (\mathrm{occurs}(\mathrm{va},\ \mathrm{trm}) \supset \mathrm{nil};\ \mathrm{oksubst}([\langle \mathrm{va},\mathrm{trm}\rangle \mid []])),\\
\neg\ \mathrm{is\_var}(\mathrm{trm}) \wedge \neg\ \mathrm{is\_const}(\mathrm{trm}) \rightarrow \mathrm{unify}(\mathrm{mkconst}(\mathrm{at}),\ \mathrm{trm}) = \mathrm{nil},\\
\neg\ \mathrm{is\_var}(\mathrm{trm}) \wedge \neg\ \mathrm{is\_list}(\mathrm{trm}) \rightarrow \mathrm{unify}(\mathrm{mklist}(\mathrm{trm}_0,\ \mathrm{trm}_1),\ \mathrm{trm}) = \mathrm{nil},\\
\neg\ \mathrm{is\_var}(\mathrm{trm}) \wedge \neg\ \mathrm{is\_struct}(\mathrm{trm}) \rightarrow \mathrm{unify}(\mathrm{struct}(\mathrm{at},\ \mathrm{trmli}),\ \mathrm{trm}) = \mathrm{nil},\\
\neg\ \mathrm{is\_var}(\mathrm{trm}) \wedge \neg\ \mathrm{is\_const}(\mathrm{trm}) \rightarrow \mathrm{unify}(\mathrm{trm},\ \mathrm{mkconst}(\mathrm{at})) = \mathrm{nil},\\
\neg\ \mathrm{is\_var}(\mathrm{trm}) \wedge \neg\ \mathrm{is\_list}(\mathrm{trm}) \rightarrow \mathrm{unify}(\mathrm{trm},\ \mathrm{mklist}(\mathrm{trm}_0,\ \mathrm{trm}_1)) = \mathrm{nil},\\
\neg\ \mathrm{is\_var}(\mathrm{trm}) \wedge \neg\ \mathrm{is\_struct}(\mathrm{trm}) \rightarrow \mathrm{unify}(\mathrm{trm},\ \mathrm{struct}(\mathrm{at},\ \mathrm{trmli})) = \mathrm{nil},\\
\mathrm{occurs}(\mathrm{va},\ \mathrm{struct}(\mathrm{at},\ \mathrm{trmli})) \leftrightarrow \mathrm{occurslist}(\mathrm{va},\ \mathrm{trmli}),\\
\mathrm{occurs}(\mathrm{va},\ \mathrm{mklist}(\mathrm{trm},\ \mathrm{trm}_1)) \leftrightarrow \mathrm{occurs}(\mathrm{va},\ \mathrm{trm}) \vee \mathrm{occurs}(\mathrm{va},\ \mathrm{trm}_1),\\
\mathrm{occurs}(\mathrm{va},\ \mathrm{mkvar}(\mathrm{va}_0)) \leftrightarrow \mathrm{va} = \mathrm{va}_0,\\
\neg\ \mathrm{occurs}(\mathrm{va},\ \mathrm{mkconst}(\mathrm{at})),\\
\mathrm{occurslist}(\mathrm{va},\ \mathrm{the\_one}(\mathrm{trm})) \leftrightarrow \mathrm{occurs}(\mathrm{va},\ \mathrm{trm}),\\
\mathrm{occurslist}(\mathrm{va},\ \mathrm{tcons}(\mathrm{trm},\ \mathrm{trmli})) \leftrightarrow \mathrm{occurs}(\mathrm{va},\ \mathrm{trm}) \vee \mathrm{occurslist}(\mathrm{va},\ \mathrm{trmli}),\\
\mathrm{unifylist}(\mathrm{the\_one}(\mathrm{trm}),\ \mathrm{the\_one}(\mathrm{trm}_1)) = \mathrm{unify}(\mathrm{trm},\ \mathrm{trm}_1),\\
\quad \mathrm{unify}(\mathrm{trm},\ \mathrm{trm}_1) = \mathrm{nil}\\
\rightarrow \mathrm{unifylist}(\mathrm{tcons}(\mathrm{trm},\ \mathrm{trmli}),\ \mathrm{tcons}(\mathrm{trm}_1,\ \mathrm{trmli}_1)) = \mathrm{nil},\\
