Von PROLOG zur WAM

Verifikation der Prozedurübersetzung mit KIV

Wolfgang Ahrendt

Diplomarbeit

Dezember 1995

Institut für Logik, Komplexität
und Deduktionssysteme
Universität Karlsruhe

Betreuer:
Prof. Dr. W. Menzel
Dipl.-Inform. Gerhard Schellhorn

Hiermit versichere ich, daß ich diese Diplomarbeit eigenständig verfaßt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Karlsruhe, im Dezember 1995

(Wolfgang Ahrendt)

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Compilerverifikation und PROLOG

Die vollständig formale Verifikation von Compilern ist bisher ein in der Informatik noch kaum bearbeitetes Gebiet. Dies mag daran liegen, daß sich dafür geeignete Formalisierungen der Problemstellung nicht unmittelbar aufdrängen (um das vorsichtig auszudrücken); zumindest nicht, wenn sie eine handhabbare(!) Basis bilden sollen für die Verifikation.

Zunächsteinmal benötigen wir die rigorose Definition einer Semantik sowohl der Quell- als auch der Zielsprache.[1] Beides ist keineswegs eine Selbstverständlichkeit.[2] Erst dann läßt sich ein korrekter Compiler spezifizieren, und zwar in etwa mit der Gleichung:

$$SEM(Db) = {}^{3}SEM'(compile(Db)) \tag{1.1}$$

Dies ist zwar eine Spezifikation, aber damit ist noch wenig gewonnen. Die kurze Gleichung (1.1) formuliert ja einen sehr komplexen Zusammenhang. Die Datenbasis Db hat eine komplexe Datenstruktur und die Beschreibungen der Semantiken SEM und SEM' sind i.a. hochgradig nichttrivial.

Das Kernproblem der Compilerverifikation ist nun, wie im folgenden noch ausführlich dargelegt wird, die Reduktion der Gleichung (1.1) auf vergleichsweise „einfache“ Compiler-Annahmen CA_i, für die gelten muß:

1. Die CA_i sind in ihrer Summe hinreichend für die Korrektheit des Compilers:

$$\bigwedge_i CA_i \Longrightarrow SEM(Db) = SEM'(compile(Db))$$

2. Menge der CA_i ist eine hinreichend handhabbare Spezifikation des Compilers (insbesondere befreit von den Semantik-Formalismen SEM und SEM').

Diese Vorgehensweise ist der Grundansatz des PROLOG-WAM[4]-Projektes von Egon Börger und Dean Rosenzweig, insbesondere der Arbeit: „The WAM - Definition and Compiler Correctness“[BörgerRosenzweig]. Hier werden gerade die angesprochenen Schritte vollzogen:

[1] Von den vielfach benötigten Zwischensprachen ganz zu schweigen.

[2] Dies zeigt auch das Beispiel PROLOG und WAM.

[3] $SEM(Db)$ und $SEM'(compile(Db))$ sind partielle Funktionen. Die Gleichheit der Funktionen ist definiert über die starke Gleichheit $=^s$ auf der Zielsorte. $F = F' :\Leftrightarrow \forall\, x.F(x) =^s F'(x)$. Das bedeutet u.a., daß beide Funktionen für die selben Inputs undefiniert sind, siehe auch completeness in [BörgerRosenzweig].

[4] Warren Abstract Mashine: Sprache zur maschinennahen und ausführungsoptimierten Repräsentation von PROLOG-Programmen, verbunden mit einem (im Original informellen) Ausführungsmodell. [Warren]

1. die Definition einer Semantik für PROLOG,

2. die Definition einer Semantik für den Code der WAM und

3. die Reduktion der Äquivalenz beider auf Compiler-Annahmen.

Hierbei ist allerdings 3. aus guten Gründen methodisch verzahnt mit einer stufenweisen Herleitung von 2..

Dieses Projekt hat seinen Ursprung in Egon Börgers Suche nach einer präzisen Semantik für PROLOG, und zwar eben nicht nur für die Horn-Klauseln, sondern für die vollständige Programmiersprache „.. with all its so called dirty features"[Börger]. Hierzu gehört insbesondere auch das eingebaute Prädikat „!", sprich „Schnitt", dessen Effekt eine lokale Änderung der Baumsuchstrategie ist. Schon deswegen mußte das gesuchte Modell ein ausführungsorientiertes sein.[5]

Erstaunlicherweise scheiterten hier die „gängigen" Konzepte (s. [Börger]), was unter anderem daran liegen mag, daß in PROLOG Sprachelemente der Objekt- und der Metaebene verschränkt sind, nämlich einerseits die logische Charakterisierung eines Suchraums durch Hornklauseln und andererseits die (imperative) Manipulation desselben („Schnitt").

Den Durchbruch brachte das Konzept der dynamischen Algebren („evolving algebras") von Yuri Gurevich. Dort wird der intuitive Begriff des „Berechnungszustandes" identifiziert mit einer (statischen) Algebra, also einer bestimmten Interpretation gegebener Funktionssymbole. Ein Berechnungsablauf entspricht dann einer Folge von Aktionen, die jeweils die Interpretation der Funktionen transformieren.[6]

Aufbauend auf die Grundidee der operationalen Semantik, wonach die Bedeutung einer Programmiersprache definiert wird mittels einer abstrakten Maschine zur Ausführung der Sprachkonstrukte, konnten [BörgerRosenzweig^a] mit dynamischen Algebren einen abstrakten Interpreter notieren, der die Semantik für vollständiges PROLOG definiert.[7]

Jetzt lag es nahe, die WAM, die ja von sich aus eben eine „abstrakte Maschine" ist, auch in Form einer dynamischen Algebra zu notieren; dies nicht als Selbstzweck, sondern als Basis eines Korrektheitsbeweises für die Übersetzung von PROLOG in WAM-Code. Das ist geschehen in [BörgerRosenzweig], welches die Grundlage der vorliegenden Arbeit darstellt.

Dort wird (nach einer Einführung in dynamische Algebren) zunächst nochmals die PROLOG definierende dynamische Algebra vorgestellt, allerdings beschränkt auf ein Kern-PROLOG, nämlich Horn-PROLOG zuzüglich der eingebauten Prädikate „!", „true" und „fail". Mit dem „!" ist das wohl wichtigste nicht-deskriptive Konstrukt in dem Kern-PROLOG enthalten.

Dieses Ausführungsmodell in Form einer dynamischen Algebra wird dann stufenweise modifiziert, und zwar sowohl die abstrakte Maschine als auch der Code, auf dem sie arbeitet. Es ergibt sich eine Folge von dynamischen Algebren, deren erste PROLOG definiert und deren letzte die WAM modelliert. Durch die Trennung in einzelne Zwischenstufen wurden die Konzepte der WAM orthogonalisiert mit dem Effekt, daß man die Korrektheit dieser Konzepte getrennt untersuchen kann im Vergleich zweier „benachbarter" dynamischer Algebren. Diese Schritte zwischen zwei abstrakten Maschinen können von zweierlei Art sein:

[5] Mit den oben erwähnten Semantik-Formalismen sind eben nicht nur denotationale Ansätze gemeint.

[6] Eine Einführung in dynamische Algebren findet sich in [BörgerRosenzweig].

[7] Dies geschah zunächst nicht im Hinblick auf die WAM, sondern mit dem Ziel einer präzisen und zugleich *intuitiven* PROLOG-Definition. Darum arbeitet die abstrakte Maschine auch bewußt nicht optimierend.

1. **Optimierungsschritt**: Ein Übergang zu einer optimierten (bzw. maschinennäheren) Abarbeitung ein und der selben Datenbasis: Hierbei wird die Äquivalenz zweier dynamischer Algebren postuliert unter der Voraussetzung, daß beide auf der gleichen Datenbasis arbeiten.

2. **Compilationsschritt**: Die Einführung bestimmter Bestandteile des WAM-Codes und die Angabe einer abstrakten Maschine, die diese neue „Zwischensprache" ausführt. Hier wird die Äquivalenz zweier dynamischer Algebren postuliert unter der Voraussetzung, daß die (jetzt verschiedenen) Datenbasen in einer bestimmten Beziehung zueinander stehen, ausgedrückt in einer *Compiler-Annahme*.

[BörgerRosenzweig] stellen mit jeder Einführung einer neuen abstrakten Maschine (als dynamische Algebra) den Satz auf, daß diese operational äquivalent ist zu der vorhergehenden Stufe, je nachdem entweder unter der Voraussetzung gleicher Datenbasis oder unter der Voraussetzung einer Compiler-Annahme.

Wenn es gelingt, diese Sätze für sämtliche Stufen als korrekt nachzuweisen, dann ist gezeigt, daß die Summe aller Compiler-Annahmen die Spezifikation einer korrekten Klasse von PROLOG-Compilern darstellt! Diese Beweise zu führen mit Mitteln des taktischen und des automatischen Theorembeweisens ist das Ziel des Karlsruher PROLOG-WAM-Projektes, in dessen Rahmen die vorliegende Arbeit entstanden ist. Aufbauend auf die Vorlage [BörgerRosenzweig], die nach eigenen Angaben „... preformal (rather than antiformal)" ist, mußten die zu zeigenden Behauptungen präzise formalisiert werden. Dazu gehören:

1. die *Konkretisierung der* einzelnen *dynamischen Algebren* in einem Formalismus, der (nach heutigem Stand) einem implementierten und leistungsfähigen Deduktionssystem zugänglich ist,

2. die *Formalisierung der Compiler-Annahmen*, die im Original [BörgerRosenzweig] umgangssprachlich formuliert sind,

3. die *Spezifikation derjenigen Datenstrukturen*, die in den betrachteten abstrakten Maschinen und Compiler-Annahmen benutzt werden, und

4. in Verbindung von 1. und 2. die *Formulierung der* Lemmata, die eben die operationale *Äquivalenz* zweier abstrakter Maschinen unter Voraussetzung einer Compiler-Annahme postulieren, und zwar in einer systemisch verifizierbaren Form.[8]

Diese Schritte werden vollzogen für die ersten 5 Stufen aus [BörgerRosenzweig], als da sind: I) der PROLOG Baum, II) der PROLOG Stack, III) die Wiederverwendung der Coicepoints, IV) die Ermittlung determinierter Prozeduren und V) die Übersetzung der Prozedur-Struktur.[9] Mit *Prozedur* ist hier wie im Titel die *Gesamtheit der Klauseln für ein gegebenes Prädikat* gemeint. Somit ist die Basis geschaffen für die Verifikation der Äquivalenzen I $\leftrightarrow$ II, II $\leftrightarrow$ III, III $\leftrightarrow$ IV und IV $\leftrightarrow$ V.

In der Stufe V wird ein neuer Zwischencode eingeführt und gleichzeitig eine optimierende Anordnung dieses Codes zugelassen. Diese zwei Aspekte mußten für die Verifikation getrennt werden, so daß eine Zwischenebene (V') eingezogen wurde. Diese besteht aus der gleichen

[8] All diese Punkte werden im folgenden noch ausführlich vertieft und zwar 1. in Kapitel 3.3, 2. in Kapitel 3.2, 3. in Kapitel 3.1 und 4. in Kapitel 3.3.

[9] Diese Stufen entsprechen genau den Abschnitten 1.1. bis 2.2. aus [BörgerRosenzweig].

abstrakten Maschine wie V; lediglich die Compiler-Annahme wird (zunächst) dahingehend vereinfacht, daß nur eine 1:1 Übersetzung von PROLOG in den Zwischencode zugelassen wird.

Dies alles wird unten noch ausführlich erläutert. An dieser Stelle soll nur der Rahmen abgesteckt werden, den die vorliegende Arbeit ausfüllt: Denn genau dieser Übersetzungsschritt, von PROLOG (und seiner Abarbeitung mittels Ermittlung determinierter Prozeduren IV) hin zur nichtoptimierenden Repräsentation im Zwischencode (V'), wurde verifiziert! Hier liegt der Schwerpunkt der Arbeit.

Zu der vorweggenommenen Zusammenstellung der Ergebnisse gehört aber noch das vielleicht wichtigste: Es ist mehr geschehen, als nur die semi-formale mathematische Analyse aus [BörgerRosenzweig] mit großem Aufwand sozusagen mundgerecht der maschinellen Überprüfung zugänglich zu machen. Verifikation hat von jeher nicht nur die Aufgabe, eine schon vermutete Korrektheit zu bestätigen, sondern, was viel wichtiger ist, Fehler zu entdecken.

Gerade dies ist auch hier geschehen. So wurde erst mitten im Verifikationsprozeß deutlich, daß die in [BörgerRosenzweig] beschriebene WAM mindestens zwei Fehler aufweist. Die Konsequenz des schwerwiegenderen ist: Durch einen versehentlich eingebauten Indeterminismus kann die Anfrage ?- p an das Programm:

```
p :- fail.
p.
```

zu einer Endlosschleife führen, obwohl sie natürlich gelingen müßte. Dieser Fehler ist zwar mit minimalem Aufwand zu beheben (durch Verstärkung einer Regeleinstiegsbedingung), aber seine Konsequenz ist fatal, und vor allem ist er für den menschlichen Leser auch bei intensivem Studium der Regeln kaum zu entdecken! Die Intention der Regelanwendungen ist so überdeutlich, daß der formale Indeterminismus beim Lesen durch Befolgung der verstandenen Intention klar aufgehoben wird.[10]

Zum zweiten ist recht früh deutlich geworden, daß die Compiler-Annahmen, auch wenn man sie so, wie sie gemeint sind, exakt formalisiert, unvollständig sind. Wir werden unten Beispiele von Compilaten angeben, die die Compiler-Annahme 1 aus [BörgerRosenzweig] erfüllen, aber sicher keine korrekte Übersetzung des jeweiligen PROLOG-Programms darstellen. Diese Ungenauigkeit war zwar viel leichter zu entdecken als die oben beschriebene, aber dafür war umgekehrt die Behebung mindestens nicht trivial. Darum ist auch die hier verwendete (und als hinreichend verifizierte) Hilfsfunktion „chain“ komplizierter als in [BörgerRosenzweig]. Auch dieser Punkt ist sehr wichtig, weil ein gegenüber unvollständigen Compiler-Annahmen als korrekt verifizierter Compiler eben noch nicht korrekt ist gegenüber PROLOG und der WAM.

Es bleibt noch zu sagen, daß das hier Geleistete aus der Sicht des Karlsruher PROLOG-WAM-Projektes erst ein Anfang ist. Ziel ist die Verifikation sämtlicher Stufen aus [BörgerRosenzweig].

[10] Details hierzu siehe Abschn.4.4.3

Kapitel 2

Grundlagen

Der Schwerpunkt dieser Arbeit liegt wie gesagt zum einen in der Formalisierung der ersten fünf Stufen aus [BörgerRosenzweig], d.h. der Spezifikation der Datenstrukturen und der Umsetzung sowohl der dynamischen Algebren als auch der Compiler-Annahmen in einer geeigneten Logik. Zum anderen liegt er in der wechselseitigen Verifikation der Stufen IV und V' (nach Fehlerverbesserung).

Eine plausible Darstellung beider Bereiche ist nicht möglich, wenn nicht vorher wenigstens eine intuitive Vorstellung vermittelt wird von:

1. den betrachteten abstrakten Maschinen zur Interpretation unübersetzten bzw. übersetzten PROLOGs,

2. dem Formalismus, in dem diese Maschinen im Original notiert sind, nämlich dynamische Algebren, und

3. den Konzepten, die uns zur Umsetzung der Vorlage im KIV-System dienten, nämlich strukturierter algebraischer Spezifikation in Logik erster Stufe und Dynamischer Logik zur Simulation von Sätzen über dynamischen Algebren.

Hierüber soll dieses Kapitel in kürzestmöglicher Weise Auskunft geben, ohne auch nur den geringsten Anspruch auf Vollständigkeit oder Exaktheit zu erheben. Zweck ist einzig und allein die Erleichterung des Zugangs zu den folgenden Kapiteln. Die beste und auch nicht ersetzbare Einführung in obige Punkte 1. und 2. ist [BörgerRosenzweig] selbst, und zwar für unsere Zwecke bis einschließlich Abschnitt 2. Darum empfehlen wir die folgenden Ausführungen auch nicht alternativ zu [BörgerRosenzweig], sondern umgekehrt zur leichteren Adaptierung dieser Arbeit an die Vorlage.

Wir werden die Einführung der abstrakten Maschinen und der dynamischen Algebren verbinden, indem das eine als ein Beispiel für das andere dient.

2.1 PROLOG-Definition mit dynamischen Algebren

Zunächst wollen wir die Grundideen derjenigen abstrakten Maschine von [BörgerRosenzweig] erläutern, die PROLOG-Programme interpretiert und damit die Semantik dieser Sprache definiert.

Die wichtigsten Datenstrukturen zur Repräsentation des Zustands einer PROLOG-Berechnung sind die Sequenz der noch zu erfüllenden Ziele und die aktuelle Substitution. Dieser Zustand wird modifiziert durch

1. die Unifikation des ersten Zieles in der Sequenz mit dem Kopf einer Klausel,
2. die Ersetzung des ersten Zieles durch den Rumpf der Klausel,
3. die Anwendung der Substitution auf die so erhaltene Zielsequenz und
4. die Verkettung der bisherigen Substitution mit der unifizierenden.

Weil beim Scheitern von Teilzielen alternative Klauseln ausprobiert werden müssen, ist Backtracking nötig. Das bedeutet, daß der PROLOG-Interpreter über die zurückliegenden Berechnungszustände und alternative Klauselauswahlen Buch führen muß. Diese „strukturierte Historie“ wird durch eine Baum dargestellt, dessen Knoten gerade die beschriebenen Berechnungszustände repräsentieren. Ein ausgezeichneter Knoten, *currnode* genannt, trägt den aktuellen Zustand. Die Verzeigerung des Baumes geht in der Hauptsache von unten nach oben (weil sie für das *Back*tracking geschaffen wurde), und zwar durch die „Vater“-Information eines jeden Knotens. Von oben nach unten verzeigert ist die Information, die zu jedem Knoten angibt, welche noch nicht betretenen Kandidaten *(cands)* noch als Nachfolgezustände in Frage kämen, sollte der Knoten wieder zum *currnode* werden.

Die alternativen Knoten (Geschwisterknoten) unterscheiden sich u.a. dadurch, daß sie auf unterschiedliche Zeilen *cll* (clauseline) des PROLOG-Programms verweisen. Die Liste dieser alternativen Programmzeilen stellt eine PROLOG-Prozedur dar. Wir nehmen an, daß die Funktion *procdef* zu gegebenem Ziel (Term) gerade diese Liste zurückgibt.

Das mögliche Auftreten eines „!“ (Schnitt) macht noch eine Erweiterung der beschriebenen Zustandsrepräsentation nötig. Die Bearbeitung des „!“ als linkestem Teilziel der Sequenz verursacht einzig und allein, daß die im Baum dargestellte Backtrackinginformation modifiziert wird. Ein eventuell später notwendiges Backtracking aus *diesem* Zustand heraus soll dann nicht mehr zum (alten) Vaterzustand führen. Aber wohin sonst?
Die Antwort lautet: Zum *Vater* desjenigen Zustandes, dessen linkestes Sequenz-Ziel durch den Klauselrumpf, der den „!“ enthält, ersetzt wurde. Nur so werden auch die alternativen Klauseln abge*schnitt*en.

Um diese kompliziert anmutende Information uniform darzustellen, wird der Zielsequenz mit jedem Klauselrumpf auch ein Verweis auf den Vaterzustand der alten Zielsequenz mitgegeben. Diesen Verweis nennen wir *cutpt* (cutpoint). Beim Auftreten eines „!“ als linkestem Ziel der Sequenz muß lediglich noch der Vater-„pfeil“ auf den *cutpt* „umgebogen“ werden.

Die Zielsequenz kann jetzt angesehen werden als Liste, innerhalb derer Teillisten (Klauselrümpfe) mit cutpoints ausgezeichnet sind. Das ergibt als Datenstruktur eine Liste von Paaren, die jeweils aus einer Zielliste und einem Knoten bestehen. Das Ganze wird bezeichnet mit *decglseq* (decorated goal sequence). Wir gehen jetzt in Anlehnung an [BörgerRosenzweig] dazu über, mit *goal* (Ziel) nicht mehr einzelne PROLOG-Terme aus der *decglseq* zu bezeichnen, sondern einzelne, mit einem gemeinsamen *cutpt* ausgezeichnete Termlisten, insbesondere das erste Element des ersten Paares aus *decglseq*.

Die *decglseq* hat alo folgende Gestalt[1]:

[1] in dieser Darstellung entnommen aus [Schmitt]

$$decglseq = \langle\, \langle\, \underbrace{\langle\, \overbrace{g_{1,1}}^{act}, g_{1,2}, \ldots, g_{1,k_1} \,\rangle}_{goal}, \overbrace{n_1}^{cutpt} \,\rangle, \ldots, \langle\, \langle\, g_{q,1}, \ldots, g_{q,k_q} \,\rangle, n_q \,\rangle \,\rangle$$

$$cont = \langle\, \langle\, \quad \langle\, g_{1,2}, \ldots, g_{1,k_1} \,\rangle, n_1 \,\rangle, \ldots, \langle\, \langle\, g_{q,1}, \ldots, g_{q,k_q} \,\rangle, n_q \,\rangle \,\rangle$$

Der erste Term des *goal* heißt *act*. Dieser Term bestimmt zunächst den weiteren Fortgang einer Berechnung. Wenn er durch einen neuen Rumpf (mit entsprechendem *cutpt*) ersetzt wird, setzt sich die folgende *decglseq* zusammen aus dem neuen Paar und dem oben dargestellten *cont* (continuation).

Die Selektoren für den Kopf bzw. den Rumpf einer Klausel heißen *hd* (head) und *bdy* (body). *subres* bezeichnet die Funktion, die eine Substitution auf alle Terme einer *decglseq* anwendet.

Wir müssen nun noch ein eher technisches Detail erwähnen. Die Variablen eines PROLOG-Programms sind ja implizit allquantifiziert, und zwar klauselloka l. Das bedeutet, daß vor einem PROLOG-Resolutionsschritt die Variablen der gewählten Klausel evtl. so umbenannt werden müssen, daß die Klausel variablendisjunkt zur aktuellen *decglseq* sind. Dieses Problem ist bei [BörgerRosenzweig] in uniformer Weise gelöst durch einen Variablenindex *vi*, der bei jeder Bildung einer neuen *decglseq* hochgezählt wird und mit Hilfe dessen alle Variablen einer ausgewählten Klausel vor der Verwendung umbenannt werden (mit *rename*).

Nach diesen Vorbemerkungen wollen wir das Experiment wagen, den Leser mit der PROLOG-definierenden dynamischen Algebra aus [BörgerRosenzweig] zu konfrontieren und erst hinterher einige Worte über dynamische Algebren als solche verlieren. Wir wollen damit auch demonstrieren, welch erstaunliche Einfachheit in diesem mächtigen Konzept steckt.

Es sei noch darauf hingewiesen, daß direkt aufeinander folgende Zuweisungen („:=“) parallel zu verstehen sind und nicht sequentiell. Die folgende dynamische Algebra entspricht genau der ersten aus [BörgerRosenzweig], sie ist aber in dieser kompakten Schreibweise entnommen aus [Schmitt].

Abkürzungen

father ≡ father(currnode)
cands ≡ cands(currnode)
sub ≡ sub(currnode)
decglseq ≡ decglseq(currnode)
goal ≡ fst(fst(decglseq))
cutpt ≡ snd(fst(decglseq))
act ≡ fst(goal)
cont ≡ [<rest(goal),cutpt> |rest(decglseq)]

final-success-rule

IF stop = 0 & decglseq(currnode) = []
THEN stop := 1

goal-success-rule

IF stop = 0 & goal = []
THEN decglseq := rest(decglseq)

call-rule

IF stop = 0 & is_user_defined(act) & mode = Call
THEN LET n = length(procdef(act,db))
EXTEND NODE by $temp_1$,. . .,$temp_n$
WITH
father($temp_i$) := currnode
cll($temp_i$) := proj(procdef(act,db),i)
cands := [$temp_1$,. . .,$temp_n$]
ENDEXTEND
mode := Select

select-rule

IF stop = 0 & is_user_defined(act) & mode = Select
THEN IF cands = []
THEN backtrack
ELSE LET clause = rename(clause(cll(fst(cands))),vi)
LET unify = unify(act,hd(clause))
IF unify = nil
THEN cands := rest(cands)
ELSE currnode := fst(cands)
decglseq(fst(cands)) :=
subres([<bdy(clause), father> |cont], unify)
sub(fst(cands)) := sub ∘ unify
cands := rest(cands)
mode := Call
vi := vi + 1

mit

backtrack ≡ IF father = root
THEN stop := -1
ELSE currnode := father
mode := Select

true-rule

IF stop = 0 & act = true
THEN decglseq := cont

fail-rule

IF stop = 0 & act = fail
THEN backtrack

cut-rule

IF stop = 0 & act = !
THEN father := cutpt
decglseq := cont

initial

root	n_0
currnode	n_1
father(n_1)	n_0
decglseq(n_1)	[<query,n_0 >]
sub(n_1)	[]
vi	0
stop	0
mode	Call
db	program

Auch wenn es sicherlich nicht leicht ist, sich ein vollständiges Bild von dem Zusammenwirken der verschiedenen Regeln zu machen, so ist doch ein Verständnis für die lokale Bedeutung der einzelnen Aktionen kaum schwierig, wenn man sich auf den Standpunkt stellt, man habe es mit Code über abstrakten Daten zu tun.

Wir wollen aber kurz festhalten, was die Charakteristik und (informelle) Bedeutung dieser Notation ausmacht.

Es handelt sich um ein regelbasiertes, also im Prinzip indeterministisches System. Jede Regel besitzt eine Einstiegsbedingung (nach dem ersten IF), die für jeden „Zustand“ entscheidet, ob diese Regel aufgerufen wird oder nicht. Wenn ja, werden, eventuell abhängig von weiteren Bedingungen, die aufgeführten Updates parallel ausgeführt. Der entscheidende Unterschied zu *Zuweisungen* in imperativen Programmiersprachen ist nun, daß die Updates nicht die Belegung von Variablen modifizieren, sondern die Interpretation von Funktionen! Eine einzelne Interpretation bezeichnet man auch als statische Algebra.

Wir betrachten also keine Konstanten und keine Variablen (auch nicht, wenn das den Anschein hat), sondern ausschließlich Funktionen. Diese befinden sich in einer Art Zustand, der aktuellen Interpretation. Diese wird durch ein Update (oder mehrere parallele Updates) transformiert, und zwar in der folgenden Weise:
In $f(t_1, \ldots, t_n) := t_0$ werden die Terme $t_0, t_1, \ldots, t_n$ in der aktuellen (alten) Interpretation I ausgewertet, sagen wir zu $d_0, d_1, \ldots, d_n$. In der neuen Interpretation I' gilt: $I'(f)(d_1, \ldots, d_n) = d_0$. An jeder anderen „Stelle“ ist $I'(f) = I(f)$. Natürlich bleibt die Interpretation der anderen Funktion konstant.[2]

Es gibt auch nullstellige Funktionen, in unserem Beispiel *currnode*. Diese kann man sich auch als Variablen oder Konstanten im herkömmlichen Sinne vorstellen, je nachdem, ob sie in irgendeiner Regel einem Update unterliegen oder nicht. (Vorsicht: z.B. *father* ist keine nullstellige Funktion, sondern eine Abkürzung für *father(currnode)*.)

Von den Universen der Interpretation haben wir hier gar nicht gesprochen. Das holen wir bei der Umsetzung in strukturierte algebraische Spezifikationen nach. Eines aber ist wichtig: [BörgerRosenzweig] verwenden, eingebettet in das Konzept der dynamischen Algebren, dynamische Universen. In unserem Beispiel ist das einzige dynamische Universum das der Baumknoten: NODE. Die Vorstellung (bezogen auf das Beispiel) ist die: Immer, wenn die abstrakte Maschine den Baum erweitern will, erweitert sie das Universum der Knoten um *neue* Elemente, und zwar mittels des EXTEND-Konstruktes.[3] Je nach Mächtigkeit der Prozedurdefinition werden neue Knoten in der benötigten Anzahl erzeugt. Dann werden im WITH-Teil

[2] Formaler siehe [BörgerRosenzweig].
[3] Erläuterung siehe [BörgerRosenzweig].

die Funktionen *cll* und *father* an der Stelle dieser Knoten modifiziert, und zwar so, daß jeder auf eine andere Zeile der Prozedurdefinition zeigt und alle den gleichen Vater *currnode* besitzen. Umgekehrt wird die *cands*-Funktion an der Stelle *currnode* gerade auf die Liste der neuen Knoten gesetzt.

Die Bezeichner $temp_1, \ldots, temp_n$ der neuen Elemente sind nur zwischen EXTEND und ENDEXTEND gültig. Danach kann man (zunächst) nur über *cands(currnode)* auf die neuen Knoten zugreifen.

An dieser Stelle wird auch deutlich, warum die einzelnen statischen Algebren partiell sind. Nach Ausführung der Call-Regel sind auf den neuen Elementen zwar die Funktionen *father* und *cll* definiert, aber z.B. noch nicht die Funktion *decglseq*. Auch die oben definierte initiale Algebra, die den Zustand vor der ersten Regelanwendung beschreibt, ist partiell. (So ist z.B. *cands(currnode)* noch nicht definiert).

Die initiale Algebra verarbeitet die Eingaben an die abstrakte Maschine, nämlich *program* und die Anfrage *query*. Als finale Algebra ist diejenige anzusehen, in der keine Regel mehr anwendbar ist. Dies ist insbesondere der Fall für $stop = \pm 1$. Im Falle von $stop = -1$ sehen wir das „-1“ (später *failure* genannt) als Ausgabe an, im Falle von $stop = 1$ den aktuellen Wert von *s(currnode)* (die Antwortsubstitution).[4]

Mit diesen Hinweisen sollte es möglich sein, sich durch das Lesen der Regeln ein Bild von der Funktionsweise der abstrakten Maschine zu machen. Wichtig ist vor allem die Selbststeuerung durch die Modi *Call* und *Select*. Eine ausführliche Einführung sprengt an dieser Stelle den Rahmen, siehe aber [BörgerRosenzweig].

Hier wollen wir uns so rasch wie möglich den Stufen nähern, die im Rahmen dieser Arbeit verifiziert wurden. Auf dem Weg dahin beschreiben wir nur informell die sukzessiven Veränderungen, denen die oben beschriebene dynamische Algebra in [BörgerRosenzweig] unterworfen ist. Erst bei der nächsten verifizierten Stufe geben wir wieder die vollständige dynamische Algebra an, in der gleichzeitig die bis dahin vollzogenen Veränderungen zusammengefaßt sind.

2.2 STACK-Verarbeitung von PROLOG

In [BörgerRosenzweig] schließen sich nun eine ganze Reihe von dynamischen Algebren der ersten an. Bei jedem Schritt wird die abstrakte Maschine weiter abgewandelt und evtl. auch die Programmdarstellung, auf der sie arbeitet. Dabei wird die Funktionsweise derjenigen der WAM immer ähnlicher, da in jedem Schritt eine neue Komponente der WAM oder des WAM-Codes eingeführt wird.

Der erste Schritt (beschrieben im Abschnitt „From Tree To Stack“ [BörgerRosenzweig]) besteht aus dem Übergang von der vorgestellten Baum-orientierten Abarbeitung zu einer Stack-orientierten. Der aktuelle Berechnungszustand wird nun nicht mehr durch einen ausgezeichneten Knoten repräsentiert, sondern durch globale Register (in dynamischen Algebren in Form von nullstelligen Funktionen). Die aktuelle Substitution steht jetzt in *subreg*, die aktuelle Zielsequenz in *decglseqreg* und die aktuelle Programmzeile in *cllreg*.[5] Wie bisher wird die Historie bisheriger Berechnungszustände in verzweigten Knoten repräsentiert, die aber nicht mehr als NODES, sondern als STATES bezeichnet werden. Gleichermaßen kosmetisch ist die

[4] Genauere Betrachtungen über die von einer dynamischen Algebra definierten Funktion unter besonderer Berücksichtigung der Eingabekodierung und der Ausgabedekodierung finden sich in [BeckertHähnle].

[5] Wir weichen hier minimal ab von der Namensgebung in [BörgerRosenzweig]. Dort wird ein Polymorphismus verwendet, den wir ohnehin noch auflösen müßten (s.u.).

Umbenennung der *father*-Funktion in *b* (wie back) und des obersten Knotens *root* in *bottom*. „*b*“ ist schon ein WAM-Bezeichner.

Der durch Backtracking nächsterreichbare Zustand (vormals *father(currnode)*) wird durch das globale Register *breg* repräsentiert (ist Wert der nullstelligen Funktion *breg*).

Ein entscheidender Unterschied ist nun, daß auf dem Knoten keine *cands*-Funktion definiert ist und entsprechend auch die Knoten der *cands*-Listen nicht angelegt sind. Statt dessen wird ein neuer Knoten erst dann angelegt, wenn er benötigt wird. Dies ist der Fall, wenn alle durch die left-to-right-Auswahlregel bevorzugten Alternativen gescheitert sind. Weil die Erzeugung neuer Knoten jetzt nicht mehr in Gruppen und zentral stattfindet, sondern einzeln und dezentral, steht das EXTEND-Konstrukt nicht mehr in der Call-Regel, sondern in der Select-Regel. Statt der Vorausschau auf alle Alternativen müssen die Zustandsknoten nur noch den Wiederaufsetzpunkt im PROLOG-Programm repräsentieren, und zwar in der *cll*-Funktion.[6]

Es wird deutlich, daß es sich bei den Zustandsknoten um Chopicepoints im Sinne der dynamischen Programmierung handelt. Sie „speichern“ einen Berechnungszustand (mit *decglseq* und *s*), einen Wiederaufsetzpunkt (mit *cll*) und ihren jeweiligen Vorgängern (mit *b*).

In Abb. 2.1 sind in vereinfachter Darstellung zwei sich entsprechende Momentaufnahmen der beiden abstrakten Maschinen dargestellt. Die dünnen Kreise stellen alte, endgültig aufgegebene Knoten dar. Wenn man sich diese wegdenkt, dann ist der Rest (bei der zweiten Stufe) in der Tat einem Stack ähnlicher als einem Baum.[7] Das Backtracking entspricht einer top/pop-Aktion auf dem Stack: Die Funktionswerte $b(breg)$, $sub(breg)$, $cll(breg)$ und $decglseq(breg)$ werden in die entsprechenden Register geladen. Insbesondere $breg := b(breg)$ „entfernt“ das oberste Kellerelement.

Das genaue Regelsystem für diese neue Stufe ist in [BörgerRosenzweig] beschrieben, bzw. wiederum in kompakterer Form in [Schmitt]. Wir selbst werden die Regeln erst dann wiedergeben, wenn noch zwei weitere Veränderungen eingearbeitet sind.

Die neue dynamische Algebra arbeitet mit der Voraussetzung einer etwas anderen Darstellung des PROLOG-Programms. Bisher war die Struktur völlig offen gelassen worden. Es wurde lediglich angenommen, daß die Funktion *procdef* eine Liste aller in Frage kommenden Klauselzeilen liefert. Jetzt aber nehmen wir explizit an, daß alle Klauseln mit gleichem Prädikatensymbol im Kopf direkt hinterienander stehen. Diese Prozedurdefinitionen seien dann durch eine Zeile mit Eintrag *nil* getrennt. *procdef* liefert jetzt nur noch die erste Zeile einer Prozedurdefinition. Beim Abstieg wird (zunächst) die darauffolgende Zeile als Wiederaufsetzpunkt gespeichert, d.h. sie wird Wert von $cll(breg)$.

Bei erneutem Abstieg vom gleichen Berechnungszustand aus wird ein neuer Choicepoint mit entsprechend inkrementiertem *cll* angelegt. Insgesamt also wird die Programmzeile innerhalb der Prozedurdefinition hochgezählt, bis schließlich *cllreg* den Eintrag *nil* hat und statt eines nächsten Versuches Backtracking ausgelöst wird.

Formell erhält man jetzt die gesamte Prozedurdefinition zu einem PROLOG-Term G mittels *clls(procdef(G,Db))*, wobei die Hilfsfunktionen *clls* definiert ist durch:

$$
\begin{aligned}
clls(Ptr) = \ & \text{IF } clause(Ptr) = nil \\
& \text{THEN } [] \\
& \text{ELSE } [Ptr \mid clls(Ptr+)]
\end{aligned}
$$

[6] *cll* wird also zu anderen Zwecken verwendet als in der vorhergehenden Stufe.

[7] Tatsächlich sind alte Knoten in den statischen Algebren noch vorhanden. Ihre, wenn gleich nutzlose, Existenz verursacht erhebliche Probleme für den Äquivalenzbeweis, siehe [SchellhornAhrendt].

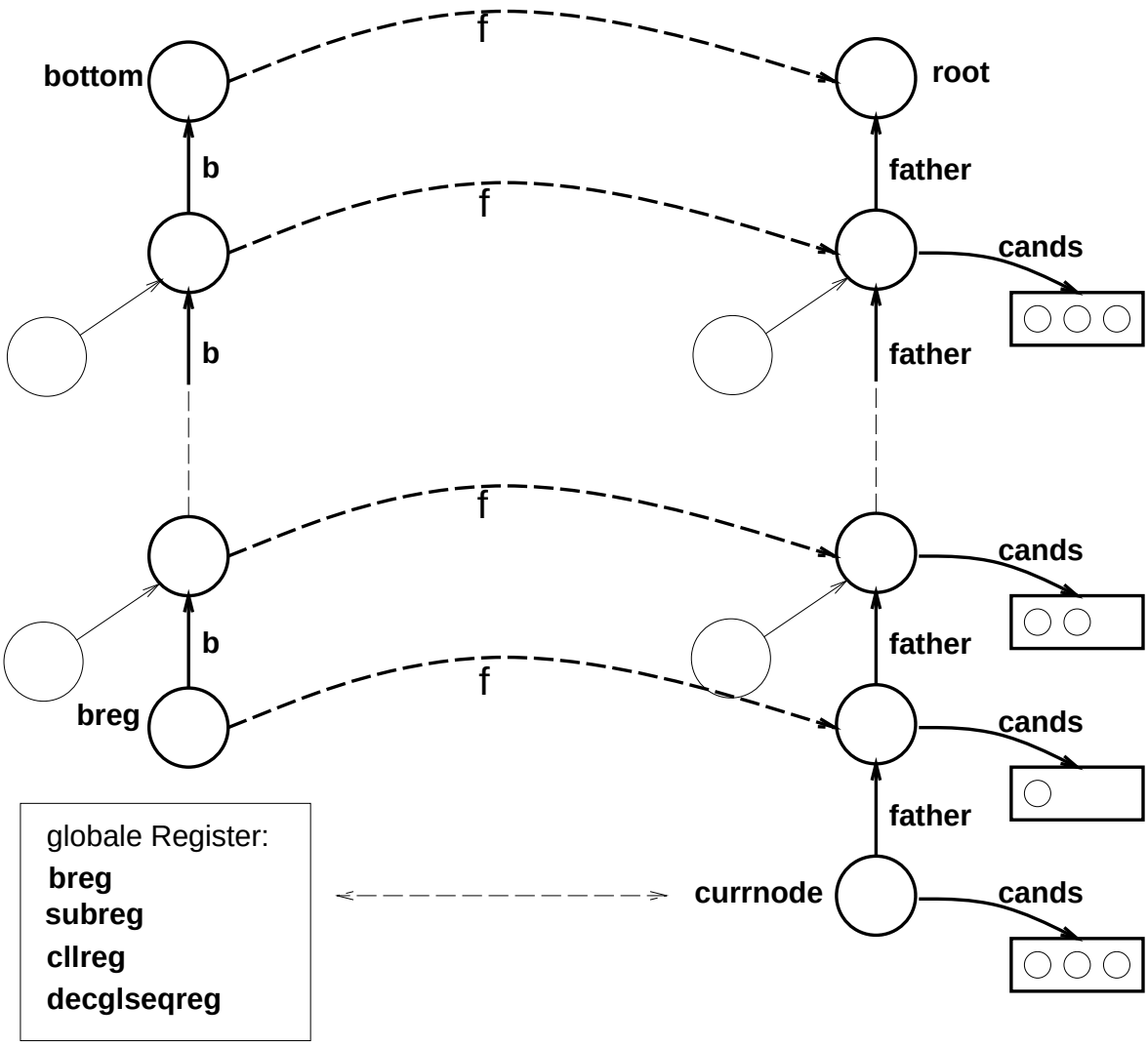


Abbildung 2.1: Momentaufnahme

[BörgerRosenzweig] verstehen die neuen Annahmen an die Programmstruktur als Einschränkung der betrachteten Modelle. Hiervon sind wir abgewichen, nach Absprache mit - und und mit Zustimmung von - Egon Börger. Es ist nicht notwendig, für die Zwecke der zweiten Stufe rückwirkend die Modelle der *Quell*sprache einzuschränken, noch dazu in einer Weise, die sich in keinem PROLOG-Dialekt wiederfindet (man beachte die *nil*-Klausel). Statt dessen formulieren wir eine zusätzliche Compiler-Annahme, die besagt, daß die beiden Prozedurdefinitionen (in neuer bzw. alter Programmdarstellung) auf die gleichen Klauseln deuten, auch wenn sie aus unterschiedlichen Zeilenlisten bestehen. Eine exakte Formalisierung dieser Aussage wird noch nachgetragen.

An dieser Stelle sollte erwähnt werden, daß die Untersuchung der Äquivalenz der beiden ersten Stufen den Ursprung des Karlsruher PROLOG-WAM-Projektes bildet. Der Grundstein wurde gelegt von P.H. Schmitt in [Schmitt]. Dort wurden die beiden dynamischen Algebren analysiert und ein Papierbeweis zur Äquivalenz beider skizziert. Insbesondere wird die Idee der induktiven Definition einer Beweisbildung F zwischen den statischen Algebren beider

Stufen, die bei [BörgerRosenzweig] nur angedeutet ist, konsequent zu Ende gedacht. Eine solche Funktion bildet das Herzstück aller vollzogenen oder noch zu vollziehenden Beweise für die Äquivalenz benachbarter Stufen. Aufbauend auf [Schmitt] wurden dann die beiden ersten Stufen formal mit dem KIV-System verifiziert, und zwar mit einer Methodik, die in dieser Arbeit noch ausführlich erläutert wird. Allerdings mußte die induktive Definition aus [Schmitt] noch erheblich erweitert werden.

Die Funktion F wurde schon in Abb. 2.1 angedeutet. Ihre Definition ist aber sehr viel komplizierter, als die Grafik das vermuten läßt. Die angesprochene Verifikation mit dem KIV-System ist dokumentiert in [SchellhornAhrendt].

2.3 Optimierungen der Stack-Maschine

2.3.1 Wiederverwendung der Choicepoints

Wenn von ein und demselben Berechnungszustand (bezogen auf *subreg* und *decglseqreg*) mehrmals ein Abstieg mit jeweils unterschiedlichen Klauseln versucht wird, dann legt die Select-Regel immer neue Choicepoints an, die sich aber von ihren Geschwisterknoten nur in dem Wiederaufsetzpunkt *cll* unterscheiden. Da aber das Anlegen von Choicepoints eine teure Operation ist, setzt hier die erste Optimierung an. In der beschriebenen Situation können die Choicepoints wiederverwendet werden. Lediglich die Funktion *cll* muß auf ihnen aktualisiert werden mittels $cll(breg) := cll(breg)+$.

Da die abstrakte Maschine jetzt unterscheiden muß zwischen dem ersten Abstieg (1. Kellern eines neuen Choicepoints und 2. Neuberechnung der Register) und den folgenden Abstiegen (3. Wiederverwendung eines Choicepoints und 2. (wie oben) Neuberechnung der Register), benötigen wir jetzt statt des Modus Select drei verschiedene Modi, nämlich 1. Try, 2. Enter, und 3. Retry. Mit Call sind das jetzt vier Modi, die das Zusammenwirken der Regeln steuern.

Für ausführliche Erklärungen und die vollständige dynamische Algebra verweisen wir wieder auf [BörgerRosenzweig] (Abschnitt „Reusing Choicepoints"). Allerdings geben wir schon im nächsten Abschnitt wieder ein vollständiges Regelsystem an, das von dem hier besprochenen (notationell) nur geringfügig abweicht.

Hier sei nur noch erwähnt, daß es jetzt ein zusätzliches Register (eine zusätzliche nullstellige Funktion) *ctreg* gibt. Dieses ist ein weiterer Vorgriff auf die WAM und erfüllt den Zweck, denjenigen Knoten, der als Cutpoint („*cutpt*") in die neu zu bildende *decglseq* eingetragen wird, zwischenzuspeichern. Sein Wert wäre zwar in den von uns betrachteten Stufen jederzeit rekonstruierbar mittels *b(breg)*. In späteren Stufen wird *ctreg* aber unabdingbar.

2.3.2 Ermittlung determinierter Prozeduren

Unter determinierten Prozeduren versteht man solche, die nur aus einer Klauselzeile bestehen, direkt gefolgt von *nil*. Eine weitere Maßnahme, um Choicepoints zu sparen, ist der völlige Verzicht auf solche, die nicht (mehr) gebraucht werden. Dies ist für diejenigen der Fall, bei denen die *cll*-Funktion den Wert einer *nil*-Zeile hat. Überflüssige Choicepoints sollen jetzt gar nicht erst angelegt und überflüssig gewordene sofort entfernt werden. Dies ist durch zusätzliche Bedingungen innerhalb der Try- bzw. der Retry-Regel gewährleistet, und zwar durch $clause(cllreg+) \neq nil$ bzw. $clause(cll(breg)+) \neq nil$.

Da die jetzt erreichte Stufe weiterhin Gegenstand intensiver Betrachtungen sein wird, geben wir hierfür die vollständige dynamische Algebra an.

Abkürzungen

goal ≡ fst(fst(decglseqreg))
cutpt ≡ snd(fst(decglseqreg))
act ≡ fst(goal)
cont ≡ [<rest(goal),cutpt> |rest(decglseqreg)]

final-success-rule

IF stop = 0 & decglseqreg = []
THEN stop := 1

goal-success-rule

IF stop = 0 & goal = []
THEN decglseqreg := rest(decglseqreg)

call-rule

IF stop = 0 & is_user_defined(act) & mode = Call
THEN IF clause(procdef(act,db)) = nil
THEN backtrack
ELSE cllreg := procdef(act,db)
mode := Try
ctreg := breg

try-rule

IF stop = 0 & mode = Try
THEN mode := Enter
IF clause(cllreg+) ≠ nil
THEN EXTEND STATE BY temp
WITH
breg := temp
b(temp) := breg
decglseq(temp) := decglseqreg
sub(temp) := subreg
cll(temp) := cllreg+
ENDEXTEND

enter-rule

IF stop = 0 & mode = Enter
THEN LET clause = rename(clause(cllreg),vi)
LET unify = unify(act, hd(clause))
IF unify = nil
THEN backtrack
ELSE decglseqreg := subres([<bdy(clause),ctreg> |cont],unify)
subreg := subreg ∘ unify
vi := vi + 1
mode := Call

mit

backtrack ≡ IF breg = bottom
THEN stop := -1
ELSE mode := Retry

retry-rule
IF stop = 0 & mode = Retry
THEN decglseqreg := decglseq(breg)
subreg := sub(breg)
cllreg := cll(breg)
cll(breg) := cll(breg)+
ctreg := b(breg)
mode := Enter
IF clause(cll(breg)+) = nil
THEN breg := b(breg)

true-rule
IF stop = 0 & act = true
THEN decglseqreg := cont

fail-rule
IF stop = 0 & act = fail
THEN backtrack

cut-rule
IF stop = 0 & act = !
THEN father := cutpt
decglseqreg := cont

initial

bottom	n_0
breg	n_0
decglseqreg	[<query,n_0 >]
subreg	[]
vi	0
stop	0
mode	Call
db	program

2.4 Übersetzung der Prozedur-Struktur

Der hier zu besprechende Übergang zu einer neuen Stufe (beschrieben in [BörgerRosenzweig], Abschnitt „Compilation of Predicate Structure“) ist von ganz anderer Art als die vorangegangenen. Bisher stand bei dem Übergang von einem Regelsystem zum nächsten immer die veränderte dynamische Verhaltensweise der abstrakten Maschine im Vordergrund. Dabei wurde die Datenbasis entweder beibehalten oder, im ersten Schritt, höchstens „technisch“ manipuliert.

Nun aber geht es um eine echte Übersetzung des PROLOG-Programms in einen neuen, ersten Zwischencode. Dieser besteht äußerlich sowohl aus WAM-Instruktionen als auch aus unübersetzten Bestandteilen von PROLOG. In den folgenden, hier nicht thematisierten Stufen aus [BörgerRosenzweig] werden nach und nach alle Elemente der PROLOG-Syntax übersetzt, bis der vollständige Code der WAM erreicht ist. Im Laufe dieser Entwicklung werden immer wieder zwei Grundlinien deutlich:

1. Die Verlagerung der Kontrolle über den dynamischen Ablauf weg von der abstrakten Maschine hin zum Code selbst. Dies ermöglicht schließlich eine maschinennahe Interpretation des WAM-Codes.

2. Die Zulassung von optimierendem Code durch verbesserte Anordnung und zusätzliche Instruktionen.

Die im Abschnitt „Compilation of Predicate Structure“ beschriebene Stufe vollzieht diese beiden Schritte gleichzeitig. Genau das aber macht einen Beweis der Äquivalenz dieser Stufe mit der vorhergehenden zunächst sehr schwierig.

Wir haben darum eine Vorstufe betrachtet, in welcher die PROLOG-Prozedur-Struktur zunächst eins zu eins in Zwischencode übersetzt wird und optimierender Code noch nicht zugelassen ist. Für diese Stufe haben wir dann die Äquivalenz zur vorhergehenden formal bewiesen!

Da die Dokumentation dieses Beweises das Ziel unserer Ausführungen darstellt, werden wir uns im folgenden auf eine Einführung in die *nichtoptimierende* Prozedurübersetzung[8] beschränken (wiederum äußerst informell). Es sei noch einmal darauf hingewiesen, daß trotz dieser Einschränkungen ein Fehler in der dynamischen Algebra und die Unvollständigkeit der Compiler-Annahme aufgedeckt und behoben werden konnten!

Wir führen jetzt drei neue Instruktionen ein: try_me_else(N), retry_me_else(N) und trust_me, wobei N eine Codezeile ist (aus dem Universum CODEAREA). try_me_else(N) entspricht dem früheren Modus Try und veranlaßt die Erzeugung eines neuen Choicepoints mit Wiederaufsetzpunkt N. retry_me_else(N) entspricht dem früheren Modus Retry, falls die aktuelle Klausel noch nicht die letzte in der Prozedurdefinition ist, und veranlaßt die Wiederverwendung eines Choicepoints, insbesondere das Überschreiben des Wiederaufsetzpunktes mit N. trust_me entspricht dem Modus Retry, falls die aktuelle Klausel die letzte in der Prozedurdefinition ist, und veranlaßt das „popen“ eines Choicepoints.

Eine *einelementige* Prozedurdefinition sei dargestellt wie bisher: als *einzelne* Zeile mit einer Klausel. Die Darstellung einer Prozedurdefinition mit $n \geq 2$ Klauseln sieht schematisch wie folgt aus:

$$
\begin{array}{rc}
\mathrm{N}_1: & \mathrm{try\_me\_else}(\mathrm{N}_2) \\
 & \vdots \\
\mathrm{N}_2: & \mathrm{retry\_me\_else}(\mathrm{N}_3) \\
 & \vdots \\
\mathrm{N}_{n-1}: & \mathrm{retry\_me\_else}(\mathrm{N}_n) \\
 & \vdots \\
\mathrm{N}_n: & \mathrm{trust\_me}
\end{array}
$$

[8] Für die Kenner von [BörgerRosenzweig]: Wir lassen noch keinen Parallelcode in Form von try-retry-trust Ketten zu und auch keine „nested chains“, sondern nur “flat chains“.

In den Zeilen N_2 bis N_{n-1} stehen retry_me_else-Instruktionen. Alle Zeilen N_1 bis N_n werden gefolgt von Zeilen, die eine Klausel enthalten. Die so entstandene Prozedur-Stuktur ist nicht mehr linear, sondern verzeigert.

Man sieht: die PROLOG-Klauseln sind noch durch sich selbst repräsentiert, aber jeder vorangestellt ist eine Instruktion, die entscheidende Aktionen der abstrakten Maschine steuert. Dadurch sind die Modi Try und Retry, aber auch (wie wir noch sehen werden) Select und Call, überflüssig geworden. Die Steuerung durch Modi der dynamischen Algebra wird ersetzt durch Steueranweisungen im Code.

Dies funktioniert natürlich nur, wenn man vom Compiler verlangt, daß er jede Prozedurdefinition *korrekt* übersetzt im Sinne des beschriebenen Schemas. Diese Forderung kommt in der Compiler-Annahme zum Ausdruck. Zu ihrer Formulierung verwenden [BörgerRosenzweig] eine Hilfsfunktion „chain“, die wie folgt definiert[9] ist:

chain: CODEAREA → CODEAREA*

```
chain(Ptr) = IF (code(Ptr) = try_me_else(N)
                    or code(Ptr) = retry_me_else(N))
             THEN [Ptr+ | chain(N)]
             ELSIF code(Ptr) = trust_me
             THEN [Ptr+]
             ELSE [Ptr]
```

Man beachte bitte, daß diese Funktion rekursiv ist und somit ihre intendierte Semantik, der minimale Fixpunkt, *partiell*. Dies ist kein unwichtiges Detail, sondern eine wichtige Eigenschaft, die die Formalisierbarkeit der Compiler-Annahme in erststufiger Prädikatenlogik verhindert.

Da die von [BörgerRosenzweig] getroffene Compiler-Annahme Gegenstand ausführlicher Diskussion sein wird, erlauben wir uns, sie wörtlich zu zitieren:

> „Compiler-Assumption: The list *chain(pocdef(G,Db))* contains pointers to all candidat clauses for G in Db, in the right ordering.“

Diese Annahme ist nicht vollständig, wie wir noch zeigen werden (siehe Abschn.3.2.2).

Jetzt wollen wir die dynamische Algebra angeben, die den beschriebenen Code interpretiert. Dazu noch einige Vorbemerkungen: Da dieProgrammzeilen jetzt nicht mehr nur auf Klauseln zeigen, ist die Bezeichnung „clauseline“ nicht mehr sinnvoll. Statt dessen werden *cll* und *cllreg* in p und *preg* umbenannt. Aus technischen Gründen benötigen wir eine ausgezeichnete Zeile *start* . Die Funktion *code* (vormals *clause*) liefert jetzt zu jeder Zeile p entweder eine Instruktion, eine Klausel, ein spezielles Element *code_of_start* oder ein *nil*[10]. Die Modi in den Einstiegsbedingungen können dann ersetzt werden durch:

$p = start$ statt $mode = call$,
$code(p) \in CLAUSE$ statt $mode = Enter$,
$code(p) = try\_me\_else(N)$ statt $mode = Try$, und
$code(p) = retry\_me\_else(N)$ bzw. $code(p) = trust\_me$ statt $mode = Retry$.

[9] Vereinfacht durch die Beschränkung auf die hier zugelassenen Instruktionen und auf „flat chains“.
[10] Dies steht für „keine passende Klausel vorhanden“.

Wo früher der *mode* neu gesetzt wurde, wird jetzt das *preg* aktualisiert, und die Compiler-Annahme muß garantieren, daß sich beide Aktionen entsprechen!!

Die Regeln lauten:[11]

Abkürzungen

goal ≡ fst(fst(decglseqreg))
cutpt ≡ snd(fst(decglseqreg))
act ≡ fst(goal)
cont ≡ [<rest(goal),cutpt> |rest(decglseqreg)]

final-success-rule

IF stop = 0 & decglseqreg = []
THEN stop := 1

goal-success-rule

IF stop = 0 & goal = []
THEN decglseqreg := rest(decglseqreg)

call-rule

IF stop = 0 & is_user_defined(act) & preg = start
THEN IF clause(procdef(act,db)) = nil
THEN backtrack
ELSE preg := procdef(act,db)
ctreg := breg

try-me-rule

IF stop = 0 & code(preg) = try_me_else(N)
THEN EXTEND STATE BY temp
WITH
breg := temp
b(temp) := breg
decglseq(temp) := decglseqreg
sub(temp) := subreg
p(temp) := N
ENDEXTEND
preg := preg+

enter-rule

IF stop = 0 & is_user_defined(act) & code(preg) ∈ CLAUSE
THEN LET clause = rename(clause(preg),vi)
LET unify = unify(act, hd(clause))

[11] Ein Vergleich mit den Regeln aus [BörgerRosenzweig] zeigt, daß wir hier die Aktion „restore_cutpoint“ durch ctreg := b(breg) ersetzt haben. Das ist korrekt, da wir nur flache Instruktionsketten betrachten: „In case of flat chains, with at most one choicepoint per clause, our old update ct := b(b) would do; ...“ [BörgerRosenzweig].

IF unify = nil
THEN backtrack
ELSE decglseqreg := subres([<bdy(clause),ctreg> |cont],unify)
subreg := subreg ∘ unify
vi := vi + 1
preg := start

where

backtrack ≡ IF breg = bottom
THEN stop := -1
ELSE perg := p(breg)

retry-me-rule

IF stop = 0 & code(preg) = retry_me_else(N)
THEN decglseqreg := decglseq(breg)
subreg := sub(breg)
ctreg := b(breg)
p(breg) := N
preg := preg+

trust-me-rule

IF stop = 0 & code(preg) = trust_me
THEN decglseqreg := decglseq(breg)
subreg := sub(breg)
breg := b(breg)
preg := preg+

true-rule

IF stop = 0 & act = true
THEN decglseqreg := cont

fail-rule

IF stop = 0 & act = fail
THEN backtrack

cut-rule

IF stop = 0 & act = !
THEN father := cutpt
decglseqreg := cont

initial

bottom	n_0
breg	n_0
decglseqreg	[<query,n_0 >]
subreg	[]
vi	0
stop	0
preg	start
db	program

Das Grundanliegen der hier dokumentierten Arbeit war es, den Beweis zu führen, daß diese beiden letztgenannten dynamischen Algebren operational äquivalent sind unter der Voraussetzung der Compiler-Annahme. Dieser Nachweis ist, nach der Verbesserung von Fehlern in der Vorlage, auch gelungen. Die Voraussetzung für die formale Verifikation ist aber zunächst die exakte Spezifikation der verwendeten Datenstrukturen und die Einbettung der dynamischen Algebren in eine „verifikationstaugliche“[12] Logik. Dies für alle bisher beschriebenen Stufen zu bewerkstelligen, war der zweite Schwerpunkt der Arbeit.

2.5 Umsetzung in KIV

In diesem Abschnitt wollen wir die Konzepte vorstellen, die es erlauben, die besprochene Aufgabenstellung formal so zu fassen, daß eine Verifikation mit dem KIV-System möglich ist. Zunächst stellen wir allgemein die Mechanismen zum Aufbau von strukturierten algebraischen Spezifikationen vor, so wie sie dann in Kapitel 3.1 benutzt werden. Dann beschreiben wir eine Vorgehensweise, die dynamische Algebren zugänglich macht für Methoden der Softwareverifikation.

2.5.1 Strukturierte Spezifikationen

Mit den Mitteln der strukturierten Spezifikation lassen sich kleinere Spezifikationen zu größeren, komplexen Systemen zusammenfügen. Das KIV-System bietet zu diesem Zweck die hier beschriebenen Strukturierungsmöglichkeiten an.
Jede strukturierte Spezifikation hat eine der folgenden Formen:

Basisspezifikation :
Diese besteht aus einer Signatur und einer Liste von Axiomen. Dies ist die Grundform für Spezifikationen.

Anreicherung :
Die Signatur einer Spezifikation wird um neue Elemente angereichert, ebenso die Menge der Axiome.

Vereinigung :
Geschrieben auch als: „Spec + Spec′“. Die Signaturen und Axiomenlisten der Spezifikationen werden miteinander vereinigt.

Generische Spezifikation :
Diese Möglichkeit ist besonders wichtig. Mit Hilfe generischer Spezifikationen können generische abstrakte Datentypen spezifiziert werden. So ist es z.B. möglich, von dem Elementtyp einer Liste oder eines Kellers zu abstrahieren, indem er als Parameter in die Spezifikation einfließt. Mit einer **Aktualisierung** (s.u.) kann der Elementtyp dann konkretisiert werden. Man kann so Listen von natürlichen Zahlen und Listen von Strings als zwei verschiedene Aktualisierungen ein und derselben generischen Spezifikation notieren.
Die Bestandteile des Tupels sind:

[12] Dies meinen wir auf den augenblicklichen Stand der Dinge bezogen. Ein Verifikationssystem für originale dynamische Algebren ist durchaus wünschenswert, wenn auch noch nicht zur Hand.

1. eine strukturierte Parameterspezifikation,
2. eine Liste von strukturierten Spezifikationen, die von der generischen benutzt werden,
3. eine Signatur und eine Axiomenliste, beide zusammen als *target* bezeichnet.

Aktualisierung :
Hier wird eine generische Spezifikation GenSpec aktualisiert mit einer strukturierten Spezifikation Spec_a, was ungefähr einer Ersetzung der Parameterspez. von GenSpec durch Spec_a entspricht. In dem Morphismus Morph wird jedem Signaturelement des Parameters ein Signaturelement aus Spec_a zugewiesen. Außerdem werden den restlichen Signaturelementen von GenSpec *neue* Namen zugewiesen. Das ist nötig, weil das KIV-System keinen Polymorphismus zuläßt. Die Operationen auf Listen natürlicher Zahlen müssen andere Namen haben als die Operationen auf Listen von Strings.
GenSpec ist entweder eine normale generische Spezifikation (s.o.) oder eine generische Datenspezifikation (s.u.).

Basisdatenspezifikation :
Mit Datensortendefinitionen können frei erzeugte Datentypen, die wechselseitig rekursiv sind, abkürzend ohne explizite Axiome notiert werden. Die Konstruktoren und Selektoren für die Sorten lassen sich besonders einfach notieren.
Beispiel: Ein Keller über natürliche Zahlen läßt sich besonders kurz spezifizieren mit

```
nat   =  0
       | succ(pred: nat),
stack =  emptystack with is_empty
       | push(top: nat, pop: stack);
Variablen st : stack; n: nat;
```

Intern wird diese Darstellung expandiert: Es werden Axiome erzeugt für den Zusammenhang zwischen den Selektoren bzw. Prädikaten und den Konstruktoren. Die so definierten Datentypen sind **freely generated by** (s.u.). Das Besondere an den Datenspezifikationen ist, daß sie in den gängigen Programmiersprachen uniform implementierbar sind, weil für Datensortendefinitionen programmiersprachliche Konstrukte existieren (vergl. insbesondere 'datatype's in ML).