\qquad \mathrm{unify}(\mathrm{trm},\ \mathrm{trm}_1) = \mathrm{oksubst}(\mathrm{su})\\
\quad \wedge \mathrm{unifylist}(\mathrm{su}\ \hat{}_{tl}\ \mathrm{trmli},\ \mathrm{su}\ \hat{}_{tl}\ \mathrm{trmli}_1) = \mathrm{oksubst}(\mathrm{su}_1)\\
\rightarrow \mathrm{unifylist}(\mathrm{tcons}(\mathrm{trm},\ \mathrm{trmli}),\ \mathrm{tcons}(\mathrm{trm}_1,\ \mathrm{trmli}_1)) = \mathrm{oksubst}(\mathrm{su}\ \mathrm{o}\ \mathrm{su}_1),\\
\qquad \mathrm{unify}(\mathrm{trm},\ \mathrm{trm}_1) = \mathrm{oksubst}(\mathrm{su})\\
\quad \wedge \mathrm{unifylist}(\mathrm{su}\ \hat{}_{tl}\ \mathrm{trmli},\ \mathrm{su}\ \hat{}_{tl}\ \mathrm{trmli}_1) = \mathrm{nil}\\
\rightarrow \mathrm{unifylist}(\mathrm{tcons}(\mathrm{trm},\ \mathrm{trmli}),\ \mathrm{tcons}(\mathrm{trm}_1,\ \mathrm{trmli}_1)) = \mathrm{nil},\\
\quad \mathrm{ttp} \ll \mathrm{ttp}_1\\
\leftrightarrow \qquad \#(\mathrm{rmdup}(\mathrm{tvarli}(\mathrm{mklist}(\mathrm{ttp.t1},\ \mathrm{ttp.t2}))))\\
\qquad < \#(\mathrm{rmdup}(\mathrm{tvarli}(\mathrm{mklist}(\mathrm{ttp}_1\mathrm{.t1},\ \mathrm{ttp}_1\mathrm{.t2}))))\\
\quad \vee \qquad \#(\mathrm{rmdup}(\mathrm{tvarli}(\mathrm{mklist}(\mathrm{ttp.t1},\ \mathrm{ttp.t2}))))\\
\qquad = \#(\mathrm{rmdup}(\mathrm{tvarli}(\mathrm{mklist}(\mathrm{ttp}_1\mathrm{.t1},\ \mathrm{ttp}_1\mathrm{.t2}))))\\
\qquad \wedge \#_t(\mathrm{ttp.t1}) < \#_t(\mathrm{ttp}_1\mathrm{.t1}),\\
\#_t(\mathrm{mkconst}(\mathrm{at})) = 1,\\
\#_t(\mathrm{mkvar}(\mathrm{va})) = 1,\\
\#_t(\mathrm{struct}(\mathrm{at},\ \mathrm{trmli})) = \#_{tl}(\mathrm{trmli})+1,\\
\#_t(\mathrm{mklist}(\mathrm{trm},\ \mathrm{trm}_0)) = \#_t(\mathrm{trm}) + \#_t(\mathrm{trm}_0)+1,\\
\#_{tl}(\mathrm{the\_one}(\mathrm{trm})) = \#_t(\mathrm{trm}),\\
\#_{tl}(\mathrm{tcons}(\mathrm{trm},\ \mathrm{trmli})) = \#_t(\mathrm{trm}) + \#_{tl}(\mathrm{trmli}),\\
\mathrm{suv}([]) = [],\\
\mathrm{suv}([\langle \mathrm{va},\mathrm{trm}\rangle \mid \mathrm{su}]) = [\mathrm{va} \mid \mathrm{append}(\mathrm{tvarli}(\mathrm{trm}),\ \mathrm{suv}(\mathrm{su}))],\\
\mathrm{sudom}([]) = [],\\
\mathrm{sudom}([\langle \mathrm{va},\mathrm{trm}\rangle \mid \mathrm{su}]) = [\mathrm{va} \mid \mathrm{sudom}(\mathrm{su})],\\
\mathrm{sucod}([]) = [],\\
\mathrm{sucod}([\langle \mathrm{va},\mathrm{trm}\rangle \mid \mathrm{su}]) = \mathrm{append}(\mathrm{tvarli}(\mathrm{trm}),\ \mathrm{sucod}(\mathrm{su})),
\end{array}
$$