Generische Datenspezifikation : Hier kommt noch eine Parameterspez. dazu, die die gleiche Bedeutung hat wie bei der obigen generischen Spez. .

Umbenennung :
Mit der Umbenennung kann man Spezifikationen „doppeln“. Jedem Signaturelement wird ein neuer Name zugewiesen.

In den Spezifikationen werden explizit oder implizit Erzeugtheitsklauseln verwendet. Die Modelle von **generated-by**-Sorten sind auf (durch die angegebenen Konstruktoren) erzeugte eingeschränkt (lose Semantik). Auf diesen Sorten können in der Logik dann Induktionsprinzipien angewendet werden!

freely bedeutet: In dem Modell der Sorte gibt es für jedes Element genau eine Darstellung mittels der angegebenen Konstruktoren. In unserem Beispiel gibt es also für jedes Element der Sorte stack einen Term, der nur aus den Konstruktoren emptystack und push aufgebaut (**generated**) ist; dieser Term ist eindeutig (**freely**).

2.5.2 Dynamische Algebren und Dynamische Logik

Um die Lemmata, die die Äquivalenz zweier dynamischer Algebren postulieren, formal zu fassen, benötigen wir eine Logik, die Aussagen treffen kann über dynamische Algebren oder ein Äquivalent. Da man dynamische Algebren auf Programme abbilden kann, wie wir gleich erläutern, wird dieser Zweck von Programm-Logiken erfüllt, speziell von der „Dynamischen Logik". Diese wird hier nur angerissen. Darüber hinaus verweisen wir auf [Reif]. Ausschlaggebend für die Wahl der Logik muß ein für sie geeignetes, leistungsfähiges Beweissystem sein. Als ein solches steht uns das KIV-System zur Verfügung. In der Syntax der Dynamischen Logik[13] stehen uns außer den üblichen prädikatenlogischen Formeln noch zwei zusätzliche Typen von Formeln zur Verfügung: $[\alpha]\varphi$ und $\langle\alpha\rangle\varphi$, wobei α ein Programm und φ eine weitere DL-Formel ist. Die informelle Semantik ist:

- $[\alpha]\varphi$: Wann immer das Programm α terminiert, gilt danach φ.
- $\langle\alpha\rangle\varphi$: Das Programm α terminiert und dananch gilt φ.

Diese Formeln können nun beliebig mit prädikatenlogischen kombiniert werden. Hierfür werden dem Leser noch genügend Beispiele begegnen. Die Syntax von α ist der von PASCAL sehr ähnlich, wobei es zusätzlich nochfolgende Programmkonstrukte gibt:

- **skip**: die leere Anweisung
- **abort**: die niemals terminierende Anweisung
- **loop** α **times** κ: ein Konstrukt zum induktiven Beweis von while-Formeln

Der in KIV verwendete Kalkül ist ein Sequenzkalkül. Darum sind die Lemmata notiert als Sequenzen über Formeln Dynamischer Logik:

$$\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$$

Die Bedeutung einer Sequenz ist: Die Konjunktion der Formeln $\varphi_1, \ldots, \varphi_n$ impliziert die Disjunktion der Formeln $\psi_1, \ldots, \psi_m$.

Um nun dynamische Algebren als PASCAL-artige Programme α notieren zu können, benötigen wir eine Repräsentation der dynamischen Funktionen als Datenstruktur. Damit ist es dann möglich, statische Algebren aufzufassen als Zustände der Programmvariablen und die Updates darzustellen durch Variablenzuweisungen. Die unterschiedlichen dynamischen Funktionen werden spezifiziert als Aktualisierungen der generischen Spezifikation finitefun:

finitefun =
generic specification
parameter elemi-ii **target**
sorts finitefun, pairdomcod;
functions

constfun	:	elem"	→	finitefun	;
. ˆ .	:	finitefun × elem'	→	elem"	;
. $+_{fun}$.	:	finitefun × pairdomcod	→	finitefun	;
. / .	:	elem' × elem"	→	pairdomcod	;

variables func_2, func_1, func: finitefun; pdc: pairdomcod; ele'_0: elem';

[13] Im folgenden abgekürzt durch DL.

axioms

finitefun **generated by** constfun, $+_{fun}$;
pairdomcod **generated by** /;
constfun(ele") ^ ele' = ele",
func $+_{fun}$ ele' / ele" ^ ele' = ele",
ele' $\neq$ ele'$_0$ $\rightarrow$ func $+_{fun}$ ele' / ele" ^ ele'$_0$ = func ^ ele'$_0$

end generic specification

Mit finitefun wird eine Funktion repräsentiert durch ihren (endlichen) Graphen. constfun initialisiert die default-Ausgabe der „Funktion". ^ entspricht der Funktionsanwendung, die in den Programmen sehr häufig auftreten wird. Mit $+_{fun}$ wird dem Graphen ein neues Paar von Elementen aus dem domain und dem codomain hinzugefügt.

In dynamischen Algebren besteht die gesamte Signatur nur aus Funktionen. Es ist sicherlich nicht adäquat, diese alle durch finitefun-Datenstrukturen zu repräsentieren. Dies ist nämlich nur bei *mehr-als-null-stelligen* Funktionen notwendig, die außerdem tatsächlich *dynamisch* sind, d.h. irgendeinem Update unterliegen. Die anderen, *statischen* Funktionen bleiben auch bei uns Funktionen. Einstellige werden als Konstanten bzw. Variablen aufgefaßt, je nachdem, ob sie statisch oder dynamisch sind. Zuletzt fehlt uns noch die Einbettung dynamischer Universen (NODE, STATE) und des EXTEND-Konstruktes. Dazu spezifizieren wir Mengen und new-Funktionen für die Erweiterung der Mengen um *neue* Elemente:

set =
generic specification
parameter elem **target**
sorts set;
constants $@_{set}$: set;
functions
. $+_{set}$. : set $\times$ elem $\rightarrow$ set ;
. $-_{set}$. : set $\times$ elem $\rightarrow$ set ;
predicates . inset . : elem $\times$ set;
variables ele$_2$, ele$_1$, ele: elem; se$_2$, se$_1$, se: set;

axioms

set **generated by** $@_{set}$, $+_{set}$;
$\neg$ ele inset $@_{set}$,
ele$_1$ inset se $+_{set}$ ele$_2$ $\leftrightarrow$ ele$_1$ = ele$_2$ $\vee$ ele$_1$ inset se,
ele$_1$ inset se $-_{set}$ ele$_2$ $\leftrightarrow$ ele$_1$ $\neq$ ele$_2$ $\wedge$ ele$_1$ inset se,
se$_1$ = se$_2$ $\leftrightarrow$ ($\forall$ ele.ele inset se$_1$ $\leftrightarrow$ ele inset se$_2$)

end generic specification

node =
specification
sorts nodesort;
variables no: nodesort;
end specification

state =
specification
sorts statesort;
variables st: statesort;
end specification

nodeset =
actualize set **with** node **by morphism**
elem → nodesort, set → nodeset, $@_{set} \rightarrow @$, $+_{set} \rightarrow +_{ns}$, $\text{-}_{set} \rightarrow \text{-}_{ns}$, inset → inn, ele → no, se → ns
end actualize

stateset =
actualize set **with** state **by morphism**
elem → statesort, set → stateset, $@_{set} \rightarrow @_s$, $+_{set} \rightarrow +_s$, $\text{-}_{set} \rightarrow \text{-}_s$, inset → ins, ele → st, se → s
end actualize

enrnodeset =
enrich nodeset **with**
constants root : nodesort;
functions new : nodeset → nodesort ;

axioms

¬ new(ns) inn ns,
new(@) = root

end enrich

enrstateset =
enrich stateset **with**
constants bottom : statesort;
functions snew : stateset → statesort ;

axioms

snew($@_s$) = bottom,
¬ snew(s) ins s

end enrich

Das Ergebnis der Transformation ist für die betrachteten dynamischen Algebren zu finden in Kapitel 3.3.

Kapitel 3

Formalisierung der Beweisaufgabe

Bisher haben wir den Leser eingeführt a) in das Objekt unserer Korrektheitsuntersuchungen, nämlich die in [BörgerRosenzweig] semi-formal beschriebenen Datenstrukturen, dynamischen Algebren, Compiler-Annahmen und Äquivalenzaussagen, und b) in die prizipiellen Möglichkeiten einer Umsetzung im KIV-System. In diesem Kapitel wird nun das konkrete Ergebnis dieser Umsetzung vorgestellt.

Zunächst erläutern wir die Spezifikation der benutzten Datentypen. Dadurch wird zum einen der Spezifikationsvorgang verdeutlicht, der typisch ist für die Softwareverifikation. Zum anderen sind die später folgenden Programme und insbesondere die Beweise leichter zu verstehen, wenn die verwendete Signatur ungefähr bekannt ist.

Danach werden die Compiler-Annahmen diskutiert und formalisiert, die dynamischen Algebren in Programme übertragen und schließlich die Äquivalenzlemmas in Dynamischer Logik formuliert.

3.1 Spezifikation

Hier geben wir die wichtigsten Spezifikationen wieder für all die Datentypen, die in Kapitel 2 nur angedeutet (oder völlig unausgesprochen benutzt) wurden. Einige sind zum Zweck der Darstellung vereinfacht.

Die Signatur aus [BörgerRosenzweig] konnte nicht buchstabengetreu beibehalten werden, die Abweichungen sind aber selbsterklärend.

3.1.1 Generische Grundsorten

Auf S.27 haben wir bereits die generische Spezifikation von Mengen angegeben. Außer dieser werden im folgenden vor allem generische Spezifikationen von Listen und von Paaren benötigt, um aus deren Aktualisierungen komplexe Datenstrukturen aufzubauen (wie z.B. die *decorated goal sequence*).

```
 list =
generic data specification
    parameter elem
    list =  nil
          | . +l . (car : elem, cdr : list)
          ;
```

variables l: list;
order predicates . $\ll$. : list $\times$ list;
end generic data specification

pair =
generic data specification
parameter elemi-ii
pair = mkpair (. .p1 : elem', . .p2 : elem");
variables vp: pair;
end generic data specification

3.1.2 PROLOG-Klauseln

Zunächst benötigen wir eine Defnition der PROLOG-Terme bzw. -Literale.

atom =
specification
sorts atomsort;
variables at: atomsort;
end specification

var =
specification
sorts varsort;
variables va: varsort;
end specification

term =
data specification
using nat, atom, var
term = struct (funct : atomsort, args : termlist) **with** is_struct
| mkconst (constsym : atomsort) **with** is_const
| mkvar (varsym : varsort) **with** is_var
| mklist (thelist : termlist) **with** is_list
| !
| true'
| fail'
;
termlist = the_one (and_only : term)
| tcons (tcar : term, tcdr : termlist)
;
variables trm_1, trm: term; trmli: termlist;
size functions tlen : termlist $\rightarrow$ nat ;
end data specification

Die Sorten term und termlist sind wechselseitig rekursiv und müssen darum in der gleichen Spezifikation definiert werden. Innerhalb von Termen treten Termlisten auf im Falle von

sogenannten „Strukturen“ und von PROLOG-Listen. PROLOG unterscheidet nicht syntaktisch zwischen Literalen und Termen. Als Konsequenz ist eine (von sich aus nicht aufrufbare) Variable als Literal in einer Klausel zulässig.

Beispiel: p(X,Y) :- X,q(Y).

Wenn X nach Unifikation mit einem aufrufbaren Literal aufgerufen wird, dann nennt man das einen Meta-Call. Umgekehrt ist jetzt auch z.B. der „!“ als Argument eines Terms sinnvoll, etwa in der Anfrage ? - p(!,a).

Die obige Spezifikation garantiert also schon auf dieser Ebene die Einbettung des Meta-Call's, ohne daß man sich bei der Beschreibung der abstrakten Maschine weiter darum kümmern müßte!

Die Unterscheidung zwischen Strukturen, Konstanten, Variablen und Listen mit den entsprechenden „is“-Prädikaten wird erst in der Stufe „Switching“ ([BörgerRosenzweig] 2.3.) richtig ausgenutzt. Um dort Strukturen von Konstanten zu differenzieren, sind nur mindestens einelementige Termlisten zugelassen worden.

Die Funktion tlen bestimmt die Anzahl der Listenkonstruktoren in einer Termliste. Mit ihrer Hilfe spezifizieren wir anschließend die Stelligkeit eines PROLOG-Terms. Auch das aus den Regeleinstiegsbedingungen bekannte is_user_defined wird jetzt definiert:

enrterm =
enrich term **with**
functions
arity : term → nat ;
predicates is_user_defined : term;

axioms

is_user_defined(trm) ↔ is_struct(trm) ∨ is_const(trm) ∨ is_var(trm)
∨ is_list(trm),
arity(trm) = tlen(args(trm))+1

end enrich

Für den Klauselrumpf benötigen wir Listen von Termen, die wir, mit der dcglseq im Hinterkopf, goal nennen:

goal =
actualize list **with** term **by morphism**
elem → term, list → goalsort, nil → gnil, $+_l$ → $+_g$, car → gcar, cdr →
gcdr
end actualize

Eine Klausel ist dann ein Paar aus einem Term und einer Termliste, die Selektoren dafür heißen hd und bdy:

clause =
actualize pair **with** goal **by morphism**
elem' → term, elem" → goalsort, pair → clausesort, mkpair → mkclause,
.p1 → hd, .p2 → bdy, vp → cl
end actualize

3.1.3 PROLOG-Programm und Zugriff

Bei [BörgerRosenzweig] gibt es ein abstraktes Universum PROGRAM und einen Adressraum CODE. Das aktuelle Programm wird bezeichnet mit $db \in$ PROGRAM. Die Kandidaten-Klauseln zu einem Term t erhält man mit *procdef(t,db)*. Dagegen ist die Funktion *clause*, die zu jeder Adresse die zugehörige Klausel liefert, im Original folgendermaßen sortiert:
clause: CODE → CLAUSE
Dies haben wir korrigiert, weil die *clause*-Funktion genau wie procdef nicht unabhängig vom aktuellen Programm sein kann:

program =
specification
  sorts program;
  variables db: program;
end specification

code =
specification
  sorts codesort;
  constants undefcode : codesort;
  variables co: codesort;
end specification

clausefun =
enrich code, program, clause **with**
  functions clause : codesort × program → clausesort ;
end enrich

procdef =
enrich term, program, codelist **with**
  functions procdef : term × program → codelist ;
end enrich

3.1.4 PROLOG-Berechnungszustände

Ein PROLOG-Berechnungszustand ist charakterisiert durch eine Substitution und eine Zielsequenz. Da sich die Verwendung von Substitution und Unifikation in den betrachteten Stufen nicht unterscheidet, bleibt die Substitution abstrakt und die Unifikation uninterpretiert:

subst =
specification
  v**sorts** substitution;
  constants $@_{su}$: substitution;
  functions . o . : substitution × substitution → substitution ;
  variables su_2, su_1, su: substitution;

axioms

$(su \circ su_1) \circ su_2 = su \circ su_1 \circ su_2$,
$su \circ @_{su} = su$,
$@_{su} \circ su = su$

end specification

substorfail =
data specification
using subst
substorfail = oksubst (the_subst : substitution)
| fail
;
variables subst: substorfail;
end data specification

unify =
enrich substorfail, term **with**
functions unify : term $\times$ term $\rightarrow$ substorfail ;
end enrich

Die mit Cut-Points bestückte Zielsequenz *decglseq* hatten wir angesehen als Liste von Paaren, bestehend aus *goals* und Zustandsknoten:

decgoal =
actualize pair **with** node, goal **by morphism**
elem' $\rightarrow$ goalsort, elem" $\rightarrow$ nodesort, pair $\rightarrow$ decgoal, mkpair $\rightarrow$ mkdecgoal,
.p1 $\rightarrow$ .1, .p2 $\rightarrow$ .2
end actualize

decgoallist =
actualize list **with** decgoal **by morphism**
elem $\rightarrow$ decgoal, list $\rightarrow$ decgoallist, nil $\rightarrow$ dnil, $+_l \rightarrow +_{dl}$, car $\rightarrow$ dcar, cdr
$\rightarrow$ dcdr
end actualize

subres war die Anwendung einer Substitution auf alle Terme einer *decglseq*:

substgaol =
enrich substterm, goal **with**
functions . $\hat{}_{sg}$. : substitution $\times$ goalsort $\rightarrow$ goalsort ;

axioms

$su \hat{}_{sg} gnil = gnil$,
$su \hat{}_{sg} trm +_g go = (su \hat{}_t trm) +_g su \hat{}_{sg} go$

end enrich

subres =
enrich decgoallist, substgaol **with**
functions subres : decgoallist $\times$ substitution $\rightarrow$ decgoallist ;

axioms

subres(dnil, su) = dnil,
subres(mkdecgoal(go, no) $+_{dl}$ dgl, su) = mkdecgoal(su $\hat{}_{sg}$ go, no) $+_{dl}$ subres(dgl, su)

end enrich

3.1.5 Dynamische Funktionen

In Abschnitt 2.5.2 hatten wir die generische Spezifikation finitefun zur Simulation von dynamischen Funktionen als Datenstrukturen vorgestellt. Um nun die konkreten dynamischen Funktionen zu erhalten, wird finitefun aktualisiert. Wir stellen nur exemplarisch die Spezifikation decglseq und father vor; sub, cands und cll kann sich der Leser dann leicht vorstellen:

decglseq =
actualize finitefun **with** node, decgoallist **by morphism**
elem' $\rightarrow$ nodesort, elem" $\rightarrow$ decgoallist, pairdomcod $\rightarrow$ pairnodedecgoallist, finitefun $\rightarrow$ decglseqfun, constfun $\rightarrow$ cdecglseq, $\hat{} \rightarrow \hat{}_d$, / $\rightarrow$ $/_d$, $+_{fun} \rightarrow +_d$, func $\rightarrow$ decglseq
end actualize

father =
actualize finitefun **with** node **by morphism**
elem' $\rightarrow$ nodesort, elem" $\rightarrow$ nodesort, pairdomcod $\rightarrow$ pairnodenode, finitefun $\rightarrow$ fatherfun, constfun $\rightarrow$ cfather, $\hat{} \rightarrow \hat{}_{fa}$, / $\rightarrow$ $/_{fa}$, $+_{fun} \rightarrow +_{fa}$, func $\rightarrow$ father
end actualize

3.1.6 Steuerung

Für die Steuerung der PROLOG-Interpretation benutzte die abstrakte Maschine zwei Typen von Modi:

```
 mode =
data specification
    modesort =  select
              | call
              ;
    variables mode: modesort;
end data specification

 stopmode =
data specification
    stopmodesort =  success
                  | failure
                  | run
                  ;
    variables stop: stopmodesort;
end data specification
```

3.1.7 Code für die Stack-Stufe

Die in der Stack-Interpretation von PROLOG verwendeten Datenstrukturen weisen kaum Unterschiede zu denen der Baum-Interpretation auf. Zusätzlich zu den Klauseln gibt es jetzt auch ein „null"-Element als mögliches Ergebnis der Funktion clause' (entspr. nil bei [BörgerRosenzweig]). Außerdem trägt der Addressraum, jetzt codearea genannt, eine Nachfolgerstruktur:

clauseornull =
data specification
using clause
clauseornull = mkclau (the_clau : clausesort)
| null
;
variables cln: clauseornull;
end data specification

codearea =
specification
sorts codearea;
constants failcode : codearea;
functions next : codearea → codearea ;
variables coa: codearea;
end specification

clause'fun =
enrich codearea, clauseornull, program' **with**
functions clause' : codearea × program' → clauseornull ;

axioms

clause'(failcode, db') = null

end enrich

(failcode ist die undefinierte Adresse für die Initialisierung und für versagendes hashing durch switch-on-structure in der switching-Stufe.)

Die anderen Spezifikationen werden nur insoweit abgeändert, als die Sorte node überall durch state ersetzt wird. Darum wird den betroffenen Bezeichnern bei Namenskollision ein „s" vorangestellt, so z.B. bei ssub, sdecglseq, scll usw.. Die Register müssen nicht spezifiziert werden, da sie später durch Programmvariablen repräsentiert werden.

3.1.8 Wiederverwendung von Choicepoints

Hier wird zur feineren Steuerung der Modus select durch drei neue Modi ersetzt.

rmode =
data specification
rmodesort = try
| retry

```
                | enter
                | call'
                ;
    variables rmode: rmodesort;
end data specification
```

3.1.9 Übersetzte Prozedur-Struktur

Jetzt spezifizieren wir den Zwischencode, der nun in stärkerem Maße die Kontrolle über die PROLOG-Interpretation übernimmt. Zu ihm gehören, neben den Klauseln und einem nil', noch die Instruktionen try_me_else und trust_me. Da die gesamte vorgestellte Spezifikation auch für weitere Stufen aus [BörgerRosenzweig] angelegt ist, sind hier die Instruktionen zur optimierenden Prozedurübersetzung gleich mitspezifiziert. Die Sorte instr_or_cl (instruction or clause) ist definiert durch:

```
 instr+clau =
data specification
    using nat, clause, codearea
    instr-or-cl =  try_me_else (where : codearea) with is_try_me
                | retry_me_else (where : codearea) with is_retry_me
                | trust_me with is_trust_me
                | mkcl (the_cl : clausesort) with is_clause
                | nil'
                | code_of_start

                | try' (what : codearea) with is_try
                | retry' (what : codearea) with is_retry
                | trust (what : codearea) with is_trust
                | switch_on_term (argindex : nat, vlabel : codearea, clabel : codearea,
                                  llabel : codearea, slabel : codearea) with is_sw_term
                | switch_on_constant (argindex : nat, tabsize : nat, table : codearea)
                                                                  with is_sw_const
                | switch_on_structure (argindex : nat, tabsize : nat, table : codearea)
                                                                  with is_sw_struct
                ;
    variables ioc: instr-or-cl;
end data specification
```

Die Funktion, die zu gegebener Adresse eine Klausel oder eine Instruktion zurückgibt, heißt jetzt code:

```
 codefun =
enrich codearea, program", instr+clau with
    constants start : codearea;
    functions code : codearea × program" → instr-or-cl ;

axioms
```

coa = start ↔ code(coa, db") = code_of_start,
code(failcode, db") = nil'

end enrich

3.1.10 Prozedurdefinitions-Tabelle

Bisher haben wir uns in den Spezifikationen so nah wie möglich an [BörgerRosenzweig] gehalten. Hier aber werden wir von der Vorlage abweichen und wollen das auch begründen.

Die in der ersten dynamischen Algebra verwendete *abstrakte* Funktion procdef hat die Aufgabe, zu gegebenem Literal G und Programm Db die Liste der Adressen aller *Kandidaten-Klauseln* zurückzugeben. Wir sind noch nicht darauf eingegangen, was das eigentlich bedeutet: Als *Kandidaten* kann man z.B. alle Klauseln ansehen, deren Köpfe das gleiche führende Prädikatensymbol besitzen wie G. Gerade diese Klauselmenge wird oft in PROLOG-Beschreibungen als „Prozedur" bezeichnet. Als *Kandidaten* kann man aber auch lediglich diejenigen Klauseln ansehen, deren Köpfe insgesamt mit G unifizierbar sind.

[BörgerRosenzweig] verlangen von der Liste der Kandidaten-Klauseln, daß sie *mindestens* die unifizierenden Klauseln und *höchstens* diejenigen mit passenden Prädikatensymbol enthält. Dazwischen ist alles erlaubt. procdef kann die Information, die in dem Literal G steckt, beliebig genau ausnutzen bei der Auswahl der Klauseln.

In der zweiten Stufe, der Interpretation durch Stackverarbeitung, definieren [BörgerRosenzweig] eine neue procdef-Funktion, die nun nicht mehr die Adressen aller Kandidaten-Klauseln liefert, sondern nur noch eine Einstiegsadresse. Die gesamte Liste enthält man mit der Hilfsfunktion clls (siehe S.15), die alle Adressen von dem Einstieg bis zur nächsten Zeile mit Eintrag nil zurückgibt. Die zu diesen Adressen gehörenden Klauseln sollen wieder genau die oben diskutierten Kandidaten-Klauseln ergeben:„...assuming now that *clls(procdef(G,Db))* yields the same as *procdef(G,Db)* of previous section, i.e. the list of (pointers to) all candidate clauses,..." [BörgerRosenzweig]

Wenn aber das alte procdef genauer differenziert als nur nach dem führenden Prädikatensymbol, dann kann man durch bloßes Aufsammeln aufeinanderfolgender Adressen nicht die gleichen Kandidaten-Klauseln erhalten. Wir demonstrieren das an einem Beispiel:

Angenommen, procdef (das alte) berücksichtigt bei der Klauselauswahl außer dem Prädikatensymbol noch den führenden Funktor des ersten Argumentes. Gegeben sei dann das Programm:

1: p(f(x)):- q.
2: p(g(x)):- r.
3: p(y) :- s.
4: nil

und die Anfrage ? - p(f(a)).
Dann sind die Kandidaten-Klauseln gegeben in der ersten Stufe durch

$$procdef(p(f(a)),Db) = [1,3]$$

und in der zweiten Stufe durch

$$clls(procdef(p(f(a)),\ Db)) = [1,2,3],$$

weil 1 und 3 auch hier enthalten sein müssen, aber clls nicht „springt". Die geforderte Übereinstimmung läßt sich also i.a. nicht erreichen.

Desweiteren könnte bei obigem Beispiel die neue procdef-Funktion das Literal $p(f(a))$ nur dann genauer ausnutzen, wenn die Klauseln vorher dupliziert würden, so daß unterschiedliche Kandidaten-Listen in unterschiedlichen Blöcken im Code stehen. Es handelt sich aber bei dem Code für die zweite Stufe nach [BörgerRosenzweig] noch um *unübersetzten* Quellcode[1].

Somit ist klar, daß duplizierter Code (zum Zwecke der feineren Kandidatenauswahl) nicht vorgesehen ist. Daran wollen wir festhalten, selbst wenn wir in Kürze doch noch einen Compiler zwischen die ersten beiden Stufen schalten werden. Das Code-Design soll so bleiben, wie es beabsichtigt ist, insbesondere weil duplizierter Code nichts beiträgt zum Beweis weiterer Stufen.[2]

In der Konsequenz kann die Struktur des Literals jetzt nicht weiter ausgenutzt werden als bis zum führenden Prädikatensymbol. Es genügt daher, der neuen procdef-Funktion eben dieses Symbol zu übergeben, zusammen mit der Stelligkeit des Prädikates, weil PROLOG gleiche Symbole mit verschiedenen Stelligkeiten unterscheidet. Dies entspricht auch der Verwendung von procdef in den finalen WAM-Regeln (siehe[BörgerRosenzweig], Appendix 4:„Rules for the WAM“).

Um das Prädikatensymbol und die Stelligkeit zu gegebenem Literal zu erhalten, definieren wir die Funktion id mit:

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

Dabei ist die Sorte ident gegeben durch:

ident =
actualize pair **with** nat, atom **by morphism**
elem' → atomsort, elem" → nat, pair → ident, mkpair → mkident, .p1 →
atom, .p2 → ari
end actualize

Eine weitere Beobachtung betrifft die Übersetzung der Prozedurstruktur in verzeigerte Instruktionen. Bis dahin war es noch gut möglich, daß die procdef-Funktion durch eine Analyse des Programms die gesuchte Einstiegsadresse berechnet. Durch die verzeigerte Programmdarstellung ist aber diese Berechnung auf dem Zwischencode schwieriger; und im finalen WAM-Code ist sie gänzlich unmöglich, weil der einem Klauselkopf entsprechende Code das Prädikatensymbol gar nicht mehr repräsentiert. Ganz abgesehen davon ist es natürlich auch nicht sinnvoll, die Einstiegsadresse der Kandidaten-Klauseln immer wieder zu berechnen. Statt dessen ist es die Aufgabe des PROLOG-Compilers, diese Information in einer Art

[1] Wenn auch in den möglichen Modellen verfeinert gegenüber der ersten Stufe, wg. nil und +.

[2] Ohne dies hier vertiefen zu wollen, sei erwähnt, daß entgegen möglicher Vermutungen selbst beim Übergang zum Switching, d.h. der verfeinerten Klausel-Auswahl, eine feinere Klausel-Auswahl in vorhergehenden Stufen keine Vorteile für den Äquivalenzbeweis bringt. Das liegt auch daran, daß die WAM mittels Switching keine *beliebig* feine Klausel-Auswahl treffen kann, in der allerersten Stufe aber sogar Unifizierbarkeit als Kriterium erlaubt ist.

Symboltabelle abzulegen. Eine *abstrakte* procdef-Funktion würde dann die gesuchte Adresse in dieser Tabelle nachschlagen.

Eine andere mögliche Modellierung wäre die, procdef genau mit dieser Symboltabelle zu *identifizieren*. Der Compiler würde dann das procdef explizit berechnen. Wir wollen nun letztere Alternative wählen, um deutlicher als [BörgerRosenzweig] herauszustellen, daß die Einstiegsadressen zur Übersetzungszeit und nicht zur Laufzeit ermittelt werden müssen. Die Prozedurdefinitionstabelle läßt sich leicht definieren mittels einer Spezifikation, die uns schon zur Verfügung steht: finitefun (S.26). Denn dort wurde im wesentlichen nichts anderes definiert als Assoziationslisten.

procdeftab =
actualize finitefun **with** codearea, ident **by morphism**
elem' → ident, elem" → codearea, finitefun → procdeftable, constfun →
cprocdef, ^ → $^{\wedge}_{pd}$, / → $/_{pd}$, $+_{fun} \rightarrow +_{pd}$, func → procdeftab
end actualize

Die Anwendung von procdeftab auf ein ident (s.o.) ergibt eine Adresse aus codearea.

3.1.11 Compiler

Der in Abschnitt 3.2.1 noch zu erläuternde Compilationsschritt zwischen den ersten Stufen erfordert noch die Definition einer compile_1-Funktion.

tree->stack =
enrich tree+stack+f, comp1result **with**
functions compile_1 : program → comp1result ;
end enrich

comp1result ist ein Paar aus dem übersetzten Programm und einer Prozedurdefinitionstabelle.

comp1result =
actualize pair **with** program', procdeftab **by morphism**
elem' → program', elem" → procdeftable, pair → comp1result, ele' → db',
ele" → procdeftab, mkpair → pair_1, .p1 → .newdb, .p2 → .symtab
end actualize

compile_2 und comp2result für den Übergang von IV nach V gehen entsprechend.

3.2 Korrektur und Formalisierung der Compiler-Annahmen

3.2.1 Blockung der Klauseln

Wir haben schon erläutert, daß [BörgerRosenzweig] für die Stackverarbeitung von PROLOG eine Programmdarstellung annehmen, in der Prozeduren in zusammenhängenden Blöcken im Adressraum stehen, getrennt durch Zeilen mit Eintrag nil. Da sie hier keinen Übersetzungsschritt vorsehen, werden rückwirkend die Modelle des Quellcodes eingeschränkt. Das ist aber nicht notwendig, denn dies wäre ein sehr speziellller PROLOG-Dialekt, insbesondere wegen der nil-Klauseln. Statt dessen kann man diese Umsetzung dem Compiler überlassen.