disj(vl, vl_0) $\leftrightarrow$ ($\forall$ va. va $\in$ vl $\rightarrow$ $\neg$ va $\in$ vl_0),
rn $\hat{}_{rs}$ [] = [],
rn $\hat{}_{rs}$ [$\langle$va,trm$\rangle$ | su] = [$\langle$rn $\hat{}_{rv}$ va,rn $\hat{}_{r}$ trm$\rangle$ | rn $\hat{}_{rs}$ su],
rn $\hat{}_{rsf}$ nil = nil,
rn $\hat{}_{rsf}$ oksubst(su) = oksubst(rn $\hat{}_{rs}$ su)

end enrich

Renaming = goalvarli + enrunify + CompClause

Literaturverzeichnis

[Ahr95] W. Ahrendt. Von PROLOG zur WAM — Verifikation der Prozedurübersetzung mit KIV. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, December 1995. (in German).

[AK91] H. Aït-Kaci. *Warren's Abstract Machine. A Tutorial Reconstruction*. MIT Press, 1991.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2:253–284, May 1991. Also appeared as SRC Research Report 29.

[Aus98] V. R. Austel. Towards computer-verified proofs of correctness of logic-programming interpreters using derivations. Technical Report 980022, University of California, Los Angeles, Computer Science Department, May 1998.

[Bae90] J. C. M. Baeten. Applications of process algebra. In *Theoretical Computer Science*, number 17 in Cambridge Tracts. Cambridge University Press, 1990.

[BD96] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.

[BDvH+96] F. Bartels, A. Dold, F. W. v. Henke, H. Pfeifer, and H. Rueß. Formalizing fixed-point theory in pvs. Technical Report Technical report UIB 96-10, Universität Ulm, Fakultät für Informatik, 1996.

[BG95] E. Börger and U. Glässer. Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger, editors, *Evolving Algebras – Mini-Course, BRICS Technical Report (BRICS-NS-95-4)*, pages 128–153. University of Aarhus, Denmark, July 1995.

[BGR95] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64, February 1998.

[BHMY89] W. R. Bevier, W. A. Jr. Hunt, J S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[BLH93] D. Bjørner, H. Langmaack, and C. A. R. Hoare. ProCoS I final deliverable. ProCoS Technical Report [ID/DTH DB 13/1], Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark, January 1993.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[BM96] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1996.

[BR94] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[BR95] E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In Christoph Beierle and Lutz Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*. North-Holland, Amsterdam, 1995.

[BS98a] E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1998.

[BS98b] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[CoF97] CoFI: The Common Framework Initiative. Casl — the CoFI algebraic specification language tentative design: Language summary, 1997. http://www.brics.dk/Projects/CoFI.

[Cyr93] D. Cyrluk. Microprocessor verification in pvs: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, December 1993.

[Dol98] A. Dold. A formal representation of abstract state machines using pvs. Verifix-Report Ulm 6.2, Universität Ulm, 1998. (revised version).

[DvHPR97] A. Dold, F. W. von Henke, H. Pfeifer, and H. Rueß. Formal verification of transformations for peephole optimization. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 459–472, 1997.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[Gau92] M. C. Gaudel. Structuring and modularizing algebraic specifications: The PLUSS language, evolutions and perspectives. In *STACS'92. Proceedings.* Springer LNCS 577, 1992.