Die notwendige Compiler-Annahme erhält man dann durch Abschwächung der Gleichung

$$clls(procdef(G,Db)) = procdef(G,Db)$$

aus [BörgerRosenzweig]: Aus dem vorderen Db wird dessen Compilat Db'. Statt der Gleichheit der Adresslisten genügt uns zunächst die Gleichheit der zugehörigen Klauseln. Dafür definieren wir die Funktionen mapclause und mapclause' (Spez. siehe Signaturindex), zur Fortsetzung von clause und clause' auf Adresslisten. als Zwischenergebnis erhalten wir:

$$mapclause'(clls(procdef(G,Db'))) = mapclause(procdef(G,Db))$$

Das ist immer noch zu stark. In Abschnitt 3.1.10 hatten wir dargelegt, daß das *procdef* der ersten Stufe Klauseln viel feiner auswählen kann als das *procdef* der zweiten Stufe. Um die obige Gleichheit zu retten, müßte man in die PROLOG-Semantik von [BörgerRosenzweig^a] eingreifen und die Bedeutung der „Kandidaten-Klauseln“ einschränken. Es ist aber niemals guter Stil, einem Compiler zuliebe die Semantik, und sei es nur minimal, zu ändern. Statt dessen fordern wir, daß die Kandidaten der zweiten Stufe *mindestens* die der ersten Stufe enthalten. Damit sind alle unifizierbaren Klauseln auf jeden Fall mit dabei. Dazu dient das „Teillisten“-Prädikat *subli_of*:

$$mapclause(pocdef(G,Db)) \quad subli\_of \quad mapclause'(clls(procdef(g,Db')))$$

Desweiteren wird procdef auf der linken Seite obiger Gleichung durch die compilierte Symboltabelle ersetzt und das Literal durch seinen identifier id(lit).

Bleibt zu überlegen, wie die Funktion clls (vergl. Def. auf S.15) darzustellen ist. Sie ist rekursiv und partiell. Letzteres zwar nur auf pathologischen Speicherbelegungen (z.B. wenn in den aufsteigenden Adressen kein nil gefunden wird), aber das hilft uns auf der logischen Seite nicht weiter. Wir können aber clls in der Dynamischen Logik als Programm auffassen, und zwar:

```
CLLS#(coa, db'; var cal)
begin
if clause'(coa, db') = null
then cal := canil
else begin CLLS#(next(coa), db'; cal);
           cal := coa +cal cal
     end
end
```

Die gesuchte Formalisierung dieser *neuen* Compiler-Annahme lautet dann als Sequenz:

comp-assum-1

$\text{compile}_1(\text{db}) = \text{pair}_1(\text{db', procdef'})$
$\vdash$
$\langle \text{clls\#(procdef' } \hat{}_{pd} \text{ id(lit), db';cal)} \rangle$
$\quad \text{mapclause(procdef(lit, db), db) subli\_of mapclause'(cal, db')}$

Im kommenden Abschnitt wird die Notwendigkeit, partielle Hilfsfunktionen formal korrekt zu behandeln, noch sehr viel deutlicher werden.

3.2.2 Prozedur-Übersetzung

In Abschnitt 2.4 hatten wir die Übersetzung der Prozedur-Struktur erläutert zusammen mit der Compiler-Annahme, in welcher die Hilfsfunktion chain verwendet wird. Wir wollen uns nochmals diese Funktion vor Augen halten:

chain: CODEAREA → CODEAREA*

```
chain(Ptr) = IF (code(Ptr) = try_me_else(N)
                    or code(Ptr) = retry_me_else(N))
             THEN [Ptr+ | chain(N)]
             ELSIF code(Ptr) = trust_me
             THEN [Ptr+]
             ELSE [Ptr]
```

Auch die Compiler-Annahme von [BörgerRosenzweig] wollen wir wieder wörtlich zitieren, um anschließend die Verbesserungen und die Formalisierung deutlich zu machen:

> „Compiler-Assumption: The list $chain(pocdef(G, Db))$ contains pointers to all candidat clauses for G in Db, in the right ordering."

In Verbesserung einer ersten kleinen Ungenauigkeit fügen wir die Datenbasis Db als zweites Argument zu chain hinzu, da sie auch für code innerhalb von chain gebraucht wird.

Auch chain ist rekursiv und partiell. Wenn die verzeigerten Instruktionen try_me_else und retry_me_else Zyklen erzeugen, dann kann der Wert von chain undefiniert sein. Dies ist aber kein formalistischer Schönheitsfehler, sondern durchaus gewollt. Denn gerade indem wir in der Compiler-Annahme die *Definiertheit* der *partiellen* Funktion chain postulieren (allerdings bisher nur implizit), verlangen wir von dem Compiler, daß er zyklenfreien Code erzeugt!

Weiter unten wird chain ebenso wie schon clls als Programm formuliert. Dann wird aus der Definiertheitsaussage eine Terminierungsbehauptung. Vorher aber muß noch die Definition von chain verbessert werden. Denn die Compiler-Annahme ist bisher unvollständig, d.h. es gibt Compiler, die zwar die Annahme erfüllen, aber falschen[3] Code ausgeben. Als Beispiel betrachten wir folgendes Programm (die bd_i kürzen die Klauselrümpfe ab):

1: p :- bd_1
2: p :- bd_2
3: p :- bd_3
4: p :- bd_4
5: nil

Die folgende Übersetzung erfüllt die Compiler-Annahme, wie der Leser selbst überprüfen möchte:

start: code_of_start
1: retry_me_else(3)
2: p :- bd_1
3: try_me_else(5)
4: p :- bd_2

[3] Dabei ist „falsch" u.a., was nicht dem Schema S.20 entspricht.

5: try_me_else(7)
6: p :- bd_3
7: p :- bd_4

Natürlich ist die Übersetzung trotzdem nicht korrekt. Das erste retry_me_else versucht, einen Choicepoint wiederzuverwenden, was mißlingen wird, da noch gar keiner angelegt ist. Erst das try_me_else legt einen an, aber dazu ist es dann schon zu spät. Das zweite try_me_else legt gar noch einen zweiten an innerhalb der gleichen Prozedurdefinition! Und zu guter Letzt fehlt noch das trust_me, das den ordnungsgemäß einzigen Choicepoint wieder vom Stack nehmen müßte. Die abstrakte Maschine würde auf diesen Code ein Chaos produzieren.

Das liegt natürlich daran, daß wir uns nicht an das auf S.20 angegebene Code-Schema gehalten haben. Dieses Schema muß also in irgendeiner Form *in* die Compiler-Annahme codiert werden!

Eine Möglichkeit, dies zu tun, ist, chain „schlauer“ zu machen. Wenn chain nicht nur beim Auftreten von Zyklen, sondern auch bei jeder anderen Abweichung vom Schema undefiniert wird („abstürzt“), dann könnten wir mit der *Annahme*, chain sei definiert (und diese Annahme brauchen wir sowieso s.o.) auch das Schema zwingend vorschreiben. Wenn jetzt chain als Programm notiert wird, dann stellt die abort-Anweisung gerade den verordneten Absturz dar. Aus der späteren Terminierungsaussage läßt sich dann folgern, daß die abort-Zweige nicht betreten werden!

Aus diesen Gründen muß das Programm natürlich strukturierter sein als obige Funktion. Als Einstieg ist nur eine Klauseladresse (für einelementige Prozeduren), eine nil-Adresse (wenn keine passende Prozedur existiert), und die Adresse einer try_me-Instruktion erlaubt. Im letzten Fall muß unmittelbar auf das try_me eine Klausel folgen. Anschließend ist nur noch ein retry_me oder ein trust_me zulässig usw. Die jeweils auf die Instruktion folgenden Klausel*adressen* werden rekursiv aufgesammelt und als cal (codearea list) zurückgegeben.

```
CHAIN#(coa, db''; var cal)
begin
var instr  = code(coa, db'')
in if is_try_me(instr)
   then CHAIN-TRY-ME#(coa, db''; cal)
   else if is_clause(instr)
        then cal := coa +cal canil
        else if instr = nil'
             then cal := canil
             else abort
end;

CHAIN-TRY-ME#(coa, db''; var cal)
begin
var instr  = code(coa, db''),
    follow = code(next(coa), db'')
in if is_try_me(instr)
   then if is_clause(follow)
        then begin
             CHAIN-RETRY-ME#(where(instr), db''; cal);
             cal := next(coa) +cal cal
```

```
            end
        else abort
   else abort
end;

CHAIN-RETRY-ME#(coa, db''; var cal)
begin
var instr  = code(coa, db''),
    follow = code(next(coa), db'')
in if is_retry_me(instr)
   then if is_clause(follow)
        then begin
             CHAIN-RETRY-ME#(where(instr), db''; cal);
             cal := next(coa) +cal cal
             end
        else abort
   else if is_trust_me(instr)
        then if is_clause(follow)
             then cal := next(coa) +cal canil
             else abort
        else abort
end
```

Mit dieser Definition können wir die Compiler-Annahme formalisieren. Der Begriff der Kandidaten-Klauseln bleibt diesmal konstant interpretiert, wir verlangen also die Gleichheit mit den Kandidaten-Klauseln der vorhergehenden Stufe. Diese hatten wir erhalten mittels clls# und mapclause'. Die Symboltabellen procdef' und procdef" auf beiden Ebenen sind natürlich verschieden. Wir formulieren jetzt die Annahme an die Funktion compile_2:

comp-assum-2

$$\begin{array}{l} \text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"}) \\ \vdash \\ [\text{clls\#}(\text{procdef'} \mathbin{\hat{}}_{pd} \text{id(lit)}, \text{db'}; \text{cal}_1)] \\ \quad \langle \text{chain\#}(\text{procdef"} \mathbin{\hat{}}_{pd} \text{id(lit)}, \text{db"}; \text{cal}_2) \rangle \\ \quad\quad \text{mapcode}(\text{cal}_2, \text{db"}) = \text{mapclause'}(\text{cal}_1, \text{db'}) \end{array}$$

Man beachte, daß wir nicht die Terminierung von clls# verlangen. Das liegt daran, daß diese Compiler-Annahme die Funktion compile_2 spezifiziert. Daß aber clls# terminiert, kann nur compile_1 garantieren, nicht aber compile_2. Für den Äquivalenzbeweis aber benötigen wir auch die Terminierung von clls#. Diese folgt aus der ersten Compiler-Annahme S.40

Mit anderen Worten: Wir benötigen zwingend die Eigenschaft des Urbildes von compile_2, Bild von compile_1 zu sein. Dazu verwenden wir eine Abschwächung der ersten Compiler-Annahme:

weak-comp-assum-1

$$\vdash \langle \text{clls\#}(\text{procdef'} \mathbin{\hat{}}_{pd} \text{id(lit)}, \text{db'}; \text{cal}_1) \rangle \text{true}$$

Aus beiden ergibt sich dann:

compass2+weak1

$$\begin{array}{l} \quad \mathrm{compile}_2(\mathrm{pair}_1(\mathrm{db'},\ \mathrm{procdef'})) = \mathrm{pair}_2(\mathrm{db''},\ \mathrm{procdef''}) \\ \vdash \\ \langle \mathrm{clls}\#(\mathrm{procdef'}\ \hat{}_{pd}\ \mathrm{id(lit)},\ \mathrm{db'};\ \mathrm{cal}_1)\rangle \\ \qquad \langle \mathrm{chain}\#(\mathrm{procdef''}\ \hat{}_{pd}\ \mathrm{id(lit)},\ \mathrm{db''};\ \mathrm{cal}_2)\rangle \\ \qquad\quad \mathrm{mapcode}(\mathrm{cal}_2,\ \mathrm{db''}) = \mathrm{mapclause'}(\mathrm{cal}_1,\ \mathrm{db'}) \end{array}$$

In dieser Form werden wir die Compiler-Annahme für den Äquivalenzbeweis verwenden.

Es sei noch bemerkt, daß die Unvollständigkeit der Ausgangs-chain *nicht* darin begründet ist, daß wir die chain-Definition aus [BörgerRosenzweig] abgekürzt haben, um eine Zwischenstufe zu betrachten. Im Gegenteil: Die originale chain-Funktion ist noch unvollständiger. Auch sie erfüllt bei obigem pathologischen Beispiel die Compiler-Annahme. Obendrein läßt sie eine beliebige Mischung von (re)try_me- und (re)try-Instruktionen zu, was aber in [BörgerRosenzweig] selbst ausgeschlossen wird.

3.3 Formalisierung der Äquivalenz

In Abschnitt 2.5.2 hatten wir schon erläutert, wie man dynamische Algebren in Dynamische Logik einbindet. Zunächst müssen die betrachteten Regelsysteme jeweils als Programm formuliert werden. Über diese Programme kann man dann mittels Dynamischer Logik Aussagen formulieren. Dies werden wir jetzt konkretisieren an den beiden dynamischen Algebren, deren Äquivalenz hier bewiesen werden soll. Wir wenden dazu folgendes Schema[4] an:

- Man trennt die Regeleinstiegsbedingungen von den Regeln.
- Die verbleibenden Regeln werden als Prozeduren definiert. Dazu muß auch analysiert werden, welche Werte *nur* als Eingabe und welche *auch* als Ausgabe fungieren. Je nachdem werden die entsprechenden Variablen in die Call-by-Value- bzw. in die Call-by-Reference-Parameterliste aufgenommen. Außerdem erfordert die sequentielle Abarbeitung evtl. kleinere Änderungen an der Reihenfolge der Updates.
- Es wird eine BODY-Prozedur definiert, in der die Regeleinstiegsbedingungen (ohne *stop*) über geschachtelte if-then- else-Konstrukte einen Aufruf der jeweiligen Regel-Prozedur ansteuern.
- Dann wird eine EVAL-Prozedur definiert, die besteht aus einer Initialisierung und einer WHILE-Schleife mit *stop* = *run* als Bedingung und der BODY-Prozedur als Schleifenrumpf.

3.3.1 Die Stufe IV als Programm

Für die dynamische Algebra von S.17 sieht das Ergebnis folgendermaßen aus:[5]
(„D-...“ steht für **D**eterminacy Detection)

[4] siehe [SchellhornAhrendt]

[5] Sollte beim Lesen noch die eine oder andere Unklarheit bezüglich der Signatur auftreten, so verweisen wir auf den Signaturindex im Anhang.

```
D-EVAL#(db', procdef', goal; var subst)
begin
(: initialisation :)
var s = @s +s bottom,
    vi = 0,
    rmode = call',
    stop = run,
    breg = bottom,
    ctreg = bottom,
    ssub = cssub(@su), (: the empty subst for every node :)
    subreg = @su,
    sdecglseq = csdecglseq(sdnil),
    decglseqreg = mksdecgoal(goal,bottom) +sdl sdnil,
    scll = cscll(failcode),
    cllreg =  failcode,
    b = cb(bottom)
in (: main program :)
   begin
   while stop = run do
         D-BODY#(db',procdef'; s,vi,rmode,stop,breg,ctreg,ssub,subreg,
                  sdecglseq,decglseqreg,scll,cllreg,b);
   if stop = failure then subst := fail else subst := oksubst(subreg)
   end
end;

D-BODY#(db',procdef'; var s,vi,rmode,stop,breg,ctreg,
                         ssub,subreg,sdecglseq,decglseqreg,scll,cllreg,b)
begin
if decglseqreg = sdnil
then D-QUERY-SUCCESS#(; stop)
else var goal = (sdcar(decglseqreg)).s1
     in if goal = gnil then D-GOAL-SUCCESS#(; decglseqreg)
        else var act = gcar(goal),
                 scutpt = (sdcar(decglseqreg)).s2
             in var scont = mksdecgoal(gcdr(goal),scutpt) +sdl sdcdr(decglseqreg)
                in if is_user_defined(act)
                   then if rmode = call'
                        then D-CALL#(act,procdef',breg,db';
                                     cllreg,rmode,ctreg,stop)
                        else if rmode = try
                        then D-TRY#(cllreg,subreg,decglseqreg,db';
                                    scll,ssub,breg,b,sdecglseq,s,rmode)
                        else if rmode = enter
                        then D-ENTER#(db',act,scont,cllreg,breg,ctreg;
                                      subreg,decglseqreg,vi,stop,rmode)
                        else (: rmode = retry :)
```

```
                              D-RETRY#(db',sdecglseq,b,ssub;
                                       cllreg,subreg,scll,breg,
                                       ctreg,decglseqreg,rmode)
                    else if act = true'
                         then D-TRUE#(scont; decglseqreg)
                         else if act = fail'
                         then D-FAIL#(breg; stop, rmode)
                         else (: act = ! :)
                              D-CUT#(scont,scutpt; breg, decglseqreg)
end;

D-QUERY-SUCCESS#(var stop)
begin stop := success end;

D-GOAL-SUCCESS#(var decglseqreg)
begin decglseqreg := sdcdr(decglseqreg) end;

D-CALL#(act, procdef', breg, db'; var cllreg,rmode,ctreg,stop)
begin
if clause'(procdef' ^pd id(act), db') = null
then D-BACKTRACK#(breg; stop, rmode)
else begin cllreg := procdef' ^pd id(act);
           rmode := try;
           ctreg := breg
     end
end;

D-TRY#(cllreg,subreg,decglseqreg,db'; var scll,ssub,breg,b,sdecglseq,s,rmode)
begin
rmode := enter;
if clause'(next(cllreg), db') # null
then var temp = snew(s)
     in begin
        s := s +s temp;
        b := b +b temp /b breg;
        sdecglseq := sdecglseq +sd temp /sd decglseqreg;
        ssub := ssub +su temp /su subreg;
        scll := scll +sc temp /sc next(cllreg);
        breg := temp
        end
end;

D-ENTER#(db',act,scont,cllreg,breg,ctreg; var subreg,decglseqreg,vi,stop,rmode)
begin
var cla = ren(the_clau(clause'(cllreg, db')),vi)
in var uni = unify(act,hd(cla))
   in if uni = fail
```

```
      then D-BACKTRACK#(breg; stop, rmode)
      else begin
           decglseqreg :=      ssubres(mksdecgoal(bdy(cla),ctreg)
                          +sdl scont,the_subst(uni));
           subreg := subreg o the_subst(uni);
           vi := vi +1;
           rmode := call'
           end
end;

D-BACKTRACK#(breg; var stop, rmode)
begin
if breg = bottom
then stop := failure
else rmode := retry
end;

D-RETRY#(db',sdecglseq,b,ssub;
         var cllreg,subreg,scll,breg,ctreg,decglseqreg,rmode)
begin
decglseqreg := sdecglseq ^sd breg;
subreg := ssub ^su breg;
cllreg := scll ^sc breg;
scll := scll +sc breg /sc next(scll ^sc breg);
ctreg := b ^b breg;
rmode := enter;
if clause'(scll ^sc breg, db') = null
then breg := b ^b breg
end;

D-TRUE#(scont; var decglseqreg)
begin
decglseqreg := scont
end;

D-FAIL#(breg; var stop, rmode)
begin
D-BACKTRACK#(breg; stop, rmode)
end;

D-CUT#(scont,scutpt; var breg, decglseqreg)
begin
breg := scutpt;
decglseqreg := scont
end;
```

3.3.2 Die Stufe V' als Programm

Mit der gleichen Vorgehensweise erhalten wir die Programmdarstellung der dynamischen Algebra von S.22:
(„CP-..." steht für **C**ompilation of **P**rocedure Structure)

```
CP-EVAL#(db'', procdef'', goal; var subst)
begin
(: initialisation :)
var s' = @s +s bottom,
    vi' = 0,
    stop' = run,
    preg =  start,
    breg' = bottom,
    ctreg' = bottom,
    ssub' = cssub(@su), (: the empty subst for every state :)
    subreg' = @su,
    sdecglseq' = csdecglseq(sdnil),
    decglseqreg' = mksdecgoal(goal,bottom) +sdl sdnil,
    p = cp(failcode),
    b' = cb(bottom)
in (: main program :)
   begin
   while stop' = run do
         CP-BODY#(db'',procdef''; s',vi',stop',breg',ctreg',ssub',subreg',
                  sdecglseq',decglseqreg',p,preg,b');
   if stop' = failure then subst := fail else subst := oksubst(subreg')
   end
end;

CP-BODY#(db'',procdef''; var s',vi',stop',breg',ctreg',
                         ssub',subreg',sdecglseq',decglseqreg',p,preg,b')
begin
if decglseqreg' = sdnil
then CP-QUERY-SUCCESS#(; stop')
else var goal = (sdcar(decglseqreg')).s1
     in if goal = gnil then CP-GOAL-SUCCESS#(; decglseqreg')
        else var act = gcar(goal),
                 scutpt = (sdcar(decglseqreg')).s2
             in var scont =     mksdecgoal(gcdr(goal),scutpt)
                              +sdl sdcdr(decglseqreg')
                in if is_user_defined(act)
                   then if preg = start
                        then CP-CALL#(act, procdef'', breg', p, db'';
                                      preg,ctreg',stop')
                        else if is_try_me(code(preg, db''))
                        then CP-TRY-ME#(subreg',decglseqreg',db'';
```

```
                                        p,ssub',breg',b',sdecglseq',s',preg)
                         else if is_clause(code(preg, db''))
                         then CP-ENTER#(db'',act,scont,breg',ctreg',p;
                                       subreg',decglseqreg',vi',stop',preg)
                         else if is_retry_me(code(preg, db''))
                         then CP-RETRY-ME#(db'',sdecglseq',b',ssub',breg';
                                           preg,subreg',p,ctreg',decglseqreg')
                         else (: is_trust_me(code(preg, db'')) :)
                              CP-TRUST-ME#(sdecglseq',b',ssub';
                                           preg,subreg',breg',
                                           ctreg',decglseqreg')
                    else if act = true'
                         then CP-TRUE#(scont; decglseqreg')
                         else if act = fail'
                         then CP-FAIL#(breg',p; stop', preg)
                         else (: act = ! :)
                              CP-CUT#(scont,scutpt; breg', decglseqreg')
end;

CP-QUERY-SUCCESS#(var stop')
begin stop' := success end;

CP-GOAL-SUCCESS#(var decglseqreg')
begin decglseqreg' := sdcdr(decglseqreg') end;

CP-CALL#(act, procdef'', breg', p, db''; var preg,ctreg',stop')
begin
if code(procdef'' ^pd id(act), db'') = nil'
then CP-BACKTRACK#(breg', p; stop', preg)
else begin preg := procdef'' ^pd id(act);
           ctreg' := breg'
     end
end;


CP-TRY-ME#(subreg',decglseqreg',db''; var p,ssub',breg',b',sdecglseq',s',preg)
begin
var temp = snew(s')
in begin
   s' := s' +s temp;
   b' := b' +b temp /b breg';
   sdecglseq' := sdecglseq' +sd temp /sd decglseqreg';
   ssub' := ssub' +su temp /su subreg';
   p := p +p temp /p where(code(preg, db''));
   breg' := temp;
   preg := next(preg)
   end
```

```
end;

CP-ENTER#(db'',act,scont,breg',ctreg',p; var subreg',decglseqreg',vi',stop',preg)
begin
var cla = ren(the_cl(code(preg, db'')),vi')
in var uni = unify(act,hd(cla))
   in if uni = fail
      then CP-BACKTRACK#(breg', p; stop', preg)
      else begin
           decglseqreg' :=      ssubres(mksdecgoal(bdy(cla),ctreg')
                           +sdl scont,the_subst(uni));
           subreg' := subreg' o the_subst(uni);
           vi' := vi' +1;
           preg := start
           end
end;

CP-BACKTRACK#(breg', p; var stop', preg)
begin
if breg' = bottom
then stop' := failure
else preg := p ^p breg'
end;

CP-RETRY-ME#(db'',sdecglseq',b',ssub',breg';
             var preg,subreg',p,ctreg',decglseqreg')
begin
decglseqreg' := sdecglseq' ^sd breg';
subreg' := ssub' ^su breg';
ctreg' := b' ^b breg';
p := p +p breg' /p where(code(preg, db''));
preg := next(preg)
end;

CP-TRUST-ME#(sdecglseq',b',ssub'; var preg,subreg',breg',ctreg',decglseqreg')
begin
decglseqreg' := sdecglseq' ^sd breg';
subreg' := ssub' ^su breg';
ctreg' := b' ^b breg';
breg' := b' ^b breg';
preg := next(preg)
end;

CP-TRUE#(scont; var decglseqreg')
begin
decglseqreg' := scont
end;
```

```
CP-FAIL#(breg', p; var stop', preg)
begin
CP-BACKTRACK#(breg', p; stop', preg)
end;

CP-CUT#(scont,scutpt; var breg', decglseqreg')
begin
breg' := scutpt;
decglseqreg' := scont
end;
```

3.3.3 Programm-Äquivalenz

Jetzt kann man in Dynamischer Logik das Äquivalenzlemma zu diesen beiden „abstrakten Maschinen“ formulieren:

d-cp-equiv

$$\begin{array}{l} \quad \mathrm{compile}_2(\mathrm{pair}_1(\mathrm{db'}, \mathrm{procdef'})) = \mathrm{pair}_2(\mathrm{db''}, \mathrm{procdef''}) \\ \vdash \\ \quad \langle \text{d-eval\#}(\mathrm{db'}, \mathrm{procdef'}, \mathrm{goal}; \mathrm{subst}) \rangle \mathrm{subst} = \mathrm{subst}_0 \\ \leftrightarrow \langle \text{cp-eval\#}(\mathrm{db''}, \mathrm{procdef''}, \mathrm{goal}; \mathrm{subst}) \rangle \mathrm{subst} = \mathrm{subst}_0 \end{array}$$

Weil die Variable subst_0 in keinem der beiden Programme vorkommt, ist ihre Belegung konstant. Das Lemma besagt, daß beide Programme die gleiche Substitution zurückgeben, falls die Eingabe des einen Compilat der Eingabe des anderen ist.

Der Beweis dieses Lemmas, unter Voraussetzung der Gültigkeit der Compiler-Annahmen, wird im Mittelpunkt der weiteren Betrachtungen stehen.

Auch für die Stufen I, II und III wurden die dynamischen Algebren in Programme übertragen und entsprechende Äquivalenzlemmata formuliert.

Kapitel 4

Verifikation

In diesem Kapitel werden nun die wichtigsten Grundlinien der theoretischen und praktischen Beweisarbeit nachgezeichnet. Auch über letzteres werden wir eher problemorientiert als systemorientiert berichten. Trotzdem sollen vorher die wichtigsten Fakten zur Systemumgebung zusammengetragen werden, soweit sie die Druchführung der hier dokumentierten Verifikation betreffen.

4.1 Die KIV-Verifikationsumgebung

Die in Kapitel 3 beschriebene Formalisierung der Beweisaufgabe ist in ihrer syntaktischen Form zugeschnitten auf die Verifikation mit dem KIV[1]-System. Dieses ist ein taktischer Theorembeweiser auf der Grundlage eines Sequenzkalküls für Formeln Dynamischer Logik[2]. In sogenannten Modulen werden mittels DL-Formeln Aussagen über Programme formuliert und bewiesen. Die Module importieren strukturierte Spezifikationen (siehe Abschn. 2.5.1), womit die dort definierten Datentypen bekannt sind und als elementar benutzt werden können. Gleichzeitig stehen im Modul die in den Spezifikationen formulierten prädikatenlogischen Axiome und Lemmata zur expliziten oder impliziten (rewriting) Verwendung in den Beweisen zur Verfügung. Die Verwaltung der Lemmata und Beweise unterliegt einem automatischen Korrektheitsmanagement, welches beispielsweise dafür sorgt, daß nach der Änderung eines Programms die betroffenen Beweise als ungültig markiert werden.

Die Beweisschritte selbst bestehen aus der Auswahl und Anwendung einer Kalkülregel auf die oberste Sequenz eines offenen Beweisastes. Eine solche Regel ist beispielsweise if-left bzw. if-right die Aufspaltung eines Konditionals auf der linken bzw. rechten Seite der Sequenz. Dabei entstehen zwei neue Beweisäste. Eine andere Regel, die wir noch erwähnen werden, ist call-right bzw. call-left. Dabei wird ein Prozeduraufruf unter Beachtung der Parameterübergabe ersetzt durch den Rumpf der zugehörigen Deklaration.

Die Auswahl von Kalkülregeln geschieht sowohl interaktiv als auch automatisch mittels Heuristiken. Deren wichtigste ist „symbolic execution“, gerade auch für unsere Anwendung. Damit werden Parameterübergabe, Variablendeklaration, Konditionale, Zuweisungen und Simplifikationen von Termen und Formeln automatisch „ausgeführt“ (d.h. die passenden Regeln angewendet).

[1] Karlsruhe Interactive Verifier
[2] vgl. Abschn. 2.5.2

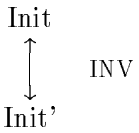
Durch die graphische Aufbereitung der Beweisbäume wird dem Benutzer die Analyse auch großer Beweise erleichtert oder überhaupt erst ermöglicht. Dies ist besonders wichtig auch im Hinblick auf scheiternde Beweise. Die entscheidende Ursache für die logische Lücke wird durch die Betrachtung der letzten zwei oder drei Verzweigungen oft sehr schnell deutlich.

Besonders wichtig für unsere Zwecke ist, daß im KIV-System eine Wiederverwendung von gültigen oder ungültigen Beweisen implementiert ist. Nur dadurch ist die noch zu erläuternde sukzessive Entwicklung einer Induktionsbehauptung überhaupt möglich. Wir wagen die These, daß der Beweis der Äquivalenz der betrachteten Stufen nicht gelingen kann ohne einen „Replay"-Mechanismus.

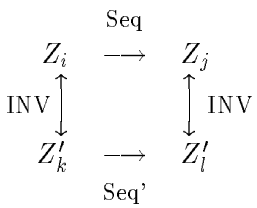
4.2 Äquivalenz regelbasierter Programme

Als Hinführung auf den Beweis des Lemmas d-cp-evuiv wollen wir kurz und informell das grundsätzliche Vorgehen erläutern, das zum Beweis der Äquivalenz regelbasierter Programme dienen kann. Dabei reden wir über solche Programme, wie sie in der Umsetzung von dynamischen Algebren nach dem in Abschnitt 3.3 beschriebenen Schema entstehen. Um so nah wie möglich an dem kommenden Beweis zu bleiben, orientieren wir uns begrifflich eher an Programmen als an dynamischen Algebren, indem wir beispielsweise von Zuständen reden statt von statischen Algebren. Allerdings nennen wir die den Regeln entsprechenden Unterprozeduren weiterhin „Regeln", um Verwechselungen mit PROLOG-Prozeduren zu vermeiden.