[GDG+96] W. Goerigk, A. Dold, Th. Gaul, G. Goos, A. Heberle, F.W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Rueß, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. Technical Report LiTH-IDA-R-96-12, Linkoping University, 1996.

[GdL94] R. Groenboom and G. R. Renardel de Lavalette. Reasoning about dynamic features in specification languages: A modal view on creation and modification. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages (SOSL'93)*, Workshops in Computing, pages 340–355, London, UK, October 1994. Springer.

[GH93] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[Gol82] R. Goldblatt. *Axiomatising the Logic of Computer Programming.* LNCS 130. Springer, Berlin, 1982.

[Gor88] M. Gordon. HOL: A Proof Generating System for Higher-order Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification and Synthesis.* Kluwer Academic Publishers, 1988.

[GR95] R. Groenboom and G. Renardel de Lavalette. A Formalization of Evolving Algebras. In *Proceedings of Accolade95.* Dutch Research School in Logic, 1995.

[Gra96] P. Graf. *Term Indexing.* Springer LNCS 1053, 1996.

[GRW95] Joshua Guttman, John Ramsdell, and Mitchell Wand. Vlisp: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[GS97] Y. Gurevich and M. Spielmann. Recursive abstract state machines. *Journal of Universal Computer Science (J.UCS)*, 3(4):233 – 246, 1997.

[Gur95] M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[JC94] C. Jorgensen and G. Caspersen. On-board software development approach for oersted micro satellite. In *Proc. of EUROSPACE On-Board Data Management Symposium*, 1994.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 2nd edition, 1990.

[Joy90] Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, editors, *Proceedings of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 177–201, Berlin, July 1990. Springer.

[Knu73] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition edition, 1973.

[McG72] C.L. McGowan. An inductive proof technique for interpreter equivalence. In R. Rustin, editor, *Formal Semantics of Programming Languages*, pages 139 – 147. Englewood Cliffs, 1972.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Moo88] J Moore. PITON: A Verified Assembly Level Language. Technical report 22, Computational Logic Inc., 1988. available at the URL: http://www.cli.com.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer, 1994.

[Pus96] C. Pusch. Verification of compiler correctness for the WAM. In J.Harrison J. von Wright, J.Grundy, editor, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 347–362. Springer, 1996.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, Berlin, 1995.

[Rei98] W. Reif. Beweisbar korrekte software. in german (Folien zur Vorlesung), 1998.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*. Kluwer Academic Publishers, Dordrecht, 1998.

[Rus92] David M. Russinoff. A verified Prolog compiler for the Warren abstract machine. *Journal of Logic Programming*, 13(4):367–412, August 1992.

[SA97] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 3(4):377–413, 1997. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.

[SA98] G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering. Kluwer Academic Publishers, 1998.

[Sch94] P. H. Schmitt. Proving WAM compiler correctness. Interner Bericht 33/94, Universität Karlsruhe, Fakultät für Informatik, 1994.

[Sch95] A. Schönegge. Extending Dynamic Logic for Reasoning about Evolving Algebras. Technical Report 49/95, Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany, 1995.

[Sch99] W. Schulte. High Integrity Compilation and Secure Execution of Java. Habilitation Thesis, University of Ulm, to appear, 1999.

[Spi88] J. M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[Ste85] W. Stephan. A logic for recursive programs. Technical Report 5/85, Universität Karlsruhe, Fakultät für Informatik, 1985.

[TvS82] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson. Using peephole optimization on intermediate code. *ACM Transactions on Programming Languages and Systems*, 4(1):21–36, January 1982.

[Vog97] H. Vogt. Verifikation reaktiver Software-Komponenten. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 1997. (in German).

[Wir90] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, Oxford, 1990.

[You88] W. D. Young. A Verified Code Generator for a Subset of Gypsy. Technical report 33, Computational Logic Inc., 1988. available at the URL: http://www.cli.com.

[ZG97] W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *Journal of Universal Computer Science (J.UCS)*, 3(5):504 – 567, 1997. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.