Es gilt, zu zeigen, daß zwischen den Endzuständen zweier Programme eine bestimmte Relation R besteht. In unserem Beispiel ist das die Gleichheit der Substitutionen. Die Programme bestehen im wesentlichen aus einer while-Schleife, innerhalb derer je nach Zustand eine Regel angesprungen wird. Der Beweis, daß am Ende beider Schleifen R gilt, muß induktiv über der Anzahl der Schleifendurchläufe geführt werden. Für die Induktionsbehauptung muß R verstärkt werden, sagen wir zu INV. Diese Formel sagt, salopp gesprochen, aus, in welcher Beziehung sich entsprechende Zustände stehen. Als Induktionsanfang zeigt man, daß die initialen Zustände Init und Init' INV erfüllen:



Als Induktionsschritt zeigt man, daß sich von zwei Zuständen, die INV erfüllen, jeweils durch eine Sequenz von Schleifendurchläufen, d.h. Regelanwendungen, wieder Zustände erreichen lassen, die INV erfüllen:



Wenn dies beides gezeigt ist, dann folgt auch für die finalen Zustände:

$$\begin{array}{c} \text{Fin} \\ \updownarrow \text{INV} \\ \text{Fin'} \end{array} \quad \text{insbesondere die Abschwächung} \quad \begin{array}{c} \text{Fin} \\ \updownarrow \text{R} \\ \text{Fin'} \end{array} \quad \text{q.e.d.}$$

Die größte Schwierigkeit besteht darin, die richtige Invariante INV zu finden. Sie kann sehr viel umfangreicher sein, als man das zunächst vermuten möchte. Die für unser Lemma benötigte Invariante werden wir noch ausführlich besprechen.

Ein beweistechnisch sehr günstiger Spezialfall ist es, wenn der Induktionsschritt die Länge eins hat, und zwar bezogen auf beide Programmschleifen. Das ist der Fall, wenn sich je zwei einzelne Schleifendurchläufe (Regelanwendungen) direkt entsprechen.

$$\begin{array}{ccc} & \overset{1}{} & \\ Z_i & \longrightarrow & Z_{i+1} \\ \text{INV}\updownarrow & & \updownarrow \text{INV} \\ Z'_k & \longrightarrow & Z'_{k+1} \\ & \underset{1}{} & \end{array}$$

Dies gilt in unserem Beispiel leider nur beinahe. Jedoch läßt sich durch eine Aufspaltung der Beweisaufgabe diese eins-zu-eins-Entsprechung im schwierigen Teil doch noch erreichen.

4.3 Aufspaltung der Beweisaufgabe

Auf den ersten Blick scheint zwischen den Programmen d-eval# und cp-eval# ebenso wie zwischen den entsprechenden dynamischen Algebren (S.17 und 22) die gewünschte Eins-zu-eins-Entsprechung zu bestehen, wenn man eine korrekte Übersetzung, d.h. die Compiler-Annahme, voraussetzt.

d-call# entspricht cp-call#, d-entry# entspricht cp-entry#, d-try# entspricht cp-trust# usw. . Nur d-retry# entspricht im „then"-Fall cp-trust# und im „else"-Fall cp-retry#, was aber nichts daran ändert, daß in beiden Fällen jeweils ein Schleifendurchlauf auf beiden Seiten wieder zu sich entsprechenden Zuständen führt.

Der kritische Punkt liegt woanders. Es gibt eine Situation, in der *zwei* Schleifendurchläufe des D-Programms *einem* Schleifendurchlauf des CP-Programms entsprechen. Dies ist der Fall, wenn mit call eine einelementige Prozedur betreten wird. d-call# setzt den Modus unabhängig davon auf try. Erst in d-try# wird getestet, ob es mehr als eine Klausel gibt (clause'(next(cllreg), db') $\neq$ null). Wenn nicht, dann ist die einzige Aktion das Setzen des Modus auf enter. Ein Choicepoint wird nicht angelegt. Im nächsten Schleifendurchlauf wird also die Regel d-enter# angesprungen. Diesen wirkungslosen Besuch der try-Regel vermeidet das CP-Programm. Die laut Compiler-Annahme einzige korrekte Übersetzung einer einelementigen Prozedur ist eben die Klausel selbst, ohne Instruktionen (wie try-me....). Durch Nachschlagen in procdef" erhält preg in cp-call# die Adresse dieser Klausel als Wert. Wegen is_clause(code(preg, db")) wird also als nächstes cp-enter# angesprungen, diesmal ohne Umweg über cp-try#.

Auch wenn dieser Unterschied nur minimal ist, wird der Induktionsbeweis dadurch beeinträchtigt. Man kommt im Äquivalenzbeweis bei solchen Schrittunterschieden nicht mehr mit

struktureller Induktion über die Anzahl der Schleifendurchläufe aus (n → n+1). Stattdessen benötigt man noethersche Induktion.

Diesen Punkt wollen wir gerne isoliert beweisen, da die eigentliche Problematik, der Vergleich der abstrakten Maschine, durchaus noch schwer genug ist. Es ist natürlich ein leichtes, das D-Programm so zu modifizieren, daß der Test auf einelementige Klauseln von d-try# nach d-call# verlagert wird und so die enter-Regel direkt angesprungen werden kann. Das Ergebnis würde lauten:

```
NEWD-CALL#(act, procdef', breg, db'; var cllreg,rmode,ctreg,stop)
begin
if clause'(procdef' ^pd id(act), db') = null
then D-BACKTRACK#(breg; stop, rmode)
else begin cllreg := procdef' ^pd id(act);
           ctreg := breg;
           if clause'(next(cllreg), db') = null
           then rmode := enter
           else rmode := try
     end
end;

NEWD-TRY#(cllreg,subreg,decglseqreg,db';
          var scll,ssub,breg,b,sdecglseq,s,rmode)
begin
rmode := enter;
var temp = snew(s)
in begin
   s := s +s temp;
   b := b +b temp /b breg;
   sdecglseq := sdecglseq +sd temp /sd decglseqreg;
   ssub := ssub +su temp /su subreg;
   scll := scll +sc temp /sc next(cllreg);
   breg := temp
   end
end;
```

Damit erreichen wir eine mit dem CP-Programm übereinstimmende Anzahl von Schleifendurchläufen. Auf der anderen Seite muß jetzt die Äquivalenz zwischen newd-... und d-... gezeigt werden, was aber trotz noetherischer Induktion einfacher ist als vorher, da sich d-... und newd-... kaum unterscheiden.

Da wir schon einmal ein neues Programm zwischen die beiden alten legen, trennen wir auch gleich das d-retry# auf in newd-retry# und newd-trust#, so daß es in newd-... und cp-... gleichviele sich paarweise entsprechende Regeln gibt.

```
NEWD-RETRY#(sdecglseq,b,ssub,breg;
            var cllreg,subreg,scll,ctreg,decglseqreg,rmode)
(: entspricht D-RETRY-2# (it else) :)
begin
```

```
decglseqreg := sdecglseq ^sd breg;
subreg := ssub ^su breg;
cllreg := scll ^sc breg;
scll := scll +sc breg /sc next(scll ^sc breg);
ctreg := b ^b breg;
rmode := enter
end;

NEWD-TRUST#(sdecglseq,b,ssub,scll;
            var cllreg,subreg,breg,ctreg,decglseqreg,rmode)
(: entspricht D-RETRY-1# (then) :)
begin
decglseqreg := sdecglseq ^sd breg;
subreg := ssub ^su breg;
cllreg := scll ^sc breg;
ctreg := b ^b breg;
rmode := enter;
breg := b ^b breg
end;
```

Diese Anpassung dient lediglich der Übersicht im schwierigen Teil der Verifikation.

newd-eval# unterscheidet sich nur in den neuen Regelaufrufen von d-eval#:

```
NEWD-BODY#(db',procdef'; var s,vi,rmode,stop,breg,ctreg,
                         ssub,subreg,sdecglseq,decglseqreg,scll,cllreg,b)
begin
if decglseqreg = sdnil
then D-QUERY-SUCCESS#(; stop)
else var goal = (sdcar(decglseqreg)).s1
     in if goal = gnil then D-GOAL-SUCCESS#(; decglseqreg)
        else var act = gcar(goal),
                 scutpt = (sdcar(decglseqreg)).s2
             in var scont =      mksdecgoal(gcdr(goal),scutpt)
                           +sdl sdcdr(decglseqreg)
                in if is_user_defined(act)
                   then if rmode = call'
                        then NEW-D-CALL#(act,procdef',breg,db';
                                         cllreg,rmode,ctreg,stop)
                        else if rmode = try
                        then NEW-D-TRY#(cllreg,subreg,decglseqreg,db';
                                        scll,ssub,breg,b,sdecglseq,s,rmode)
                        else if rmode = enter
                        then D-ENTER#(db',act,scont,cllreg,breg,ctreg;
                                      subreg,decglseqreg,vi,stop,rmode)
                        else (: rmode = retry :)
                             if clause'(next(scll ^sc breg), db') # null
                             then NEW-D-RETRY#(sdecglseq,b,ssub,breg;
```

```
                                        cllreg,subreg,scll,ctreg,
                                        decglseqreg,rmode)
                          else NEW-D-TRUST#(sdecglseq,b,ssub,scll;
                                        cllreg,subreg,breg,ctreg,
                                        decglseqreg,rmode)
               else if act = true'
                    then D-TRUE#(scont; decglseqreg)
                    else if act = fail'
                    then D-FAIL#(breg; stop, rmode)
                    else (: act = ! :)
                         D-CUT#(scont,scutpt; breg, decglseqreg)
end;
```

Mit Hilfe dieses Hilfsprogramms reduziert sich der Beweis des Lemmas d-cp-equiv auf:

d-newd-equiv

$\vdash$
$\quad \langle$d-eval#(db', procdef', goal; subst)$\rangle$subst = subst$_0$
$\quad \leftrightarrow \langle$newd-eval#(db', procdef', goal; subst)$\rangle$subst = subst$_0$

und:

newd-cp-equiv

$\quad$compile$_2$(pair$_1$(db', procdef')) = pair$_2$(db", procdef")
$\vdash$
$\quad \langle$newd-eval#(db', procdef', goal; subst)$\rangle$subst = subst$_0$
$\quad \leftrightarrow \langle$cp-eval#(db", procdef", goal; subst)$\rangle$subst = subst$_0$

Der Beweis von d-newd-equiv ist nur technisch aufwendig, aber inhaltlich uninteressant. Im folgenden wird uns nur noch der Beweis new-cp-equiv beschäftigen. Dieses Äquivalenzlemma wird aufgespalten in zwei Implikationen:

newd-cp-imp

$\quad$compile$_2$(pair$_1$(db', procdef')) = pair$_2$(db", procdef")
$\vdash$
$\quad \langle$newd-eval#(db', procdef', goal; subst)$\rangle$subst = subst$_0$
$\quad \rightarrow \langle$cp-eval#(db", procdef", goal; subst)$\rangle$subst = subst$_0$

und:

cp-newd-imp

$\quad$compile$_2$(pair$_1$(db', procdef')) = pair$_2$(db", procdef")
$\vdash$
$\quad \langle$cp-eval#(db", procdef", goal; subst)$\rangle$subst = subst$_0$
$\quad \rightarrow \langle$newd-eval#(db', procdef', goal; subst)$\rangle$subst = subst$_0$

newd-cp-imp wird bewiesen mit einem Induktionslemma newd-cp-ind. Darin werden die while-Schleifen aufgebrochen in loop-Konstrukte. Gleichzeitig führen wir die invariante Relation zwischen Zuständen beider Programme ein. Diese Invariante soll aber erst in den kommenden Abschnitten erläutert werden. Darum wollen wir sie hier nur schematisch darstellen mit INV($\overline{x}, \overline{y}$). Dabei seien $\overline{x}$ und $\overline{y}$ die Zustandsvektoren der body#-Programme.

newd-cp-ind

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$,
$\text{INV}(\overline{x}, \overline{y})$,
$\langle$**loop**
  if stop = run **then**
    newd-body#(db', procdef'; $\overline{x}$)
 times kappa$\rangle$ $\overline{x} = \overline{x}_0$
$\vdash$
$\exists$ kappa.$\langle$**loop**
  if stop' = run **then**
    cp-body#(db", procdef"; $\overline{y}$)
 times kappa$\rangle$
  $\text{INV}(\overline{x}_0, \overline{y})$

Dies läßt sich beweisen mit dem Induktionsschritt newd-cp-step, in dem jeweils nur ein loop-Schritt (entspricht einem Schleifendurchlauf) betrachtet werden muß. Das gelingt nur, weil wir zwischen dem newd-Programm und den cp-Programm eine genaue Übereinstimmung bezüglich der Schritte geschaffen haben!

newd-cp-step

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$,
$\text{INV}(\overline{x}, \overline{y})$,
$\langle$**if** stop = run **then**
  newd-body#(db', procdef'; $\overline{x}$)$\rangle$ $\overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
  cp-body#(db", procdef"; $\overline{y}$)$\rangle$
  $\text{INV}(\overline{x}_0, \overline{y})$

Da sich je zwei Regeln wechselseitig entsprechen, werden wir new-cp-step beweisen mit zehn weiteren Lemmata (für die zehn Regeln): call-newd-cp, try-newd-cp usw. . Sie unterscheiden sich von newd-cp-step nur darin, daß die jeweiligen newd-Einstiegsbedingungen noch zusätzlich vorausgesetzt werden. Daß damit auch jeweils die zugehörige cp-Einstiegsbedingung gilt, muß aus der Invarianten folgen, was zu beweisen ist.
Beispiele:

call-newd-cp

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$,
$\text{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
is_user_defined(gcar(sdcar(decglseqreg).s1))
$\langle$**if** stop = run **then**
  newd-body#(db', procdef'; $\overline{x}$)$\rangle$ $\overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**

cp-body#(db", procdef"; $\overline{y}$)$\rangle$
INV($\overline{x}_0, \overline{y}$)

try-newd-cp

compile$_2$(pair$_1$(db', procdef')) = pair$_2$(db", procdef"),
INV($\overline{x}, \overline{y}$),
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = try
$\langle$**if** stop = run **then**
newd-body#(db', procdef'; $\overline{x}$)$\rangle$ $\overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)$\rangle$
INV($\overline{x}_0, \overline{y}$)

p-cp-qua
d-newd-equiv
dup-newd-p
newd-d-ind
p-newd-dup
d-newd-ind
newd-cp-dup
newd-cp-ind
newd-cp-equiv
cp-newd-dup
cp-newd-ind
cp-newd-step
newd-cp-step
q-succ-newd-cp
g-succ-newd-cp
call-newd-cp
true-newd-cp
fail-newd-cp
cut-newd-cp
try-newd-cp
enter-newd-cp
retry-newd-cp
trust-newd-cp
q-succ-cp-newd
g-succ-cp-newd
call-cp-newd
true-cp-newd
fail-cp-newd
cut-cp-newd
try-cp-newd
enter-cp-newd
retry-cp-newd
trust-cp-newd
compass2+weak1
weak-comp-assum-1
comp-assum-2

Die Compilerannahmen werden nur zum Beweisen call-newd-cp gebraucht. In den anderen Lemmata kommen sie indirekt durch die Invariante INV zum Tragen. Der Beweis dieser zehn Lemmata ist das Herzstück des Nachweises des Hauptlemmas d-cp-equiv.

Die andere Implikation, cp-newd-imp, wird entsprechend bewiesen. Allerdings lassen sich die Lemmata für die einzelnen Regeln, z.B. call-cp-newd, sehr leicht auf die Gegenrichtung zurückführen.

Auf S.60 ist der Lemma-Graph abgebildet. Es handelt sich dabei um keinen Beweis im engeren Sinne, sondern um einen Abhängigkeitsgraphen. So wird beispielsweise im Beweis für call-newd-cp an irgendeiner Stelle das Lemma compass2+weak1 benutzt.

Diese Übersicht sollte vor allem helfen, zu verstehen, „wo“ sich die Suche nach der Invarianten INV abgespielt hat. Beweishistorisch gesehen wird INV nämlich nicht in newd-cp-ind oder cp-newd-ind en bloc eingeführt durch die bloße Eingebung des Beweisingenieurs. Statt dessen muß sie auf der Ebene der zehn Lemmata call-newd-cp, try-newd-cp,... usw. erst entwickelt werden durch Beweisversuche, Korrekturen und erneute Beweisversuche. Dieser Prozeß stellt die eigentliche Beweisarbeit dar.

Für eine erfolgreiche Invariantenentwicklung sind vor allem zwei Eigenschaften des Beweissystems von entscheidender Bedeutung:

1. Eine benutzerfreundliche, übersichtliche Darstellung der Beweise, denn nur aus der Analyse der Beweise erwachsen Verbesserungen der Invarianten.

2. Die Wiederverwertbarkeit vorhandener Teilbeweise.

Ist die richtige Invariante erst einmal gefunden, dann ist der Beweis der in dem Graphen dargestellten Lemmata so gut wie bewältigt.

4.4 Herleitung einer vollständigen Invarianten

Wir werden den Leser noch davon zu überzeugen wissen, daß es nicht möglich ist, die richtige Invariante auf Anhieb vollständig hinzuschreiben. Dies bestätigen auch die Erfahrungen aus der Verifikation der ersten beiden Stufen (siehe [SchellhornAhrendt]). Nichtsdestoweniger ist es von entscheidender Bedeutung, daß der erste Ansatz bereits ein solides Fundament für die weitere Entwicklung darstellt. Darum ist es angebracht, beim Beweisen ebenso wie in dieser Dokumentation, der Formulierung einer ersten Hypothese den nötigen Raum zu gewähren.

4.4.1 Ausgangspunkt der Invarianten

Gesucht sind diejenigen invarianten Eigenschaften sich entsprechenener Zustände der Programme newd-eval# und cp-eval#, die induktiv die Gleichheit der Substitution in den Endzuständen garantieren.
Dazu gehören:

- invariante Eigenschaften der Zustände des einen Programms,

- invariante Eigenschaften der Zustände des anderen Programms, und

- die invariante Beziehung zwischen den Zuständen beider Programme.

Letzeres steht im Mittelpunkt der Betrachtungen. Es darf aber nicht übersehen werden, daß auch sehr selbstverständlich erscheinende Eigenschaften der dynamisch aufgebauten Strukturen (Stacks, Bäume) nirgendwo anders her abgeleitet werden können als induktiv über die Regeln. Dies gilt allgemein für dynamische Algebren. Hierin sehen wir eine Präzisierung der Sichtweise auch von [BörgerRosenzweig], z.B.: „We require of course the *cands*-list to be consistent with *father*, i.e. whenever *Son* is among *cands(Father)*, the *father(Son)* = *Father*." Dies klingt ein wenig nach einer Annahme im axiomatischen Sinne. Hingegen ist die beschriebene Eigenschaft der dynamischen Algebra nicht orthogonal zu dem Regelsystem, sondern eine unmittelbare Folgerung daraus, was man glauben oder auch beweisen kann; letzteres geht nur induktiv über die Regeln.

Hierauf wollten wir hinweisen, obwohl (oder auch weil) solche invarianten Eigenschaften in anderen Äquivalenzbeweisen eine größere Rolle spielen als in dem hier dokumentierten. Beispielsweise war es bei der Verifikation der ersten beiden Stufen notwendig, die aufwendig zu formalisierende Eigenschaft jeder möglichen Struktur, eben ein Baum zu sein und nicht nur ein azyklischer oder gar zyklischer Graph, explizit nachzuweisen.

Zurück zu der Beziehung zwischen den Programmen: Um diese zu beschreiben, ist es notwendig, eine Abbildung zwischen den beiden dynamischen Strukturen zu definieren, konkret zwischen den Stackelementen vom Typ state. Auch diese Definition ist nur induktiv möglich über die Schleifendurchläufe (Regelanwendungen), da dynamische Daten aufeinander abgebildet werden. Darum handelt es sich bei dieser Funktion f um eine dynamische, von uns simuliert mit der Datenstruktur finitefun:[3]

f:st->st =
actualize finitefun **with** state **by morphism**
elem' → statesort, ele' → st, elem" → statesort, ele" → st_0, pairdomcod → pairstatestate, finitefun → funstatestate, constfun → cf, ^ → $^\wedge_f$, / → $/_f$, $+_{fun} \rightarrow +_f$, func → f
end actualize

Statt die Eigenschaften einer festen Funktion f zu beschreiben, postuliert die gesuchte Invariante die *Existenz* einer Funktion f, die bestimmte Eigenschaften erfüllt. Im Beweis wird dann der Existenzquantor *nach* einer Regel instanziiert entweder mit einer dynamischen Erweiterung des *vor* der Regel gültigen f oder diesem f selbst, je nachdem, ob neue Knoten hinzugekommen sind oder nicht.

Wir wollen nun in die erste Version der Invarianten einführen, so wie sie im Beweisverlauf tatsächlich verwendet wurde, indem wir die Teilformeln motivieren. Dazu vergegenwärtigen wir uns nochmals die Arbeitsweise der beiden Programme (Abschn. 3.3.1 und 3.3.2) bzw. der entsprechenden dynamischen Algebren (S.17 und 22) und Compiler-Annahmen in Form des Lemmas compass2+weak1 (S.44) mitsamt der Deklaration der darin auftretenden Hilfsprozeduren clls# (S.40) und chain# (S.42).

Der Hauptunterschied liegt in den Modi und den Adresszeigern. Dadurch, daß in der cp-Stufe Steuerinstruktionen im Code stehen anstelle der Modi aus der newd-Stufe, übernimmt preg die Rolle sowohl von rmode als auch von cllreg. Im Falle von rmode = call' zeigt preg auf start, im Falle von rmode = enter auf eine Klauselzeile, und zwar eine, deren Eintrag cllreg entspricht:

[3] Notationell entspricht unser f dem F aus [Schmitt] und [BörgerRosenzweig], nicht dem $\mathcal{F}$. Wir betrachten hier keine Abbildung zwischen Algebren, da wir in DL *beide* statischen Algebren durch *einen* Zustand über einer disjunkt aufteilbaren Menge von Variablen darstellen.

(rmode = call' → preg = start)
∧ (rmode = enter → code(preg, db") = mkcl(the_clau(clause'(cllreg, db'))))

Wenn mode = retry ist, dann zeigt preg auf die zum nächsten Choicepoint breg' gehörende Klausel p $\hat{}_p$ breg' (vgl. d-backtrack# und cp-backtrack#):

(rmode = retry → preg = p $\hat{}_p$ breg')

Damit ist schon gewährleistet, daß preg auf eine retry_me- oder auf eine trust_me-Instruktion zeigt, weil die Funktionswerte von p dies im gesamten Stack tun, wie wir noch sehen werden.

Schließlich muß noch, wenn rmode = try ist, preg auf eine try_me-Instuktion zeigen. Aber das genügt noch lange nicht. rmode = try gilt genau nach der Durchführung von newd-call# bzw. cp-call#. Innerhalb dieser Regeln wurden die Prozedur-Einstiegsadressen in procdef' bzw. procdef" nachgeschlagen und in cllreg bzw. preg festgehalten. Dank der Compiler-Annahme wissen wir einiges über diese Adressen und die dort beginnenden Prozeduren, nämlich daß sie sich inhaltlich entsprechen (vgl. compass2+weak1). Diese Informationen werden wir natürlich noch für spätere Regeln brauchen. Wenn z.B. nach dem dritten erfolglosen Abstieg die jeweils vierte Klausel der Prozedur versucht werden muß, dann benötigen wir unbedingt die Information, daß es sich hierbei jeweils um die gleiche Klausel handelt. Wir müssen also die compiler-Annahme über die Zeit retten, und zwar innerhalb der Invarianten. Das geschieht, indem festgehalten wird, daß im Modus try folgendes gilt:

rmode = try
→ ⟨clls#(cllreg, db'; cal_1)⟩
⟨chain-try-me#(preg, db"; cal_2)⟩
mapcode(cal_2, db") = mapclause'(cal_1, db')

Das heißt, grob gesagt, daß die Compiler-Annahme jetzt für cllreg und preg statt für die Einträge der Prozedurdefinitions-Tabellen gilt. Wir wollen noch kurz überlegen, was aus obiger Formel alles folgerbar ist für den augenblicklichen Zustand (vgl. dazu Deklaration von chain#):

1. preg zeigt auf nichts anderes, als eine try_me-Instruktion, denn sonst würde chain-try-me# nicht, wie behauptet, terminieren!

2. Die auf preg folgende Zeile zeigt auf die gleiche Klausel wie cllreg, weil chain-try-me# next(preg) vorne in cal_2 (codearealist) einhängt und mapcode(cal_2,db") = mapclause'(cal_1,db').

Da in cp-try-me# preg inkrementiert wird, folgt aus 2., daß in den auf try folgenden enter-Regeln die gleichen Klauseln verwendet werden!

Wir betrachten nun das weitere Schicksal dieser Information aus der Compiler-Annahme. Vor den try-Regeln hatten wir sie in obiger Formel zur Verfügung. Innerhalb von newd-try# bzw. cp-try# werden beim Anlegen der choicepoints die Wiederaufsetzpunkte next(cllreg) bzw. where(code(preg,db")) gekellert, indem sie Wert von scll $\hat{}_{sc}$ breg bzw. p $\hat{}_p$ breg' werden. Danach gilt:

⟨clls#(scll $\hat{}_{sc}$ breg, db'; cal_1)⟩
⟨chain-retry-me#(p $\hat{}_p$ breg', db"; cal_2)⟩
mapcode(cal_2, db") = mapclause'(cal_1, db'))

Davon kann man sich überzeugen, indem man in der „rmode = try“-Formel nachvollzieht, wie der „cdr“ der Listen cal_2 und cal_1 zustande kommt, nämlich durch rekursiven Aufruf von clls# und chain-retry-me# mit den Wiederaufsetzpunkten als Argumenten!

Nun kann man auch folgern, daß p $\hat{}_p$ breg' auf eine retry_me-Instruktion zeigt, da chain-retry-me# sonst nicht, wie behauptet, terminiert. Im ersteren Fall werden bei einem erneuten Abstieg von diesen Knoten aus die Wiederaufsetzpunkte durch die retry-Regeln überschrieben in dem gleichen Sinne, wie das vorher durch die try-Regeln geschah. Dann läßt sich die obige Formel aus sich selbst nachweisen, indem man wieder die „cdr“- Listen betrachtet.

Weil jeder Knoten einmal als breg entstanden ist, kann man die Formel für den ganzen Stack verallgemeinern. Alle cp-Knoten zeigen mit scll auf eine retry-me- bzw. trust-me-Instruktion, und bei sich entsprechenden Knoten st und f $\hat{}_f$ st sind die noch zu bearbeitenden Klauseln gleich:

$$\begin{aligned}
&\forall\ \text{st.}\quad \text{st ins s}\\
&\quad\rightarrow \langle \text{clls\#(scll } \hat{}_{sc} \text{ st, db'; } cal_1)\rangle\\
&\qquad\qquad \langle \text{chain-retry-me\#(p } \hat{}_p \text{ f } \hat{}_f \text{ st, db"; } cal_2)\rangle\\
&\qquad\qquad\qquad \text{mapcode}(cal_2, \text{db"}) = \text{mapclause'}(cal_1, \text{db'}))
\end{aligned}$$

Glücklicherweise gilt dies, eher zufällig, auch auf verlassenen Knoten. Das erspart uns eine Charakterisierung des Stack.

Somit ist die invariante Beziehung zwischen cllreg, rmode und scll auf der einen Seite und preg, p auf der anderen Seite geklärt, wenigstens, was die erste Version betrifft. Für die verbleibenden globalen Register gilt:

- stop, vi und subreg sind jeweils identisch.
- breg und ctreg entsprechen sich jeweils über f.
- In decglseqreg sind die Terme identisch; die cutpoints entsprechen sich über f: fd(f, decglseqreg) = decglseqreg'.[4]

Ähnliche Gleichungen gelten für die dynamischen Funktionen auf den sich über f entsprechenden Knoten der beiden Stacks:

- ssub $\hat{}_{su}$ st = ssub' $\hat{}_{su}$ f $\hat{}_f$ st
- f $\hat{}_f$ b $\hat{}_b$ st = b' $\hat{}_b$ f $\hat{}_f$ st
- fd(f, sdecglseq $\hat{}_{sd}$ st) = sdecglseq' $\hat{}_{sd}$ f $\hat{}_f$ st

Außerdem wird die Injektivität von f benötigt. Da die Knotenmengen s und s' dynamisch wachsen, müssen die notwendigen Aussagen zu diesen Mengen auch in die Invariante: Daß breg, ctreg und b $\hat{}_b$ st immer in s liegen und das Bild von s unter f in s' liegt[5].

Somit haben wir die Formel erklärt, die uns als Ausgangspunkt für die Entwicklung einer vollständigen Invarianten günstig erschien. Sie lautet insgesamt:

[4] fd ist die Fortsetzung von f auf 'decorated goal sequences'.

[5] s und s' sind die Mengen von states, die die dynamischen Universen der beiden abstrakten Maschinen modellieren.

INV.1

$$
\begin{aligned}
&\exists\ f.\quad stop = stop' \\
&\quad \wedge\ vi = vi' \\
&\quad \wedge\ subreg = subreg' \\
&\quad \wedge\ f\ \hat{}_f\ breg = breg' \\
&\quad \wedge\ f\ \hat{}_f\ ctreg = ctreg' \\
&\quad \wedge\ fd(f, decglseqreg) = decglseqreg' \\
&\quad \wedge\ breg\ ins\ s \\
&\quad \wedge\ ctreg\ ins\ s \\
&\quad \wedge\ (rmode = call' \rightarrow preg = start) \\
&\quad \wedge\ (rmode = retry \rightarrow preg = p\ \hat{}_p\ breg') \\
&\quad \wedge\ (rmode = enter \rightarrow code(preg, db'') = mkcl(the\_clau(clause'(cllreg, db')))) \\
&\quad \wedge\ (\quad rmode = try \\
&\qquad \rightarrow \langle clls\#(cllreg, db'; cal_1)\rangle \\
&\qquad\qquad \langle chain\text{-}try\text{-}me\#(preg, db''; cal_2)\rangle \\
&\qquad\qquad\quad mapcode(cal_2, db'') = mapclause'(cal_1, db')) \\
&\quad \wedge\ (\forall\ st.\quad st\ ins\ s \\
&\qquad\quad \rightarrow\quad (b\ \hat{}_b\ st)\ ins\ s \\
&\qquad\qquad \wedge\ (f\ \hat{}_f\ st)\ ins\ s' \\
&\qquad\qquad \wedge\ ssub\ \hat{}_{su}\ st = ssub'\ \hat{}_{su}\ f\ \hat{}_f\ st \\
&\qquad\qquad \wedge\ f\ \hat{}_f\ b\ \hat{}_b\ st = b'\ \hat{}_b\ f\ \hat{}_f\ st \\
&\qquad\qquad \wedge\ fd(f, sdecglseq\ \hat{}_{sd}\ st) = sdecglseq'\ \hat{}_{sd}\ f\ \hat{}_f\ st \\
&\qquad\qquad \wedge\ (\forall\ st_1.st_1\ ins\ s \rightarrow (f\ \hat{}_f\ st = f\ \hat{}_f\ st_1 \rightarrow st = st_1)) \\
&\qquad\qquad \wedge\ \langle clls\#(scll\ \hat{}_{sc}\ st, db'; cal_1)\rangle \\
&\qquad\qquad\qquad \langle chain\text{-}retry\text{-}me\#(p\ \hat{}_p\ f\ \hat{}_f\ st, db''; cal_2)\rangle \\
&\qquad\qquad\qquad\quad mapcode(cal_2, db'') = mapclause'(cal_1, db'))
\end{aligned}
$$

Den Existenzquantor ganz nach vorne zu schieben fördert die Übersicht in den Beweisen. Um hier keinen falschen Eindruck aufkommen zu lassen: Die Teilformeln sind kein Sammelsurium *irgendwelcher* invarianten Eigenschaften. Sie erfüllen einzig und allein den Zweck, die Invarianz von subreg = subreg' zu garntieren! Alle Aussagen, die zum Nachweis dessen benötigt werden, müssen wiederum als invariant nachgewiesen werden durch weitere Aussagen usw. . Dieser Prozeß terminiert zwar irgendwann, aber sicher noch nicht beim ersten Versuch, auch nicht bei dem unsrigen.

4.4.2 Sukzessive Vervollständigung mit KIV

Die Ausgangsformel INV.1 ist noch nicht invariant im Sinne des angestrebten Induktionsbeweises. Die Methode, sie zu vervollständigen, besteht gerade aus dem Versuch, ihre Invarianz streng formal mit Hilfe des Systems zu beweisen. Aus dem scheiternden Beweis lassen sich dann Rückschlüsse ziehen auf fehlende Invarianzen. Diese werden, meist in Form neuer Konjunktionsglieder, der unvollständigen Formel hinzugefügt. Ein neuer Beweisversuch offenbart dann neue Lücken usw...

Dieses führen wir durch an den zehn Lemmata *rule*-newd-cp mit *rule* $\in$ {call, try, enter, retry, trust, true, fail, cut, g-succ, q-succ}[6]. Unter der Vorbedingung INV.i zuzüglich der jeweiligen Einstiegsbedingungen soll jetzt gezeigt werden, daß nach „paralleler“ Ausführung

[6] siehe Anhang E

von newd-*rule*# und cp-*rule*# wieder INV.i als Nachbedingung gilt. Wenn dieser Nachweis auch nur für eines der zehn Lemmata scheitert, werden sie alle neu formuliert mit einer erweiterten Formelversion INV.i'. Unter Wiederverwendung der fertigen oder unfertigen, in jedem Fall aber ungültigen Beweise der vorhergehenden Version versucht man, diese neuen Lemmata zu beweisen. Wie sich anhand dieser Methode die Invariante, ausgehend von INV.1, tatsächlich entwickelt hat, soll nun nachgezeichnet werden.

Der Beweis von call-newd-cp mit INV.1 bleibt schon stecken in dem Zweig, der die symbolische Ausführung von d-backtrack# und cp-backtrack# betrachtet. Eine Analyse der Beweissituation ergibt, daß die aktuell vorletzte Aufspaltung des Beweisbaumes ausging von der Sequenz[7]:

$\langle$**if** breg_0 = bottom **then** stop_2 := failure **else** rmode_1 := retry$\rangle$ $\mathrm{stop}_2 = \mathrm{stop}_0 \wedge \varphi$,
f_0 ˆ$_f$ bottom = bottom, $\mathrm{stop'}_1$ = run,stop_2 = run, Γ
$\vdash$
$\langle$**if** f_0 ˆ$_f$ breg_0 = bottom **then** $\mathrm{stop'}_1$:= failure **else** preg_0 := p ˆ$_p$ f_0 ˆ$_f$ $\mathrm{breg}_0\rangle$ stop_0 = $\mathrm{stop'}_1 \wedge \psi$

Hier geht es vor allem darum, zu zeigen, daß entweder *beide* Programme endgültig „failen“ oder *beide* noch Alternativen durchspielen. Nach Anwendung der Regel if-left ergeben sich zwei Zweige für **then** und **else**. Auf dem else-Zweig liegt die Sequenz:

$\mathrm{breg}_0 \neq$ bottom, rmode_1 = retry, $\mathrm{stop'}_1$ = run, Γ
$\vdash$
$\langle$**if** f_0 ˆ$_f$ breg_0 = bottom **then** $\mathrm{stop'}_1$:= failure **else** preg_0 := p ˆ$_p$ f_0 ˆ$_f$ $\mathrm{breg}_0\rangle$ $\mathrm{stop'}_1$ = run $\wedge \psi$

Die intuitive Erwartung ist, daß in *diesem* Zweig auch auf der rechten Seite der else-Fall „betreten“ wird, indem die if-Bedingung negativ entschieden werden kann. Genauer gesagt: Der then-Ast mit der Sequenz

$\mathrm{breg}_0 \neq$ bottom, f_0 ˆ$_f$ breg_0 = bottom, Γ
$\vdash$
Δ

müßte schließbar sein, schon weil sich die cp-then-Bedingung f_0 ˆ$_f$ breg_0 = bottom und die d-else-Bedingung $\mathrm{breg}_0 \neq$ bottom gegenseitig ausschließen.

Auch wenn es einleuchtend ist, daß zwei sich über f entsprechende Knoten der beiden abstrakten Maschinen nur entweder beide gleich bottom sein können oder keiner von beiden, so läßt sich doch obige Sequenz nicht schließen. Zwar hatten wir schon in Voraussicht auf diese Situation die Injektivität von f und die Invariante INV.1 aufgenommen (s.o.), aber diese hilft uns nur weiter, wenn außerdem noch die Formel f ˆ bottom = bottom zur Verfügung stünde. Dann ließe sich mit breg $\neq$ bottom auf f ˆ breg $\neq$ bottom schließen.

Darum erweitern wir jetzt INV.1 um f ˆ$_f$ bottom = bottom und außerdem bottom ins s, weil die Injektivivtät von f auf die Menge aller states s eingeschränkt ist. Das Ergebnis lautet:

[7] Die Sequenzen aus den Systembeweisen werden hier stark abgekürzt wiedergegeben. φ und ψ stehen für DL-Formeln, Γ und Δ für Formellisten. Diese Metavariablen gelten nur sequenz-lokal. Indizes, die nicht aus einem Lemma stammen, wurden automatisch erzeugt.

INV.2

$\exists$ f. stop = stop'
$\wedge$ vi = vi'
$\wedge$ subreg = subreg'
$\wedge$ f $\hat{}_f$ breg = breg'
$\wedge$ f $\hat{}_f$ ctreg = ctreg'
$\wedge$ **f $\hat{}_f$ bottom = bottom**
$\wedge$ fd(f, decglseqreg) = decglseqreg'
$\wedge$ breg ins s
$\wedge$ ctreg ins s
$\wedge$ **bottom ins s**
$\wedge$ (rmode = call' $\rightarrow$ preg = start)
$\wedge$ (rmode = retry $\rightarrow$ preg = p $\hat{}_p$ breg')
$\wedge$ (rmode = enter $\rightarrow$ code(preg, db") = mkcl(the_clau(clause'(cllreg, db'))))
$\wedge$ (rmode = try
$\rightarrow$ $\langle$clls#(cllreg, db'; cal_1)$\rangle$
$\langle$chain-try-me#(preg, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))
$\wedge$ ($\forall$ st. st ins s
$\rightarrow$ (b $\hat{}_b$ st) ins s
$\wedge$ (sdecglseq $\hat{}_{sd}$ st) ctpelem s
$\wedge$ (f $\hat{}_f$ st) ins s'
$\wedge$ ssub $\hat{}_{su}$ st = ssub' $\hat{}_{su}$ f $\hat{}_f$ st
$\wedge$ f $\hat{}_f$ b $\hat{}_b$ st = b' $\hat{}_b$ f $\hat{}_f$ st
$\wedge$ fd(f, sdecglseq $\hat{}_{sd}$ st) = sdecglseq' $\hat{}_{sd}$ f $\hat{}_f$ st
$\wedge$ ($\forall$ st_1.st_1 ins s $\rightarrow$ (f $\hat{}_f$ st = f $\hat{}_f$ st_1 $\rightarrow$ st = st_1))
$\wedge$ $\langle$clls#(scll $\hat{}_{sc}$ st, db'; cal_1)$\rangle$
$\langle$chain-retry-me#(p $\hat{}_p$ f $\hat{}_f$ st, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))

Man muß sich gerade wegen der Trivialität der neuen Teilformeln noch einmal klar machen, daß es nicht genügt, wenn der Beweisingenieur eine gerade benötigte invariante Eigenschaft als selbstverständlich ansieht. Herleiten läßt sie sich nur als Teil von INV! Weiterhin werden wir noch sehen, daß man sich in den Vermutungen über „selbstverständliche“ Invarianzen auch sehr täuschen kann.

Der Versuch, die Lemmata mit INV.2 zu beweisen, scheitert daran, daß sich die Eigenschaft breg ins s bisher nicht durch d-cut# vererben läßt, denn breg wird aktualisiert mit dem ersten cutpoint aus decglseqreg. Daß dieser aber Element von s sein muß, folgt nicht aus INV.2. Um als invariant nachweisbar zu sein, verstärken wir diese Aussage daraufhin, daß alle cutpoints der decglseqreg in s liegen müssen. Dafür wird ein neues Prädikat ctpelem zur Spezifikation hinzugefügt und INV.2 erweitert zu

INV.3

$\exists$ f. stop = stop'
$\wedge$ vi = vi'
$\wedge$ subreg = subreg'
$\wedge$ f $\hat{}_f$ breg = breg'
$\wedge$ f $\hat{}_f$ ctreg = ctreg'
$\wedge$ f $\hat{}_f$ bottom = bottom
$\wedge$ fd(f, decglseqreg) = decglseqreg'
$\wedge$ breg ins s
$\wedge$ ctreg ins s
$\wedge$ bottom ins s
$\wedge$ **decglseqreg ctpelem s**
$\wedge$ (rmode = call' $\rightarrow$ preg = start)
$\wedge$ (rmode = retry $\rightarrow$ preg = p $\hat{}_p$ breg')
$\wedge$ (rmode = enter $\rightarrow$ code(preg, db") = mkcl(the_clau(clause'(cllreg, db'))))
$\wedge$ (rmode = try
$\rightarrow$ $\langle$clls#(cllreg, db'; cal_1)$\rangle$
$\langle$chain-try-me#(preg, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))
$\wedge$ ($\forall$ st. st ins s
$\rightarrow$ (b $\hat{}_b$ st) ins s
$\wedge$ (sdecglseq $\hat{}_{sd}$ st) ctpelem s
$\wedge$ (f $\hat{}_f$ st) ins s'
$\wedge$ ssub $\hat{}_{su}$ st = ssub' $\hat{}_{su}$ f $\hat{}_f$ st
$\wedge$ f $\hat{}_f$ b $\hat{}_b$ st = b' $\hat{}_b$ f $\hat{}_f$ st
$\wedge$ fd(f, sdecglseq $\hat{}_{sd}$ st) = sdecglseq' $\hat{}_{sd}$ f $\hat{}_f$ st
$\wedge$ ($\forall$ st_1.st_1 ins s $\rightarrow$ (f $\hat{}_f$ st = f $\hat{}_f$ st_1 $\rightarrow$ st = st_1))
$\wedge$ $\langle$clls#(scll $\hat{}_{sc}$ st, db'; cal_1)$\rangle$
$\langle$chain-retry-me#(p $\hat{}_p$ f $\hat{}_f$ st, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))

Aber auch diese Version scheitert an dem cut-Lemma. Um INV.3 als Nachbedingung der cut-Regeln zu beweisen, muß auch das Konjunktionsglied

(rmode = retry $\rightarrow$ preg = p $\hat{}_p$ breg')

gezeigt werden. Dies folgt aber nicht aus der gleichen Teilformel der Vorbedingung INV.3, da das breg' in cp-cut# neu belegt wurde. Was wir benötigen, um obige Implikation zu zeigen, ist die Eigenschaft, daß *nach* der Ausführung von d-cut# rmode $\neq$ retry gilt. Da aber der rmode in d-cut# nicht aktualisiert wird, ist dies gleichbedeutend damit, daß *vor* d-cut# rmode $\neq$ retry gilt. Tatsächlich gilt immer, wenn der act = ! ist (also gcar(sdcar(decglseqreg).s1) = !), daß dann der rmode =call' ist. Um die anderen beiden nicht benutzerdefinierten Literale gleich mit zu erledigen, wurde die Eigenschaft verallgemeinert zu

$\neg$ is_user_defined(gcar(sdcar(decglseqreg).s1)) $\rightarrow$ rmode = call'.

Da aber die Listenoperatoren bei leeren Listen unspezifiziert sind, wäre die Implikation nicht immer entscheidbar. Darum werden die leeren Listen explizit berücksichtigt:

INV.4

$$
\begin{array}{l}
\exists\ \mathrm{f}.\quad \mathrm{stop} = \mathrm{stop'} \\
\quad \wedge\ \mathrm{vi} = \mathrm{vi'} \\
\quad \wedge\ \mathrm{subreg} = \mathrm{subreg'} \\
\quad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{breg} = \mathrm{breg'} \\
\quad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{ctreg} = \mathrm{ctreg'} \\
\quad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{bottom} = \mathrm{bottom} \\
\quad \wedge\ \mathrm{fd(f, decglseqreg)} = \mathrm{decglseqreg'} \\
\quad \wedge\ \mathrm{breg\ ins\ s} \\
\quad \wedge\ \mathrm{ctreg\ ins\ s} \\
\quad \wedge\ \mathrm{bottom\ ins\ s} \\
\quad \wedge\ \mathrm{decglseqreg\ ctpelem\ s} \\
\quad \wedge\ (\mathrm{rmode} = \mathrm{call'} \rightarrow \mathrm{preg} = \mathrm{start}) \\
\quad \wedge\ (\mathrm{rmode} = \mathrm{retry} \rightarrow \mathrm{preg} = \mathrm{p}\ \hat{}_p\ \mathrm{breg'}) \\
\quad \wedge\ (\mathrm{rmode} = \mathrm{enter} \rightarrow \mathrm{code(preg,\ db")} = \mathrm{mkcl(the\_clau(clause'(cllreg,\ db'))))} \\
\quad \wedge\ (\quad \mathrm{rmode} = \mathrm{try} \\
\qquad \rightarrow \langle \mathrm{clls\#(cllreg,\ db';\ cal_1)} \rangle \\
\qquad\quad \langle \mathrm{chain\text{-}try\text{-}me\#(preg,\ db";\ cal_2)} \rangle \\
\qquad\qquad \mathrm{mapcode(cal_2,\ db")} = \mathrm{mapclause'(cal_1,\ db')}) \\
\quad \wedge\ (\quad \mathbf{decglseqreg = sdnil} \\
\qquad \vee\ \mathbf{sdcar(decglseqreg).s1 = gnil} \\
\qquad \vee\ \neg\ \mathbf{is\_user\_defined(gcar(sdcar(decglseqreg).s1))} \\
\qquad \rightarrow \mathbf{rmode = call')} \\
\quad \wedge\ (\forall\ \mathrm{st}.\quad \mathrm{st\ ins\ s} \\
\qquad\quad \rightarrow \quad (\mathrm{b}\ \hat{}_b\ \mathrm{st})\ \mathrm{ins\ s} \\
\qquad\qquad \wedge\ (\mathrm{sdecglseq}\ \hat{}_{sd}\ \mathrm{st})\ \mathrm{ctpelem\ s} \\
\qquad\qquad \wedge\ (\mathrm{f}\ \hat{}_f\ \mathrm{st})\ \mathrm{ins\ s'} \\
\qquad\qquad \wedge\ \mathrm{ssub}\ \hat{}_{su}\ \mathrm{st} = \mathrm{ssub'}\ \hat{}_{su}\ \mathrm{f}\ \hat{}_f\ \mathrm{st} \\
\qquad\qquad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{b}\ \hat{}_b\ \mathrm{st} = \mathrm{b'}\ \hat{}_b\ \mathrm{f}\ \hat{}_f\ \mathrm{st} \\
\qquad\qquad \wedge\ \mathrm{fd(f,\ sdecglseq}\ \hat{}_{sd}\ \mathrm{st)} = \mathrm{sdecglseq'}\ \hat{}_{sd}\ \mathrm{f}\ \hat{}_f\ \mathrm{st} \\
\qquad\qquad \wedge\ (\forall\ \mathrm{st_1.st_1\ ins\ s} \rightarrow (\mathrm{f}\ \hat{}_f\ \mathrm{st} = \mathrm{f}\ \hat{}_f\ \mathrm{st_1} \rightarrow \mathrm{st} = \mathrm{st_1})) \\
\qquad\qquad \wedge\ \langle \mathrm{clls\#(scll}\ \hat{}_{sc}\ \mathrm{st,\ db';\ cal_1)} \rangle \\
\qquad\qquad\quad \langle \mathrm{chain\text{-}retry\text{-}me\#(p}\ \hat{}_p\ \mathrm{f}\ \hat{}_f\ \mathrm{st,\ db";\ cal_2)} \rangle \\
\qquad\qquad\qquad \mathrm{mapcode(cal_2,\ db")} = \mathrm{mapclause'(cal_1,\ db')})
\end{array}
$$

Der Nachweis des neuen Lemmas retry-newd-cp mit INV.4 scheitert jedoch genau an dem zuletzt eingeführten Konjunktionsglied. Sobald im Beweis das newd-retry# symbolisch ausgeführt ist, steht auf der rechten Seite der Sequenz u.a.:

$$
\begin{array}{l}
\quad \mathrm{sdecglseq_0}\ \hat{}_{sd}\ \mathrm{breg_0} = \mathrm{sdnil} \\
\vee\ \mathrm{sdcar(sdecglseq_0}\ \hat{}_{sd}\ \mathrm{breg_0).s1} = \mathrm{gnil} \\
\vee\ \mathrm{gcar(sdcar(sdecglseq_0}\ \hat{}_{sd}\ \mathrm{breg_0).s1)} = \mathrm{!} \\
\vee\ \mathrm{gcar(sdcar(sdecglseq_0}\ \hat{}_{sd}\ \mathrm{breg_0).s1)} = \mathrm{true'} \\
\rightarrow \mathrm{enter} = \mathrm{call'}
\end{array}
$$

Da aber enter $\neq$ call' ist, kann die Implikation nur gelten, wenn ihre linke Seite falsch ist. Dazu müsste man von der $\mathrm{sdecglseq_0}\ \hat{}_{sd}\ \mathrm{breg_0}$ wissen, daß sie werder leer ist noch ein erstes leeres goal hat und außerdem von einem benutzerdefinierten Literal angeführt wird. Dies

alles gilt tatsächlich, denn in der try-Regel wird niemals ein decglseqreg gekellert, das obige Bedingungen verletzt. Da jeder gekellerte Knoten wieder zum aktuellen breg werden kann, muß die Invariante diese Eigenschaft für alle states $\in$ s festhalten, ausgenommen bottom, weil dessen bedeutungslose decglseq die Bedingung verletzt:

INV.5

$\exists$ f. stop = stop'
$\wedge$ vi = vi'
$\wedge$ subreg = subreg'
$\wedge$ f $\hat{}_f$ breg = breg'
$\wedge$ f $\hat{}_f$ ctreg = ctreg'
$\wedge$ f $\hat{}_f$ bottom = bottom
$\wedge$ fd(f, decglseqreg) = decglseqreg'
$\wedge$ breg ins s
$\wedge$ ctreg ins s
$\wedge$ bottom ins s
$\wedge$ decglseqreg ctpelem s
$\wedge$ (rmode = call' $\rightarrow$ preg = start)
$\wedge$ (rmode = retry $\rightarrow$ preg = p $\hat{}_p$ breg')
$\wedge$ (rmode = enter $\rightarrow$ code(preg, db") = mkcl(the_clau(clause'(cllreg, db'))))
$\wedge$ (rmode = try
$\rightarrow$ $\langle$clls#(cllreg, db'; cal_1)$\rangle$
$\langle$chain-try-me#(preg, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))
$\wedge$ (decglseqreg = sdnil
$\vee$ sdcar(decglseqreg).s1 = gnil
$\vee$ $\neg$ is_user_defined(gcar(sdcar(decglseqreg).s1))
$\rightarrow$ rmode = call')
$\wedge$ ($\forall$ st. st ins s
$\rightarrow$ (b $\hat{}_b$ st) ins s
$\wedge$ (sdecglseq $\hat{}_{sd}$ st) ctpelem s
$\wedge$ (f $\hat{}_f$ st) ins s'
$\wedge$ ssub $\hat{}_{su}$ st = ssub' $\hat{}_{su}$ f $\hat{}_f$ st
$\wedge$ f $\hat{}_f$ b $\hat{}_b$ st = b' $\hat{}_b$ f $\hat{}_f$ st
$\wedge$ fd(f, sdecglseq $\hat{}_{sd}$ st) = sdecglseq' $\hat{}_{sd}$ f $\hat{}_f$ st
$\wedge$ ($\forall$ st_1.st_1 ins s $\rightarrow$ (f $\hat{}_f$ st = f $\hat{}_f$ st_1 $\rightarrow$ st = st_1))
$\wedge$ (**st $\neq$ bottom**
$\rightarrow$ **sdecglseq $\hat{}_{sd}$ st $\neq$ sdnil**
$\wedge$ **sdcar(sdecglseq $\hat{}_{sd}$ st).s1 $\neq$ gnil**
$\wedge$ **is_user_defined(gcar(sdcar(sdecglseq $\hat{}_{sd}$ st).s1)))**
$\wedge$ $\langle$clls#(scll $\hat{}_{sc}$ st, db'; cal_1)$\rangle$
$\langle$chain-retry-me#(p $\hat{}_p$ f $\hat{}_f$ st, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))

Etwas anders gelagert ist die folgende Modifikation: Bei dem erneuten Versuch des Nachweises von retry-newd-cp mit INV.5 tut sich an einer Stelle ein Beweisast auf, auf dem breg = bottom gilt. Da in newd-retry# die Werte der dynamischen Funktionen auf breg in die Register geladen werden, müssen in der *Nach*bedingung INV.5 viele Aussagen über diese

Funktionswerte nachgewiesen werden. Auf dem Ast mit breg = bottom heißt das, daß etliche Formeln bewiesen werden müssen über Funktionswerte dynamischer Funktionen auf bottom. Da abzusehen war, daß sich aus diesem Ast ein beträchtlicher Baum entwickelt haben würde, ist der Beweis gar nicht erst versucht worden. Man kann sich nämlich überlegen, daß im Modus retry der Wert von breg niemals gleich bottom sein kann, denn dieser Modus wird nur über d-backtrack# erreicht, wo der bottom-Fall abgefangen wird und zur Terminierung führt: die Berücksichtigung dieser Invarianz in der INV.6-Formel erspart uns den möglicherweise zwecklosen, in jedem Fall aber unsinnigen Versuch, in retry-newd-cp einen bottom-Ast zu beweisen.

INV.6

$$
\begin{array}{l}
\exists\, \mathrm{f}.\quad \mathrm{stop} = \mathrm{stop'} \\
\quad \wedge\ \mathrm{vi} = \mathrm{vi'} \\
\quad \wedge\ \mathrm{subreg} = \mathrm{subreg'} \\
\quad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{breg} = \mathrm{breg'} \\
\quad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{ctreg} = \mathrm{ctreg'} \\
\quad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{bottom} = \mathrm{bottom} \\
\quad \wedge\ \mathrm{fd(f,\ decglseqreg)} = \mathrm{decglseqreg'} \\
\quad \wedge\ \mathrm{breg\ ins\ s} \\
\quad \wedge\ \mathrm{ctreg\ ins\ s} \\
\quad \wedge\ \mathrm{bottom\ ins\ s} \\
\quad \wedge\ \mathrm{decglseqreg\ ctpelem\ s} \\
\quad \wedge\ (\mathrm{rmode} = \mathrm{call'} \rightarrow \mathrm{preg} = \mathrm{start}) \\
\quad \wedge\ (\mathrm{rmode} = \mathrm{retry} \rightarrow \mathbf{breg} \neq \mathbf{bottom} \wedge \mathrm{preg} = \mathrm{p}\ \hat{}_p\ \mathrm{breg'}) \\
\quad \wedge\ (\mathrm{rmode} = \mathrm{enter} \rightarrow \mathrm{code(preg,\ db")} = \mathrm{mkcl(the\_clau(clause'(cllreg,\ db'))))} \\
\quad \wedge\ (\quad \mathrm{rmode} = \mathrm{try} \\
\qquad \rightarrow \langle \mathrm{clls\#(cllreg,\ db';\ cal_1)} \rangle \\
\qquad\qquad \langle \mathrm{chain\text{-}try\text{-}me\#(preg,\ db";\ cal_2)} \rangle \\
\qquad\qquad\qquad \mathrm{mapcode(cal_2,\ db")} = \mathrm{mapclause'(cal_1,\ db')}) \\
\quad \wedge\ (\quad \mathrm{decglseqreg} = \mathrm{sdnil} \\
\qquad \vee\ \mathrm{sdcar(decglseqreg).s1} = \mathrm{gnil} \\
\qquad \vee\ \neg\ \mathrm{is\_user\_defined(gcar(sdcar(decglseqreg).s1))} \\
\qquad \rightarrow \mathrm{rmode} = \mathrm{call'}) \\
\quad \wedge\ (\forall\ \mathrm{st}.\quad \mathrm{st\ ins\ s} \\
\qquad\quad \rightarrow \quad (\mathrm{b}\ \hat{}_b\ \mathrm{st})\ \mathrm{ins\ s} \\
\qquad\qquad \wedge\ (\mathrm{sdecglseq}\ \hat{}_{sd}\ \mathrm{st})\ \mathrm{ctpelem\ s} \\
\qquad\qquad \wedge\ (\mathrm{f}\ \hat{}_f\ \mathrm{st})\ \mathrm{ins\ s'} \\
\qquad\qquad \wedge\ \mathrm{ssub}\ \hat{}_{su}\ \mathrm{st} = \mathrm{ssub'}\ \hat{}_{su}\ \mathrm{f}\ \hat{}_f\ \mathrm{st} \\
\qquad\qquad \wedge\ \mathrm{f}\ \hat{}_f\ \mathrm{b}\ \hat{}_b\ \mathrm{st} = \mathrm{b'}\ \hat{}_b\ \mathrm{f}\ \hat{}_f\ \mathrm{st} \\
\qquad\qquad \wedge\ \mathrm{fd(f,\ sdecglseq}\ \hat{}_{sd}\ \mathrm{st}) = \mathrm{sdecglseq'}\ \hat{}_{sd}\ \mathrm{f}\ \hat{}_f\ \mathrm{st} \\
\qquad\qquad \wedge\ (\forall\ \mathrm{st_1.st_1\ ins\ s} \rightarrow (\mathrm{f}\ \hat{}_f\ \mathrm{st} = \mathrm{f}\ \hat{}_f\ \mathrm{st_1} \rightarrow \mathrm{st} = \mathrm{st_1})) \\
\qquad\qquad \wedge\ (\quad \mathrm{st} \neq \mathrm{bottom} \\
\qquad\qquad\quad \rightarrow \quad \mathrm{sdecglseq}\ \hat{}_{sd}\ \mathrm{st} \neq \mathrm{sdnil} \\
\qquad\qquad\qquad \wedge\ \mathrm{sdcar(sdecglseq}\ \hat{}_{sd}\ \mathrm{st).s1} \neq \mathrm{gnil} \\
\qquad\qquad\qquad \wedge\ \mathrm{is\_user\_defined(gcar(sdcar(sdecglseq}\ \hat{}_{sd}\ \mathrm{st).s1)))} \\
\qquad\qquad \wedge\ \langle \mathrm{clls\#(scll}\ \hat{}_{sc}\ \mathrm{st,\ db';\ cal_1)} \rangle \\
\qquad\qquad\qquad \langle \mathrm{chain\text{-}retry\text{-}me\#(p}\ \hat{}_p\ \mathrm{f}\ \hat{}_f\ \mathrm{st,\ db";\ cal_2)} \rangle
\end{array}
$$

$$\text{mapcode}(cal_2, db'') = \text{mapclause'}(cal_1, db'))$$

4.4.3 Aufdeckung des Indeterminismus mit KIV

Die Analyse des nächsten scheiternden Beweises offenbart nicht nur einen Fehler in unserer Invarianten INV.6, sondern auch einen Fehler in unseren Programmen, der schon auf [BörgerRosenzweig] zurückgeht. Hier zeigt es sich, daß die Methode nicht nur geeignet ist, korrekte Stufen zu verifizieren, sondern auch, Fehler aufzudecken.

Der Beweisversuch von fail-newd-cp mit INV.6 führt zu einem nicht schließbaren Ast mit der Sequenz:

$$\begin{array}{l} \neg\ (\quad \text{decglseqreg}_0 = \text{sdnil} \\ \qquad \vee\ \text{sdcar(decglseqreg).s1} = \text{gnil} \\ \qquad \vee\ \neg\ \text{is\_user\_defined(fail)'} \\ \quad \rightarrow\ \text{retry} = \text{call'}), \\ \Gamma \\ \vdash \\ \Delta \end{array}$$

Diese Sequenz läßt sich nicht beweisen, denn sie ist äquivalent mit:

$\neg$ (true$\rightarrow$ false), $\Gamma \vdash \Delta$

oder kurz

true,$\Gamma \vdash \Delta$

(Γ und Δ tragen nichts bei.)
Bisher hatten wir angenommen, daß gilt:

$\neg$ is_user_defined(act) $\rightarrow$ mode = call

Statt dessen gilt aber nach der Ausführung von newd-fail#:

$\neg$ is_user_defined(act) $\wedge$ mode =retry,

weil das decglseqreg in der fail-Regel nicht überschrieben wird und darum immer noch act = fail' gilt. Das zeigt, daß unsere in INV.4 eingeführte Teilformel zu stark war. Für act = ! oder true' gilt, daß rmode = call' ist, nicht aber für fail'.

Zunächst verbessern wir die Invariante:

INV.7

$$\begin{array}{l} \exists\ \text{f.}\quad \text{stop} = \text{stop'} \\ \quad \wedge\ \text{vi} = \text{vi'} \\ \quad \wedge\ \text{subreg} = \text{subreg'} \\ \quad \wedge\ \text{f}\ \hat{}_f\ \text{breg} = \text{breg'} \\ \quad \wedge\ \text{f}\ \hat{}_f\ \text{ctreg} = \text{ctreg'} \\ \quad \wedge\ \text{f}\ \hat{}_f\ \text{bottom} = \text{bottom} \\ \quad \wedge\ \text{fd(f, decglseqreg)} = \text{decglseqreg'} \\ \quad \wedge\ \text{breg ins s} \\ \quad \wedge\ \text{ctreg ins s} \end{array}$$

$\wedge$ bottom ins s
$\wedge$ decglseqreg ctpelem s
$\wedge$ (rmode = call' $\rightarrow$ preg = start)
$\wedge$ (rmode = retry $\rightarrow$ breg $\neq$ bottom $\wedge$ preg = p $\hat{}_p$ breg')
$\wedge$ (rmode = enter $\rightarrow$ code(preg, db") = mkcl(the_clau(clause'(cllreg, db'))))
$\wedge$ (rmode = try
$\rightarrow$ $\langle$clls#(cllreg, db'; cal_1)$\rangle$
$\langle$chain-try-me#(preg, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))
$\wedge$ (decglseqreg = sdnil
$\vee$ sdcar(decglseqreg).s1 = gnil
$\vee$ **gcar(sdcar(decglseqreg).s1) = !**
$\vee$ **gcar(sdcar(decglseqreg).s1) = true'**
$\rightarrow$ rmode = call')
$\wedge$ ($\forall$ st. st ins s
$\rightarrow$ (b $\hat{}_b$ st) ins s
$\wedge$ (sdecglseq $\hat{}_{sd}$ st) ctpelem s
$\wedge$ (f $\hat{}_f$ st) ins s'
$\wedge$ ssub $\hat{}_{su}$ st = ssub' $\hat{}_{su}$ f $\hat{}_f$ st
$\wedge$ f $\hat{}_f$ b $\hat{}_b$ st = b' $\hat{}_b$ f $\hat{}_f$ st
$\wedge$ fd(f, sdecglseq $\hat{}_{sd}$ st) = sdecglseq' $\hat{}_{sd}$ f $\hat{}_f$ st
$\wedge$ ($\forall$ st_1.st_1 ins s $\rightarrow$ (f $\hat{}_f$ st = f $\hat{}_f$ st_1 $\rightarrow$ st = st_1))
$\wedge$ (st $\neq$ bottom
$\rightarrow$ sdecglseq $\hat{}_{sd}$ st $\neq$ sdnil
$\wedge$ sdcar(sdecglseq $\hat{}_{sd}$ st).s1 $\neq$ gnil
$\wedge$ is_user_defined(gcar(sdcar(sdecglseq $\hat{}_{sd}$ st).s1)))
$\wedge$ $\langle$clls#(scll $\hat{}_{sc}$ st, db'; cal_1)$\rangle$
$\langle$chain-retry-me#(p $\hat{}_p$ f $\hat{}_f$ st, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))

Die schwerwiegendere Konsequenz aber ist, daß unsere Programme (d-..., newd-..., cp-...) an dieser Stelle in eine Endlosschleife laufen, weil immer wieder die fail-Regel angesprungen wird. Ein Blick auf die dazugehörigen dynamischen Algebren (Kap. 2 oder [BörgerRosenzweig]) zeigt, daß der Fehler schon dort liegt. Die Regeleinstiegsbedingungen von fail und retry schließen sich nicht aus, weil nach der fail-Regel immer noch ihre eigene Einstiegsbedingung gilt und außerdem diejenige der retry-Regel. Dieser Indeterminismus ist natürlich nicht intendiert. „... in this restricted context, ... only deterministic terminating computations are of interest ..."[BörgerRosenzweig]. Sonst wären auch die Stufen größergleich III nicht äquivalent zu I und II.

Bei der Übertragung von *deterministischen* dynamischen Algebren in Programme hätte die Schachtelung der Einstiegsbedingungen keine Rolle gespielt. Anders ist dies bei Indeterminismen, die durch if-then-else-Konstrukte immer aufgelöst werden, und zwar in einer festen Richtung.

Den unvermuteten und auch fehlerhaften Indeterminismus hatten wir zufällig gerade so aufgelöst, daß ein fail in den endgültigen Absturz führt:

```
...
if is_user_defined(act)
   then if rmode = call'
        ...
   ...
```

Die notwendige Verbesserung der Regeln in den dynamischen Algebren ist minimal: Die fail-Regeln müssen beginnen mit „if act = fail & mode = call ...“ bzw. „if act = fail & preg = start ...“. Für unsere Programme bedeutet dies, daß die Bedingungen is_user_defined(act) und rmode = call' (bzw. preg = start) genau anders herum geschachtelt werden müssen.

```
NEW-D-BODY#(db',procdef'; var s,vi,rmode,stop,breg,ctreg,
                          ssub,subreg,sdecglseq,decglseqreg,scll,cllreg,b)
begin
if decglseqreg = sdnil
then D-QUERY-SUCCESS#(; stop)
else var goal = (sdcar(decglseqreg)).s1
     in if goal = gnil then D-GOAL-SUCCESS#(; decglseqreg)
        else var act = gcar(goal),
                  scutpt = (sdcar(decglseqreg)).s2
             in var scont =      mksdecgoal(gcdr(goal),scutpt)
                              +sdl sdcdr(decglseqreg)
                in if rmode = call'
                   then if is_user_defined(act)
                        then NEW-D-CALL#(act,procdef',breg,db';
                                         cllreg,rmode,ctreg,stop)
                        else if act = true'
                        then D-TRUE#(scont; decglseqreg)
                        else if act = fail'
                        then D-FAIL#(breg; stop, rmode)
                        else (: act = ! :)
                             D-CUT#(scont,scutpt; breg, decglseqreg)
                   else if rmode = try
                        then NEW-D-TRY#(cllreg,subreg,decglseqreg,db';
                                        scll,ssub,breg,b,sdecglseq,s,rmode)
                        else if rmode = enter
                        then D-ENTER#(db',act,scont,cllreg,breg,ctreg;
                                      subreg,decglseqreg,vi,stop,rmode)
                        else (: rmode = retry :)
                             if clause'(next(scll ^sc breg), db') # null
                             then NEW-D-RETRY#(sdecglseq,b,ssub,breg;
                                               cllreg,subreg,scll,
                                               ctreg,decglseqreg,rmode)
                             else NEW-D-TRUST#(sdecglseq,b,ssub,scll;
                                               cllreg,subreg,breg,
                                               ctreg,decglseqreg,rmode)
```

```
end;

CP-BODY#(db'',procdef''; var s',vi',stop',breg',ctreg',
                         ssub',subreg',sdecglseq',decglseqreg',p,preg,b')
begin
if decglseqreg' = sdnil
then CP-QUERY-SUCCESS#(; stop')
else var goal = (sdcar(decglseqreg')).s1
     in if goal = gnil then CP-GOAL-SUCCESS#(; decglseqreg')
        else var act = gcar(goal),
                 scutpt = (sdcar(decglseqreg')).s2
             in var scont = mksdecgoal(gcdr(goal),scutpt) +sdl sdcdr(decglseqreg')
                in if preg = start
                   then if is_user_defined(act)
                        then CP-CALL#(act,procdef'',breg',p,db'';
                                      preg,ctreg',stop')
                        else if act = true'
                        then CP-TRUE#(scont; decglseqreg')
                        else if act = fail'
                        then CP-FAIL#(breg',p; stop',preg)
                        else (: act = ! :)
                             CP-CUT#(scont,scutpt; breg', decglseqreg')
                   else if is_try_me(code(preg, db''))
                        then CP-TRY-ME#(subreg',decglseqreg',db'';
                                        p,ssub',breg',b',sdecglseq',s',preg)
                        else if is_clause(code(preg, db''))
                        then CP-ENTER#(db'',act,scont,breg',ctreg',p;
                                       subreg',decglseqreg',vi',stop',preg)
                        else if is_retry_me(code(preg, db''))
                        then CP-RETRY-ME#(db'',sdecglseq',b',ssub',breg';
                                          preg,subreg',p,ctreg',decglseqreg')
                        else (: is_trust_me(code(preg, db'')) :)
                             CP-TRUST-ME#(sdecglseq',b',ssub';
                                          preg,subreg',breg',ctreg',decglseqreg')
end;
```

Was das Ausmaß der Verbesserung angeht, handelt es sich um einen sehr kleinen Fehler. Es gilt aber hier wie allgemein, daß die syntaktische Geringfügigkeit eines Fehlers nicht als Maß für seine Konsequenz taugt. (Hier: der indeterministische „Absturz“ der Interpretation.) So werden z.B. häufig schwerwiegende Softwarefehler durch Indexvertauschungen und ähnliche „Kleinigkeiten“ verursacht. Je unauffälliger eine Ungenauigkeit ist, desto schwieriger ist ihre Entdeckung. Gerade hier liegt unbestreitbar die Stärke der formalen Methoden, die wir anwenden.

Nun werden wieder die *rule*-newd-cp-Lemmata modifiziert, diesmal nicht nur in der Invarianten, sondern auch in den Prozedurdeklarationen.

4.4.4 Fortsetzung der Vervollständigung

Leider fehlt immer noch eine Kleinigkeit zur vollständigen Invarianz von INV.7. Beim Beweis von try-newd-cp ergibt sich noch eine einzige, nicht schließbare Sequenz:

$\neg$ is_user_defined(act), $\Gamma \vdash$

Eine Analyse des Beweisbaumes zeigt, daß dieser Ast entstanden ist, um die Formel

st $\neq$ bottom
$\rightarrow$ sdecglseq $\hat{}_{sd}$ st $\neq$ sdnil
$\wedge$ sdcar(sdecglseq $\hat{}_{sd}$ st).s1 $\neq$ gnil
$\wedge$ is_user_defined(gcar(sdcar(sdecglseq $\hat{}_{sd}$ st).s1))

auch für den in newd-try# neu geschaffenen Knoten zu zeigen.

Man kann sich überlegen, daß im Modus try der act immer benutzerdefiniert ist und gleiches nach der try-Regel für die decglseq des neuen Knoten gilt, da sie mit der decglseqreg vor der try-Regel übereinstimmt. Wir erweitern die Invariante entsprechend.

INV.8

$\exists$ f. stop = stop'
$\wedge$ vi = vi'
$\wedge$ subreg = subreg'
$\wedge$ f $\hat{}_f$ breg = breg'
$\wedge$ f $\hat{}_f$ ctreg = ctreg'
$\wedge$ f $\hat{}_f$ bottom = bottom
$\wedge$ fd(f, decglseqreg) = decglseqreg'
$\wedge$ breg ins s
$\wedge$ ctreg ins s
$\wedge$ bottom ins s
$\wedge$ decglseqreg ctpelem s
$\wedge$ (rmode = call' $\rightarrow$ preg = start)
$\wedge$ (rmode = retry $\rightarrow$ breg $\neq$ bottom $\wedge$ preg = p $\hat{}_p$ breg')
$\wedge$ (rmode = enter $\rightarrow$ code(preg, db") = mkcl(the_clau(clause'(cllreg, db'))))
$\wedge$ (rmode = try
$\rightarrow$ **is_user_defined(gcar(sdcar(decglseqreg).s1))**
$\wedge$ $\langle$clls#(cllreg, db'; cal_1)$\rangle$
$\langle$chain-try-me#(preg, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))
$\wedge$ (decglseqreg = sdnil
$\vee$ sdcar(decglseqreg).s1 = gnil
$\vee$ gcar(sdcar(decglseqreg).s1) = !
$\vee$ **gcar(sdcar(decglseqreg).s1) = true'**
$\rightarrow$ rmode = call')
$\wedge$ ($\forall$ st. st ins s
$\rightarrow$ (b $\hat{}_b$ st) ins s
$\wedge$ (sdecglseq $\hat{}_{sd}$ st) ctpelem s
$\wedge$ (f $\hat{}_f$ st) ins s'
$\wedge$ ssub $\hat{}_{su}$ st = ssub' $\hat{}_{su}$ f $\hat{}_f$ st
$\wedge$ f $\hat{}_f$ b $\hat{}_b$ st = b' $\hat{}_b$ f $\hat{}_f$ st

$$
\begin{array}{l}
\wedge\ \mathrm{fd(f, sdecglseq\ \hat{}_{sd}\ st) = sdecglseq'\ \hat{}_{sd}\ f\ \hat{}_{f}\ st} \\
\wedge\ (\forall\ \mathrm{st_1.st_1\ ins\ s} \rightarrow (\mathrm{f\ \hat{}_{f}\ st = f\ \hat{}_{f}\ st_1} \rightarrow \mathrm{st = st_1})) \\
\wedge\ (\quad \mathrm{st} \neq \mathrm{bottom} \\
\quad \rightarrow \quad \mathrm{sdecglseq\ \hat{}_{sd}\ st} \neq \mathrm{sdnil} \\
\qquad \wedge\ \mathrm{sdcar(sdecglseq\ \hat{}_{sd}\ st).s1} \neq \mathrm{gnil} \\
\qquad \wedge\ \mathrm{is\_user\_defined(gcar(sdcar(sdecglseq\ \hat{}_{sd}\ st).s1)))} \\
\wedge\ \langle \mathrm{clls\#(scll\ \hat{}_{sc}\ st,\ db';\ cal_1)} \rangle \\
\qquad \langle \mathrm{chain\text{-}retry\text{-}me\#(p\ \hat{}_{p}\ f\ \hat{}_{f}\ st,\ db";\ cal_2)} \rangle \\
\qquad\quad \mathrm{mapcode(cal_2,\ db") = mapclause'(cal_1,\ db'))}
\end{array}
$$

Mit INV.8 lassen sich alle *rule*-newd-cp-Lemmata beweisen. Die Formel ist tatsächlich invariant gegenüber den Regeln. Damit ist der Induktionsschritt im Kern gelungen. Um nun den kompletten Induktionsbeweis zu führen, müssen noch newd-cp-step, newd-cp-ind und newd-cp-imp[8] bewiesen werden. Dies macht auch erst jetzt Sinn, nach der Entwicklung einer echten Invarianten.

Beim Beweis von newd-cp-imp entsteht ein Ast, der (implizit) den Induktionsanfang behandelt. Dort muß gezeigt werden, daß unmittelbar vor Eintritt in die while-Schleifen in newd-eval#, d.h. nach der Initialiserung der Variablen, die Formel INV.8 gilt. Auf diesem Ast ergibt sich die Sequenz:

$$
\begin{array}{l}
\quad \mathrm{instr_0 = code(failcode,\ db"),\ \Gamma} \\
\vdash \\
\quad \langle \mathrm{abort} \rangle\ \mathrm{mapcode(cal_0,\ db") = mapclause'(canil,db')}
\end{array}
$$

failcode ist die undefinierte Adresse, mit der die dynamische Funktion p initialisiert wird. Diese Sequenz ist natürlich falsch, weil abort nicht terminiert. Verfolgen wir die Ursache zurück: Die Vorgängersequenz lautete:

$$
\begin{array}{l}
\quad \mathrm{instr_0 = code(failcode,\ db"),\ \Gamma} \\
\vdash \\
\quad \langle \mathbf{if}\ \mathrm{is\_trust\_me(instr_0)} \\
\quad\ \mathbf{then}\ \alpha \\
\quad\ \mathbf{else}\ \mathrm{abort} \rangle\ \mathrm{mapcode(cal_2,\ db") = mapclause'(canil,db')}
\end{array}
$$

Diese wiederum ging hervor aus:

$$
\begin{array}{l}
\quad \Gamma \\
\vdash \\
\quad \langle \mathrm{chain\text{-}retry\text{-}me\#(failcode,\ db"));\ cal_2} \rangle \\
\qquad \mathrm{mapcode(cal_0,\ db") = mapclause'(canil,db')}
\end{array}
$$

Die Ursache liegt darin, daß für alle st aus s in INV.8 gefordert wird, daß chain-retry-me# für p $\hat{}_p$ f $\hat{}_f$ st terminiert. Dies ist aber für bottom nicht der Fall. Da wir aber für bottom keine solche Eigenschaft benötigen, schließen wir es aus. Das Ergebnis ist die endgültige Invariante INV, mit der schließlich alle Beweise gelungen sind. Sie wird nun abschließend angegeben. Um noch einmal einen Eindruck für die Entwicklung von INV.1 bis INV zu vermitteln, sind alle Teilformeln, die nicht schon zu INV.1 gehören, kursiv dargestellt.

[8] vgl. Abschn. 4.3

INV

$\exists$ f. stop = stop'
$\wedge$ vi = vi'
$\wedge$ subreg = subreg'
$\wedge$ f $\hat{}_f$ breg = breg'
$\wedge$ f $\hat{}_f$ ctreg = ctreg'
$\wedge$ *f* $\hat{}_f$ *bottom = bottom*
$\wedge$ fd(f, decglseqreg) = decglseqreg'
$\wedge$ breg ins s
$\wedge$ ctreg ins s
$\wedge$ *bottom ins s*
$\wedge$ *decglseqreg ctpelem s*
$\wedge$ (rmode = call' $\rightarrow$ preg = start)
$\wedge$ (rmode = retry $\rightarrow$ *breg* $\neq$ *bottom* $\wedge$ preg = p $\hat{}_p$ breg')
$\wedge$ (rmode = enter $\rightarrow$ code(preg, db") = mkcl(the_clau(clause'(cllreg, db'))))
$\wedge$ (rmode = try
$\rightarrow$ *is_user_defined(gcar(sdcar(decglseqreg).s1))*
$\wedge$ $\langle$clls#(cllreg, db'; cal_1)$\rangle$
$\langle$chain-try-me#(preg, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db'))
$\wedge$ (*decglseqreg = sdnil*
$\vee$ *sdcar(decglseqreg).s1 = gnil*
$\vee$ *gcar(sdcar(decglseqreg).s1) = !*
$\vee$ *gcar(sdcar(decglseqreg).s1) = true'*
$\rightarrow$ *rmode = call')*
$\wedge$ ($\forall$ st. st ins s
$\rightarrow$ (b $\hat{}_b$ st) ins s
$\wedge$ (sdecglseq $\hat{}_{sd}$ st) ctpelem s
$\wedge$ (f $\hat{}_f$ st) ins s'
$\wedge$ ssub $\hat{}_{su}$ st = ssub' $\hat{}_{su}$ f $\hat{}_f$ st
$\wedge$ f $\hat{}_f$ b $\hat{}_b$ st = b' $\hat{}_b$ f $\hat{}_f$ st
$\wedge$ fd(f, sdecglseq $\hat{}_{sd}$ st) = sdecglseq' $\hat{}_{sd}$ f $\hat{}_f$ st
$\wedge$ ($\forall$ st_1.st_1 ins s $\rightarrow$ (f $\hat{}_f$ st = f $\hat{}_f$ st_1 $\rightarrow$ st = st_1))
$\wedge$ (*st* $\neq$ *bottom*
$\rightarrow$ *sdecglseq* $\hat{}_{sd}$ *st* $\neq$ *sdnil*
$\wedge$ *sdcar(sdecglseq* $\hat{}_{sd}$ *st).s1* $\neq$ *gnil*
$\wedge$ *is_user_defined(gcar(sdcar(sdecglseq* $\hat{}_{sd}$ *st).s1)))*
$\wedge$ (**st** $\neq$ **bottom**
$\rightarrow$ $\langle$clls#(scll $\hat{}_{sc}$ st, db'; cal_1)$\rangle$
$\langle$chain-retry-me#(p $\hat{}_p$ f $\hat{}_f$ st, db"; cal_2)$\rangle$
mapcode(cal_2, db") = mapclause'(cal_1, db')))

4.5 Beweisstruktur der zentralen Lemmata

Die Beweise der *rule*-newd-cp-Lemmata bildeten das Herzstück der Verifikation, speziell der Herleitung einer vollständigen Invariante. Obwohl wir im letzten Abschnitt immer wieder auf einzelne Beweise hingewiesen haben, wollen wir abschließend einen Überblick geben über die ihnen innewohnende Struktur.

Zunächst einmal gilt es, mit Hilfe der jeweiligen Regeleinstiegsbedingungen die richtigen Regeln anzusteuern. Dazu wird zunächst nach Anwendung von call-left die Prozedur newd-body# ausgeführt. Da die zusätzlichen Vorbedingungen gerade auf die Ansteuerungen der jeweiligen newd-*rule*# zugeschnitten sind, gelingt selbige ohne Interaktion. Bevor diese Regel betreten wird, soll erst mit call-left und symbolischer Ausführung die entsprechende cp-*rule*# angesteuert werden. Unmittelbar zur Verfügung stehen aber nur die newd-Einstiegsbedingungen. Von diesen muß auf die entsprechenden cp-Bedingungen geschlossen werden, und zwar mit Hilfe der Invarianten.

Im Falle rmode = call' geht das sehr leicht mit

$$\text{rmode} = \text{call'} \rightarrow \text{preg} = \text{start}$$

und zwar vollautomatisch. Auch für den enter-Fall war hierfür lediglich eine Interaktion nötig.

Etwas schwieriger ist es, im try-Fall das cp-try-me# anzusteuern. Die dafür notwendige Information, nämlich daß preg auf eine try_me_else-Instruktion zeigt, müssen wir erst aus der Invarianten „herauskitzeln". Dort steht abgekürzt:

$$\text{rmode} = \text{try} \rightarrow \langle \text{chain-try-me\#(preg,db";cal}_2\text{)} \rangle\ \varphi$$

D.h. chain-try-me# muß terminieren. Diese Prozedur ist aber gerade so konstruiert, daß sie abstürzt, wenn preg nicht auf eine try_me-Instruktion zeigt. Im Beweis machen wir uns das zunutze, indem mit if-right auch die auszuschließenden Prozeduräste von cp-eval# betreten werden, z.B. den mit preg = start bzw. is_clause() usw... In den betreffenden Ästen wird dann chain-try-me# aufgerufen und symbolisch ausgeführt, was immer zu einer Sequenz führt wie:

$$\langle \text{abort} \rangle\ \varphi, \Gamma \vdash \Delta$$

Diese wird automatisch geschlossen. Bei den Beweisen mit retry und trust wird entsprechend vorgegangen.

Der verbleibende Ast ist derjenige, der die „richtige" cp-*rule*# ansteuert. Jetzt stehen in der Sequenz links newd-*rule*# und rechts cp-*rule*# sozusagen zum Aufruf bereit. Nun wird wieder rechts und links symbolisch ausgeführt. Bei den flachen Regeln geht das wieder vollautomatisch. Auf die Schwierigkeiten, die bei Konditionalen auftreten können, haben wir beim Übergang von INV.1 nach INV.2 bereits hingewiesen. Danach sehen die offenen Sequenzen ungefähr so aus:

$$\text{INV}(\textit{inputs}), \Gamma \vdash \text{INV}(\textit{outputs})$$

Jetzt wird rechts das f instanziiert, und zwar mit dem linken f oder (nur bei try-newd-cp) mit einer Erweiterung davon. Dann gilt es, alle Konjunktionsglieder von INV auf der rechten Seite, die nicht trivialerweise gleich verschwinden, nachzuweisen. Dieses durchaus noch aufwendige Unterfangen wollen wir hier nicht weiter darstellen. Wichtig ist, daß das Prinzip der *rule*-newd-cp-Beweise deutlich geworden ist.

Die Compiler-Annahmen werden direkt nur in call-newd-cp verwendet, weil nur hier in den Prozedurdefinitions-Tabellen nachgeschlagen wird. Indirekt bleiben sie durch die Invariante präsent für die Verwendung in anderen Beweisen!

Wie schon erwähnt, lassen sich die Lemmata der Gegenrichtung *rule*-cp-newd sehr leicht auf die besprochenen zurückführen, da sie sich „logisch“ nur in der trivialen Voraussetzung der Regelterminierung unterscheiden. Die Beweise der Induktionslemmata cp-newd-step, sp-newd-ind und cp-newd-imp verlaufen nach dem gleichen Muster wie die der newd-cp-Spiegelbilder.

Insgesamt haben wir es also geschafft, die beiden Implikationsrichtungen zu beweisen, womit die Äquivalenz der Programme newd-eval# und cp-eval# gezeigt ist. Zusammen mit d-newd-equiv ist also der Nachweis gelungen, daß die Stufen IV und V' unter der Voraussetzung der Compiler-Annahmen äquivalent sind.

4.6 Beweisstatistik

Es wurden vierunddreißig Lemmata bewiesen mit zusammengenommen 1886 Beweisschritten, davon 468 Interaktionen. Damit liegt der Automatisierungsgrad bei 75,1%. Der Hauptanteil aller Beweisschritte, nämlich 44,8%, wird von der Heuristik „symbolic execution“ automatisch ausgeführt. Nach der Formalisierung der Beweisaufgabe wurden für die Verifikation des Top-Lemmas d-cp-equiv vier Personenwochen benötigt. Währenddessen wurden 390 Zeilen Programmcode bewiesen. Das sind im Schnitt 19,5 Zeilen pro Tag.

Der Vergleich mit anderen Verifikationsprojekten ist allerdings schwierig, weil es dort meistens um die Korrektheit von Programmen gegenüber einer algebraischen Spezifikation geht. Hier jedoch stand der Nachweis der Äquivalenz von Programmen im Mittelpunkt.

Kapitel 5

5.1 Ausblick

In der vorliegenden Arbeit wurde gezeigt, wie sich die Äquivalenz von sprachinterpretierenden abstrakten Maschinen streng formal verifizieren läßt. Dabei ist deutlich geworden, daß sich die Problemkomplexität nur durch ein leistungsfähiges Verifikationssystem beherrschen läßt. Außerdem muß das System die Fähigkeit besitzen, ungültige Beweise korrigierter Lemmata wiederzuverwenden, da sich die notwendigen Induktionsbehauptungen nur sukzessive entwickeln lassen.

Wir konnten den Nachweis erbringen, daß streng formale Methoden in der Lage sind, unvollständige Annahmen zu vervollständigen und Fehler zu finden, gerade wenn diese sehr unauffällig sind.

Tatsächlich nachgewiesen wurde nur *eine* der in [BörgerRosenzweig] postulierten Äquivalenzen. In der Fortsetzung ergeben sich daraus weitere Projekte:

- die Verifikation der im Rahmen dieser Arbeit bereits formalisierten Äquivalenzlemmata,
- die Spezifikation und Verifikation aller dynamischen Algebren aus [BörgerRosenzweig], zusammen mit der Formalisierung der Compiler-Annahmen.

Das Ergebnis wäre der Nachweis, daß die Compiler-Annahmen formal als hinreichend für die Äquivalenz von WAM und PROLOG bewiesen sind. Darauf aufbauend läßt sich als weiters Projekt ein realer PROLOG-Compiler implementieren und verifizieren.

Über die WAM-Fallstudie hinaus ist diese Arbeit eine exemplarische Untersuchung bezüglich dynamischer Algebren. Unabhängig von den verwendeten Logikformalismen und dem Beweissystem kommt man bei dem Nachweis von Eigenschaften regelbasierter Systeme nicht um die Behandlung von Invarianzen herum. Wir hoffen, daß die vorgestellten Methoden in diesem Sinne als ein Beitrag zur formalen Beherrschung von dynamischen Algebren verstanden werden können.

Anhang A

Signaturindex

Anhang B

Programme für dynamische Algebren

```
D-EVAL#(db', procdef', goal; var subst)
begin
(: initialisation :)
var s = @s +s bottom,
    vi = 0,
    rmode = call',
    stop = run,
    breg = bottom,
    ctreg = bottom,
    ssub = cssub(@su), (: the empty subst for every node :)
    subreg = @su,
    sdecglseq = csdecglseq(sdnil),
    decglseqreg = mksdecgoal(goal,bottom) +sdl sdnil,
    scll = cscll(failcode),
    cllreg =  failcode,
    b = cb(bottom)
in (: main program :)
   begin
   while stop = run do
         D-BODY#(db',procdef'; s,vi,rmode,stop,breg,ctreg,ssub,subreg,
                  sdecglseq,decglseqreg,scll,cllreg,b);
   if stop = failure then subst := fail else subst := oksubst(subreg)
   end
end;

D-BODY#(db',procdef'; var s,vi,rmode,stop,breg,ctreg,
                          ssub,subreg,sdecglseq,decglseqreg,scll,cllreg,b)
begin
if decglseqreg = sdnil
then D-QUERY-SUCCESS#(; stop)
```

```
else var goal = (sdcar(decglseqreg)).s1
     in if goal = gnil then D-GOAL-SUCCESS#(; decglseqreg)
        else var act = gcar(goal),
                  scutpt = (sdcar(decglseqreg)).s2
             in var scont = mksdecgoal(gcdr(goal),scutpt) +sdl sdcdr(decglseqreg)
                in if rmode = call'
                   then if is_user_defined(act)
                        then D-CALL#(act,procdef',breg,db'; cllreg,rmode,ctreg,stop)
                        else if act = true'
                        then D-TRUE#(scont; decglseqreg)
                        else if act = fail'
                        then D-FAIL#(breg; stop, rmode)
                        else (: act = ! :)
                             D-CUT#(scont,scutpt; breg, decglseqreg)
                   else if rmode = try
                        then D-TRY#(cllreg,subreg,decglseqreg,db';
                                        scll,ssub,breg,b,sdecglseq,s,rmode)
                        else if rmode = enter
                        then D-ENTER#(db',act,scont,cllreg,breg,ctreg;
                                      subreg,decglseqreg,vi,stop,rmode)
                        else (: rmode = retry :)
                             D-RETRY#(db',sdecglseq,b,ssub;
                                      cllreg,subreg,scll,breg,ctreg,decglseqreg,rmode)
end;

D-QUERY-SUCCESS#(var stop)
begin stop := success end;

D-GOAL-SUCCESS#(var decglseqreg)
begin decglseqreg := sdcdr(decglseqreg) end;

D-CALL#(act, procdef', breg, db'; var cllreg,rmode,ctreg,stop)
begin
if clause'(procdef' ^pd id(act), db') = null
then D-BACKTRACK#(breg; stop, rmode)
else begin cllreg := procdef' ^pd id(act);
           rmode := try;
           ctreg := breg
     end
end;

D-TRY#(cllreg,subreg,decglseqreg,db'; var scll,ssub,breg,b,sdecglseq,s,rmode)
begin
rmode := enter;
if clause'(next(cllreg), db') # null
then var temp = snew(s)
     in begin
```

```
        s := s +s temp;
        b := b +b temp /b breg;
        sdecglseq := sdecglseq +sd temp /sd decglseqreg;
        ssub := ssub +su temp /su subreg;
        scll := scll +sc temp /sc next(cllreg);
        breg := temp
        end
end;

D-ENTER#(db',act,scont,cllreg,breg,ctreg; var subreg,decglseqreg,vi,stop,rmode)
begin
var cla = ren(the_clau(clause'(cllreg, db')),vi)
in var uni = unify(act,hd(cla))
   in if uni = fail
      then D-BACKTRACK#(breg; stop, rmode)
      else begin
           decglseqreg := ssubres(mksdecgoal(bdy(cla),ctreg)
                          +sdl scont,the_subst(uni));
           subreg := subreg o the_subst(uni);
           vi := vi +1;
           rmode := call'
           end
end;

D-BACKTRACK#(breg; var stop, rmode)
begin
if breg = bottom
then stop := failure
else rmode := retry
end;

D-RETRY#(db',sdecglseq,b,ssub;
         var cllreg,subreg,scll,breg,ctreg,decglseqreg,rmode)
begin
decglseqreg := sdecglseq ^sd breg;
subreg := ssub ^su breg;
cllreg := scll ^sc breg;
(: scll := scll +sc breg /sc next(scll ^sc breg);  wird in Abweichung von Boerger
   in den else-Fall geschoben, da sonst auf einem fast toten Knoten gearbeitet wird. :)
ctreg := b ^b breg;
rmode := enter;
if clause'(next(scll ^sc breg), db') = null
then breg := b ^b breg
else scll := scll +sc breg /sc next(scll ^sc breg)  (: <- das ist die Abweichung :)
end;

D-TRUE#(scont; var decglseqreg)
```

```
begin
decglseqreg := scont
end;

D-FAIL#(breg; var stop, rmode)
begin
D-BACKTRACK#(breg; stop, rmode)
end;

D-CUT#(scont,scutpt; var breg, decglseqreg)
begin
breg := scutpt;
decglseqreg := scont
end;

(:  ############ Ende der EA "Determinacy Detection"
    ########################################  :)
(:  ############ jetzt Zwischenebene: wie oben,
                 nur mit Regelabbildung 1:1 zur CP-EA ##########  :)

NEW-D-EVAL#(db', procdef', goal; var subst)
begin
(: initialisation :)
var s = @s +s bottom,
    vi = 0,
    rmode = call',
    stop = run,
    breg = bottom,
    ctreg = bottom,
    ssub = cssub(@su), (: the empty subst for every node :)
    subreg = @su,
    sdecglseq = csdecglseq(sdnil),
    decglseqreg = mksdecgoal(goal,bottom) +sdl sdnil,
    scll = cscll(failcode),
    cllreg =  failcode,
    b = cb(bottom)
in (: main program :)
   begin
   while stop = run do
         NEW-D-BODY#(db',procdef';
                     s,vi,rmode,stop,breg,ctreg,ssub,subreg,
                     sdecglseq,decglseqreg,scll,cllreg,b);
   if stop = failure then subst := fail else subst := oksubst(subreg)
   end
end;
```

```
NEW-D-CALL#(act, procdef', breg, db'; var cllreg,rmode,ctreg,stop)
begin
if clause'(procdef' ^pd id(act), db') = null
then D-BACKTRACK#(breg; stop, rmode)
else begin cllreg := procdef' ^pd id(act);
           ctreg := breg;
           if clause'(next(cllreg), db') = null
           then rmode := enter
           else rmode := try
     end
end;

NEW-D-TRY#(cllreg,subreg,decglseqreg,db';
           var scll,ssub,breg,b,sdecglseq,s,rmode)
begin
rmode := enter;
var temp = snew(s)
in begin
   s := s +s temp;
   b := b +b temp /b breg;
   sdecglseq := sdecglseq +sd temp /sd decglseqreg;
   ssub := ssub +su temp /su subreg;
   scll := scll +sc temp /sc next(cllreg);
   breg := temp
   end
end;

NEW-D-RETRY#(sdecglseq,b,ssub,breg;
             var cllreg,subreg,scll,ctreg,decglseqreg,rmode)
(: entspricht D-RETRY-2# (it else skip) :)
begin
decglseqreg := sdecglseq ^sd breg;
subreg := ssub ^su breg;
cllreg := scll ^sc breg;
scll := scll +sc breg /sc next(scll ^sc breg);
ctreg := b ^b breg;
rmode := enter
end;

NEW-D-TRUST#(sdecglseq,b,ssub,scll;
             var cllreg,subreg,breg,ctreg,decglseqreg,rmode)
(: entspricht D-RETRY-1# (then) :)
begin
decglseqreg := sdecglseq ^sd breg;
subreg := ssub ^su breg;
cllreg := scll ^sc breg;
ctreg := b ^b breg;
```

```
rmode := enter;
breg := b ^b breg
end;


NEW-D-BODY#(db',procdef'; var s,vi,rmode,stop,breg,ctreg,
                          ssub,subreg,sdecglseq,decglseqreg,scll,cllreg,b)
begin
if decglseqreg = sdnil
then D-QUERY-SUCCESS#(; stop)
else var goal = (sdcar(decglseqreg)).s1
     in if goal = gnil then D-GOAL-SUCCESS#(; decglseqreg)
        else var act = gcar(goal),
                  scutpt = (sdcar(decglseqreg)).s2
             in var scont = mksdecgoal(gcdr(goal),scutpt)
                            +sdl sdcdr(decglseqreg)
                in if rmode = call'
                   then if is_user_defined(act)
                        then NEW-D-CALL#(act,procdef',breg,db';
                                         cllreg,rmode,ctreg,stop)
                        else if act = true'
                        then D-TRUE#(scont; decglseqreg)
                        else if act = fail'
                        then D-FAIL#(breg; stop, rmode)
                        else (: act = ! :)
                             D-CUT#(scont,scutpt; breg, decglseqreg)
                   else if rmode = try
                        then NEW-D-TRY#(cllreg,subreg,decglseqreg,db';
                                        scll,ssub,breg,b,sdecglseq,s,rmode)
                        else if rmode = enter
                        then D-ENTER#(db',act,scont,cllreg,breg,ctreg;
                                      subreg,decglseqreg,vi,stop,rmode)
                        else (: rmode = retry :)
                             if not clause'(next(scll ^sc breg), db') = null
                             then NEW-D-RETRY#(sdecglseq,b,ssub,breg;
                                               cllreg,subreg,scll,
                                               ctreg,decglseqreg,rmode)
                             else NEW-D-TRUST#(sdecglseq,b,ssub,scll;
                                               cllreg,subreg,breg,
                                               ctreg,decglseqreg,rmode)
end;
```

```
CP-EVAL#(db'', procdef'', goal; var subst)
begin
(: initialisation :)
var s' = @s +s bottom,
    vi' = 0,
    stop' = run,
    preg =  start,
    breg' = bottom,
    ctreg' = bottom,
    ssub' = cssub(@su), (: the empty subst for every state :)
    subreg' = @su,
    sdecglseq' = csdecglseq(sdnil),
    decglseqreg' = mksdecgoal(goal,bottom) +sdl sdnil,
    p = cp(failcode),
    b' = cb(bottom)
in (: main program :)
   begin
   while stop' = run do
         CP-BODY#(db'',procdef''; s',vi',stop',breg',ctreg',ssub',subreg',
                  sdecglseq',decglseqreg',p,preg,b');
   if stop' = failure then subst := fail else subst := oksubst(subreg')
   end
end;

CP-BODY#(db'',procdef''; var s',vi',stop',breg',ctreg',
                         ssub',subreg',sdecglseq',decglseqreg',p,preg,b')
begin
if decglseqreg' = sdnil
then CP-QUERY-SUCCESS#(; stop')
else var goal = (sdcar(decglseqreg')).s1
     in if goal = gnil then CP-GOAL-SUCCESS#(; decglseqreg')
        else var act = gcar(goal),
                 scutpt = (sdcar(decglseqreg')).s2
             in var scont = mksdecgoal(gcdr(goal),scutpt)
                            +sdl sdcdr(decglseqreg')
                in if preg = start
                   then if is_user_defined(act)
                        then CP-CALL#(act, procdef'', breg', p, db'';
                                      preg,ctreg',stop')
                        else if act = true'
                        then CP-TRUE#(scont; decglseqreg')
                        else if act = fail'
                        then CP-FAIL#(breg',p; stop', preg)
                        else (: act = ! :)
                             CP-CUT#(scont,scutpt; breg', decglseqreg')
```

```
                    else if is_try_me(code(preg, db''))
                         then CP-TRY-ME#(subreg',decglseqreg',db'';
                                        p,ssub',breg',b',sdecglseq',s',preg)
                         else if is_clause(code(preg, db''))
                         then CP-ENTER#(db'',act,scont,breg',ctreg',p;
                                       subreg',decglseqreg',vi',stop',preg)
                         else if is_retry_me(code(preg, db''))
                         then CP-RETRY-ME#(db'',sdecglseq',b',ssub',breg';
                                           preg,subreg',p,ctreg',decglseqreg')
                         else (: is_trust_me(code(preg, db'')) :)
                              CP-TRUST-ME#(sdecglseq',b',ssub';
                                           preg,subreg',breg',ctreg',decglseqreg')
end;

CP-QUERY-SUCCESS#(var stop')
begin stop' := success end;

CP-GOAL-SUCCESS#(var decglseqreg')
begin decglseqreg' := sdcdr(decglseqreg') end;

CP-CALL#(act, procdef'', breg', p, db''; var preg,ctreg',stop')
begin
if code(procdef'' ^pd id(act), db'') = nil'
then CP-BACKTRACK#(breg', p; stop', preg)
else begin preg := procdef'' ^pd id(act);
           ctreg' := breg'
     end
end;


CP-TRY-ME#(subreg',decglseqreg',db''; var p,ssub',breg',b',sdecglseq',s',preg)
begin
var temp = snew(s')
in begin
   s' := s' +s temp;
   b' := b' +b temp /b breg';
   sdecglseq' := sdecglseq' +sd temp /sd decglseqreg';
   ssub' := ssub' +su temp /su subreg';
   p := p +p temp /p where(code(preg, db''));
   breg' := temp;
   preg := next(preg)
   end
end;

CP-ENTER#(db'',act,scont,breg',ctreg',p; var subreg',decglseqreg',vi',stop',preg)
begin
var cla = ren(the_cl(code(preg, db'')),vi')
```

```
in var uni = unify(act,hd(cla))
   in if uni = fail
      then CP-BACKTRACK#(breg', p; stop', preg)
      else begin
           decglseqreg' := ssubres(mksdecgoal(bdy(cla),ctreg')
                           +sdl scont,the_subst(uni));
           subreg' := subreg' o the_subst(uni);
           vi' := vi' +1;
           preg := start
           end
end;

CP-BACKTRACK#(breg', p; var stop', preg)
begin
if breg' = bottom
then stop' := failure
else preg := p ^p breg'
end;

CP-RETRY-ME#(db'',sdecglseq',b',ssub',breg';
             var preg,subreg',p,ctreg',decglseqreg')
begin
decglseqreg' := sdecglseq' ^sd breg';
subreg' := ssub' ^su breg';
ctreg' := b' ^b breg';
p := p +p breg' /p where(code(preg, db''));
preg := next(preg)
end;

CP-TRUST-ME#(sdecglseq',b',ssub'; var preg,subreg',breg',ctreg',decglseqreg')
begin
decglseqreg' := sdecglseq' ^sd breg';
subreg' := ssub' ^su breg';
ctreg' := b' ^b breg';
breg' := b' ^b breg';
preg := next(preg)
end;

CP-TRUE#(scont; var decglseqreg')
begin
decglseqreg' := scont
end;

CP-FAIL#(breg', p; var stop', preg)
begin
CP-BACKTRACK#(breg', p; stop', preg)
end;
```

```
CP-CUT#(scont,scutpt; var breg', decglseqreg')
begin
breg' := scutpt;
decglseqreg' := scont
end;
```

Anhang C

clls# und chain#

```
CLLS#(coa, db'; var cal)
begin
if clause'(coa, db') = null
then cal := canil
else begin CLLS#(next(coa), db'; cal);
           cal := coa +cal cal
     end
end;


(################################################)


CHAIN#(coa, db''; var cal)
begin
var instr  = code(coa, db'')
in if is_try_me(instr)
   then CHAIN-TRY-ME#(coa, db''; cal)
   else if is_clause(instr)
        then cal := coa +cal canil
        else if instr = nil'
             then cal := canil
             else abort
end;

CHAIN-TRY-ME#(coa, db''; var cal)
begin
var instr  = code(coa, db''),
    follow = code(next(coa), db'')
in if is_try_me(instr)
   then if is_clause(follow)
        then begin
```

```
            CHAIN-RETRY-ME#(where(instr), db''; cal);
            cal := next(coa) +cal cal
            end
       else abort
  else abort
end;

CHAIN-RETRY-ME#(coa, db''; var cal)
begin
var instr  = code(coa, db''),
    follow = code(next(coa), db'')
in if is_retry_me(instr)
  then if is_clause(follow)
       then begin
            CHAIN-RETRY-ME#(where(instr), db''; cal);
            cal := next(coa) +cal cal
            end
       else abort
  else if is_trust_me(instr)
       then if is_clause(follow)
            then cal := next(coa) +cal canil
            else abort
       else abort
end;
```

Anhang D

Die Invariante

$$
\begin{array}{l}
\exists\ f.\quad stop = stop' \\
\quad \wedge\ vi = vi' \\
\quad \wedge\ subreg = subreg' \\
\quad \wedge\ f\ \hat{}_f\ breg = breg' \\
\quad \wedge\ f\ \hat{}_f\ ctreg = ctreg' \\
\quad \wedge\ f\ \hat{}_f\ bottom = bottom \\
\quad \wedge\ fd(f, decglseqreg) = decglseqreg' \\
\quad \wedge\ breg\ ins\ s \\
\quad \wedge\ ctreg\ ins\ s \\
\quad \wedge\ bottom\ ins\ s \\
\quad \wedge\ decglseqreg\ ctpelem\ s \\
\quad \wedge\ (rmode = call' \rightarrow preg = start) \\
\quad \wedge\ (rmode = retry \rightarrow breg \neq bottom \wedge preg = p\ \hat{}_p\ breg') \\
\quad \wedge\ (rmode = enter \rightarrow code(preg, db") = mkcl(the\_clau(clause'(cllreg, db')))) \\
\quad \wedge\ (\quad rmode = try \\
\qquad \rightarrow \quad is\_user\_defined(gcar(sdcar(decglseqreg).s1)) \\
\qquad\quad \wedge\ \langle clls\#(cllreg, db'; cal_1)\rangle \\
\qquad\qquad\quad \langle chain\text{-}try\text{-}me\#(preg, db"; cal_2)\rangle \\
\qquad\qquad\qquad mapcode(cal_2, db") = mapclause'(cal_1, db')) \\
\quad \wedge\ (\quad decglseqreg = sdnil \\
\qquad\quad \vee\ sdcar(decglseqreg).s1 = gnil \\
\qquad\quad \vee\ gcar(sdcar(decglseqreg).s1) = ! \\
\qquad\quad \vee\ gcar(sdcar(decglseqreg).s1) = true' \\
\qquad \rightarrow rmode = call') \\
\quad \wedge\ (\forall\ st.\quad st\ ins\ s \\
\qquad\quad \rightarrow \quad (b\ \hat{}_b\ st)\ ins\ s \\
\qquad\qquad \wedge\ (sdecglseq\ \hat{}_{sd}\ st)\ ctpelem\ s \\
\qquad\qquad \wedge\ (f\ \hat{}_f\ st)\ ins\ s' \\
\qquad\qquad \wedge\ ssub\ \hat{}_{su}\ st = ssub'\ \hat{}_{su}\ f\ \hat{}_f\ st \\
\qquad\qquad \wedge\ f\ \hat{}_f\ b\ \hat{}_b\ st = b'\ \hat{}_b\ f\ \hat{}_f\ st \\
\qquad\qquad \wedge\ fd(f, sdecglseq\ \hat{}_{sd}\ st) = sdecglseq'\ \hat{}_{sd}\ f\ \hat{}_f\ st \\
\qquad\qquad \wedge\ (\forall\ st_1.st_1\ ins\ s \rightarrow (f\ \hat{}_f\ st = f\ \hat{}_f\ st_1 \rightarrow st = st_1)) \\
\qquad\qquad \wedge\ (\quad st \neq bottom
\end{array}
$$

$$
\begin{array}{l}
\quad \rightarrow \quad \text{sdecglseq } \hat{}_{sd} \text{ st} \neq \text{sdnil} \\
\qquad \wedge \text{ sdcar(sdecglseq } \hat{}_{sd} \text{ st).s1} \neq \text{gnil} \\
\qquad \wedge \text{ is\_user\_defined(gcar(sdcar(sdecglseq } \hat{}_{sd} \text{ st).s1)))} \\
\wedge \; (\quad \text{st} \neq \text{bottom} \\
\quad \rightarrow \langle \text{clls\#(scll } \hat{}_{sc} \text{ st, db'; cal}_1) \rangle \\
\qquad\qquad \langle \text{chain-retry-me\#(p } \hat{}_{p} \text{ f } \hat{}_{f} \text{ st, db"; cal}_2) \rangle \\
\qquad\qquad\qquad \text{mapcode(cal}_2\text{, db")} = \text{mapclause'(cal}_1\text{, db')))}
\end{array}
$$

Anhang E

Lemmata

call-cp-newd

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$,
$\text{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
is_user_defined(gcar(sdcar(decglseqreg).s1)),
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \text{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (call-newd-cp)

- benutzt von : (cp-newd-step)
- Interaktionen : 6
- Beweisschritte : 28

call-newd-cp

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$,
$\text{INV}(\overline{x}, \overline{y})$,
$\text{stop} = \text{run}$,
$\text{decglseqreg} \neq \text{sdnil}$,
$\text{sdcar}(\text{decglseqreg}).\text{s1} \neq \text{gnil}$,
$\text{rmode} = \text{call'}$,
$\text{is\_user\_defined}(\text{gcar}(\text{sdcar}(\text{decglseqreg}).\text{s1}))$,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \text{INV}(\overline{x}_0, \overline{y})$

- benutzte Lemmata : (compass2+weak1)
- benutzt von : (call-cp-newd newd-cp-step)
- Interaktionen : 35
- Beweisschritte : 212

comp-assum-2

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$
$\vdash$
$[\text{clls\#(procdef' }\hat{}_{pd}\text{ id(lit), db'; cal}_1)]\langle\text{chain\#(procdef" }\hat{}_{pd}\text{ id(lit), db"; cal}_2)\rangle\text{mapcode}(\text{cal}_2, \text{db"}) = \text{mapclause'}(\text{cal}_1, \text{db'})$

- benutzt von : (compass2+weak1)

compass2+weak1

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$
$\vdash$
$\langle\text{clls\#(procdef' }\hat{}_{pd}\text{ id(lit), db'; cal}_1)\rangle$
$\langle\text{chain\#(procdef" }\hat{}_{pd}\text{ id(lit), db"; cal}_2)\rangle$
$\text{mapcode}(\text{cal}_2, \text{db"}) = \text{mapclause'}(\text{cal}_1, \text{db'})$

- benutzte Lemmata : (comp-assum-2 weak-comp-assum-1)
- benutzt von : (call-newd-cp)
- Interaktionen : 2
- Beweisschritte : 5

cp-newd-imp

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"})$
$\vdash$
$\langle \text{cp-eval\#(db", procdef", goal; subst)} \rangle \text{subst} = \text{subst}_0$
$\rightarrow \langle \text{new-d-eval\#(db', procdef', goal; subst)} \rangle \text{subst} = \text{subst}_0$

- benutzte Lemmata : (cp-newd-ind)
- benutzt von : (newd-cp-equiv)
- Interaktionen : 19
- Beweisschritte : 82

cp-newd-ind

$\text{compile}_2(\text{pair}_1(\text{db'}, \text{procdef'})) = \text{pair}_2(\text{db"}, \text{procdef"}),$
$\text{INV}(\overline{x}, \overline{y}),$
$\langle$**loop**
  if stop' = run **then**
    cp-body#(db", procdef"; $\overline{y}$)
  else
    skip
times kappa$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\exists$ kappa.$\langle$**loop**
  if stop = run **then**
    new-d-body#(db', procdef'; $\overline{x}$)
  else
    skip
times kappa$\rangle$
  $\text{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (cp-newd-step)
- benutzt von : (cp-newd-imp)
- Interaktionen : 31
- Beweisschritte : 76

cp-newd-step

$\mathrm{compile}_2(\mathrm{pair}_1(\text{db'}, \text{procdef'})) = \mathrm{pair}_2(\text{db"}, \text{procdef"})$,
$\mathrm{INV}(\overline{x}, \overline{y})$,
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \mathrm{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (trust-cp-newd retry-cp-newd enter-cp-newd try-cp-newd cut-cp-newd fail-cp-newd true-cp-newd call-cp-newd g-succ-cp-newd q-succ-cp-newd)
- benutzt von : (cp-newd-ind)
- Interaktionen : 39
- Beweisschritte : 64

cut-cp-newd

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
gcar(sdcar(decglseqreg).s1) = !,
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \mathrm{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (cut-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 5
- Beweisschritte : 21

cut-newd-cp

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
gcar(sdcar(decglseqreg).s1) = !,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \mathrm{INV}(\overline{x}_0, \overline{y})$

- keine Lemmata benutzt.
- benutzt von : (cut-cp-newd newd-cp-step)
- Interaktionen : 5
- Beweisschritte : 37

d-cp-equiv

$\mathrm{compile}_2(\mathrm{pair}_1(\text{db', procdef'})) = \mathrm{pair}_2(\text{db", procdef"})$
$\vdash$
$\langle$d-eval#(db', procdef', goal; subst)$\rangle$subst = subst_0
$\leftrightarrow \langle$cp-eval#(db", procdef", goal; subst)$\rangle$subst = subst_0

- benutzte Lemmata : (newd-cp-equiv d-newd-equiv)
- Interaktionen : 2
- Beweisschritte : 3

d-newd-equiv

$\vdash$
$\langle$d-eval#(db', procdef', goal; subst)$\rangle$subst = subst_0
$\leftrightarrow \langle$new-d-eval#(db', procdef', goal; subst)$\rangle$subst = subst_0

- benutzte Lemmata : (d-newd-imp newd-d-imp)
- benutzt von : (d-cp-equiv)
- Interaktionen : 2

- Beweisschritte : 3

d-newd-imp

$\langle$d-eval#(db', procdef', goal; subst)$\rangle$subst = subst$_0$
$\vdash$ $\langle$new-d-eval#(db', procdef', goal; subst)$\rangle$subst = subst$_0$

- benutzte Lemmata : (d-newd-ind)
- benutzt von : (d-newd-equiv)
- Interaktionen : 12
- Beweisschritte : 62

d-newd-ind

$\langle$**loop**
  if stop = run **then**
    d-body#(db', procdef'; $\overline{x}$)
  else
    skip
times kappa$\rangle$ $\overline{x} = \overline{x}_0$
$\vdash$
$\exists$ kappa.$\langle$**loop**
  if stop = run **then**
    new-d-body#(db', procdef'; $\overline{x}$)
  else
    skip
times kappa$\rangle$
  $\overline{x} = \overline{x}_0$

- benutzt von : (d-newd-imp)

enter-cp-newd

INV($\overline{x}, \overline{y}$),
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = enter,
$\langle$**if** stop' = run **then**
  cp-body#(db", procdef"; $\overline{y}$)
else
  skip$\rangle$ $\overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
  new-d-body#(db', procdef'; $\overline{x}$)
else
  skip$\rangle$ INV($\overline{x}, \overline{y}_0$)

- benutzte Lemmata : (enter-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 6
- Beweisschritte : 30

enter-newd-cp

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = enter,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \mathrm{INV}(\overline{x}_0, \overline{y})$

- keine Lemmata benutzt.
- benutzt von : (enter-cp-newd newd-cp-step)
- Interaktionen : 18
- Beweisschritte : 114

fail-cp-newd

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
gcar(sdcar(decglseqreg).s1) = fail',
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \mathrm{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (fail-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 6
- Beweisschritte : 23

fail-newd-cp

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
gcar(sdcar(decglseqreg).s1) = fail',
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \mathrm{INV}(\overline{x}_0, \overline{y})$

- keine Lemmata benutzt.
- benutzt von : (fail-cp-newd newd-cp-step)
- Interaktionen : 16
- Beweisschritte : 64

g-succ-cp-newd

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 = gnil,
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \mathrm{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (g-succ-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 4
- Beweisschritte : 13

g-succ-newd-cp

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 = gnil,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \mathrm{INV}(\overline{x}_0, \overline{y})$

- keine Lemmata benutzt.
- benutzt von : (g-succ-cp-newd newd-cp-step)
- Interaktionen : 5
- Beweisschritte : 23

newd-cp-equiv

$\mathrm{compile}_2(\mathrm{pair}_1(\text{db', procdef'})) = \mathrm{pair}_2(\text{db", procdef"})$
$\vdash$
$\langle$new-d-eval#(db', procdef', goal; subst)$\rangle$subst = subst_0
$\leftrightarrow \langle$cp-eval#(db", procdef", goal; subst)$\rangle$subst = subst_0

-
- benutzte Lemmata : (cp-newd-imp newd-cp-imp)
- benutzt von : (d-cp-equiv)
- Interaktionen : 4
- Beweisschritte : 7

newd-cp-imp

$\mathrm{compile}_2(\mathrm{pair}_1(\mathrm{db'}, \mathrm{procdef'})) = \mathrm{pair}_2(\mathrm{db''}, \mathrm{procdef''})$
$\vdash$
$\langle \text{new-d-eval\#}(\mathrm{db'}, \mathrm{procdef'}, \mathrm{goal}; \mathrm{subst})\rangle \mathrm{subst} = \mathrm{subst}_0$
$\rightarrow \langle \text{cp-eval\#}(\mathrm{db''}, \mathrm{procdef''}, \mathrm{goal}; \mathrm{subst})\rangle \mathrm{subst} = \mathrm{subst}_0$

-
- benutzte Lemmata : (newd-cp-ind)
- benutzt von : (newd-cp-equiv)
- Interaktionen : 18
- Beweisschritte : 80

newd-cp-ind

$\mathrm{compile}_2(\mathrm{pair}_1(\mathrm{db'}, \mathrm{procdef'})) = \mathrm{pair}_2(\mathrm{db''}, \mathrm{procdef''}),$
$\mathrm{INV}(\overline{x}, \overline{y}),$
$\langle$**loop**
if stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip
times kappa$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\exists$ kappa.$\langle$**loop**
if stop' = run **then**
cp-body#(db'', procdef''; $\overline{y}$)
else
skip
times kappa$\rangle$
$\mathrm{INV}(\overline{x}_0, \overline{y})$

- benutzte Lemmata : (newd-cp-step)
- benutzt von : (newd-cp-imp)
- Interaktionen : 28
- Beweisschritte : 72

newd-cp-step

$\mathrm{compile}_2(\mathrm{pair}_1(\mathrm{db'}, \mathrm{procdef'})) = \mathrm{pair}_2(\mathrm{db''}, \mathrm{procdef''}),$
$\mathrm{INV}(\overline{x}, \overline{y}),$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)

```
   else
     skip⟩x̄ = x̄₀
⊢
⟨if stop' = run then
     cp-body#(db", procdef"; ȳ)
   else
     skip⟩INV(x̄₀, ȳ)
```

- benutzte Lemmata : (trust-newd-cp retry-newd-cp enter-newd-cp try-newd-cp cut-newd-cp fail-newd-cp true-newd-cp call-newd-cp g-succ-newd-cp q-succ-newd-cp)
- benutzt von : (newd-cp-ind)
- Interaktionen : 36
- Beweisschritte : 53

newd-d-imp

```
   ⟨new-d-eval#(db', procdef', goal; subst)⟩subst = subst₀
⊢ ⟨d-eval#(db', procdef', goal; subst)⟩subst = subst₀
```

- benutzt von : (d-newd-equiv)

newd-d-ind

```
   ⟨loop
       if stop = run then
         new-d-body#(db', procdef'; x̄)
       else
         skip
    times kappa⟩x̄ = x̄₀
⊢
∃ kappa.⟨loop
                if stop = run then
                  d-body#(db', procdef'; x̄)
                else
                  skip
           times kappa⟩
              x̄ = x̄₀)
```

- benutzt von : (newd-d-imp)

q-succ-cp-newd

```
  INV(x̄, ȳ),
  stop = run,
  decglseqreg = sdnil,
```

$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle$ $\overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle$ INV($\overline{x}, \overline{y}_0$)

- benutzte Lemmata : (q-succ-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 4
- Beweisschritte : 11

q-succ-newd-cp

INV($\overline{x}, \overline{y}$),
stop = run,
decglseqreg = sdnil,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle$ $\overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle$ INV($\overline{x}_0, \overline{y}$)

- keine Lemmata benutzt.
- benutzt von : (newd-cp-step q-succ-cp-newd)
- Interaktionen : 5
- Beweisschritte : 19

retry-cp-newd

INV($\overline{x}, \overline{y}$),
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = retry,

clause'(next(scll $\hat{}_{sc}$ breg), db') $\neq$ null,
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle$ $\overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle$ INV($\overline{x}, \overline{y}_0$)

- benutzte Lemmata : (retry-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 5
- Beweisschritte : 25

retry-newd-cp

INV($\overline{x}, \overline{y}$),
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = retry,
clause'(next(scll $\hat{}_{sc}$ breg), db') $\neq$ null,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle$ $\overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle$ INV($\overline{x}_0, \overline{y}$)

- keine Lemmata benutzt.
- benutzt von : (newd-cp-step retry-cp-newd)
- Interaktionen : 45
- Beweisschritte : 236

true-cp-newd

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
gcar(sdcar(decglseqreg).s1) = true',
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; s', $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \mathrm{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (true-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 5
- Beweisschritte : 19

true-newd-cp

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = call',
gcar(sdcar(decglseqreg).s1) = true',
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \mathrm{INV}(\overline{x}_0, \overline{y})$

- keine Lemmata benutzt.
- benutzt von : (newd-cp-step true-cp-newd)

- Interaktionen : 5
- Beweisschritte : 33

trust-cp-newd

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = retry,
clause'(next(scll $\hat{}_{sc}$ breg), db') = null,
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \mathrm{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (trust-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 5
- Beweisschritte : 25

trust-newd-cp

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = retry,
clause'(next(scll $\hat{}_{sc}$ breg), db') = null,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \mathrm{INV}(\overline{x}_0, \overline{y})$

- keine Lemmata benutzt.
- benutzt von : (newd-cp-step trust-cp-newd)
- Interaktionen : 32
- Beweisschritte : 161

try-cp-newd

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = try,
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \overline{y} = \overline{y}_0$
$\vdash$
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \mathrm{INV}(\overline{x}, \overline{y}_0)$

- benutzte Lemmata : (try-newd-cp)
- benutzt von : (cp-newd-step)
- Interaktionen : 6
- Beweisschritte : 26

try-newd-cp

$\mathrm{INV}(\overline{x}, \overline{y})$,
stop = run,
decglseqreg $\neq$ sdnil,
sdcar(decglseqreg).s1 $\neq$ gnil,
rmode = try,
$\langle$**if** stop = run **then**
new-d-body#(db', procdef'; $\overline{x}$)
else
skip$\rangle \overline{x} = \overline{x}_0$
$\vdash$
$\langle$**if** stop' = run **then**
cp-body#(db", procdef"; $\overline{y}$)
else
skip$\rangle \mathrm{INV}(\overline{x}_0, \overline{y})$

- keine Lemmata benutzt.
- benutzt von : (newd-cp-step try-cp-newd)
- Interaktionen : 49
- Beweisschritte : 203

weak-comp-assum-1

$\vdash \langle \text{clls}\#(\text{procdef' } \hat{}_{pd}\ \text{id(lit)}, \text{db'}; \text{cal}_1)\rangle \text{true}$

- benutzt von : (compass2+weak1)

Literaturverzeichnis

[BeckertHähnle] Bernhard Beckert und Reiner Hähnle: *Proving Compiler Correctness with Evolving Algebra Specifications*. Internes Papier, Institut für Logik, Komplexität und Deduktionssysteme, Fakultät für Informatik, Universität Karlsruhe, 1994.

[Börger] Egon Börger: *Logic Programming: The Evolving Algebra Approach*. In B. Pehrson und I. Simon (Hrsg.): *IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations*. Elsevier, Amsterdam 1994, S. 391-395.

[BörgerRosenzweig] Egon Börger und Dean Rosenzweig: *The WAM - Definition and Compiler Correctness*. In C. Beierle und L. Plümer (Hrsg.): *Logic Programming: Formal Methods and Practical Applications*. Series in Computer science and Artificial Intelligence. Elsevier Science B. V./North-Holland, 1995, S. 20-90 (Kapitel 2).

[BörgerRosenzweig^a] Egon Börger und Dean Rosenzweig: *A mathematical definition of full Prolog*. In: *Science of Computer Programming*. 1994

[Reif] Wolfgang Reif: *Vollständigkeit einer modifizierten Goldblatt-Logik und Approximation der Omega-Regel durch Induktion*. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 1984.

[SchellhornAhrendt] Gerhard Schellhorn und Wolfgang Ahrendt: *From PROLOG to the WAM – first steps in compiler verification*. Arbeitsbericht, Fakultät für Informatik, Universität Karlsruhe, Universität Ulm, 1996.

[Schmitt] Peter H. Schmitt: *Proving WAM Compiler Correctness*. Interner Bericht 33/94, Institut für Logik, Komplexität und Deuktionssysteme, Fakultät für Informatik, Universität Karlsruhe, 1994.

[Warren] D. H. D. Warren: *An Abstract Prolog Instruction Set*, Tecnical Note 309, Arteficial Intelligence Center, SRI International, 1983.

[Börger], [BörgerRosenzweig] und [BörgerRosenzweig^a] sind auch erhältlich über:
ftp.di.unipi.it/pub/Papers/boerger